数字逻辑与部件设计实验

实验报告

16307130194 陈中钰

16 级 计算机科学技术学院

座位号：30

指导老师：唐志强

目录

1 实验准备

2017 年 10 月 23 日 第七周 星期一

1.1 实验环境

- IDE: Xilinx Vivado 2015.4
- Language: Verilog HDL
- Operating System: Microsoft Windows 10 Enterprise
- Hardware: Digilent Nexys4 DDR board

1.2 Vivado 设计流程

- Create a Vivado Project using IDE
- Simulate the Design using Vivado Simulator
- Synthesize the Design
- Implement the Design
- Perform the Timing Simulation
- Verify Functionality in Hardware

1.3 Project Summary

- Default part: xc7a100tcsg324-1
- Project type: RTL Project
- Product category: General Purpose
- Family: Artix-7
- Speed grade: -1
- Package: csg324

1.4 注意事项

- project 应保存在纯英文路径下
- 变量定义必须与引脚文件对应
- module()括号中的最后一个变量后面没有逗号
- 引脚文件中的引脚必须与所用到的输入输出一一对应，不能多也不能少
- 实验前必须清楚每一个输入、输出是高有效的还是低有效的

1.5 其他

- 由于引脚锁定只需对应设置好即可，故本实验报告省略引脚锁定文件的代码展示

2 实验一：译码器和编码器

2017年10月30日 第八周 星期一

2.1 实验目的

- 通过实现计算机系统中最常用的逻辑部件之一的译码器，来熟悉实验操作、设计流程
- 了解并掌握自顶向下的设计流程，通过实现模块化，了解分层设计思想
- 了解译码器原理和作用，并设计、实现译码器，完成对操作码的译码

- 了解编码器的分类、原理和作用，并设计、实现译码器，完成对电平信号的编码
- 了解并掌握 de Morgan's Law

2.2 实验原理

2.2.1 74LS138：3-8译码器

- 输入：S[2:0]；输出：Y[7:0]
- 真值表

输入			输出							
S2	S1	S0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

- K-Map（以 Y0为例）

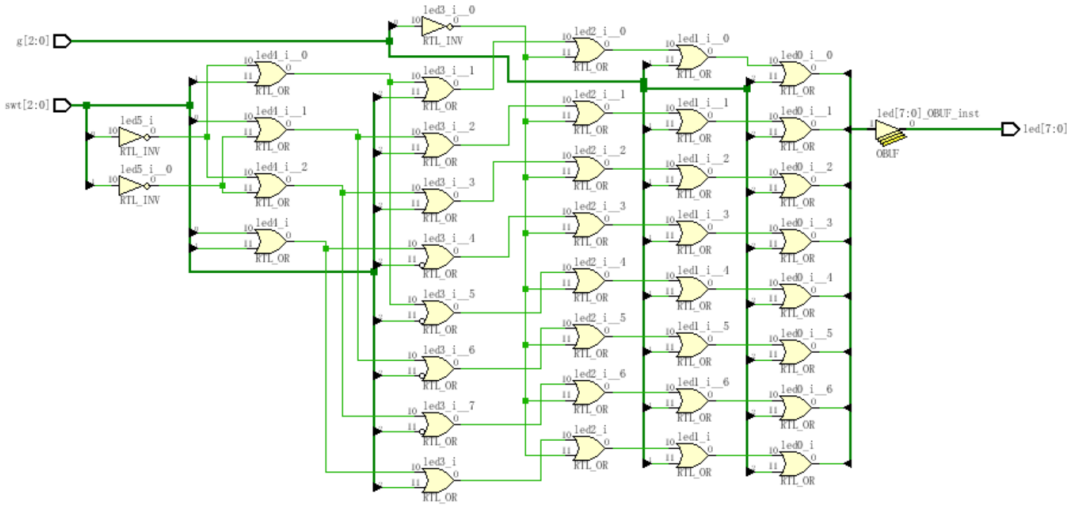
S0 \ S2S1	00	01	11	10
0		1	1	1
1	1	1	1	1

可得反相 Y0'=S0'S1'S2'，那么 Y0=S0+S1+S2

- bool 代数式

Y0=S0+S1+S2	Y1=S0'+S1+S2	Y2=S0+S1'+S2	Y3=S0'+S1'+S2
Y4=S0+S1+S2'	Y5=S0'+S1+S2'	Y6=S0+S1'+S2'	Y7=S0'+S1'+S2'

- 逻辑电路图（带有三个控制开关）



2.2.2 利用上述3-8译码器和与非门，设计一个4-16译码器

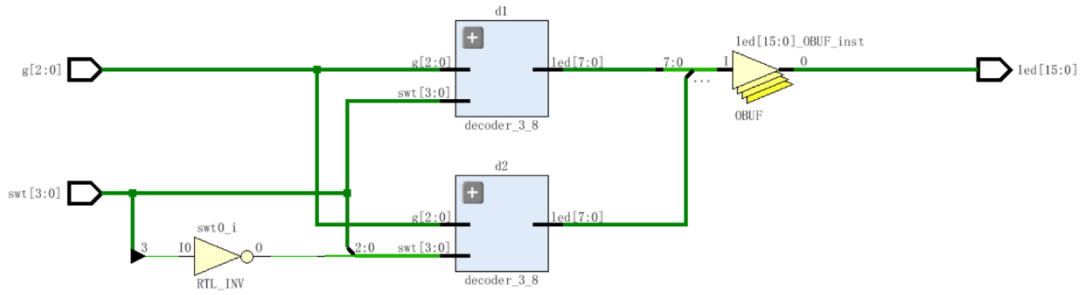
- 采用的是分层设计的思想
- 用3-8译码器构造4-16译码器和用2-4译码器构造3-8译码器原理类似，相比于3-8译码器，多一个 A[3]输入，共4个输入，若 A[3]为0，与3-8译码器的输出 D[7:0]

组合，可以有8种输出，当 A[3]为1时，还能组合出另外8种，一共16种的输出

- 真值表

输入 S				第二个3-8译码器的输出 Y								第一个3-8译码器的输出 Y							
S3	S2	S1	S0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	0	0	1								1	1	1	1	1	1	1	0
	0	0	1									1	1	1	1	1	1	0	1
	0	1	0									1	1	1	1	1	0	1	1
	0	1	1									1	1	1	1	0	1	1	1
	1	0	0									1	1	1	0	1	1	1	1
	1	0	1									1	1	0	1	1	1	1	1
	1	1	0									1	0	1	1	1	1	1	1
	1	1	1									0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	1	1	0	1							
	0	0	1	1	1	1	1	1	1	0	1								
	0	1	0	1	1	1	1	1	0	1	1								
	0	1	1	1	1	1	1	0	1	1	1								
	1	0	0	1	1	1	0	1	1	1	1								
	1	0	1	1	1	0	1	1	1	1	1								
	1	1	0	1	0	1	1	1	1	1	1								
	1	1	1	0	1	1	1	1	1	1	1								

- 逻辑电路图



2.2.3 8-3普通编码器

- 输入：I[7:0]；输出：F[2:0]
- 真值表

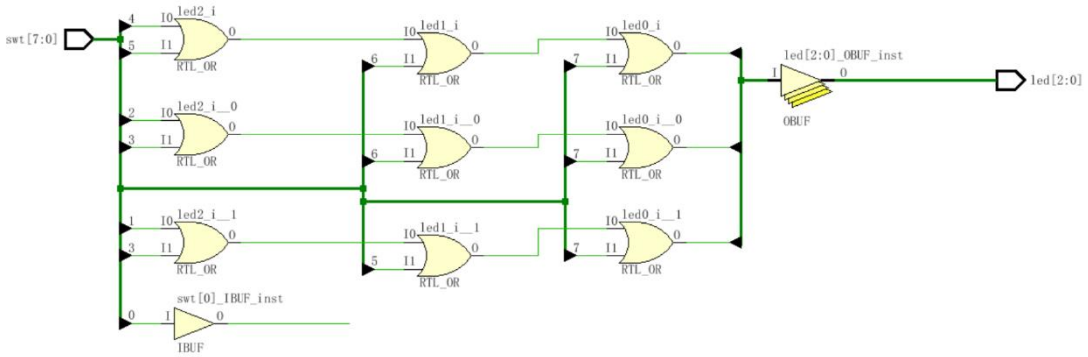
输入								输出		
I7	I6	I5	I4	I3	I2	I1	I0	F0	F1	F2
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0

1	0	0	0	0	0	0	0	1	1	1

- bool 代数式（可以直接观察得到）

F0=I4+I5+I6+I7	F1=I2+I3+I6+I7	F2=I1+I3+I5+I7

- 逻辑电路图

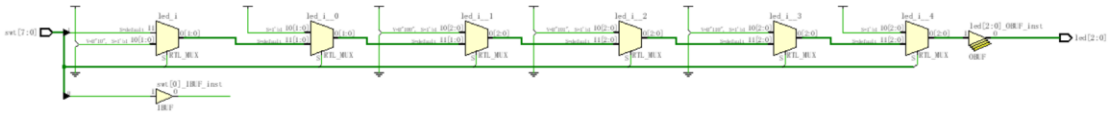


2.2.4 8-3优先编码器

- 输入：I[7:0]；输出：F[2:0]
- 真值表

输入								输出		
I7	I6	I5	I4	I3	I2	I1	I0	F0	F1	F2
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

- 通过 if else 结构可以很好地表达“优先”的效果，而不需要求出 bool 代数式，描述会更为简洁、易懂
- 逻辑电路图



2.3 实验内容

2.3.1 74LS138：3-8译码器

- 要求加入三个控制信号：G1，G2A，G2B，当 G1=1，G2A=G2B=0时实现正常编码，否则 Y[7:0]=7’b1111111。那么可以在原来的基础上给 Y 的每一位 OR (~G1+G2A+G2B)，当满足条件时这一部分为0，对原结果没影响，否则这一部分为1，Y 每一位结果都是1，为高电平
- 控制信号表

G1G2G2B	~G1+G2A+G2B	输出

100	0	不变
其他	1	全1

- Verilog 代码实现

```
module decoder_3_8(input [2:0] swt, input [2:0] g, output [7:0] led);
    assign led[0]=swt[0] | swt[1] | swt[2]| ~g[0] | g[1] | g[2];
    assign led[1]=~swt[0] | swt[1] | swt[2]| ~g[0] | g[1] | g[2];
    assign led[2]=swt[0] | ~swt[1] | swt[2]| ~g[0] | g[1] | g[2];
    assign led[3]=~swt[0] | ~swt[1] | swt[2]| ~g[0] | g[1] | g[2];
    assign led[4]=swt[0] | swt[1] | ~swt[2]| ~g[0] | g[1] | g[2];
    assign led[5]=~swt[0] | swt[1] | ~swt[2]| ~g[0] | g[1] | g[2];
    assign led[6]=swt[0] | ~swt[1] | ~swt[2]| ~g[0] | g[1] | g[2];
    assign led[7]=~swt[0] | ~swt[1] | ~swt[2]| ~g[0] | g[1] | g[2];
endmodule
```

