南京理工大学
NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

数字系统设计实验报告

学　　院：电子工程与光电技术学院

姓　　名：徐靖秋

学　　号：922104780732

班　　级：9221042402

指导老师：谭雪琴

2024 年 11 月

摘要

本设计使用 Xilinx 公司的 Vivado 软件、EGO1 Aritix-7 实验板卡硬件平台（xc7a35tcsg324-1）和 Verilog HDL 语言完成了一系列任务：

1. 实现四种波形 DDS 设计，AM 调制与 FM 调制；

2. 用示波器观察实验板上 8bit dac 转换器输出波形；改变频率控制字，观察波形的频率变化；

3. 设计测频电路，将理论频率显示在实验板卡上的左面 4 位数码管，测量的波形频率值显示在实验板卡上的右面 4 位数码管；

4 所有功能合为一体通过 5 个按键控制，并将每个模式的 4 个功能用呼吸流水灯对应、实现波形选择、数据串口输入、可调音量的音乐播放、VGA 显示等任务。

关键词：FPGA Verilog dds vga pwm

Abstract

This design uses Xilinx's Vivado software, the EGO1 Aritix-7 experimental board hardware platform (xc7a35tcsg324-1) and the Verilog HDL language to accomplish a number of tasks:

1. implement four waveform DDS designs, AM modulation and FM modulation;

2. use an oscilloscope to observe the output waveform of the 8bit dac converter on the experimental board; change the frequency control word to observe the frequency change of the waveform. 3. design a frequency measurement circuit to measure the frequency of the waveform;

3. design the frequency measurement circuit, the theoretical frequency will be displayed on the left side of the experimental board card 4-bit digital tube, the measured waveform frequency value is displayed on the right side of the experimental board card 4-bit digital tube;

4. all functions into one through the 5 keys control, and each mode of the 4 functions with breathing running lights corresponding to the realisation of the waveform selection, data serial port input, adjustable volume of music playback, VGA display and other tasks.

Keywords: fpga, verilog, dds, vga, pwm.

目录

一．设计要求说明

1. 基本要求：
   1） 使用 VIVADO 与 EGO1 实现 DDS
   2） DDS 频率控制字可变
   3） 设计测频电路，测量频率显示在右 4 位数码管
   4） 计算波形频率的理论值，将理论值显示在左 4 位数码管
2. 拓展要求：
   1) 输出三角波、方波、锯齿波等多种波形
   2) 班级学号显示与 DDS 电路二合一
   3) 分量程显示频率
   4) ASK、PSK、FSK 调制
   5) AM、FM 调制
   6) 自主发挥添加其他功能

二．方案论证

1. 时钟方案

图 1 时钟规划

数字系统时钟规划如上图所示，板卡上 100M 时钟通过分频模块降频为合适的时钟信号，作为部分外设的输入时钟。

EGO1 板卡 VGA 实测支持 640*480@60Hz，该分辨率下时钟要求为 25MHz；DDS 部分原为 10k 时钟，但为了提高较高控制字下 DDS 分辨率，改进为快时钟配合内部分频的方案；本系统提高 5 个按键控制不同功能的切换，模式选择部分采用 250KHz 时钟维护；而按键由于其机械结构特性，需要消除抖动，通过 1KHz 扫描减少误触发情况；PWM 用于板卡 LED 呼吸效果以及不同流水灯显示，250KHz 时钟提高内置 100 分频（分配器用于改变占空比），故实际输出 PWM 频率为 2.5KHz ；数码管采用动态扫描方式实现显示，扫描时钟为 10KHz；测频使用闸门法，只在时钟高电平期间（持续 1s）计数，低电平状态将计数值输出。

2. 数码管方案

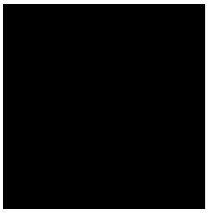


图 2 数码管示意图

板卡 8 个数码管分为左 4 位和右 4 位，每个 4 位数码管数据输入是独立的，组成它们的四个数码管则共用数据输入信号。每一个数码管都带有一个片选 CS 信号，高电平有效。在显示时，需要使用扫描信号逐个使能对应数码管，让它们按照一定次序亮起熄灭。利用人眼视觉残留效应，最终呈现出来效果为所有数码管都亮起。

3. 按键消抖方案

按键按下输出不是理想的，由于按键本身弹簧与触点的机械结构性质，按下按键后并不是理想的方波，而会在短时间内产生间断性的抖动，如下图所示

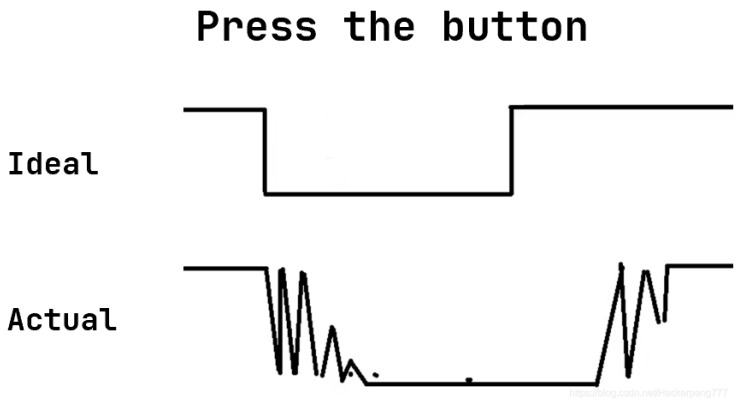


图 3 低电平触发按键按下波形示意图

本数字系统通过延时方式，对抖动进行消除，以尽可能减少对实际的影响。

4. 模式选择方案

本数字系统将学号显示、DDS 输出、FM、AM 作为不同功能集成到一个整体中。不同于普遍修改拨码开关实现模式选择，本人采用 5 个按键来选择模式控制，使用左右按键切换功能子种类、上下按键切换具体功能，极大的加快模式选择与切换速度。

5. 串口方案

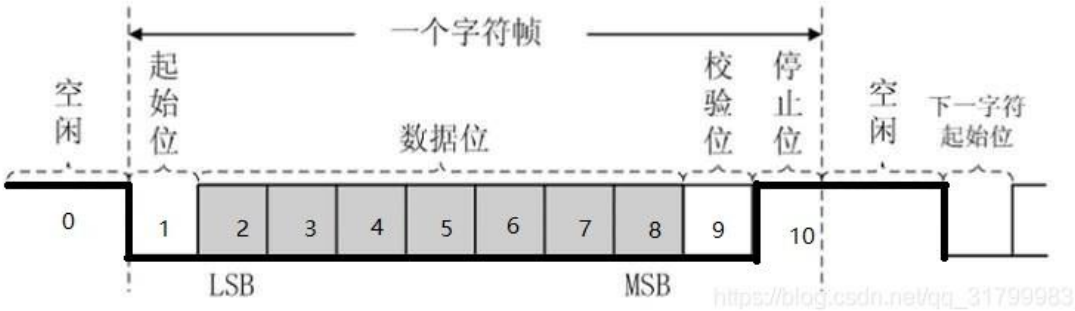


图 4 串口时序图

本方案串口波特率选择 9600bps，8bit 数据位，1bit 停止位，无校验位。

6. DDS 方案

DDS 输出频率可以通过公式计算得到：

$$f_{out} = \frac{f_{clk}\Delta\theta}{2^{B_{\theta(n)}}}$$

DDS 输出由内部地址控制，地址会随时钟上升沿周期性增加控制字个数。如果地址位宽为 8，那么当控制字为 128~255 时，输出波形实际有效位只有 1 位，最终呈现输出了也只可能是方波，为了提高输出精度，本方案采用 1MHz 高速时

钟配合地址截位，以实现在较高频率输出不失真。

DDS 波形使用 MATLAB 生成的 256 点 COE 文件存储，$f_{clk}=\frac{1MHz}{2^7}$，$2^{B_{\theta(n)}}=256$，计算可得通过拨码开关选择频率控制字符，能够完成从 0Hz 开始以 30.5Hz 为步进，极限频率 7.782KHz 的任意波形输出。

AM 调制全程为振幅调制，其公式可以表述为：

$$s(t)=U_c(1+m_\alpha\, cos\, 2\omega_m t)\, cos\, 2\omega_c t$$

式中$s(t)$为调制后信号，m_α为调制度，通常为 0.5，$cos\, 2\omega_m t$代指基带调制信号，$cos\, 2\omega_c t$代指高频载波；

在 FPGA 内不存在浮点数、DAC 输出不会区别有无符号且 DDS 的幅度与数据位宽有关，需要将调制后信号截取高位使其位宽固定为 8 并加上一个直流分量让 DAC 实际输出无符号数，故上式在 FPGA 实际输出时需要调整为：

$$s(t)=\frac{2^{8-1}(2^8+2^{8-1}\, cos\, 2\omega_m t)\, cos\, 2\omega_c t}{2^{10}}+96$$

FM 单音调频可以表述为

$$s_{FM}(t)=A\, cos(\omega_c t+m_f\, sin\, \omega_m\, t)$$

m_f为调频系数其公式如下所示：

$$m_f=\frac{K_f A_m}{\omega_m}=\frac{\Delta\omega}{\omega_m}=\frac{\Delta f}{f_m}$$

其载波频率计算与 DDS 一致，我们选择 1.495KHz 为载波频率，频偏 7.8KHz，即$m_f=5.2$，相关频率输出可以表征为：

$$f_{out}=\frac{1M\ \Delta\theta}{2^{16}}$$

$$\Delta\theta\ =\ 98+4\times 128\, sin\, \omega_m\, t$$

AM 载波部分直接使用 DDS IP 核生成 20KHz 载波，AM、FM 频率没有选择太高是因为板载 DAC 最大工作频率有限，过高频率会直接导致 DAC 输出失真。

7. 音乐播放方案

通过改变方波频率来和占空比实现对音调和响度的控制，在控制程序内预先定义了一些基本音色的频率，在播放音乐时只需要输入指定参数就能够实现对方波的调整。

8. VGA 显示方案

显示屏逐行刷新过程中需要按照一定时序刷新，实际需要输入的地址多于有

效像素，行同步与场同步信号出现在 Sync 区间用于像素显示垂直同步。

EGO1 板卡所能够支持的 VGA 时序参数为 640*480@60Hz，其时序要求如下图所示：

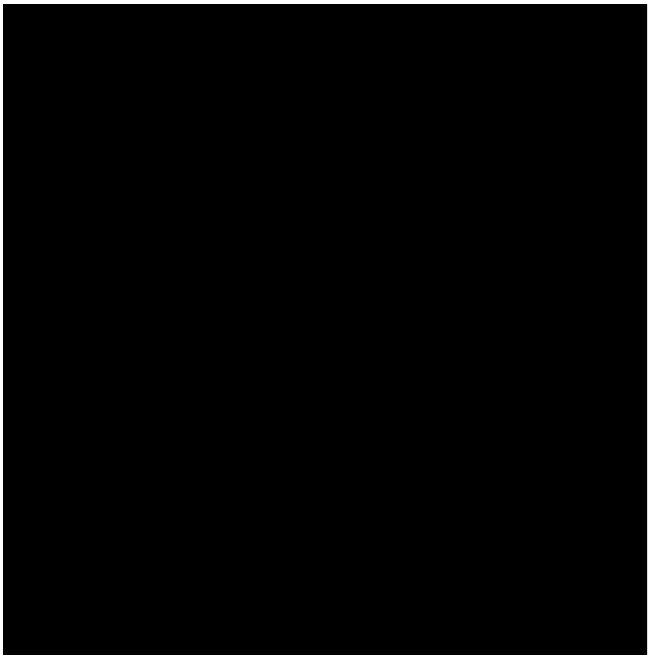


图 5 VGA 时序

因此，我们可以根据参数编写出合适代码，完成了彩条、灰度显示。

三．子模块原理

1. 分频模块

a) 设计思路

计数器计数到指定 cnt 数值时翻转寄存器数值，频率相当于$f_{out} = \frac{f_{clk}}{2 \times cnt}$

b) 源代码

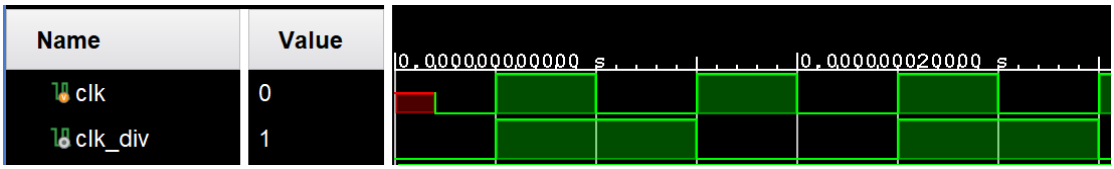
```
module clk_gen #(
    parameter cnt_div = 16'd5000
) (
    input  clk,
    output clk_div
);
  reg r_clk_div = 1'b0;
  assign clk_div = r_clk_div;
  reg [15:0] clk_div_cnt = 16'd0;
  always @(posedge clk) begin
    clk_div_cnt <= (clk_div_cnt == cnt_div - 1'b1) ? 16'b0 : clk_div_cnt + 1'b1;
    r_clk_div   <= (clk_div_cnt == cnt_div - 1'b1) ? ~r_clk_div : r_clk_div;
```

```
  end
endmodule
```

c) 仿真测试：

```
`timescale 1ns / 1ps
module clk_gen_tb ();
  reg  clk;
  wire clk_div;
  initial begin
    #2 clk = 0;
  end
  parameter CLK_FREQ = 100;  //Mhz
  always #(1000 / CLK_FREQ / 2) clk = ~clk;
  clk_gen #(
      .cnt_div(16'd1)
  ) u_clk_gen (
      .clk    (clk),
      .clk_div(clk_div)
  );
endmodule
```

仿真效果如下：



可以看到模块正常工作，成功将 100m 时钟分频为 50m。

2. 数码管模块

a) 数码管部分分割为三个模块：数码管顶层驱动，二进制转 BCD 码，BCD 码转数码管实际输出（Latency=1）。其中二进制转 BCD 码直接使用除法器是一种极为糟糕的写法，一是除法器占用资源多，二是除法器综合得出的电路是不能够具体确定的并且可能会有较高 Latency，在高频情况下出问题。虽然数码管时钟工作在 10k 低频情况下，且硬件逻辑资源 35k 非常充足，但是本方案仍然采用状态机控制的加三法来尽可能优化资源消耗。

假如有一个八位二进制数255,我把他转255的十进制数

0　　1111 1111　原数

1　　0000 0001　;左移一次

2　　0000 0011　; 左移二次

3　　0000 0111　;左移三次,检查低四位+3>7?

3.1　0000 1010　;大于7,加3进行调整

4　　0001 0101　;左移四次, 检查低四位+3>7?

4.1　0001 1000　;大于7,加3进行调整

5　　0011 0001　;左移五次

6　　0110 0011　;左移六次,检查高四位+3>7?

