项目描述

项目介绍

本文中，我们将通过硬禾学堂的“基于小脚丫FPGA的电赛训练平台”来实现DDS任意波形发生器，并借助该训练平台上的“10位/125Msps高速DAC模块”生成最终波形，同时在OLED屏幕上显示相关参数。

项目要求

1. 通过板上的高速DAC（10bits/125Msps）配合FPGA内部DDS的逻辑，生成波形可调（正弦波、三角波、方波）、频率可调（DC-）、幅度可调的波形。
2. 生成模拟信号的频率范围为DC-20MHz，调节精度为1Hz；
3. 生成模拟信号的幅度为最大1Vpp，调节范围为0.1V-1V；
4. 在OLED上显示当前波形的形状、波形的频率以及幅度；
5. 利用板上旋转编码器和按键能够对波形进行切换、进行参数调节。

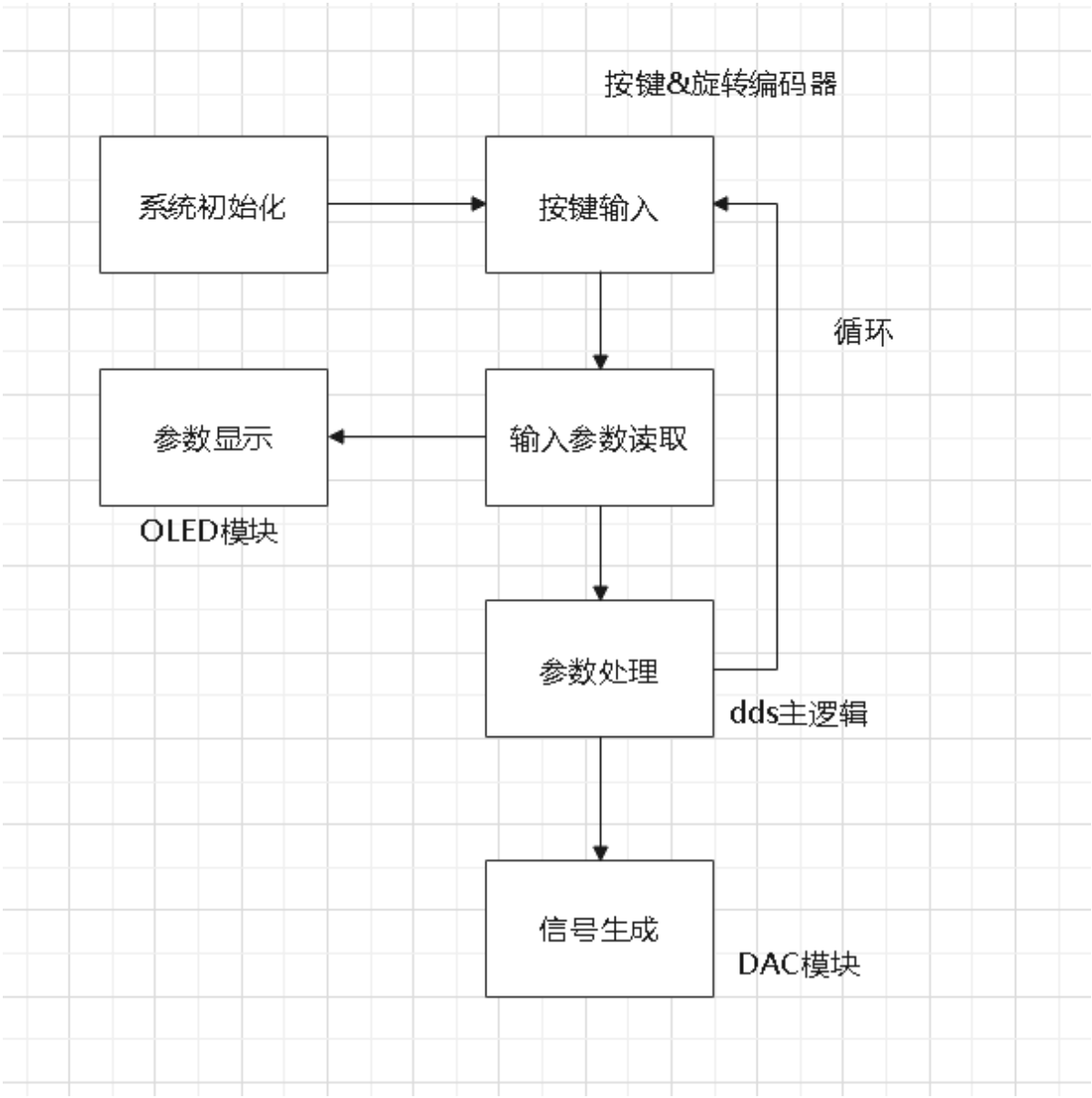
完成的功能及达到的性能

1. 波形可调（正弦波、三角波、方波）、频率可调（DC-）、幅度可调；
2. 生成模拟信号的频率范围为DC-20MHz，调节精度为1Hz（分辨率 0.028Hz * 36 = 1.0058Hz）；
3. 生成模拟信号的幅度为最大1Vpp，调节范围为0.1V-1V （0.09844V-1.00625V 分辨率 0.0109V）；
4. 在OLED上显示当前波形的形状、波形的频率以及幅度；
5. 利用板上旋转编码器和按键能够对波形进行切换、进行参数调节 （编码器顺时针旋转存在问题，加之以编码器按键辅助）。

