西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

《Verilog数字系统设计》

实验报告

学　　院：航海学院

学　　号：2020301020

姓　　名：邱梁城

专　　业：信息工程

实验地点：教学东楼B305

指导教师：严胜刚

西北工业大学

2023年7月10日

目录

实验一：译码电路

一、实验目的

（1）设计一个译码电路，实现 74LS154 的所有功能。

（2）能够对译码电路原理、时序图进行正确的分析。

（3）掌握 FPGA 开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与方法

74LS154，是在单片机系统中常用的 4 线—16 线译码器，当选通端（G1、G2）均为低电平时，可将地址端（ABCD）的二进制编码在一个对应的输出端，以低电平译出。如果将 G1 和 G2 中的一个作为数据输入端，由 ABCD 对输出寻址，74LS154 还可作 1 线-16 线数据分配器。其真值表如图 1 所示：

Function Table

Inputs						Outputs													
G1	G2	D	C	B	A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	L	L	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L
L	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H

H = HIGH Level
L = Low Level
X = Don't Care

图 1　74LS154 真值表

我们只需要按照 4-16 的译码方式进行代码编写，为了使得代码更加简介，考虑 4-16 的转换方式，由于其是以 2 进制的方式进行编码，所以每一的值是 0-2-4-8，而我们只需要对其进行移动相应的位使得转换后的那一位的值为低即可

对 74LS154 进行总结则是：

4 线 - 16 线译码器

A、B、C、D 译码地址输入端（高电平有效）

G1、G2 选通端（低电平有效）

0 - 15 输出端口（低电平有效）

输出功能表：当 G1、G2 为低时，输出由输入译码获得，低电平有效；任意为高时，输出全为高。

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、译码器实验流程框图

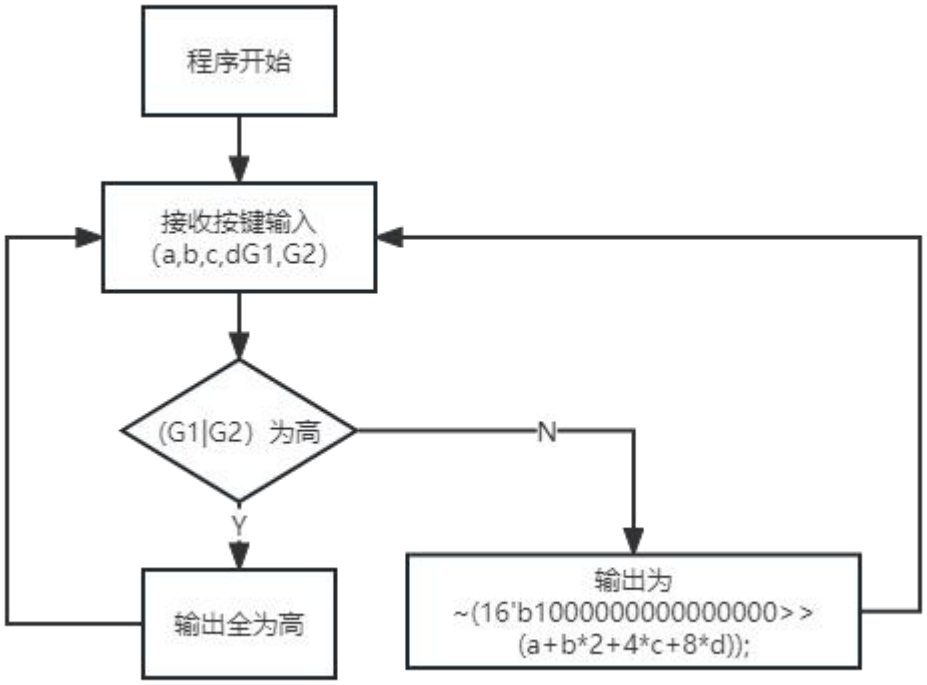


图 2 74LS154 实现流程框图

五、实验内容与结果

（1）实验内容

1、设计一个译码电路，实现 74LS154 的所有功能。

（2）实验步骤

1、建立工程

2、编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板（由于此项工程需要的按键数开发板不满足，所以只进行仿真设计）

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

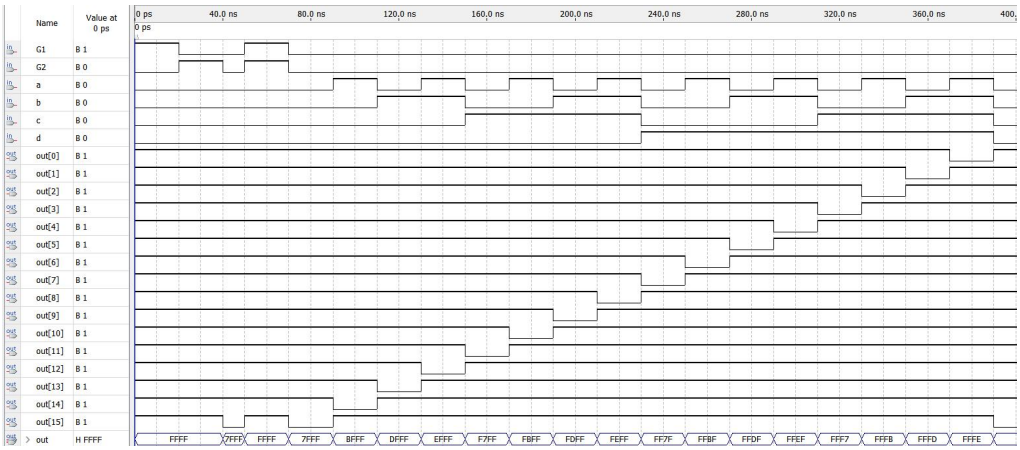


图 3 74LS154 仿真波形图

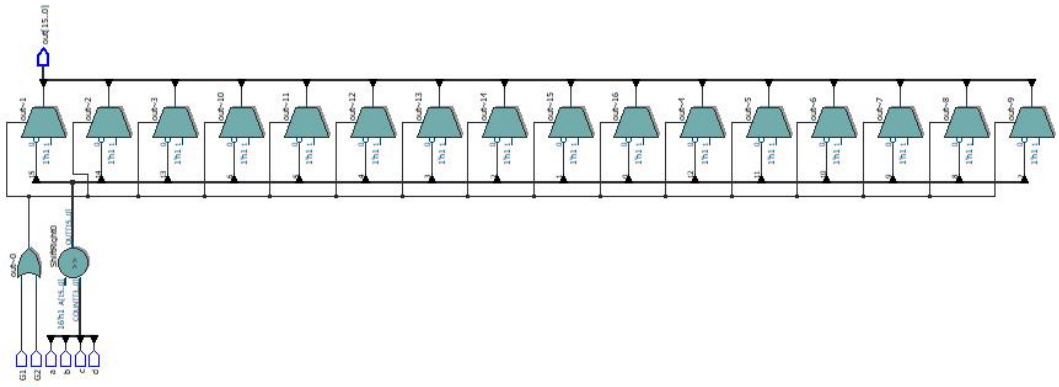


图 4 74LS154 的 RTL 视图

六、实验讨论

1、我们能够发现根据波形图去对应真值表其能够满足一一符合的条件。

2、对于 RTL 视图进行分析，我们根据其生成的电路图对输入进行进行改变，输出信号依旧能满足 74LS154 的功能，所以可以判断代码书写无误。

3、通过这次的实验，我不仅对 FPGA 开发的流程更加熟悉，并且也对译码器实现的原理有了更深的理解，加深了我对于课上知识的掌握和认识。

七、实验代码

```
module four_16
(
	input wire a,
	input wire b,
	input wire c,
	input wire d,
	input wire G1,
	input wire G2,
	output wire [15:0] out
);
assign	out=(G1|G2)?16'h0ffff:~(16'b1000000000000000>>(a+b*2+4*c+8*d));

endmodule
```

实验二：优先编码电路

一、实验目的

（1）设计一个优先编码电路，实现 74LS148 的所有功能。

（2）能够对优先编码电路原理、时序图进行正确的分析。

（3）掌握 FPGA 开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与方法

74LS148 是 8 线—3 线优先编码器，共有 54/74148 和 54/74LS148 两种线路结构型式，将 8 条数据线（0—7）进行 3 线（4-2-1）二进制（八进制）优先编码，即对最高位数据线进行译码。利用选通端（EI）和输出选通端（EO）可进行八进制扩展。

74ls148 真值表如下：

输入									输出				
$\overline{ST}$	$\overline{I}_0$	$\overline{I}_1$	$\overline{I}_2$	$\overline{I}_3$	$\overline{I}_4$	$\overline{I}_5$	$\overline{I}_6$	$\overline{I}_7$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$	$\overline{Y}_{ES}$	Y_S
1	×	×	×	×	×	×	×	×	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	×	×	×	×	×	×	×	0	0	0	0	0	1
0	×	×	×	×	×	×	0	1	0	0	1	0	1
0	×	×	×	×	×	0	1	1	0	1	0	0	1
0	×	×	×	×	0	1	1	1	0	1	1	0	1
0	×	×	×	0	1	1	1	1	1	0	0	0	1
0	×	×	0	1	1	1	1	1	1	0	1	0	1
0	×	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1

图 1 74LS148 真值表

74LS148 管脚图如下：

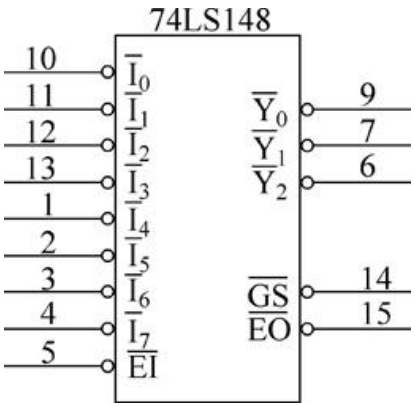


图 2 74LS148 管脚图

我们对真值表进行分析能够发现只有当 ST 为 0 或者输入全为 1 时 YES 的输出为 0，只有当 ST 为 1 并且输入全为高电平时才为 0 其他则全为 1，除此之外。便可以通过移位的方式来判断哪一位是最早出现为 1 的状态去改变输出。

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、74LS148 实验代码流程框图

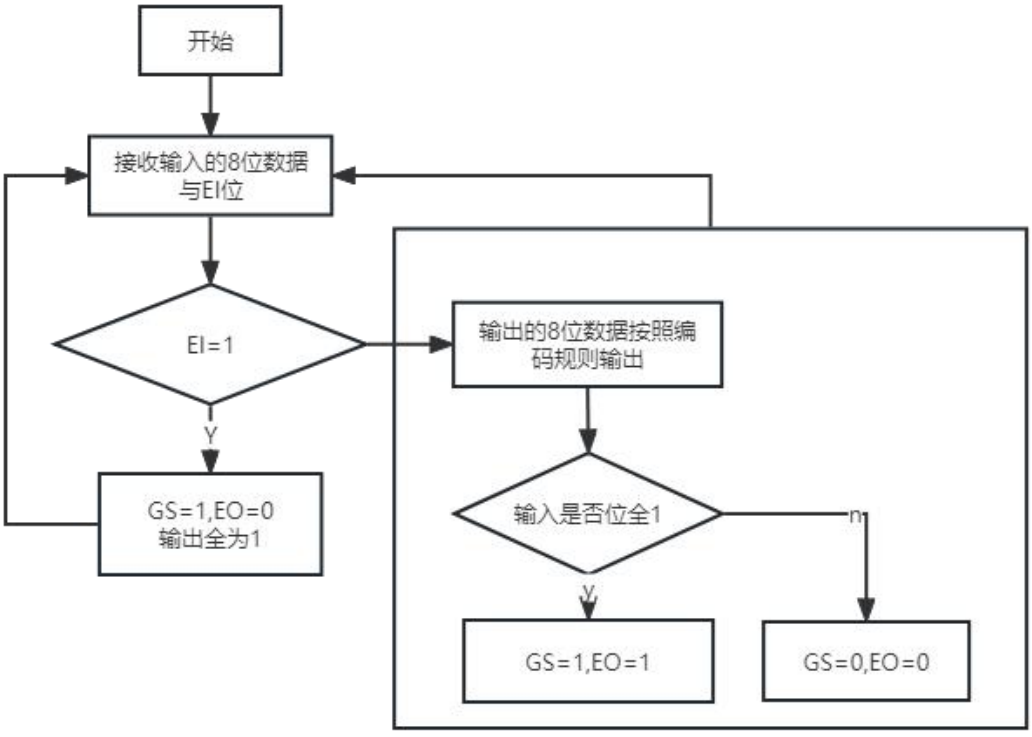


图 3 74LS148 实验代码流程框图

五、实验内容与结果

（1）实验内容

1、设计一个优先编码电路，实现 74LS148 的所有功能。

（2）实验步骤

1、建立工程

2、依照真值表编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板（由于此项工程需要的按键数开发板不满足，所以只进行仿真设计）