6.1　1001 0011　;大于7,加3进行调整

7　　1 0010 0111　;左移七次,检查低四位+3>7?

7.1　1 0010 1010　;大于7,加3进行调整

8　　10 0101 0101　;左移八次(得到BCD码255)

图 6 加三法示意图

b)　数码管源代码

数码管顶层驱动代码:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////
// Engineer: Infinite_Cloisters
// CopyRight: 922104780716 JingQiu Xu
// Create Date: 2024/10/19 20:37:45
// Module Name: seg
//////////////////////////////////////////////////////////////
module seg (
    input clk,
    input seg_en_i,

    input [15:0] bin_1_i,  //4 位数
    input [15:0] bin_2_i,  //4 位数
    input bin_valid_i,
    input        bcd_en_i,
    input [31:0] bcd_i,

    output [3:0] o_seg_1_cs,   //H vaild g2 c2 c1 h1
    output [7:0] o_seg_1_data,
    output [3:0] o_seg_2_cs,   //H vaild g1 f1 e1 g6
    output [7:0] o_seg_2_data
);
  localparam BIT1 = 4'b1000;
  localparam BIT2 = 4'b0100;
  localparam BIT3 = 4'b0010;
  localparam BIT4 = 4'b0001;

  wire [31:0] w_bcd_tmp;
  reg  [31:0] r_bcd_o = 32'd0;
```

```
wire        w_bcd_vaild_1_o;
wire        w_bcd_vaild_2_o;
reg  [ 3:0] r_seg_1_bcd;
reg  [ 3:0] r_seg_2_bcd;
wire w_bin_valid = seg_en_i & bin_valid_i;

BIN2BCD u_BIN2BCD_1 (
    .clk        (clk),
    .bin_valid_i(w_bin_valid),  //Start Convert
    .bin_i      (bin_1_i),

    .bcd_o      (w_bcd_tmp[31:16]),
    .bcd_vaild_o(w_bcd_vaild_1_o)    //Data Vaild
);
BIN2BCD u_BIN2BCD_2 (
    .clk        (clk),
    .bin_valid_i(w_bin_valid),  //Start Convert
    .bin_i      (bin_2_i),

    .bcd_o      (w_bcd_tmp[15:0]),
    .bcd_vaild_o(w_bcd_vaild_2_o)   //Data Vaild
);
always @(posedge clk) begin
  if (seg_en_i == 1) begin
    if (bcd_en_i == 0) begin
      if (w_bcd_vaild_1_o) begin
        r_bcd_o[31:16] <= w_bcd_tmp[31:16];
      end
      if (w_bcd_vaild_2_o) begin
        r_bcd_o[15:0] <= w_bcd_tmp[15:0];
      end
    end else if (bcd_en_i == 1) begin
      r_bcd_o <= bcd_i;
    end else begin
    end
  end else begin
    r_bcd_o <= 32'd0;
  end
end
reg [3:0] r_seg_cs = 4'b1000; //共用一个 CS 信号
reg [7:0] r_seg_cs_dly = 8'd0; //seg 赋值一拍，转换 bcd 一拍，需要延时 2 拍
always @(posedge clk) begin
  r_seg_cs_dly <= {r_seg_cs_dly[3:0], r_seg_cs};
end
assign o_seg_1_cs = r_seg_cs_dly[7:4];
assign o_seg_2_cs = r_seg_cs_dly[7:4];
//状态机实现扫描
always @(posedge clk) begin
```

```
    case (r_seg_cs)
      BIT1: begin
        r_seg_1_bcd <= r_bcd_o[31:28];
        r_seg_cs    <= BIT2;
        r_seg_2_bcd <= r_bcd_o[15:12];
      end
      BIT2: begin
        r_seg_1_bcd <= r_bcd_o[27:24];
        r_seg_cs    <= BIT3;
        r_seg_2_bcd <= r_bcd_o[11:8];
      end
      BIT3: begin
        r_seg_1_bcd <= r_bcd_o[23:20];
        r_seg_cs    <= BIT4;
        r_seg_2_bcd <= r_bcd_o[7:4];
      end
      BIT4: begin
        r_seg_1_bcd <= r_bcd_o[19:16];
        r_seg_cs    <= BIT1;
        r_seg_2_bcd <= r_bcd_o[3:0];
      end
      default: begin
        r_seg_1_bcd <= 4'b0;
        r_seg_2_bcd <= 4'b0;
        r_seg_cs    <= BIT1;
      end
    endcase
  end
  BCD2SEG u_BCD2SEG_1 (
      .clk(clk),
      .BCD(r_seg_1_bcd),
      .DP (1'b0),
      .SEG(o_seg_1_data)
  );
  BCD2SEG u_BCD2SEG_2 (
      .clk(clk),
      .BCD(r_seg_2_bcd),
      .DP (1'b0),
      .SEG(o_seg_2_data)
  );
endmodule
```

数码管二进制转 BCD 代码:

```
module BIN2BCD #(
    parameter INPUT_WIDTH = 16,
    parameter DECIMAL_DIGITS = 4
) (
    input                                clk,
    input  [       INPUT_WIDTH - 1 : 0] bin_i,
```

```
    input                               bin_valid_i,
    output [DECIMAL_DIGITS * 4 - 1 : 0] bcd_o,
    output                              bcd_vaild_o
);
  localparam s_IDLE = 3'b000;
  localparam s_SHIFT = 3'b001;
  localparam s_CHECK_SHIFT_INDEX = 3'b010;
  localparam s_ADD = 3'b011;
  localparam s_CHECK_DIGIT_INDEX = 3'b100;
  localparam s_BCD_DONE = 3'b101;

  reg  [                    2:0] r_SM_Main = s_IDLE;
  // The vector that contains the output BCD
  reg  [DECIMAL_DIGITS*4 - 1 : 0] r_BCD = 0;
  // The vector that contains the input binary value being shifted.
  reg  [         INPUT_WIDTH-1:0] r_BIN = 0;
  // Keeps track of which Decimal Digit we are indexing
  reg  [      DECIMAL_DIGITS-1:0] r_Digit_Index = 0;
  // Keeps track of which loop iteration we are on.
  // Number of loops performed = INPUT_WIDTH
  reg  [                    7:0] r_Loop_Count = 0;

  wire [                    3:0] w_BCD_Digit;
  reg                            r_bcd_vaild = 1'b0;

  always @(posedge clk) begin
    case (r_SM_Main)
      // Stay in this state until bin_valid_i comes along
      s_IDLE: begin
        r_bcd_vaild <= 1'b0;
        if (bin_valid_i == 1'b1) begin
          r_BIN     <= bin_i;
          r_SM_Main <= s_SHIFT;
          r_BCD     <= 0;
        end else r_SM_Main <= s_IDLE;
      end
      // Always shift the BCD Vector until we have shifted all bits through
      // Shift the most significant bit of r_BIN into r_BCD lowest bit.
      s_SHIFT: begin
        r_BCD     <= r_BCD << 1;
        r_BCD[0]  <= r_BIN[INPUT_WIDTH-1];
        r_BIN     <= r_BIN << 1;
        r_SM_Main <= s_CHECK_SHIFT_INDEX;
      end
      // Check if we are done with shifting in r_BIN vector
      s_CHECK_SHIFT_INDEX: begin
        if (r_Loop_Count == INPUT_WIDTH - 1) begin
          r_Loop_Count <= 0;
```

```
          r_SM_Main    <= s_BCD_DONE;
        end else begin
          r_Loop_Count <= r_Loop_Count + 1;
          r_SM_Main    <= s_ADD;
        end
      end
      // Break down each BCD Digit individually.  Check them one-by-one to
      // see if they are greater than 4.  If they are, increment by 3.
      // Put the result back into r_BCD Vector.
      s_ADD: begin
        if (w_BCD_Digit > 4) begin
          r_BCD[(r_Digit_Index*4)+:4] <= w_BCD_Digit + 3;
        end

        r_SM_Main <= s_CHECK_DIGIT_INDEX;
      end
      // Check if we are done incrementing all of the BCD Digits
      s_CHECK_DIGIT_INDEX: begin
        if (r_Digit_Index == DECIMAL_DIGITS - 1) begin
          r_Digit_Index <= 0;
          r_SM_Main     <= s_SHIFT;
        end else begin
          r_Digit_Index <= r_Digit_Index + 1;
          r_SM_Main     <= s_ADD;
        end
      end
      s_BCD_DONE: begin
        r_bcd_vaild <= 1'b1;
        r_SM_Main   <= s_IDLE;
      end
      default: r_SM_Main <= s_IDLE;
    endcase
  end  // always @ (posedge clk)

  assign w_BCD_Digit = r_BCD[r_Digit_Index*4+:4];
  assign bcd_o       = r_BCD;
  assign bcd_vaild_o = r_bcd_vaild;
endmodule  // Binary_tbcd_o
```

数码管 BCD 转显示代码

```
module BCD2SEG (
    input        clk,
    input  [3:0] BCD,
    input        DP,
    output [7:0] SEG
);
  reg [7:0] SEG_REG;
  assign SEG = SEG_REG;
  //abcd_efg_dp
```

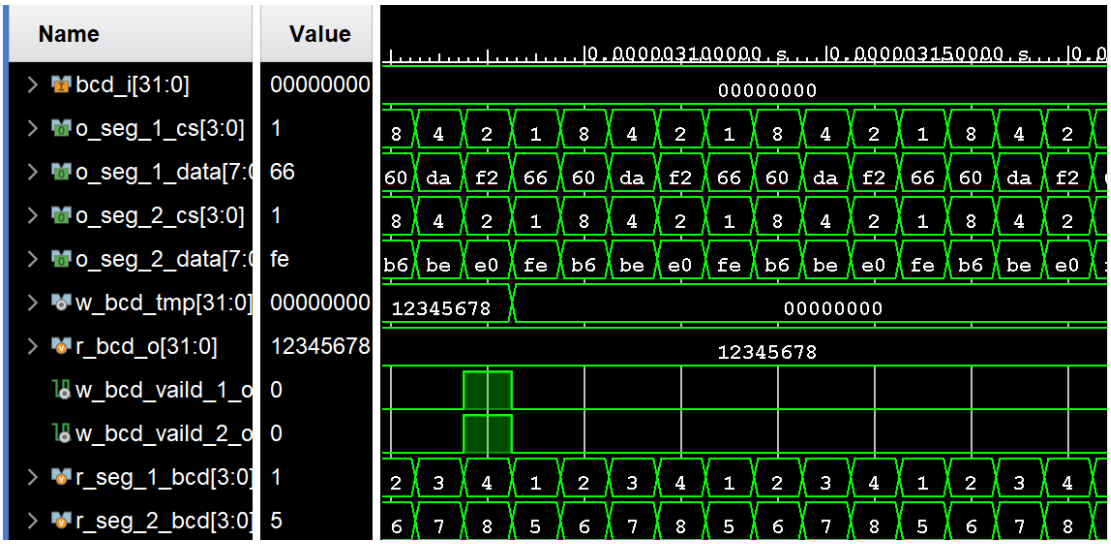
```
  always @(posedge clk) begin
    case (BCD)
      4'b0000: SEG_REG <= 8'b1111_110_0 + DP;  //0
      4'b0001: SEG_REG <= 8'b0110_000_0 + DP;  //1
      4'b0010: SEG_REG <= 8'b1101_101_0 + DP;  //2
      4'b0011: SEG_REG <= 8'b1111_001_0 + DP;  //3
      4'b0100: SEG_REG <= 8'b0110_011_0 + DP;  //4
      4'b0101: SEG_REG <= 8'b1011_011_0 + DP;  //5
      4'b0110: SEG_REG <= 8'b1011_111_0 + DP;  //6
      4'b0111: SEG_REG <= 8'b1110_000_0 + DP;  //7
      4'b1000: SEG_REG <= 8'b1111_111_0 + DP;  //8
      4'b1001: SEG_REG <= 8'b1111_011_0 + DP;  //9
      4'b1010: SEG_REG <= 8'b1110_111_0 + DP;  //a
      4'b1011: SEG_REG <= 8'b0011_111_0 + DP;  //b
      4'b1100: SEG_REG <= 8'b1001_111_0 + DP;  //c
      4'b1101: SEG_REG <= 8'b0111_101_0 + DP;  //d
      4'b1110: SEG_REG <= 8'b1101_111_0 + DP;  //e
      4'b1111: SEG_REG <= 8'b1000_111_0 + DP;  //f
      default: SEG_REG <= 8'h00;
    endcase
  end
endmodule
```

c) 数码管仿真：

```
module seg_tb ();
  reg         clk;
  reg         seg_en_i;
  reg  [15:0] bin_1_i;
  reg  [15:0] bin_2_i;
  reg         bin_valid_i;
  reg         bcd_en_i;
  reg  [31:0] bcd_i;
  wire [ 3:0] o_seg_1_cs;
  wire [ 7:0] o_seg_1_data;
  wire [ 3:0] o_seg_2_cs;
  wire [ 7:0] o_seg_2_data;
  initial begin
    #2
    clk = 0;
    seg_en_i = 1;
    bin_valid_i = 1;
    bcd_en_i = 0;
    bcd_i = 0;
    bin_1_i = 1234;
    bin_2_i = 5678;
  end
  parameter CLK_FREQ = 100;  //Mhz
  always #(1000 / CLK_FREQ / 2) clk = ~clk;
  seg u_seg (
```

```
    .clk         (clk),
    .seg_en_i    (seg_en_i),
    .bin_1_i     (bin_1_i),       // 4 位数
    .bin_2_i     (bin_2_i),       // 4 位数
    .bin_valid_i (bin_valid_i),
    .bcd_en_i    (bcd_en_i),
    .bcd_i       (bcd_i),
    .o_seg_1_cs  (o_seg_1_cs),    // H vaild g2 c2 c1 h1
    .o_seg_1_data(o_seg_1_data),
    .o_seg_2_cs  (o_seg_2_cs),    // H vaild g1 f1 e1 g6
    .o_seg_2_data(o_seg_2_data)
);
endmodule
```

可以看到在 BCD 有效上升沿后，数码管数据输出开始按照正确次序循环刷新



3. DDS 输出模块

a) 设计思路

本模块正常输出与 AM 调制输出复用一个输出端口，通过 am_en 信号控制具体输出内容，此外，模块预留了部分接口用于改变输出相位与幅度。fm_en 信号用于控制 fm 调频信号的输出。

b) 源代码

```
`timescale 1ns / 1ps
module dds (
    input clk,
    input clk_am,  //clk 100m
    input dds_en_i,
    input [ 3:0] type_i,   //input [3:0] type_i
    input [ 7:0] Freq_i,   //input [7:0] Freq_i
    input [ 7:0] Phase_i,  //input [7:0] Phase_i
    input [15:0] Amp_i,    //input [15:0] Amp_i
    input am_en,  // input am_en
    input fm_en,
    output [7:0] wave_o,    //output [7:0] wave_o
    output [7:0] wave_fm_o
```

