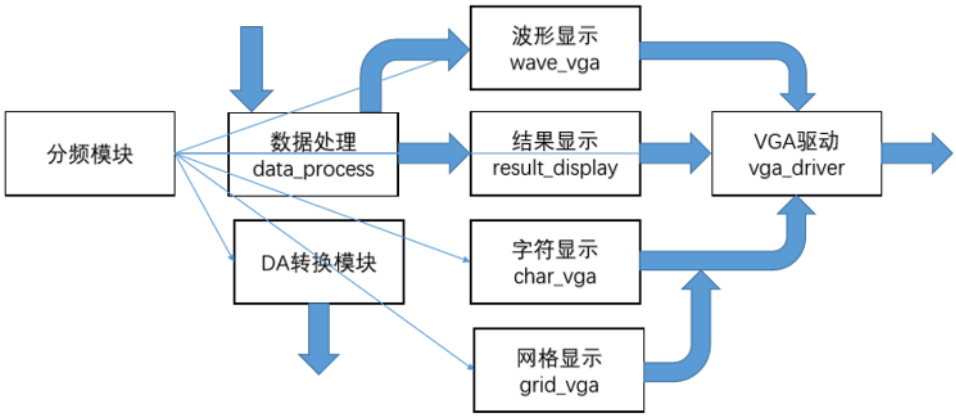
数字存储示波器

一， 设计概述

利用黑金 FPGA 开发板 AX515 进行数字存储示波器的设计，利用 ALTERA 公司的 CYCLONE IV芯片作为主控。读取信号源电压，利用模数转换芯片 AD9280 芯片实现 AD 转换并用双口 ram 进行数据存储，最后利用 VGA 显示，实现了波形的横纵轴缩放、自动测量与手动光标测量幅值与频率等功能。

二， 结构框图

本设计主要包括分频模块、字符显示模块、网格显示模块、数据处理模块、波形显示模块、结果显示模块，VGA 驱动模块以及 DA 转换模块组成，结构框图如下：



三， 模块设计

1，分频模块

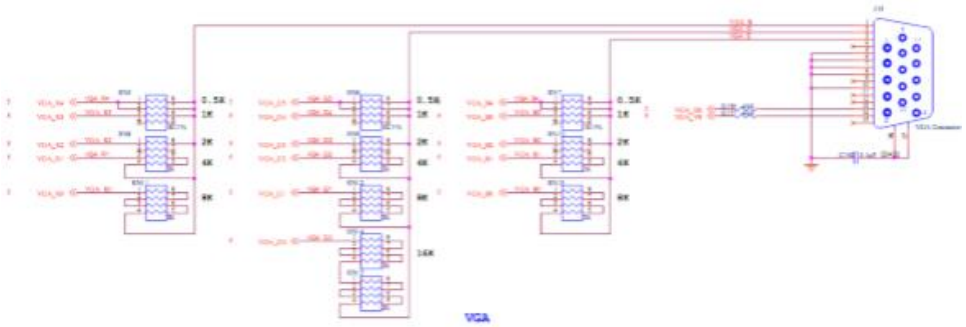
分频模块利用 quartus 自带 IP 核中的 PLL 模块，将晶振频率 50MHz 分频至 VGA 显示的 40MHz，数模转换芯片 AD9708 的 125MHz，以及模数转换芯片 AD9280 的 32MHz，其中 pll_adda 实现 40MHz 至 125MHz 与 32MHz 的分频，pll_display 实现 50MHz 至 40MHz 的分频。顶层模块 Oscill 中对于两个锁相环的例化如下：

```
pll_display pd_u1(
 .inclk0(clk),
 .c0(clk_40M)
);
pll_adda pa_u1(
 .inclk0(clk),
 .c0(clk_32M),
 .c1(clk_125M)
);
```

2，VGA 驱动模块 driver_vga

VGA 驱动时序主要包括行扫描与场扫描，每个行扫描周期包括同步时间、消隐后肩、显示期与消隐前肩，逐行扫描完一个屏幕则称为一个场，场的时序同样包括同步时间、消隐后肩、显示期与消隐前肩，利用 Verilog 语言实现行与场的时序控制，并控制在显示期的颜色输入，即可控制 VGA 显示。

本设计采用 600*800 的分辨率显示，其标准参数在之后的代码中利用 parameter 声明。色彩部分根据开发板接口采用 R：G：B = 5：6：5 的 16 位数据输入，VGA 接口部分接口图如下：



VGA 驱动模块端口主要包括 40MHz 时钟输入、复位信号输入、RGB 数据输入，RGB 数据输出、行同步、场同步信号输出、已经像素横纵坐标输出。

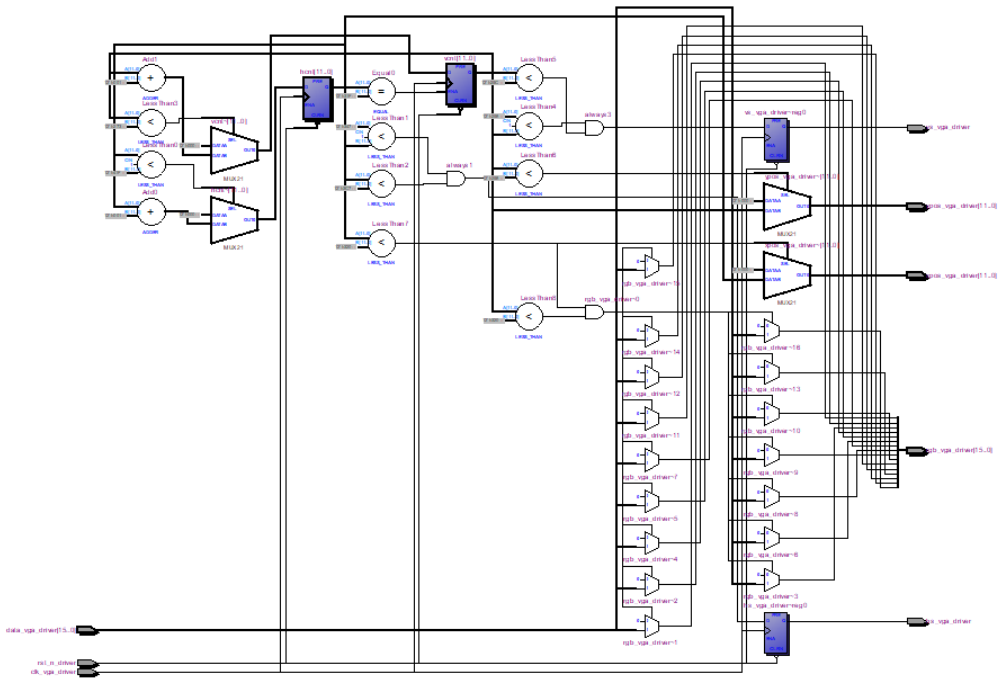
模块代码如下：

```
module driver_vga (
    input              clk_vga_driver,      //40MHz 像素时钟
    input              rst_n_driver,
    input [15:0]   data_vga_driver,      //RGB565 格式

    output [15:0] rgb_vga_driver,
    output reg      hs_vga_driver,      //行同步
    output reg      vs_vga_driver,      //场同步
    output [11:0] xpos_vga_driver,    //横坐标
    output [11:0] ypos_vga_driver     //纵坐标
);

//++++++++++++++++++++++++++++++++++
//标准参数
//++++++++++++++++++++++++++++++++++

//800*600 60Hz 40MHz
parameter    H_DISP    = 12'd800 ,
             H_FRONT = 12'd40    ,
             H_SYNC    = 12'd128 ,
             H_BACK    = 12'd88    ,
```

```
                H_TOTAL = 12'd1056,

                V_DISP  = 12'd600,
                V_FRONT = 12'd1  ,
                V_SYNC  = 12'd4  ,
                V_BACK  = 12'd23 ,
                V_TOTAL = 12'd628;

    //+++++++++++++++++++++++++++++++++
    //行 场 同步信号
    //+++++++++++++++++++++++++++++++++

    reg [11:0] hcnt;      //行计数
    always @(posedge clk_vga_driver or negedge rst_n_driver)
    begin
        if (!rst_n_driver) begin
          hcnt <= 12'd0;
     end
          else begin
           if (hcnt <= H_TOTAL - 12'd1)
                hcnt <= hcnt + 12'd1;
           else
                 hcnt <= 12'd0;
     end
    end

    always @(posedge clk_vga_driver or negedge rst_n_driver)
    begin
          if (!rst_n_driver)
                hs_vga_driver <= 1'b0;
          else
                begin
                     if (hcnt >= H_DISP + H_FRONT - 12'd1 && hcnt < H_DISP
+H_FRONT + H_SYNC - 12'd1)
                          hs_vga_driver <= 1'b1;
                     else
                          hs_vga_driver <= 1'b0;
                end
    end

    reg [11:0] vcnt;      //场计数
    always @(posedge clk_vga_driver or negedge rst_n_driver)
    begin
          if (!rst_n_driver)
```

```
            vcnt <= 12'd0;
        else if (hcnt == H_DISP - 12'd1)  //每扫描完一行，列扫描信号+1
            begin
                if (vcnt < V_TOTAL - 12'd1)
                    vcnt <= vcnt + 12'd1;
                else
                    vcnt <= 12'd0;
            end
        else
            vcnt <= vcnt;
    end

    always @ (posedge clk_vga_driver or negedge rst_n_driver)
    begin
        if (!rst_n_driver)
            vs_vga_driver <= 1'b0;
        else
            begin
                if (vcnt >= V_DISP + V_FRONT - 12'd1 && vcnt < V_DISP +
V_FRONT + V_SYNC - 12'd1)
                    vs_vga_driver <= 1'b1;
                else
                    vs_vga_driver <= 1'b0;
            end
    end

    //+++++++++++++++++++++++++++++++++
    //行列坐标
    //+++++++++++++++++++++++++++++++++

    assign xpos_vga_driver = (hcnt < H_DISP) ? hcnt : 12'd0;
    assign ypos_vga_driver = (vcnt < V_DISP) ? vcnt : 12'd0;
    assign rgb_vga_driver   = (hcnt < H_DISP && vcnt < H_DISP) ? data_vga_driver :
16'd0;

    //如果不在显示时期，横纵坐标以及颜色信号输出置 0；

    endmodule
```

RTL 视图如下：



3， 网格显示模块 grid_vga

网格显示模块作用为建立示波器的背景，包括网格与背景颜色。利用从 VGA 驱动模块得到的 xy 横纵坐标值，在相应位置输出指定的颜色，并输出给 VGA 驱动模块。输入端口包括时钟复位输入、横纵左边输入、网格与背景颜色输入，输出为 RGB 数据。显示设计如下图所示：