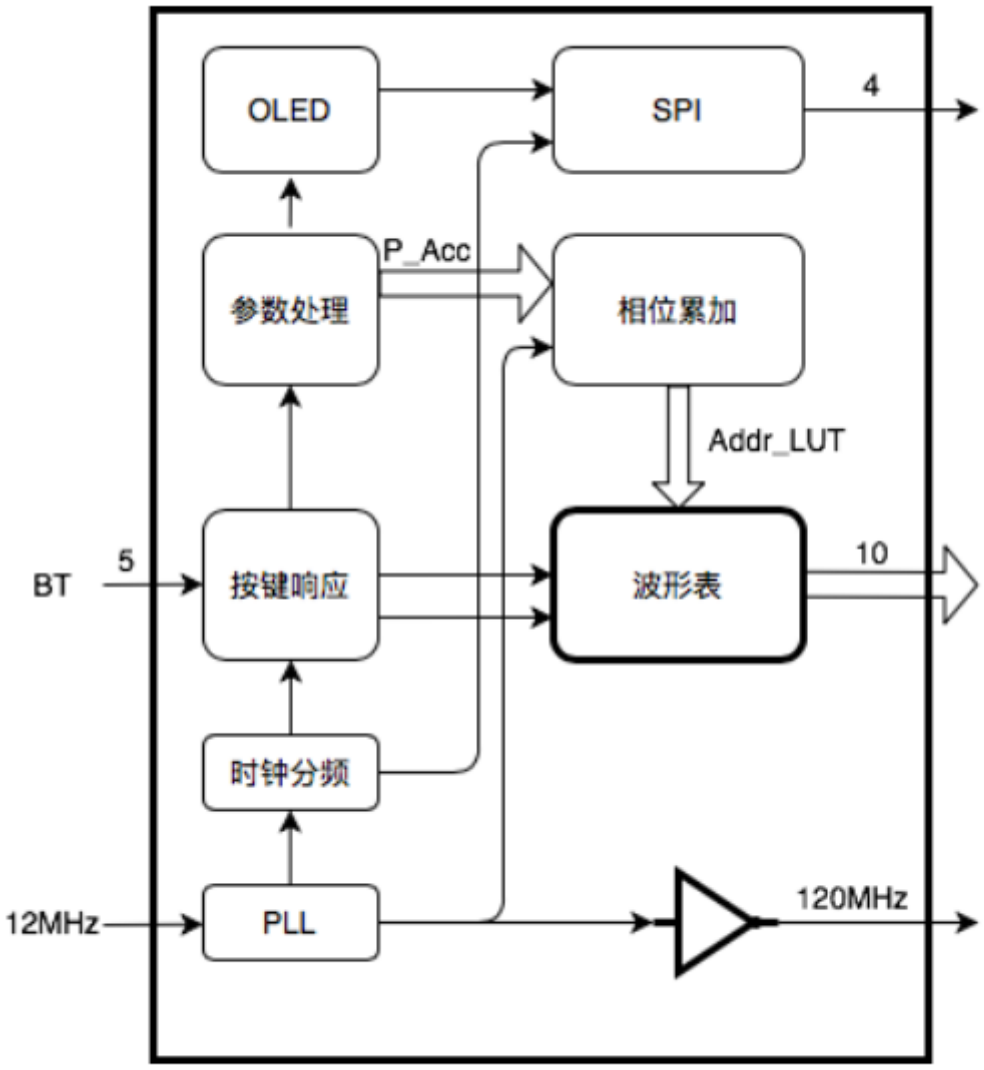
设计思路

本文通过按键输入模块（按键+旋转编码器）进行参数（波形/频率/幅值）输入，OLED模块直接将参数显示出来，同时通过DDS主逻辑将三个参数转换为对应的波形输出。

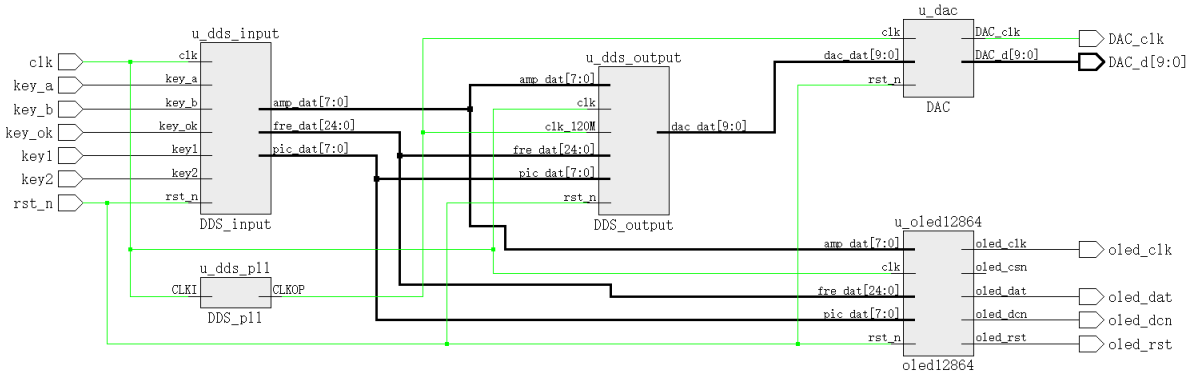
流程图

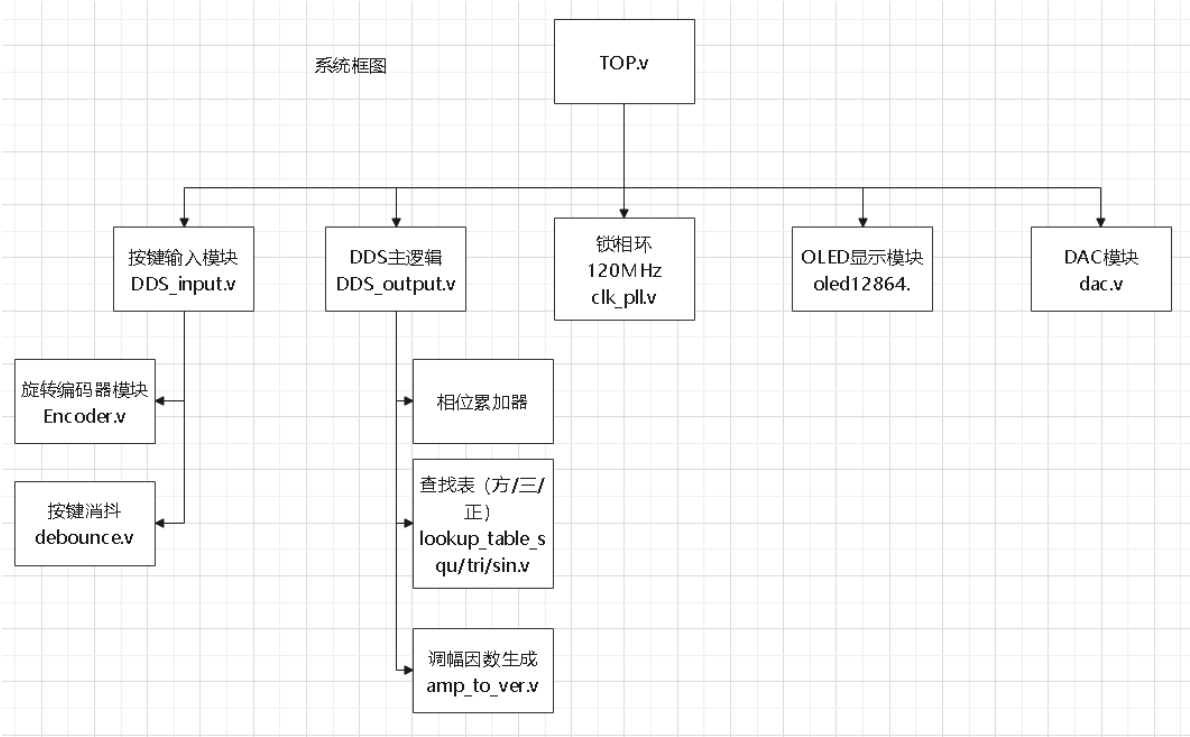

按键&旋转编码器
系统初始化
按键输入
循环
参数显示
输入参数读取
OLED模块
参数处理
dds主逻辑
信号生成
DAC模块

整体系统框架如下：



FPGA内部逻辑





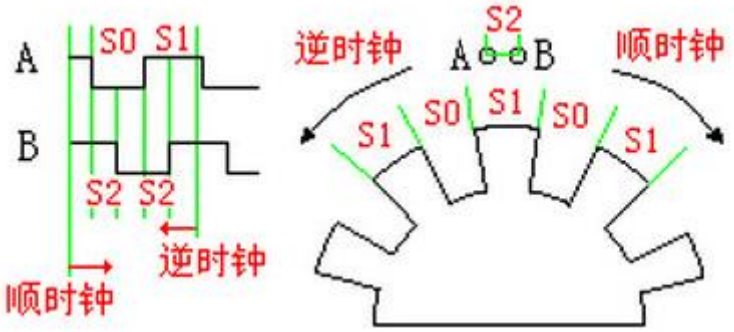
按键&编码器实现

按键

按键刚闭合的时候，会产生机械抖动。