```
);
  assign clka = clk;
  //使能信号产生
  wire w_sin_en = ((type_i == 4'd0) && dds_en_i) ? 1'b1 : 1'b0;
  wire w_tri_en = ((type_i == 4'd2) && dds_en_i) ? 1'b1 : 1'b0;
  wire w_squ_en = ((type_i == 4'd1) && dds_en_i) ? 1'b1 : 1'b0;
  wire w_saw_en = ((type_i == 4'd3) && dds_en_i) ? 1'b1 : 1'b0;

  wire signed [ 7:0] w_sin;
  wire signed [ 7:0] w_squ;
  wire signed [ 7:0] w_tri;
  wire signed [ 7:0] w_saw;
  wire signed [ 7:0] w_dds_am_data;
  wire               w_dds_am_data_valid;
  wire signed [ 7:0] w_amp;

  reg signed  [ 7:0] r_wave_o = 8'd0;
  reg signed  [19:0] r_wave_am = 20'd0;
  reg signed  [15:0] r_wave_add = 16'd0;  //am 调制先相加
  reg signed  [ 7:0] r_wave_dly = 8'd0;
  assign wave_o     = r_wave_dly;
  assign wave_dds_o = w_dds_am_data;
  assign wave_am_o  = r_wave_am;
  assign w_amp      = Amp_i;
  //数据选择并打一拍，以防亚稳态
  always @(posedge clk) begin
    r_wave_dly <= r_wave_o;
    if (am_en == 0) begin
      case (type_i)
        4'd0: begin
          r_wave_o <= w_sin + 8'd128;
        end
        4'd1: begin
          r_wave_o <= w_squ + 8'd128;
        end
        4'd2: begin
          r_wave_o <= w_tri + 8'd128;
        end
        4'd3: begin
          r_wave_o <= w_saw + 8'd128;
        end
      endcase
    end else begin
      r_wave_o <= r_wave_am[19:9] + 10'd96;  //截位
      case (type_i)
        4'd0: begin
          r_wave_add = w_sin + w_amp * 128;
          r_wave_am  = r_wave_add * w_dds_am_data;
```

```
        end
        4'd1: begin
          r_wave_add = w_squ + w_amp * 128;
          r_wave_am  = r_wave_add * w_dds_am_data;
        end
        4'd2: begin
          r_wave_add = w_tri + w_amp * 128;
          r_wave_am  = r_wave_add * w_dds_am_data;
        end
        4'd3: begin
          r_wave_add = w_saw + w_amp * 128;
          r_wave_am  = r_wave_add * w_dds_am_data;
        end
      endcase
    end
end

//地址维护
reg [14:0] r_addr = 15'd0;
reg [ 7:0] r_addr_true = 8'd0;  //with phase
always @(posedge clk) begin
  r_addr      <= r_addr + Freq_i;
  r_addr_true <= r_addr[14:7] + Phase_i;
end
rom_sin u_rom_sin (
    .clka (clka),         // input wire clka
    .ena  (w_sin_en),     // input wire ena
    .addra(r_addr_true),  // input wire [7 : 0] addra
    .douta(w_sin)         // output wire [7 : 0] douta
);
rom_tri u_rom_tri (
    .clka (clka),         // input wire clka
    .ena  (w_tri_en),     // input wire ena
    .addra(r_addr_true),  // input wire [7 : 0] addra
    .douta(w_tri)         // output wire [7 : 0] douta
);
rom_squ u_rom_squ (
    .clka (clka),         // input wire clka
    .ena  (w_squ_en),     // input wire ena
    .addra(r_addr_true),  // input wire [7 : 0] addra
    .douta(w_squ)         // output wire [7 : 0] douta
);
rom_saw u_rom_saw (
    .clka (clka),         // input wire clka
    .ena  (w_saw_en),     // input wire ena
    .addra(r_addr_true),  // input wire [7 : 0] addra
    .douta(w_saw)         // output wire [7 : 0] douta
);
```

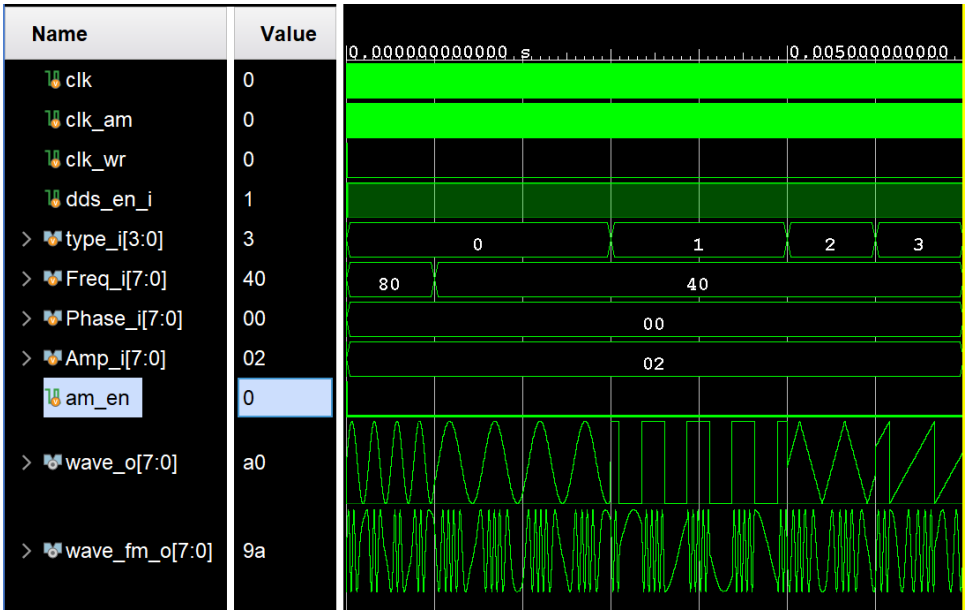
```
  dds_am u_dds_am (
      .aclk  (clk_am),  // input wire aclk
      .aclken(am_en),   // input wire aclken
      .m_axis_data_tvalid(w_dds_am_data_valid),
      .m_axis_data_tdata (w_dds_am_data)
  );
  wire [7:0] w_wave_fm;
  reg [15:0] r_wave_fm_dly = 16'd0;
  dds_fm u_dds_fm (
      .aclk  (clk),  // input wire aclk
      .aclken(fm_en),   // input wire aclken
      .s_axis_phase_tvalid(1'd1),
      .s_axis_phase_tdata (16'd98 + {6'd0, r_wave_dly[7:0],2'd0}),
      .m_axis_data_tvalid(m_axis_data_tvalid),
      .m_axis_data_tdata (w_wave_fm)
  );
  //signed -> unsigned
  always @(posedge clk) begin
    r_wave_fm_dly <= {r_wave_fm_dly[7:0],~w_wave_fm[7], w_wave_fm[6:0]};
  end
  assign wave_fm_o = r_wave_fm_dly[15:8];
endmodule
```

c) testbench 代码:

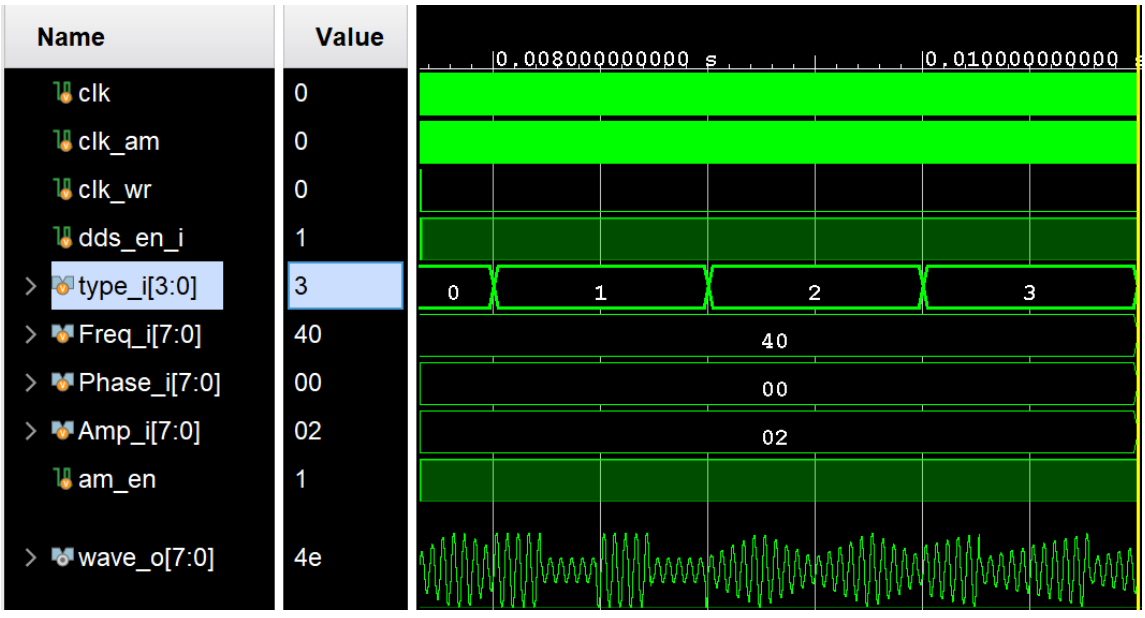
```
module dds_tb ();
  reg        clk;
  reg        clk_am;
  reg        clk_wr;
  reg        dds_en_i;
  reg  [3:0] type_i;
  reg  [7:0] Freq_i;
  reg  [7:0] Phase_i;
  reg  [7:0] Amp_i;
  reg        am_en;
  wire [7:0] wave_o;
  wire [7:0] wave_fm_o;
  initial begin
    #2 clk = 0;
    clk_am = 0;
    clk_wr = 0;
    #10 dds_en_i = 1;
    type_i  = 0;
    Freq_i  = 128;
    Amp_i   = 2;
    am_en   = 0;
    Phase_i = 0;
  End
  always #5 clk_am = ~clk_am;  //100m
  always #500 clk = ~clk;  //1m
```

```
  dds u_dds (
      .clk     (clk),
      .clk_am  (clk_am),    // clk 100m
      .dds_en_i(dds_en_i),
      .type_i  (type_i),    // input [3:0] type_i
      .Freq_i  (Freq_i),    // input [7:0] Freq_i
      .Phase_i (Phase_i),   // input [7:0] Phase_i
      .Amp_i   (Amp_i),     // input [7:0] Amp_i
      .am_en   (am_en),     // input am_en
      .fm_en   (1'd1),
      .wave_o   (wave_o),    // output [7:0] wave_o
      .wave_fm_o(wave_fm_o)
  );
endmodule
```

DDS 使用 testbench 仿真效果如下:



其中 type_i 表示了波形的种类，0 为正弦波，1 为方波，2 为三角波，4 为锯齿波



可见仿真效果正常。

4. 闸门法测频率模块
   a) 设计思路

      选择信号最高位上升边沿作为触发条件，闸门只有在时钟高电平时开启，内置计数器开始计数；时钟低电平时，计数器将数据转移到输出寄存器内。

   b) 源代码

```
module measure_freq (
    input         clk_0d5Hz,
    input  [ 7:0] signal_i,
    output [15:0] freq_o
);
  reg [15:0] r_freq_cnt = 16'd0;
  reg [15:0] r_freq_cnt_dly = 16'd0;
  reg [15:0] r_freq = 16'd0;
  always @(posedge signal_i[7]) begin
    if (clk_0d5Hz == 1) begin
      r_freq_cnt     = r_freq_cnt + 1'b1;
      r_freq_cnt_dly = r_freq_cnt;
    end else begin
      r_freq_cnt <= 16'd0;
      r_freq     <= r_freq_cnt_dly;
    end
  end
  assign freq_o = r_freq;
endmodule
```

5. 按键消除抖设计
   a) 设计原理

      采用连续打拍方式来实现消抖，当所有打拍节点数值均一致时认为状态处于稳定状态。

   b) 源代码

```
module btn_debiting (
    input clk,
    input rstn,
    input i_btn_up,     //u4
    input i_btn_down,   //r17
    input i_btn_left,   //v1
    input i_btn_right,  //r11
    input i_btn_mid,    //r15

    output btn_up_o,     //u4
    output btn_down_o,   //r17
    output btn_left_o,   //v1
    output btn_right_o,  //r11
    output btn_mid_o     //r15
);
  reg [3:0] btn_up_dly;
  reg [3:0] btn_down_dly;
```

```
  reg [3:0] btn_left_dly;
  reg [3:0] btn_right_dly;
  reg [3:0] btn_mid_dly;
  always @(posedge clk or negedge rstn) begin
    if (rstn == 0) begin
      btn_up_dly    <= 4'd0;
      btn_down_dly  <= 4'd0;
      btn_left_dly  <= 4'd0;
      btn_right_dly <= 4'd0;
      btn_mid_dly   <= 4'd0;
    end else begin
      btn_up_dly    <= {btn_up_dly[2:0], i_btn_up};
      btn_down_dly  <= {btn_down_dly[2:0], i_btn_down};
      btn_left_dly  <= {btn_left_dly[2:0], i_btn_left};
      btn_right_dly <= {btn_right_dly[2:0], i_btn_right};
      btn_mid_dly   <= {btn_mid_dly[2:0], i_btn_mid};
    end
  end
  assign btn_up_o    = btn_up_dly == 4'hf;
  assign btn_down_o  = btn_down_dly == 4'hf;
  assign btn_left_o  = btn_left_dly == 4'hf;
  assign btn_right_o = btn_right_dly == 4'hf;
  assign btn_mid_o   = btn_mid_dly == 4'hf;
endmodule
```

6. 模式选择设计

a) 设计原理

模式数目可以通过例化时改变 parameter 的数值来更改，其需要输入为消抖后按键状态，输出口输出当前模式与种类，与此同时，模块也引出了属于输入时钟时钟域的按键上升沿输出。

b) 源代码

```
module mode_sel #(
    parameter MODE_MAX = 4'd2,
    parameter TYPE_MAX = 4'd2
) (
    input clk,
    input rstn,
    input btn_up_i,
    input btn_down_i,
    input btn_left_i,
    input btn_right_i,
    input btn_mid_i,

    output [3:0] mode_o,
    output [3:0] type_o,
    output btn_mid_pos,
    output btn_up_pos,
    output btn_down_pos,
    output btn_left_pos,
```

