一、设计课题的任务要求

一、基本要求

1. 在实验板上的DAC(TLV5638)的DA_A和DA_B分别输出频率和相位满足图4-1的正弦信号，将两路信号分别接到示波器的两个通道，则可以在示波器上(X-Y显示模式)观测到图4-1所示的李萨如图形。

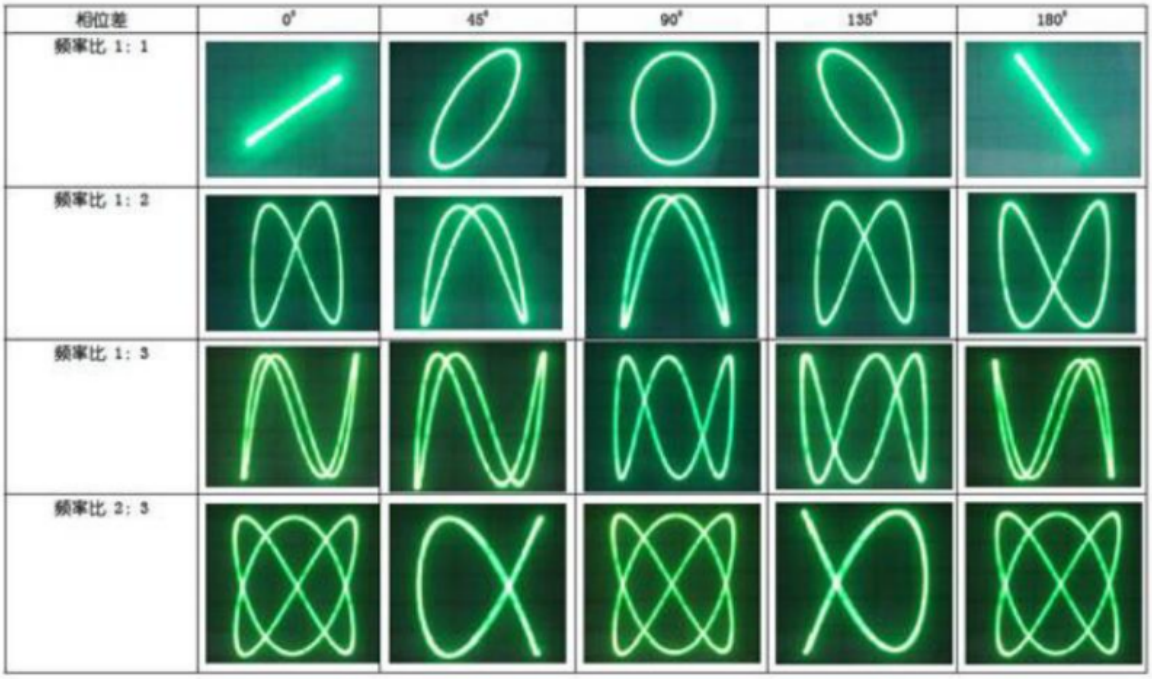


图 4-1 不同相位差和频率比的信号对应的李萨如图形

2. 输出信号的峰峰值≥1V，起始频率≥200Hz（即频率比中1对应的频率），波形每个周期大于100个点，波形无明显失真；

3. 用SW7和SW6选择频率比，“00”时为1:1，“01”时为1:2，“10”时为1:3，“11”时为2:3；并在数码管DISP上显示频率比；
4. 用SW2、SW1和SW0选择相位差，“000”时为0°，“001”时为45°，“010”时为90°，“011”时为135°，“100”时为180°;并在数码管DISP2、DISP1和DISP0上显示相位差。

二、提高要求

1. 可通过4*4键盘任意设置频率比和相位差；
2. 自拟其他功能。

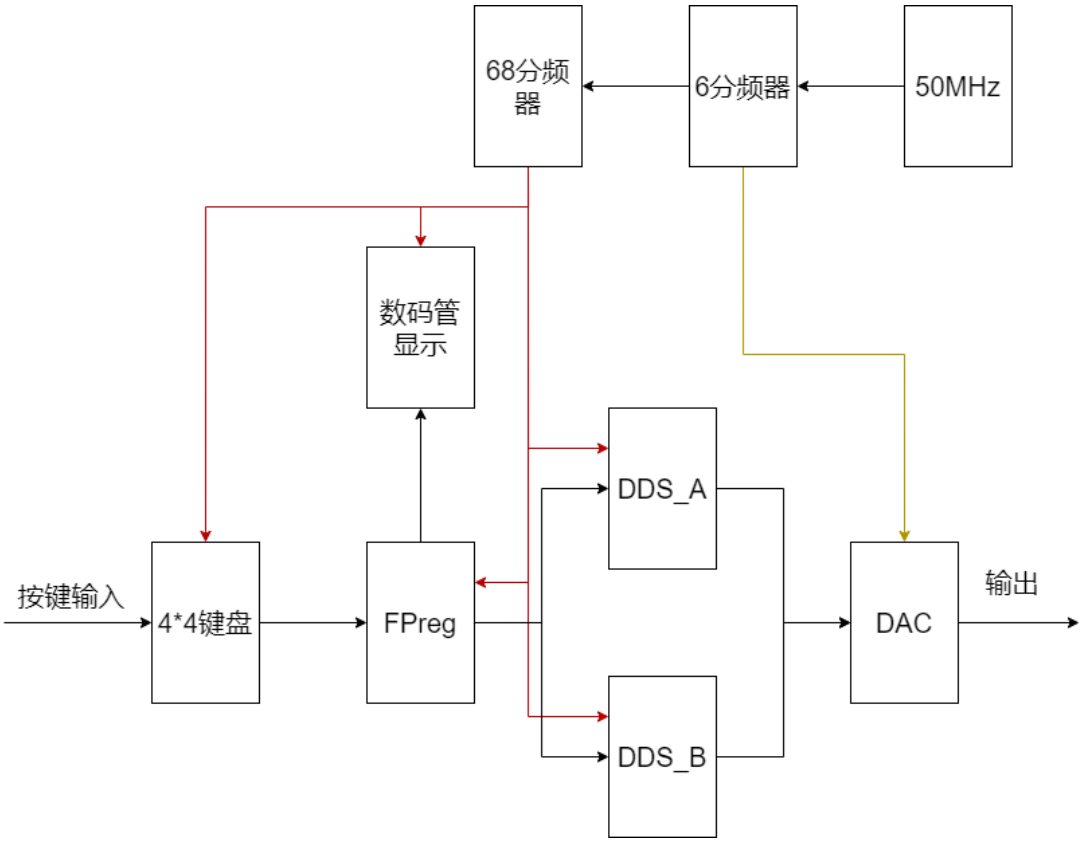
本设计实现了基本功能的1、2，并实现了提高功能，可利用4*4键盘任意设置频率比和相位差。

二、系统设计

一、设计思路

本设计实现了基于DDS的李萨如图形生成器。首先，本设计使用的正弦波波形是利用matlab生成的512点波形，然后以`case`语句的形式存储在rom中，构成正弦ROM查找表。本设计中频率与相位的概念利用频率字和相位字实现。频率字的含义为相位累加器在一个时钟周期内增加的点数，相位字的含义为波形输出从第几点开始。比如，当频率字为2，相位字为127时，波形的初始相位为$\frac{\pi}{2}$,频率为DDS模块所接入的时钟频率的两倍。相位累加器中寄存器的值与频率字相加就得到了当前相位字，也即rom的地址，将相位调制器中的地址输入rom即可得到应输出的波形，再将波形输出D/A即可得到对应的电信号输出。李萨图图形生成器实例化了两个上述的DDS，并通过一个`FPreg`（存储两个DDS相位字和频率字的寄存器）实现同步化，以产生符合要求的李萨图图形，此`FPreg`可通过矩阵键盘修改。

二、总体框图



三、分块设计

1. `Lissahous`模块

> 本模块是整个设计的顶层模块，本模块中实例化了两个分频器，对50MHz的系统时钟进行了6分频和6*68分频，DAC模块使用6分频后的时钟，其他模块均是6*68分频；例化了两个DDS模块，用FPreg同时输出两组频率字和相位字控制两个DDS模块，同时将这两组数据输出到数码管模块进行显示，DDS输出的数据输入到DAC转化为相应电信号显示在示波器上；FPreg中的相位字和频率字可以通过4*4键盘修改。

2. `rom`模块

> 本模块使用`case`语句存储了512点正弦波的波形信息。
>
> 为节约资源，本模块只存储了前256点，后256点通过对称的数学变化得到。

3. `DDS`模块

> 本模块由相位字和频率字的寄存器、相位累加器、相位调制器、与一个实例化的rom组成，由FPreg提供的频率字和相位字进入本模块后被寄存器存储。相位累加器在每个时钟周期增加频率字的值，相位累加器的值加上相位字即最终的地址，将地址输入rom后即可得到相应的正弦波幅值输出。