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

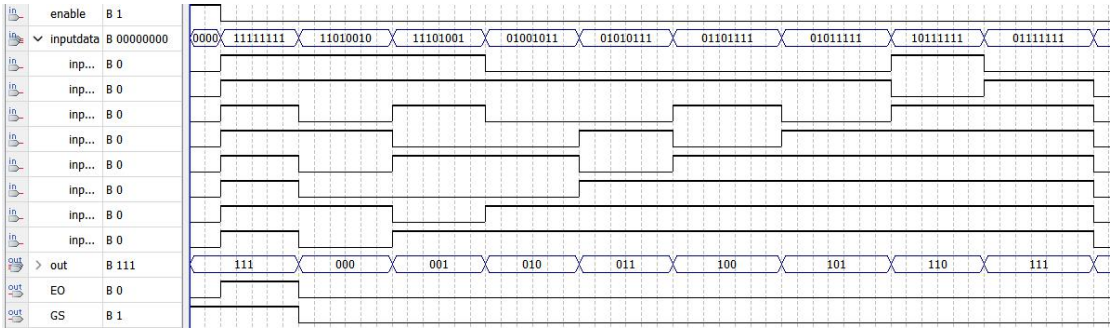


图 4 74ls148 仿真波形图

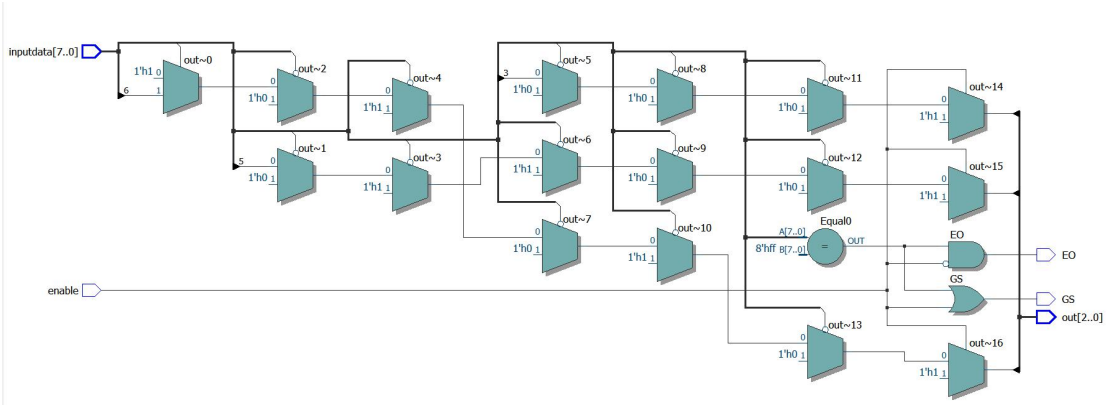


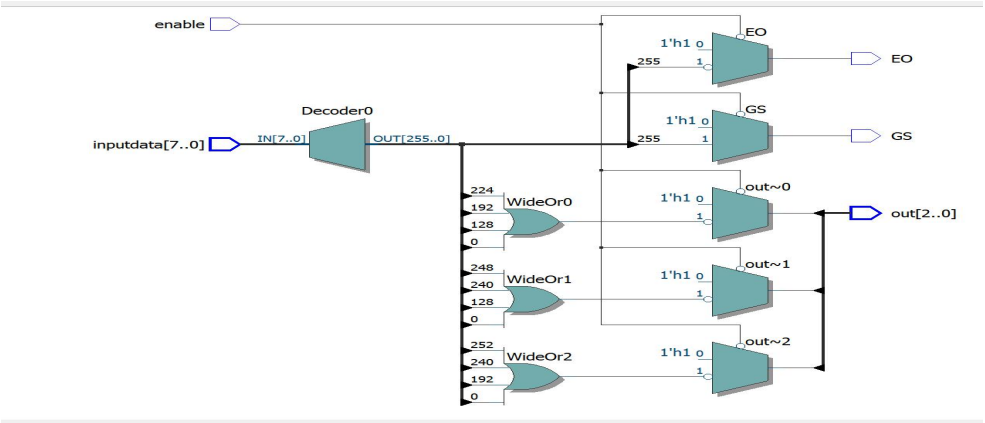
图 5 74LS148 的 RTL 视图

六、实验讨论

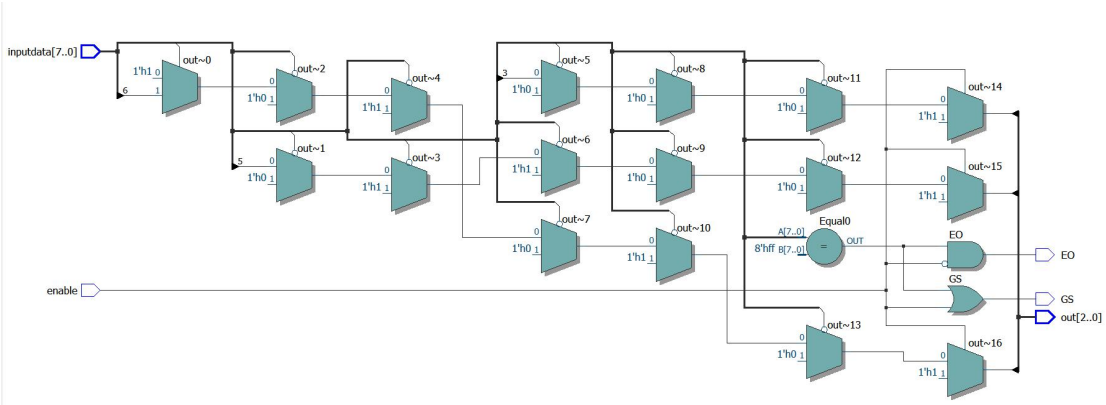
1、我们能够发现根据波形图去对应真值表其能够满足一一符合的条件。

2、对于 RTL 视图进行分析，我们根据其生成的电路图对输入进行进行改变，输出信号依旧能满足 74LS148 的功能，但是我们发现其代码可以进行多种形式的编写从而改变 RTL 电路视图，在这里给出两种不同代码模式的 RTL 视图。

①使用行为级建模



②使用逐位比较



3、通过这次的实验，我不仅对 FPGA 开发的流程更加熟悉，并且也对优先编码电路实现的原理有了更深的理解，加深了我对于课上知识的掌握和认识。

七、实验代码

行为建模代码

```
module impl_74LS148(inputdata,out,enable,GS,EO);
input[7:0] inputdata;
input enable;
output reg[2:0] out;
output reg GS;
output reg EO;
always@(inputdata,enable)
begin
    out=3'b111;
    EO=1;
    GS=1;
    if(~enable)
    begin
        GS=0;
        case(inputdata)
            8'b11111111:

            begin
                EO=0;
                GS=1;
                out=3'b111;
            end
            8'b00000000: out=3'b000;
            8'b10000000: out=3'b001;
8'b11000000: out=3'b010;
            8'b11100000: out=3'b011;
            8'b11110000: out=3'b100;
            8'b11111000: out=3'b101;
            8'b11111100: out=3'b110;
            8'b11111110: out=3'b111;
            default: out=3'b111;
        endcase
    end
end
endmodule
```

逐位比较

```
module impl_74LS148
(
	input wire enable,
	input wire [7:0] inputdata,
	output wire [2:0] out,
	output wire GS,
	output wire EO
);
assign EO=(~enable&(inputdata==8'h0ff))?1'b1:1'b0;
assign GS=(enable|(inputdata==8'h0ff))?1'b1:1'b0;
assign
out=enable?3'b111:(~inputdata[0]?3'b000:(~inputdata[1]?3'b001:(~inputdata[2]?3'b010:(~inp
utdata[3]?3'b011:(~inputdata[4]?3'b100:(~inputdata[5]?3'b101:(~inputdata[6]?3'b110:3'b111)
))))));
endmodule
```

实验三：触发器电路

一、实验目的

（1）设计一个触发器电路，实现 74LS76 的所有功能。

（2）能够对触发器电路原理、时序图进行正确的分析。

（3）掌握 FPGA 开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与分析

74LS76 带有独立的 **JK 时钟脉冲、直接清除输入**和**直接设置**的**双 JK 触发器**。触发器的开发方式是，**当时钟设置为高电平时，将接收数据使能输入。**

74LS76 IC 中的 JK 触发器还具有**预设**和**清除**功能，允许 IC 绕过时钟和输入并提供不同的输出。它是基于 TTL 的，可以与任何基于 TTL 的设备或任何微控制器一起操作。IC 有多种封装形式，使 IC 可以根据要求使用任何硬件。如有必要，可以使用多个 IC 制作更多 IC。74LS76 的实际应用是在位的存储中，尽管它对其他应用也很有价值。几个特性使 Jk 触发器成为最常见的类型之一。它们包括以下内容：

①时钟输入属性

②预设输入引脚的存在

同样 **JK 触发器可以通过施加时钟脉冲信号来改变它们的状态**。此时钟信号可以是上升沿或下降沿。并且 74LS76 能够忽略无效输出。

74LS76 封装如下：

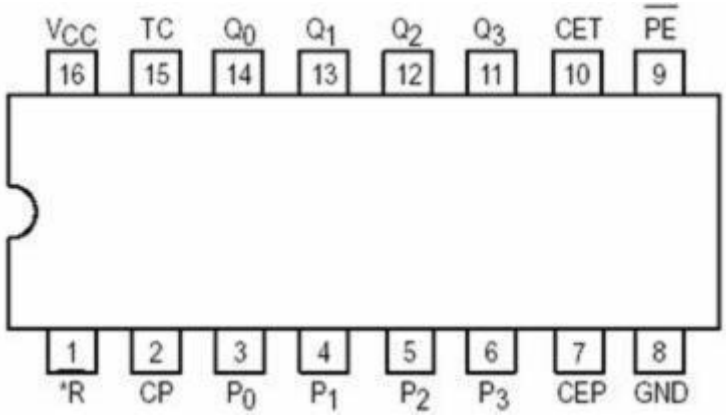


图 1 74LS76 引脚排列图

74LS76 双 JK 触发器引脚说明如下：

引脚序号	引脚名称	引脚描述
引脚 1	1 CLK（时钟）	引脚 1 是输入引脚。它用于将时钟脉冲提供给第一个 JK 触发器。HIGH 到 LOW 脉冲只会影响触发器。
引脚 2	1 Pre'（反转预设）	引脚 2 是预设输入引脚。它用于使第一个触发器的输出（1Q）为高电平。这是一个低电平有效引脚。
引脚 3	1CLR'（反转清除）	引脚 3 是第一个触发器的清零输入引脚。它用于复位第一个触发器的输出。这是一个低电平有效引脚。
引脚 4	1J（数据输入）	引脚 4 是第一个触发器的第一个输入引脚。它用于将第一个输入数据位提供给 IC。它可以是高或低。
引脚 5	VCC （电源）	引脚 5 用作电源引脚。它用于为 IC 上电以使其正常工作。
引脚 6	2 CLK（时钟）	引脚 6 是输入引脚。它用于将时钟脉冲提供给第二个 JK 触发器的时钟。HIGH 到 LOW 脉冲只会影响 IC。
引脚 7	2 Pre'（反转预设）	引脚 7 是第二个触发器的预设输入引脚。它用于使第二个触发器的输出（2Q）为高电平。这是一个低电平有效引脚。
引脚 8	2 CLR' （反转清除）	引脚 8 是第二个触发器的清零输入引脚。它用于复位第二个触发器的输出。这是一个低电平有效引脚。
引脚 9	2J（数据输入）	引脚 9 是第二个触发器的第一个输入引脚。它用于将第一个输入数据位提供给 IC。它可以是高或低。
引脚 10	2Q'（反相输出）	引脚 10 是第二个触发器的第二个输出引脚。它将提供引脚 11 的反相输出。
引脚 11	2Q（输出）	引脚 11 是第二个第一个触发器的第一个输出引脚。它将给出第二个触发器的输出位。
引脚 12	2K（数据输入）	引脚 12 是第二个触发器的第二个输入引脚。它用于将第二个输入数据位提供给 IC。它可以是高或低。
引脚 13	GND（接地）	引脚 13 是接地引脚。它用于与电源和其他设备（如果有）进行公共接地。
引脚 14	1Q' （反相输出）	引脚 14 是第一个触发器的第二个输出引脚。它将提供引脚 15 的反相输出。
引脚 15	1Q （输出）	引脚 15 是第一个触发器的第一个输出引脚。它将给出第一个触发器的输出位。
引脚 16	1K （数据输入）	引脚 16 是第一个触发器的第二个输入引脚。它用于将第二个输入数据位提供给 IC。它可以是高或低。