2.3.2 利用上述3-8译码器和与非门，设计一个4-16译码器

- 要构造出4-16译码器，则需要2个3-8译码器
- 需要对上述3-8译码器进行改造，添加多一个 swt[3]的输入，作用与使能相似，输入两个3-8译码器的值是相反的，使其中一个正常工作，另一个则全亮，那么可以在原来的基础上给每一个输出 OR swt[3]，那么，swt[3]=1的3-8译码器则输出全亮，而另一个3-8译码器的 swt[3]=0，正常工作，那么所有的16个灯仅1个灯不亮，符合要求

```
module decoder_4_16(input [3:0] swt, input [2:0] g, output [15:0] led);
    decoder_3_8 d1( {swt[3],swt[2:0]}, g[2:0], led[7:0]);
    decoder_3_8 d2( {~swt[3],swt[2:0]}, g[2:0], led[15:8]);
endmodule

module decoder_3_8(input [3:0] swt, input [2:0] g, output [7:0] led);
    assign led[0]=swt[0] | swt[1] | swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[1]=~swt[0]|swt[1] | swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[2]=swt[0]|~swt[1] | swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[3]=~swt[0]|~swt[1] | swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[4]=swt[0]|swt[1] | ~swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[5]=~swt[0]|swt[1] | ~swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[6]=swt[0]|~swt[1] | ~swt[2]| ~g[0] | g[1] | g[2]| swt[3];
    assign led[7]=~swt[0]|~swt[1]| ~swt[2]| ~g[0] | g[1] | g[2]| swt[3];
endmodule
```

2.3.3 8-3普通编码器

```
module encoder_8_3(input [7:0] swt,output [2:0] led);
    assign led[0]=swt[4]|swt[5]|swt[6]|swt[7];
    assign led[1]=swt[2]|swt[3]|swt[6]|swt[7];
    assign led[2]=swt[1]|swt[3]|swt[5]|swt[7];
endmodule
```

2.3.4 8-3优先编码器

```
module priority_encoder_8_3(input [7:0] swt, output reg [2:0] led);
    always@(swt)
```

```
    begin
    if(swt[7]) led=3'b111; else if(swt[6]) led=3'b110; else if(swt[5])
led=3'b101;
    else if(swt[4]) led=3'b100; else if(swt[3]) led=3'b011; else
if(swt[2]) led=3'b010;
    else if(swt[1]) led=3'b001; else if(swt[0]) led=3'b000; else
led=3'b000;
    end
endmodule
```

2.4 实验结论

- 实现了带3个控制的3-8译码器
- 使用3-8译码器构造了4-16译码器
- 实现了8-3普通编码器
- 实现了8-3优先编码器

2.5 实验感想

- 能更熟练地使用 Vivado，对 Verilog 语言更加熟悉了
- 对译码器、编码器的功能和设计更加了解
- 了解了分层设计、模块化的思想，为之后的实验设计打下基础（后续实验会经常用到模块化）

3 实验二：七段显示译码器的设计

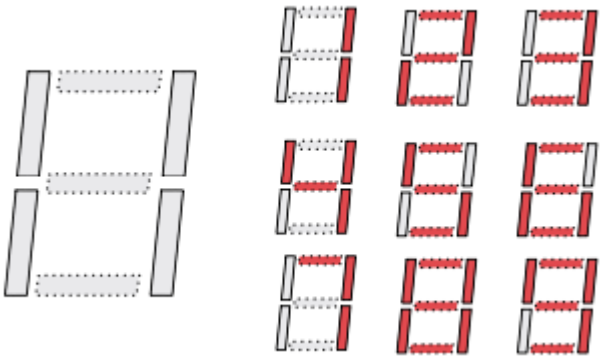
2017年11月6日 第九周 星期一

3.1 实验目的

- 熟练掌握 BCD 码
- 了解并设计 BCD 7段显示译码器，熟悉引脚与显示 LED 灯的对应关系
- 为后续对时序数字钟的设计奠定基础

3.2 实验原理

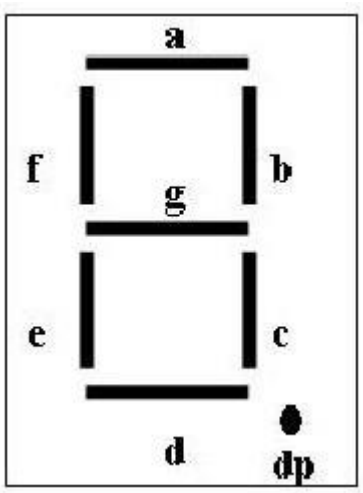
- BCD 码0000~1010与十进制数0~9一一对应；BCD 码0000~1111与十六进制数0~f 一一对应。其中十六进制0~9部分的显示与十进制的完全一致；之后的 a~f 为了能与数字区分开，采用 A，b，C，d，E，F 的输出方式



- BCD 7 segment display 对应关系：引脚文件中的 CA~CG 按顺序与下图中标识

a~g 的 LED 一一对应

- AN[7:0]：AN[0]~AN[7]分别与 board 上面从右往左的八个7段显示数字一一对应，当 AN[i]为0时，从右往左的第 i 个7端显示有效；而且八个7段显示数字均共用 CA~CG 的引脚（后续实验实现数字钟的时候要采用动态扫描的方式才能"同时"显示不同的数字）
- board 上7段显示的 AN 以及 CA~CG 输出都是0（低）有效的
- 真值表（输出为低有效）（这里的是显示1位十六进制数的版本，如果是只要显示1位十进制数，输入1010~1111的对应输出为XXXXXXX）



输入				输出							
D	C	B	A	a	b	c	d	e	f	g	字形
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	1	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0	2
0	0	1	1	0	0	0	0	1	1	0	3
0	1	0	0	1	0	0	1	1	0	0	4
0	1	0	1	0	1	0	0	1	0	0	5
0	1	1	0	0	1	0	0	0	0	0	6
0	1	1	1	0	0	0	1	1	1	1	7
1	0	0	0	0	0	0	0	0	0	0	8
1	0	0	1	0	0	0	0	1	0	0	9
1	0	1	0	0	0	0	1	0	0	0	A
1	0	1	1	1	1	0	0	0	0	0	b
1	1	0	0	0	1	1	0	0	0	1	C
1	1	0	1	1	0	0	0	0	1	0	d
1	1	1	0	0	1	1	0	0	0	0	E
1	1	1	1	0	1	1	1	0	0	0	F

3.3 实验内容

- 使用 case()语句，在输入和输出之间建立简单明了的对应关系，而不必像器件手册一样以基本门为基础来进行描述，这样可以大大简化代码，也更直观易懂、检验正确性
- Verilog 代码（以下是输出1位十六进制数的版本，如只需输出1位十进制数，那么 swt 为4'b1001~4'b1111都可以归为 default，而 default 可以设置为不显示）

```
module BCD_7_segment_display(input [3:0] swt, output reg [6:0] C,
output [7:0] AN);
    assign AN=8'b11111110;
    always@(swt)
    begin
    case(swt)
```

```
    4'b0000:C=7'b0000001;
    4'b0001:C=7'b1001111;
    4'b0010:C=7'b0010010;
    4'b0011:C=7'b0000110;
    4'b0100:C=7'b1001100;
    4'b0101:C=7'b0100100;
    4'b0110:C=7'b0100000;
    4'b0111:C=7'b0001111;
    4'b1000:C=7'b0000000;
    4'b1001:C=7'b0000100;
    4'b1010:C=7'b0001000; //hexadecimal part
    4'b1011:C=7'b1100000;
    4'b1100:C=7'b0110001;
    4'b1101:C=7'b1000010;
    4'b1110:C=7'b0110000;
    4'b1111:C=7'b0111000;
    //default:C=7'b1111111;
    endcase
    end
endmodule
```

3.4 实验结论

- 该 BCD 7段译码显示译码器可以成功显示1位十六进制数（十进制数）

3.5 实验感想

- 试验前需清楚输入与输出是高有效的还是低有效的
- 能更熟练使用 case()语句、always@()语句，为后续实验打下基础（后续实验经常用到这两种语句）

4 实验三：加法器及快速进位电路的设计

2017年11月13日 第十周 星期一

4.1 实验目的

- 了解并掌握半加器、全加器的设计和功能
- 了解并设计行波进位加法器、超前进位加法器
- 对比上述两者的延时，总体了解并对比两者的优缺点
- 加深对分层设计思想的了解，更加熟悉模块化的设计，理解模块化的优点

4.2 实验原理

4.2.1 1位全加器

- 输入：a，b，cin；输出：s，cout
- 真值表

输入	输出

a	b	cin	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

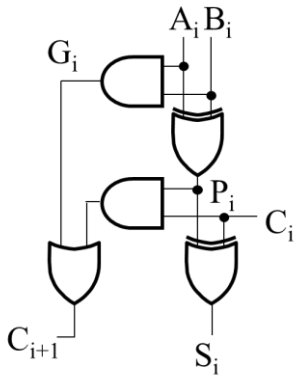
- K-Map（以 s 为例）

cin \ ab	00	01	11	10
0		1		1
1	1		1	

可得 s=a'b'cin+a'bcin'+ab'cin'+abcin=a^b^cin

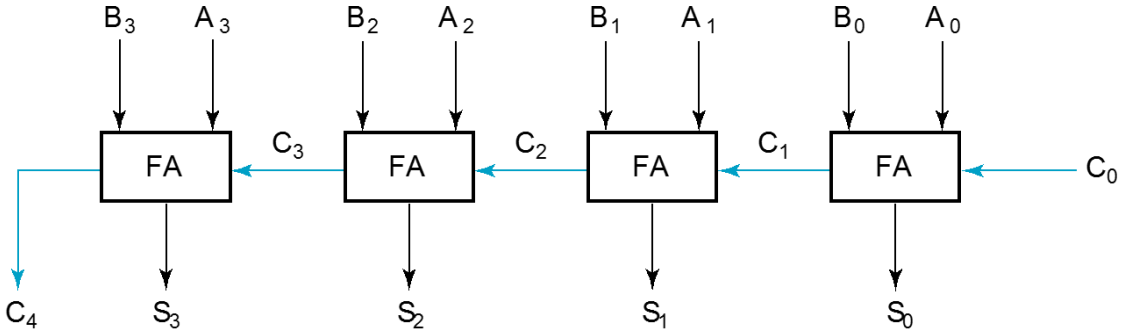
同理可得 cout=ab+acin+bcin

- 逻辑电路图（右图）。其中，Ai，Bi，Si 对应 a，b，s 而 Ci，Ci+1对应 cin，cout



4.2.2 以上述1位全加器为模块，设计4位行波进位加法器

- 输入 A[3:0]，B[3:0]，C[0]；输出 S[3:0]，C[4]
- 模拟实际的加法运算过程，从低位开始，用1位全加器求和，并进位到高位，再用1位全加器计算高位的和，直至计算完毕。那么可以使用4个全加器模块，串行设计，低位的进位输出为高位的进位输入，每一个全加器都只需要计算两个加数在这一位的数字A[i]、B[i]以及进位输入 C[i]的三者求和结果 S[i]，还有该位的进位输出 C[i+1]，那么，先从低位开始算，并一位接一位地算下去，直至计算完毕
- 逻辑电路图



4.2.3 4位超前进位加法器

- 输入 A[3:0]，B[3:0]，C[0]；输出 S[3:0]，C[4]
- 由于进位需要一位一位地从低位往高位传送，使得每一位的运算都是按从低位到高位的顺序进行的，耗时长，那么需要找到一种方法使得每一位都能同时计算，而要实现每一位的求和同时进行，则需要把每一位求和所需要的进位输入事先、并行计算出来，再同时计算出每一位的求和结果 s[i]
- 并行计算进位原理

先观察进位计算的规律

A	B	cin	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

那么，可以发现，AB=00时一定没有进位，AB=11时一定有进位，AB=01或10时，若同时有 cin 为1，则有进位，那么可以有 cout=AB+(A^B)cin，那么令