4. `4*4键盘`模块

> 本模块的功能是根据按下的按键输出键值与一个持续一时钟周期的脉冲。输出10000代表没有按键被按下，输出0****代表有按键被按下且键值为后四位二进制数所代表的值的大小。
>
> 本模块基于一个三进程状态机实现。
>
> 具体实现原理为：
>
> 当状态为idle，列信号输出0000，检测此时行信号是否为1111。
>
> - 如果是，则证明没有任何按键被按下，继续保持idle。
> - 如果不是，则有按键被按下，进入下一状态check_1st。
>
> 当状态为check_1st，列信号输出0111。
>
> - 如果此时行信号为1111，则证明被按下的按键不在第一列，进入下一状态check_2nd；
> - 如果行信号不是1111，则证明被按下的按键在第一列，此时根据行值判断按下的按键的键值即可。
>
> 后面的状态以此类推。
>
> 显然状态机的状态只有在按键按下状态变化时才会变化，当按键持续被按下时状态机不会改变，所以可以根据状态机状态判断键值，非常的方便。
>
> 本模块内有两个寄存器存储上一时钟周期的键值与当前键值，当键值改变时会输出一个持续一个时钟周期的脉冲信号来触发`FPreg`模块进行频率字和相位字的修改。

5. `FPreg`模块

> 本模块是整个设计运行的**核心模块**之一，是实现任意频率字和相位字的关键。
>
> 本模块内有四个寄存器，分别存储两路DDS的频率字和相位字。
>
> 因为相位差是相对概念，所以固定第一路DDS的相位字恒为0，只改变第二路DDS的相位字即可实现任意相位差。

> 频率字最高为8位，是正弦函数波形点数（9位，512点）的一半，因为根据奈奎斯特采样定理，一个正弦波至少得有两个采样点才能包含全部信息。
>
> 第一路DDS的频率字最高可到63， 因为原本的计划是在八位数码管上分别十进制显示第一二路DDS的频率字与第二路DDS的相位字，这样如果第一路DDS的频率字超过一百的话，就需要显示三个三位十进制数，需要九个数码管，资源显然是不够用的，所以把第一路DDS的频率字限制为6位，但后来发现十进制译码器过于复杂，消耗的资源也较多，所以改成了16进制限制，这样是可以显示8位频率字的，但如此之高的频率字必然会伴随极度失真，所以也未再做修改，即保留了第一路DDS的频率字的大小。
>
> 第二路DDS的频率字为8位。
>
> 本模块接受来自`4*4键盘`模块的键值与脉冲，当脉冲到来时，模块会根据当前的键值做出相应操作。具体为当按下第一行的第1、2、3、4个按键时，分别对应第一路DDS频率字+1、+10、归1、归63；当按下第二行的第1、2、3、4个按键时，分别对应第二路DDS频率字+1、+10、+100、归1；第三行按键对应第一路DDS的相位字，因为第一路DDS的相位字恒为0，所以本行按键做保留用；当按下第四行的第1、2、3、4个按键时，对应第二路DDS的相位字+1、+10、+100、归0。
>
> 本模块也做了类似矩阵键盘的设计，设置了四个寄存器存储前一时钟周期的频率字和相位字（事实上因为一路DDS 相位字恒为0，三个即可，但增加了可拓展性），当频率字和相位字被修改时，本模块会产生一个持续一个时钟周期的脉冲，这个脉冲会被接入DDS模块的`reset`上，当频率字和相位字改变时，DDS会被复位，以实现两路DDS的同步。这个设计是**保持相位差稳定的关键**。

6. `分频器`模块

> 可根据输入的分频因子对时钟进行分频，比如时钟因子为2对应2分频。

7. 数码管显示模块

> 本模块接受来自`FPreg`的频率字和相位字信息，并将其显示在数码管上。
>
> 左二位数码管显示第一路DDS的频率字的16进制，其右二位显示第二路DDS的频率字的16进制，其右一位显示第一路DDS的相位字（恒为0），最右三个数码管显示第二路DDS的相位字的16进制。比如第一路和第二路频率字分别为1、63，第一路和第二路相位字分别为0、255，则数码管显示01.3F.0.0FF.，其中小数点为分隔符。

8. `DAC`模块

> 由例程改编而来的数模转化模块，可以接受两路DDS信号，并通过告诉切换通道使两路DDS信号分别输出到AB两个通道。

三、源程序

1. `Lissajous`模块

```
/*************************************************************
// Copyright 2023.11.01-2023.11.15
// Contact with 210124996@qq.com
================ Lissajous.v ======================
>> Author       : Liu Deyang
>> Date         : 2023.11.01
>> Description  : DDS Lissajous
>> note         : (1)The copyright of the core FRreg.v and matrix_keypad.v belongs to the author, and >>                no one is allowed to copy.
>>              : (2)The phase difference carries a minus sign.
>>              : (3)Top-Level Entity
*************************************************************/
```

```
module Lissajous(
    input           clk,                //时钟输入
    input           reset,              //复位信号
    input [3:0]     row,                //矩阵键盘行输入
    output[3:0]     col,                //矩阵键盘列输出
    output [7:0]    seg_output,         //数码管显示
    output [7:0]    cat_output,         //数码管位选
    output          cs_o,               //选择信号输出
    output          data_o,             //数据输出信号
    output          dclock_o            //数据时钟输出
);

    wire            main_clk;
    wire            clk1;
    wire[11:0]      data1;
    wire[11:0]      data2;
    wire[4:0]       keynum;
    wire            pulse;
    wire[5:0]       Fword1;
    wire[7:0]       Fword2;
    wire[8:0]       Pword1;
    wire[8:0]       Pword2;
    wire[3:0]       r_col;
    wire            DDS_rst;

    DAC DAC(
        .clk_i(main_clk),
        .reset_i(reset),
        .dataa(data2),
        .datab(data1),
        .cs_o(cs_o),
        .data_o(data_o),
        .dclock_o(dclock_o)
    );

    matrix_keypad matrix_keypad(
          .clk(clk1),
          .reset(reset),
          .row(row),
          .col(col),
          .key(keynum),
          .keypress(pulse)
);

    DDS DDS1(
         .clk(clk1),
         .reset(DDS_rst),
         .Fword({2'b0,Fword1}),
         .Pword(Pword1),
         .DA_Data(data1)
);
    DDS DDS2(
         .clk(clk1),
         .reset(DDS_rst),
         .Fword(Fword2),
         .Pword(Pword2),
```

```
        .DA_Data(data2)
);

    divider divider1(
     .clk(clk),
     .n(7'd6),
     .reset(reset),
     .output_clk(main_clk)
);
    divider divider2(
     .clk(main_clk),
     .n(7'd68),
     .reset(reset),
     .output_clk(clk1)
);

    FPreg FPreg(
        .clk(clk1),
        .reset(reset),
        .keynum(keynum),
        .pulse(pulse),
        .Fword1(Fword1),
        .Fword2(Fword2),
        .Pword1(Pword1),
        .Pword2(Pword2),
        .DDS_rst(DDS_rst)
);

    seg_select seg_select(
        .seg_clock(clk1),
        .sys_rst(reset),
        .Fword1(Fword1),
        .Fword2(Fword2),
        .Pword2(Pword2),
        .seg_output(seg_output),
        .cat_output(cat_output)
);


endmodule
```