图 2 74LS76 双 JK 触发器引脚说明

74LS76 真值表如下：

输入					输出	
PR	CLR	CLK	J	K	Q	/Q
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	*	*
H	H	↓	L	L	Q_0	$/Q_0$
H	H	↓	H	L	H	L
H	H	↓	L	H	L	H
H	H	↓	H	H	$/Q_0$	Q_0
H	H	H	X	X	Q_0	$/Q_0$

图 3 74LS76 真值表

在此芯片中预设和清除是异步低电平有效输入。当预设和清除设置为低时，它们会覆盖时钟和 JK 输入，强制输出达到稳态电平。

74LS76 有 5 个输入引脚和两个输出引脚。输出将取决于几乎每个输入引脚。当 IC 在复位引脚处于低电平状态时，输出引脚将为低电平，在反相输出时，状态将为高电平。

现在另一个输入引脚知道为预设。当预设将处于高电平状态时，输出引脚将为高电平，而在反相输出时，状态将为低电平。要使用 IC，我们需要将它们保持在低电平，如果两个引脚上的高电平状态，输出和反相输出都会给出高电平状态。复位引脚和清除将在不同输入处具有这些状态。当 clear = 1 和 preset = 1 时，输出将根据 J 和 K 输入在 HIGH 到 LOW 时钟脉冲上变化。当 J 和 K 两个输入都为低电平时，输出不会有任何变化。输出将取决于先前的状态。在 J = 1 和 K = 1 的情况下，输出将在每个时钟脉冲处保持翻转。其他时刻将根据真值表发生变化

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、74LS76 实验流程框图

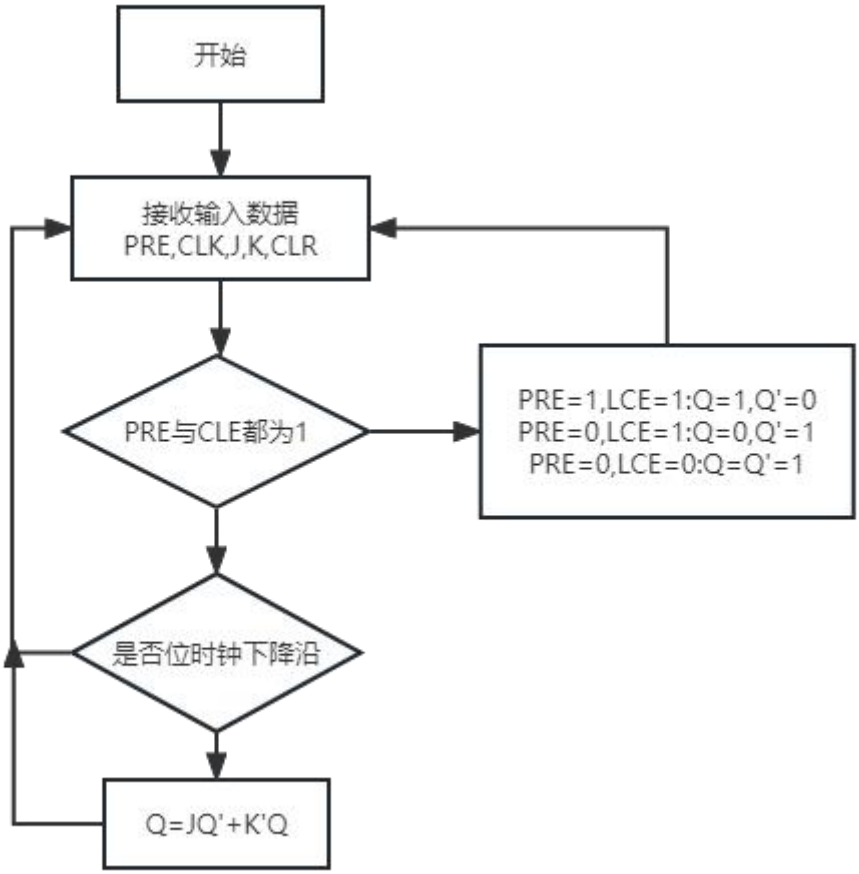


图 4 74LS76 实现流程框图

五、实验内容与结果

（1）实验内容

1、设计一个触发器电路，实现 74LS76 的所有功能。

（2）实验步骤

1、建立工程

2、依照真值表编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板（由于此项工程需要的按键数开发板不满足，所以只进行仿真设计）

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

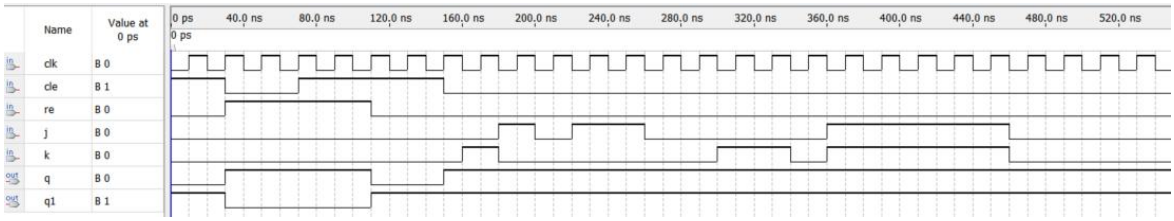


图 5 74LS76 仿真波形图

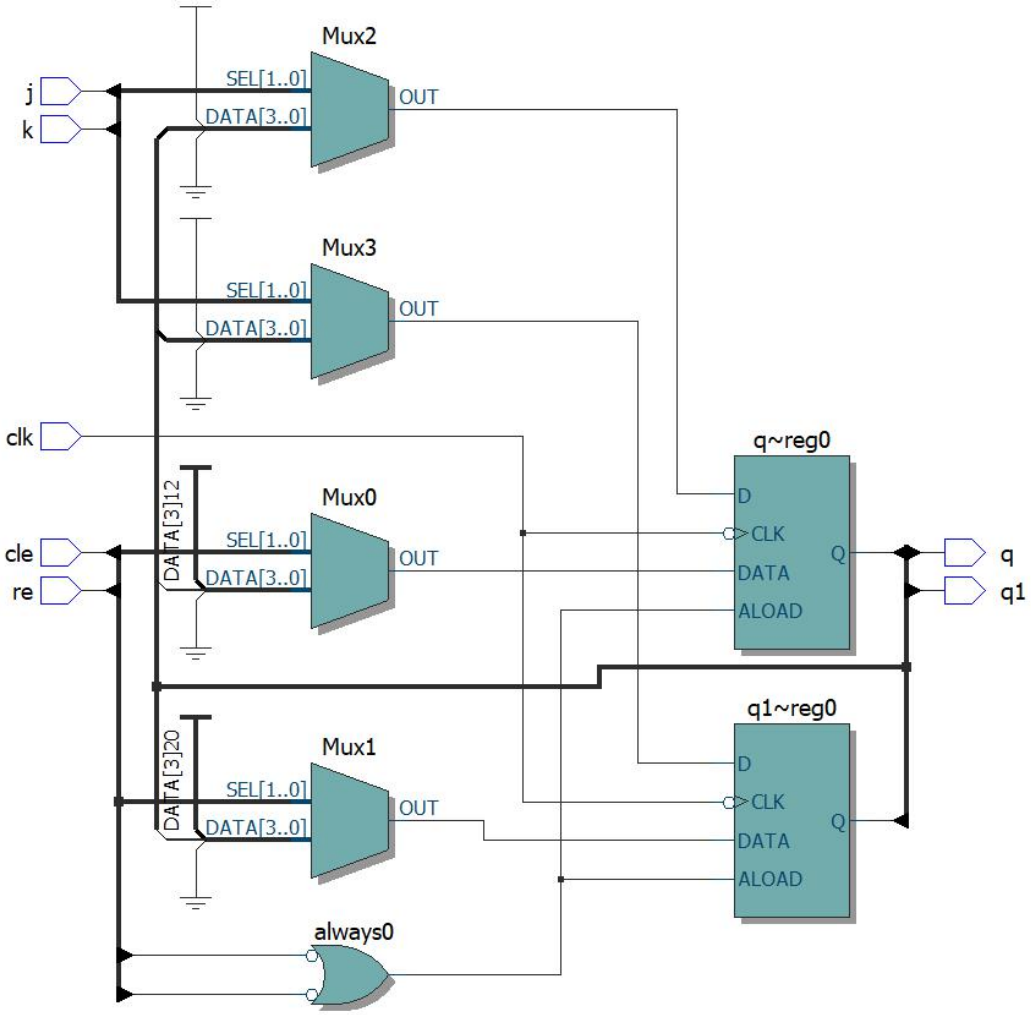


图 6 74LS76 的 RTL 视图

六、实验讨论

1、我们能够发现根据波形图去对应真值表其能够满足一一符合的条件并且其在时钟下降沿也能按照预期进行，同时具有异步操作的特性。

2、对于 RTL 视图进行分析，我们根据其生成的电路图对输入进行进行改变，输出信号依旧能满足 74LS76 的功能，但是其与课本上最原始的电路依旧有一些差距，由于代码的编写方式不同导致了电路生成也有差异。

3、通过这次的实验，我不仅对 FPGA 开发的流程更加熟悉，并且也对触发器电路实现的原理有了更深的理解，加深了我对于课上知识的掌握和认识。

七、实验代码

```
module trigger(clk,re,cle,j,k,q,q1);
    input clk,re,cle,j,k;
    output reg q,q1;
    always@(negedge clk,negedge re,negedge cle)
    if (re == 0 | cle == 0)begin
        case ({re,cle})
            2'b10:begin
                q <= 1;
                q1 <= 0;
            end
            2'b01:begin
                q <= 0;
                q1 <= 1;
            end
            2'b00:begin
                q <= 1;
                q1 <= 1;
            end
        endcase
    end
    else    begin
                case ({j,k})
                    2'b00:begin
                        q <=q;
                        q1 <=q1;
                    end
```

```
                    2'b01:begin
                        q <= 0;
                        q1 <= 1;
                    end
                    2'b10:begin
                        q <= 1;
                        q1 <= 0;
                    end
                    2'b11:begin
                        q <= q1;
                        q1 <=q;
                    end
                endcase
            end
endmodule
```

实验四：彩灯控制电路

一、实验目的

（1）设计一个汽车尾灯控制电路。已知汽车左右两侧各有 3 个尾灯，采用 K0、K1 进行状态控制，要求尾灯按一定规则亮灭。

（2）能够对流水灯电路原理、时序图进行正确的分析。

（3）掌握 FPGA 开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与分析

在这里对于彩灯的控制，我们可以使用状态机进行控制，其中通过对每个状态进行一段时间的显示后，到达指定的状态显示时间后则可以进行下一个状态显示，在此题的实现方法位： 共 6 种状态，分别对应 6 种演示花型 以一定周期去切换状态。 考虑到第二种情况最少需要 8 个时钟周期，我们以 8 个时钟周期为一个大周期。去显示第 2,3,4,5 种状态而以 6 秒为一个大周期显示第 1，3 种状态。可以用下面这幅图来展现状态切换：

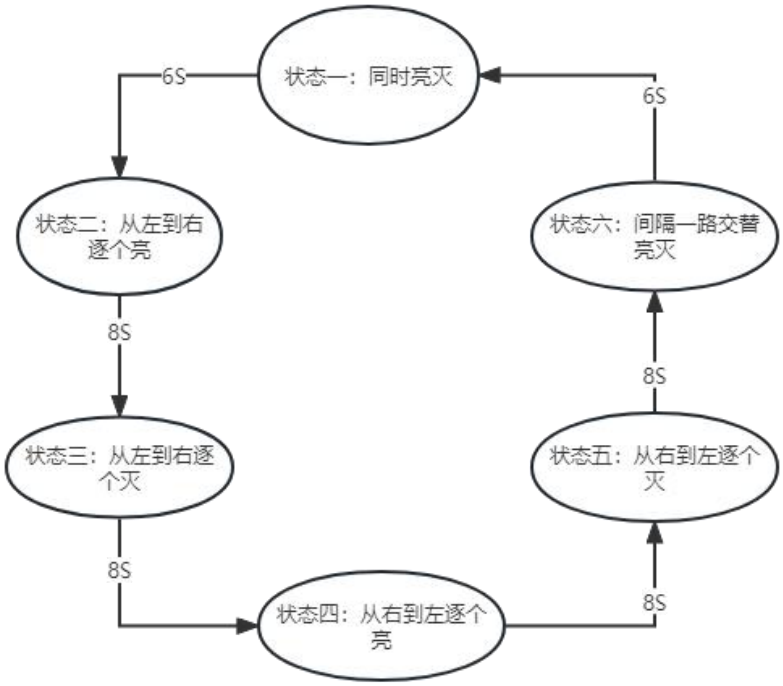


图 1 彩色灯状态切换图

在这里我们也需要考虑秒时钟信号的生成，这里为了使用共用的时钟管理系统,所以进行生成秒时钟信号时我采用脉冲生成,即每计数到 1 秒对应的一个值，则产生一个时钟的脉冲，以 50MHz 晶振源为例，在 1 秒钟内会产生 50M 个时钟周期，那么我们就可以每过 50M 个时钟周期对计数器进行加一。这样就可以实现任意时间段的计数器了，比如 0.5s（每过 25M 个时钟周期对计数器进行加一），这样就构成了我们的秒计数器，但是为了使得更易于观察，我采用将 1 秒进行细化，细化为 1000MS，同时将 1MS 细化为 1000US，使得对于时钟的观察更加细致。