模块代码如下：

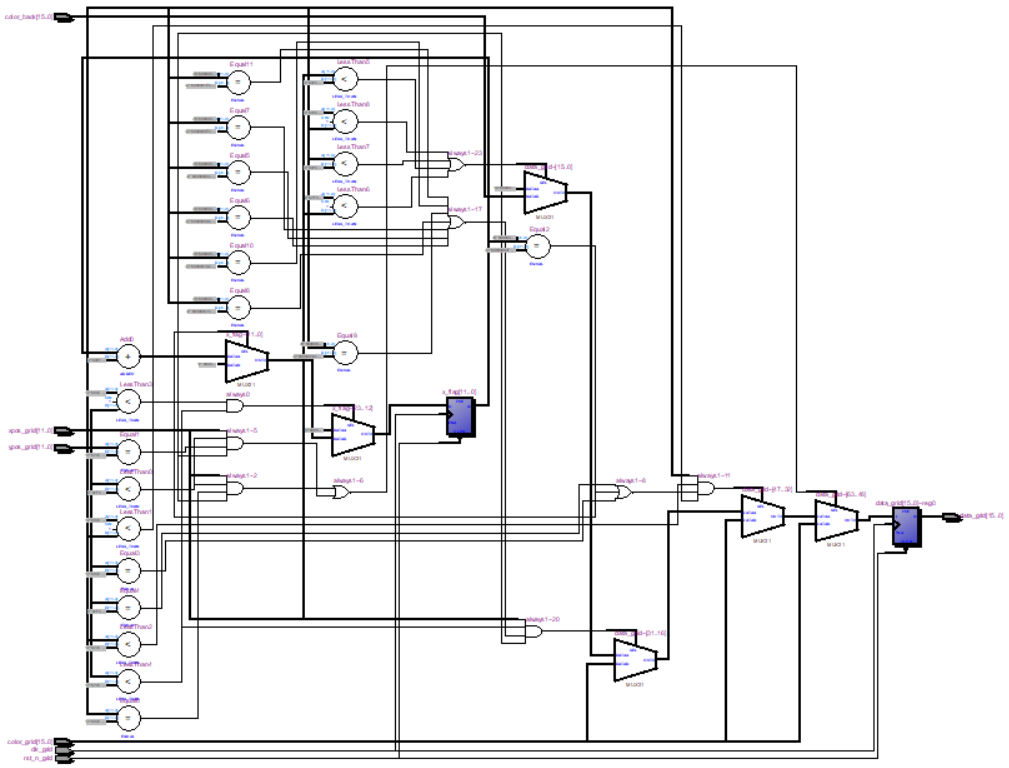
```
module grid_vga(
    input                clk_grid,
    input                rst_n_grid,
    input       [11:0]   xpos_grid,   //输入横坐标
    input       [11:0]   ypos_grid,   //输入纵坐标
    input       [15:0]   color_grid,
    input       [15:0]   color_back,

    output reg  [15:0]   data_grid     //输出产生的图像数据
);
parameter    H_DISP    = 12'd800 ,     //显示水平像素
             V_DISP    = 12'd600 ;     //显示垂直像素

localparam   BORDER_WIDTH    = 12'd44 ,     //显示边沿
             SCREEN_WIDTH    = 12'd512,     //波形显示水平宽度
             SCREEN_LENGTH = 12'd512,     //波形显示垂直宽度
```

```
                    WORD_AREA       = 12'd200,   //字符显示部分宽度
                    UNIT_WIDTH      = 12'd64 ;   //单元格边长

    reg [11:0] x_flag;

    always @(posedge clk_grid or negedge rst_n_grid) begin
     if (!rst_n_grid) begin
          x_flag <= 12'd0;
     end
     else if (xpos_grid >= BORDER_WIDTH && xpos_grid < BORDER_WIDTH +
SCREEN_LENGTH) begin
          x_flag <= (x_flag == UNIT_WIDTH - 1) ? 12'd0 : x_flag + 12'd1;   //确定出
现纵轴的 x 坐标位置
     end
     else begin
          x_flag <= 12'd0;
     end
    end


    always @(posedge clk_grid or negedge rst_n_grid)
    begin
     if (!rst_n_grid) begin
          data_grid <= 16'd0;
     end
     else if ((ypos_grid == BORDER_WIDTH && xpos_grid >= BORDER_WIDTH &&
xpos_grid < BORDER_WIDTH + SCREEN_LENGTH + WORD_AREA && xpos_grid[2]
== 1'b1) || (ypos_grid == BORDER_WIDTH + SCREEN_WIDTH && xpos_grid >=
BORDER_WIDTH && xpos_grid < BORDER_WIDTH + SCREEN_LENGTH +
WORD_AREA && xpos_grid[2] == 1'b1))    //上下两根线
      begin
          data_grid <= color_grid;
     end
     else if ((x_flag == UNIT_WIDTH - 1 || xpos_grid == BORDER_WIDTH || xpos_grid
== BORDER_WIDTH + SCREEN_LENGTH + WORD_AREA) && ypos_grid >=
BORDER_WIDTH && ypos_grid < BORDER_WIDTH + SCREEN_WIDTH &&
ypos_grid[2] == 1'b1)    //垂直网格线
    begin
          data_grid <= color_grid;
     end
     else if ((ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 1 ||
                    ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 2 ||
                    ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 3 ||
                    ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 4 ||
```

```
                ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 5 ||
                ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 6 ||
                ypos_grid == BORDER_WIDTH + UNIT_WIDTH * 7) && xpos_grid >= BORDER_WIDTH && xpos_grid < BORDER_WIDTH + SCREEN_LENGTH && xpos_grid[2] == 1'b1)    //水平网格线
        begin
            data_grid <= color_grid;
        end
        else if(xpos_grid < BORDER_WIDTH || xpos_grid >= BORDER_WIDTH + SCREEN_LENGTH + WORD_AREA || ypos_grid < BORDER_WIDTH || ypos_grid >= BORDER_WIDTH + SCREEN_WIDTH) begin
            data_grid <= color_back;
        end
        else begin
            data_grid <= 16'd0;
        end
        end

        endmodule
```

模块中 xpos_grid[2] == 1'b1 以及 ypos_grid[2] == 1'b1 的条件表示每隔 4 个像素点画一条 4 个像素点长的线。

RTL 视图如下：



4，字符显示模块 char_vga

字符显示模块用于显示测量的物理量与单位。对于 VGA 显示的字符显示，需要两种方式得到字符库 :一是利用字模软件 PCtoLCD2002 生成字模然后生成.mif 文件，之后利用 quartus 自带的 IP 核 rom 生成每个字符对应的 rom 核，但是.mif 文件的生成有些繁琐。二是利用 Verilog 直接编写对应的字模模块，代码的处理可能比较麻烦，但是调用比较方便，本设计采用第二种方式。

① 符号字模模块 char_set

本模块中仅设置了在 vga 显示中必要的字符。模块根据数据输入选择相应的字模数据输出。本模块在字符显示模块中例化。

模块代码如下：

```
module char_set(
 input clk,
 input rst_n,
 input [3:0] data,
 output reg [7:0] col0,
 output reg [7:0] col1,
 output reg [7:0] col2,
 output reg [7:0] col3,
 output reg [7:0] col4,
 output reg [7:0] col5,
 output reg [7:0] col6
 );

 always @(posedge clk or negedge rst_n)
      begin
           if (!rst_n)
                begin
                     col0 <= 8'b0000_0000;
                     col1 <= 8'b0000_0000;
                     col2 <= 8'b0000_0000;
                     col3 <= 8'b0000_0000;
                     col4 <= 8'b0000_0000;
                     col5 <= 8'b0000_0000;
                     col6 <= 8'b0000_0000;
                end
           else
                begin
                     case (data)
                          4'd1: // "F"
                               begin
                                    col0 <= 8'b0000_0000;
                                    col1 <= 8'b0111_1111;
                                    col2 <= 8'b0000_1001;
```

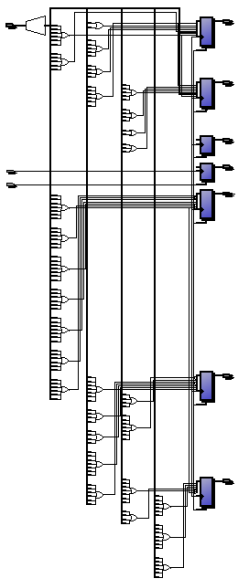
```
                col3 <= 8'b0000_1001;
                col4 <= 8'b0000_1001;
                col5 <= 8'b0000_0001;
                col6 <= 8'b0000_0000;
            end
        4'd2: // "H"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0111_1111;
                col2 <= 8'b0000_1000;
                col3 <= 8'b0000_1000;
                col4 <= 8'b0000_1000;
                col5 <= 8'b0111_1111;
                col6 <= 8'b0000_0000;
            end
        4'd3: // "I"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0000_0000;
                col2 <= 8'b0100_0001;
                col3 <= 8'b0111_1111;
                col4 <= 8'b0100_0001;
                col5 <= 8'b0000_0000;
                col6 <= 8'b0000_0000;
            end
        4'd4: // "K"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0111_1111;
                col2 <= 8'b0000_1000;
                col3 <= 8'b0001_0100;
                col4 <= 8'b0010_0010;
                col5 <= 8'b0100_0001;
                col6 <= 8'b0000_0000;
            end
        4'd5: // "U"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0011_1111;
                col2 <= 8'b0100_0000;
                col3 <= 8'b0100_0000;
                col4 <= 8'b0100_0000;
                col5 <= 8'b0011_1111;
                col6 <= 8'b0000_0000;
```

```
        end
    4'd6: // "Q"
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0011_1110;
            col2 <= 8'b0100_0001;
            col3 <= 8'b0101_0001;
            col4 <= 8'b0110_0001;
            col5 <= 8'b0111_1110;
            col6 <= 8'b0000_0000;
        end
    4'd7: // "△"
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0111_0000;
            col2 <= 8'b0100_1100;
            col3 <= 8'b0100_0011;
            col4 <= 8'b0100_1100;
            col5 <= 8'b0111_0000;
            col6 <= 8'b0000_0000;
        end
    4'd8: // "T"
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0000_0001;
            col2 <= 8'b0000_0001;
            col3 <= 8'b0111_1111;
            col4 <= 8'b0000_0001;
            col5 <= 8'b0000_0001;
            col6 <= 8'b0000_0000;
        end
    4'd9: // "V"
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0001_1111;
            col2 <= 8'b0010_0000;
            col3 <= 8'b0100_0000;
            col4 <= 8'b0010_0000;
            col5 <= 8'b0001_1111;
            col6 <= 8'b0000_0000;
        end
    4'd10: // "X"
        begin
            col0 <= 8'b0000_0000;
```

```
                col1 <= 8'b0110_0011;
                col2 <= 8'b0001_0100;
                col3 <= 8'b0000_1000;
                col4 <= 8'b0001_0100;
                col5 <= 8'b0110_0011;
                col6 <= 8'b0000_0000;
            end
        4'd11: // "Y"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0000_0011;
                col2 <= 8'b0000_0100;
                col3 <= 8'b0111_1000;
                col4 <= 8'b0000_0100;
                col5 <= 8'b0000_0011;
                col6 <= 8'b0000_0000;
            end
        4'd12: // "Z"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0110_0001;
                col2 <= 8'b0101_0001;
                col3 <= 8'b0100_1001;
                col4 <= 8'b0100_0101;
                col5 <= 8'b0100_0011;
                col6 <= 8'b0000_0000;
            end
        4'd13: // " "
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0000_0000;
                col2 <= 8'b0000_0000;
                col3 <= 8'b0000_0000;
                col4 <= 8'b0000_0000;
                col5 <= 8'b0000_0000;
                col6 <= 8'b0000_0000;
            end
        4'd14: // ":"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0000_0000;
                col2 <= 8'b0011_0110;
                col3 <= 8'b0011_0110;
                col4 <= 8'b0000_0000;
```

```
                        col5 <= 8'b0000_0000;
                        col6 <= 8'b0000_0000;
                    end
                4'd15: // "S"
                    begin
                        col0 <= 8'b0000_0000;
                        col1 <= 8'b0010_0110;
                        col2 <= 8'b0100_1001;
                        col3 <= 8'b0100_1001;
                        col4 <= 8'b0100_1001;
                        col5 <= 8'b0011_0010;
                        col6 <= 8'b0000_0000;
                    end
                default: // "*"
                    begin
                        col0 <= 8'b0000_0000;
                        col1 <= 8'b0010_0010;
                        col2 <= 8'b0001_0100;
                        col3 <= 8'b0000_1000;
                        col4 <= 8'b0001_0100;
                        col5 <= 8'b0010_0010;
                        col6 <= 8'b0000_0000;
                    end
            endcase
        end
    end

endmodule
```

RTL 视图如下：



② 字符显示 char_vga

字符显示模块对字符字模模块进行了例化，并在确定的位置显示相应的物理量与单位，输入端口包括时钟复位输入、xy 横纵坐标输入、字符颜色输入，输出为字符颜色数据。

模块代码如下：

```
module char_vga(
input                   clk_char,
 input                   rst_n_char,
 input          [11:0]   xpos_char,
 input          [11:0]   ypos_char,
input           [15:0]   color_char,

 output reg    [15:0]   data_char
);

localparam     BORDER_WIDTH    = 12'd44 ,
               SCREEN_WIDTH    = 12'd512,
               SCREEN_LENGTH = 12'd512,
               WORD_AREA        = 12'd200,
               UNIT_WIDTH       = 12'd64 ,

               WORD_WIDTH       = 12'd7    ,
               WORD_HIGH     = 12'd7    ,

               F = 4'd1, H = 4'd2, I = 4'd3,
               K = 4'd4, U = 4'd5, Q = 4'd6,
               TR= 4'd7, T = 4'd8, V = 4'd9,
               X = 4'd10,Y = 4'd11,Z = 4'd12,
               NO= 4'd13,qq= 4'd14,S = 4'd15;


//+++++++++++++++++++++++++++++++
//display settled
//+++++++++++++++++++++++++++++++

reg [15:0] word;
wire [7:0] p[27:0];

//  例化符号字模模块
char_set u1(
 .clk(clk_char),
 .rst_n(rst_n_char),
 .data(word[15:12]),
 .col0(p[0]),
```

```
 .col1(p[1]),
 .col2(p[2]),
 .col3(p[3]),
 .col4(p[4]),
 .col5(p[5]),
 .col6(p[6])
);
char_set u2(
 .clk(clk_char),
 .rst_n(rst_n_char),
 .data(word[11:8]),
 .col0(p[7]),
 .col1(p[8]),
 .col2(p[9]),
 .col3(p[10]),
 .col4(p[11]),
 .col5(p[12]),
 .col6(p[13])
);
char_set u3(
 .clk(clk_char),
 .rst_n(rst_n_char),
 .data(word[7:4]),
 .col0(p[14]),
 .col1(p[15]),
 .col2(p[16]),
 .col3(p[17]),
 .col4(p[18]),
 .col5(p[19]),
 .col6(p[20])
);
char_set u4(
 .clk(clk_char),
 .rst_n(rst_n_char),
 .data(word[3:0]),
 .col0(p[21]),
 .col1(p[22]),
 .col2(p[23]),
 .col3(p[24]),
 .col4(p[25]),
 .col5(p[26]),
 .col6(p[27])
);
```