```
    output btn_right_pos
);
  //flip flop generate
  reg [3:0] btn_up_dly;
  reg [3:0] btn_down_dly;
  reg [3:0] btn_left_dly;
  reg [3:0] btn_right_dly;
  reg [3:0] btn_mid_dly;
  always @(posedge clk or negedge rstn) begin
    if (rstn == 0) begin
      btn_up_dly    <= 4'd0;
      btn_down_dly  <= 4'd0;
      btn_left_dly  <= 4'd0;
      btn_right_dly <= 4'd0;
      btn_mid_dly   <= 4'd0;
    end else begin
      btn_up_dly    <= {btn_up_dly[2:0], btn_up_i};
      btn_down_dly  <= {btn_down_dly[2:0], btn_down_i};
      btn_left_dly  <= {btn_left_dly[2:0], btn_left_i};
      btn_right_dly <= {btn_right_dly[2:0], btn_right_i};
      btn_mid_dly   <= {btn_mid_dly[2:0], btn_mid_i};
    end
  end
  assign btn_up_pos    = ~btn_up_dly[3] & btn_up_dly[2];
  assign btn_down_pos  = ~btn_down_dly[3] & btn_down_dly[2];
  assign btn_left_pos  = ~btn_left_dly[3] & btn_left_dly[2];
  assign btn_right_pos = ~btn_right_dly[3] & btn_right_dly[2];
  assign btn_mid_pos   = ~btn_mid_dly[3] & btn_mid_dly[2];
  reg [3:0] r_mode = 0;
  reg [3:0] r_type = 0;
  assign mode_o = r_mode;
  assign type_o = r_type;
  always @(posedge clk or negedge rstn) begin
    if (rstn == 0) begin
      r_mode <= 4'd0;
      r_type <= 4'd0;
    end else if (btn_up_pos) begin
      r_mode <= (r_mode == MODE_MAX) ? MODE_MAX : r_mode + 1'b1;
    end else if (btn_down_pos) begin
      r_mode <= (r_mode == 0) ? 4'd0 : r_mode - 1'b1;
    end else if (btn_right_pos) begin
      r_type <= (r_type == TYPE_MAX) ? TYPE_MAX : r_type + 1'b1;
    end else if (btn_left_pos) begin
      r_type <= (r_type == 4'd0) ? 4'd0 : r_type - 1'b1;
    end else begin
    end
  end
endmodule
```

7. 流水灯设计

a) 设计原理

方案内为了便于观察当前具体种类，设计了四种不同流水灯来表征当前功能，别且流水灯均带有呼吸灯性质。

b) 源代码

```
module led_waterlamps (
    input       clk_1k,
    input [7:0] i_sw,
    input [3:0] type_i,

    output [7:0] o_leds
);
  reg [7:0] r_leds = 8'd0;
  reg [7:0] r_leds_buf1;
  reg [7:0] r_leds_buf2;
  reg [2:0] r_led_pos = 3'd0;
  assign o_leds = r_leds;

  reg [8:0] r_tim_cnt = 9'd0;
  reg [8:0] r_tim = 9'd1;
  reg [8:0] r_tim_dly = 9'd1;
  reg [3:0] r_type = 4'd0;
  reg [1:0] r_dir = 2'd0;

  localparam S_LINE = 4'd0;
  localparam S_BUMP1 = 4'd1;
  localparam S_BUMP2 = 4'd2;
  localparam S_CROS = 4'd3;

  wire [4:0] map[0:7];
  assign map[0] = 5'b11111;
  assign map[1] = 5'b10000;
  assign map[2] = 5'b01000;
  assign map[3] = 5'b00000;
  assign map[7] = 5'b11111;
  assign map[6] = 5'b10000;
  assign map[5] = 5'b01000;
  assign map[4] = 5'b00000;
  always @(posedge clk_1k) begin
    //update sw
    r_tim     <= {4'b0_001, map[r_led_pos]};
    r_tim_dly <= r_tim;
    r_type    <= type_i;
    if (r_type == type_i) begin
      if (r_tim_cnt >= r_tim_dly) begin
        //计时完毕
        case (type_i)
          S_LINE: begin
```

```
            r_leds[r_led_pos] <= ~r_leds[r_led_pos];
          end
          S_BUMP1: begin
            if (r_led_pos == 4'd7) begin
              r_dir <= r_dir + 1'b1;
            end else begin
            end
            if (r_dir[1] == 0) begin
              r_leds[r_led_pos] <= ~r_leds[r_led_pos];
            end else begin
              r_leds[3'd7-r_led_pos] <= ~r_leds[3'd7-r_led_pos];
            end
          end
          S_BUMP2: begin
            if (r_led_pos == 4'd7) begin
              r_dir <= r_dir + 1'b1;
            end else begin
            end
            if (r_dir[1] == 0) begin
              if (r_dir[0] == 0) begin
                r_leds[r_led_pos] <= ~r_leds[r_led_pos];
              end else begin
                r_leds[3'd7-r_led_pos] <= ~r_leds[3'd7-r_led_pos];
              end
            end else begin
              if (r_dir[0] == 0) begin
                r_leds[3'd7-r_led_pos] <= ~r_leds[3'd7-r_led_pos];
              end else begin
                r_leds[r_led_pos] <= ~r_leds[r_led_pos];
              end
            end
          end
          S_CROS: begin
            if (r_leds == 0) begin
              r_leds      <= 8'b1000_0001;
              r_leds_buf1 <= 8'b1000_0000;
              r_leds_buf2 <= 8'b0000_0001;
            end else begin
              if (r_dir[0] == 0) begin
                r_leds_buf1 <= {r_leds_buf1[0], r_leds_buf1[7:1]};  // >>
                r_leds_buf2 <= {r_leds_buf2[6:0], r_leds_buf2[7]};  // <<
                r_leds      <= r_leds_buf1 | r_leds_buf2;
              end else begin
              end
              r_dir <= r_dir + 1'b1;
            end
          end
        endcase
```

```
        r_tim_cnt <= 9'd0;
        r_led_pos <= r_led_pos + 1'b1;
      end else begin
        r_tim_cnt <= r_tim_cnt + 1'b1;
      end
    end else begin
      r_leds    <= 8'd0;
      r_led_pos <= 3'd0;
      r_dir = 2'd0;
    end
  end
endmodule
```

```
module pwm_lite (
    input clk,
    input [7:0] duty,  //input [7:0] duty range (0,100)

    output [7:0] pwm  // freq = clk /100
);
  reg [7:0] r_pwm = 8'b0;
  assign pwm = r_pwm;
  reg [7:0] r_pwm_cnt = 8'd0;
  always @(posedge clk) begin
    r_pwm_cnt <= (r_pwm_cnt == 8'd99) ? 8'd0 : r_pwm_cnt + 1'b1;
    r_pwm     <= {8{r_pwm_cnt <= duty}};
  end
endmodule
```

```
module led_pwm (
    input clk,
    input clk_1k,
    input [7:0] i_sw,

    output [7:0] pwm_o,
    output [7:0] o_leds
);
  wire [7:0] w_leds;
  assign pwm_o  = w_leds;
  assign o_leds = w_leds & i_sw;
  reg [7:0] r_duty = 8'd0;
  pwm_lite u_pwm_lite (
      .clk (clk),
      .duty(r_duty),
      .pwm (w_leds)
  );
  reg       r_pwm_dir = 1'b0;
  reg [3:0] r_pwm_duty_dly = 4'd0;
  always @(posedge clk_1k) begin
    r_pwm_duty_dly <= r_pwm_duty_dly + 1'b1;
    if (r_pwm_duty_dly == 4'hf) begin
```

```
      r_duty <= (r_pwm_dir == 0) ? r_duty + 1'b1 : r_duty - 1'b1;
      if (r_duty == 8'd64) begin
        r_pwm_dir <= 1'b1;
      end else if (r_duty == 8'd1) begin
        r_pwm_dir <= 1'b0;
      end else begin
      end
    end else begin
    end
  end
endmodule
```

PWM 仿真代码

```
module led_driver_tb ();
  reg        clk;
  reg        clk_2;
  wire [7:0] w_led;
  initial begin
    #2 clk =0;
    clk_2 = 0;
  end
  always #(1000 / 250 / 2) clk = ~clk;
  always #(1000 / 1 / 2) clk_2 = ~clk_2;
  led_pwm u_pwm (
      .clk   (clk),
      .clk_1k(clk_2),
      .i_sw  (8'hff),
      .o_leds(w_led)
  );
endmodule
```

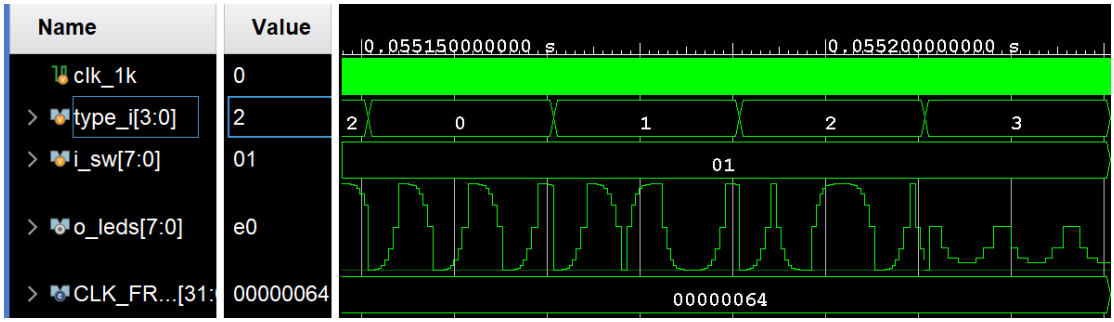
PWM 仿真结果如下，通过拖动时间轴可以看到占空比从低到高再到低的过程，表示呼吸灯实现完成

流水灯仿真代码

```
module led_waterlamps_tb ();
  reg        clk_1k;
  reg [3:0] type_i;
```

```
  reg  [7:0] i_sw;
  wire [7:0] o_leds;
  initial begin
    #2 clk_1k = 0;
    i_sw = 8'd1;
    type_i = 0;
  end
  always #(1000 / 100 / 2) clk_1k = ~clk_1k;
  led_waterlamps u_led_waterlamps (
      .clk_1k(clk_1k),
      .type_i(type_i),
      .i_sw  (i_sw),
      .o_leds(o_leds)
  );
endmodule
```

四种模式流水灯仿真效果



8. 串口设计

串口部分由顶层驱动模块、分频器模块、串口接收模块、串口发生模块组成

```
module uart_transfer (
    //gobal clock
    input               clk,
    input               rst_n,
    //uart interface
    input               clken_16bps,//clk_bps * 16
    output  reg         txd,        //uart txd interface
    //user interface
    input               txd_en,     //uart data transfer enable
    input       [7:0]   txd_data,   //uart transfer data
    output  reg         txd_flag    //uart data transfer done
);
/*************************************************************************
..IDLE...Start.............UART DATA.....................End...IDLE...
--------                                                 --------------
        |____< D0 >< D1 >< D2 >< D3 >< D4 >< D5 >< D6 >< D7 >
        Bit0  Bit1  Bit2  Bit3  Bit4  Bit5  Bit6  Bit7  Bit8  Bit9
*************************************************************************/
//parameter of uart transfer
localparam  T_IDLE  =   2'b0;   //test the flag to transfer data
localparam  T_SEND  =   2'b1;   //uart transfer data
```

```
reg [1:0]   txd_state;          //uart transfer state
reg [3:0]   smp_cnt;            //16 * clk_bps, the center for sample
reg [3:0]   txd_cnt;            //txd data counter
localparam  SMP_TOP     =   4'd15;
localparam  SMP_CENTER  =   4'd7;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        smp_cnt <= 0;
        txd_cnt <= 0;
        txd_state <= T_IDLE;
        end
    else
        begin
        case(txd_state)
        T_IDLE:
            begin
            smp_cnt <= 0;
            txd_cnt <= 0;
            if(txd_en == 1)
                txd_state <= T_SEND;
            else
                txd_state <= T_IDLE;
            end
        T_SEND:
            begin
            if(clken_16bps == 1)
                begin
                smp_cnt <= smp_cnt + 1'b1;
                if(smp_cnt == SMP_TOP)
                    begin
                    if(txd_cnt < 4'd9)
                        begin
                        txd_cnt <= txd_cnt + 1'b1;
                        txd_state <= T_SEND;
                        end
                    else
                        begin
                        txd_cnt <= 0;
                        txd_state <= T_IDLE;
                        end
                    end
                else
                    begin
                    txd_cnt <= txd_cnt;
                    txd_state <= txd_state;
                    end
```

```
                end
            else
                begin
                txd_cnt <= txd_cnt;
                txd_state <= txd_state;
                end
            end
        endcase
        end
end
//uart 8 bit data transfer
always@(*)
begin
    if(txd_state == T_SEND)
        case(txd_cnt)
        4'd0:   txd = 0;
        4'd1:   txd = txd_data[0];
        4'd2:   txd = txd_data[1];
        4'd3:   txd = txd_data[2];
        4'd4:   txd = txd_data[3];
        4'd5:   txd = txd_data[4];
        4'd6:   txd = txd_data[5];
        4'd7:   txd = txd_data[6];
        4'd8:   txd = txd_data[7];
        4'd9:   txd = 1;
        default:txd = 1;
        endcase
    else
        txd = 1'b1; //default state
end
//Capture the falling of data transfer over
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        txd_flag <= 0;
    else if(clken_16bps == 1 &&  txd_cnt == 4'd9 && smp_cnt == SMP_TOP) //Totally 8
data is done
        txd_flag <= 1;
    else
        txd_flag <= 0;
end
endmodule
```

```
`timescale 1 ns / 1 ns
module uart_receiver
(
    //gobal clock
    input               clk,
    input               rst_n,
```

```
    //user interface
    input               clken_16bps,//clk_bps * 16
    //uart interface
    input               rxd,        //uart txd interface
    output  reg [7:0]   rxd_data,   //uart data receive
    output  reg         rxd_flag    //uart data receive done
);
/*****************************************************************************
..IDLE...Start.............UART DATA.....................End...IDLE...
--------                                                 --------------
        |____< D0 >< D1 >< D2 >< D3 >< D4 >< D5 >< D6 >< D7 >
        Bit0  Bit1  Bit2  Bit3  Bit4  Bit5  Bit6  Bit7  Bit8  Bit9
*****************************************************************************/
//sync the rxd data: rxd_sync
reg rxd_sync;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        rxd_sync <= 1;
    else
        rxd_sync <= rxd;
end
//parameter of uart transfer
localparam  R_IDLE      =   2'd0; //detect if the uart data is begin
localparam  R_START     =   2'd1; //uart transfert start mark bit
localparam  R_SAMPLE    =   2'd2; //uart 8 bit data receive
localparam  R_STOP      =   2'd3; //uart transfer stop mark bit
localparam  SMP_TOP     =   4'd15;
localparam  SMP_CENTER  =   4'd7;
reg [1:0]   rxd_state;                  //uart receive state
reg [3:0]   rxd_cnt;                    //uart 8 bit data counter
reg [3:0]   smp_cnt;                    //16 * clk_bps, the center for sample
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        smp_cnt <= 0;
        rxd_cnt <= 0;
        rxd_state <= R_IDLE;
        end
    else
        case(rxd_state)
        R_IDLE:     //Wait for start bit
            begin
            rxd_cnt <= 0;
            smp_cnt <= 0;
            if(rxd_sync == 1'b0)            //uart rxd start bit
                rxd_state <= R_START;
```