在FPGA逻辑中处理按键消抖的基本原理：将键检测信号bt_pushed作为计数器的清零型号，当没有键按下的时候，bt_pushed为高电平，计数器一直处于清零状态，当按键按下的时候，bt_pushed变为低电平，计数器在时钟信号bt_check_clock作用下进行计数。

编码器

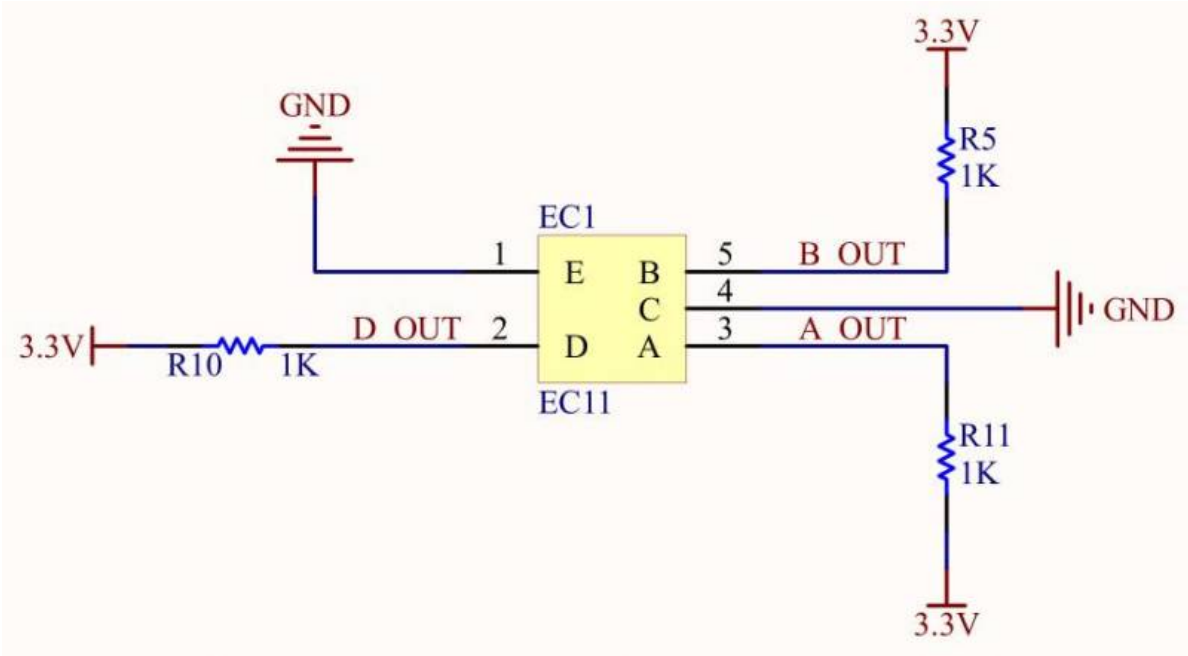
旋转编码器是一种位置传感器，它可以根据旋转运动生成的模拟或数字电信号来确定旋转轴的角位置。我们常用的旋转编码器是增量式编码器，是用来测量旋转的最简单的位置传感器。



如上图所示，当顺时针旋转时A信号提前B信号90度相位，当逆时针旋转时B信号提前A信号90度相位,FPGA接收到旋转编码器的A、B信号时，可以根据A、B的状态组合判定编码器的旋转方向。
程序设计中我们可以对A、B信号检测，检测A信号的边沿及B信号的状态，

- 当A信号上升沿时B信号为低电平，或当A信号下降沿时B信号为高电平，证明当前编码器为顺时针转动
- 当A信号上升沿时B信号为高电平，或当A信号下降沿时B信号为低电平，证明当前编码器为逆时针转动