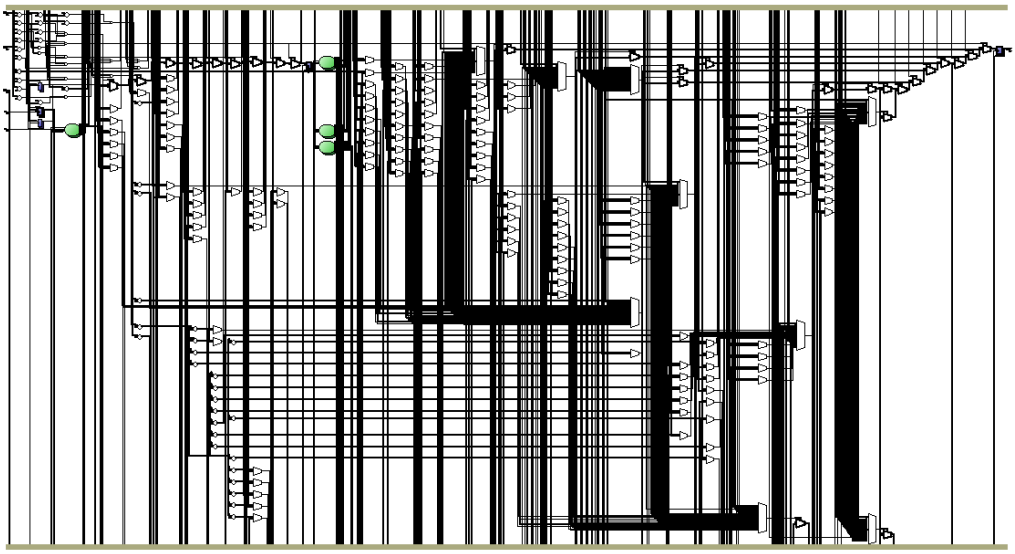
```
//+++++++++++++++++++++++++++++++
//VFF:
//+++++++++++++++++++++++++++++++
localparam    UP_vff    = BORDER_WIDTH + UNIT_WIDTH,
          DOWN_vff = UP_vff + WORD_HIGH,
          LEFT_vff = BORDER_WIDTH + SCREEN_LENGTH + 5 * WORD_WIDTH,

          RIGHT_vff= LEFT_vff + 4 * WORD_WIDTH,

          UNIT_vff_l = RIGHT_vff + 7 * WORD_WIDTH,
          UNIT_vff_r = UNIT_vff_l + 4 * WORD_WIDTH;


//+++++++++++++++++++++++++++++++
//FQ:
//+++++++++++++++++++++++++++++++
localparam    UP_fq    = BORDER_WIDTH + UNIT_WIDTH * 2,
          DOWN_fq = UP_fq + WORD_HIGH,
          LEFT_fq = BORDER_WIDTH + SCREEN_LENGTH + 5 * WORD_WIDTH,

          RIGHT_fq= LEFT_fq + 4 * WORD_WIDTH,

          UNIT_fq_l = RIGHT_fq + 7 * WORD_WIDTH,
          UNIT_fq_r = UNIT_fq_l + 4 * WORD_WIDTH;

//+++++++++++++++++++++++++++++++
//T:
//+++++++++++++++++++++++++++++++
localparam    UP_t    = BORDER_WIDTH + UNIT_WIDTH * 3,
          DOWN_t = UP_t + WORD_HIGH,
          LEFT_t = BORDER_WIDTH + SCREEN_LENGTH + 5 * WORD_WIDTH,

          RIGHT_t= LEFT_t + 4 * WORD_WIDTH,

          UNIT_t_l = RIGHT_t + 7 * WORD_WIDTH,
          UNIT_t_r = UNIT_t_l + 4 * WORD_WIDTH;

//+++++++++++++++++++++++++++++++
//V:
//+++++++++++++++++++++++++++++++
localparam    UP_v    = BORDER_WIDTH + UNIT_WIDTH * 4,
          DOWN_v = UP_v + WORD_HIGH,
          LEFT_v = BORDER_WIDTH + SCREEN_LENGTH + 5 * WORD_WIDTH,
```

```
                RIGHT_v= LEFT_v + 4 * WORD_WIDTH,

                UNIT_v_l = RIGHT_v + 7 * WORD_WIDTH,
                UNIT_v_r = UNIT_v_l + 4 * WORD_WIDTH;

    always @ (posedge clk_char or negedge rst_n_char)
     begin
        if (!rst_n_char) begin
            data_char <= 15'b0;
            word <= NO;
        end
        else if (ypos_char >= UP_vff && ypos_char <= DOWN_vff && xpos_char >=
LEFT_vff && xpos_char <= RIGHT_vff) begin
                word <= {NO, V, F, qq};
                if (p[xpos_char - LEFT_vff][ypos_char - UP_vff]) begin
                    data_char <= color_char;
                end
                else begin
                    data_char <= 15'b0;
                end
        end
        else if (ypos_char >= UP_fq && ypos_char <= DOWN_fq && xpos_char >=
LEFT_fq && xpos_char <= RIGHT_fq) begin
                word <= {NO, F, Q, qq};
                if (p[xpos_char - LEFT_fq][ypos_char - UP_fq]) begin
                    data_char <= color_char;
                end
                else begin
                    data_char <= 15'b0;
                end
        end
        else if (ypos_char >= UP_t && ypos_char <= DOWN_t && xpos_char >= LEFT_t
&& xpos_char <= RIGHT_t) begin
                word <= {NO, TR, T, qq};
                if (p[xpos_char - LEFT_t][ypos_char - UP_t]) begin
                    data_char <= color_char;
                end
                else begin
                    data_char <= 15'b0;
                end
        end
        else if (ypos_char >= UP_v && ypos_char <= DOWN_v && xpos_char >=
LEFT_v && xpos_char <= RIGHT_v) begin
                word <= {NO, TR, V, qq};
```

```
                    if (p[xpos_char - LEFT_v][ypos_char - UP_v]) begin
                        data_char <= color_char;
                    end
                    else begin
                        data_char <= 15'b0;
                    end
            end
            //UNIT Display
            else if (ypos_char >= UP_vff && ypos_char <= DOWN_vff && xpos_char >=
UNIT_vff_l && xpos_char <= UNIT_vff_r) begin
                    word <= {NO, V, NO, NO};
                    if (p[xpos_char - UNIT_vff_l][ypos_char - UP_vff]) begin
                        data_char <= color_char;
                    end
                    else begin
                        data_char <= 15'b0;
                    end
            end
            else if (ypos_char >= UP_fq && ypos_char <= DOWN_fq && xpos_char >=
UNIT_fq_l && xpos_char <= UNIT_fq_r) begin
                    word <= {NO, K, H, Z};
                    if (p[xpos_char - UNIT_fq_l][ypos_char - UP_fq]) begin
                        data_char <= color_char;
                    end
                    else begin
                        data_char <= 15'b0;
                    end
            end
            else if (ypos_char >= UP_t && ypos_char <= DOWN_t && xpos_char >=
UNIT_t_l && xpos_char <= UNIT_t_r) begin
                    word <= {NO, U, S, NO};
                    if (p[xpos_char - UNIT_t_l][ypos_char - UP_t]) begin
                        data_char <= color_char;
                    end
                    else begin
                        data_char <= 15'b0;
                    end
            end
            else if (ypos_char >= UP_v && ypos_char <= DOWN_v && xpos_char >=
UNIT_v_l && xpos_char <= UNIT_v_r) begin
                    word <= {NO, V, NO, NO};
                    if (p[xpos_char - UNIT_v_l][ypos_char - UP_v]) begin
                        data_char <= color_char;
                    end
```

```
                else begin
                    data_char <= 15'b0;
                end
        end
        else begin
            data_char <= 15'b0;
        end
    end

endmodule
```

RTL 视图如下：



5， 数据处理模块 data_process

数据处理模块可以说是整个示波器的核心模块，在此模块中利用双口 ram 存储输入数据并送至波形显示模块，并对输入的数据进行处理得到峰峰值与频率值，送至结果显示模块显示。数据处理模块的子模块包括 bin2BCD， rom_256to10，data_ram。

① 数据缓冲模块 data_ram

由于 AD 转换数据输入时钟为 32MHz，而 VGA 显示所用像素时钟为 40MHz，故需要一个双口 ram 进行缓冲。

模块例化如下：

```
data_ram    data_ram_u0(
 .data(ad_data),      //输入 AD 数据
 .rdaddress(xpos_dp[8:0] - BORDER_WIDTH[8:0]),    //读取地址
 .rdclock(clk_dp),    //输入读取时钟（像素时钟）
 .rden(data_en),      //读取使能

 .wraddress(ad_vga_addr),    //写入地址
```

```
    .wrclock(clk_ad),          //写入时钟（AD 转换时钟）
    .wren(wr_en),              //写入使能

    .q(ad_vga_data)            //数据读取输出
   );
```

②数据转换模块 rom_256to10

为了节约资源，避免使用除法器，尝试利用 rom 来实现峰峰值的数据转换，构建一个从 0~10 的 256 位的等差数列，并生成其.mif 文件，在数据处理模块中例化如下：

```
rom_256to10 u1(
    .address(addr),
    .clock(clk_ad),
    .q(data_rom)
);
```

③二级制与 BCD 码转换模块 bin2BCD

由于从 ram 与 rom 中读取到的数据均为二进制数据，而在显示时因为要按位显示对应的十进制值，所以需要 bin2BCD 模块将相应数据转换为 BCD 码再输出。本代码利用了“加三移位法”。

模块代码如下：

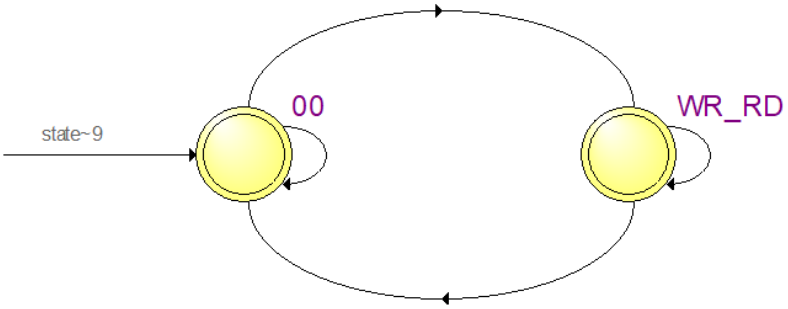
```
module bin2BCD (
    output reg [3:0]   ten_thou,
    output reg [3:0]   thou,
    output reg [3:0]   hun ,
    output reg [3:0]   ten ,
    output reg [3:0]   unit,

    input   [19:0] in_bin
);

integer i;
always @(in_bin) begin
    ten_thou = 4'd0;
    thou= 4'd0;
    hun = 4'd0;
    ten = 4'd0;
    unit= 4'd0;

    for(i=19; i>=0; i=i-1) begin
        if(ten_thou>=4'd5)          //若此位大于 5，则给此位加 3
            ten_thou   = ten_thou   + 4'd3;
        if(thou>=4'd5)
            thou   = thou   + 4'd3;
```

```
            if(hun>=4'd5)
                hun    = hun    + 4'd3;
            if(ten>=4'd5)
                ten    = ten    + 4'd3;
            if(unit>=4'd5)
                unit = unit + 4'd3;

            ten_thou = ten_thou << 1;
            ten_thou[0] = thou[3];
            thou = thou << 1;
            thou[0] = hun[3];
            hun = hun << 1;
            hun[0] = ten[3];
            ten = ten << 1;
            ten[0] = unit[3];
            unit = unit << 1;
            unit[0] = in_bin[i];        //从最高位不断左移

        end
    end

    endmodule
```