```
            else
                rxd_state <= R_IDLE;
            end
        R_START:
            begin
            if(clken_16bps == 1)            //clk_bps * 16
                begin
                smp_cnt <= smp_cnt + 1'b1;
                if(smp_cnt == SMP_CENTER && rxd_sync != 1'b0)   //invalid data
                    begin
                    rxd_cnt <= 0;
                    rxd_state <= R_IDLE;
                    end
                else if(smp_cnt == SMP_TOP) //Count for 16 clocks
                    begin
                    rxd_cnt <= 1;
                    rxd_state <= R_SAMPLE;  //start mark bit is over
                    end
                else
                    begin
                    rxd_cnt <= 0;
                    rxd_state <= R_START;   //wait start mark bit over
                    end
                end
            else                            //invalid data
                begin
                smp_cnt <= smp_cnt;
                rxd_state <= rxd_state;
                end
            end
        R_SAMPLE:   //Sample 8 bit of Uart: {LSB, MSB}
            begin
            if(clken_16bps == 1)            //clk_bps * 16
                begin
                smp_cnt <= smp_cnt + 1'b1;
                if(smp_cnt == SMP_TOP)
                    begin
                    if(rxd_cnt < 4'd8)      //Totally 8 data
                        begin
                        rxd_cnt <= rxd_cnt + 1'b1;
                        rxd_state <= R_SAMPLE;
                        end
                    else
                        begin
                        rxd_cnt <= 4'd9;    //Turn of stop bit
                        rxd_state <= R_STOP;
                        end
                    end
```

```
                else
                    begin
                    rxd_cnt <= rxd_cnt;
                    rxd_state <= rxd_state;
                    end
                end
            else
                begin
                smp_cnt <= smp_cnt;
                rxd_cnt <= rxd_cnt;
                rxd_state <= rxd_state;
                end
            end
        R_STOP:
            begin
            if(clken_16bps == 1)        //clk_bps * 16
                begin
                smp_cnt <= smp_cnt + 1'b1;
                if(smp_cnt == SMP_TOP)
                    begin
                    rxd_state <= R_IDLE;
                    rxd_cnt <= 0;       //Stop data bit is done
                    end
                else
                    begin
                    rxd_cnt <= 9;       //Stop data bit
                    rxd_state <= R_STOP;
                    end
                end
            else
                begin
                smp_cnt <= smp_cnt;
                rxd_cnt <= rxd_cnt;
                rxd_state <= rxd_state;
                end
            end

        endcase
end
//uart data receive in center point
reg [7:0]   rxd_data_r;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        rxd_data_r <= 0;
    else if(rxd_state == R_SAMPLE)
        begin
        if(clken_16bps == 1 && smp_cnt == SMP_CENTER)   //sample center point
```

```
            case(rxd_cnt)
            4'd1:   rxd_data_r[0] <= rxd_sync;
            4'd2:   rxd_data_r[1] <= rxd_sync;
            4'd3:   rxd_data_r[2] <= rxd_sync;
            4'd4:   rxd_data_r[3] <= rxd_sync;
            4'd5:   rxd_data_r[4] <= rxd_sync;
            4'd6:   rxd_data_r[5] <= rxd_sync;
            4'd7:   rxd_data_r[6] <= rxd_sync;
            4'd8:   rxd_data_r[7] <= rxd_sync;
            default:;
            endcase
        else
            rxd_data_r <= rxd_data_r;
        end
    else if(rxd_state == R_STOP)
        rxd_data_r <= rxd_data_r;
    else
        rxd_data_r <= 0;
end
//update uart receive data and receive flag signal
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        rxd_data <= 0;
        rxd_flag <= 0;
        end
    else if(clken_16bps == 1 &&  rxd_cnt == 4'd9 && smp_cnt == SMP_TOP) //Start + 8
Bit + Stop Bit
        begin
        rxd_data <= rxd_data_r;
        rxd_flag <= 1;
        end
    else
        begin
        rxd_data <= rxd_data;
        rxd_flag <= 0;
        end
end
endmodule
```

```
`timescale 1ns/1ns
module  integer_divider #(
    parameter       DEVIDE_CNT = 16'd651    //9600bps * 16
)(
    //global clock
    input           clk,
    input           rst_n,
    //user interface
```

```
    output          divide_clken
);
//RTL1: Precise fractional frequency for uart bps clock
reg [31:0]  cnt;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        cnt <= 0;
    else
        cnt <= (cnt < DEVIDE_CNT - 1'b1) ? cnt + 1'b1 : 16'd0;
end
assign divide_clken = (cnt == DEVIDE_CNT - 1'b1) ? 1'b1 : 1'b0;
endmodule
```

```
`timescale 1ns / 1ns
module uart (
    //CLK
    input        CLK_100M,
    input        CLK_LOCKED,
    //Interface
    input      UART_RX,
    output       UART_TX,
    //MODULE_OUTPUT
    output [7:0] RX_DATA,     //OUTPUT
    output       RX_OK,        //OUTPUT
    input [7:0] TX_DATA,
    input TX_EN,
    output TX_OK
);
  wire clk_ref = CLK_100M;  //96MHz
  wire sys_rst_n = CLK_LOCKED;
  //分频产生
  wire                            divide_clken;
  integer_divider #(
      .DEVIDE_CNT(651)  //9600bps * 16
  ) u_integer_devider (
      //global
      .clk  (clk_ref),
      .rst_n(sys_rst_n), //global reset

      .divide_clken(divide_clken)
  );
  wire clken_16bps = divide_clken;
  //RX
  uart_receiver u_uart_receiver (
      //GOBEL_CLK
      .clk  (clk_ref),
      .rst_n(sys_rst_n),
      //uart interface
```

```
      .clken_16bps(clken_16bps),  //clk_bps * 16
      .rxd        (UART_RX),      //uart txd interface
      //user interface
      .rxd_data(RX_DATA),  //uart data receive [7:0]
      .rxd_flag(RX_OK)     //uart data receive done H
  );
  // Data receive for PC to FPGA.
  uart_transfer u_uart_transfer (
      //gobal clock
      .clk  (clk_ref),
      .rst_n(sys_rst_n),
      //uaer interface
      .clken_16bps(clken_16bps),  //clk_bps * 16
      .txd        (UART_TX),      //uart txd interface
      //user interface
      .txd_en  (TX_EN),    //uart data transfer enable
      .txd_data(TX_DATA),  //uart transfer data
      .txd_flag(TX_OK)          //uart data transfer done
  );
endmodule
```

9. XADC 设计

a) 设计思路

Artix7 芯片内置了一个多通道 12bit 的 ADC，其返回数据位宽为 16，其中高 12 位是有效数据。

b) 源代码

```
module xadc (
    input clk_100m,
   input xadc_en,
    input XADC_AUX_v_n,
    input XADC_AUX_v_p,
    input XADC_VP_VN_v_n,
    input XADC_VP_VN_v_p,

    output [11:0] adcx_data_o
);
  wire          eoc_out;
  wire [ 4 : 0] channel_out;
  wire [15 : 0] do_out;
  assign adcx_data_o = do_out[15:4];
  xadc_wiz u_xadc (
      .di_in   (16'd0),             // input wire [15 : 0] di_in
      .daddr_in(channel_out),  // input wire [6 : 0] daddr_in
      .den_in  (xadc_en & eoc_out),      // input wire den_in
      .dwe_in  (1'b0),         // input wire dwe_in
      .drdy_out(),  // output wire drdy_out
      .do_out (do_out),   // output wire [15 : 0] do_out
      .dclk_in(clk_100m), // input wire dclk_in
      .reset_in(1'b0),
```

```
      .vp_in(XADC_VP_VN_v_p),  // input wire vp_in
      .vn_in(XADC_VP_VN_v_n),  // input wire vn_in
      .vauxp1(XADC_AUX_v_p),  // input wire vauxp1
      .vauxn1(XADC_AUX_v_n),  // input wire vauxn1
      .channel_out(channel_out),  // output wire [4 : 0] channel_out
      .eoc_out  (eoc_out),  // output wire eoc_out
      .alarm_out(),         // output wire alarm_out
      .eos_out  (),         // output wire eos_out
      .busy_out ()          // output wire busy_out
  );
endmodule
```

10. 音频播放设计

a) 设计思路

上文已经介绍了音频播放的原理，系统中使用状态机来逐个遍历乐谱来实现歌曲欢乐颂的播放，响度控制则是由 XADC 采样数值控制，数值越大则声音越大，这里也是使用了整个项目中唯一的除法器。

b) 源代码

```
`timescale 1ns / 1ps
module audio_music (
    input clk,  // 输入时钟信号
    input  [11:0] loud,

    output        beep   // 输出蜂鸣器信号
);
  reg         r_beep = 1'd0; // 蜂鸣器状态寄存器
  reg  [ 7:0] S_MUSIC = 8'd0; // 乐谱状态机

  reg  [16:0] r_cnt = 17'd0;
  reg  [16:0] r_cnt_lim = 17'd0;  // 计数器
  reg  [25:0] r_tim_cnt = 26'd0;  // 用于计时的计数器
  //降低精度优化资源
  wire [ 5:0] w_loud;
  reg  [ 5:0] r_loud_dly = 6'd0;
  assign w_loud = loud[11:6];
  // 定义音符频率参数
  parameter L_1 = 18'd127552;
  parameter L_2 = 18'd113636;
  parameter L_3 = 18'd101236;
  parameter L_4 = 18'd95548;
  parameter L_5 = 18'd85136;
  parameter L_6 = 18'd75838;
  parameter L_7 = 18'd67567;
  parameter M_1 = 18'd63776;
  parameter M_2 = 18'd56818;
  parameter M_3 = 18'd50607;
  parameter M_4 = 18'd47778;
  parameter M_5 = 18'd42553;
  parameter M_6 = 18'd37936;
```

```
parameter M_7 = 18'd33783;
parameter TIME = 50000000;  // 每种音阶持续时长为 500ms
assign beep = r_beep;  // 将蜂鸣器状态赋给输出端口

reg [16:0] r_cnt_lim_div = 17'd0;
reg [ 1:0] r_ctrl_dly = 2'd0;

always @(posedge clk) begin
  r_loud_dly <= w_loud;
  if (r_loud_dly != w_loud) begin
    if (w_loud == 6'd0) begin
      r_cnt_lim_div <= 17'd0;
    end else begin
      r_cnt_lim_div <= (r_cnt_lim / (7'd64 - w_loud));
    end
  end
end
// 在时钟上升沿触发的逻辑
always @(posedge clk) begin
  r_cnt <= r_cnt + 1'b1;  // 计数器递增
  if (r_cnt < r_cnt_lim_div) begin
    r_beep <= 1'b1;
  end else if (r_cnt == {r_cnt_lim[15:0], 1'd0}) begin
    r_cnt <= 17'h0;
    // 翻转蜂鸣器状态，形成音符波形
  end else begin
    r_beep <= 1'b0;
  end
end
// 乐谱状态机逻辑
always @(posedge clk) begin
  if (r_tim_cnt < TIME) begin
    r_tim_cnt = r_tim_cnt + 1'b1;
  end else begin
    r_tim_cnt = 26'd0;
    if (S_MUSIC >= 8'd63) begin
      S_MUSIC = 8'd0;
    end else begin
      S_MUSIC = S_MUSIC + 1'b1;
    end
    case (S_MUSIC)
      // 小欢乐颂曲谱
      8'D0:  r_cnt_lim = M_3;
      8'D1:  r_cnt_lim = M_3;
      8'D2:  r_cnt_lim = M_4;
      8'D3:  r_cnt_lim = M_5;
      8'D4:  r_cnt_lim = M_5;
      8'D5:  r_cnt_lim = M_4;
```

```
        8'D6:  r_cnt_lim = M_3;
        8'D7:  r_cnt_lim = M_2;
        8'D8:  r_cnt_lim = M_1;
        8'D9:  r_cnt_lim = M_1;
        8'D10: r_cnt_lim = M_2;
        8'D11: r_cnt_lim = M_3;
        8'D12: r_cnt_lim = M_3;
        8'D13: r_cnt_lim = M_2;
        8'D14: r_cnt_lim = M_2;
        8'D15: r_cnt_lim = M_3;
        8'D16: r_cnt_lim = M_3;
        8'D17: r_cnt_lim = M_4;
        8'D18: r_cnt_lim = M_5;
        8'D19: r_cnt_lim = M_5;
        8'D20: r_cnt_lim = M_4;
        8'D21: r_cnt_lim = M_3;
        8'D22: r_cnt_lim = M_2;
        8'D23: r_cnt_lim = M_1;
        8'D24: r_cnt_lim = M_1;
        8'D25: r_cnt_lim = M_2;
        8'D26: r_cnt_lim = M_3;
        8'D27: r_cnt_lim = M_2;
        8'D28: r_cnt_lim = M_1;
        8'D29: r_cnt_lim = M_1;
        8'D30: r_cnt_lim = M_2;
        8'D31: r_cnt_lim = M_2;
        8'D32: r_cnt_lim = M_3;
        8'D33: r_cnt_lim = M_1;
        8'D34: r_cnt_lim = M_2;
        8'D35: r_cnt_lim = M_3;
        8'D36: r_cnt_lim = M_4;
        8'D37: r_cnt_lim = M_3;
        8'D38: r_cnt_lim = M_1;
        8'D39: r_cnt_lim = M_2;
        8'D40: r_cnt_lim = M_3;
        8'D41: r_cnt_lim = M_4;
        8'D42: r_cnt_lim = M_3;
        8'D43: r_cnt_lim = M_2;
        8'D44: r_cnt_lim = M_1;
        8'D45: r_cnt_lim = M_2;
        8'D46: r_cnt_lim = M_5;
        8'D47: r_cnt_lim = M_3;
        8'D48: r_cnt_lim = M_3;
        8'D49: r_cnt_lim = M_3;
        8'D50: r_cnt_lim = M_4;
        8'D51: r_cnt_lim = M_5;
        8'D52: r_cnt_lim = M_5;
        8'D53: r_cnt_lim = M_4;
```

```
        8'D54: r_cnt_lim = M_3;
        8'D55: r_cnt_lim = M_2;
        8'D56: r_cnt_lim = M_1;
        8'D57: r_cnt_lim = M_1;
        8'D58: r_cnt_lim = M_2;
        8'D59: r_cnt_lim = M_3;
        8'D60: r_cnt_lim = M_2;
        8'D61: r_cnt_lim = M_1;
        8'D62: r_cnt_lim = M_1;
        default: begin
          r_cnt_lim = 16'h0;
          S_MUSIC   = 8'd0;
        end
      endcase
    end
  end
endmodule
```

11. VGA 显示设计

a) 设计思路

上文中已经介绍了 VGA 时序，因此我们只需要通过计数器维护行同步和场同步信号就能实现一帧图像的显示，该系统能够实现横竖彩条、灰度、坐标显示四种演示图像，每隔 2 秒切换一次。

b) 源代码