2. `rom`模块

```
module rom(
    input           clk,        //时钟输入
    input[8:0]      address,    //地址输入
    output[11:0]    data        //正弦波输出
);

    reg[11:0]       r_data;     //输出数据寄存器

always @(posedge clk) begin
    case(address[7:0])          //存储的正弦波数据
        8'd0:r_data<=12'b011111111111;
        8'd1:r_data<=12'b100000011000;
```

```
        8'd2:r_data<=12'b100000110001;
        8'd3:r_data<=12'b100001001010;
        8'd4:r_data<=12'b100001100011;
        8'd5:r_data<=12'b100001111100;
        8'd6:r_data<=12'b100010010101;
        8'd7:r_data<=12'b100010101110;
        8'd8:r_data<=12'b100011000111;
        8'd9:r_data<=12'b100011100000;
        8'd10:r_data<=12'b100011111001;
        8'd11:r_data<=12'b100100010010;
        8'd12:r_data<=12'b100100101011;
        8'd13:r_data<=12'b100101000100;
        8'd14:r_data<=12'b100101011100;
        8'd15:r_data<=12'b100101110101;
        8'd16:r_data<=12'b100110001110;
        8'd17:r_data<=12'b100110100110;
        8'd18:r_data<=12'b100110111111;
        8'd19:r_data<=12'b100111010111;
        8'd20:r_data<=12'b100111110000;
        8'd21:r_data<=12'b101000001000;
        8'd22:r_data<=12'b101000100000;
        8'd23:r_data<=12'b101000111001;
        8'd24:r_data<=12'b101001010001;
        8'd25:r_data<=12'b101001101001;
        8'd26:r_data<=12'b101010000001;
        8'd27:r_data<=12'b101010011000;
        8'd28:r_data<=12'b101010110000;
        8'd29:r_data<=12'b101011001000;
        8'd30:r_data<=12'b101011011111;
        8'd31:r_data<=12'b101011110111;
        8'd32:r_data<=12'b101100001110;
        8'd33:r_data<=12'b101100100101;
        8'd34:r_data<=12'b101100111100;
        8'd35:r_data<=12'b101101010011;
        8'd36:r_data<=12'b101101101010;
        8'd37:r_data<=12'b101110000000;
        8'd38:r_data<=12'b101110010111;
        8'd39:r_data<=12'b101110101101;
        8'd40:r_data<=12'b101111000011;
        8'd41:r_data<=12'b101111011010;
        8'd42:r_data<=12'b101111101111;
        8'd43:r_data<=12'b110000000101;
        8'd44:r_data<=12'b110000011011;
        8'd45:r_data<=12'b110000110000;
        8'd46:r_data<=12'b110001000110;
        8'd47:r_data<=12'b110001011011;
        8'd48:r_data<=12'b110001110000;
        8'd49:r_data<=12'b110010000101;
        8'd50:r_data<=12'b110010011001;
        8'd51:r_data<=12'b110010101110;
        8'd52:r_data<=12'b110011000010;
        8'd53:r_data<=12'b110011010110;
        8'd54:r_data<=12'b110011101010;
        8'd55:r_data<=12'b110011111110;
        8'd56:r_data<=12'b110100010001;
        8'd57:r_data<=12'b110100100100;
```

```
        8'd58:r_data<=12'b110100111000;
        8'd59:r_data<=12'b110101001010;
        8'd60:r_data<=12'b110101011101;
        8'd61:r_data<=12'b110101110000;
        8'd62:r_data<=12'b110110000010;
        8'd63:r_data<=12'b110110010100;
        8'd64:r_data<=12'b110110100110;
        8'd65:r_data<=12'b110110111000;
        8'd66:r_data<=12'b110111001001;
        8'd67:r_data<=12'b110111011010;
        8'd68:r_data<=12'b110111101011;
        8'd69:r_data<=12'b110111111100;
        8'd70:r_data<=12'b111000001101;
        8'd71:r_data<=12'b111000011101;
        8'd72:r_data<=12'b111000101101;
        8'd73:r_data<=12'b111000111101;
        8'd74:r_data<=12'b111001001100;
        8'd75:r_data<=12'b111001011100;
        8'd76:r_data<=12'b111001101011;
        8'd77:r_data<=12'b111001111010;
        8'd78:r_data<=12'b111010001000;
        8'd79:r_data<=12'b111010010110;
        8'd80:r_data<=12'b111010100101;
        8'd81:r_data<=12'b111010110010;
        8'd82:r_data<=12'b111011000000;
        8'd83:r_data<=12'b111011001101;
        8'd84:r_data<=12'b111011011010;
        8'd85:r_data<=12'b111011100111;
        8'd86:r_data<=12'b111011110100;
        8'd87:r_data<=12'b111100000000;
        8'd88:r_data<=12'b111100001100;
        8'd89:r_data<=12'b111100010111;
        8'd90:r_data<=12'b111100100011;
        8'd91:r_data<=12'b111100101110;
        8'd92:r_data<=12'b111100111001;
        8'd93:r_data<=12'b111101000100;
        8'd94:r_data<=12'b111101001110;
        8'd95:r_data<=12'b111101011000;
        8'd96:r_data<=12'b111101100010;
        8'd97:r_data<=12'b111101101011;
        8'd98:r_data<=12'b111101110100;
        8'd99:r_data<=12'b111101111101;
        8'd100:r_data<=12'b111110000110;
        8'd101:r_data<=12'b111110001110;
        8'd102:r_data<=12'b111110010110;
        8'd103:r_data<=12'b111110011110;
        8'd104:r_data<=12'b111110100101;
        8'd105:r_data<=12'b111110101101;
        8'd106:r_data<=12'b111110110011;
        8'd107:r_data<=12'b111110111010;
        8'd108:r_data<=12'b111111000000;
        8'd109:r_data<=12'b111111000110;
        8'd110:r_data<=12'b111111001100;
        8'd111:r_data<=12'b111111010001;
        8'd112:r_data<=12'b111111010110;
        8'd113:r_data<=12'b111111011011;
```

```
        8'd114:r_data<=12'b111111011111;
        8'd115:r_data<=12'b111111100100;
        8'd116:r_data<=12'b111111100111;
        8'd117:r_data<=12'b111111101011;
        8'd118:r_data<=12'b111111101110;
        8'd119:r_data<=12'b111111110001;
        8'd120:r_data<=12'b111111110100;
        8'd121:r_data<=12'b111111110110;
        8'd122:r_data<=12'b111111111000;
        8'd123:r_data<=12'b111111111010;
        8'd124:r_data<=12'b111111111011;
        8'd125:r_data<=12'b111111111100;
        8'd126:r_data<=12'b111111111101;
        8'd127:r_data<=12'b111111111101;
        8'd128:r_data<=12'b111111111110;
        8'd129:r_data<=12'b111111111101;
        8'd130:r_data<=12'b111111111101;
        8'd131:r_data<=12'b111111111100;
        8'd132:r_data<=12'b111111111011;
        8'd133:r_data<=12'b111111111010;
        8'd134:r_data<=12'b111111111000;
        8'd135:r_data<=12'b111111110110;
        8'd136:r_data<=12'b111111110100;
        8'd137:r_data<=12'b111111110001;
        8'd138:r_data<=12'b111111101110;
        8'd139:r_data<=12'b111111101011;
        8'd140:r_data<=12'b111111100111;
        8'd141:r_data<=12'b111111100100;
        8'd142:r_data<=12'b111111011111;
        8'd143:r_data<=12'b111111011011;
        8'd144:r_data<=12'b111111010110;
        8'd145:r_data<=12'b111111010001;
        8'd146:r_data<=12'b111111001100;
        8'd147:r_data<=12'b111111000110;
        8'd148:r_data<=12'b111111000000;
        8'd149:r_data<=12'b111110111010;
        8'd150:r_data<=12'b111110110011;
        8'd151:r_data<=12'b111110101101;
        8'd152:r_data<=12'b111110100101;
        8'd153:r_data<=12'b111110011110;
        8'd154:r_data<=12'b111110010110;
        8'd155:r_data<=12'b111110001110;
        8'd156:r_data<=12'b111110000110;
        8'd157:r_data<=12'b111101111101;
        8'd158:r_data<=12'b111101110100;
        8'd159:r_data<=12'b111101101011;
        8'd160:r_data<=12'b111101100010;
        8'd161:r_data<=12'b111101011000;
        8'd162:r_data<=12'b111101001110;
        8'd163:r_data<=12'b111101000100;
        8'd164:r_data<=12'b111100111001;
        8'd165:r_data<=12'b111100101110;
        8'd166:r_data<=12'b111100100011;
        8'd167:r_data<=12'b111100010111;
        8'd168:r_data<=12'b111100001100;
        8'd169:r_data<=12'b111100000000;
```

```
        8'd170:r_data<=12'b111011110100;
        8'd171:r_data<=12'b111011100111;
        8'd172:r_data<=12'b111011011010;
        8'd173:r_data<=12'b111011001101;
        8'd174:r_data<=12'b111011000000;
        8'd175:r_data<=12'b111010110010;
        8'd176:r_data<=12'b111010100101;
        8'd177:r_data<=12'b111010010110;
        8'd178:r_data<=12'b111010001000;
        8'd179:r_data<=12'b111001111010;
        8'd180:r_data<=12'b111001101011;
        8'd181:r_data<=12'b111001011100;
        8'd182:r_data<=12'b111001001100;
        8'd183:r_data<=12'b111000111101;
        8'd184:r_data<=12'b111000101101;
        8'd185:r_data<=12'b111000011101;
        8'd186:r_data<=12'b111000001101;
        8'd187:r_data<=12'b110111111100;
        8'd188:r_data<=12'b110111101011;
        8'd189:r_data<=12'b110111011010;
        8'd190:r_data<=12'b110111001001;
        8'd191:r_data<=12'b110110111000;
        8'd192:r_data<=12'b110110100110;
        8'd193:r_data<=12'b110110010100;
        8'd194:r_data<=12'b110110000010;
        8'd195:r_data<=12'b110101110000;
        8'd196:r_data<=12'b110101011101;
        8'd197:r_data<=12'b110101001010;
        8'd198:r_data<=12'b110100111000;
        8'd199:r_data<=12'b110100100100;
        8'd200:r_data<=12'b110100010001;
        8'd201:r_data<=12'b110011111110;
        8'd202:r_data<=12'b110011101010;
        8'd203:r_data<=12'b110011010110;
        8'd204:r_data<=12'b110011000010;
        8'd205:r_data<=12'b110010101110;
        8'd206:r_data<=12'b110010011001;
        8'd207:r_data<=12'b110010000101;
        8'd208:r_data<=12'b110001110000;
        8'd209:r_data<=12'b110001011011;
        8'd210:r_data<=12'b110001000110;
        8'd211:r_data<=12'b110000110000;
        8'd212:r_data<=12'b110000011011;
        8'd213:r_data<=12'b110000000101;
        8'd214:r_data<=12'b101111101111;
        8'd215:r_data<=12'b101111011010;
        8'd216:r_data<=12'b101111000011;
        8'd217:r_data<=12'b101110101101;
        8'd218:r_data<=12'b101110010111;
        8'd219:r_data<=12'b101110000000;
        8'd220:r_data<=12'b101101101010;
        8'd221:r_data<=12'b101101010011;
        8'd222:r_data<=12'b101100111100;
        8'd223:r_data<=12'b101100100101;
        8'd224:r_data<=12'b101100001110;
        8'd225:r_data<=12'b101011110111;
```

```
        8'd226:r_data<=12'b101011011111;
        8'd227:r_data<=12'b101011001000;
        8'd228:r_data<=12'b101010110000;
        8'd229:r_data<=12'b101010011000;
        8'd230:r_data<=12'b101010000001;
        8'd231:r_data<=12'b101001101001;
        8'd232:r_data<=12'b101001010001;
        8'd233:r_data<=12'b101000111001;
        8'd234:r_data<=12'b101000100000;
        8'd235:r_data<=12'b101000001000;
        8'd236:r_data<=12'b100111110000;
        8'd237:r_data<=12'b100111010111;
        8'd238:r_data<=12'b100110111111;
        8'd239:r_data<=12'b100110100110;
        8'd240:r_data<=12'b100110001110;
        8'd241:r_data<=12'b100101110101;
        8'd242:r_data<=12'b100101011100;
        8'd243:r_data<=12'b100101000100;
        8'd244:r_data<=12'b100100101011;
        8'd245:r_data<=12'b100100010010;
        8'd246:r_data<=12'b100011111001;
        8'd247:r_data<=12'b100011100000;
        8'd248:r_data<=12'b100011000111;
        8'd249:r_data<=12'b100010101110;
        8'd250:r_data<=12'b100010010101;
        8'd251:r_data<=12'b100001111100;
        8'd252:r_data<=12'b100001100011;
        8'd253:r_data<=12'b100001001010;
        8'd254:r_data<=12'b100000110001;
        8'd255:r_data<=12'b100000011000;
    endcase
end

    assign data = address[8] ? 12'b111111111110 - r_data : r_data; //前256点直接输出，后256点对称变换

endmodule
```