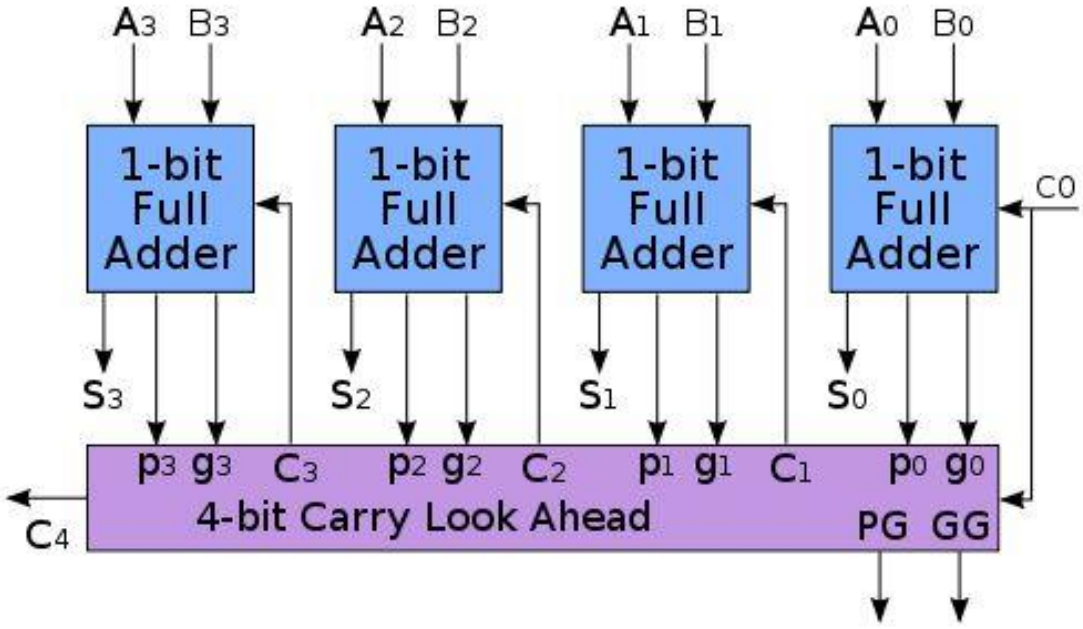
g[i]=A[i]B[i]	0<=i<=3
p[i]=A[i]^B[i]	
c[i+1]=g[i]+p[i]c[i]	
s[i]=A[i]^B[i]^c[i]=p[i]^c[i]	

那么进一步有

c[1]=g[0]+p[0]c[0]
c[2]=g[1]+p[1]c[1]=g[1]+p[i](g[0]+p[0]c[0])
c[3]=g[2]+p[2]c[2]= g[2]+p[2]g[1]+p[i](g[0]+p[0]c[0]))
c[4]=g[3]+p[3]c[3]= g[3]+p[3](g[2]+p[2]g[1]+p[i](g[0]+p[0]c[0])))

首先先计算出 g[3:0]和 p[3:0]的值，再据此计算出 c[4:1]（其中 c[0]为 cin，c[4]为 cout），那么每一位的求和运算都能直接取进位输入，能并行完成计算。

- 逻辑电路图



4.3 实验内容

4.3.1 1位全加器

- Verilog 代码

```
module full_adder_1(input a, input b, input cin, output s, output
cout);
    assign s=~a&~b&cin | ~a&b&~cin | a&~b&~cin | a&b&cin;
    assign cout=a&b | a&cin | b&cin;
endmodule
```

4.3.2 以上述1位全加器为模块，设计4位行波进位加法器

- Verilog 代码

```
module ripple_carry_adder_4(input [3:0] a, input [3:0] b, input cin,
output [3:0] s, output cout);
    wire [4:0] c;
    assign c[0]=cin;
    full_adder_1 bit0(a[0+:1],b[0+:1],c[0+:1],s[0+:1],c[1+:1]);
    full_adder_1 bit1(a[1+:1],b[1+:1],c[1+:1],s[1+:1],c[2+:1]);
    full_adder_1 bit2(a[2+:1],b[2+:1],c[2+:1],s[2+:1],c[3+:1]);
    full_adder_1 bit3(a[3+:1],b[3+:1],c[3+:1],s[3+:1],c[4+:1]);
    assign cout=c[4];
endmodule
```

4.3.3 4位超前进位加法器（其中 s[3:0]的计算按要求使用模块化来实现）

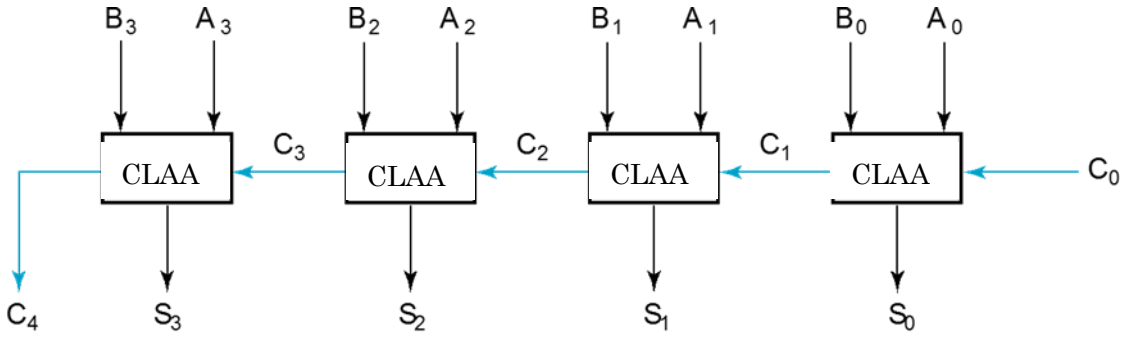
```
module carry_look_ahead_adder_4(input [3:0] a, input [3:0] b, input
cin, output [3:0] s, output cout);
    wire [3:0] g; wire [3:0] p; wire [4:0] c;
    assign g[3:0]=a[3:0]&b[3:0];
    assign p[3:0]=a[3:0]^b[3:0];
    assign c[0]=cin;
    assign c[1]=g[0]|p[0]&c[0];
    assign c[2]=g[1]|p[1]&g[0]|p[1]&p[0]&c[0];
    assign c[3]=g[2]|p[2]&g[1]|p[2]&p[1]&g[0]|p[2]&p[1]&p[0]&c[0];
    assign c[4]=g[3]|p[3]&g[2]|p[3]&p[2]&g[1]|p[3]&p[2]&p[1]&g[0]|
              p[3]&p[2]&p[1]&p[0]&c[0];
    assign cout=c[4];
    xor_2 x(p[3:0], c[3:0], s[3:0]);
endmodule

module xor_2(input [3:0] p, input [3:0] c, output [3:0] s);
    assign ans[3:0]=p[3:0]^c[3:0];
endmodule
```

4.4 实验结论

- 实现了1位全加器
- 以1位全加器为底层模块，设计了4位行波进位加法器、4位超前进位加法器
- 行波进位加法器需要从低位到高位一步步串行计算，而超前进位加法器则是并行计算，后者更快

4.5 实验感想

- 虽然超前进位加法器计算更快，但是额外使用了更多的逻辑门和线，成本会更高，而且当超前进位位数更多时，逻辑门个数呈指数增长，成本也随之快速增加，因此超前进位加法器的位数不能过多，一般是4位宽的。而虽然行波进位加法器计算较慢，但是成本较低。如果要实现更多位的加法，则需要超前进位加法器和行波进位加法器搭配使用（如下图中的16位加法器，每4位使用1个4位宽的超前进位加法器进行计算，而这4个超前进位加法器之间，用一个4位宽的行波进位加法器进行串行连接），既能提高速度，成本也不会增加太多。其中 CLAA 为超前进位加法器，A[i]、B[i]、S[i]均为宽度为4的向量，C[i]宽度为1



5 实验四：算术逻辑单元的设计

2017年11月20日 第十一周 星期一

5.1 实验目的

- 了解并设计 CPU 的核心部件算术逻辑单元 ALU
- 加深对减法运算以及补码的理解
- 熟练并准确设计控制信号
- 熟练加减法、与或非、异或运算

5.2 实验原理

- 功能控制表

S1	S0	M=0 逻辑运算	M=1 算术运算	
			Cn=0	Cn=1
0	0	F=not A	A+B	A+B+1
0	1	F=A and B	A-B	A-B-1
1	0	F=A or B		
1	1	F=A xor B		

- 逻辑运算：not，and，or，xor 分别对应~，&，|，^运算
- 加法运算：利用实验三实现的4位超前进位加法器，可以直接求 A+B+Cn
- 减法运算：

  A-B=A+(-B)=A+(~B+1)=A+~B+~Cn

  A-B-1=A+(-B)-1=A+(~B+1)-1=A+~B+~Cn

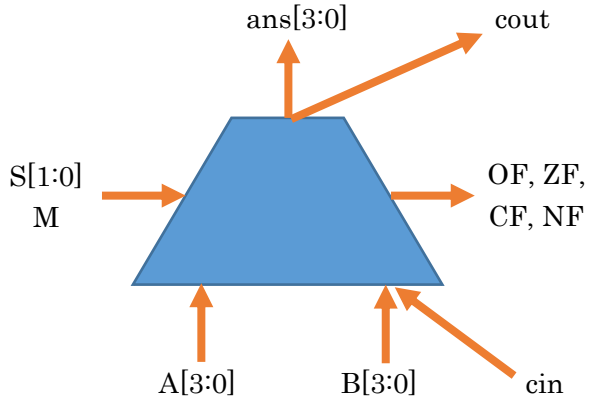
  那么同样可以利用上述超前加法器，只需把输入的 B，Cn 取反即可

  （注意：当 B 为0000时不可以取反，因为0没有补码！因此当 B 为0时需要特判）
- 条件码

条件码	意义	表达式
OF	溢出	cout^c[3]
ZF	ans 为0	~(ans[3]|ans[2]|ans[1]|ans[0])
CF	产生进位	cout
NF	ans 为负数	ans[3]

其中 ans[3:0]为计算结果，cout，c[3]均为进位

- 器件图



5.3 实验分析

- 由于 always@()语句中不能使用模块，因此可以在 always@()外面使用超前加法器模块，并把加法运算、减法运算的值、进位事先算出来，再在 always@()语句中判决最终的值、进位应该取哪一组结果
- 减法运算，在上述判断的时候，还要判断 B 是否为0，若 B 为0，则不能用超前进位加法器模块的计算结果，而需要重新计算

5.4 实验内容

5.4.1 Verilog 代码（加减法运算可以直接使用实验三的4位超前进位加法器的模块，故在此省略它的实现代码）

```
module ALU(input [3:0] a, input [3:0] b, input cin, input m, input
[1:0] s, output reg [3:0] ans, output reg cout);
    wire [3:0] ans1,ans2;
    wire cout1,cout2;
    carry_look_ahead_adder_4 plus1(a[3:0],b[3:0],cin,ans1[3:0],cout1);
    carry_look_ahead_adder_4 plus2(a[3:0],~b[3:0],~cin,ans2[3:0],cout2);
    always@(a,b,cin,m,s,ans1,ans2,cout1,cout2)
    begin
    if(m)
        begin
        case(s)
            2'b00: begin ans[3:0]=ans1[3:0]; cout=cout1; end
            2'b01:
            begin
            if(b==4'b0000) begin ans[3:0]=a[3:0]-cin; cout=0; end
```

```
            else begin ans[3:0]=ans2[3:0]; cout=cout2; end
            end
            default: begin ans[3:0]=4'b0000; cout=0; end
        endcase
        end
    else
        begin
        case(s)
            2'b00:ans[3:0]=~a[3:0];
            2'b01:ans[3:0]=a[3:0]&b[3:0];
            2'b10:ans[3:0]=a[3:0]|b[3:0];
            2'b11:ans[3:0]=a[3:0]^b[3:0];
        endcase
        cout=0;
        end
    end
endmodule
```