④ 数据处理模块 data_process

模块中比较重要的部分为数据的读取以及峰峰值、频率值的获得，将在下文中一一介绍。

数据读取部分采用了 2 个状态的状态机设计，状态转换图如下：



初始状态为 IDLE，若 ad 输入数据达到触发条件（触发电平 TRIGGER 与上升沿）则转入读写状态，并将读取使能信号置为 1；读写状态中再次检验触发条件，若检验到则记录此触发条件的位置，若读取地址宽度达到满屏（512 个像素位）则转入 IDLE 状态，并将读取使能信号置为 0，进入新一轮的循环。从而实现数据的不断读取。

相关代码如下：

```
//++++++++++++++++++++++++++++++++
//Verse_state
//++++++++++++++++++++++++++++++++
reg [1:0] state;
wire posedge_ad_data;

reg [7:0] ad_data_r;
always @(posedge clk_ad or negedge rst_dp_n) begin
    if (!rst_dp_n) begin
        ad_data_r <= 0;
    end
    else begin
        ad_data_r <= ad_data;
    end
end

assign posedge_ad_data = (ad_data_r < ad_data) ? 1'b1 : 1'b0;

always @(posedge clk_ad or negedge rst_dp_n) begin
    if (!rst_dp_n) begin
        state <= 0;
        data_en <= 1'b0;
    end
    else begin
        case(state)
            IDLE: if (ad_data == TRIGGER && posedge_ad_data) begin
                state <= WR_RD;
                data_en <= 1'b1;
            end
            WR_RD:  if (ad_vga_addr == DATA_WIDTH - 1) begin
                state <= IDLE;
                data_en <= 1'b0;
            end
        endcase
    end
end
```

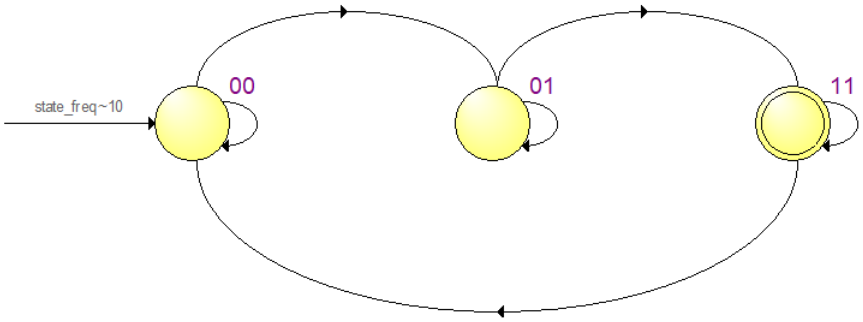
峰峰值通过记录最大与最小值然后求差值获得，具体方法为在一个显示周期内不断比较最大值寄存器 max_data_r，最小值寄存器 min_data_r 与输入 ad 数据的大小，不断更新最大值与最小值寄存器，在一个显示周期结束后（IDLE 状态）将值赋给寄存器 max_data 与最小值 min_data，根据二者的差输入 rom_256to10 或直接利用比例除法运算来得到二进制实际电压值，并通过 bin2BCD 转化后输出。

相关代码如下：

```
//++++++++++++++++++++++++++++++
//Vff
//++++++++++++++++++++++++++++++
wire     [7:0]   addr;
reg      [7:0]           max_data;
reg      [7:0]           max_data_r;
reg      [7:0]           min_data;
reg      [7:0]         min_data_r;
wire    [15:0]   data_rom;

always @(posedge clk_ad or negedge rst_dp_n) begin
      if (!rst_dp_n) begin
            max_data_r <= 8'b0;
            max_data <= 8'b0;
      end
      else if (data_en == 1'b0 && max_data_r > 8'b0) begin
            max_data <= max_data_r;
            max_data_r <= 8'b0;
      end
      else if (data_en == 1'b1 && ad_data > max_data_r) begin
            max_data_r <= ad_data;
            max_data <= max_data;
      end
      else begin
            max_data_r <= max_data_r;
            max_data <= max_data;
      end
end

always @(posedge clk_ad or negedge rst_dp_n) begin
      if (!rst_dp_n) begin
            min_data_r <= 8'b0;
            min_data <= 8'b0;
      end
      else if (data_en == 1'b0 && min_data_r < 8'd255) begin
            min_data <= min_data_r;
            min_data_r <= 8'd255;
      end
      else if (data_en == 1'b1 && ad_data < min_data_r) begin
            min_data_r <= ad_data;
            min_data <= min_data;
      end
      else begin
            min_data_r <= min_data_r;
```

```
                min_data <= min_data;
            end
        end

        assign addr = max_data - min_data;

        assign data_rom = addr * 13 / 2;
        /*
        rom_256to10 u1(
            .address(addr),
            .clock(clk_ad),
            .q(data_rom)
        );
        */

        bin2BCD bB_u2(
            .in_bin({8'd0,data_rom[15:4]}),

            .ten_thou(),
            .thou(),
            .hun (data_vff[11:8]),
            .ten (data_vff[7:4]) ,
            .unit(data_vff[3:0])
        );
```

频率值的获得是根据在状态机中得到的触发位置的地址值与 AD 芯片采样频率（32MHz）的比值来得到，利用 32000 除以地址值即可得到频率值，单位 kHz。需要注意的是在触发条件检测时发现相应的触发电平值并不稳定，故利用状态机设计，状态转换图如下：



在初始状态，满足触发条件即跳转到下一个状态，此部分跳转与数据读取模块跳转一致，在第二个状态下，若检测到波形下降即进入第三个检测状态，在第三个状态若检测到上升沿并满足电平大于触发电平，即记录此位置 pos，然后转入初始状态，这样设计就避免了触发电平不稳定的问题。最后将得到的频率值通过 bin2BCD 转化后输出。

相关部分代码如下：

```
    //+++++++++++++++++++++++++++++++
    //freq
    //+++++++++++++++++++++++++++++++
    reg [8:0] pos   ;
    reg [1:0] state_freq;

    always @(posedge clk_ad or negedge rst_dp_n) begin
        if (!rst_dp_n) begin
            pos     <= 9'd0;
            state_freq <= 2'b00;
        end
        else begin
            case(state_freq)
                2'b00:    if (ad_data == TRIGGER && posedge_ad_data && state
== IDLE) begin
                    pos <= pos;
                    state_freq <= 2'b01;
                end
                2'b01:    if (!posedge_ad_data) begin
                    pos <= pos;
                    state_freq <= 2'b11;
                end
                2'b11:    if (ad_data >= TRIGGER && posedge_ad_data) begin
                    pos <= ad_vga_addr;
                    state_freq <= 2'b00;
                end
                default: state_freq <= 2'b00;
            endcase
        end
    end

    wire [19:0] freq_result;
    //assign freq_result = 32000 * cnt / pos;
    assign freq_result = ad_data ? 32000 / pos : 20'd0;

    bin2BCD bB_u1(
    //    .in_bin(data_div[23:4]) ,
        .in_bin(freq_result),

        .ten_thou(data_freq[19:16]),
        .thou(data_freq[15:12]),
        .hun (data_freq[11:8]),
        .ten (data_freq[7:4]),
        .unit(data_freq[3:0])
```

```
);
```

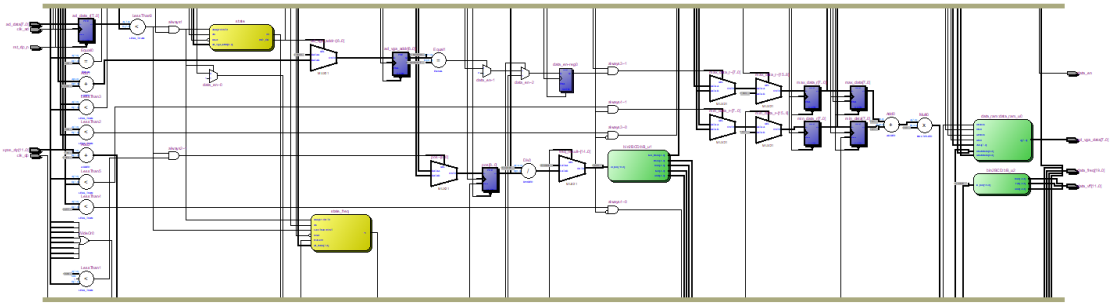
数据处理模块输入端口包括像素时钟、AD 时钟、复位信号、ad 数据输入以及 x 位置输入（用于读取地址），输出端口包括 ram 数据输出，数据使能输出，信号峰峰值与频率值。

模块剩余代码如下：

```
module data_process(
    input                   clk_dp,
    input                   clk_ad,
    input                   rst_dp_n,
    input           [7:0]   ad_data,
    input           [11:0] xpos_dp,

    output          [7:0]   ad_vga_data, //max:256
    output  reg             data_en,
    output          [11:0] data_vff,
    output     [19:0] data_freq
);

parameter IDLE   = 2'b00,
          WR_RD = 2'b10,
//          TEST   = 2'b11,

          TRIGGER      = 8'd128,   //trigger 0V
          DATA_WIDTH = 10'd512,

          BORDER_WIDTH   = 12'd44 ,
          SCREEN_WIDTH   = 12'd512,
          SCREEN_LENGTH = 12'd640,
          WORD_AREA       = 12'd72 ,
          UNIT_WIDTH      = 12'd64 ,

          H_DISP   = 12'd800 ,
          V_DISP   = 12'd600 ;

//+++++++++++++++++++++++++++++++
//Verse_state
//+++++++++++++++++++++++++++++++
...
//+++++++++++++++++++++++++++++++
//Vff
//+++++++++++++++++++++++++++++++
...
```

```
//+++++++++++++++++++++++++++++++
//freq
//+++++++++++++++++++++++++++++++
...
//+++++++++++++++++++
//ad_vag_addr
//+++++++++++++++++++
wire wr_en;

reg [8:0] ad_vga_addr;
reg [8:0] ad_vga_addr_n;

always @(posedge clk_ad or negedge rst_dp_n)
begin
    if(!rst_dp_n) begin
        ad_vga_addr <= 9'd0;
    end
    else if(state == WR_RD && ad_vga_addr < DATA_WIDTH) begin
        ad_vga_addr <= ad_vga_addr + 1'b1;
    end
    else begin
//      ad_vga_addr <= ad_vga_addr;
        ad_vga_addr <= 9'd0;
    end
end

assign wr_en    = (state == WR_RD) ? 1'b1 : 1'b0;

//512*8
data_ram    data_ram_u0(
    .data(ad_data),
    .rdaddress(xpos_dp[8:0] - BORDER_WIDTH[8:0]),
    .rdclock(clk_dp),
    .rden(data_en),

    .wraddress(ad_vga_addr),
    .wrclock(clk_ad),
    .wren(wr_en),

    .q(ad_vga_data)
);

endmodule
```

RTL 视图如下：

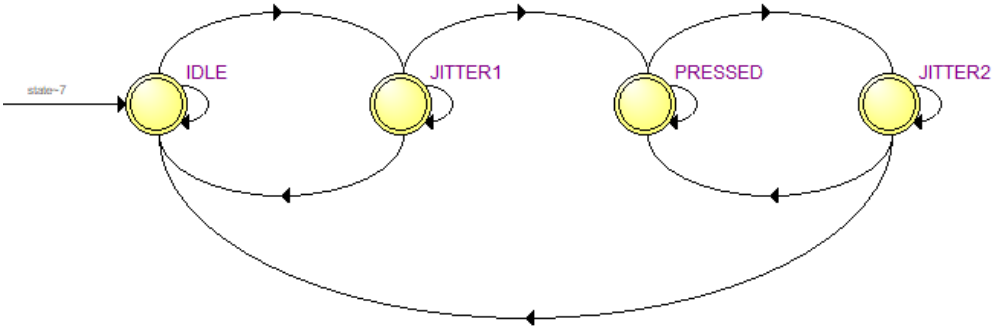


6， 波形显示模块 wave_vga

波形显示模块也是一个较为重要的模块，本模块除了将从数据处理模块输入的波形数据输出至 VGA 驱动模块之外，还实现了按键去抖动，波形的水平、垂直放缩与水平与垂直的光标调节，将在后文一一介绍。