3. DDS 模块

```
module DDS(
    input           clk,        //输入时钟
    input           reset,      //复位信号，此复位信号不是来自顶层模块，而是来自FPreg
    input[7:0]      Fword,      //频率字
    input[8:0]      Pword,      //相位字
    output[11:0]    DA_Data     //数据输出
);
    reg[7:0]        r_Fword;    //频率字寄存器
    reg[8:0]        r_Pword;    //相位字寄存器

    reg[8:0]        Fcnt;       //相位累加器寄存器

    wire[8:0]       rom_addr;   //相位调制器寄存器

```

```
    always@(posedge clk) begin  //寄存频率字和相位字
        r_Fword <= Fword;
        r_Pword <= Pword;
    end

    always@(posedge clk or posedge reset) begin //相位累加器
        if(reset) begin
            Fcnt <= 9'b000000000;
        end
        else
            Fcnt <= Fcnt + r_Fword;
    end


    assign rom_addr = Fcnt + r_Pword; //相位调制器

    rom rom(
        .clk(clk),
        .address(rom_addr),
        .data(DA_Data)
    );


endmodule
```

4. 4*4键盘模块

```
module matrix_keypad(
    input               clk,        // 主时钟
    input               reset,      // 复位信号
    input [3:0]         row,        // 从键盘驱动的行信号
    output reg [3:0]    col,        // 从键盘驱动的列信号
    output reg [4:0]    key,        // 输出的键值
    output reg          keypress    // 键被按下的脉冲信号
);

    parameter idle=3'd0,check_1st=3'd1,check_2nd=3'd2,check_3rd=3'd3,check_4th=3'd4; //状态机的状态

    reg[2:0] present_state, next_state;     //现在状态和下一状态
    reg [4:0] prev_key;                     // 上一个时钟周期的键值

    always @(posedge clk or posedge reset) begin //三进程之一，将下一状态赋值给现在状态
        if (reset) begin
            present_state <= idle;
        end
        else begin
            present_state <= next_state;
    end
     end

    always @(*) begin //三进程至二，根据当前状态和输入判断下一状态
        case (present_state)
```

```
            idle:begin
                        if(row == 4'b1111) next_state = idle;
                        else next_state = check_1st;
                    end
                    check_1st:begin
                        if(row == 4'b1111) next_state = check_2nd;
                        else next_state = check_1st;
                    end
                    check_2nd:begin
                        if(row == 4'b1111) next_state = check_3rd;
                        else next_state = check_2nd;
                    end
                    check_3rd:begin
                        if(row == 4'b1111) next_state = check_4th;
                        else next_state = check_3rd;
                    end
                    check_4th:begin
                        if(row == 4'b1111) next_state = idle;
                        else next_state = check_4th;
                    end
        endcase
    end

    always@(present_state) begin    //三进程之三，根据当前状态决定输出
        case(present_state)
            idle: col = 4'b0000;
            check_1st: col = 4'b0111;
            check_2nd: col = 4'b1011;
            check_3rd: col = 4'b1101;
            check_4th: col = 4'b1110;
        endcase
    end

    always@(posedge clk or posedge reset) begin //根据当前的行值和列值得出所对应的键值
        if(reset) begin
            key <= 5'b10000;
            prev_key <= 5'b10000;
        end
        else begin
            case(col)
                4'b0111:
                        case(row)
                            4'b0111: key <= 5'b00011;
                            4'b1011: key <= 5'b00111;
                            4'b1101: key <= 5'b01011;
                            4'b1110: key <= 5'b01111;
                            default: key <= 5'b10000;
                        endcase
                    4'b1011:
                        case(row)
                                4'b0111: key <= 5'b00010;
                                4'b1011: key <= 5'b00110;
                                4'b1101: key <= 5'b01010;
                                4'b1110: key <= 5'b01110;
                                default: key <= 5'b10000;
```

```
                            endcase
                    4'b1101:
                        case(row)
                                4'b0111: key <= 5'b00001;
                                4'b1011: key <= 5'b00101;
                                4'b1101: key <= 5'b01001;
                                4'b1110: key <= 5'b01101;
                                default: key <= 5'b10000;
                            endcase
                    4'b1110:
                        case(row)
                                4'b0111: key <= 5'b00000;
                                4'b1011: key <= 5'b00100;
                                4'b1101: key <= 5'b01000;
                                4'b1110: key <= 5'b01100;
                                default: key <= 5'b10000;
                            endcase
                    default:key <= 5'b10000;
            endcase

            // 当键值改变时，输出一个脉冲触发FPreg修改
            if (key != prev_key) begin
                keypress <= 1'b1;
                prev_key <= key;
            end
            else begin
                keypress <= 1'b0;
                     prev_key <= key;
            end
        end
    end

endmodule