5.4.2 操作

- 当 M 为0时，可以实现4种逻辑运算，当 M 为1时，可以实现加减法运算

5.4.3 改进

- 添加 OF、ZF、CF、NF 共4个条件码
- 其中除了 OF 以外均可以直接 assign 赋值

assign ZF=~(ans[0]|ans[1]|ans[2]|ans[3]);
assign CF=cout;
assign NF=ans[3];

- 而 OF 要在超前进位加法器模块中计算，则需要修改实验三中的4位超前进位加法器，添加 OF 作为输出，且在应用该模块时也要加上 OF 作为输出，其中 OF 计算式为 assign OF=c[3]^cout;

```
module carry_look_ahead_adder_4(input [3:0] a, input [3:0] b, input
cin, output [3:0] s, output cout, output OF);
    wire [3:0] g; wire [3:0] p; wire [4:0] c;
    assign g[3:0]=a[3:0]&b[3:0]; assign p[3:0]=a[3:0]^b[3:0];
    assign c[0]=cin;
    assign c[1]=g[0]|p[0]&c[0];
    assign c[2]=g[1]|p[1]&g[0]|p[1]&p[0]&c[0];
    assign c[3]=g[2]|p[2]&g[1]|p[2]&p[1]&g[0]|p[2]&p[1]&p[0]&c[0];
    assign c[4]=g[3]|p[3]&g[2]|p[3]&p[2]&g[1]|p[3]&p[2]&p[1]&g[0]|
    p[3]&p[2]&p[1]&p[0]&c[0];
    assign cout=c[4];
    assign OF=c[3]^cout;
    xor_2 x(p[3:0], c[3:0], s[3:0]);
endmodule
```

5.5 实验结论

- 能够实现4位 ALU 的四种逻辑运算、加法减法运算，并能输出 OF、ZF、NF、CF 共四个条件码

5.6 实验感想

- 实验的时候，定义变量 cout 时不小心误写为 out，而且没有报错，结果导致算术运算都是错误的，而且完全不知道哪里有 bug，最后仔细查看一次又一次后才最终找出来并改正
- 大大加深了对补码运算的理解
- 加强了对电路控制的设计能力，当判断分支多了之后，一定要淡定，清楚每个分支的走向和执行条件，才能很好地把握整个电路

6 实验五：触发器和寄存器

2017年11月27日 第十二周 星期一

6.1 实验目的

- 了解并掌握时序电路的基本器件触发器的设计，包括 D 触发器、T 触发器、JK 触发器
- 了解并掌握时序电路中的异步清零、异步置数功能的实现，知道异步操作的作用和必要性
- 熟练掌握各种触发器的特征方程，能实现触发器之间的转换
- 熟悉触发器的功能和实现，能将触发器应用于功能器件的设计和实现
- 了解计数器的工作原理，并设计4位二进制计数器
- 加强对时序电路的理解，以及设计时序电路的能力

6.2 实验原理

6.2.1 异步操作

- always@()语句的敏感表中，除了有 posedge clk 以外，异步控制开关也应该是敏感项，即当异步开关电平值改变时，也应该触发
- 异步操作优先级高于同步操作，可以通过 if else 来控制
- 当异步置零为1时，q<=0；当异步置数为1时，q<=1
- always@()敏感表中不能同时有上升（下降）沿触发和电平触发，因此只能把电平触发也改成上升（下降）沿触发
- 异步操作控制表

reset	myset	功能
0	0	正常运行
0	1	异步置数
1	X	异步清零

6.2.2 D 触发器

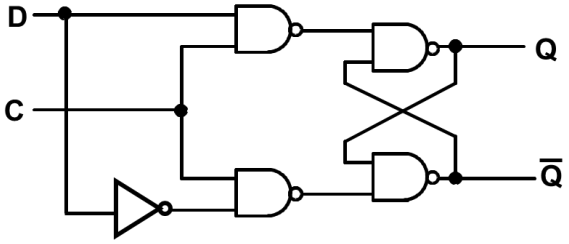
- 输入 D，状态 Q(n)，下一状态 Q(n+1)
- 状态表

D	Q(n)	Q(n+1)	操作
0	0	0	复位
0	1	0	

1	0	1	置位
1	1	1	

可得 Q=D

- 逻辑电路图



6.2.3 JK 触发器

- 输入 J，K，状态 Q(n)，下一状态 Q(n+1)
- 状态表

J	K	Q(n)	Q(n+1)	操作
0	0	0	0	无变化
0	0	1	1	
0	1	0	0	复位
0	1	1	0	
1	0	0	1	置位
1	0	1	1	
1	1	0	1	翻转
1	1	1	0	

可得，当 JK=00时保持 q<=q；为01时复位 q<=0；为10时置位 q<=1；为11时翻转 q<=~q

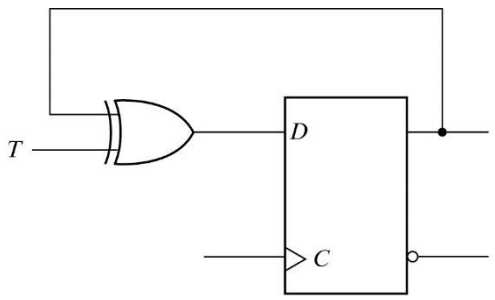
- 通过 case()语句控制，逻辑更明晰，更容易检验正误

6.2.4 T 触发器

- 输入 T，状态 Q(n)，下一状态 Q(n+1)
- 状态表

T	Q(n)	Q(n+1)	操作
0	0	0	无变化
0	1	1	
1	0	1	翻转
1	1	0	

- 电路图



6.2.5 D 触发器转换成 JK 触发器

- 综合 JK 触发器、D 触发器的状态表，列出包含 J，K，D 的状态表

J	K	Q(n)	Q(n+1)	D	操作

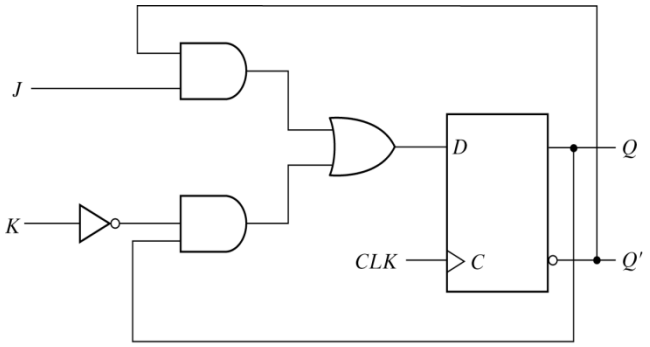
0	0	0	0	0	无变化
0	0	1	1	1	
0	1	0	0	0	复位
0	1	1	0	0	
1	0	0	1	1	置位
1	0	1	1	1	
1	1	0	1	1	翻转
1	1	1	0	0	

- 画 J、K、Q 的 K-Map，求出 D

Q \ JK	00	01	11	10
0			1	1
1	1			1

可得 D=JQ'+K'Q

- 那么将 D 触发器的输入 D 改为 JQ'+K'Q，就能构造出 JK 触发器
- 逻辑电路图

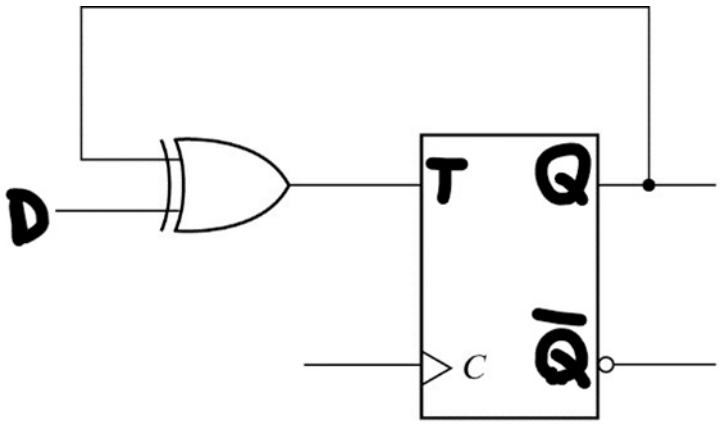


6.2.6 T 触发器转换成 D 触发器

- 综合 T 触发器、D 触发器的状态表，列出包含 T，D 的状态表

D	Q(n)	Q(n+1)	T	操作
0	0	0	0	复位
0	1	0	1	
1	0	1	1	置位
1	1	1	0	

- 观察可得 T= D^Q
- 那么将 T 触发器的输入 T 改为 D^Q，就能构造出 D 触发器
- 逻辑电路图

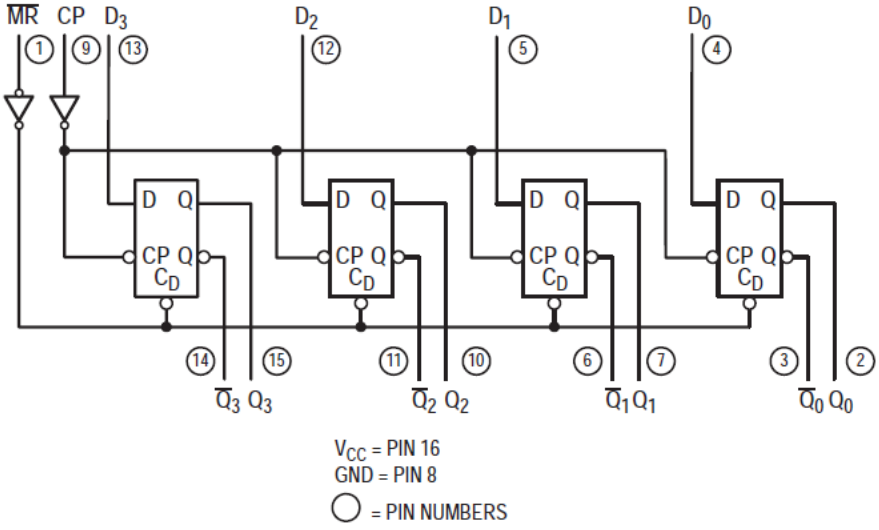


6.2.7 用 D 触发器构成74LS175

- 性质：4位宽的 D 触发器
- 输入 D[3:0]，当 reset 无效，而且时钟上升沿到达的时候，输出 Q[3:0]被输入更新为 D[3:0]的值
- 通过本实验的 D 触发器模块作为底层模块，各模块共用 CLK 和 reset 信号，可以构成该4位宽 D 触发器
- 其中 reset 信号为低有效（active low），故要在应用 D 触发器模块的时候，输入的 reset 信号要取反
- 状态表