① 按键去抖动

利用状态机设计实现按键去抖动，状态转换图如下所示：



波形显示模块有四个按键输入，作用分别为水平、垂直波形放缩，光标选择与光标移动，状态机分为初始状态 IDLE，抖动状态 JITTER1，按下状态 PRESSED，以及抖动状态 JITTER2；启动时为 IDLE 状态，当四个按键中有一个按下时，状态机启动，进入 JITTER 状态，计数器开始计数，计数 20ms 后若按键仍然为低电平（按下为低），则将相应按键值置 1，并进入 PRESSED 状态；若按键为高电平说明时抖动，则回到 IDLE 状态；PRESSED 状态若检测到所有按键均未被按压，则进入 JITTER2，同理，当 JITTER2 计数到 20ms，检测按键若仍均为高电平，则回到 IDLE 状态，否则返回 PRESSED 状态。

20ms 计数采用 PLL 将像素频率分至 10kHz。

此部分相关代码如下：

```
wire clk_10k;

pll_nsk pn_u1(
    .inclk0(clk_wave), //40M
    .c0(clk_10k)
);
```

```
//++++++++++++++++++++++++++++++++++
//去抖动
//++++++++++++++++++++++++++++++++++
reg [1:0] state;
reg [7:0] cnt;              //200 * 0.0001s = 20ms
reg [1:0] size_cg_r;
reg cs_r;
reg move_r;

always @(posedge clk_10k or negedge rst_n_wave) begin
    if (!rst_n_wave) begin
        state <= IDLE;
        cnt <= 8'd0;
        size_cg_r <= 2'b11;
        cs_r <= 1'd1;
        move_r <= 1'd1;
    end
    else begin
        case(state)
            IDLE:
                if(size_cg != 2'b11 || cs_cursor == 1'b0 || move_cursor == 1'b0)
begin
                    state <= JITTER1;
                end
                else begin
                    state <= IDLE;
                end
            JITTER1:
                if(cnt < 8'd200) begin
                    cnt <= cnt + 8'd1;
                end
                else if(size_cg != 2'b11 || cs_cursor == 1'b0 || move_cursor ==
1'b0) begin
                    cnt <= 8'd0;
                    state <= PRESSED;
                    size_cg_r <= size_cg;
                    cs_r <= cs_cursor;
                    move_r <= move_cursor;
                end
                else begin
                    state <= IDLE;
                end
            PRESSED:
                if(size_cg == 2'b11 && cs_cursor == 1'b1 && move_cursor ==
```

```
1'b1) begin
                        state <= JITTER2;
                    end
                    else begin
                        state <= PRESSED;
                    end
                JITTER2:
                    if(cnt < 8'd200) begin
                        cnt <= cnt + 8'd1;
                    end
                    else if(size_cg == 2'b11 && cs_cursor == 1'b1 && move_cursor
== 1'b1) begin
                        cnt <= 8'd0;
                        state <= IDLE;
                        size_cg_r <= size_cg;
                        cs_r <= cs_cursor;
                        move_r <= move_cursor;
                    end
                    else begin
                        state <= PRESSED;
                    end
            endcase
        end
end
```

② 波形水平垂直放缩

模块中采用移位的方式放缩波形，横轴与纵轴均有 1，2，4 三档，当输入 size_c 信号完成去抖动后，使相关横纵放缩计数器在 1~3 之间变化，并将此计数值用于横纵的移位。

对于纵轴电压显示来说，直接对输入的 ram 数据进行放缩即可；但是对于横轴时间来说，放缩代表着需要改变读取地址，从每个地址均读到隔 2 个地址读一个数，所以在此模块中也需要一个 ram，当然，可以也可以把放缩功能写入数据处理模块，这样可以节省一个 ram。最初设计的时候考虑的时利用一个 1024 位的 ram 记录两个显示周期的值，在一个周期中显示，则可实现当前坐标值的缩小，但是由于触发条件的存在，在一个显示周期内显示两个周期的值会使得在交界的地方波形不连续，故无法实现，但在此模块中采用的仍时 1024 位的 ram，所以有生成读取地址的部分。

此部分相关代码如下：

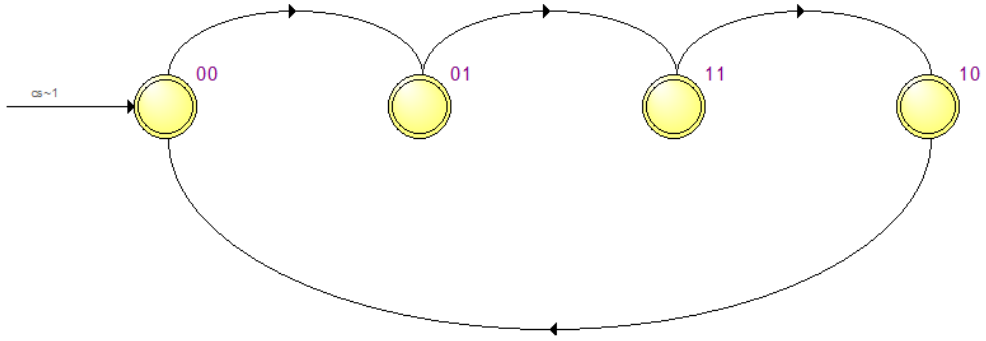
```
//++++++++++++++++++++++++++++++++
//波形放缩
//++++++++++++++++++++++++++++++++
reg [1:0] move_h;
always @(negedge size_cg_r[1] or negedge rst_n_wave) begin
  if(!rst_n_wave) begin
```

```
            move_h <= 2'b00;
        end
        else if(move_h < 2'b10) begin
            move_h <= move_h + 2'b01;
        end
        else begin
            move_h <= 2'b00;
        end
    end

    reg [1:0] move_v;
    always @(negedge size_cg_r[0] or negedge rst_n_wave) begin
        if(!rst_n_wave) begin
            move_v <= 2'b00;
        end
        else if(move_v < 2'b10) begin
            move_v <= move_v + 2'b01;
        end
        else begin
            move_v <= 2'b00;
        end
    end

    //+++++++++++++++++++++++++++++++++
    //ram_r
    //+++++++++++++++++++++++++++++++++
    wire [7:0] data_ram;
    reg [9:0] wr_addr;
    reg db;

    always @(posedge clk_wave or negedge rst_n_wave) begin
        if (!rst_n_wave) begin
            db <= 1'd0;
        end
        else if(xpos_wave == BORDER_WIDTH + SCREEN_LENGTH) begin
            db <= ~db;
        end
        else begin
            db <= db;
        end
     end

    always @(posedge clk_wave or negedge rst_n_wave) begin
        if (!rst_n_wave) begin
```

```
            wr_addr <= 10'd0;
        end
        else if (~db && xpos_wave >= BORDER_WIDTH && xpos_wave <
    BORDER_WIDTH + SCREEN_LENGTH) begin
            wr_addr <= xpos_wave[9:0] - BORDER_WIDTH[9:0];
        end
        else if (db && xpos_wave >= BORDER_WIDTH && xpos_wave <
    BORDER_WIDTH + SCREEN_LENGTH) begin
            wr_addr <= xpos_wave[9:0] - BORDER_WIDTH[9:0] + 10'd512;
        end
        else begin
            wr_addr <= wr_addr;
        end
      end

    ram_r rr_u1(
        .clock(clk_wave),
        .data(data),
        .rdaddress((xpos_wave[9:0] - BORDER_WIDTH[9:0]) >> move_h),
        .rden(1'b1),
        .wraddress(wr_addr),
        .wren(data_en),
        .q(data_ram)
    );
```

③ 光标测量

波形显示模块中设置了上下左右四个光标以实现 V 与 T 的手动测量，结合真实的示波器可知，光标需要实现连续移动，需要在 VGA 显示上表现连续移动需要利用 wire 变量，但同时需要 reg 变量记录切换状态时各个光标的位置，故也采用状态机设计来切换选择光标，状态转换图如下：



四个状态下，分别控制不同的光标移动。为了肉眼能够看出光标的移动，控制时钟为 10kHz 信号，计数 500，即 1 秒移动 20 个像素位置。

最终将上下位置的寄存器作差，经过数据转换，得到ΔV 值，而将左右的寄存器作差，经过数据转换，得到ΔT 值，再利用 bin2BCD 转换后输出到数据显示模块。

此部分相关代码如下：

```
//++++++++++++++++++++++++++++++++
//cursor
//++++++++++++++++++++++++++++++++
wire [8:0] cursor_u;
wire [8:0] cursor_d;
wire [8:0] cursor_l;
wire [8:0] cursor_r;

reg [8:0] cursor_u_r;
reg [8:0] cursor_d_r;
reg [8:0] cursor_l_r;
reg [8:0] cursor_r_r;

reg [1:0] cs;
reg [8:0] move;

//move cursor
reg [8:0] cnt_move; //0.0001 * 500 = 0.05
always @(posedge clk_10k or negedge rst_n_wave or negedge cs_r) begin
    if (!rst_n_wave || !cs_r) begin
        move <= 9'b0;
        cnt_move <= 9'b0;
    end
    else if (!move_r) begin
        if (cnt_move < 9'd500) begin
            cnt_move <= cnt_move + 1'b1;
        end
        else begin
            cnt_move <= 9'd0;
        end
        //位置移动
        if(cnt_move == 9'd500) begin
            move <= move + 1'b1;
        end
        else begin
            move <= move;
        end
    end
end

//choose cursor
always @(negedge cs_r or negedge rst_n_wave) begin
```

```
        if (!rst_n_wave) begin
            cs <= 2'b00;
            cursor_u_r = V_CS_U;
            cursor_d_r = V_CS_D;
            cursor_l_r = T_CS_R;
            cursor_r_r = T_CS_L;
        end
        else begin
            case(cs)
                2'b00:    if (!cs_r) begin
                    cs <= 2'b01;
                    cursor_u_r = cursor_u;
                    cursor_d_r = cursor_d;
                    cursor_l_r = cursor_l;
                    cursor_r_r = cursor_r;
                end
                2'b01:    if (!cs_r) begin
                    cs <= 2'b11;
                    cursor_u_r = cursor_u;
                    cursor_d_r = cursor_d;
                    cursor_l_r = cursor_l;
                    cursor_r_r = cursor_r;
                end
                2'b11:    if (!cs_r) begin
                    cs <= 2'b10;
                    cursor_u_r = cursor_u;
                    cursor_d_r = cursor_d;
                    cursor_l_r = cursor_l;
                    cursor_r_r = cursor_r;
                end
                2'b10:    if (!cs_r) begin
                    cs <= 2'b00;
                    cursor_u_r = cursor_u;
                    cursor_d_r = cursor_d;
                    cursor_l_r = cursor_l;
                    cursor_r_r = cursor_r;
                end
            endcase
        end
    end

    assign cursor_u = (cs == 2'b00) ? (move + cursor_u_r) : cursor_u_r;
    assign cursor_d = (cs == 2'b01) ? (move + cursor_d_r) : cursor_d_r;
    assign cursor_l = (cs == 2'b11) ? (move + cursor_l_r) : cursor_l_r;
```

```
assign cursor_r = (cs == 2'b10) ? (move + cursor_r_r) : cursor_r_r;

//T
wire [31:0] vira_T;
wire [10:0] vira_T_r;

assign vira_T = 3125 * ((cursor_u > cursor_d) ? (cursor_u - cursor_d) : (cursor_d - cursor_u));
assign vira_T_r = (vira_T / 1000) >> move_h;

bin2BCD bB_u1(
    .in_bin({9'd0,vira_T_r}),

    .ten_thou(),
    .thou(data_T[15:12]),
    .hun (data_T[11:8]),
    .ten (data_T[7:4]) ,
    .unit(data_T[3:0])
);
//V
wire [8:0] vira_V;
wire [15:0] vira_V_r;

assign vira_V = ((cursor_r > cursor_l) ? (cursor_r - cursor_l) : (cursor_l - cursor_r));
assign vira_V_r = (vira_V * 39 / 5) >> move_v;

bin2BCD bB_u2(
    .in_bin({9'd0,vira_V_r}),

    .ten_thou(),
    .thou(data_V[15:12]),
    .hun (data_V[11:8]),
    .ten (data_V[7:4]) ,
    .unit(data_V[3:0])
);
```