```

5. `FPreg` 模块

```
module FPreg(
    input           clk,            //时钟
    input           reset,          //复位信号
    input[4:0]      keynum,         //键值输入
    input           pulse,          //脉冲输入
    output[5:0]     Fword1,         //一路频率字输出
    output[7:0]     Fword2,         //二路频率字输出
    output[8:0]     Pword1,         //一路相位字输出
    output[8:0]     Pword2,         //二路相位字输出
    output reg      DDS_rst         //DDS复位信号
);

    reg[5:0]        r_Fword1;       //一路频率字寄存器
    reg[7:0]        r_Fword2;       //二路频率字寄存器
    reg[8:0]        r_Pword1;       //一路相位字寄存器
    reg[8:0]        r_Pword2;       //二路相位字寄存器
    reg[5:0]        prev_Fword1;    //前一状态寄存器
    reg[7:0]        prev_Fword2;    //同上
```

```
    reg[8:0]        prev_Pword1;    //同上
    reg[8:0]        prev_Pword2;    //同上

    always@(posedge clk or posedge reset) begin
        if(reset) begin
            r_Fword1 <= 6'd1;
            r_Fword2 <= 8'd1;
            r_Pword1 <= 9'd0;
            r_Pword2 <= 9'd0;
        end

        else if(pulse) begin //当接收到脉冲时，证明按键被按下（或由按下到释放，但此种状态不做任何操作）
            case(keynum) //读取键值，根据键值做修改操作
                4'b00000:   r_Fword1 <= r_Fword1 + 1'b1;
                4'b00001:   r_Fword1    <= r_Fword1 + 4'd10;
                4'b00010:   r_Fword1 <= 1'b1;
                4'b00011:   r_Fword1 <= 6'd63;
                4'b00100:   r_Fword2 <= r_Fword2 + 1'b1;
                4'b00101:   r_Fword2 <= r_Fword2 + 4'd10;
                4'b00110:   r_Fword2 <= r_Fword2 + 7'd100;
                4'b00111:   r_Fword2 <= 1'b1;
                4'b01000:   r_Pword1 <= r_Pword1 + 1'b1;
                4'b01001:   r_Pword1 <= r_Pword1 + 4'd10;
                4'b01010:   r_Pword1 <= r_Pword1 + 7'd100;
                4'b01011:   r_Pword1 <= 9'b0;
                4'b01100:   r_Pword2 <= r_Pword2 + 1'b1;
                4'b01101:   r_Pword2 <= r_Pword2 + 4'd10;
                4'b01110:   r_Pword2 <= r_Pword2 + 7'd100;
                4'b01111:   r_Pword2 <= 9'd0;
                default:;
            endcase
        end
        else if((r_Fword1 != prev_Fword1) || (r_Fword2 != prev_Fword2) || (r_Pword1 != prev_Pword1) || (r_Pword2 != prev_Pword2)) begin //如果键值改变，则给脉冲赋值1
            DDS_rst <= 1'b1;
            prev_Fword1 <= r_Fword1;
            prev_Fword2 <= r_Fword2;
            prev_Pword1 <= r_Pword1;
            prev_Pword2 <= r_Pword2;
        end
        else begin  //否则脉冲置0
            DDS_rst <= 1'b0;
            prev_Fword1 <= r_Fword1;
            prev_Fword2 <= r_Fword2;
            prev_Pword1 <= r_Pword1;
            prev_Pword2 <= r_Pword2;
        end
    end

assign Fword1 = r_Fword1;
assign Fword2 = r_Fword2;
assign Pword1 = r_Pword1;
assign Pword2 = r_Pword2;

```

```

endmodule
```

6. 分频器模块

```
module divider (
    input           clk,        //时钟输入
    input [6:0]     n,          //分频因子
    input           reset,      //复位信号
    output          output_clk  //时钟输出
);
    reg             clk_temp;   //输出寄存器
    reg [6:0]       counter_ou; //计数器寄存器，用于翻转时钟信号

    always @(posedge clk or posedge reset) begin
        if(reset) begin
            counter_ou <= 7'd0;
            clk_temp <= 1'b0;
        end else if(counter_ou < n/2-1) begin //不到一半++
            counter_ou <= counter_ou + 1'b1;
        end else begin  //到一半翻转
            counter_ou <= 7'd0;
            clk_temp <= ~clk_temp;
        end
    end

    assign output_clk = clk_temp;

endmodule
```

7. 数码管显示模块

```
module seg_select(
    input           seg_clock,      //时钟
    input           sys_rst,        //复位信号
    input[5:0]      Fword1,         //三组数据输入
    input[7:0]      Fword2,
    input[8:0]      Pword2,
    output [7:0]    seg_output,     //数码管显示
    output [7:0]    cat_output      //数码管位选
);

    reg[7:0]        seg;
    reg[7:0]        cat;

    always @(posedge seg_clock or posedge sys_rst) begin
        if(sys_rst) begin
            seg <= 8'b00000000;
            cat <= 8'b11111111;
        end
        else begin
            if(cat == 8'b11111111) begin
                cat <= 8'b11111110;
            end
```

```
            else begin
                case(cat)
                    8'b10111111:
                        case(Fword1[5:4])
                            2'b00:seg <= 8'b11111100;//0
                            2'b01:seg <= 8'b01100000;//1
                            2'b10:seg <= 8'b11011010;//2
                            2'b11:seg <= 8'b11110010;//2
                        endcase
                    8'b11011111:
                        case(Fword1[3:0])
                            4'b0000: seg <= 8'b11111101;//0
                            4'b0001: seg <= 8'b01100001;//1
                            4'b0010: seg <= 8'b11011011;//2
                            4'b0011: seg <= 8'b11110011;//3
                            4'b0100: seg <= 8'b01100111;//4
                            4'b0101: seg <= 8'b10110111;//5
                            4'b0110: seg <= 8'b10111111;//6
                            4'b0111: seg <= 8'b11100001;//7
                            4'b1000: seg <= 8'b11111111;//8
                            4'b1001: seg <= 8'b11110111;//9
                            4'b1010: seg <= 8'b11101111;//A
                            4'b1011: seg <= 8'b00111111;//b
                            4'b1100: seg <= 8'b10011101;//C
                            4'b1101: seg <= 8'b01111011;//d
                            4'b1110: seg <= 8'b10011111;//E
                            4'b1111: seg <= 8'b10001111;//F
                        endcase
                    8'b11101111:
                        case(Fword2[7:4])
                            4'b0000: seg <= 8'b11111100;
                            4'b0001: seg <= 8'b01100000;
                            4'b0010: seg <= 8'b11011010;
                            4'b0011: seg <= 8'b11110010;
                            4'b0100: seg <= 8'b01100110;
                            4'b0101: seg <= 8'b10110110;
                            4'b0110: seg <= 8'b10111110;
                            4'b0111: seg <= 8'b11100000;
                            4'b1000: seg <= 8'b11111110;
                            4'b1001: seg <= 8'b11110110;
                            4'b1010: seg <= 8'b11101110;
                            4'b1011: seg <= 8'b00111110;
                            4'b1100: seg <= 8'b10011100;
                            4'b1101: seg <= 8'b01111010;
                            4'b1110: seg <= 8'b10011110;
                            4'b1111: seg <= 8'b10001110;
                        endcase
                    8'b11110111:
                        case(Fword2[3:0])
                            4'b0000: seg <= 8'b11111101;
                            4'b0001: seg <= 8'b01100001;
                            4'b0010: seg <= 8'b11011011;
                            4'b0011: seg <= 8'b11110011;
                            4'b0100: seg <= 8'b01100111;
                            4'b0101: seg <= 8'b10110111;
                            4'b0110: seg <= 8'b10111111;
```

```
                                4'b0111: seg <= 8'b11100001;
                                4'b1000: seg <= 8'b11111111;
                                4'b1001: seg <= 8'b11110111;
                                4'b1010: seg <= 8'b11101111;
                                4'b1011: seg <= 8'b00111111;
                                4'b1100: seg <= 8'b10011101;
                                4'b1101: seg <= 8'b01111011;
                                4'b1110: seg <= 8'b10011111;
                                4'b1111: seg <= 8'b10001111;
                            endcase
                        8'b11111011: seg <= 8'b11111101;
                        8'b11111101:
                            case(Pword2[8])
                                1'b0: seg <= 8'b11111100;
                                1'b1: seg <= 8'b01100000;
                            endcase
                        8'b11111110:
                            case(Pword2[7:4])
                                4'b0000: seg <= 8'b11111100;
                                4'b0001: seg <= 8'b01100000;
                                4'b0010: seg <= 8'b11011010;
                                4'b0011: seg <= 8'b11110010;
                                4'b0100: seg <= 8'b01100110;
                                4'b0101: seg <= 8'b10110110;
                                4'b0110: seg <= 8'b10111110;
                                4'b0111: seg <= 8'b11100000;
                                4'b1000: seg <= 8'b11111110;
                                4'b1001: seg <= 8'b11110110;
                                4'b1010: seg <= 8'b11101110;
                                4'b1011: seg <= 8'b00111110;
                                4'b1100: seg <= 8'b10011100;
                                4'b1101: seg <= 8'b01111010;
                                4'b1110: seg <= 8'b10011110;
                                4'b1111: seg <= 8'b10001110;
                            endcase
                        8'b01111111:
                            case(Pword2[3:0])
                                4'b0000: seg <= 8'b11111101;
                                4'b0001: seg <= 8'b01100001;
                                4'b0010: seg <= 8'b11011011;
                                4'b0011: seg <= 8'b11110011;
                                4'b0100: seg <= 8'b01100111;
                                4'b0101: seg <= 8'b10110111;
                                4'b0110: seg <= 8'b10111111;
                                4'b0111: seg <= 8'b11100001;
                                4'b1000: seg <= 8'b11111111;
                                4'b1001: seg <= 8'b11110111;
                                4'b1010: seg <= 8'b11101111;
                                4'b1011: seg <= 8'b00111111;
                                4'b1100: seg <= 8'b10011101;
                                4'b1101: seg <= 8'b01111011;
                                4'b1110: seg <= 8'b10011111;
                                4'b1111: seg <= 8'b10001111;
                            endcase
                        default:seg <= 8'b00000000;
                    endcase
```

```
                cat <= {cat[6:0], cat[7]};
            end
        end
    end

    assign seg_output = seg;
    assign cat_output = cat;

endmodule
```