reset	D[3:0]	Q[3:0]	Q[3:0](n+1)	操作
0	XXXX	XXXX	0000	异步清零
1	XXXX	XXXX	D[3:0]	同步置数

- 逻辑电路图



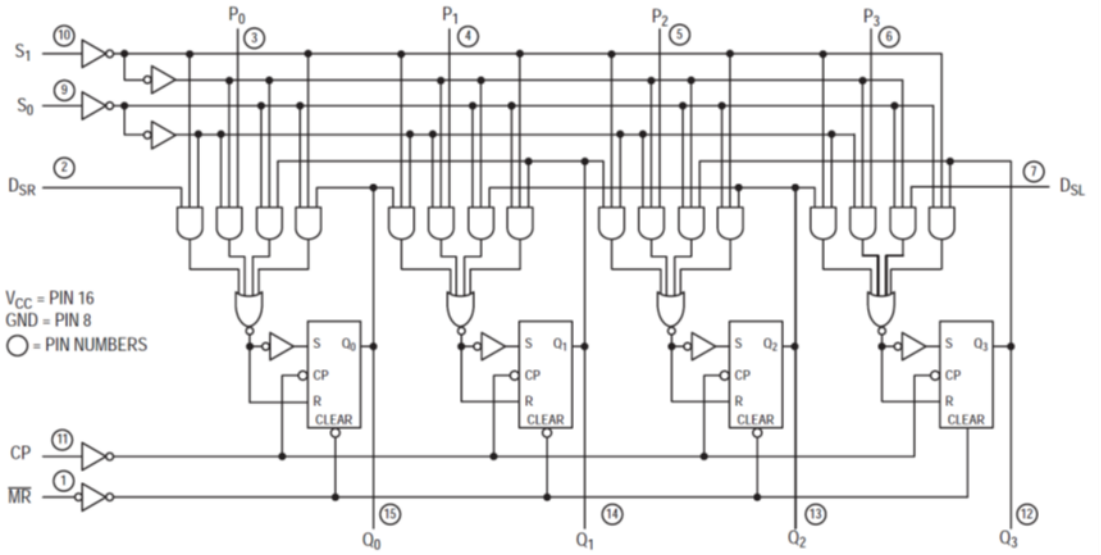
6.2.8 74LS194双向移位寄存器

- 具有异步清零、数据保持、左移右移、并行置数的功能
- 异步清零优先级最高，当 reset 有效时，输出设置为0000
- 左移时把右三位赋值给左三位，并在最右边补上 Dsl 的值；右移时把左三位赋值给右三位，并在最左边补上 Dsr 的值
- 并行置数时，同步置数，把 Q[3:0]置为输入的 P[3:0]

- 状态表

reset	S1	S2	工作状态
0	X	X	异步清零
1	0	0	数据保持
1	0	1	右移，补 Dsr 的值
1	1	0	左移，补 Dsl 的值
1	1	1	并行置数

- 逻辑电路图



6.2.9 74LS163 4位二进制计数器

- 状态表

clear	load	P	T	clk	Q	RCO	操作
0	X	X	X	↑	0000	0	清0
1	0	X	X	↑	DCBA	*	并行预设
1	1	1	1	↑	计数	*	加1计数
1	1	0	X	X	Q	*	保持
1	1	X	0	X	Q	0	保持

其中，RCO 保持为0，当且仅当 DCBA=4'b1110，且要加1、将产生进位时，RCO 为1

6.3 实验内容

6.3.1 时钟分频

- Vivado 预设的时钟品率为100MHz，太快了，需要分频以减慢变化的速度，减慢到约4Hz
- 分频机制：定义了一个28位长的计数器 q，每次都输出 q[24]的值，而从0开始，每次时钟上升沿时 q 增加1，当2^24次时钟周期后，此时 q=1000…（24个0），q[24]从0变为1，此时输出有效，而且 q[24]=1会保持接下来的2^24次时钟周期，直到 q=1000…（25个0），q[24]从1变回0，无效。也就是说输出的时钟信号的周期是输入信号的周期的2^24倍，实现了降低频率的分频效果
- 取 q[24]约为4Hz，增加值的大小会降低频率，反之提高
- 以下每个触发器都使用了此时钟分频的模块，故之后不再展示代码

- Verilog 代码

```
module clkdiv(input mclk, output clk1_4hz);
    reg [27:0]q;
always@(posedge mclk)
    q<=q+1;
assign clk1_4hz=q[24];
endmodule
```

6.3.2 带异步操作的 D 触发器

- 通过开关输入 reset，myset，d 信号；通过 LED 输出 q 信号
- reset 操作的优先级最高，其次是 myset，最后才是同步置数
- Verilog 代码

```
module D_flip_flop(input clk, input d, input myset, input reset, output
reg q);
    wire myclk;
    clkdiv c1(clk, myclk);
    always@(posedge myclk,posedge myset,posedge reset)
    begin
        if(reset) q<=0;
        else if(myset) q<=1; else q<=d;
    end
endmodule
```

- 操作：当 reset 有效时，q 被异步置零；当 reset 无效而 myset 有效时，q 被异步置1；均无效时，且时钟上升沿到达时，q 被同步置数为 d

6.3.3 带异步操作的 JK 触发器

- 通过开关输入 reset，myset，J，K 信号；通过 LED 输出 q 信号
- 优先级：reset > myset > JK
- Verilog 代码

```
module JK_flip_flop(input clk, input myset, input reset, input j, input
k, output reg q);
    wire myclk;
    clkdiv c1(clk, myclk);
    always@(posedge myclk, posedge myset, posedge reset)
    begin
        if(reset) q<=0;
        else if(myset) q<=1;
        else if(j&~k) q<=1;
        else if(k&~j) q<=0;
        else if(k&j) q<=~q;
        else q<=q;
    end
endmodule
```

- 操作：当 reset 有效时，q 被异步置零；当 reset 无效而 myset 有效时，q 被异步置1；均无效时，且时钟上升沿到达时，若 JK=00，则保持，若为01则置零，若为10时则置1，若为11时则取反

6.3.4 带异步操作的 T 触发器

- 通过开关输入 reset，myset，T 信号；通过 LED 输出 q 信号
- 优先级：reset > myset > T
- Verilog 代码

```
module T_flip_flop(input clk, input t, input myset, input reset, output
reg q);
    wire myclk;
    clkdiv c1(clk, myclk);
    always@(posedge myclk,posedge myset,posedge reset)
    begin
        if(reset) q<=0;
        else if(myset) q<=1;
        else if(t) q<=~q;
        else q<=q;
    end
endmodule
```

- 操作：当 reset 有效时，q 被异步置零；当 reset 无效而 myset 有效时，q 被异步置1；均无效时，且时钟上升沿到达时，若 T 为0，则 q 保持，若 T 为1，则 q 取反

6.3.5 D 触发器转换成 JK 触发器

- 通过开关输入 reset，myset，J，K 信号；通过 LED 输出 q 信号
- 按照特征方程 D=JQ'+K'Q，复用 D 触发器即可
- Verilog 代码

```
module JK_flip_flop(input clk, input myset, input reset, input j, input
k, output q);
    D_flip_flop d1(clk,j&~q|~k&q,myset,reset,q);
endmodule
```

- 操作与 JK 触发器一致

6.3.6 T 触发器转换成 D 触发器

- 通过开关输入 reset，myset，D 信号；通过 LED 输出 q 信号
- 按照特征方程 T=D^Q，复用 T 触发器即可
- Verilog 代码

```
module D_flip_flop(input clk, input d, input myset, input reset, output
q);
    T_flip_flop t1(clk,d&~q|~d&q,myset,reset,q);
endmodule
```

- 操作与 D 触发器一致

6.3.7 用 D 触发器构成74LS175

- 通过开关输入 reset，myset，d[3:0]；通过 LED 输出 q[3:0]，qn[3:0]
- 复用4个 D 触发器，并共用 CLK，reset 信号（低有效）
- Verilog 代码

```
module Quad_D_flip_flop(input clk, input [3:0] d, input reset, output
[3:0] q, output [3:0] qn);
    D_flip_flop d0(clk,d[0],0,~reset,q[0]);
```

```
    D_flip_flop d1(clk,d[1],0,~reset,q[1]);
    D_flip_flop d2(clk,d[2],0,~reset,q[2]);
    D_flip_flop d3(clk,d[3],0,~reset,q[3]);
    assign qn[0]=~q[0]; assign qn[1]=~q[1];
    assign qn[2]=~q[2]; assign qn[3]=~q[3];
endmodule
```

- 操作：当 reset 有效时，q 被异步置零；当 reset 无效而 myset 有效时，q 被异步置1；均无效时，且时钟上升沿到达时，q[3:0]被同步置数为 d[3:0]，并显示在 LED[3:0]上

6.3.8 74LS194 双向移位寄存器

- 输入 reset 异步清零信号，输入 a[1:0]控制信号，输入异步置数值 data[3:0]，输入左右移补上的值 d[1:0]，输出 q[3:0]，并在 LED[3:0]上显示
- Verilog 代码

```
module Shift_register(input clk, input reset, input [1:0] d, input
[1:0] a, input [3:0] data, output reg [3:0] q);
    wire myclk;
    clkdiv c1(clk,myclk);
    always@(posedge myclk,posedge reset)
    begin
        if(~reset) q[3:0]=4'b0000;
        else
            case(a)
            2'b00:q[3:0]=q[3:0];
            2'b01:q[3:0]={d[1],q[3:1]};
            2'b10:q[3:0]={q[2:0],d[0]};
            2'b11:q[3:0]=data[3:0];
            endcase
    end
endmodule
```

- 操作：当 reset 有效时，异步置零；当 reset 无效且时钟上升沿到达时，若 a 为00时，q 保持，a 为01时右移，且补上左输入数据 d[1]，a 为10时左移，且补上右输入数据 d[0]，a 为11时，q 同步置数为输入数据 data

6.3.9 74LS163 4位二进制计数器

- 时钟分频函数：为了结果更明显，特意把时钟分频调的更慢，把 assign clk1_4hz=q[24]；改为 assign clk1_4hz=q[26]；那么新频率约为原来的四分之一，即1Hz
- Verilog 代码

```
module binary_counter(input A, input B, input C, input D, input clear,
input load, input P, input T, input clk, output reg [3:0] Q, output reg
RCO);
    wire myclk;
    clkdiv a(clk,myclk);
    always@(posedge myclk)
    begin
```

```
        if(!clear) begin Q<=4'b0000; end
        else
            if(!load) begin Q<={D,C,B,A}; end
            else
            case({P,T})
                2'b11:
                    begin
                    if(Q==4'b1110) begin Q<=Q+1; RCO<=1; end
                    else begin Q<=Q+1; RCO<=0; end
                    end
                default:Q<=Q;
            endcase
    end
endmodule
```

- 操作：当 reset 为0时，Q 被同步清0；当 reset 为1时，若 PT 不为11，则 Q 保持，若 PT 为11，4位宽二进制计数器 Q 则进行加1计数，并在 Q 为1110并要进行加1计数，要产生进位时，RCO 为1，其他时候 RCO 为0

6.4 实验结论

- 完成了 D 触发器，JK 触发器，T 触发器的设计，并证明了触发器具有储存的功能
- D 触发器，JK 触发器，T 触发器之间可以通过特征方程进行转换，也就是通过加入若干门对触发器输入信号进行处理，可以获得另一个触发器
- 由多个 D 触发器可以构成一个多位宽的 D 触发器，也就是一个4位的寄存器
- 实现了74LS194双向移位寄存器，实现了数据的储存、左右移位、清0
- 实现了74LS63四位二进制计数器，实现了计数功能

6.5 实验感想

- 通过利用一些 Verilog 语言的优点，可以大大简化设计（如双向移位寄存器中，可以通过错位赋值直接实现移位操作，而不需死板按照器件手册中的逻辑电路图，一个逻辑门不差地实现，这样会十分复杂、繁琐、难以 debug 和理解）
- D 触发器是很多时序电路的基础，如寄存器、移位器等，需要深入学习理解这一个基本器件
- D 触发器，JK 触发器，T 触发器之间可以实现转换，说明它们的本质其实是一致的，只是由于微小的设计差异而导致有不同的功能