④ 其他

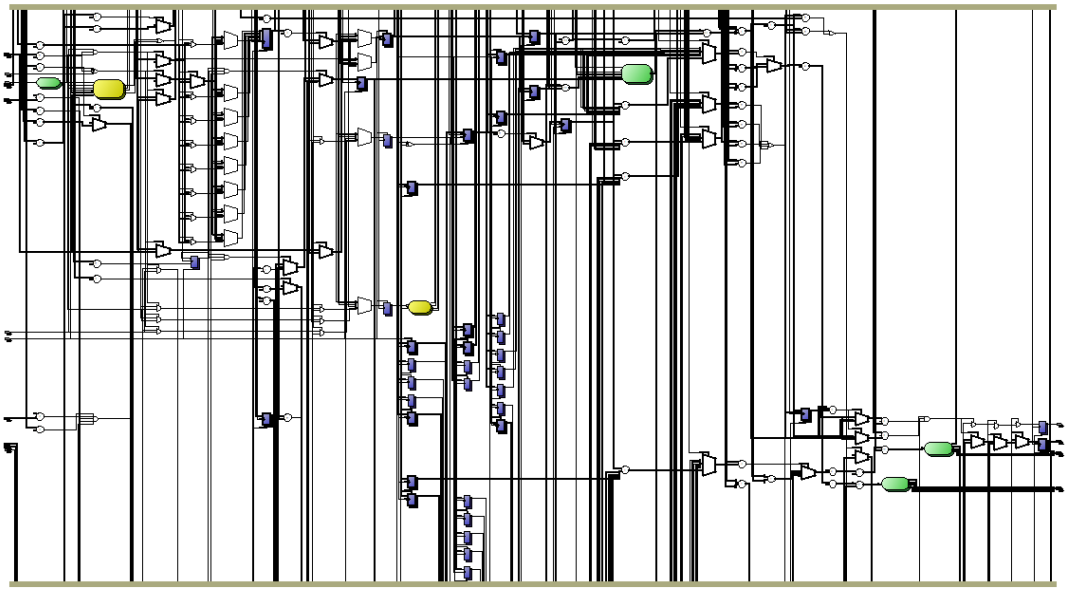
剩余部分代码包括端口声明、常数声明与数据显示 ;输入端口包括时钟端、复位端、xy 横纵坐标、波形颜色、光标颜色、数据及其使能输入、横纵波形缩放、光标选择与光标移动，输出包括ΔT、ΔV 数据输出，波形有效输出以及波形 RGB 信号输出；显示部分与网格、字符显示类似，不再赘述，代码如下：

```
module wave_vga(
    input                   clk_wave,
```

```
    input                   rst_n_wave,
    input       [11:0]      xpos_wave,
    input       [11:0]      ypos_wave,
    input       [15:0]    color_wave,
    input       [15:0]    color_cursor,
    input       [7:0]       data,
    input                   data_en,
    input       [1:0]       size_cg,   //1:h 0:v
    input                   cs_cursor,
    input                   move_cursor,

    output reg  [15:0]      data_wave,
    output      [15:0]      data_V,
    output      [15:0]      data_T,
    output reg              wave_valid
);
parameter   H_DISP    = 12'd800 ,
            V_DISP    = 12'd600 ;

localparam  BORDER_WIDTH    = 12'd44 ,
            SCREEN_WIDTH    = 12'd512,
            SCREEN_LENGTH = 12'd512,
            WORD_AREA         = 12'd200,
            UNIT_WIDTH        = 12'd64 ,

            //去抖动各状态
            IDLE       = 2'b00,
            JITTER1 = 2'b01,
            PRESSED = 2'b11,
            JITTER2 = 2'b10,

            //光标初始位置
            T_CS_L = BORDER_WIDTH + 2 * UNIT_WIDTH,
            T_CS_R = BORDER_WIDTH + 6 * UNIT_WIDTH,
            V_CS_U = BORDER_WIDTH + 2 * UNIT_WIDTH,
            V_CS_D = BORDER_WIDTH + 6 * UNIT_WIDTH;

//++++++++++++++++++++++++++++++++++
//去抖动
//++++++++++++++++++++++++++++++++++
...
//++++++++++++++++++++++++++++++++++
//cursor
//++++++++++++++++++++++++++++++++++
```

```
    ...
    //++++++++++++++++++++++++++++++++++
    //波形放缩
    //++++++++++++++++++++++++++++++++++
    ...
    //++++++++++++++++++++++++++++++++++
    //display
    //++++++++++++++++++++++++++++++++++

    always @(posedge clk_wave or negedge rst_n_wave)
    begin
        if (!rst_n_wave) begin
            data_wave   <= 16'd0;
            wave_valid <= 1'b0;
        end
        else if (data_en && xpos_wave >= BORDER_WIDTH && xpos_wave < BORDER_WIDTH + SCREEN_LENGTH && ypos_wave >= BORDER_WIDTH && ypos_wave < BORDER_WIDTH + SCREEN_WIDTH) begin
            if (xpos_wave == cursor_u || xpos_wave == cursor_d || ypos_wave == cursor_l || ypos_wave == cursor_r)begin
                data_wave <= color_cursor;
                wave_valid <= 1'b1;
            end
            else if(ypos_wave - BORDER_WIDTH == 12'd256 + (12'd64 << move_v) - ((data_ram >> 1'b1) << move_v) || ypos_wave - BORDER_WIDTH + 1'd1 == 12'd256 + (12'd64 << move_v) - ((data_ram >> 1'b1) << move_v)) begin
                data_wave   <= color_wave;
                wave_valid <= 1'b1;
            end
            else begin
                data_wave   <= 16'b0;
                wave_valid <= 1'b0;
            end
        end
        else begin
            data_wave   <= 16'd0;
            wave_valid <= 1'b0;
        end
    end

    endmodule
```

RTL 视图如下：



7，结果显示模块 result_display

结果显示模块架构与字符显示模块类似，根据不同的位置参数在 VGA 显示的不同位置输出数据。

① 数字字模模块

与字符显示模块不同的是其中应用的字模模块存储的是数字与小数点。

字模储存模块代码如下：

```
module digit_set(
    input clk,
    input rst_n,
    input [3:0] data,
    output reg [7:0] col0,
    output reg [7:0] col1,
    output reg [7:0] col2,
    output reg [7:0] col3,
    output reg [7:0] col4,
    output reg [7:0] col5,
    output reg [7:0] col6
);

always @(posedge clk or negedge rst_n)
    begin
        if (!rst_n)
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0000_0000;
                col2 <= 8'b0000_0000;
                col3 <= 8'b0000_0000;
```

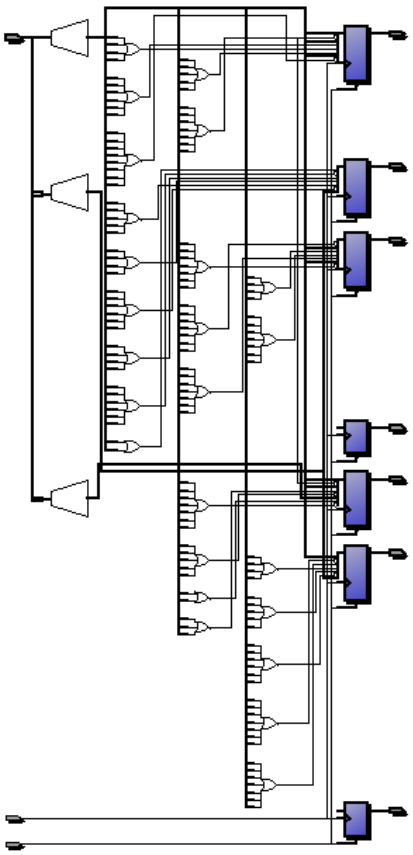
```
                col4 <= 8'b0000_0000;
                col5 <= 8'b0000_0000;
                col6 <= 8'b0000_0000;
            end
        else
            begin
                case (data)
                    4'b0000: // "0"
                        begin
                            col0 <= 8'b0000_0000;
                            col1 <= 8'b0011_1110;
                            col2 <= 8'b0101_0001;
                            col3 <= 8'b0100_1001;
                            col4 <= 8'b0100_0101;
                            col5 <= 8'b0011_1110;
                            col6 <= 8'b0000_0000;
                        end
                    4'b0001: // "1"
                        begin
                            col0 <= 8'b0000_0000;
                            col1 <= 8'b0000_0000;;
                            col2 <= 8'b0100_0010;
                            col3 <= 8'b0111_1111;
                            col4 <= 8'b0100_0000;
                            col5 <= 8'b0000_0000;;
                            col6 <= 8'b0000_0000;
                        end
                    4'b0010: // "2"
                        begin
                            col0 <= 8'b0000_0000;
                            col1 <= 8'b0100_0010;
                            col2 <= 8'b0110_0001;
                            col3 <= 8'b0101_0001;
                            col4 <= 8'b0100_1001;
                            col5 <= 8'b0100_0110;
                            col6 <= 8'b0000_0000;
                        end
                    4'b0011: // "3"
                        begin
                            col0 <= 8'b0000_0000;
                            col1 <= 8'b0010_0010;
                            col2 <= 8'b0100_0001;
                            col3 <= 8'b0100_1001;
                            col4 <= 8'b0100_1001;
```

```
                col5 <= 8'b0011_0110;
                col6 <= 8'b0000_0000;
            end
        4'b0100: // "4"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0001_1000;
                col2 <= 8'b0001_0100;
                col3 <= 8'b0001_0010;
                col4 <= 8'b0111_1111;
                col5 <= 8'b0001_0000;
                col6 <= 8'b0000_0000;
            end
        4'b0101: // "5"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0010_0111;
                col2 <= 8'b0100_0101;
                col3 <= 8'b0100_0101;
                col4 <= 8'b0100_0101;
                col5 <= 8'b0011_1001;
                col6 <= 8'b0000_0000;
            end
        4'b0110: // "6"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0011_1110;
                col2 <= 8'b0100_1001;
                col3 <= 8'b0100_1001;
                col4 <= 8'b0100_1001;
                col5 <= 8'b0011_0010;
                col6 <= 8'b0000_0000;
            end
        4'b0111: // "7"
            begin
                col0 <= 8'b0000_0000;
                col1 <= 8'b0110_0001;
                col2 <= 8'b0001_0001;
                col3 <= 8'b0000_1001;
                col4 <= 8'b0000_0101;
                col5 <= 8'b0000_0011;
                col6 <= 8'b0000_0000;
            end
        4'b1000: // "8"
```

```
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0011_0110;
            col2 <= 8'b0100_1001;
            col3 <= 8'b0100_1001;
            col4 <= 8'b0100_1001;
            col5 <= 8'b0011_0110;
            col6 <= 8'b0000_0000;
        end
    4'b1001: // "9"
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0010_0110;
            col2 <= 8'b0100_1001;
            col3 <= 8'b0100_1001;
            col4 <= 8'b0100_1001;
            col5 <= 8'b0011_1110;
            col6 <= 8'b0000_0000;
        end
    4'b1110: // " "
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0000_0000;
            col2 <= 8'b0000_0000;
            col3 <= 8'b0000_0000;
            col4 <= 8'b0000_0000;
            col5 <= 8'b0000_0000;
            col6 <= 8'b0000_0000;
        end
    4'b1100: // "."
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0000_0000;
            col2 <= 8'b1100_0000;
            col3 <= 8'b1100_0000;
            col4 <= 8'b0000_0000;
            col5 <= 8'b0000_0000;
            col6 <= 8'b0000_0000;
        end
    4'b1111: // "-"
        begin
            col0 <= 8'b0000_0000;
            col1 <= 8'b0000_1000;
            col2 <= 8'b0000_1000;
```

```
                        col3 <= 8'b0000_1000;
                        col4 <= 8'b0000_1000;
                        col5 <= 8'b0000_1000;
                        col6 <= 8'b0000_0000;
                    end
                default: // "*"
                    begin
                        col0 <= 8'b0000_0000;
                        col1 <= 8'b0010_0010;
                        col2 <= 8'b0001_0100;
                        col3 <= 8'b0000_1000;
                        col4 <= 8'b0001_0100;
                        col5 <= 8'b0010_0010;
                        col6 <= 8'b0000_0000;
                    end
            endcase
        end
    end

endmodule
```

RTL 视图如下：



② 结果显示模块

结果显示模块对数字字模模块进行了例化，并在确定的位置显示相应的数值，输入端口包括时钟复位输入、xy 横纵坐标输入、数字颜色输入，峰峰值、频率、ΔT、ΔV 输入，输出为字符颜色数据。

模块代码如下：

```
module result_display(
    input           clk_rd,
    input           rst_n_rd,
    input [11:0] data_vff,
    input [19:0] data_freq,
    input [11:0] xpos_rd,
    input [11:0] ypos_rd,
    input [15:0] color_rd,
    input [15:0]   data_V,
    input [15:0]   data_T,

    output reg [15:0] data_rd
);

localparam   BORDER_WIDTH   = 12'd44 ,
             SCREEN_WIDTH   = 12'd512,
             SCREEN_LENGTH = 12'd512,
             WORD_AREA       = 12'd200,
             UNIT_WIDTH      = 12'd64 ,

             WORD_WIDTH      = 12'd7   ,
             WORD_HIGH    = 12'd7   ,

             MINUS = 4'b1111,
             NO      = 4'b1110,
             DOT     = 4'b1100;

reg [19:0] word;
wire [7:0] p[34:0];

digit_set u1(
    .clk(clk_rd),
    .rst_n(rst_n_rd),
    .data(word[19:16]),
    .col0(p[0]),
    .col1(p[1]),
    .col2(p[2]),
    .col3(p[3]),
    .col4(p[4]),
```