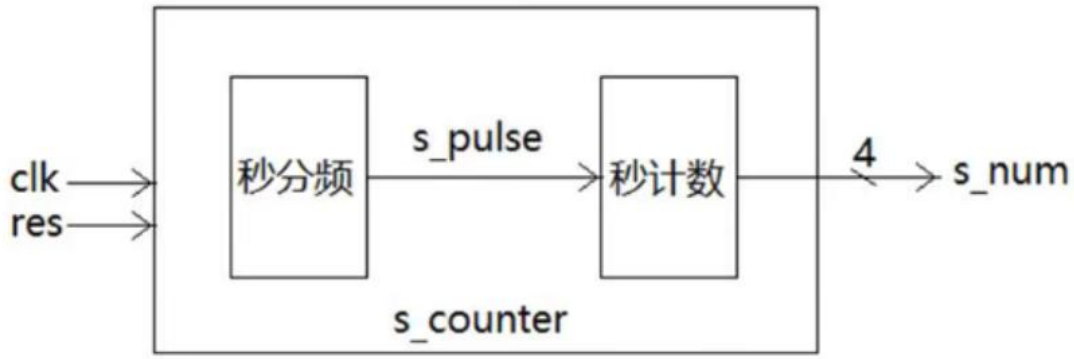


图 2 秒时钟产生原理

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、八路彩灯实验流程代码图

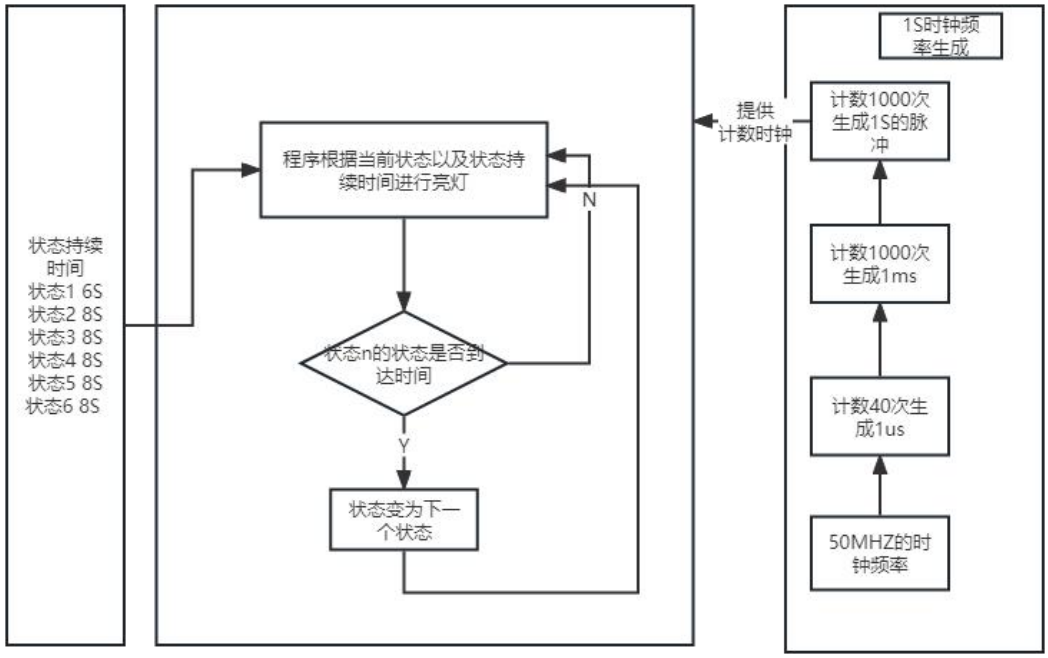


图 3 实验代码流程图

五、实验内容与结果

（1）实验内容

设计一个 8 路彩灯控制程序，要求彩灯重复显示以下 6 种演示花型，在演示过程中，只有当一种花型演示完毕才能转向其他演示花型。

(1)8 路彩灯同时亮灭；

(2)从左至右逐个亮（每次只有 1 路亮）；

(3)从左至右逐个灭（每次只有 1 路亮）；

(4)从右至左逐个亮（每次只有 1 路亮）；

(5)从右至左逐个灭（每次只有 1 路亮）；

(6)8 路彩灯每次间隔 1 路灯亮，1 路灯灭，且亮灭相间，交替亮灭。

（2）实验步骤

1、建立工程

2、依照真值表编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

（附件中带有视频，展现的为关键状态切换时的截图）

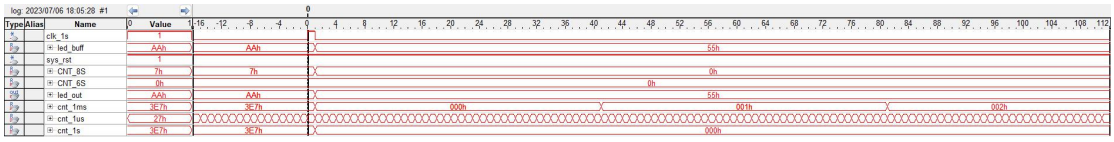


图 4 1s 时钟脉冲上升沿，同时灯处于交替亮灭状态（AA-55）

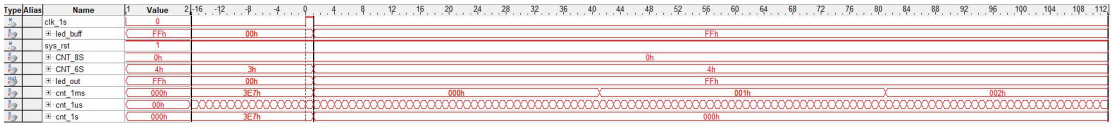


图 5 1s 时钟脉冲下降沿，同时灯处于第一个状态从全灭变为全亮（FF-00）

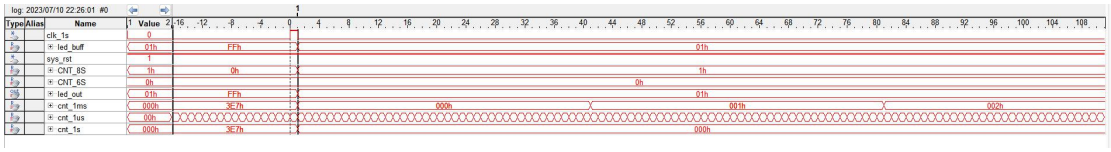


图 6　灯从第一个状态（全亮灭）变到第二个状态（流水灯亮）

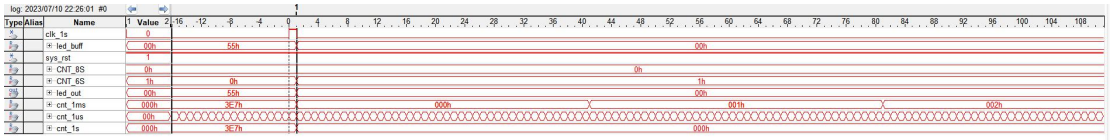


图 7　灯从第六个状态（交替亮灭）变到第一个状态（全亮全灭）

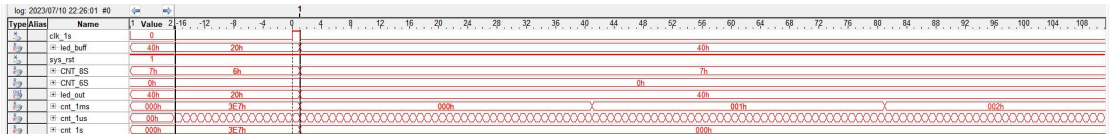


图 8　灯进行流水灯亮的过程图

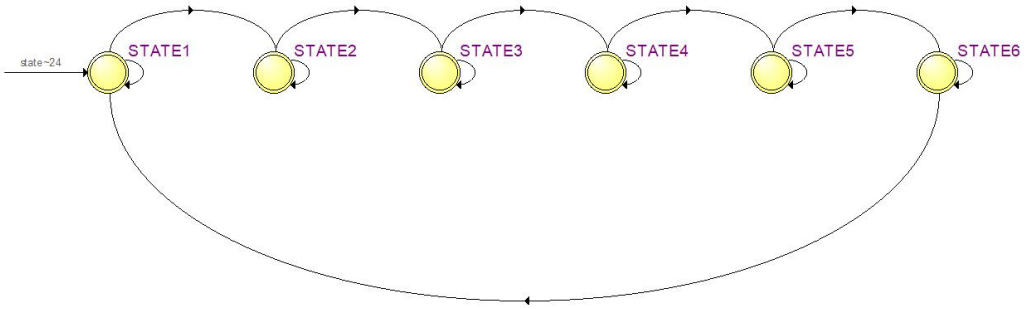


图 9 彩色灯程序的状态转移图

六、实验讨论

1、我们通过对烧入进 FPGA 开发板的程序进行观测，发现其与实验的要求一致并进行了视频记录,但是我们需要对波形图进一步观测才能够进一步确认代码的编写是否正确。

2、在代码的开发过程中，出现了逻辑问题，对于状态的判断应该与计数器的判断同步进行，否则会时钟有一个 1S 的计数周期内为空状态没有进行使用。

3、对于波形图我们进行状态转换时刻进行观察，能够发现其与预期一致，在 1S 时钟上升沿进入状态更换，在 1S 的下降沿完成的状态的切换，满足预期。

4、通过这次的实验，我首次对状态机进行实践编写，对状态的切换，同步时序的梳理有了更深一步的认识，同时也加深了我对 FPGA 开发板整个时钟管理的理解。

七、实验代码

```
module led
#(parameter CNT_MAX=11'd999,
   parameter CNT_MAX_1ms=11'd999,
   parameter CNT_MAX_1us=6'd39,
   parameter CNT6_MAX=4'd6,
   parameter CNT8_MAX=4'd8
)
(
      input wire clk,
      input wire sys_rst,
      output wire [7:0] led_out
);
parameter   STATE1=6'b000001,
                  STATE2=6'b000010,
                  STATE3=6'b000100,
                  STATE4=6'b001000,
                  STATE5=6'b010000,
                  STATE6=6'b100000;
reg [10:0] cnt_1s;
reg [5:0] cnt_1us;
reg [10:0] cnt_1ms;
reg [3:0]CNT_8S;
reg [5:0]state;
reg [3:0]CNT_6S;
reg clk_1s;
reg [7:0]led_buff;
assign led_out=led_buff;
always@(posedge clk or negedge sys_rst)
      if(sys_rst==1'b0)
            cnt_1s<=11'd0;
      else      if      (cnt_1s==CNT_MAX      &&      cnt_1ms==CNT_MAX_1ms      &&
cnt_1us==CNT_MAX_1us)
            cnt_1s<=11'd0;
      else if(cnt_1ms==CNT_MAX_1ms && cnt_1us==CNT_MAX_1us)
            cnt_1s<=cnt_1s+11'd1;
      else
            cnt_1s<=cnt_1s;
always@(posedge clk or negedge sys_rst)
      if(sys_rst==1'b0)
            cnt_1ms<=11'd0;
      else if (cnt_1ms==CNT_MAX_1ms && cnt_1us==CNT_MAX_1us)
```

```
cnt_1ms<=11'd0;
    else if(cnt_1us==CNT_MAX_1us)
        cnt_1ms<=cnt_1ms+11'd1;
    else
        cnt_1ms<=cnt_1ms;
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        clk_1s<=1'b0;
    else if (cnt_1s==CNT_MAX && cnt_1ms==CNT_MAX_1ms &&
cnt_1us==CNT_MAX_1us-1)
        clk_1s<=1'b1;
    else
        clk_1s<=1'b0;
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        cnt_1us<=6'b0;
    else if ( cnt_1us==CNT_MAX_1us)
        cnt_1us<=6'b0;
    else
        cnt_1us<=cnt_1us+6'd1;
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
    begin
        state<=STATE1;
        CNT_6S<=4'd0;;
        CNT_8S<=4'd0;
        led_buff<=8'b00000000;
    end
    else if(clk_1s==1'b1)
        case(state)
            STATE1:begin
                if(CNT_6S==CNT6_MAX-1)begin
                    CNT_6S<=4'd0;
                    state<=STATE2;
                    end
                else