7 实验六：有限状态机

2017年12月4日 第十三周 星期一

7.1 实验目的

- 了解并掌握有限状态机的概念和设计
- 设计交通灯控制器，实现东南西北四个方向的红绿灯正常交替变化
- 加深对 BCD 7段显示译码器的理解，了解其实际应用

- 了解并掌握扫描显示技术，清楚其作用及优点
- 了解并设计显示时分的数字钟

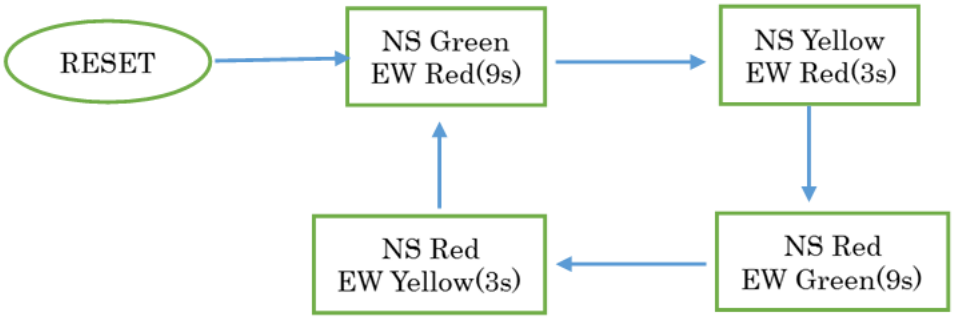
7.2 实验原理

7.2.1 交通灯控制器 1

- 使用3个 LED 灯分别代表一个路口的红、黄、绿灯，每种情况下仅只有一个灯亮（如：亮第一个灯代表红灯），四个路口用4组 LED 灯，共12个 LED 灯

C	B	A	状态	持续时间
0	0	1	Green	9s
0	1	0	Yellow	3s
1	0	0	Red	12s

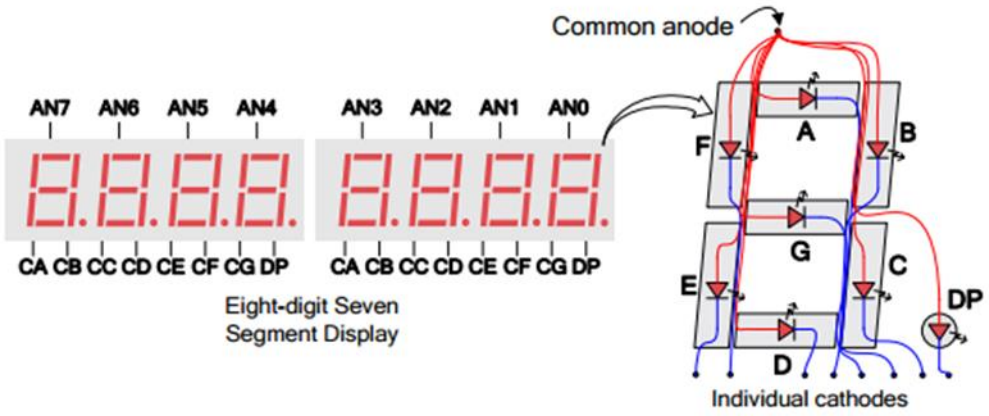
- 南北和东西向的在同一时间亮同样的灯，也就是状态相同
- 每个路口的灯的变化次序为：绿、黄、红，并循环变化，其中绿灯持续约9s，黄灯持续约3s，红灯持续约12s，可以通过使用一个6位二进制计数器进行计时，每个时钟周期增加1，当值为0~9为状态一，9~12为状态二，12~21为状态三，21~24为状态四，在状态交接处转换状态并正常计数，其他时候只计数，状态不变，而且在值为24时还同时把计数器置0
- 通过异步 RESET 开关可以设置状态为“NS 方向为绿灯”的开始
- 通过以上要求，可以得到状态图



7.2.2 显示时分的数字钟

- BCD 7段显示模块的 CA~CG、AN 都是0（低）有效的
- 显示时、分，时间范围00:00~23:59，每个数字分别用一个 BCD 码表示，也就是用1个 BCD 7段显示译码器（和实验二中的一致）来显示，一共4*4=16位二进制
- 输入时钟约1Hz，通过时钟分频获得
- 由于 board 上的8个 BCD 7段显示器是共用 CA~CG 引脚的，故需要采用扫描显示的方式进行多位数字的显示
- 扫描显示：利用人的0.1s~0.4s 视觉残留，每次只显示一位数字，并在一个周期里轮流显示4个数字，频率控制合适，能同时、稳定看到4位数字，在本次实验中采用约3kHz 的频率（如果频率过高，数字交替过快，每个数字显示的时间过短，视觉感官来不及感受灯光，而只能看见部分数字，而且在快速闪烁，不稳定，或甚至全部数字都看不到；如果频率过低，数字交替过慢，不足以形成视觉残留，会看见数字轮流点亮，而不是“同时亮”）
- 扫描显示实现：独立计算数字钟的四个数字，轮流给 CA~CG 赋值，并在对应时段使 AN 中该数字对应位置值为0，点亮该数字对应位置的7段显示器

- 数字计算的实现：使用约1Hz 的时钟，每过一个周期使时钟计数器增加1.但是与一般计数器不同的是，分的低位、时的低位为10进制，分的高位为6进制，时的高位只有0、1、2，因此要注意对进位情况的判断及赋值
- 轮流赋值的实现：使用约3kHz 的时钟，使用2位计数器的原理，使2位计数器从00~11不断循环计数，在对应时刻给 CA~CG 赋上对应数字的值，并使对应 AN 对应位置为0，其他 AN 为1，那么可以在这一个时钟周期在相应位置显示出数字钟的1位，在合适的频率下不断轮流显示后，便能“同时”看到4位数字显示
- 器件图



7.3 实验分析

7.3.1 交通灯控制器 1

- 通过计数器计数，用 case()语句判断是哪一种状态转换，以及是否要转换状态或只是单纯计数
- 通过 RESET 可以异步设置为初始状态，应该把该控制开关纳入 always@()的敏感表中，并可以通过 if 语句控制，而且该异步操作优先级最高
- 时钟分频取用的是 q[26]，约为 1Hz

7.3.2 显示时分的数字钟

- 使用到了实验二中的 BCD 7段显示译码器，用于把数字钟的4个数字显示出来，而由于真值表相同，此处不再展示
- 由于时间计数的特殊性，时间需要用特殊的计数器进行计数，每位数字用4位 BCD 码计数，并设置特殊的进位机制

优先级	当前时间	下一时间
high	23:59	00:00
	X9:59	(X+1)0:00
	XX:59	X(X+1):00
	XX:X9	XX:(X+1)0
low	XX:XX	XX:X(X+1)

- 添加异步置0功能（优先级最高）：当 reset 开关为1时，把4位 BCD 数字均置0000，且 reset 也在计数器部分的 always@()的敏感表中，且优先级最高
- 添加异步置数功能（优先级仅次于异步置0）：使用 myset 开关控制是否加载输入的时间，用 XX:XX <- sw[13:12] sw[11:8]:sw[7:4] sw[3:0]代表时间，由于时的高位只有0、1、2三个数，故只需要2个开关即可，一共需要2+4+4+4=14个开关

（同时可以剩下2个开关，一个作为重置开关 reset，另一个作为置数开关 myset），而为了使这个数字的7段显示也能套用 BCD 7段显示译码器的模块，可以把数字输入向量补为{0, 0, sw[13], sw[12]}，而另外的三个数字则可以正常使用 BCD 段显示模块，当 myset 为1时，会把 sw[13:0]输入的时间加载到 BCD 7段显示译码器的显示上，并且从该时间开始继续计数

- 添加 DP 闪烁：设置时钟频率也是约为1Hz，在每次时钟上升沿来临时，对 DP 的值取反，那么就能看到闪烁的效果了，每次亮的时间约为1s
- 控制开关真值表

Reset	myset	功能
0	0	正常计数
0	1	异步置数
1	X	异步置0

7.4 实验内容

7.4.1 交通灯控制器 1

7.4.1.1 Verilog 代码

```
module traffic_lights_1(input clk, input reset, output reg [2:0] N,
output reg [2:0] S, output reg [2:0] E, output reg [2:0] W);
    wire myclk;
    reg [5:0] counter;
    clkdiv c1(clk,myclk);
    always@(posedge myclk, posedge reset)
    begin
    if(reset)
    begin N<=3'b001; S<=3'b001; E<=3'b100; W<=3'b100; end
    else
    begin
        case(counter)
        6'b001001:
    begin N<=3'b010; S<=3'b010; E<=3'b100; W<=3'b100;
          counter<=counter+1; end
        6'b001100:
    begin N<=3'b100; S<=3'b100; E<=3'b001; W<=3'b001;
          counter<=counter+1; end
         6'b010101:
    begin N<=3'b100; S<=3'b100; E<=3'b010; W<=3'b010;
          counter<=counter+1; end
         6'b011000:
    begin N<=3'b001; S<=3'b001; E<=3'b100; W<=3'b100;
          counter<=6'b000000; end
        default
    begin N<=N; S<=S; E<=E; W<=W; counter<=counter+1; end
        endcase
    end
```

```
    end
endmodule

module clkdiv(input inclk, output outclk);
    reg [35:0] q;
    always@(posedge inclk)
         q<=q+1;
    assign outclk=q[26];//around 0.75Hz
endmodule
```

7.4.1.2 操作

- 当 reset 为0时，LED 按照状态图中状态不断循环显示
- 当 reset 设为1时，LED 马上变为 NS 为 Green，EW 为 Red 的状态开始

7.4.1.3 改进

- 计数器最大计数为24（11000），故只需要5位，而不需要6位
- 使用 parameter 定义3种不同的状态，每种状态代表一种灯（如：设置 parameter yellow=3'b010，代表黄灯），可以避免直接的向量赋值，而且更加直观易懂
- 设置 state 指示4种状态（0，1，2，3），通过计数器设置 state，再通过 state 设置状态转换和显示，这样虽然在计数器和显示之间插入了 state，但是会使逻辑结构更清晰，而且可以把计数部分与状态转换部分的代码分别分开到2个 always@() 语句中，功能更独立明确，一个负责计数，另一个负责状态转换，代码会更规整而简洁易懂

7.4.1.4 改进结果（省略 clkdiv 模块）

```
module traffic_lights_1(input clk, input reset, output reg [2:0] N,
output reg [2:0] S, output reg [2:0] E, output reg [2:0] W);
    wire myclk;
    reg [5:0] counter; reg [2:0] state;
    clkdiv c1(clk,myclk);
    parameter yellow=3'b010; parameter green=3'b001; parameter
red=3'b100;
    always@(posedge myclk, posedge reset)
    begin
        if(reset)begin counter<=5'b00000;state<=0;end
        else
        begin counter<=counter+1;
            if(counter==5'b01001) state<=1;
            else if(counter==5'b01100) state<=2;
            else if(counter==5'b10101) state<=3;
            else if(counter==5'b11000) begin
                  state<=0;counter<=5'b00000;end
        end
    end
    always@(state)
    case(state)
```

```
    0: begin N<=green; S<=green; E<=red; W<=red; end
    1: begin N<=yellow; S<=yellow; E<=red; W<=red; end
    2: begin N<=red; S<=red; E<=green; W<=green; end
    3: begin N<=red; S<=red; E<=yellow; W<=yellow; end
    endcase
endmodule
```