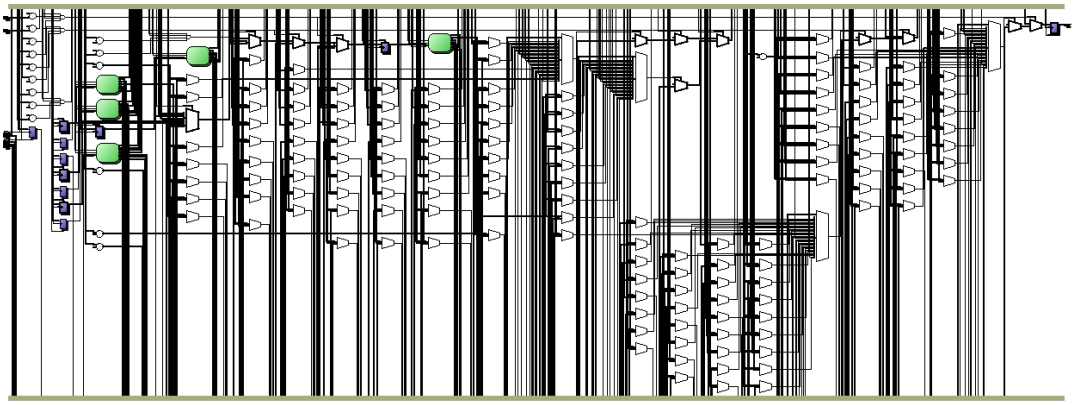
```
        .col5(p[5]),
        .col6(p[6])
    );
    digit_set u2(
        .clk(clk_rd),
        .rst_n(rst_n_rd),
        .data(word[15:12]),
        .col0(p[7]),
        .col1(p[8]),
        .col2(p[9]),
        .col3(p[10]),
        .col4(p[11]),
        .col5(p[12]),
        .col6(p[13])
    );
    digit_set u3(
        .clk(clk_rd),
        .rst_n(rst_n_rd),
        .data(word[11:8]),
        .col0(p[14]),
        .col1(p[15]),
        .col2(p[16]),
        .col3(p[17]),
        .col4(p[18]),
        .col5(p[19]),
        .col6(p[20])
    );
    digit_set u4(
        .clk(clk_rd),
        .rst_n(rst_n_rd),
        .data(word[7:4]),
        .col0(p[21]),
        .col1(p[22]),
        .col2(p[23]),
        .col3(p[24]),
        .col4(p[25]),
        .col5(p[26]),
        .col6(p[27])
    );
    digit_set u5(
        .clk(clk_rd),
        .rst_n(rst_n_rd),
        .data(word[3:0]),
        .col0(p[28]),
```

```
        .col1(p[29]),
        .col2(p[30]),
        .col3(p[31]),
        .col4(p[32]),
        .col5(p[33]),
        .col6(p[34])
    );

    //+++++++++++++++++++++++++++++++
    //VFF:
    //+++++++++++++++++++++++++++++++
    localparam    UP_vff     = BORDER_WIDTH + UNIT_WIDTH,
                  DOWN_vff = UP_vff + WORD_HIGH,
                 LEFT_vff = BORDER_WIDTH + SCREEN_LENGTH + 11 * WORD_WIDTH,

                  RIGHT_vff= LEFT_vff + 4 * WORD_WIDTH;

    //+++++++++++++++++++++++++++++++
    //FQ:
    //+++++++++++++++++++++++++++++++
    localparam    UP_fq     = BORDER_WIDTH + UNIT_WIDTH * 2,
                  DOWN_fq = UP_fq + WORD_HIGH,
                 LEFT_fq = BORDER_WIDTH + SCREEN_LENGTH + 11 * WORD_WIDTH,

                  RIGHT_fq= LEFT_fq + 5 * WORD_WIDTH;

    //+++++++++++++++++++++++++++++++
    //T:
    //+++++++++++++++++++++++++++++++
    localparam    UP_t     = BORDER_WIDTH + UNIT_WIDTH * 3,
                  DOWN_t = UP_t + WORD_HIGH,
                  LEFT_t = BORDER_WIDTH + SCREEN_LENGTH + 11 * WORD_WIDTH,

                  RIGHT_t= LEFT_t + 4 * WORD_WIDTH;

    //+++++++++++++++++++++++++++++++
    //V:
    //+++++++++++++++++++++++++++++++
    localparam    UP_v     = BORDER_WIDTH + UNIT_WIDTH * 4,
                  DOWN_v = UP_v + WORD_HIGH,
                  LEFT_v = BORDER_WIDTH + SCREEN_LENGTH + 11 * WORD_WIDTH,

                  RIGHT_v= LEFT_v + 4 * WORD_WIDTH;
```

```
always @ (posedge clk_rd or negedge rst_n_rd)
    begin
        if (!rst_n_rd) begin
            data_rd <= 15'b0;
            word <= {NO, NO, NO, NO, NO};
        end
        else if (ypos_rd >= UP_vff && ypos_rd <= DOWN_vff && xpos_rd >= LEFT_vff && xpos_rd <= RIGHT_vff) begin
                word <= {data_vff[11:4], DOT, data_vff[3:0], NO};
                if (p[xpos_rd - LEFT_vff][ypos_rd - UP_vff]) begin
                    data_rd <= color_rd;
                end
                else begin
                    data_rd <= 15'b0;
                end
        end
        else if (ypos_rd >= UP_fq && ypos_rd <= DOWN_fq && xpos_rd >= LEFT_fq && xpos_rd <= RIGHT_fq) begin
                word <= {data_freq};
                if (p[xpos_rd - LEFT_fq][ypos_rd - UP_fq]) begin
                    data_rd <= color_rd;
                end
                else begin
                    data_rd <= 15'b0;
                end
        end
        else if (ypos_rd >= UP_t && ypos_rd <= DOWN_t && xpos_rd >= LEFT_t && xpos_rd <= RIGHT_t) begin
                word <= {data_T[15:8], DOT, data_T[7:0]};
                if (p[xpos_rd - LEFT_t][ypos_rd - UP_t]) begin
                    data_rd <= color_rd;
                end
                else begin
                    data_rd <= 15'b0;
                end
        end
        else if (ypos_rd >= UP_v && ypos_rd <= DOWN_v && xpos_rd >= LEFT_v && xpos_rd <= RIGHT_v) begin
                word <= {data_V[15:8], DOT, data_V[7:0]};
                if (p[xpos_rd - LEFT_v][ypos_rd - UP_v]) begin
                    data_rd <= color_rd;
                end
                else begin
```

```
                    data_rd <= 15'b0;
                end
            end
            else begin
                data_rd <= 15'b0;
            end
        end

    endmodule
```

RTL 视图如下：



8， DA 转换

由于调试的时候不是常有信号源，所以采用 AD9708 芯片来产生波形输入，在 rom 中保存长度 512 位正弦波数据以 125MHz 输出。

此部分内容代码如下：

```
//++++++++++++++++++++++++++
//D2A
//++++++++++++++++++++++++++
reg [8:0] rom_da_addr;
always @(posedge clk_125M)
begin
 rom_da_addr <= rom_da_addr + 1'b1;
end

rom_sin rs_u1(
 .address(rom_da_addr),
 .clock(clk_125M),
 .q(da_data_o)
);
```

9， 顶层模块 Oscill_top

根据结构框图将以上模块在顶层模块中例化并连接，值得一提的是顶层模块中保存了 8 中颜色的 RGB 数据，可随时在顶层更改显示颜色；顶层部分还有一个选择器，作用是在波形输出有效的时候覆盖掉网格输出。

模块输入包括 50MHz 晶振输入、复位输入、四个按键输入、ad 数据输入，输出包括行、列同步信号输出、颜色 rgb 输出，输出至数模转换芯片的 125MHz 时钟输出、8 位数字波形信号输出，以及输出至模数转换芯片的 32MHz 时钟信号输出。

顶层模块代码如下：

```
module Oscill_top(
 input clk,
 input rst_n,
 input [7:0] ad_data_in,
 input [1:0] size_cg,
 input       cs_cursor,
 input       move_cursor,

 output hs_o,
 output vs_o,
 output [15:0] rgb_o,
 output clk_125M,
 output clk_32M,
 output [7:0] da_data_o,

 output flag
);

parameter   H_DISP  = 12'd800 ,
            H_FRONT = 12'd40  ,
            H_SYNC  = 12'd128 ,
            H_BACK  = 12'd88  ,
            H_TOTAL = 12'd1056,

            V_DISP  = 12'd600,
            V_FRONT = 12'd1   ,
            V_SYNC  = 12'd4   ,
            V_BACK  = 12'd23 ,
            V_TOTAL = 12'd628;

parameter    RED     = 16'hF800,    //11111_000000_00000
          GREEN   = 16'h07E0,    //00000_111111_00000
          BLUE    = 16'h001F,    //00000_000000_11111
          WHITE   = 16'hFFFF,    //11111_111111_11111
          BLACK   = 16'h0000,    //00000_000000_00000
```

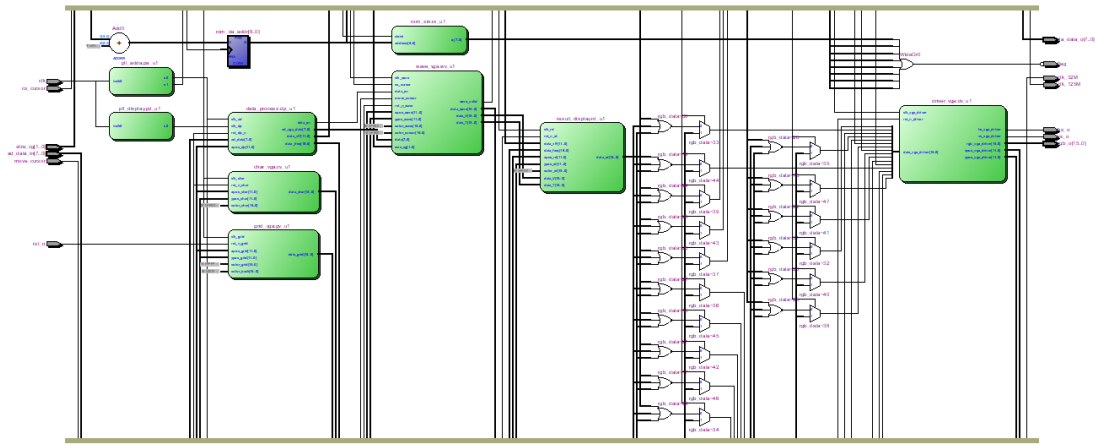
```
        YELLOW = 16'hFFE0,     //11111_111111_00000
        MAGENTA= 16'hF81F,     //11111_000000_11111
        CYAN     = 16'h07FF,     //00000_111111_11111

        DATA_WIDTH = 10'd512;

wire [11:0] xpos;
wire [11:0] ypos;
wire [15:0] rgb_data;
wire clk_40M;

wire [15:0] data_wave;
wire [15:0] data_grid;
wire [15:0] data_char;
wire [15:0] data_rd;
wire wave_valid;

assign flag = da_data_o ? 1'b0 : 1'b1 ;

assign rgb_data = wave_valid ? data_wave : (data_grid | data_char | data_rd);

pll_display pd_u1(
 .inclk0(clk),
 .c0(clk_40M)
);

pll_adda pa_u1(
 .inclk0(clk),
 .c0(clk_32M),
 .c1(clk_125M)
);

grid_vga gv_u1(
 .clk_grid(clk_40M),
 .rst_n_grid(rst_n),
 .xpos_grid(xpos),
 .ypos_grid(ypos),
 .color_grid(WHITE),
 .color_back(BLUE),

 .data_grid(data_grid)
);

driver_vga dv_u1(
```

```
 .clk_vga_driver(clk_40M),
 .rst_n_driver(rst_n),
 .data_vga_driver(rgb_data),

 .rgb_vga_driver(rgb_o),
 .hs_vga_driver(hs_o),
 .vs_vga_driver(vs_o),
 .xpos_vga_driver(xpos),
 .ypos_vga_driver(ypos)
);

wire [7:0] ad_vga_data;
wire       data_en;

data_process dp_u1(
 .clk_dp(clk_40M),
 .clk_ad(clk_32M),
 .rst_dp_n(rst_n),
 .ad_data(ad_data_in),
 .xpos_dp(xpos),

 .ad_vga_data(ad_vga_data), //max:256
 .data_en(data_en),
 .data_vff(data_vff),
 .data_freq(data_freq)
);

wave_vga wv_u1(
 .clk_wave(clk_40M),
 .rst_n_wave(rst_n),
 .xpos_wave(xpos),
 .ypos_wave(ypos),
 .color_wave(YELLOW),
 .color_cursor(RED),
 .data(ad_vga_data),
 .data_en(data_en),
 .size_cg(size_cg),
 .cs_cursor(cs_cursor),
 .move_cursor(move_cursor),

 .data_wave(data_wave),
 .data_T(data_T),
 .data_V(data_V),
 .wave_valid(wave_valid)
```

```
);

//++++++++++++++++++++++++++
//D2A
//++++++++++++++++++++++++++
reg [8:0] rom_da_addr;
always @(posedge clk_125M)
begin
 rom_da_addr <= rom_da_addr + 1'b1;
end

rom_sin rs_u1(
 .address(rom_da_addr),
 .clock(clk_125M),
 .q(da_data_o)
);

//++++++++++++++++++++++++++++
//char
//++++++++++++++++++++++++++++

char_vga cv_u1(
 .clk_char(clk_40M),
      .rst_n_char(rst_n),
      .xpos_char(xpos),
      .ypos_char(ypos),
 .color_char(YELLOW),
      .data_char(data_char)
);

wire [11:0] data_vff;
wire [19:0] data_freq;
wire [15:0] data_T;
wire [15:0] data_V;

result_display rd_u1(
 .clk_rd(clk_40M),
 .rst_n_rd(rst_n),

 .data_vff(data_vff),
 .data_freq(data_freq),
 .data_T(data_T),
 .data_V(data_V),
```

```
.xpos_rd(xpos),
      .ypos_rd(ypos),
.color_rd(YELLOW),

.data_rd(data_rd)
);

Endmodule
```