```
`timescale 1ns/1ps
// lcd_para.v
//Define the color parameter RGB--8|8|8
`define RED    24'hFF0000   /*11111111,00000000,00000000 */
`define GREEN  24'h00FF00   /*00000000,11111111,00000000 */
`define BLUE   24'h0000FF   /*00000000,00000000,11111111 */
`define WHITE  24'hFFFFFF   /*11111111,11111111,11111111 */
`define BLACK  24'h000000   /*00000000,00000000,00000000 */
`define YELLOW 24'hFFFF00   /*11111111,11111111,00000000 */
`define CYAN   24'hFF00FF   /*11111111,00000000,11111111 */
`define ROYAL  24'h00FFFF   /*00000000,11111111,11111111 */
//--------------------------------
`define  SYNC_POLARITY 1'b0
`define  H_FRONT  12'd16
`define  H_SYNC   12'd96
`define  H_BACK   12'd48
`define  H_DISP   12'd640
`define  H_TOTAL  12'd800
`define  V_FRONT  12'd10
`define  V_SYNC   12'd2
`define  V_BACK   12'd33
`define  V_DISP   12'd480
`define  V_TOTAL  12'd525
```

```
`timescale 1ns/1ps
`include "lcd_para.v"
```

```
module lcd_driver #(
    //  Setting up default timing
    parameter   H_FRONT = `H_FRONT,
            H_SYNC  = `H_SYNC,
            H_BACK  = `H_BACK,
            H_DISP  = `H_DISP,
            H_TOTAL = `H_TOTAL,

            V_FRONT = `V_FRONT,
            V_SYNC  = `V_SYNC,
            V_BACK  = `V_BACK,
            V_DISP  = `V_DISP,
            V_TOTAL = `V_TOTAL,
            DATA_WIDTH  = 24
)(
    //global clock
    input                   clk,            //system clock
    input                   rst_n,          //sync reset
    //lcd interface
    output              lcd_dclk,       //lcd pixel clock
    output              lcd_blank,      //lcd blank
    output              lcd_sync,       //lcd sync
    output reg          lcd_hs,         //lcd horizontal sync
    output reg          lcd_vs,         //lcd vertical sync
    output reg          lcd_en,         //lcd display enable
    output  [DATA_WIDTH-1:0]    lcd_rgb,        //lcd display data
    //user interface
    output reg          lcd_request,    //lcd data request
    output reg  [11:0]  lcd_xpos,       //lcd horizontal coordinate
    output reg  [11:0]  lcd_ypos,       //lcd vertical coordinate
    input   [DATA_WIDTH-1:0]    lcd_data        //lcd data
);
/*****************************************
        SYNC--BACK--DISP--FRONT
*****************************************/
//h_sync counter & generator
reg [13:0] hcnt;
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        hcnt <= 12'd0;
    else
        begin
        if(hcnt < H_TOTAL - 1'b1)       //line over
            hcnt <= hcnt + 1'b1;
        else
            hcnt <= 12'd0;
```

```
        lcd_hs <= (hcnt <= H_SYNC - 1'b1) ? 1'b0 : 1'b1;
        end
end
//v_sync counter & generator
reg [11:0] vcnt;
always@(posedge clk or negedge rst_n)
begin
    if (!rst_n)
        vcnt <= 12'b0;
    else if(hcnt == H_TOTAL - 1'b1)     //line over
        begin
        if(vcnt < V_TOTAL - 1'b1)       //frame over
            vcnt <= vcnt + 1'b1;
        else
            vcnt <= 12'd0;

        lcd_vs <= (vcnt <= V_SYNC - 1'b1) ? 1'b0 : 1'b1;
        end
end
//LCELL LCELL(.in(clk),.out(lcd_dclk));
assign  lcd_dclk = ~clk;
assign  lcd_blank = lcd_hs & lcd_vs;
assign  lcd_sync = 1'b0;
//---------------------------------------
reg     [DATA_WIDTH-1:0]  r_lcd_rgb = 0;
always@(posedge clk) begin
    lcd_en      <=  (hcnt >= H_SYNC + H_BACK  && hcnt < H_SYNC + H_BACK + H_DISP)
&&
                        (vcnt >= V_SYNC + V_BACK  && vcnt < V_SYNC + V_BACK +
V_DISP)
                        ? 1'b1 : 1'b0;
    r_lcd_rgb   <=  lcd_data;
end
assign lcd_rgb = lcd_en ? r_lcd_rgb : 0;
//ahead x clock
localparam  H_AHEAD =   12'd1;
//  Standard FIFO Request Port
 always@(posedge clk) begin
    lcd_request <=  (hcnt >= H_SYNC + H_BACK - H_AHEAD && hcnt < H_SYNC + H_BACK +
H_DISP - H_AHEAD) &&
                         (vcnt >= V_SYNC + V_BACK && vcnt < V_SYNC + V_BACK +
V_DISP)
                        ? 1'b1 : 1'b0;

// lcd xpos & ypos
    lcd_xpos    <=  lcd_request ? (hcnt - (H_SYNC + H_BACK - H_AHEAD)) : 11'd0;
    lcd_ypos    <=  lcd_request ? (vcnt - (V_SYNC + V_BACK)) : 12'd0;
end
```

```
endmodule
```

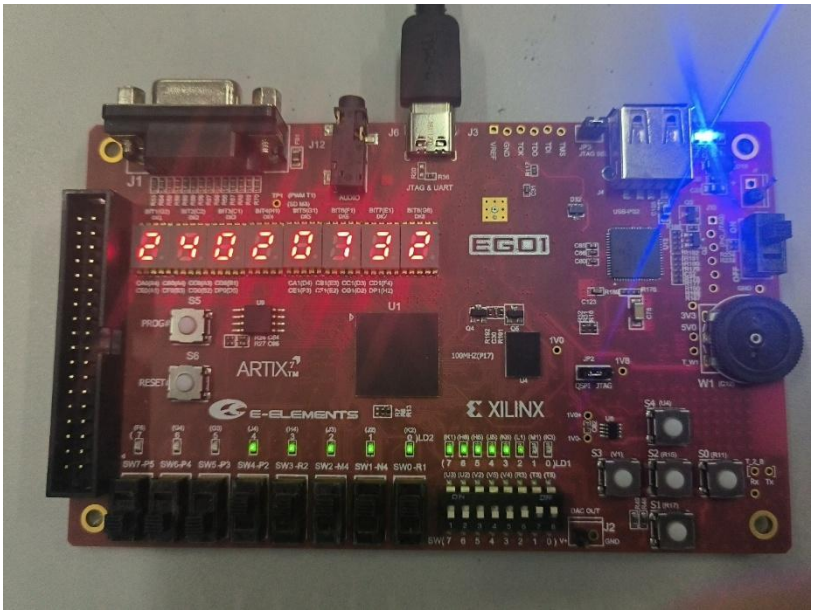
```
`timescale 1ns/1ns
module lcd_display(
    input               clk,        //system clock
    input               rst_n,      //sync clock

    input       [11:0]  lcd_xpos,   //lcd horizontal coordinate
    input       [11:0]  lcd_ypos,   //lcd vertical coordinate
    output  reg [23:0]  lcd_data    //lcd data
);
`include "lcd_para.v"
`define VGA_HORIZONTAL_COLOR
`define VGA_VERICAL_COLOR
`define VGA_GRAY_GRAPH
`define VGA_GRAFTAL_GRAPH
initial begin
    lcd_data <= 0;
end
localparam  DELAY_TOP = 40_000000;
//-----------------------------------------
`ifdef VGA_HORIZONTAL_COLOR
reg [23:0]  lcd_data0;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        lcd_data0 <= 0;
    else
        begin
        if  (lcd_ypos >= 0 && lcd_ypos < (`V_DISP/8)*1)
            lcd_data0 <= `RED;
        else if(lcd_ypos >= (`V_DISP/8)*1 && lcd_ypos < (`V_DISP/8)*2)
            lcd_data0 <= `GREEN;
        else if(lcd_ypos >= (`V_DISP/8)*2 && lcd_ypos < (`V_DISP/8)*3)
            lcd_data0 <= `BLUE;
        else if(lcd_ypos >= (`V_DISP/8)*3 && lcd_ypos < (`V_DISP/8)*4)
            lcd_data0 <= `WHITE;
        else if(lcd_ypos >= (`V_DISP/8)*4 && lcd_ypos < (`V_DISP/8)*5)
            lcd_data0 <= `BLACK;
        else if(lcd_ypos >= (`V_DISP/8)*5 && lcd_ypos < (`V_DISP/8)*6)
            lcd_data0 <= `YELLOW;
        else if(lcd_ypos >= (`V_DISP/8)*6 && lcd_ypos < (`V_DISP/8)*7)
            lcd_data0 <= `CYAN;
        else// if(lcd_ypos >= (`V_DISP/8)*7 && lcd_ypos < (`V_DISP/8)*8)
            lcd_data0 <= `ROYAL;
        end
end
`endif
//-----------------------------------------
```

```
`ifdef VGA_VERICAL_COLOR
reg [23:0]  lcd_data1;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        lcd_data1 <= 0;
    else
        begin
        if  (lcd_xpos >= 0 && lcd_xpos < (`H_DISP/8)*1)
            lcd_data1 <= `RED;
        else if(lcd_xpos >= (`H_DISP/8)*1 && lcd_xpos < (`H_DISP/8)*2)
            lcd_data1 <= `GREEN;
        else if(lcd_xpos >= (`H_DISP/8)*2 && lcd_xpos < (`H_DISP/8)*3)
            lcd_data1 <= `BLUE;
        else if(lcd_xpos >= (`H_DISP/8)*3 && lcd_xpos < (`H_DISP/8)*4)
            lcd_data1 <= `WHITE;
        else if(lcd_xpos >= (`H_DISP/8)*4 && lcd_xpos < (`H_DISP/8)*5)
            lcd_data1 <= `BLACK;
        else if(lcd_xpos >= (`H_DISP/8)*5 && lcd_xpos < (`H_DISP/8)*6)
            lcd_data1 <= `YELLOW;
        else if(lcd_xpos >= (`H_DISP/8)*6 && lcd_xpos < (`H_DISP/8)*7)
            lcd_data1 <= `CYAN;
        else// if(lcd_xpos >= (`H_DISP/8)*7 && lcd_xpos < (`H_DISP/8)*8)
            lcd_data1 <= `ROYAL;
        end
end
`endif
//------------------------------------------
`ifdef VGA_GRAFTAL_GRAPH
reg [23:0]  lcd_data2;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        lcd_data2 <= 0;
    else
        lcd_data2 <= lcd_xpos * lcd_ypos;
end
`endif
//------------------------------------------
`ifdef VGA_GRAY_GRAPH
reg [23:0]  lcd_data3;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        lcd_data3 <= 0;
    else
        begin
        if(lcd_ypos < `V_DISP/2)
```

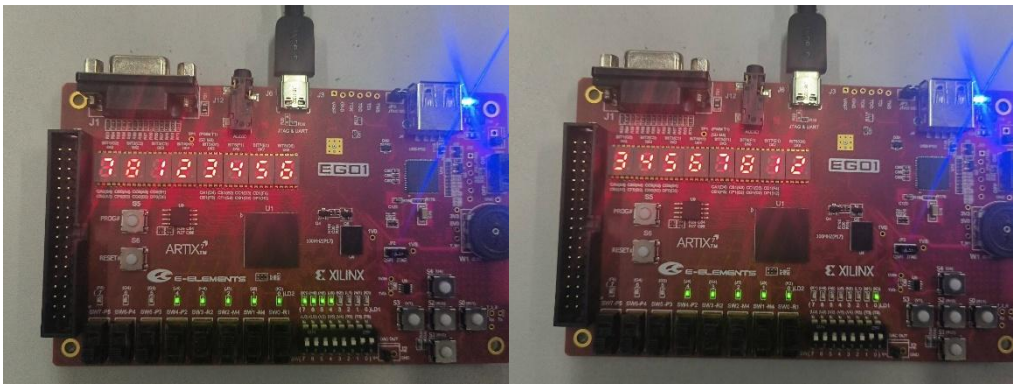
```
            lcd_data3 <= {lcd_ypos[7:0], lcd_ypos[7:0], lcd_ypos[7:0]};
        else
            lcd_data3 <= {lcd_xpos[7:0], lcd_xpos[7:0], lcd_xpos[7:0]};
        end
end
`endif