7.4.2 显示时分的数字钟

7.4.2.1 设计

- 使用 BCD 7段显示译码器模块，显示4位数字
- 用分频后约1Hz 的时钟，在该4位数字上实行时钟式计数，可以通过 if else 语句实现
- 扫描显示单个数字，使用分频后约3kHz 的时钟
- 一般来说，数字钟的时和分之间会有一个冒号在闪烁，因此原本想设置中间的冒号进行闪烁。但是引脚文件中没有找到，只找到数字右下角的 DP 点，所以最后改为设置 DP 点在闪烁，使用约为1Hz 的时钟

7.4.2.2 Verilog 代码

```
module digital_clock(input clk, input [13:0] swt, input reset, input
myset, output reg [6:0] C, output reg DP, output reg [7:0] AN);
    wire [6:0] C1; wire [6:0] C2; wire [6:0] C3; wire [6:0] C4;
    reg [3:0] T1; reg [3:0] T2; reg [3:0] T3; reg [3:0] T4; reg [1:0]
cnt;
    wire myclk1,myclk2;
    clkdiv1 c1(clk,myclk1);//time
    clkdiv2 c2(clk,myclk2);//display
    always@(posedge myclk1)//beep
        DP<=~DP;
    always@(posedge myclk1, posedge reset, posedge myset)//set time
    begin
    if(reset)
    begin T4<=4'b0000; T3<=4'b0000; T2<=4'b0000; T1<=4'b0000; end
    else if(myset)
    begin T4<={0,0,swt[13],swt[12]}; T3<=swt[11:8]; T2<=swt[7:4];
T1<=swt[3:0];
    end
    else if(T4==4'b0010&T3==4'b0011&T2==4'b0101&T1==4'b1001)//23:59
    begin T4<=4'b0000; T3<=4'b0000; T2<=4'b0000; T1<=4'b0000; end
    else if(T3==4'b1001&T2==4'b0101&T1==4'b1001)//x9:59
    begin T4<=T4+1; T3<=4'b0000; T2<=4'b0000; T1<=4'b0000; end
    else if(T2==4'b0101&T1==4'b1001)//xx:59
    begin T4<=T4; T3<=T3+1; T2<=4'b0000; T1<=4'b0000; end
    else if(T1==4'b1001)//xx:x9
    begin T4<=T4; T3<=T3; T2<=T2+1; T1<=4'b0000; end
    else//xx:xx
```

```
    begin T4<=T4; T3<=T3; T2<=T2; T1<=T1+1; end
    end
    BCD_7_segment_display b1(T1[3:0],C1[6:0]);
    BCD_7_segment_display b2(T2[3:0],C2[6:0]);
    BCD_7_segment_display b3(T3[3:0],C3[6:0]);
    BCD_7_segment_display b4(T4[3:0],C4[6:0]);
    always@(posedge myclk2)
    begin
        case(cnt)
        2'b00: begin AN<=8'b11111110; C<=C1; cnt<=cnt+1; end
        2'b01: begin AN<=8'b11111101; C<=C2; cnt<=cnt+1; end
        2'b10: begin AN<=8'b11111011; C<=C3; cnt<=cnt+1; end
        2'b11: begin AN<=8'b11110111; C<=C4; cnt<=cnt+1; end
        endcase
    end
endmodule

module BCD_7_segment_display(input [3:0] swt, output reg [6:0] C);
    always@(swt)
    begin
    case(swt)
    4'b0000:C=7'b0000001;
    4'b0001:C=7'b1001111;
    4'b0010:C=7'b0010010;
    4'b0011:C=7'b0000110;
    4'b0100:C=7'b1001100;
    4'b0101:C=7'b0100100;
    4'b0110:C=7'b0100000;
    4'b0111:C=7'b0001111;
    4'b1000:C=7'b0000000;
    4'b1001:C=7'b0000100;
    default:C=7'b0000001;//consider it 0
    endcase
    end
endmodule

module clkdiv1(input inclk, output outclk);
    reg [35:0]q;
    always@(posedge inclk)
         q<=q+1;
    assign outclk=q[26];//around 0.75Hz
endmodule

module clkdiv2(input inclk,output outclk);
```

```
    reg [35:0]q;
    always@(posedge inclk)
         q<=q+1;
    assign outclk=q[16];//around 3kHz
endmodule
```

7.4.2.3 操作

- 当 myset，reset 均为0时，数字钟从00:00~23:59，正常变化，并跳回00:00
- 当 reset 为1时，数字钟马上变回00:00，并从这个时间开始计时
- 当 reset 为0，myset 为1时，数字钟设计为 sw[13:0]的输入时间，并从这个时间开始计时

7.4.2.4 改进

- 由于该时钟有异步置数的功能，因此输入的数字可能非法。BCD 7段显示译码器模块中已经有一个安全机制，当时的低位和分的低位输入值超过9时，会设置为0。但是，时的高位为3，或者分的高位在5~9，此时非法，但是没有安全设置，应该补全这个安全机制，使得非法时间输入统一改为某个合法时间输入，以避免时钟异步跳进一个非法时间进行计时，而导致之后的奇怪显示

	时的高位	时的低位	分的高位	分的低位
开关输入	>2	>9	>5	>9
安全设置	0	0	0	0

7.5 实验结论

- reset 为0时，交通信号等能正常按照状态图变化；reset 为1时重置
- 数字钟能正常计时；reset 为1时重置，myset 为1且 reset 不为1时，置数

7.6 实验感想

- 针对有限状态机，可以使用 parameter 对状态进行定义，并加入 state，next_state 等设置，会简化 case()或 if else 判断过程，使得代码更简洁明了，而不必要像非时序电路那样单纯依靠判断、赋值来进行设计，反而会更繁琐
- 有限状态机在交通信号灯、自动售卖机等均有所应用，因此了解并掌握有限状态机的设计及应用是十分重要的，也会是将来进一步学习时序电路的重要基础

8 实验七：总线实验

2017年12月11日 第十四周 星期一

8.1 实验目的

- 了解并掌握寄存器的原理和设计，加深对时序电路的理解
- 掌握 RAM 的储存功能的实现机制
- 能熟练设计、运用多于1位宽度的多路选择器
- 能理解实验的数据通路，加深模块化的理解，并能熟练进行电路微操作

8.2 实验原理

8.2.1 74LS377寄存器：带使能的4位宽 D 触发器

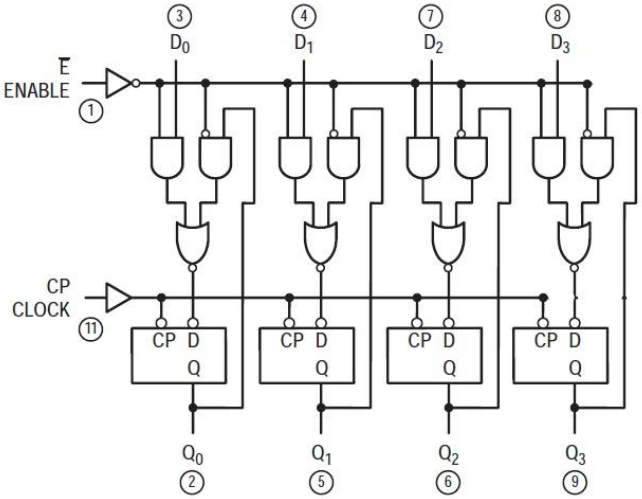
- 输入：输入信号 D[3:0]，开关输入使能信号 EN，时钟信号 CLK；输出：输出信

号 Q[3:0]，并在对应的 LED 上显示

- 状态表

EN	D[i]	Q[i](n)	Q[i](n+1)	操作
0	0	0	0	无变化
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	置数
1	0	1	0	
1	1	0	1	
1	1	1	1	

- 具体实现：当 EN=0时，Q 不变；当 EN=1时，Q<=D 置数
- 逻辑电路图



8.2.2 RAM（4位）

- 输入：时钟信号 clk，控制写入信号 wen，地址 addr[3:0]，输入数据 din[3:0]；输出：输出数据 qout[3:0]
- 存储机制：模块中定义了 reg[3:0]ram[0:15]，也就是16个 reg 型的 ram[3:0]变量，这些变量从0~15编号，而这些编号也就是地址，每个地址对应一个 ram[3:0]变量，也就是对应一个存储在 RAM 中的值。通过 SW[4:1]输入二进制地址 addr[3:0]，值的范围为0000~1111，也就是0~15，恰好与编号一一对应，故 ram[addr]就是存储在 addr 的4位二进制数
- 写入：当 wen 控制信号为1，且时钟上升沿到达时，会把 din 中的4位二进制数值写入 ram[addr]，为同步过程，可以通过 always@()语句实现
- 读取：异步向 qout 输出 ram[addr]的值，并在对应 LED 上显示

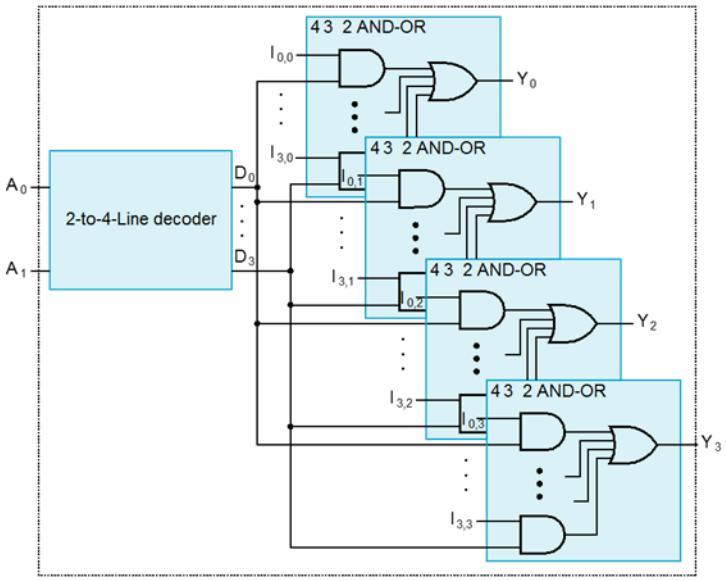
8.2.3 4位宽4-1 MUX

- 输入：选择信号 s[1:0]，被选择的4个4位宽向量 i0[3:0]，i1[3:0]，i2[3:0]，i3[3:0]；输出选择出的长度为4的向量 y[3:0]
- s[1:0]=00，01，10，11时分别对应选择数据通路中该器件的 a，b，c，d 输入
- 可以通过 case()语句，对 s[1:0]的值进行判断，把对应输入向量赋值给输出向量
- 由于 MUX 是异步的，因此所有输入值都在 always@()语句的敏感表中，可以用* 代替

- 选择表格

控制		输出
s[1]	s[0]	y
0	0	i0 / a
0	1	i1 / b
1	0	i2 / c
1	1	i3 / d

- 逻辑电路图



8.2.4 4位宽2-1 MUX

- 输入：选择信号 s，被选择的2个4位宽向量 i0[3:0]，i1[3:0]；输出被选择的长度为4的向量 y[3:0]
- s=0，1时分别对应选择数据通路中的该器件的 a，b 输入
- 可以通过 case()语句，对 s 的值进行判断，把对应输入向量赋值给输出向量
- 由于 MUX 是异步的，因此所有输入值都在 always@()语句的敏感表中，可以用* 代替
- 选择表格