RTL 视图如下：

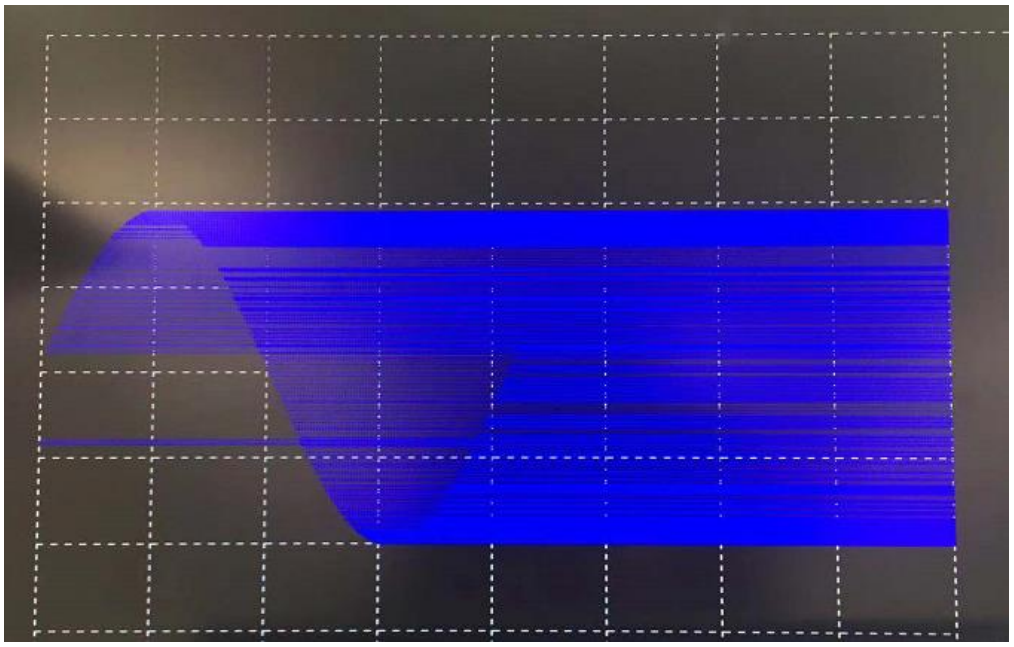


四， 制作调试

本实验设计的顺序是先完成显示，再附加相应的功能与结果输出，在设计过程中遇到了一些问题与困难，在此记录：

① 波形显示

在第一次写波形显示模块时出现了如下图所示的问题：



首先想到的是触发时机不对，检查代码发现触发只设定了电平而未设定上升沿或下降沿，增加上升沿触发条件，发现波形仍然不稳定，于是设置代码只显示一组波形数据，结果仍如上图所示，后来发现原因是在相关像素点将颜色输出赋值为 BLUE 后没有返回 BLACK，所以导致前面部分只要有一个点为 BLUE，后面一行点均为 BLUE，后通过增加逻辑结构解决了这个问题。

② ram 读写地址冲突

由于 ad 转换时钟与像素时钟频率不同，当读写信号地址相同时数据的值时不定的，有效的解决方法是读写分不同状态进行，单考虑到只有一个信号位的值是不定的，对于 VGA 显示来说影响不大，故忽略这个问题。

③ 除法运算

考虑到直接用除法占用资源较多，所以自己写了一个除法器，代码如下：

```
module div(
    input[31:0] a,
    input[31:0] b,     //若 b==0，则 quo=0, rem=a;

    output reg [31:0] quo,
    output reg [31:0] rem
);

reg[31:0] tempa;
reg[31:0] tempb;
reg[63:0] temp_a;
reg[63:0] temp_b;

integer i;
always @(a or b)
begin
    tempa <= a;
    tempb <= b;
end

always @(tempa or tempb)
begin
    temp_a = {32'h00000000,tempa};
    temp_b = {tempb,32'h00000000};
    for(i = 0;i < 32;i = i + 1)
        begin
            temp_a = {temp_a[62:0],1'b0};
            if(temp_a[63:32] >= tempb)
                temp_a = temp_a - temp_b + 1'b1;
            else
                temp_a = temp_a;
```

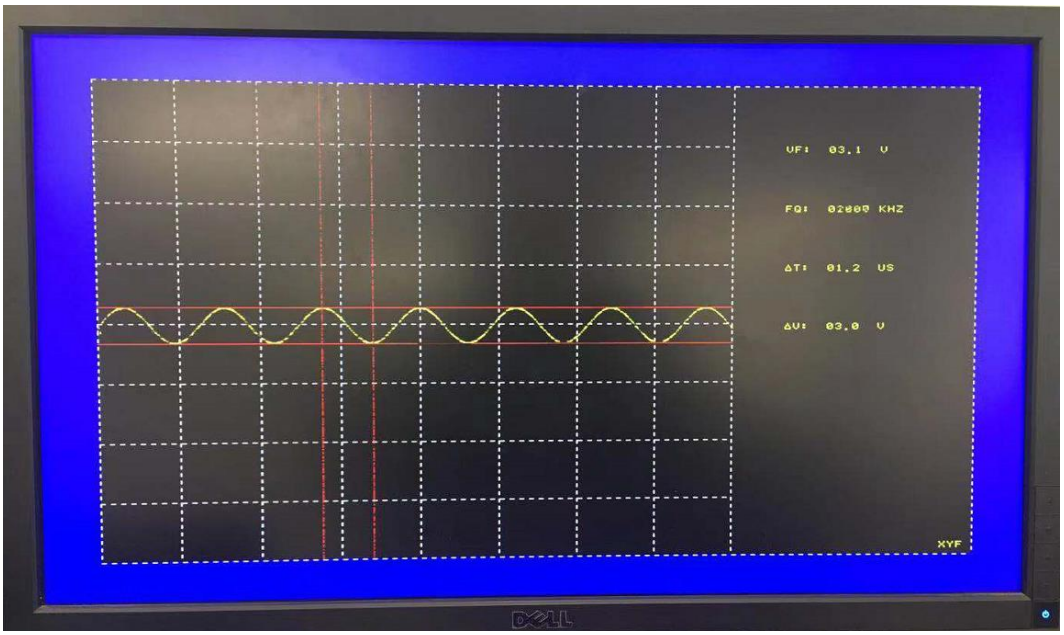
```
            end
        quo = temp_a[31:0];
        rem = temp_a[63:32];
    end

    endmodule
```

后来从资源占用的数据来看其实资源是足够的，所以在设计中直接使用除法来保证代码的间接。

④频率测量

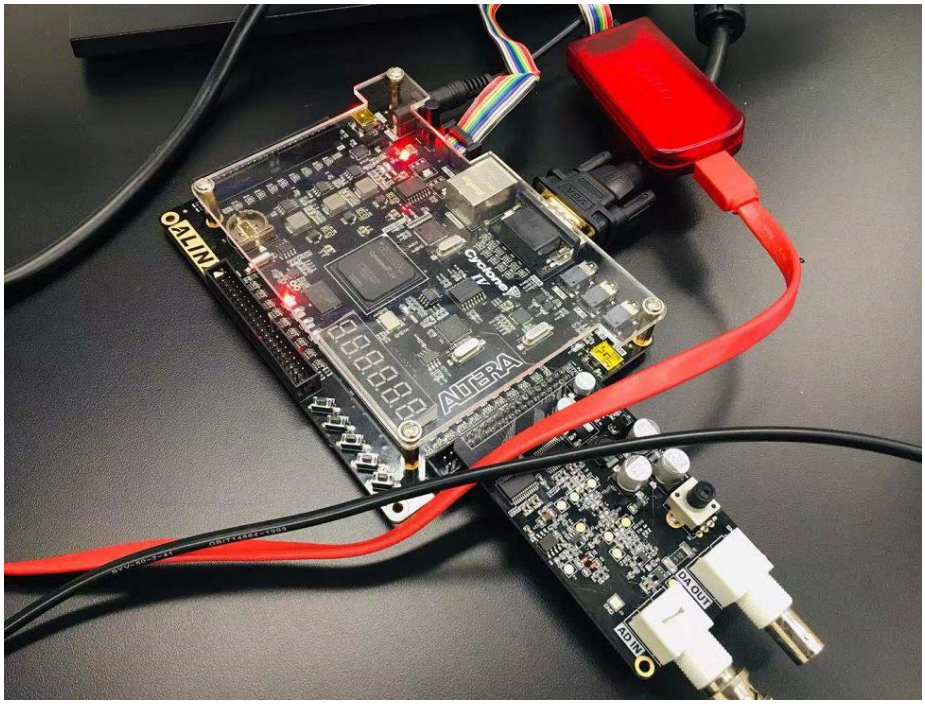
测量频率时首先的想法是根据一个显示周期内最大值的个数来确定频率，但是发现计算频率无法稳定，检查后发现原因是最大值是不定的，所以每个显示周期内记录的个数也是不定的；后考虑计数触发条件，但由于触发电平是不定的，所以也无法得到稳定的频率值，之后才考虑用状态机设计来计数，只要大于触发电平即记录并返回，这样所得到的频率基本是稳定的。

最终通过 VGA 显示结果如下：



可通过按键来实现波形的放缩与手动测量，同时也可以在代码中直接改变各部分颜色与触发电平。

开发板连接：



五， 资源消耗

Flow Summary	
Flow Status	Successful - Fri Dec 14 09:03:17 2018
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	Oscil
Top-level Entity Name	Oscill_top
Family	Cyclone IV E
Device	EP4CE15F23C8
Timing Models	Final
Total logic elements	2,667 / 15,408 (17 %)
Total combinational functions	2,654 / 15,408 (17 %)
Dedicated logic registers	475 / 15,408 (3 %)
Total registers	475
Total pins	43 / 344 (13 %)
Total virtual pins	0
Total memory bits	14,848 / 516,096 (3 %)
Embedded Multiplier 9-bit elements	4 / 112 (4 %)
Total PLLs	3 / 4 (75 %)

六， 参考资料

[1] VGA 显示：https://www.cnblogs.com/chensimin1990/p/5783156.html
[2] 加 3 移位法：https://blog.csdn.net/tokeyman/article/details/50326127
[3] rom/ram 核的使用：http://www.fpga.gs/h-nr.html?m31pageno=1#fai_31_top
[4] .mif 文件的生成：https://blog.csdn.net/wzshtuhao/article/details/79420725
[5] 结构设计：http://tech.hqew.com/fangan_753511