本设计电路图如下：



DDS实现

波形控制

在DDS中，波形的最终输出由DAC决定，而波形的产生有多种方式，其中方波、三角波可通过计数器来实现，如下（代码来自电子森林）：

```
1  module dds_main(clk, dac_data, dac_clk);
2  input clk;                        //12MHz的外部时钟送给FPGA;
3  output [9:0] dac_data;              //10位的并行数据输出到R-2R DAC;
4  output dac_clk;                     //输出给DAC的并行时钟，在R-2R的DAC中不需要这个时钟信号
5
6  //创建一个24位的自由运行的二进制计数器，方便取出最高位（0.7秒一个周期）送给LED用作心跳灯指示用，在此程序中不列出LED的部分
7  //后面也会讲到为什么DDS波表的地址只有8～12位，而我们选用24位或32位相位累加器的原因
8  reg [23:0] cnt;
9  always @(posedge clk) cnt <= cnt + 24'h1;
10
11 //用它来产生DAC的信号输出
12 wire cnt_tap = cnt[7];               // 取出计数器的其中1位(bit 7 = 第8位)
13 assign dac_data = {10{cnt_tap}};    // 重复10次作为10位DAC的值
14 assign dac_clk= clk;
15
16 endmodule
```

在这个例子中，我们使用了计数器的第8位来输出，计数器的时钟为12MHz（未使用内部锁相环）的时候，第8位的翻转频率为12MHz/2^8 = 46.875KHz， 所以DAC的输出为46.875KHz的方波，使用计数器的其它位数，得到的方波的频率也会发生改变。

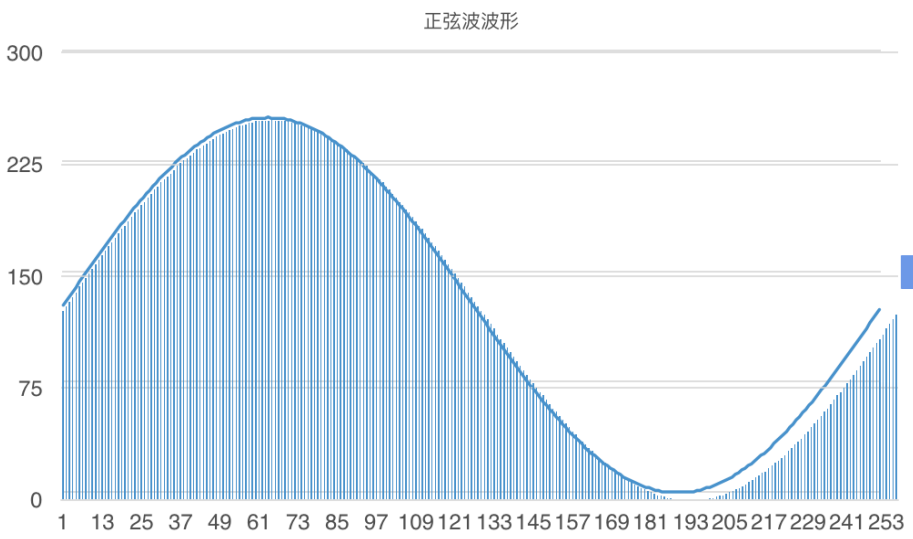
相应的锯齿波即将上述代码的 12、13行改为下述即可：

```
1  assign dac_data = cnt[9:0];
```

三角波则可用如下代码代替：

```
assign dac_data = cnt[10] ? ~cnt[9:0] : cnt[9:0];
```

而为了产生任意波形，则上述的方式不能胜任。故而改用LUT-Lookup Table（查找表）的方式，对波形进行采样，并保存在LUT中，按输入的地址进行波形数据的查找，在输出至DAC生成波形，并通过改变输出的速度来改变波形的频率。而本文对于方波、三角波、正弦波均采用查找表的方式进行波形输出，以方便进行统一调用。



地址	数据
0	7F
1	82
2	85
3	88
4	8B
5	8F
6	92
7	95
8	98
9	9B
10	9E
11	A1
12	A4
13	A7
14	AA
15	AD
16	B0
17	B3
18	B6

频率控制

而对于频率，DDS则通过相位累加器实现对其的控制。就实质而言，计数器即相位累加器，对一个8位计数器，其每一次累加，就会移动波形 360°/256 ≈ 1.4° 的相位。

```
reg [7:0] phase_acc;   // 8位
always @(posedge clk) phase_acc <= phase_acc + 8'h1;
lookup_tables u_lookup_table(.phase(phase_acc[7:0]), .sin_out(dac_data));
```

上述代码实现了频率为1/（256*clk*）*Hz*的波形的输出，若将每次相位的累加值改为*2*，则频率为*2/*（*256*clk）Hz'，以此类推 x3，x4.....

综上，可以得知，输出波形的频率与相位累加器位数、时钟频率、累加值三者相关，具体公式为：

$$频率 = \frac{累加值}{时钟频率 * 2^{相位累加器位数}}$$

本文中DDS的频率参数精度为1Hz，那么为了准确的数值，相位累加器位数需要够高，同时生成的频率误差要合适。故而采用了 32 为相位累加器 + 120MHz 的时钟信号，其分辨率为 120M/2^32 约等于 0.028Hz ，为了达到1Hz的精度，则累加值取 36 （1/0.028 ≈ 35.79）.

而其频率输出上限，受制于时钟频率，采样频率。对于120MHz的时钟，上限频率一般为20MHz，正好符合0 ~20 MHz的调频要求。

幅度控制

信号发生器通常采用“DAC参考电压”配合“模拟通道信号调理”进行信号幅值的调节，市面上大多数信号发生器产品都是采用这种调幅方案。这样在调节的过程中信号的分辨率不会受影响，最大程度保证信号的性能指标，同时也需要额外的DAC等电路控制参考电压。

另外我们也可以在FPGA中增加对信号幅值调节的设计，例如我们将信号的数据乘以一个8bit的因数，然后再将结果右移8位（相当于除以256），然后再把结果输出给DAC电路。将8bit的因数作为变量可调，最后就实现了在FPGA中调节幅值的功能。这种方法不需要额外的DAC改变参考电压，即可实现对幅值的调节，但是由于采用量化后的数据进行，会造成信号数据分辨率的降低。

```
wire [9:0] signal_dat; //未调幅的波形数据
wire [7:0] a_ver; //用于调幅的因数

reg [17:0] amp_dat; //调幅后的波形数据
always @(posedge clk) amp_dat = signal_dat * a_ver;  //波形数据乘以调幅因数

wire [9:0] dac_dat; //输出给DAC电路的数据端口
assign dac_dat = amp_dat[17:8]; //取高十位输出，相当于右移8位
```