8. DAC模块

```
-- DAC_A and DAC_B output the same sawtooth wave
-- 一次16位的数据传输需要16个输出时钟+1个片选信号（维持一个输出时钟的高电平，其余时刻为低电平）
-- 对应了一次数据传输需要32+2个系统时钟，也就是说数据传输的周期为34个系统时钟周期
-- 但同时，data_a与data_b交替输出，那么DDS_DA需要2*34=68个系统时钟周期输出一次dataA和dataB
-- 那么控制DDS_DA的时钟需要进行系统时钟的68分频。
-- 如果要求输出波形的基频>200HZ，那么就要求dds的时钟频率>200*512=102400HZ，那么就要求系统时钟的频率>68*102400hz=6963200HZ=6.9632Mhz，很明显应该选50MHZ的原始时钟。

-- 引入必要的IEEE标准库
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;


entity DAC is
    port (
        clk_i :             in std_logic;                       -- 时钟输入
        reset_i :           in std_logic;                       -- 复位输入
        dataa :             in std_logic_vector(11 downto 0);
        datab :             in std_logic_vector(11 downto 0);

        cs_o :              out std_logic;                      -- 选择信号输出
        data_o :            out std_logic;                      -- 数据输出信号
        dclock_o :          out std_logic                       -- 数据时钟输出
    );
    end DAC;



    architecture DAC_arch of DAC is

        component tlv5638 is
        port (
            cs_o :              out std_logic;                      -- 选择信号输出
            data_o :            out std_logic;                      -- 数据输出信号
```

```
            dclock_o :          out std_logic;                      -- 数据时钟输出

            reset_i :           in std_logic;                       -- 复位输入
            clk_i :             in std_logic;                       -- 时钟输入

            start_i :           in std_logic;                       -- 启动输入
            eoc_o :             out std_logic;                      -- 转换结束输出
            data_i :            in std_logic_vector(15 downto 0)    -- 16位输入数据
        );
        end component;

            signal start_da :               STD_LOGIC;
            signal eoc_da :                 STD_LOGIC;
            signal data, data_a, data_b :   std_logic_vector(15 downto 0);

            signal timer0 :                 std_logic;
            signal state :                  integer range 0 to 2;
            signal waveform_val :           std_logic_vector(7 downto 0);
            signal cnt :                    integer range 0 to 3;
            signal delay :                  integer range 0 to 63;
            signal channel :                integer range 0 to 1 ;  `--用于切换ab，0输出b

begin
    -- 实例化tlv5638组件并进行端口映射
    u1: tlv5638 PORT MAP(cs_o => cs_o, data_o => data_o, reset_i => reset_i, clk_i => clk_i, dclock_o => dclock_o, start_i => start_da, eoc_o => eoc_da, data_i => data);

        -- 进程处理，用于在设备初始化期间等待一段时间
        process(clk_i, reset_i)
        begin
            if reset_i = '1' then
                delay <= 0;
            timer0 <= '0';
        elsif clk_i'event and clk_i = '1' then --每次高电平delay+1，复位之后等待输入时钟64个周期--再等待2个输入时钟周期（初始化状态）
            if (delay = 63) then
                timer0 <= '1';
            else
                delay <= delay + 1;
            end if;
                end if;
                    end process;

                    -- 另一个进程处理，用于初始化和控制DAC
                    process(reset_i, clk_i)
                    begin
                        if reset_i = '1' then
                            state <= 0;
```

```
                        channel <= 0;
                    elsif clk_i'event and clk_i = '0' then --时钟下降沿触发
                        case state is
                            when 0 =>     -- 等待设备初始化的一段时间
                            if (timer0 = '1') then
                                state <= 1;
                            end if;
                                when 1 =>
                                data <= "1101000000000010"; -- 设置参考电压为2.048 V
                            start_da <= '1';
                        state <= 2;
                    when 2 =>  --开始工作
                    if (eoc_da = '1') then  --每次结束channel翻转一次，对应ab输出翻转一次
                        data_b <=  "0001" & datab;  -- 写入 DAC B 的数据到缓冲区
                    data_a <=  "1000" & dataa;  -- 同时写入新的 DAC A 值并更新 DAC A 和 B


                if (channel = 0) then
                    data <= data_b; --相当于t=0
                else
                    data <= data_a; --相当于t=1
                end if;
                    if (channel < 1) then
                        channel <= channel + 1;
                    else
                        channel <= 0;
                    end if;
                        start_da <= '1';

                    else
                        start_da <= '0';    --如果还没结束，那么开始符号一直为低电平
                    end if;
                        when others => null;
                    end case;
                        end if;
                            end process;

                            end;
```

9. `tlv5638`

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity tlv5638 is
    port (
        cs_o : out std_logic; --片选输出 --片选信号高到低转换，实现data_b到data_a的转变，一下输出16位，输出时钟高电平时输出数据有效。--而输出时钟为系统时钟的二分之一。--外部test控制ab转换
```

```
        data_o : out std_logic;
        dclock_o : out std_logic; --输出时钟

        reset_i : in std_logic;
        clk_i : in std_logic;

        start_i  : in std_logic; --启动输入
        eoc_o  : out std_logic; --结束输出
        data_i : in std_logic_vector( 15 downto 0) --16bit信号
    );
end tlv5638;

architecture tlv5638_arch of tlv5638 is
    signal start : std_logic;   --表示开始的电平
    type States is ( IDEL , BUSY );
    signal state : States; --表示状态的信号
    signal eoc : std_logic; --表示结束
    signal reset_int : std_logic;
    signal data_out : std_logic_vector( 15 downto 0);
    signal data_out_addr : integer range 0 to 15;

begin
    eoc_o <= eoc; --output输出结束
    process(clk_i , reset_i , eoc) --状态机一
    begin
        if ( reset_i = '1' or eoc = '1' ) then
        state <= IDEL; --高电平复位或者结束输出为1时，输出空闲状态

        elsif ( clk_i 'event and clk_i = '0' ) then --输入时钟下降沿时
            if ( start_i = '1' and state = IDEL ) then --当输入符号为高电平并且状态为空闲时
                state <=  BUSY; --转换为工作状态
                data_out <= data_i; --此时将16bit数据赋值给data_out
            end if;
        end if;
    end process;

    process( clk_i , reset_i )--状态机二
    begin
    if ( reset_i = '1' ) then --当复位时
    reset_int <= '0'; --reset_int表示复位标志赋值低电平
        elsif ( clk_i 'event and clk_i = '0' ) then --如果时钟下降
            if ( eoc = '1' ) then --如果结束
                reset_int <= '1'; --赋值1
            else
                reset_int <= '0'; --如果没有结束，每一个低电平就给他赋值为0
            end if;
        end if;
    end process;