                    CNT_6S<=CNT_6S+4'd1;
                if(CNT_6S==4'd0)
                    led_buff<=8'b00000000;
                else
                    led_buff<=~led_buff;
                end
            led_buff[6:0],led_buff[7]};
```

```
STATE2:begin
    if(CNT_8S==CNT8_MAX-1)begin
        CNT_8S<=4'd0;
        state<=STATE3;
        end
    else
        CNT_8S<=CNT_8S+1;
    if(CNT_8S==4'd0)
        led_buff<=8'b00000001;
    else
        led_buff<={led_buff[6:0],led_buff[7]};
    end
STATE3:begin
    if(CNT_8S==CNT8_MAX-1)begin
        CNT_8S<=4'd0;
        state<=STATE4;
        end
    else
        CNT_8S<=CNT_8S+1;
    if(CNT_8S==4'd0)
        led_buff<=8'b11111110;
    else
        led_buff<={led_buff[6:0],led_buff[7]};
    end
STATE4:begin
    if(CNT_8S==CNT8_MAX-1)begin
        CNT_8S<=4'd0;
        state<=STATE5;
        end
    else
        CNT_8S<=CNT_8S+1;
    if(CNT_8S==4'd0)
        led_buff<=8'b10000000;
    else
        led_buff<={led_buff[0],led_buff[7:1]};
    end
STATE5:begin
    if(CNT_8S==CNT8_MAX-1)begin
        CNT_8S<=4'd0;
        state<=STATE6;
        end
    else
        CNT_8S<=CNT_8S+1;
```

```
                    if(CNT_8S==4'd0)
                        led_buff<=8'b01111111;
                    else
                        led_buff<={led_buff[0],led_buff[7:1]};
                    end
                STATE6:begin
                    if(CNT_8S==CNT8_MAX-1)begin
                        CNT_8S<=4'd0;
                        state<=STATE1;
                        end
                    else
                        CNT_8S<=CNT_8S+1;
                    if(CNT_8S==4'd0)
                        led_buff<=8'b10101010;
                    else
                        led_buff<=~led_buff;
                    end
            endcase
        else
        begin
        CNT_8S<=CNT_8S;
        CNT_6S<=CNT_6S;
        state<=state;
        led_buff<=led_buff;
        end
endmodule
```

实验五：交通灯控制电路

一、实验目的

（1）设计一个 8 路彩灯控制程序，要求彩灯重复显示以下 6 种演示花型，在演示过程中，只有当一种花型演示完毕才能转向其他演示花型。

（2）能够对状态机原理与书写、时序图进行正确的分析。

（3）掌握 FPGA 开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与分析

由于当输入为 11 时其下一个状态需要根据上一个状态的值去改变，所以我们也考虑状态机的实现，依据输入的按键去改变状态值，当输入为 11 时，依据上一个的状态值去改变将要出现的尾灯亮灭规则，同时处于某个状态时，为了使得对于结果的观察能够更为清楚，所以我们将会产生一个 1S 的时钟，并以它来控制灯的亮灭时长，即如果需要尾灯顺序点亮时，灯亮的时长为 1S，灭的时长也为 1S。其状态切换表为如下：

表 1 交通灯状态切换图

当前状态	当前输入	下一时刻输出状态
直行	00	直行
直行	10	左转
直行	01	右转
直行	11	直行时刹车
直行时刹车	00	直行
直行时刹车	10	左转
直行时刹车	01	右转
直行时刹车	11	直行时刹车
其他状态依照直行状态类似进行		

值得注意的是，由于我们所使用并且保存的为上一个状态的值去改变下一个状态，那么我们所需要用到的状态实际上只有三个，分别是直行、左转、右转，而输入为 11 时会依据上一个状态去调整输入，而不会改变上一个状态的值。

在这里我们也需要考虑秒时钟信号的生成，这里为了使用共用的时钟管理系统，所以进行生成秒时钟信号时我采用脉冲生成，即每计数到 1 秒对应的一个值，则产生一个时钟的脉冲，以 50MHz 晶振源为例，在 1 秒钟内会产生 50M 个时钟周期，那么我们就可以每过 50M 个时钟周期对计数器进行加一。这样就可以实现任意时间段的计数器了，比如 0.5s（每过 25M 个时钟周期对计数器进行加一），这样就构成了我们的秒计数器，但是为了使得更易于观察，我采用将 1 秒进行细化，细化为 1000MS，同时将 1MS 细化为 1000US，使得对于时钟的观察更加细致。

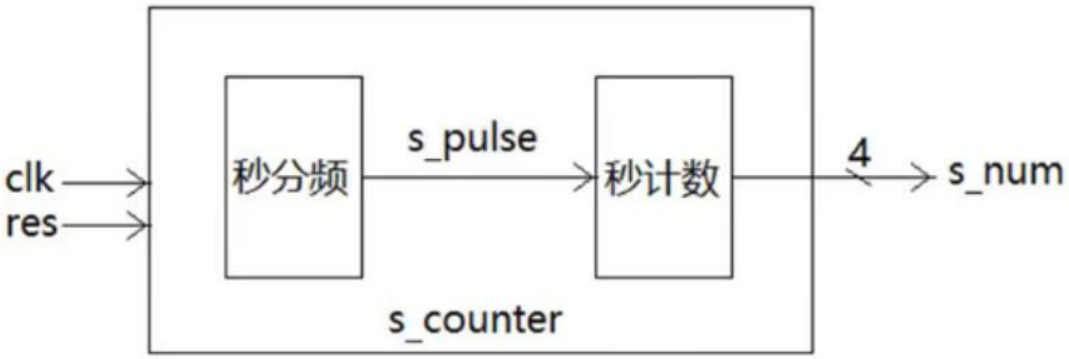


图 1 秒时钟产生原理

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、实验内容与结果

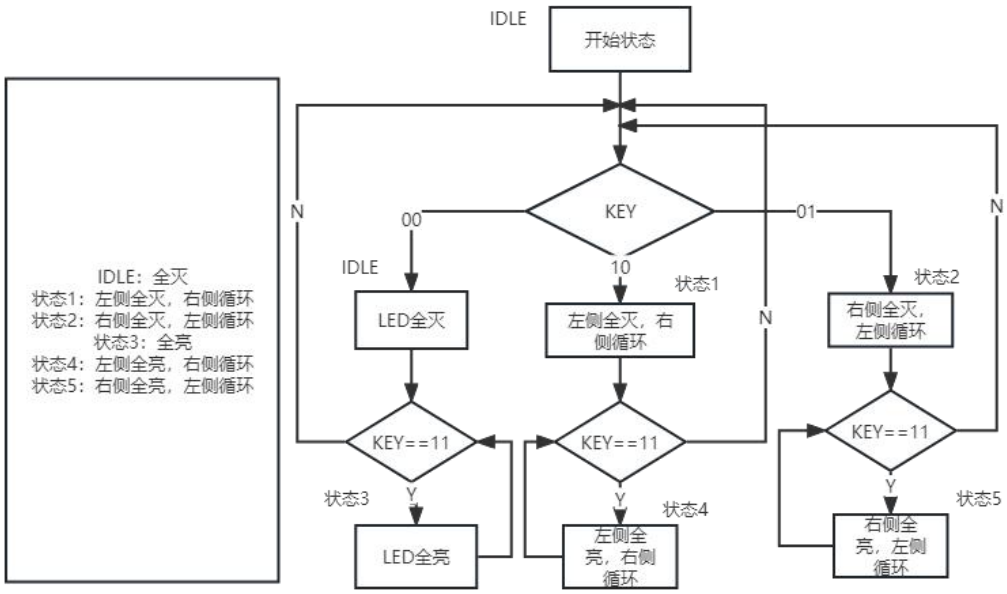


图 2 交通灯程序流程图

五、实验内容与结果

（1）实验内容

设计一个汽车尾灯控制电路。已知汽车左右两侧各有 3 个尾灯，采用 K0、K1 进行状态控制，要求尾灯按如下规则亮灭。

(1)汽车沿直线行驶时，两侧的指示灯全灭； （KEY=2’b00)

(2)右转弯时，左侧的指示灯全灭，右侧的指示灯按 000，100，010，001，000 循环顺序点亮；（KEY=2’b01)

(3)左转弯时，右侧的指示灯全灭，左侧的指示灯按与右侧同样的循环顺序点亮；（KEY=2’b10)

(4)如果在直行时刹车，两侧的指示灯全亮；如果在转弯时刹车，转弯这一侧的指示灯按上述的循环顺序点亮，另一侧的指示灯全亮。（KEY=2’b11)