而本文中对于幅值的要求为 最大 1 Vpp，精度为 0.1~1V，则最终的转换公式如下：

$$幅值 = \frac{调幅因数}{256} * 波形数据$$

而最终的电压输出、调幅因数的选择取决于ADC芯片参数以及实际测量结果。

代码实现

DDS顶层

代码主要思路为利用有限状态机，将整个DDS输出分为三个状态——方波输出、三角波输出及正弦波输出。并在对应状态下，通过频率、幅值数据对查找表的波形数据进行调整，使其频率、幅值符合输入要求。

该模块中，pic_dat、fre_dat、amp_dat对应波形、频率、幅值输入数据，而dac_dat即为调整后的波形数据，将其传递给DAC模块实现波形可调、频率可调、幅度可调波形的生成。原理见DDS实现部分。

```
module DDS_output(clk,clk_120M,rst_n,pic_dat,fre_dat,amp_dat,dac_dat);
input clk,clk_120M,rst_n;//120MHz/12MHz
input  [7:0]    pic_dat,amp_dat;
input  [24:0]  fre_dat;

output reg [9:0] dac_dat;
wire [9:0]  sin_dat,squ_dat,tri_dat;
reg [31:0] phase_acc;//32位相位累加器，精度0.028Hz

always @(posedge clk_120M) phase_acc <= phase_acc + fre_dat * 24'd36;   //在120MHz的主时钟时，输出对应频率的波形
//未调幅的波形数据
lookup_tables_sin u_lookup_table_sin(.phase(phase_acc[31:24]),.sin_out(sin_dat)); //波形查找表 正弦波
lookup_tables_squ u_lookup_table_squ(.phase(phase_acc[31:24]),.squ_out(squ_dat)); //波形查找表 方波
lookup_tables_tri u_lookup_table_tri(.phase(phase_acc[31:24]),.tri_out(tri_dat)); //波形查找表 三角波

wire [7:0] a_ver; //用于调幅的因数
reg [17:0] sin_data; //调幅后的波形数据
reg [17:0] squ_data; //调幅后的波形数据
reg [17:0] tri_data; //调幅后的波形数据
amp_to_ver u_mp2ver(amp_dat,a_ver);
```

```
always @(posedge clk) begin
    sin_data = sin_dat * a_ver;  //波形数据乘以调幅因数
    squ_data = squ_dat * a_ver;
    tri_data = tri_dat * a_ver;
end


always @(posedge clk or negedge rst_n)begin
    case(pic_dat)
        8'd12:begin dac_dat <= squ_data[17:8]; end//方波
        8'd20:begin dac_dat <= tri_data[17:8]; end//三角波
        8'd28:begin dac_dat <= sin_data[17:8]; end//正弦波
    endcase
    end
endmodule
```

波形表

方波、三角波思路与下方正弦波一致。

```
module sin_table(address,sin);
output [8:0] sin;          //实际波形表为9位分辨率（1/4周期）
input  [5:0] address;      //64个点来生成1/4个周期的波形，完整的一个周期为256个点

    reg    [8:0] sin;//正弦波波形，方波，三角波同一思路实现统一调配。

always @(address)
    begin
                case(address)
                    6'h0: sin=9'h0;
                    6'h1: sin=9'hC;
                    6'h2: sin=9'h19;
                    6'h3: sin=9'h25;
                    6'h4: sin=9'h32;
                    6'h5: sin=9'h3E;
                    6'h6: sin=9'h4B;
                    6'h7: sin=9'h57;
                    6'h8: sin=9'h63;
                    6'h9: sin=9'h70;
                    6'ha: sin=9'h7C;
                    6'hb: sin=9'h88;
                    6'hc: sin=9'h94;
                    6'hd: sin=9'hA0;
                    6'he: sin=9'hAC;
                    6'hf: sin=9'hB8;
                    6'h10: sin=9'hC3;
                    6'h11: sin=9'hCF;
                    6'h12: sin=9'hDA;
                    6'h13: sin=9'hE6;
                    6'h14: sin=9'hF1;
                    6'h15: sin=9'hFC;
                    6'h16: sin=9'h107;
                    6'h17: sin=9'h111;
                    6'h18: sin=9'h11C;
                    6'h19: sin=9'h126;
                    6'h1a: sin=9'h130;
                    6'h1b: sin=9'h13A;
```

```
                    6'h1c: sin=9'h144;
                    6'h1d: sin=9'h14E;
                    6'h1e: sin=9'h157;
                    6'h1f: sin=9'h161;
                    6'h20: sin=9'h16A;
                    6'h21: sin=9'h172;
                    6'h22: sin=9'h17B;
                    6'h23: sin=9'h183;
                    6'h24: sin=9'h18B;
                    6'h25: sin=9'h193;
                    6'h26: sin=9'h19B;
                    6'h27: sin=9'h1A2;
                    6'h28: sin=9'h1A9;
                    6'h29: sin=9'h1B0;
                    6'h2a: sin=9'h1B7;
                    6'h2b: sin=9'h1BD;
                    6'h2c: sin=9'h1C3;
                    6'h2d: sin=9'h1C9;
                    6'h2e: sin=9'h1CE;
                    6'h2f: sin=9'h1D4;
                    6'h30: sin=9'h1D9;
                    6'h31: sin=9'h1DD;
                    6'h32: sin=9'h1E2;
                    6'h33: sin=9'h1E6;
                    6'h34: sin=9'h1E9;
                    6'h35: sin=9'h1ED;
                    6'h36: sin=9'h1F0;
                    6'h37: sin=9'h1F3;
                    6'h38: sin=9'h1F6;
                    6'h39: sin=9'h1F8;
                    6'h3a: sin=9'h1FA;
                    6'h3b: sin=9'h1FC;
                    6'h3c: sin=9'h1FD;
                    6'h3d: sin=9'h1FE;
                    6'h3e: sin=9'h1FF;
                    6'h3f: sin=9'h1FF;
                endcase
            end
endmodule
```