    process( clk_i , reset_i , reset_int )
    variable cnt_wr : integer range 0 to 16; --传输计数器
    variable t : integer range 0 to 1; --整数01变量
    begin
        if ( reset_i = '1' or reset_int = '1' ) then --复位且空闲状态
            cs_o <= '1'; --片选高电平
```

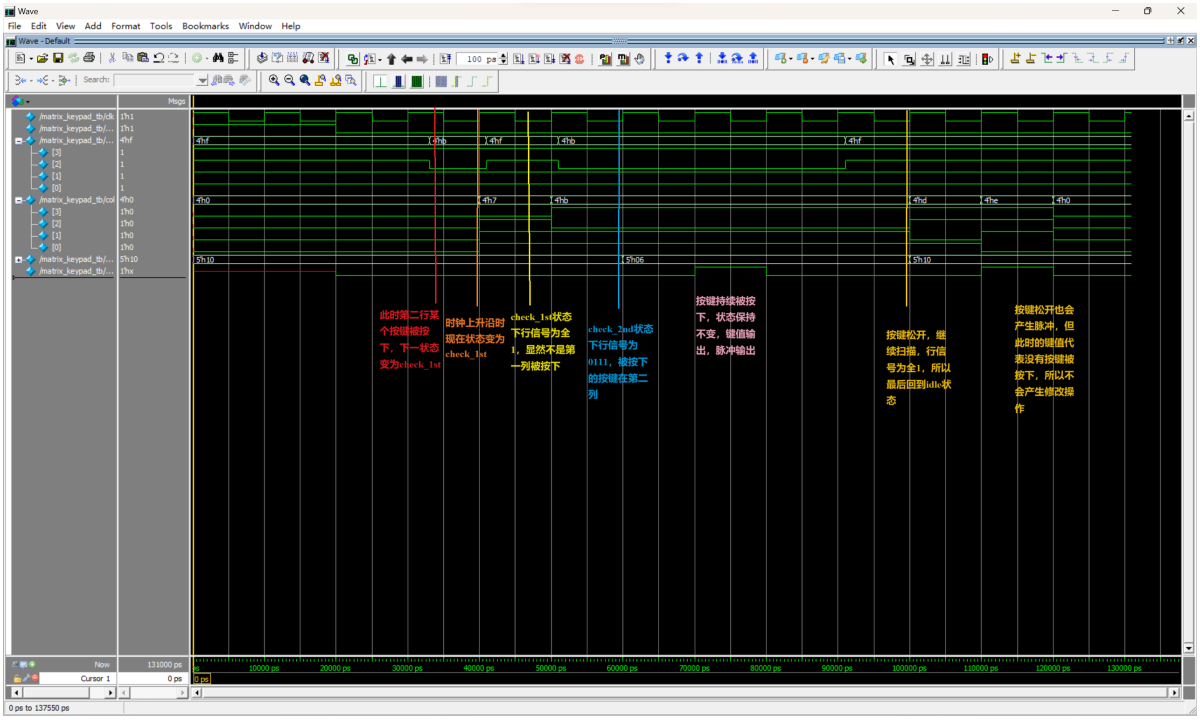
```
            cnt_wr := 0; --重置计数器
            t := 0; --重置t
            dclock_o <= '0'; --输出时钟始终为低电平
            eoc <= '0'; --结束标志重置
            data_out_addr <= 15; --输出数据的索引地址变为15
            data_o <= '0'; --输出模拟量为0
        elsif ( clk_i 'event and clk_i = '1' ) then --输入时钟高电平
            if ( state = BUSY ) then    --如果工作
                if ( cnt_wr < 16 ) then --计数小于16
                    eoc <= '0';
                    cs_o <= '0';
                    if ( t = 0 ) then
                        data_o <= data_out(data_out_addr); --高电平赋值一次，高位向低位一位一位输出是由DAC的内部配置所决定的
                        dclock_o <= '1'; --有模拟量输出时，输出时钟为1，正常的频率为输入时钟的二分之一
                        if (  data_out_addr > 0 ) then
                            data_out_addr <= data_out_addr - 1; --高位向低位输出
                        end if ;
                    else
                        dclock_o <= '0';
                        cnt_wr := cnt_wr + 1;
                    end if;
                    if ( t < 1 ) then --一次高电平完成一次01反转
                        t := t + 1;
                    else
                        t := 0 ;
                    end if ;
                else
                    cs_o <= '1'; --如果16位都传输完成
                    eoc <= '1';
                end if;
            end if;
        end if;
    end process;