//------------------------------------
//0.5S Delay
reg [27:0]  delay_cnt;
//localparam    DELAT_TOP 50_000000/2
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        delay_cnt <= 0;
    else
        delay_cnt <= (delay_cnt < DELAY_TOP - 1'b1) ? delay_cnt + 1'b1 : 28'd0;
end
wire    delay_0S5_flag = (delay_cnt == DELAY_TOP - 1'b1) ? 1'b1 : 1'b0;
//------------------------------------
//4 picture cycle cnt
reg [1:0]   image_cnt;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        image_cnt <= 0;
    else if(delay_0S5_flag)
        image_cnt <= image_cnt + 1'b1;
    else
        image_cnt <= image_cnt;
end
//------------------------------------
//4 picture cycle
always@(*)
begin
    case(image_cnt)
    0:  lcd_data <= lcd_data0;
    1:  lcd_data <= lcd_data1;
    2:  lcd_data <= lcd_data2;
    3:  lcd_data <= lcd_data3;
    endcase
end
endmodule
```

四．调试结果

1. 学号显示

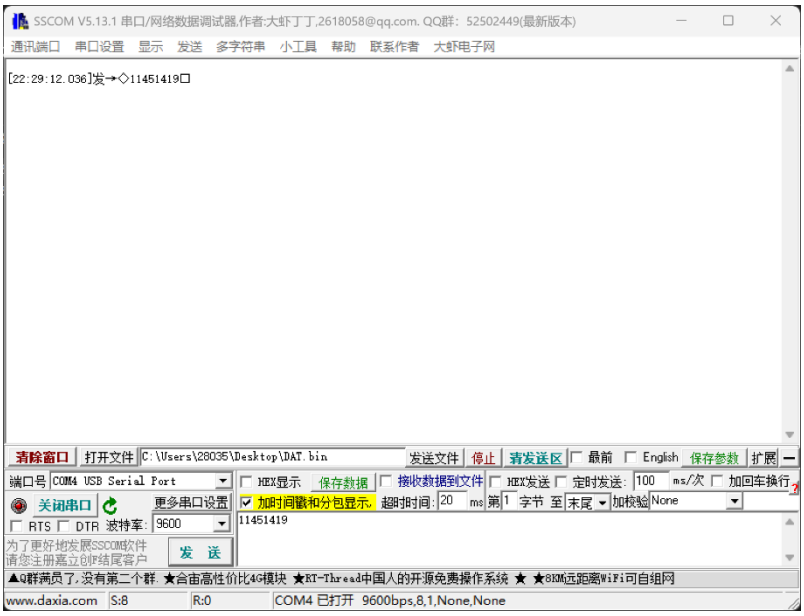


2. 数值循环位移



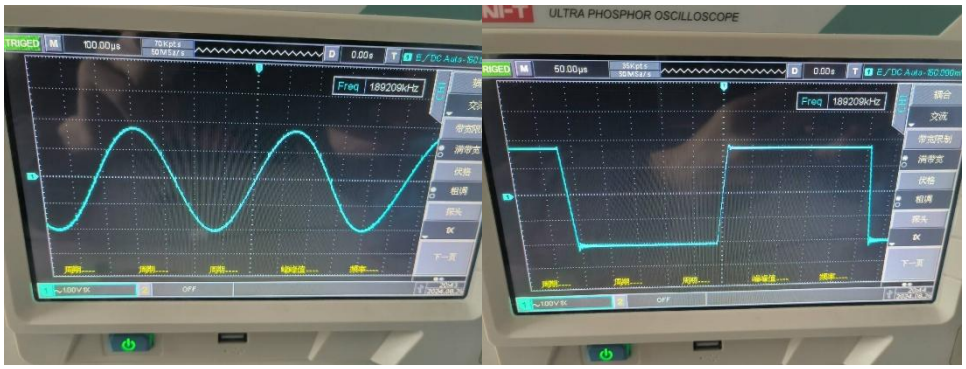
3. 串口接受数据

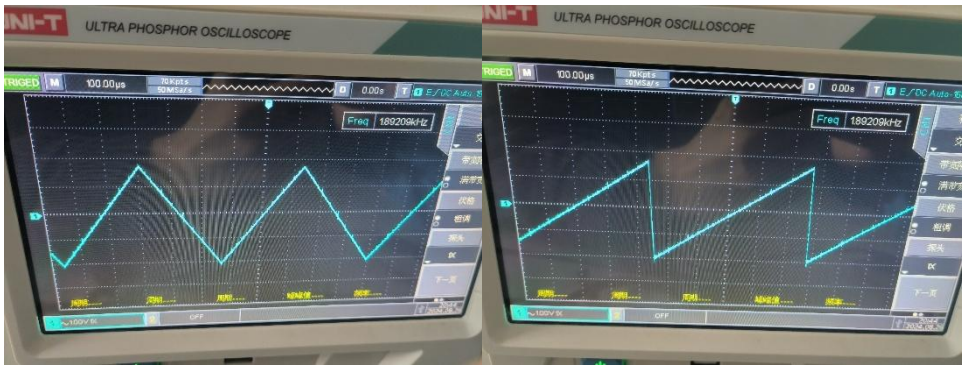
SSCOM 发送数据:



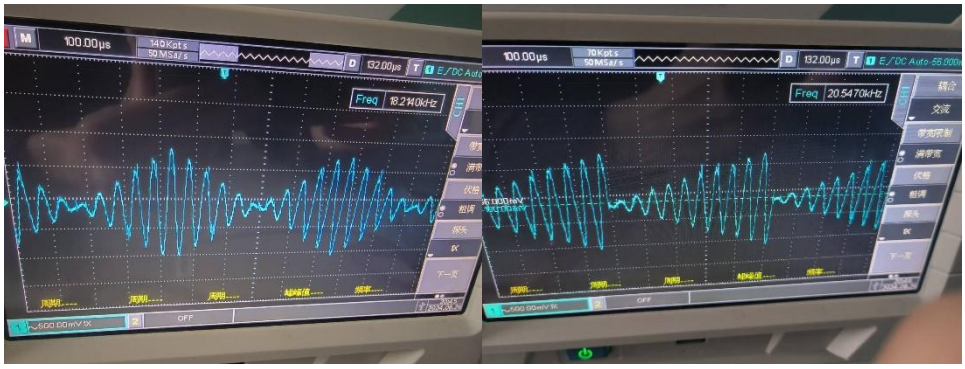
显示效果:

4. 四种波形的 dds（以 1893Hz 为图例）

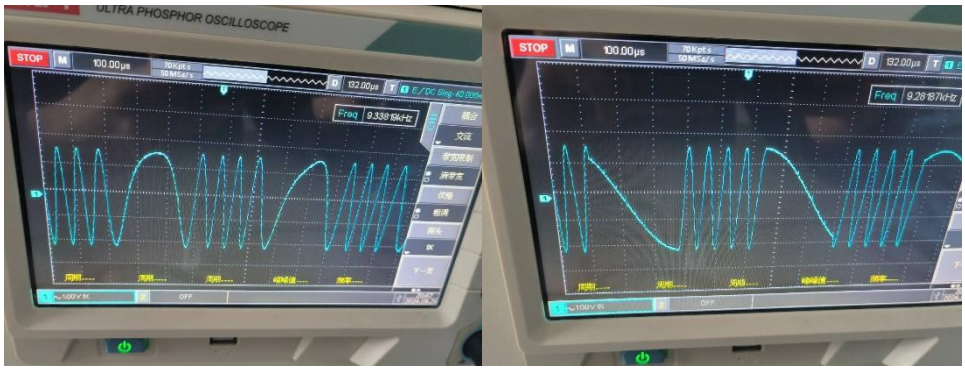


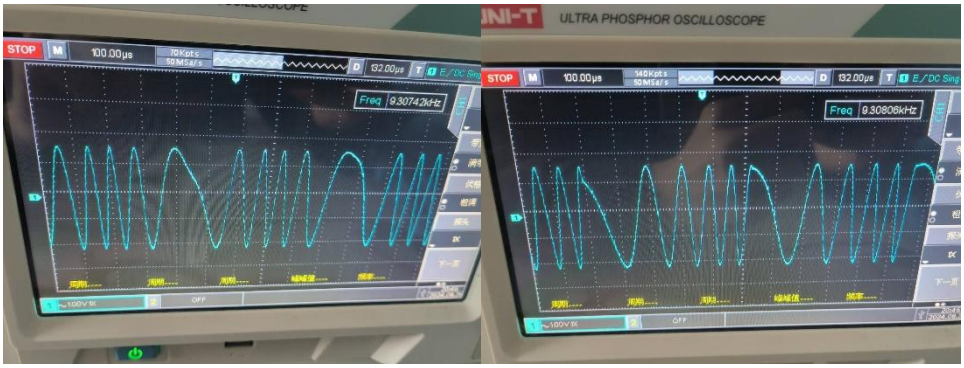


5. 四种波形的 am（以 1893Hz 为图例，顺序与基本波形顺序一致）

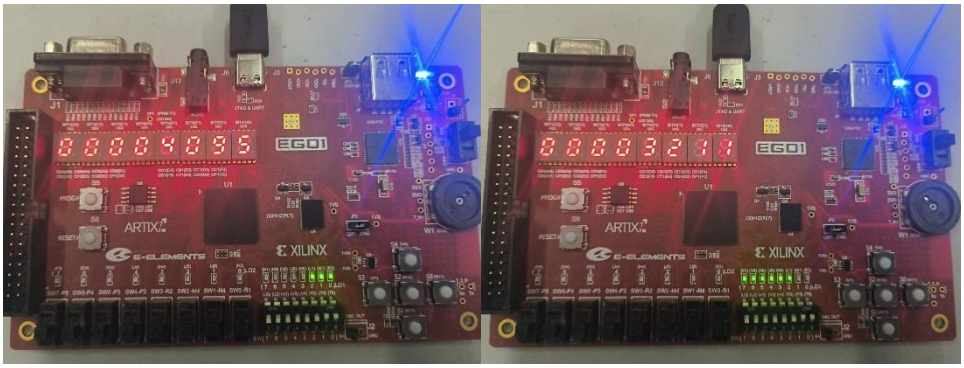


6. 四种波形的 fm（以 1893Hz 为图例，顺序与基本波形顺序一致）

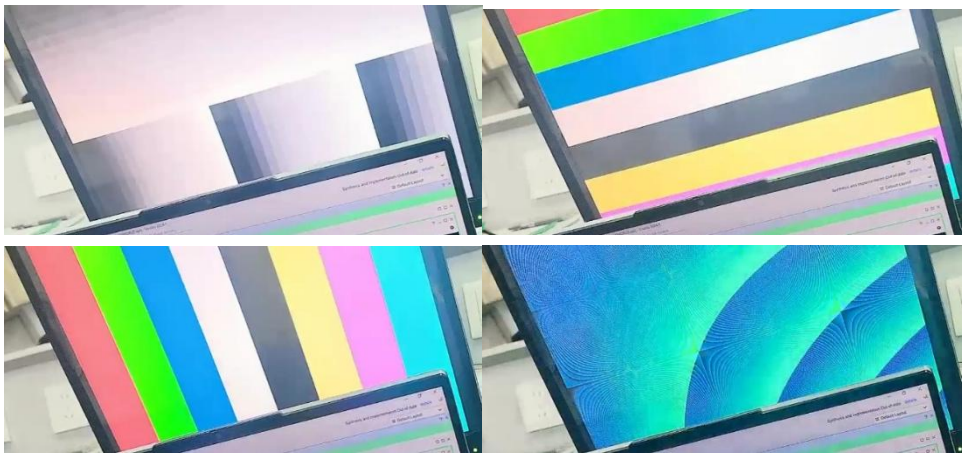




7. XADC 采样



8. VGA 显示



五．总结与感悟

1） 设计过程中遇到的问题及解决问题的方法

设计过程中实际烧录后无效果，最终发现时钟线网打错字符，更正后恢复正常；DDS 波形高控制字下失真，最终采用高速时钟配合截位改善效果；FM、AM 仿真波形正常，实际上板输出波形糟糕，发现是 dac 性能较弱导致，降低载波频率后波形不会出现失真。

2） 设计的收获与感受

由于暑期电赛备赛和参与过 FPGA 相关比赛，所以相比其他同学进度更快一些。由于实验课不涉及高速时钟，甚至几乎不用考虑亚稳态情况，很多基础模块是我备赛和比赛期间准备的，只要绑定对管脚正常上板一次性调通，因此也就完全没有必要再去专门补对应的仿真文件。

这次实验对我帮助最多的地方应该是加深了我对截位的应用，以及给了我两周时间尽可能实现 FPGA 代码设计中的优化：在遇到 BUG 时能够通过 RTL 原理图分析出代码中哪里写错了，在写法正确时优化代码，尽可能减少逻辑资源的使用。

3） 期望及要求

由于本次实验比较简单，所有本人期望板卡内所有外设都尽可能使用一遍，但最终还有蓝牙、ps2 usb、sram 没有能应用。实验课有待改进的一点是忽略了 FPGA 在控制领域的应用，不应该全部关注于信号方面的有限内容，而这些基础信号调制功能在实际工程中可能早已写好能够随手调用的模块。

FPGA 的精髓在于高速时钟对时序逻辑的约束，跨时钟域的异步数据流处理，而这些在实验中并没有体现，应用局限于低速表层。

六．附录

顶层驱动文件

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: NJUST
// Engineer: Infinite_Cloisters
// Author: 922104780732 JingQiu Xu
// Create Date: 2024/10/19 18:40:27
// Module Name: main_top
// Project Name: EGO1_TEST
// Tool Versions: 2024.1
// Revision:0.38
//////////////////////////////////////////////////////////////////////////////////
```

```
module main_top (
    ////////////////////////////////////////////////////////////////
    //board clk and rstn
    input        i_clk_100m,     //p17
    input        i_rstn_sys,     //p15
    ////////////////////////////////////////////////////////////////
    //uart interface
    input        i_uart_rxd,     //n5
    output       o_uart_txd,     //t4
    ////////////////////////////////////////////////////////////////
    //bluetooth interface
    input        i_bt_uart_rxd,  //l3
    output       o_bt_uart_txd,  //n2
    ////////////////////////////////////////////////////////////////
    output       o_bt_p1_3,      //e18
    output       o_bt_p0_6,      //c16
    output       o_bt_p0_7,      //h15
    output       o_bt_rstn,      //m2
    output       o_bt_3v3,       //d18
    output       o_bt_scl,       //m3
    output       o_bt_sda,       //n1
    output       i_bt_int,       //bluetooth interrupt c17
    ////////////////////////////////////////////////////////////////
    //five buttons  default:L
    input        i_btn_up,       //u4
    input        i_btn_down,     //r17
    input        i_btn_left,     //v1
    input        i_btn_right,    //r11
    input        i_btn_mid,      //r15
    ////////////////////////////////////////////////////////////////
    //sw-smt    default:L    [7]u3 [6]u2 [5]v2 [4]v5 [3]v4 [2]r3 [1]t3 [0]t5
    input  [7:0] i_sw_smt,
    //sw-dip    default:L    [7]p5 [6]p4 [5]p3 [4]p2 [3]r2 [2]m4 [1]n4 [0]r1
    input [7:0] i_sw_dip,
    //LD1   [7]k1 [6]h6 [5]h5 [4]j5 [3]k6 [2]l1 [1]m1 [0]k3
    output [7:0] o_leds_1,
    //LD2   [7]f6 [6]g4 [5]g3 [4]j4 [3]h4 [2]j3 [1]j2 [0]k2
    output [7:0] o_leds_2,
    ////////////////////////////////////////////////////////////////
    //seg interface
    output [3:0] o_seg_1_cs,   //H vaild g2 c2 c1 h1
    output [7:0] o_seg_1_data,
    output [3:0] o_seg_2_cs,     //H vaild g1 f1 e1 g6
    output [7:0] o_seg_2_data,
    ////////////////////////////////////////////////////////////////
    //8bit dac
    output [7:0] o_dac8,
    output       o_dac8_ile,
```

```
    output       o_dac8_cs_n,
    output       o_dac8_wr1_n,
    output       o_dac8_wr2_n,
    output       o_dac8_xfer_n,
    ////////////////////////////////////////////////////////////////
    //audio
    output       o_audio_pwm,
    output       o_audio_sd,
    ////////////////////////////////////////////////////////////////
    //xadc
    input        vauxn1,
    input        vauxp1,         //C12
    input XADC_VP_VN_v_n,
    input XADC_VP_VN_v_p,
    ////////////////////////////////////////////////////////////////
    //vga
    output o_vga_hs,
    output o_vga_vs,
    output [3:0] o_vga_r,
    output [3:0] o_vga_g,
    output [3:0] o_vga_b,
    //exp io unused
    output [13:0] o_dac14,
    input  [ 9:0] i_adc
);
  ////////////////////////////////////////////////////////////////
  wire w_rstn_sys = i_rstn_sys;
  //Generate CLK
  wire clk_100m = i_clk_100m;
  wire                          clk_1m;
  wire                          clk_250k;
  wire                          clk_10k;
  wire                          clk_1k;
  clk_gen #(
      .cnt_div(50000)
  ) u_clk_1k (
      .clk(clk_100m),
      .clk_div   (clk_1k)
  );
  clk_gen #(
      .cnt_div(5000)
  ) u_clk_10k (
      .clk(clk_100m),
      .clk_div   (clk_10k)
  );
  clk_gen #(
      .cnt_div(200)
  ) u_clk_250k (
```

```
    .clk(clk_100m),
    .clk_div   (clk_250k)
);
clk_gen #(
    .cnt_div(50)
) u_clk_1m (
    .clk(clk_100m),
    .clk_div   (clk_1m)
);
////////////////////////////////////////////////////////////////
//时钟域的 rstn 生成
reg   r_rstn_1m = 1'b0;
reg   r_rstn_250k = 1'b0;
reg   r_rstn_10k = 1'b0;
reg   r_rstn_1k = 1'b0;

always @(posedge clk_1m) begin
  r_rstn_1m <= w_rstn_sys;
end
wire w_rstn_1m = r_rstn_1m;
always @(posedge clk_250k) begin
  r_rstn_250k <= w_rstn_sys;
end
wire w_rstn_250k = r_rstn_250k;
always @(posedge clk_10k) begin
  r_rstn_10k <= w_rstn_sys;
end
wire w_rstn_10k = r_rstn_10k;
always @(posedge clk_1k) begin
  r_rstn_1k <= w_rstn_sys;
end
wire w_rstn_1k = r_rstn_1k;
////////////////////////////////////////////////////////////////
//UART RX MODULE
wire                        [ 7:0] w_rx_data;
reg                         [ 7:0] w_tx_data;
reg                                r_tx_ready;
wire                               w_tx_valid;
wire                               w_rx_valid;
reg                         [ 7:0] r_rx_bcd;
reg                         [31:0] r_rx_bcd_all;
uart u_uart (
    //CLK
    .CLK_100M  (clk_100m),
    .CLK_LOCKED(w_rstn_sys),
    //Interface
    .UART_RX   (i_uart_rxd),
    .UART_TX   (o_uart_txd),
```

```
    //MODULE_OUTPUT
    .RX_DATA   (w_rx_data),   //OUTPUT
    .RX_OK     (w_rx_valid),  //OUTPUT
    .TX_DATA(r_tx_data),
    .TX_EN  (r_tx_ready),
    .TX_OK  (w_tx_valid)
);
always @(posedge w_rx_valid) begin
  //这里用阻塞赋值，防止显示延迟 1 位
  r_rx_bcd     = w_rx_data - 8'd48;
  r_rx_bcd_all = {r_rx_bcd_all[27:0], r_rx_bcd[3:0]};
end
////////////////////////////////////////////////////////////////
//button debiting
wire w_btn_up;
wire w_btn_down;
wire w_btn_left;
wire w_btn_right;
wire w_btn_mid;

btn_debiting u_btn_debiting (
    //key scan clk
    .clk        (clk_1k),
    .rstn       (w_rstn_1k),
    //button interface
    .i_btn_up   (i_btn_up),     // u4
    .i_btn_down (i_btn_down),   // r17
    .i_btn_left (i_btn_left),   // v1
    .i_btn_right(i_btn_right),  // r11
    .i_btn_mid  (i_btn_mid),    // r15
    //Output
    .btn_up_o   (w_btn_up),     // u4
    .btn_down_o (w_btn_down),   // r17
    .btn_left_o (w_btn_left),   // v1
    .btn_right_o(w_btn_right),  // r11
    .btn_mid_o  (w_btn_mid)     // r15
);
//sel mode by btn
wire [3:0] w_mode;
wire [3:0] w_type;
//wire       w_btn_mid_pos;
mode_sel #(
    .MODE_MAX(4'd3),  //0 1 2 3
    .TYPE_MAX(4'd3)   //0 1 2 3
) u_mode_sel (
    .clk (clk_250k),
    .rstn(w_rstn_250k),
    .btn_up_i   (w_btn_up),
```

```