查找表

根据phase（相位累加器的高8位），进行波形查找。由于波形表波形一个周期256个采样点，1/4个周期64个采样点，故而取phase的高2位划分状态，共计四个状态构成一个完整周期，并对波形数据进行处理，使四个状态共同构成一个完整的周期信号。

```
module lookup_tables_sin(phase, sin_out);
input   [7:0]   phase;
output  [9:0]   sin_out;

wire     [9:0]   sin_out;

reg     [5:0]   address;
wire    [1:0]   sel;
wire    [8:0]   sine_table_out;

reg     [9:0]   sine_onecycle_amp;
```

```
12
13  //assign sin_out = {4'b0, sine_onecycle_amp[9:4]} + 9'hff;  // 可以调节输出信号的幅度
14  assign sin_out = sine_onecycle_amp[9:0];
15
16  assign sel = phase[7:6];
17
18  sin_table u_sin_table(address,sine_table_out);
19
20  always @(sel or sine_table_out)
21  begin
22      case(sel)
23      2'b00:  begin
24              sine_onecycle_amp = 9'h1ff + sine_table_out[8:0];
25              address = phase[5:0];
26              end
27      2'b01:  begin
28              sine_onecycle_amp = 9'h1ff + sine_table_out[8:0];
29              address = ~phase[5:0];
30              end
31      2'b10:  begin
32              sine_onecycle_amp = 9'h1ff - sine_table_out[8:0];
33              address = phase[5:0];
34              end
35      2'b11:  begin
36              sine_onecycle_amp = 9'h1ff - sine_table_out[8:0];
37              address = ~ phase[5:0];
38              end
39      endcase
40  end
41
42  endmodule
```

按键&编码器

主模块

主要思想为 按键1 控制输入参数的位数（针对频率：个、十、百.....），按键2 实现 波形/频率/幅值 调节的模式转换。编码器顺时针旋转对应的数据累加，逆时针相反。同时，由于编码器顺时钟旋转存在问题（视频中说明），便将编码器的按键逻辑设计为与顺时钟旋转相同。消抖及编码器逻辑参考电子森林教程。

```
1   module
    DDS_input(clk,rst_n,key1,key2,key_a,key_b,key_ok,pic_dat,amp_dat,fre_dat);
2
3   input             clk;
4   input             rst_n;
5   input             key1,key2;
6   input             key_a,key_b,key_ok;
7
8   output      [7:0] pic_dat,amp_dat;
9   output      [24:0] fre_dat;
10  wire       key1_pulse,key2_pulse,Right_paulse,Left_paulse,OK_paulse;
11  reg [7:0]   pic_dat,amp_dat;
12  reg [24:0]  fre_dat;
13  //状态计数器，0~2计数，共三个状态：波形、频率、幅度，key2控制
```

```
reg [2:0] cnt_state;
always @(posedge clk  or  negedge rst_n)
     begin
       if (!rst_n) begin
            cnt_state <= 2'd0;
            end
       else if (key2_pulse) begin
            cnt_state <= cnt_state + 2'd1;
            end
       else if(cnt_state >= 3)
           cnt_state <= 2'd0;
       else begin
           cnt_state <= cnt_state;
           end
       end

reg [3:0] flag;//单位计数器，用以控制调频时的单位，如调整个位，每次频率+1Hz，计数器置0，每次个位数值+1
always @(posedge clk  or  negedge rst_n)
     begin
       if (!rst_n) begin
            fre_dat <= 25'd0;
            amp_dat <= 8'd0;
            pic_dat <= 8'd12;
            flag <= 3'd0;
            end
    else    if(fre_dat > 25'd20_000_000)//频率大于20MHz，清零处理
        fre_dat <= 25'd0;
    else if (Left_pulse) begin//编码器左转，减数处理
            case(cnt_state)
                2'd0:begin  fre_dat <= fre_dat;amp_dat <= amp_dat;end//波形状态，频率、幅度数值不变
                2'd1:begin//调频状态
                if(flag == 3'd0)//根据单位计数器状态，进行数据处理：0-个位、1-十位，以此类推。
                fre_dat <= fre_dat-25'd1;
                else if(flag == 3'd1)
                fre_dat <= fre_dat-25'd10;
                else if(flag == 3'd2)begin
                fre_dat <= fre_dat-25'd100;
                end

                else if(flag == 3'd3)begin
                fre_dat <= fre_dat-25'd1_000;
                end
                else if(flag == 3'd4)begin
                fre_dat <= fre_dat-25'd10_000;
                end
                else if(flag == 3'd5)begin
                fre_dat <= fre_dat-25'd100_000;
                end

                else if(flag == 3'd6)begin
                fre_dat <= fre_dat-25'd1_000_000;
                end
                else if(flag == 3'd7)begin
                fre_dat <= fre_dat-25'd10_000_000;
                end
```

```
                else begin
                fre_dat <= fre_dat;
                end
                end
                2'd2:begin if(amp_dat > 8'd0)amp_dat <= amp_dat-8'd1; else amp_dat <= amp_dat; end//调幅状态
                default:begin   fre_dat <= fre_dat;amp_dat <= amp_dat; end
            endcase
        end
       else if (Right_pulse || OK_pulse) begin//编码器右旋/按键，执行加数处理，下逻辑同上（-改为+）。右旋会同时出现左&右脉冲导致逻辑执行不稳定，故加按键以辅助。
            case(cnt_state)
                2'd0:begin  fre_dat <= fre_dat;amp_dat <= amp_dat;pic_dat <= pic_dat + 8'd8; end//波形状态，频率、幅度数值不变，按键执行波形切换
                2'd1:begin
                if(flag == 3'd0)begin
                fre_dat <= fre_dat+25'd1;
                end
                else if(flag == 3'd1)begin
                fre_dat <= fre_dat+25'd10;
                end
                else if(flag == 3'd2)begin
                fre_dat <= fre_dat+25'd100;
                end

                else if(flag == 3'd3)begin
                fre_dat <= fre_dat+25'd1_000;
                end
                else if(flag == 3'd4)begin
                fre_dat <= fre_dat+25'd10_000;
                end
                else if(flag == 3'd5)begin
                fre_dat <= fre_dat+25'd100_000;
                end

                else if(flag == 3'd6)begin
                fre_dat <= fre_dat+25'd1_000_000;
                end
                else if(flag == 3'd7)begin
                fre_dat <= fre_dat+25'd10_000_000;
                end
                else begin
                fre_dat <= fre_dat;
                end
                end
                2'd2:begin if(amp_dat <= 8'd9)amp_dat <= amp_dat+8'd1; else amp_dat <= amp_dat; end
                default:begin   fre_dat <= fre_dat;amp_dat <= amp_dat; end
            endcase
        end
        else if (key1_pulse)begin//单位计数器，key2控制
                if(flag <= 3'd6) begin
                    flag <= flag + 3'd1;
                    end
                else
                    flag <= 3'd0;
            end
        else if(pic_dat >= 8'd29)
```

```
                pic_dat <= 8'd12;
        else begin
            fre_dat <= fre_dat;
            amp_dat <= amp_dat;
            pic_dat <= pic_dat;
            flag <= flag;
            end
       end

//例化消抖module，这里没有传递参数N，采用了默认的N=1
debounce  u1 (
.clk (clk),
.rst (rst_n),
.key (key1),
.key_pulse (key1_pulse)
);

debounce  u2 (
.clk (clk),
.rst (rst_n),
.key (key2),
.key_pulse (key2_pulse)
);

Encoder u3//编码器模块
(
.clk_in(clk),            //系统时钟
.rst_n_in(rst_n),        //系统复位，低有效
.key_a(key_a),           //旋转编码器A管脚
.key_b(key_b),           //旋转编码器B管脚
.key_ok(key_ok),             //旋转编码器D管脚
.Left_pulse(Left_pulse),         //左旋转脉冲输出
.Right_pulse(Right_pulse),  //右旋转脉冲输出
.OK_pulse(OK_pulse)      //按动脉冲输出
);

endmodule
```