end tlv5638_arch;
```

四、仿真波形及波形分析（篇幅有限，testbenches见附件）

一、DDS模块仿真

从图中可以看出频率字为当频率字为1，相位字为0时，可以正常输出一组初相为0的基频(本设计所能达到最小绝对频率)正弦波；当频率字为2，相位字为255(512点正弦波的一半，所以初相是180)，可以输出一组初相为180度的正弦波，且频率为前一组正弦波的频率的二倍。

二、matrix_keypad模块仿真



如图所示，本仿真波形演示了第二行第二列按键，即键值对应6(从0开始)的按键被按下的波形。

首先，按键被按下，由于列扫描信号在idle状态为全0，所以只有被按下的按键对应的那一行会出现0，其他仍为1；下一状态的判定是不依赖于时序的，所以按键被按下时下一状态会立刻便为check_1st，这个状态一直持续到了下一上升沿，所以会被同步到现在状态。

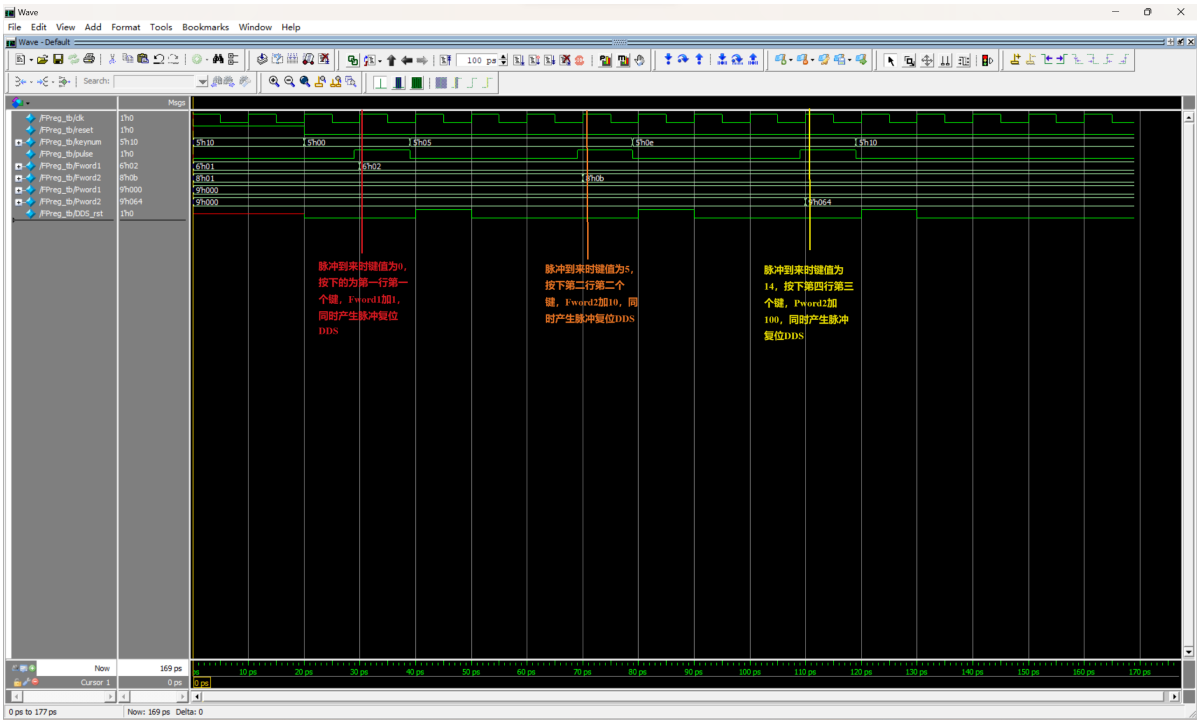
当现在状态为check_1st时，列扫描信号输出为0111，假如被按下的按键在第一列，则行信号应该为1011，但此时行信号为全1，所以进入下一状态check_2nd。

当现在状态为check_2nd时，列扫描信号输出1011，此时行信号为1011，证明列信号给出的0位置恰好在被按下按键所处的那一列，状态保持在check_2nd。键值变为6，然后输出脉冲。

当按键松开时，行信号又回到全1，状态会继续变化，直到回到idle，发现行信号还是全1，证明没有按键按下，保持为idle状态。

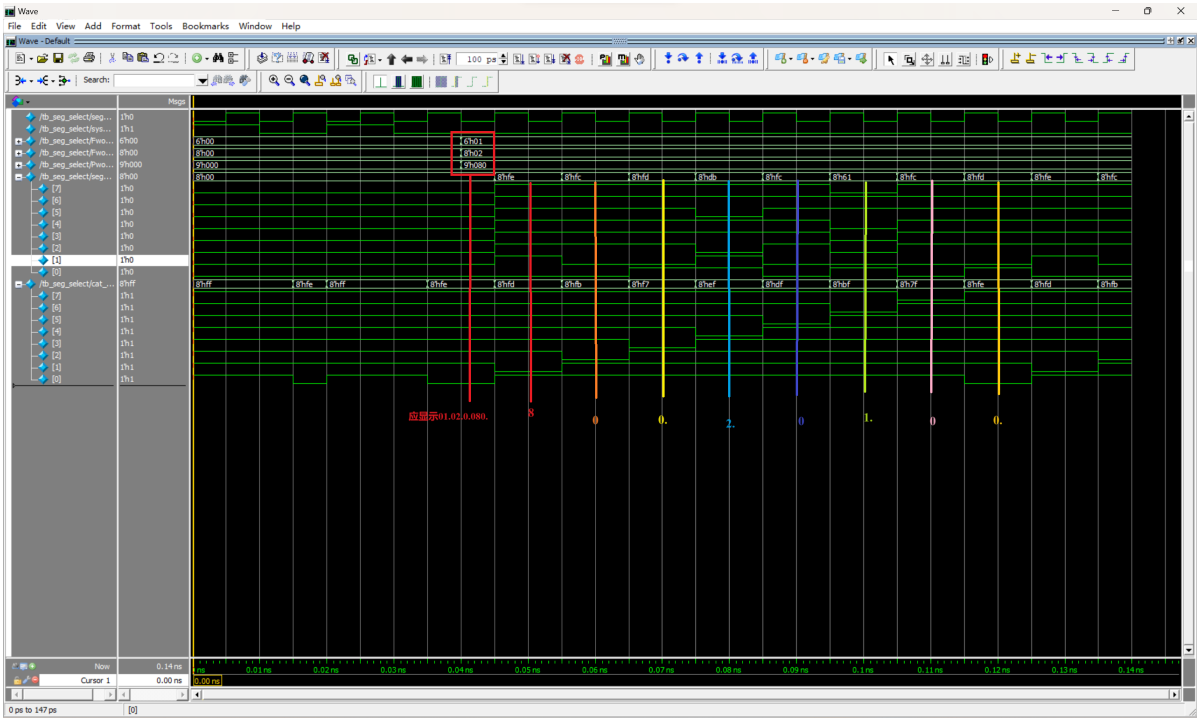
状态机在变换状态进行扫描时键值会不会随之变化呢？显然是不会的。判断键值的逻辑是根据行列中0的位置判断按键的位置，对应到相应的键值上。这是通过case语句实现的，当状态机变换状态时，行信号肯定是为全1的，case语句中没有对应项，所以归为default，default状态下键值会保持不变。

三、FPreg模块仿真



本仿真波形中仿真了Fword1加1，Fword2加10以及Pword2加100操作。在寄存器内的频率字和相位字改变时会同时产生脉冲复位DDS相位调制器内的累加值，使两路DDS同步，从而达到相位稳定。

四、seg_select模块仿真



如图所示，Fword1为01，Fword2为02，Pword2为080，所以应该显示01.02.0.080，仿真结果显示是正常的。

五、仿真总结

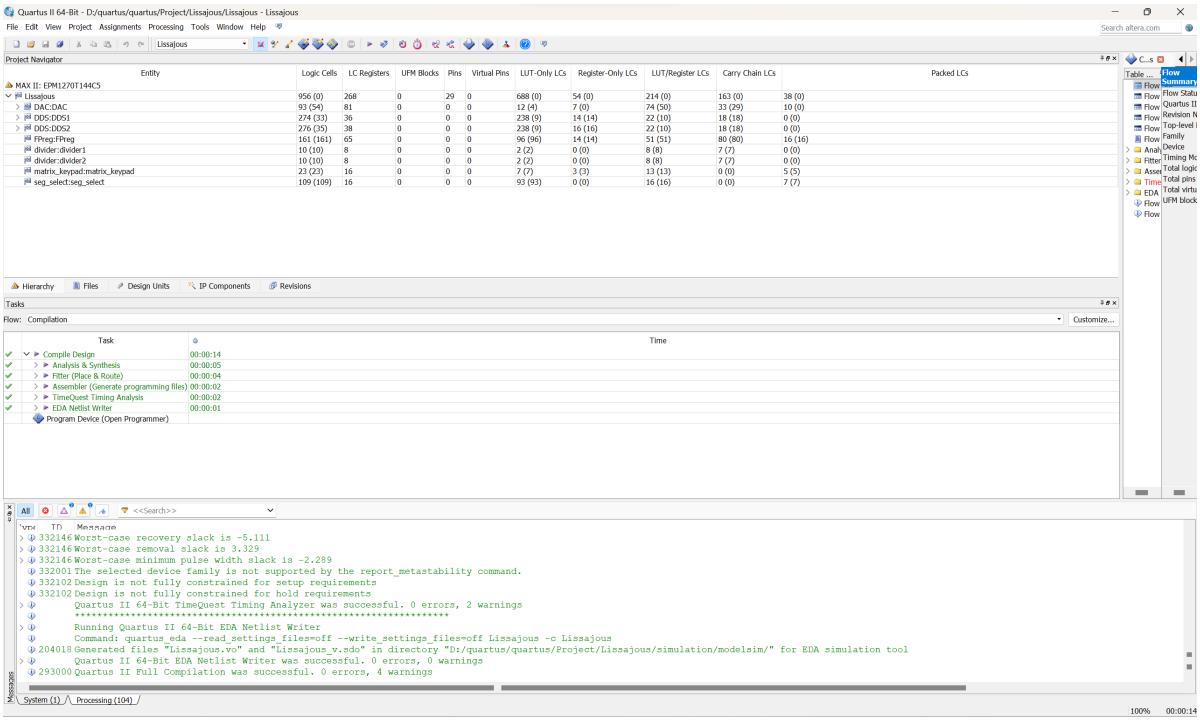
在仿真部分中，我对四个主要模块进行了仿真，其余模块如分频器较为简单，不需要仿真，Lissajous模块是对其他模块的例化，DAC模块仿真意义不大，可直接用示波器进行测试。至此，整个设计的仿真部分验证完毕，在所有模块功能均正确的情况下，总体出错的概率不大，所以没有对整体进行仿真，而是选择了进行下载测试，进而观察有无其他问题并进行修改。

五、功能说明及资源利用情况

一、功能说明

本设计实现了一个基于DDS的李萨如图形生成器，当将AB通道分别接入示波器并打开X-Y显示模式时，可以观察到对应频率比和相位差的李萨如图形。DDS的频率和相位通过离散的频率字和相位字控制，DDS使用的时钟为50/(6*68)MHz，所以DDS频率字为1时对应的频率为240Hz左右，其他情况下的频率即为240Hz*Fword，相位字表示从512点正弦波的第几个点开始，所以相位即为Pword/512*2π。两路DDS的频率和相位可通过矩阵键盘任意设置。两路DDS的频率字和相位字会以16进制形式显示在数码管上。

二、资源利用情况



MAXⅡ资源利用情况，如图所示。版上资源使用了tlv5638，矩阵键盘，数码管，按钮等。

六、故障及问题分析

一、相位不稳定的问题

在仿真无误后，我进行了下载测试，在测试过程中发现，当使用键盘修改频率字时，相位差会发生一个随机跳动。分析后发现，产生该问题的原因应该是修改频率字影响到了高速的相位累加器计数，从而导致两路DDS出现不受控制的相位差，此时相位字已经不具有意义。为解决该问题，我在FPreg中加了一个脉冲信号，当FPreg中存储的频率字或者相位字被修改时，就会产生一个复位脉冲复位DDS，使两路

DDS都是从0+Pword点开始计数，从而保证了相位差绝对稳定于Pword。

七、总结和结论

本报告详细介绍了基于直接数字频率合成（DDS）技术的李萨如图形生成器的设计和实现。首先概述了设计任务的目标，即创建一个能够生成李萨如图形的电子系统。

在实现部分，我展示了该系统如何通过精确的频率控制和相位调整来生成不同形状和大小的李萨如图形。报告中包含了详细的源程序代码、仿真波形以及功能测试结果，这些内容展示了系统的有效性和灵活性。此外，我还分析了系统的性能，包括资源利用情况和响应速度。

通过本项目，我成功实现了一个基于DDS的李萨如图形生成器，它不仅满足了设计要求，还展示了良好的性能和可靠性。该生成器能够准确生成各种李萨如图形，操作简便。尽管如此，我也意识到系统仍有改进空间。例如，数码管显示可以改进，使其可以直接显示十进制模拟频率及相位。

总的来说，通过这个项目，我积累了宝贵的设计经验，提升了自己的设计水平。