（2）实验步骤

1、建立工程

2、依照真值表编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

（附件中带有视频，展现的为关键状态切换时的截图）

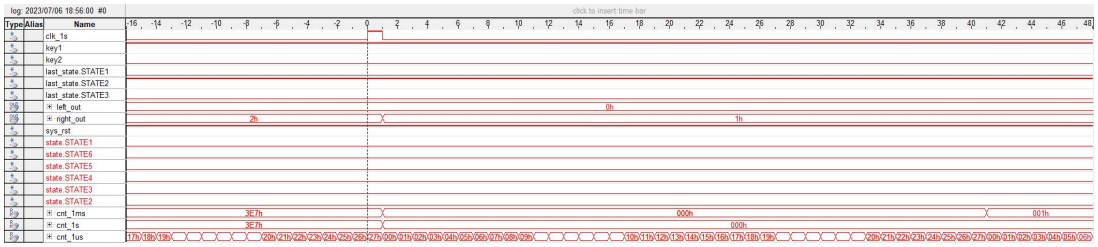


图 3 输入 KEY=2’b10 左灯全灭右灯循环

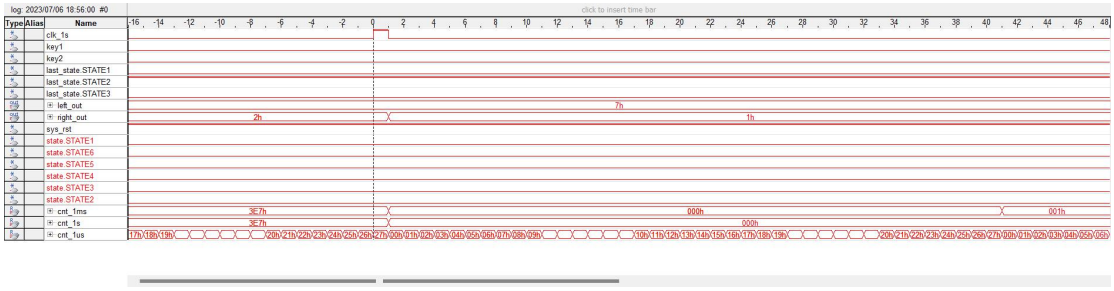


图 4 输入 KEY=2’b10 到输入 2’b00 左灯全亮右灯循环

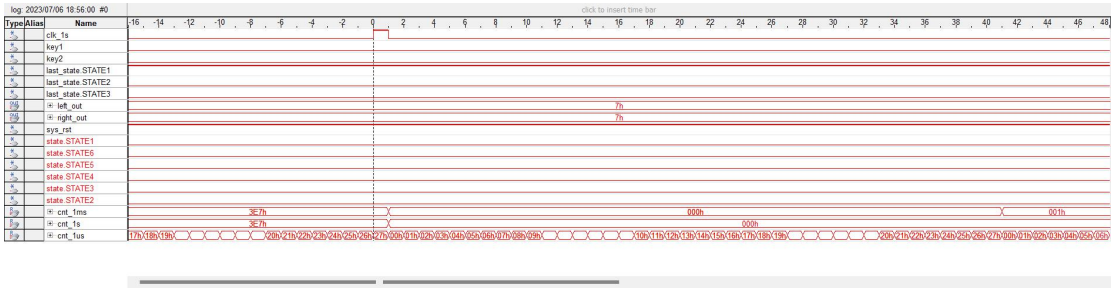


图 5 输入 KEY=2’b11 到输入 2’b00 两侧尾灯全亮

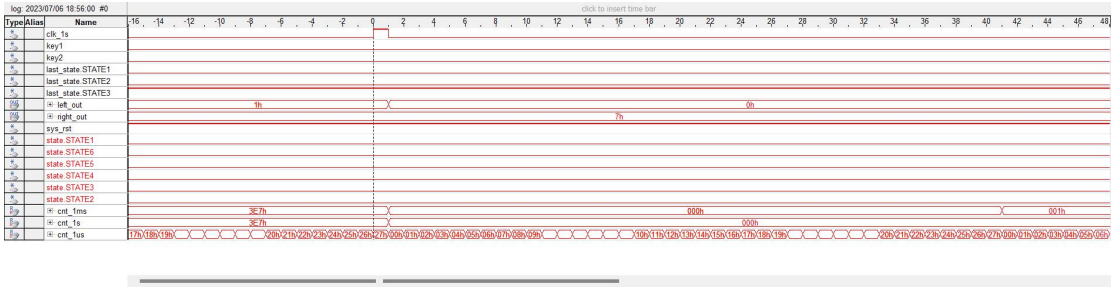


图 6 输入 KEY=2’b01 到输入 2’b00 右灯全亮左灯开始循环

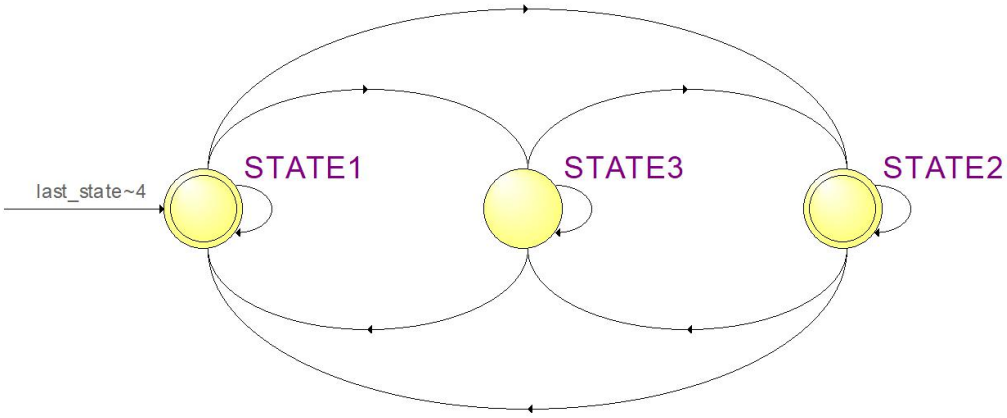


图 7 交通灯的 last 状态转移图

六、实验讨论

1、我们通过对烧入进 FPGA 开发板的程序进行观测，并根据开发板自带的按键进行测试，发现其与实验的要求一致并进行了视频记录，但是我们需要对波形图进一步观测才能够进一步确认代码的编写是否正确。

2、对于波形图我们进行状态转换时刻进行观察，能够发现其与预期一致，在 1S 时钟上升沿进入状态更换，在 1S 的下降沿完成的状态的切换，满足预期。并且其 last 的状态也能够根据输入的按键在 1S 的时钟沿及时的发生变化，与所预想的也十分符合。

3、通过这次的实验，对状态机的理解更深，状态的切换、同步时序的梳理有了更深一步的认识，同时也加深了我对 FPGA 开发板整个时钟管理的理解。

七、实验代码

```
module home_light
#(parameter CNT_MAX=11'd999,
   parameter CNT_MAX_1ms=11'd999,
   parameter CNT_MAX_1us=6'd39)
(
      input wire key1,
      input wire key2,
      input wire clk,
      input wire sys_rst,
      output reg [2:0] left_out,
      output reg [2:0] right_out
);
parameter STATE1=6'b000001,
                STATE2=6'b000010,
                STATE3=6'b000100,
                STATE4=6'b001000,
                STATE5=6'b010000,
                STATE6=6'b100000;
reg [10:0] cnt_1s;
reg [5:0] cnt_1us;
reg [10:0] cnt_1ms;
reg [5:0] last_state;
reg clk_1s;
wire [1:0]flag;
assign flag={key1,key2};
always@(posedge clk or negedge sys_rst)
      if(sys_rst==1'b0)
            cnt_1s<=11'd0;
      elseif(cnt_1s==CNT_MAX&&cnt_1ms==CNT_MAX_1ms&&cnt_1us==CNT_MAX_
1us)
            cnt_1s<=11'd0;
      else if(cnt_1ms==CNT_MAX_1ms && cnt_1us==CNT_MAX_1us)
            cnt_1s<=cnt_1s+11'd1;
      else
            cnt_1s<=cnt_1s;
always@(posedge clk or negedge sys_rst)
      if(sys_rst==1'b0)
            cnt_1ms<=11'd0;
      else if (cnt_1ms==CNT_MAX_1ms && cnt_1us==CNT_MAX_1us)
            cnt_1ms<=11'd0;
      else if(cnt_1us==CNT_MAX_1us)
            cnt_1ms<=cnt_1ms+11'd1;
      else
            cnt_1ms<=cnt_1ms;
```

```
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        clk_1s<=1'b0;
    elseif(cnt_1s==CNT_MAX&&cnt_1ms==CNT_MAX_1ms&&cnt_1us==CNT_MAX_
1us-1)
        clk_1s<=1'b1;
    else
        clk_1s<=1'b0;
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        cnt_1us<=6'b0;
    else if ( cnt_1us==CNT_MAX_1us)
        cnt_1us<=6'b0;
    else
        cnt_1us<=cnt_1us+6'd1;
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
    begin
        left_out<=3'b000;
        right_out<=3'b000;
        last_state<=STATE1;
    end
    else if(clk_1s==1'b1)
    begin
        case(flag)
            2'b11:begin
                    left_out<=3'b000;
                    right_out<=3'b000;
                    last_state<=STATE1;
                end
            2'b10:begin
                    left_out<=3'b000;
                    if(right_out==3'b000)
                        right_out<=3'b100;
                    else
                        right_out<=right_out>>1;
                    last_state<=STATE2;
                end
            2'b01:begin
                    right_out<=3'b000;
                    if(left_out==3'b000)
                        left_out<=3'b100;
```

```
                    else
                        left_out<=left_out>>1;
                    last_state<=STATE3;
                end
            2'b00:begin
                    if(last_state==STATE1)
                        begin
                            left_out<=3'b111;
                            right_out<=3'b111;
                        end
                    else if(last_state==STATE2)
                    begin
                        left_out<=3'b111;
                        if(right_out==3'b000)
                            right_out<=3'b100;
                        else
                            right_out<=right_out>>1;
                    end
                    else
                    begin
                        right_out<=3'b111;
                        if((left_out==3'b000)|(left_out==3'b111) )
                            left_out<=3'b100;
                        else
                            left_out<=left_out>>1;
                    end
                end
            default:last_state<=STATE1;
        endcase
    end
else
    begin
        last_state<=last_state;
        left_out<=left_out;
        right_out<=right_out;
    end
endmodule
```

实验六：序列检测电路

一、实验目的

（1）设计一个“10010”串行数据检测器。当检测到数据串时，Y 输出高电平，其余时间 Y 输出低电平。输入 X：
00010101001001110100101010001010100100111010010101010101。

（2）能够对序列检测电路原理、时序图进行正确的分析。

（3）掌握 FPGA 开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与分析

对于串行序列检测电路，我们一开始想到的便是利用状态机进行检测，即根据需要检测的需要去设定当前输入下所处于的状态并根据输入去判断下一个状态，由于此实验所要求的序列检测为 10010，所以我们可以设定 6 个状态进行检测，并根据输入去更改下一状态，具体的状态转移图如下图所示：

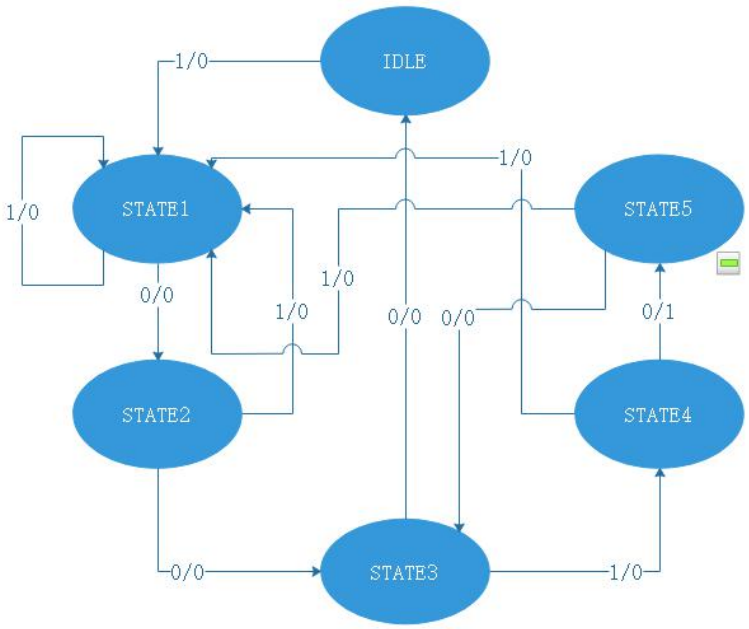


图 1 序列检测状态检测图

这幅图的意思即是根据当前所处的状态下去判断输入来决定输出与下一个状态。

在这个实验中，为了使得检测结果更加清晰可见，所以我们引出二个输出引

脚，一个作为是否有状态发生改变，一个作为是否有检测到所期望的序列分别为 LED0 与 LED1。

在实际中为了检测所希望的序列，并能够清楚的观察结果，所以我在实际应用中使用 1S 检测一个序列信号输入，检测到正确序列则 LED1 亮 1S，若发生了状态改变则 LED0 亮 1S.

在这里我们也需要考虑秒时钟信号的生成，这里为了使用共用的时钟管理系统,所以进行生成秒时钟信号时我采用脉冲生成,即每计数到 1 秒对应的一个值，则产生一个时钟的脉冲，以 50MHz 晶振源为例，在 1 秒钟内会产生 50M 个时钟周期，那么我们就可以每过 50M 个时钟周期对计数器进行加一。这样就可以实现任意时间段的计数器了，比如 0.5s（每过 25M 个时钟周期对计数器进行加一），这样就构成了我们的秒计数器，但是为了使得更易于观察，我采用将 1 秒进行细化，细化为 1000MS，同时将 1MS 细化为 1000US，使得对于时钟的观察更加细致。

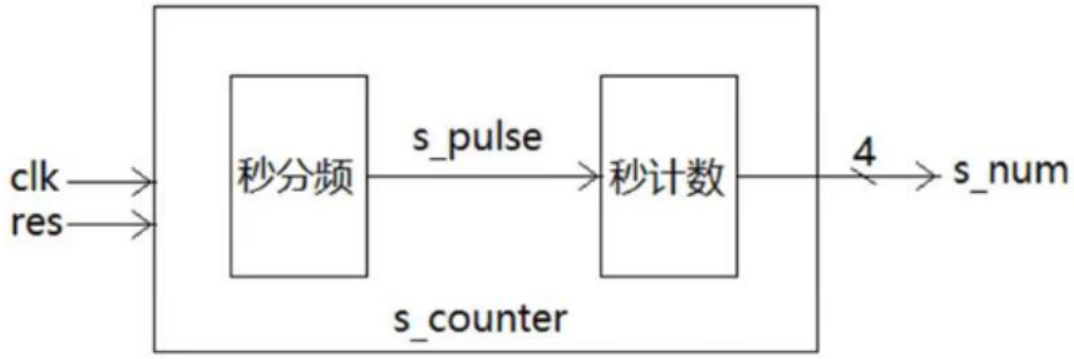


图 2 秒时钟产生原理

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、序列检测程序流程框图

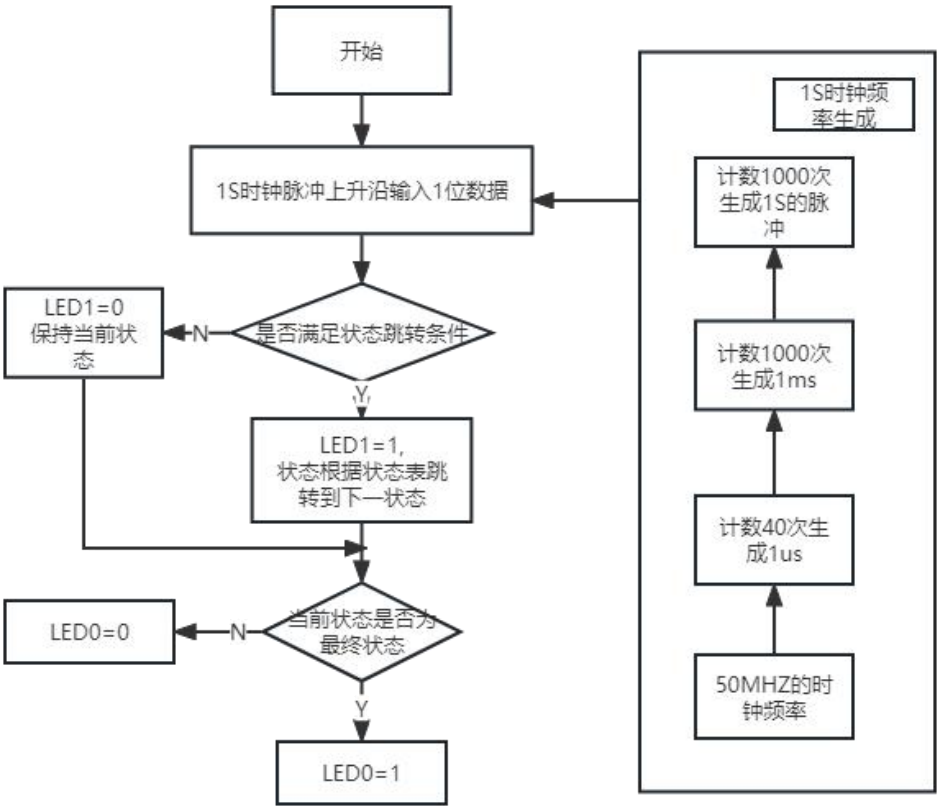


图 3 序列检测程序流程图

五、实验内容与结果

（1）实验内容

设计一个“10010”串行数据检测器。当检测到数据串时，Y 输出高电平，其余时间 Y 输出低电平。输入 X:

“00010101001001110100101010001010100100111010010101010101”。

（2）实验步骤

1、建立工程

2、依照真值表编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

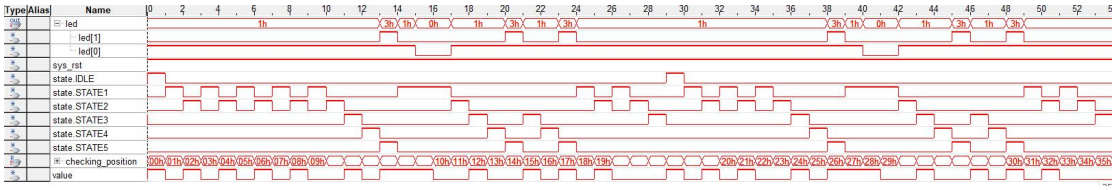


图 4 序列检测时序图（其中 value 为要检测的序列，led1 为检测到正确序列为高，led0 为状态改变时为高）

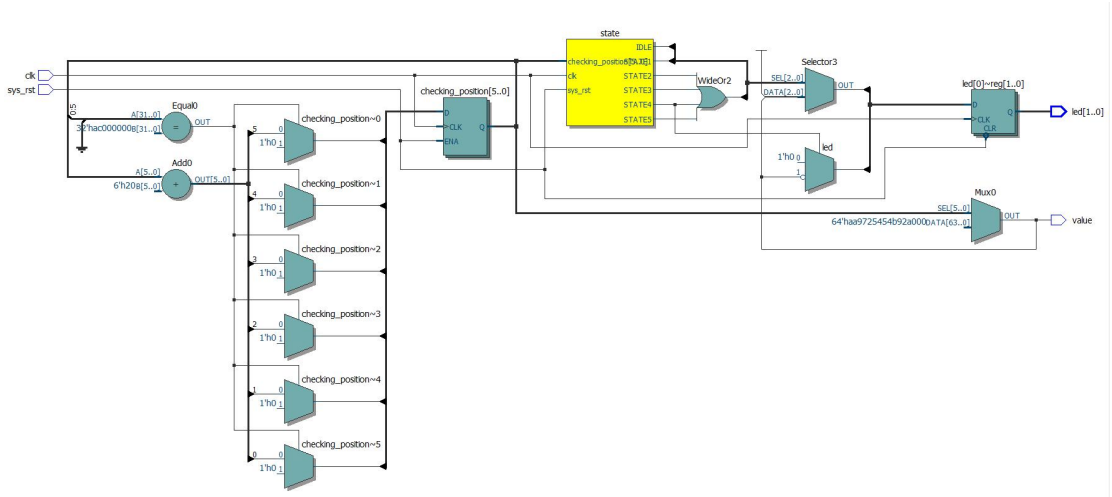


图 5 10010 序列检测 RTL 电路视图

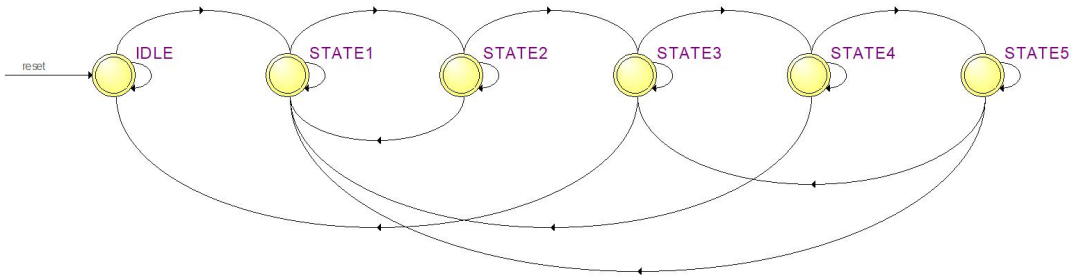


图 6 10010 序列检测状态转移图

在这里需要注意的是由于进行示波器观察时，如果进行 1S 检测一个输入的话无法进行实际观测，所以在进行检测代码逻辑时，我并没有进行分频处理，而是使用系统的时钟进行检测，能够发现其高电平的时间点与预期的一致。

六、实验讨论

1、我们通过对烧入进 FPGA 开发板的程序进行观测，并对照所希望进行检测

的序列进行人为检测与代码所输出的高电平位置相对比，发现其与实验的要求一致但是我们需要对波形图进一步观测才能够进一步确认代码的编写是否正确。

2、对于波形图我们进行状态转换时刻、序列输入与输出时刻进行观察，能够发现其与预期一致。

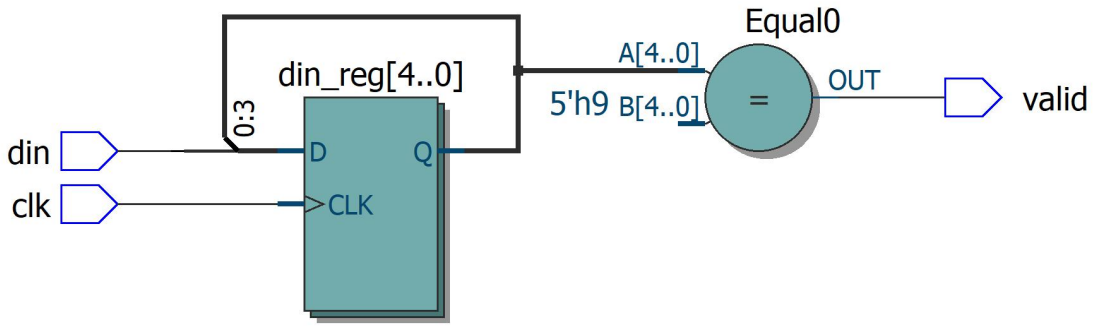
3、通过这次的实验，对状态机的理解更深，状态的切换、同步时序的梳理有了更深一步的认识，同时也加深了我对 FPGA 开发板整个时钟管理的理解。

4、对于此实验我认为还可以考虑使用移位寄存器的方法进行，使得效率更高即代码结构如下图

```
module check_num(clk,din,valid);
        input clk;
        input din;
        output valid;
reg[4:0] din_reg;
always @(posedge clk)
begin
        din_reg <= {din_reg[4:0],din};
end

assign valid = (din_reg == 5'b10010)?1'b1:1'b0;
```

其 RTL 视图如下图所示



七、实验代码

```
module check_num   //需要一秒检测一个时使用参数
#(parameter CNT_MAX=11'd999,
   parameter CNT_MAX_1ms=11'd999,
   parameter CNT_MAX_1us=6'd39
)
(
      input wire clk,
      output reg [1:0] led,
      input wire sys_rst,
      output reg value
);
reg [10:0] cnt_1s;
reg [5:0] cnt_1us;
reg [10:0] cnt_1ms;
parameter IDLE=6'b000001,
               STATE1=6'b000010,
               STATE2=6'b000100,
               STATE3=6'b001000,
               STATE4=6'b010000,
               STATE5=6'b100000;

reg[53:0]checking_value
=54'b00010101_00100111_01001010_10001010_10010011_10100101_010101;
reg[5:0] checking_position=6'd0;
parameter sequence_length = 54;
reg [5:0]state=IDLE;
reg clk_1s;
           cnt_1ms<=cnt_1ms;
//需要一秒检测一个数才需要产生一秒一个的时钟
always@(posedge clk or negedge sys_rst)
      if(sys_rst==1'b0)
           cnt_1s<=11'd0;
      else      if      (cnt_1s==CNT_MAX      &&      cnt_1ms==CNT_MAX_1ms      &&
cnt_1us==CNT_MAX_1us)
           cnt_1s<=11'd0;
      else if(cnt_1ms==CNT_MAX_1ms && cnt_1us==CNT_MAX_1us)
           cnt_1s<=cnt_1s+11'd1;
      else
           cnt_1s<=cnt_1s;
always@(posedge clk or negedge sys_rst)
      if(sys_rst==1'b0)
           cnt_1ms<=11'd0;
      else if (cnt_1ms==CNT_MAX_1ms && cnt_1us==CNT_MAX_1us)
```

```
            cnt_1ms<=11'd0;
        else if(cnt_1us==CNT_MAX_1us)
            cnt_1ms<=cnt_1ms+11'd1;
        else
            cnt_1ms<=cnt_1ms;
always@(posedge clk or negedge sys_rst)
        if(sys_rst==1'b0)
            clk_1s<=1'b0;
        else if (cnt_1s==CNT_MAX && cnt_1ms==CNT_MAX_1ms &&
cnt_1us==CNT_MAX_1us-1)
            clk_1s<=1'b1;
        else
            clk_1s<=1'b0;
always@(posedge clk or negedge sys_rst)
        if(sys_rst==1'b0)
            cnt_1us<=6'b0;
        else if ( cnt_1us==CNT_MAX_1us)
            cnt_1us<=6'b0;
        else
            cnt_1us<=cnt_1us+6'd1;
always@(posedge clk or negedge sys_rst)
        if(sys_rst==1'b0)
        begin
            led[0]<=1'b0;
            led[1]<=1'b0;
        end
        else if(clk_1s==1'b1)//需要一秒检测一个数才需要
        begin
            if(checking_position==sequence_length-1)
                checking_position<=6'd0;
            else
                checking_position<=checking_position+6'd1;
            value<=checking_value[checking_position];
            case(state)
                IDLE:begin

                            if(checking_value[checking_position]==1'b1)
                            begin
                                state<=STATE1;
                                led[0]<=1'b1;
                                led[1]<=1'b0;

                            end
```

```
                else
                begin
                    state<=state;
                    led[0]<=1'b0;
                    led[1]<=1'b0;
                end
            end
        STATE1:begin

                if(checking_value[checking_position]==1'b0)
                begin
                    state<=STATE2;
                    led[0]<=1'b1;
                    led[1]<=1'b0;

                end
                else
                begin
                    state<=state;
                    led[0]<=1'b0;
                    led[1]<=1'b0;
                end
            end
        STATE2:begin

                if(checking_value[checking_position]==1'b0)
                begin

            End
                    state<=STATE3;
                    led[0]<=1'b1;
                    led[1]<=1'b0;

                end
                else
                begin
                    state<=STATE1;
                    led[0]<=1'b1;
                    led[1]<=1'b0;
                end
            End
        STATE3:begin
```

```
            if(checking_value[checking_position]==1'b1)
            begin
                state<=STATE4;
                led[0]<=1'b1;
                led[1]<=1'b0;
            end
            else
            begin
                state<=IDLE;
                led[0]<=1'b1;
                led[1]<=1'b0;
            end
        end
    STATE4:begin
            if(checking_value[checking_position]==1'b0)
            begin
                state<=STATE5;
                led[0]<=1'b1;
                led[1]<=1'b1;
            end
            else
            begin
                state<=STATE1;
                led[0]<=1'b1;
                led[1]<=1'b0;
            end
        end
    STATE5:begin
            if(checking_value[checking_position]==1'b0)
            begin
                state<=STATE3;
                led[0]<=1'b1;
                led[1]<=1'b0;
            end
            else
            begin
                state<=STATE1;
                led[0]<=1'b1;
                led[1]<=1'b0;
            end
        end
    default:state<=STATE1;
```

```
            endcase

        end
        else
        begin
            led[0]<=led[0];
            led[1]<=led[1];
            state<=state;
            checking_position<=checking_position;
        end
endmodule
```