编码器

按键逻辑与普通按键相同，注意消抖即可。旋转编码部分，主要利用key_a & key_b 之间的90°相位差，确定了四个状态，两两对应 顺时针/逆时针，当期旋转时进行状态检定，产生对应的脉冲即可得到输入。具体原理参考电子森林。

```
// ----------------------------------------------------------------------
// >>>>>>>>>>>>>>>>>>>>>>>> COPYRIGHT NOTICE <<<<<<<<<<<<<<<<<<<<<<<<<
// ----------------------------------------------------------------------
// Module: Encoder
//
// Author: Step
//
// Description: Driver for rotary encoder
//
// Web: www.stepfapga.com
//
// ----------------------------------------------------------------------
// Code Revision History :
```

```
// ----------------------------------------------------------------------
// Version: |Mod. Date:   |Changes Made:
// V1.0     |2016/04/20   |Initial ver
// ----------------------------------------------------------------------
module Encoder
(
input                   clk_in,         //系统时钟
input                   rst_n_in,       //系统复位，低有效
input                   key_a,          //旋转编码器A管脚
input                   key_b,          //旋转编码器B管脚
input                   key_ok,         //旋转编码器D管脚
output  reg             Left_pulse,     //左旋转脉冲输出
output  reg             Right_pulse,    //右旋转脉冲输出
output                  OK_pulse        //按动脉冲输出
);

localparam              NUM_500US   =   6_000;

reg             [12:0]  cnt;
//计数器周期为500us，控制键值采样频率
always@(posedge clk_in or negedge rst_n_in) begin
    if(!rst_n_in) cnt <= 0;
    else if(cnt >= NUM_500US-1) cnt <= 1'b0;
    else cnt <= cnt + 1'b1;
end

reg             [5:0]   cnt_20ms;
reg                     key_a_r,key_a_r1;
reg                     key_b_r,key_b_r1;
reg                     key_ok_r;

//针对A、B、D管脚分别做简单去抖操作，
//如果对旋转编码器的要求比较高，建议现对旋转编码器的输出做严格的消抖处理后再来做旋转编码器的驱动
//对旋转编码器的输入缓存，消除亚稳态同时延时锁存
always@(posedge clk_in or negedge rst_n_in) begin
    if(!rst_n_in) begin
        key_a_r     <=  1'b1;
        key_a_r1    <=  1'b1;
        key_b_r     <=  1'b1;
        key_b_r1    <=  1'b1;
        cnt_20ms    <=  1'b1;
        key_ok_r    <=  1'b1;
    end else if(cnt == NUM_500US-1) begin
        key_a_r     <=  key_a;
        key_a_r1    <=  key_a_r;
        key_b_r     <=  key_b;
        key_b_r1    <=  key_b_r;
        if(cnt_20ms >= 6'd40) begin //对于按键D信号还是采用20ms周期采样的方法，40*500us = 20ms
            cnt_20ms <= 6'd0;
            key_ok_r <= key_ok;
        end else begin
            cnt_20ms <= cnt_20ms + 1'b1;
            key_ok_r <= key_ok_r;
        end
    end
end
```

```
70
71  reg                     key_ok_r1;
72  //对按键D信号进行延时锁存
73  always@(posedge clk_in or negedge rst_n_in) begin
74      if(!rst_n_in) key_ok_r1 <= 1'b1;
75      else key_ok_r1 <= key_ok_r;
76  end
77
78  wire    A_state     = key_a_r1 && key_a_r && key_a; //旋转编码器A信号高电平状态
    检测
79  wire    B_state     = key_b_r1 && key_b_r && key_b; //旋转编码器B信号高电平状态
    检测
80  assign  OK_pulse    = key_ok_r1 && (!key_ok_r);     //旋转编码器D信号下降沿检测
81
82  reg                     A_state_reg;
83  //延时锁存
84  always@(posedge clk_in or negedge rst_n_in) begin
85      if(!rst_n_in) A_state_reg <= 1'b1;
86      else A_state_reg <= A_state;
87  end
88
89  //旋转编码器A信号的上升沿和下降沿检测
90  wire    A_pos   = (!A_state_reg) && A_state;
91  wire    A_neg   = A_state_reg && (!A_state);
92
93  //通过旋转编码器A信号的边沿和B信号的电平状态的组合判断旋转编码器的操作，并输出对应的脉
    冲信号
94  always@(posedge clk_in or negedge rst_n_in)begin
95      if(!rst_n_in)begin
96          Right_pulse <= 1'b0;
97          Left_pulse <= 1'b0;
98      end else begin
99          if(A_pos && B_state) Left_pulse <= 1'b1;
100         else if(A_neg && B_state) Right_pulse <= 1'b1;
101         else begin
102             Right_pulse <= 1'b0;
103             Left_pulse <= 1'b0;
104         end
105     end
106 end
107
108 endmodule
```