控制 s	输出 y
0	i0 / a
1	i1 / b

8.2.5 时钟分频模块

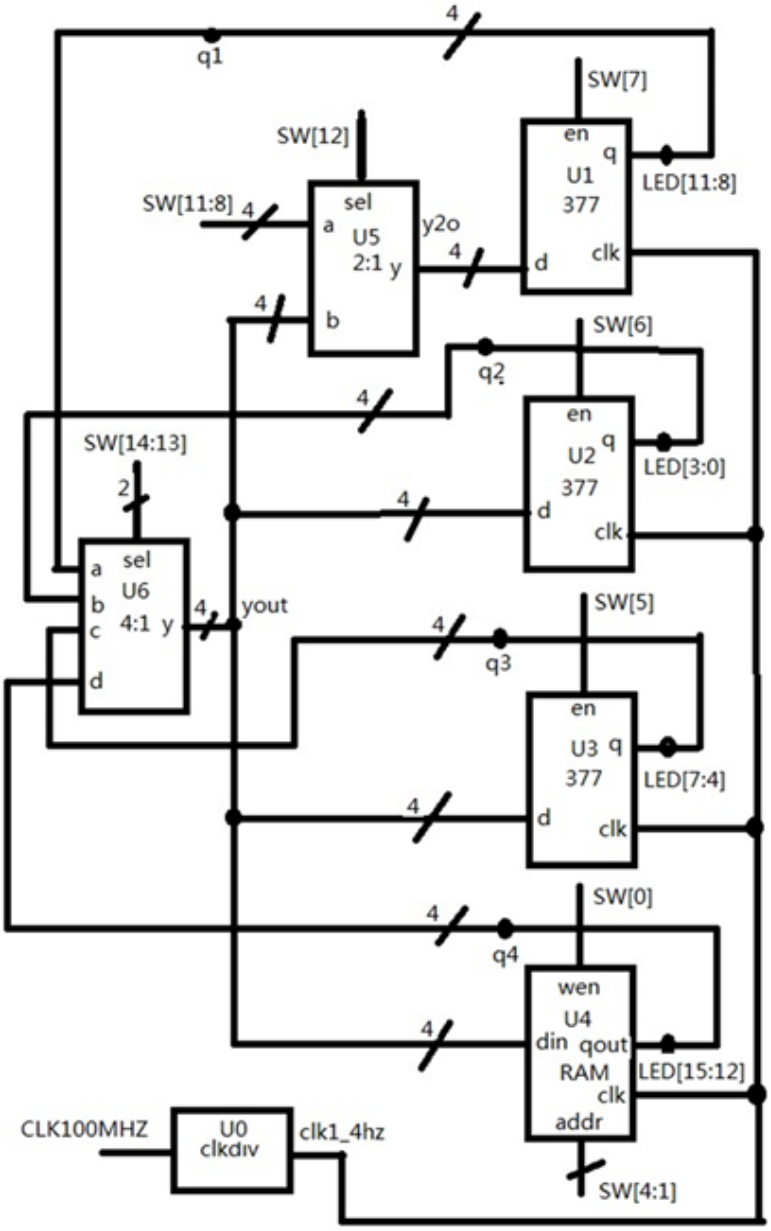
- 取的是 q[24]的值，输出时钟信号约为4Hz

8.3 实验分析

- U0为时钟分频模块；U1，U2，U3为3个4位宽寄存器；U4为 RAM；U5为4位宽的2-1MUX；U6为4位宽的4-1MUX
- U0把时钟频率从100MHz 降低到约4Hz，并控制 U1~U4
- 2-1多路复用器 U5的 SW[11:8]是唯一的数据输入处，通过调 SW[11:8]输入4位二

进制数，通过调节 SW[12]控制 MUX 选择 a 或 b 输入，并传送给 U1寄存器

- 通过 SW[7]使能控制 U1，当 SW[7]=1时，U1的值被输入值更新，并输出到 LED[11:8]以及4-1MUX 的输入端 a
- 通过调节 U6的 SW[14:13]可控制 MUX 选择 a，b，c，d 输入中的一个，并传送给2-1MUX 的 b 输入、U2、U3、U4的输入端
- 当 U2的 SW[6]=1时 U2被触发，U2储存的值被4-1MUX 传来的值更新，并输出到 LED[3:0]以及4-1MUX 的输入端 b
- 当 U3的 SW[5]=1时 U3被触发，U3储存的值被4-1MUX 传来的值更新，并输出到 LED[7:4]以及4-1MUX 的输入端 c
- 当 U4的 SW[0]=1时 U4的写入被触发，U4中储存在地址 SW[4:1]处的值被4-1MUX 输出值更新，并输出到 LED[15:12]以及4-1MUX 的 d 输入端；其他时候，不断输出地址 SW[4:1]处储存的值到4-1MUX 的 d 输入端，并在 LED[15:12]显示
- 逻辑电路图



8.4 实验内容

8.4.1 设计思想

- 独立实现各个模块
- 把各个模块的输入、输出连接起来

8.4.2 Verilog 代码

```
module bus(input clk, input [14:0] SW, output [15:0] LED);
    wire myclk;
    wire [3:0] d1;
    wire [3:0] y;
    wire [15:0] light;
    clkdiv U0(clk,myclk);
    register U1(myclk,SW[7],d1[3:0],light[11:8]);
    assign LED[11:8]=light[11:8];
    register U2(myclk,SW[6],y[3:0],light[3:0]);
    assign LED[3:0]=light[3:0];
    register U3(myclk,SW[5],y[3:0],light[7:4]);
    assign LED[7:4]=light[7:4];
    ram2 U4(myclk,SW[0],SW[4:1],y[3:0],light[15:12]);
    assign LED[15:12]=light[15:12];
    MUX_2_1 U5(SW[12],SW[11:8],y[3:0],d1[3:0]);
    MUX_4_1
U6(SW[14:13],light[11:8],light[3:0],light[7:4],light[15:12],y[3:0]);
endmodule

module register(input myclk, input en, input [3:0] d, output reg [3:0]
q);
    always@(posedge myclk,posedge en)
    begin
        if(en)
            q<=d;
    end
endmodule

module ram2(input clk, input wen, input [3:0] addr, input [3:0] din,
output [3:0] qout);
    reg[3:0]ram[0:15];
    always@(posedge clk)
      if(wen)
        ram[addr]<=din;
      assign qout=ram[addr];
endmodule

module clkdiv(input mclk, output clk1_4hz);
    reg [27:0]q;
```

```
    always@(posedge mclk)
         q<=q+1;
    assign clk1_4hz=q[24];
endmodule

module MUX_4_1(input [1:0] s, input [3:0] i0, input [3:0] i1, input
[3:0] i2, input [3:0] i3, output reg [3:0] y);
    always@(*)
    begin
        case(s)
        2'b00:y<=i0;
        2'b01:y<=i1;
        2'b10:y<=i2;
        2'b11:y<=i3;
        endcase
    end
endmodule

module MUX_2_1(input s, input [3:0] i0, input [3:0] i1, output reg
[3:0] y);
    always@(*)
    begin
        case(s)
        1'b0:y<=i0;
        1'b1:y<=i1;
        endcase
    end
endmodule
```

8.4.3 改进

- 不必使用 light[15:0]来作为 LED[15:0]的中间量，在模块中，LED[15:0]信号可以直接作为输入、输出信号，代码会更简洁

8.4.4 实验操作（输入1111，0000两个值，分别存在 U2，U3寄存器，并利用 RAM 为中介进行值的交换）

- SW[14:0]=15'b000000000000000置零
- SW[11:8]=4'b1111，而2-1MUX 的选择控制 SW[12]=0，选择输出的是 a，也就是开关 SW[11:8]输入的值1111，输出到 U1的输入口，使 SW[7]=1触发 U1，LED[11:8]全亮，表示1111存入 U1，SW[7]=0关闭 U1
- 此时 SW[14:13]==2'b00，选择输出的是 a，也就是 U1的输出值1111，并输出到 U2、U3、U4的输入口，使 SW[6]=1触发 U2，LED[3:0]全亮，表示1111存入 U2，关闭 SW[6]
- SW[11:8]=4'b0000，而2-1MUX 的选择控制 SW[12]=0，选择输出的是 a，也就是开关 SW[11:8]输入的值0000，输出到 U1的输入口，SW[7]=1触发 U1，LED[11:8]全暗，表示0000存入 U1，SW[7]=0关闭 U1
- 此时 SW[14:13]==2'b00，选择输出的是 a，也就是 U1的输出值1111，并输出到

U2、U3、U4的输入口，使 SW[5]=1触发 U3，LED[7:4]仍然是暗的，表示0000存入了 U3，关闭 SW[5]

- SW[14:13]=2'b01，使4-1MUX 选择来自 U2输出的 b 输入1111，输出到 y，也就是输出到 U2、U3、U4的输入口
- SW[0]=1触发 U4，而 SW[4:1]为0000，此时把输入值1111写入地址0000处，并输出到 LED[15:12]，全亮，并输出到4-1MUX 的 d 输入端
- SW[14:13]=2'b10，选择来自 U3输出的 c 输入0000，并输出到 y，也就是输出到 U2、U3、U4的输入口，使 SW[6]=1，触发 U2，把0000写入 U2，LED[3:0]全灭，表示0000存入 U2，SW[6]=0关闭 U2
- SW[14:13]=2'b11，选择来自 U4在地址0000储存的值的输出1111，并输出到 y，也就是输出到 U2、U3、U4的输入口，使 SW[5]=1触发 U3，把1111写入 U3，LED[7:4]全亮，表示1111存入 U3，SW[5]=0关闭 U3
- 操作完成

8.5 实验结论

- 实现了把1111写入 U2，0000写入 U3，并利用 RAM 为中介，交换 U2、U3中的值
- 寄存器具有储存、转移的功能
- RAM 具有储存多个数据的功能

8.6 实验感想

- 最初疏忽没有把4-1MUX 的 y 输出和 U2、U3的 d 输入连接起来，导致 SW 改变时 U2、U3完全没有反应，后来通过仔细检查才发现并更正
- 虽然4位宽的 D 触发器可以使用4个1位宽 D 触发器以及若干与门、或门，通过分层设计、模块化的思想来实现，但是这样做反而会复杂化，更不直观，且使得 debug 更困难；4-1MUX、2-1MUX 也是如此。因此需要利用 Verilog 语言的便捷之处，以简化问题以及代码设计，以避免粗暴的硬件翻译，也更直观，便于 debug
- 要能熟练进行微操作，则必须要十分清楚各个器件的作用，了解各个输入、输出，以及整个数据通路的关系，并明晰操作流程和效果
- 在较大电路的设计中，模块化具有很强的优势，能使得整个电路虽然庞大但是不乱

9 实验感想

2017年12月15日 第十四周 星期五

这7周以来，这7次实验，学会了很多很多，学会了使用 Vivado，学会了 Verilog HDL，学会了使用板子，学会了在高压下找 bug……同时，也对计算机硬件领域有了更深地了解，也能明显感觉到在数字逻辑与部件设计课程上也更容易理解课程知识了，而且也对计算机系统基础等课程的学习有很大的帮助。

这7周以来，虽然每次都提前在周末花上大半天去写代码，每次都 generate bitstream 之后，信心满满地去上实验课，结果每次都是一接上板子就有问题了。虽然每

次都是一些小细节的问题，但是却是会对实验结果有很大影响的大问题！可以说，不仅要有提前预习的认真态度，而且要有一颗仔细的心才行啊！

相信这些实验、这门课程会对将来的体系结构实验学习、计算机系统基础学习打下良好的基础，也相信会对将来科研、工作带来莫大的帮助！