    .btn_down_i (w_btn_down),
    .btn_left_i (w_btn_left),
    .btn_right_i(w_btn_right),
    .btn_mid_i  (w_btn_mid),
    .mode_o       (w_mode),
    .type_o       (w_type),
    //posedge
    .btn_mid_pos  (),
    .btn_up_pos   (),
    .btn_down_pos (),
    .btn_left_pos (),
    .btn_right_pos()
);
reg [3:0] r_mode_dly;
reg [3:0] r_type_dly;
//状态机选择模式
localparam M_DISP = 4'd0;
localparam M_FREQ = 4'd1;
localparam M_AM = 4'd2;
localparam M_FM = 4'd3;

reg                             r_bcd_en = 1'b0;
reg                             r_shl_en = 1'b0;
reg                             r_dds_en = 1'b0;
reg                             r_seg_en = 1'b1;
reg                             r_am_en = 1'b0;
reg                             r_fm_en = 1'b0;

reg  [  15:0]                   r_bin_1 = 16'd2024;
reg  [  15:0]                   r_bin_2 = 16'd0732;

reg  [  15:0]                   r_bin_1_dly = 16'd2024;
reg  [  15:0]                   r_bin_2_dly = 16'd0732;

reg  [  31:0]                   r_bcd = 32'h07320732;
reg  [  31:0]                   r_bcd_dly = 32'h07320732;
reg  [  15:0]                   r_bcd_shl_cnt = 16'd0;
wire [  15:0] w_bin_1 = r_bin_1_dly;
wire [  15:0] w_bin_2 = r_bin_2_dly;

wire [  15:0]                   w_freq;
wire [15 : 0]                   w_freq_the;

wire [  11:0]                   w_adcx_data;

always @(posedge clk_250k) begin
  r_mode_dly  <= w_mode;
  r_type_dly  <= w_type;
```

```
  r_bin_1_dly <= r_bin_1;
  r_bin_2_dly <= r_bin_2;
end
always @(posedge clk_250k) begin
  if ((r_mode_dly[3:0] == w_mode)) begin
    case (w_mode)
      M_DISP: begin
        r_dds_en <= 1'b0;
        r_am_en  <= 1'b0;
        r_fm_en  <= 1'b0;
        case (r_type_dly[3:0])
          4'd0: begin//显示学号
            r_seg_en <= 1'b1;
            r_bcd_en <= 1'b0;
            r_shl_en = 1'b0;
            r_bin_1 <= 16'd2402;
            r_bin_2 <= 16'd0732;
          end
          4'd1: begin//循环位移
            r_shl_en = 1'b1;
            r_seg_en <= 1'b1;
            r_bcd_en <= 1'b1;
          end
          4'd2: begin//取消 en
            r_shl_en = 1'b0;
            r_seg_en <= 1'b0;
            r_bcd_en <= 1'b0;
          end
          4'd3: begin//test
            r_seg_en <= 1'b1;
            r_bcd_en <= 1'b1;
            r_shl_en = 1'b0;
          end
        endcase
      end
      M_FREQ: begin
        r_dds_en <= 1'b1;
        r_seg_en <= 1'b1;
        r_bcd_en <= 1'b0;
        r_am_en  <= 1'b0;
        r_fm_en  <= 1'b0;
        r_bin_1  <= w_freq_the;
        r_bin_2  <= w_freq;
      end
      M_AM: begin
        r_dds_en <= 1'b1;
        r_bcd_en <= 1'b0;
        r_seg_en <= 1'b1;
```

```
          r_bin_1  <= w_freq_the;
          r_bin_2  <= i_sw_smt;
          r_am_en  <= 1'b1;
          r_fm_en  <= 1'b0;
        end
        M_FM: begin
          r_seg_en <= 1'b1;
          r_bcd_en <= 1'b0;
          r_dds_en <= 1'b1;
          r_bin_1  <= w_freq_the;
          r_bin_2  <= {4'd0, w_adcx_data};
          r_am_en  <= 1'b0;
          r_fm_en  <= 1'b1;
        end
      endcase
    end else begin
    end
end
//数码管 bcd 输入控制
always @(posedge clk_1k) begin
  if (r_bcd_en == 1) begin
    if (r_shl_en == 1) begin//数码管循环
      if (r_bcd_shl_cnt == 1024 - 1) begin
        {r_bcd[27:0], r_bcd[31:28]} <= r_bcd;
        r_bcd_dly                   <= r_bcd;
        r_bcd_shl_cnt               <= 16'd0;
      end else begin
        r_bcd_shl_cnt <= r_bcd_shl_cnt + 1'b1;
      end
    end else begin
      r_bcd     <= r_rx_bcd_all;
      r_bcd_dly <= r_bcd;
    end
  end else begin
    r_bcd_shl_cnt <= 16'd0;
    r_bcd         <= 32'h12345678;
  end
end
//////////////////////////////////////////////////////////////////
//Turn LED on when SW on
wire [7:0] w_led_pwm;
led_pwm u_led_pwm (
    .clk   (clk_250k),
    .clk_1k(clk_1k),
    .i_sw  (i_sw_dip),  //[7:0] i_sw
    .pwm_o (w_led_pwm),
    .o_leds(o_leds_2)    //[7:0] o_leds
);
```

```
wire [7:0] w_leds_1;
led_waterlamps u_led_waterlamps (
    .clk_1k(clk_1k),
    .type_i(r_type_dly[3:0]),
    .i_sw  (i_sw_smt),
    .o_leds(w_leds_1)
);
assign o_leds_1 = w_leds_1 & w_led_pwm;
////////////////////////////////////////////////////////////////
//SEG DISPLAY
seg u_seg (
    .clk     (clk_10k),
    .seg_en_i(r_seg_en & w_rstn_10k),
    .bin_1_i    (w_bin_1),
    .bin_2_i    (w_bin_2),
    .bin_valid_i(1),
    .bcd_en_i(r_bcd_en),  //优先
    .bcd_i   (r_bcd_dly),
    .o_seg_1_cs  (o_seg_1_cs),   //H vaild g2 c2 c1 h1
    .o_seg_1_data(o_seg_1_data),
    .o_seg_2_cs  (o_seg_2_cs),   //H vaild g1 f1 e1 g6
    .o_seg_2_data(o_seg_2_data)
);
////////////////////////////////////////////////////////////////
//8 BITS DAC CTRL
assign o_dac8_ile    = 1'd1;
assign o_dac8_cs_n   = 1'd0;
assign o_dac8_wr1_n  = 1'd0;
assign o_dac8_wr2_n  = 1'd0;
assign o_dac8_xfer_n = 1'd0;
wire [7:0] w_wave_fm;
wire [7:0] w_wave_nor;
dds u_dds (
    .clk   (clk_1m),   // clk
    .clk_am(clk_100m),
    .dds_en_i(r_dds_en & w_rstn_sys),
    .type_i  (r_type_dly[3:0]),        // input [3:0] type_i
    .Freq_i  (i_sw_dip),               // input [7:0] Freq_i
    .am_en(r_am_en),
    .fm_en(r_fm_en),
    .Phase_i(8'b0),  // input [7:0] Phase_i
    .Amp_i  (8'd1),  // input [7:0] Amp_i
    .wave_fm_o(w_wave_fm),
    .wave_o   (w_wave_nor)  // output [7:0] wave_o
);
assign o_dac8 = r_fm_en ? w_wave_fm : w_wave_nor;
////////////////////////////////////////////////////////////////
//measure
```

```
wire clk_0d5Hz;
clk_gen #(
    .cnt_div(1000)
) u_clk_0d5Hz (
    .clk(clk_1k),
    .clk_div   (clk_0d5Hz)
);
measure_freq u_freq (
    .clk_0d5Hz(clk_0d5Hz),
    .signal_i (o_dac8),
    .freq_o   (w_freq)
);
mem_theo u_theory (
    .clka (clk_1m),     // input wire clka
    .ena  (r_dds_en),   // input wire ena
    .addra(i_sw_dip),   // input wire [7 : 0] addra
    .douta(w_freq_the)  // output wire [15 : 0] douta
);
xadc u_xadc (
    .clk_100m(clk_100m),

    .xadc_en(i_sw_smt[1] & w_rstn_sys),

    .XADC_AUX_v_n(vauxn1),
    .XADC_AUX_v_p(vauxp1),  // C12

    .XADC_VP_VN_v_n(XADC_VP_VN_v_n),
    .XADC_VP_VN_v_p(XADC_VP_VN_v_p),

    .adcx_data_o(w_adcx_data)
);
////////////////////////////////////////////////////////////////
//audio
assign o_audio_sd = i_sw_smt[0] & w_rstn_sys;  // enable audio output
audio_music u_audio (
    .clk (clk_100m),
    .loud(w_adcx_data),
    .beep(o_audio_pwm)
);
wire       lcd_hs;
wire       lcd_vs;
wire       lcd_de;
wire [7:0] lcd_red;
wire [7:0] lcd_green;
wire [7:0] lcd_blue;
localparam CLOCK_PIXEL = 50_000000;
//------------------------------------
//LCD driver timing
```

```
  wire                          [11:0] lcd_xpos;  //lcd horizontal coordinate
  wire                          [11:0] lcd_ypos;  //lcd vertical coordinate
  wire                          [23:0] lcd_data;  //lcd data
  // clk_vga = 25m
  reg                                  clk_vga = 1'b0;
  reg                                  r_rstn_vga = 1'b1;
  wire w_rstn_vga = r_rstn_vga;
  reg                                  clk_cnt = 1'b0;
  always @(posedge clk_100m or negedge w_rstn_sys)//实现对系统时钟的分频得到 vga_clk 用于 VGA
图像显示时钟信号
        begin
    if (!w_rstn_sys) begin
      clk_vga    <= 1'b0;
      r_rstn_vga <= 1'b0;
    end else if (clk_cnt == 1) begin
      clk_vga = ~clk_vga;
      clk_cnt    <= 0;
      r_rstn_vga <= 1'b1;
    end else begin
      clk_cnt    <= clk_cnt + 1;
      r_rstn_vga <= 1'b1;
    end
  end
  lcd_driver u_lcd_driver (
      //global clock
      .clk  (clk_vga),
      .rst_n(w_rstn_vga),
      //lcd interface
      .lcd_dclk (),                              //(lcd_dclk),
      .lcd_blank(),                              //lcd_blank
      .lcd_sync (),
      .lcd_hs   (lcd_hs),
      .lcd_vs   (lcd_vs),
      .lcd_en   (lcd_de),
      .lcd_rgb  ({lcd_red, lcd_green, lcd_blue}),
      //user interface
      .lcd_request(),
      .lcd_data   (lcd_data),
      .lcd_xpos   (lcd_xpos),
      .lcd_ypos   (lcd_ypos)
  );
  assign o_vga_hs = lcd_hs;
  assign o_vga_vs = lcd_vs;

  assign o_vga_r  = lcd_red[7:4];
  assign o_vga_g  = lcd_green[7:4];
  assign o_vga_b  = lcd_blue[7:4];
  //-------------------------------------
```

```
  //lcd data simulation
  lcd_display u_lcd_display (
      //global clock
      .clk  (clk_vga),
      .rst_n(w_rstn_vga),
      .lcd_xpos(lcd_xpos),
      .lcd_ypos(lcd_ypos),
      .lcd_data(lcd_data)
  );
endmodule
```