顶层例化

顶层模块主要有 5个输入端口：key_1&2 ,key_a&B&ok，15个输出端口：OLED SPI协议端口，以及 DAC 时钟 、10位DAC输入端口。其中输入端口外界数据读取进主要模块。OLED端口输出显示信息，DAC端口输出数据对应的波形数据，并经由DAC模块生成对应波形。

```
1  module TOP_dds (clk,rst_n,
2  key1,key2,
3  key_a,key_b,key_ok,
4  oled_rst,oled_dcn,oled_clk,oled_dat,
5  DAC_clk,DAC_d);
6
7  input               clk;//系统时钟，12MHz
8  input               rst_n;//系统复位
```

```
input             key1,key2; //按键输入
input             key_a,key_b,key_ok;//编码器输入

output            DAC_clk;//DAC时钟，120MHz
    output   [9:0]    DAC_d;//DAC 输出
output            oled_rst,oled_dcn,oled_clk,oled_dat;//oled输出

wire            key1_pulse,key2_pulse,Right_paulse,Left_paulse,OK_paulse;
wire    [7:0]   pic_dat,amp_dat;//波形、幅值数据
wire    [24:0]  fre_dat;//频率数据
wire    [9:0]   dac_dat;//dac数据，10bit



//按键&编码器输入
DDS_input u_dds_input(
.clk(clk),
.rst_n(rst_n),
.key1(key1),
.key2(key2),
.key_a(key_a),
.key_b(key_b),
.key_ok(key_ok),
.pic_dat(pic_dat),
.amp_dat(amp_dat),
.fre_dat(fre_dat)
);

//oled显示输出
oled12864 u_oled12864(
    .clk(clk),      //系统时钟
    .rst_n(rst_n),      //系统复位

    .amp_dat(amp_dat),
    .fre_dat(fre_dat),
    .pic_dat(pic_dat),
//  .debug(debug),

    .oled_csn(),    //OLED 使能
    .oled_rst(oled_rst),    //OLED 复位
    .oled_dcn(oled_dcn),    //OLED 数据/命令 控制
    .oled_clk(oled_clk),    //OLED 时钟
    .oled_dat(oled_dat) //OLED 数据
);

wire clk_pll;//锁相环输出 120MHz
//DAC模块输出
DAC u_dac(
.clk(clk_pll),//120MHz
.rst_n(rst_n),
.dac_dat(dac_dat),
.DAC_clk(DAC_clk),
.DAC_d(DAC_d)
);
//锁相环，12MHz 转 120MHz
DDS_pll u_dds_pll(
.CLKI(clk),
.CLKOP(clk_pll)
```

```
);
//DDS参数生成
DDS_output u_dds_output(
.clk(clk),//12MHz
.clk_120M(clk_pll),//120MHz
.rst_n(rst_n),
.pic_dat(pic_dat),
.fre_dat(fre_dat),
.amp_dat(amp_dat),
.dac_dat(dac_dat));//dac数据输出，用以数模转换，生成最终波形
 endmodule
```

其他

资源使用报告

由下图可知，虽未使用全部资源，但在资源利用率上还有可进步的空间，例如：方波、三角波的生成可不采用查找表的方式而是利用计数器以减少资源使用，该方面内容需纳入后续考量中。

```
Design Summary
   Number of registers:    509 out of  4635 (11%)
      PFU registers:           509 out of  4320 (12%)
      PIO registers:             0 out of   315 (0%)
   Number of SLICEs:      1244 out of  2160 (58%)
      SLICEs as Logic/ROM:   1244 out of  2160 (58%)
      SLICEs as RAM:            0 out of  1620 (0%)
      SLICEs as Carry:        321 out of  2160 (15%)
   Number of LUT4s:        2484 out of  4320 (58%)
      Number used as logic LUTs:         1842
      Number used as distributed RAM:       0
      Number used as ripple logic:        642
      Number used as shift registers:       0
   Number of PIO sites used: 22 + 4(JTAG) out of 105 (25%)
   Number of block RAMs:  0 out of 10 (0%)
   Number of GSRs:        1 out of 1 (100%)
   EFB used :         No
   JTAG used :        No
   Readback used :    No
   Oscillator used :  No
   Startup used :     No
   POR :              On
   Bandgap :          On
   Number of Power Controller:  0 out of 1 (0%)
   Number of Dynamic Bank Controller (BCINRD):  0 out of 6 (0%)
   Number of Dynamic Bank Controller (BCLVDSO):  0 out of 1 (0%)
   Number of DCCA:  0 out of 8 (0%)
   Number of DCMA:  0 out of 2 (0%)
   Number of PLLs:  1 out of 2 (50%)
   Number of DQSDLLs:  0 out of 2 (0%)
   Number of CLKDIVC:  0 out of 4 (0%)
   Number of ECLKSYNCA:  0 out of 4 (0%)
   Number of ECLKBRIDGECS:  0 out of 2 (0%)
```

主要难题及解决方法

1、频率参数位数过多，数据显示及调参模块难以编写。

改写了BCD译码模块使其支持28bit二级制数转8位十进制数（10_000_000），并改写了UI及数据调整逻辑；

2、对于DDS生成波形的检测以及相位累加器、调幅因数的数值确认。

理论结合实践，对波形进行调整，虽然大致达到要求，但受限与仪器、经验，无法达到更好的效果。

未来的计划

通过本次实践，认识并尝试了DDS任意信号发生器的实现，受益良多。计划在后续的学习生活中加深对该模块的学习，并深入了解其应用场景，尝试将其投入实际场景。并尝试提高已实现的DDS信号发生器的精度。