实验七：半分频电路

一、实验目的

（1）设计一个9.5分频的分频器。

（2）能够对时钟分频电路原理、时序图进行正确的分析。

（3）掌握FPGA开发流程，代码书写规范，时序图绘制清楚。

二、实验原理与分析

利用时钟的双边沿逻辑，可以对时钟进行半整数的分频。但是无论怎么调整，半整数分频的占空比不可能是 50%。半整数分频的方法有很多，我这里采用一种和奇数分频调整占空比类似的方法。

(1) 例如进行 3.5 倍分频时，计数器循环计数到 7，分别产生由 4 个和 3 个源时钟周期组成的 2 个分频时钟。从 7 个源时钟产生了 2 个分频时钟的角度来看，该过程完成了 3.5 倍的分频，但是每个分频时钟并不是严格的 3.5 倍分频。

(2) 下面对周期不均匀的分频时钟进行调整。一次循环计数中，在源时钟下降沿分别产生由 4 个和 3 个源时钟周期组成的 2 个分频时钟。相对于第一次产生的 2 个周期不均匀的时钟，本次产生的 2 个时钟相位一个延迟半个源时钟周期，一个提前半个源时钟周期。

(3) 将两次产生的时钟进行"或操作"，便可以得到周期均匀的 3.5 倍分频时钟。分频波形示意图如下所示。

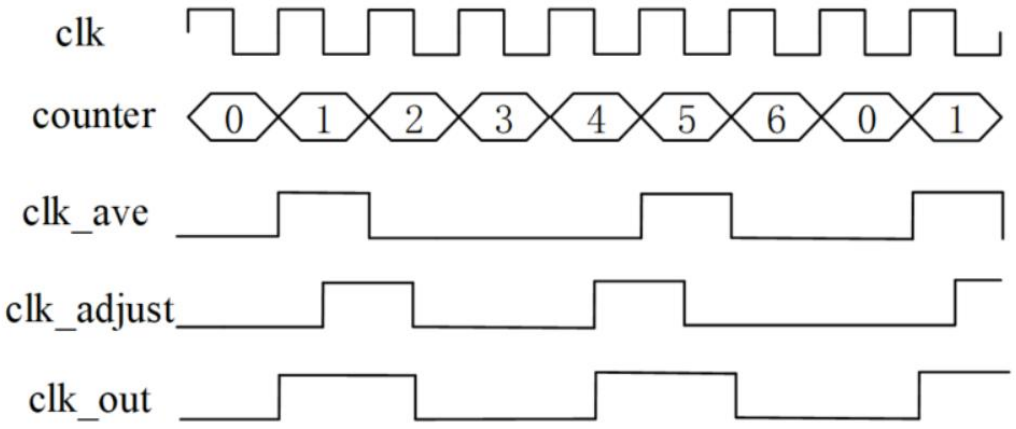


图 1 3.5 倍分频案例

值得注意的是，在示波器观察软件中，其不能检测到下降沿，所以在示波器的显示中会有一定的误差在里面，但是在仿真中其并没有错误，所以我们主要通过仿真来观测代码编写是否正确。我们将根据 3.5 倍分频案例去编写 9.5 分频的代码并进行验证。

三、实验设备与软件

实验设备：

①Altera Cyclone® IV EP4CE22F17C6N FPGA 开发板

②Quartus II 13.1 烧录软件

仿真软件：

①Quartus II 13.1 仿真软件（底层利用 Modelsim）

四、分频器实验程序框图

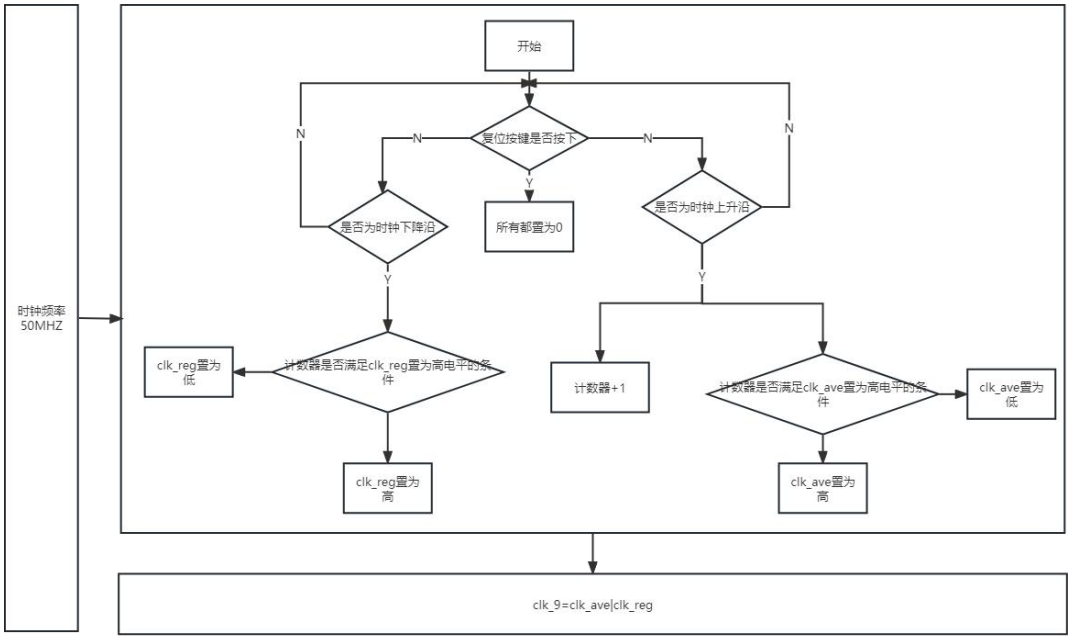


图 2 9.5 倍分频程序框图

五、实验内容与结果

（1）实验内容

1、设计一个 9.5 分频的分频器。

（2）实验步骤

1、建立工程

2、依照真值表编写符合要求的代码文件

3、添加文件到工程并编译文件

4、查看编译后的设计单元

5、将信号加入波形窗口

6、烧录进 FPGA 开发板并配置仿真文件

7、运行仿真并观察波形图是否符合实验要求

（3）实验结果

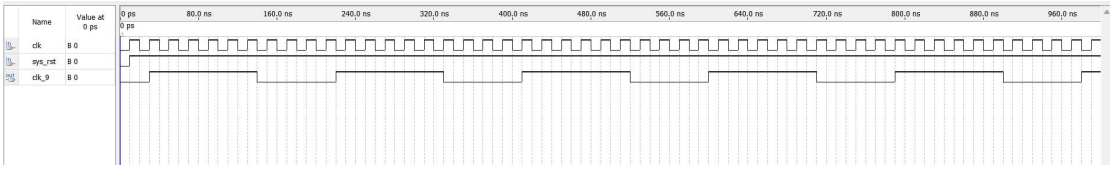


图 3 9.5 分频仿真波形图

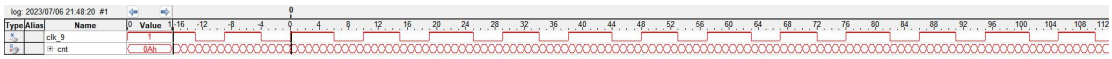


图 4 9.5 分频示波器观测的波形图

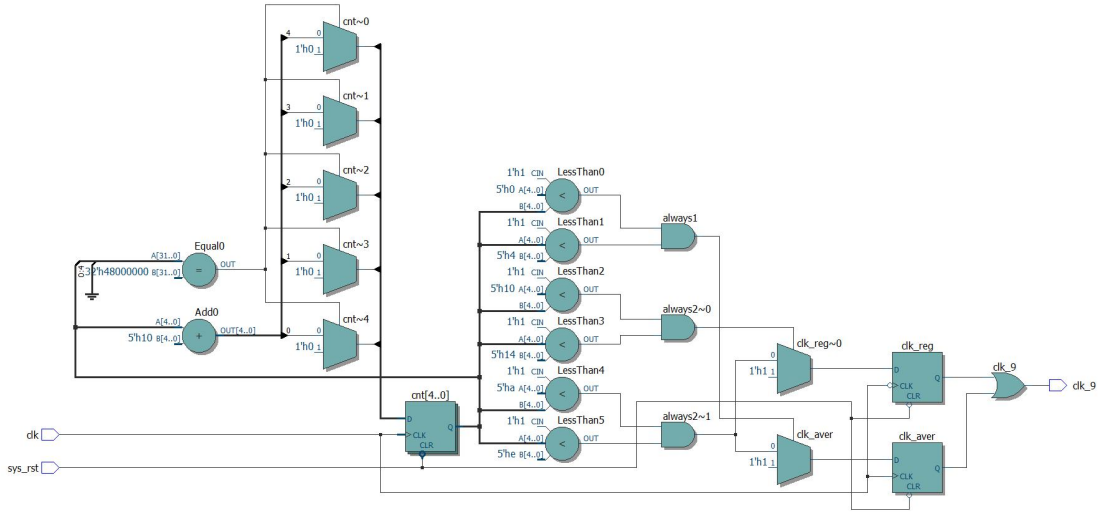


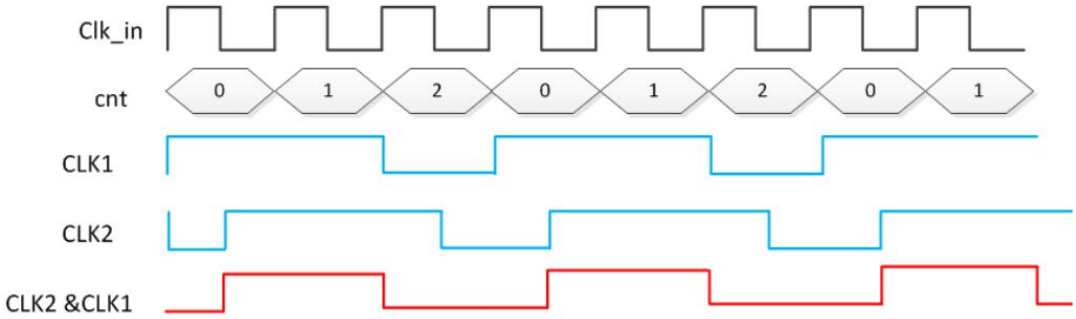
图 5 9.5 分频 RTL 视图

在这里我们能够发现由于示波器无法对下降沿进行检测所以会导致波形与仿真有一定差异。

六、实验讨论

1、我们通过对仿真波形进行观察能顾实际的观测到我们的代码结果与预期一致，完美的实现了 9.5 分频。

2、由于进行 9.5 分频前需要了解奇分频与偶分频的原理，所以我们需要提前对他们进行进一步的学习。其中奇分频原理如下：



3、通过这次的实验，对时钟分频，包括偶分频、奇分频、半整数分频都有了更深一步的认识并且也能够独自完成此类开发，加深了我对 FPGA 开发板整个时钟管理的理解。

七、实验代码

```
module split
(
      input wire clk,
      input wire sys_rst,
      output wire clk_9
);
parameter CNT_MAX=19,
                  hold=4;
reg clk_aver;
reg clk_reg;
reg [4:0]cnt=5'd0;
```

```
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        cnt<=5'd0;
    else if(cnt==CNT_MAX-1)
        cnt<=5'd0;
    else
        cnt<=cnt+1;
always@(posedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        clk_aver<=1'b0;
    else if((cnt>=5'd0)&(cnt<=hold))
        clk_aver<=1'b1;
    else if((cnt>=CNT_MAX/2+1)&(cnt<=CNT_MAX/2+1+hold))
        clk_aver<=1'b1;
    else
        clk_aver<=1'b0;
always@(negedge clk or negedge sys_rst)
    if(sys_rst==1'b0)
        clk_reg<=1'b0;
    else if((cnt>=5'd1)&(cnt<=hold+1))
        clk_reg<=1'b1;
    else if((cnt>=CNT_MAX/2+1)&(cnt<=CNT_MAX/2+1+hold))
        clk_reg<=1'b1;
    else
        clk_reg<=1'b0;
assign clk_9=clk_aver|clk_reg;
endmodule
```