基于 FPGA 的十通道逻辑分析仪设计教程

小梅哥编写， 可适用于芯航线 FPGA 学习套件， 作者保留一切所有权

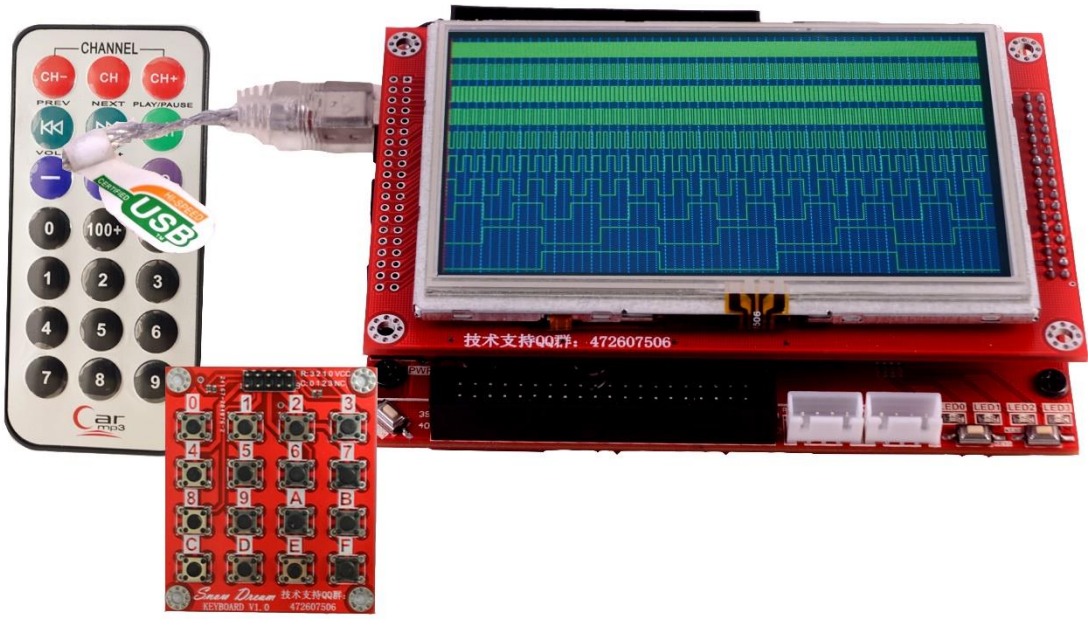
2016 年 7 月 11 日星期一

设计初衷

基于 FPGA 的十通道逻辑分析仪为芯航线为用户编写的精品例程，旨在通过例程给大家展示一个原理正确、代码规范、架构科学、层次清晰的示例设计方案，本实例中的所有模块均由小梅哥亲自编写并仿真测试通过，大家可以直接提取并用于自己的设计中。

基本功能介绍

10 路逻辑分析仪实例基于芯航线 FPGA 学习套件进行开发，实例使用到了芯航线 FPGA 学习套件的 FPGA 主板、4.3 寸 TFT 屏、矩阵键盘、红外遥控外设。设计使用 FPGA 采集 10 路数字波形数据，并存储在 FPGA 片上缓存 RAM 中，然后使用 4.3 寸 TFT 触摸液晶显示组件进行波形显示。整个逻辑分析仪系统可使用矩阵键盘或者红外遥控进行控制，以调整数据采样率、数据采样触发方式，并调整波形显示位置。在调整过程中，蜂鸣器在接收到控制信号后发出按键音乐，以给用户提供明确的按键反馈信号，提升人机交互的体验。同时，设计还支持外接基于 74HC595 驱动方案的 7 段 8 位数码管模块（接在 GPIO 0 上），用于显示当前系统的工作状态。系统整体如下图所示：

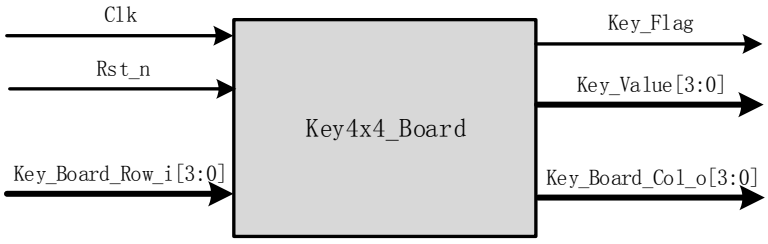


基本组件设计

矩阵键盘驱动设计

矩阵键盘原理与驱动设计

本例中，矩阵键盘主要用于实现人机交互的控制信号输入部分，通过矩阵键盘输入控制信号，来实现控制数据采样率、数据采样触发方式，并调整波形显示位置等功能，矩阵键盘的具体设计思路和方案见《芯航线 FPGA 数字系统设计教程+实例解析》“芯航线 FPGA 数字逻辑设计精品教程”部分的“FPGA 矩阵键盘驱动设计与验证”小节，本实例直接使用该小节设计的矩阵键盘驱动模块，因此不再重复介绍。设计完成的矩阵键盘驱动模块如下图所示。



模块每个信号的功能简介如下。

端口类型	端口名	描述
Input	Clk	系统时钟 50MHz
Input	Rst_n	系统复位，低有效
Input	Key_Board_Row_i	矩阵键盘行控制线，按键未按下为高电平
Output	Key_Board_Col_o	矩阵键盘列控制线，驱动列信号为低，实现按键状态扫描，
Output	Key_Value	输出键盘键值
Output	Key_Flag	按键检查成功标志信号，每当按键检测成功，产生一个时钟周期的高脉冲

矩阵键盘驱动使用示例

在使用本模块时，只需要将 Clk 连接到系统时钟输入，Rst_n 连接到系统复位引脚/信号，Key_Board_Row_i 和 Key_Board_Col_o 分别与矩阵键盘的行列引脚相连即可。然后每次当按下一次按键，待本模块进行抖动滤除并确认得到了按键按下信息后，会在 Key_Value 端口上输出当前检测到的按键值，并驱动 Key_Flag 信号产生一个时钟周期的高脉冲，其他模块在使用矩阵键盘的扫描结果时，只需要在 Key_Flag 信号有效时（高脉冲期间）读取 Key_Value 的值，并根据 Key_Value 的值进行相应的操作即可，例如，使用矩阵键盘的 1、3、5、7 键简单控制 4 个 LED 灯状态翻转的代码即可如下编写：

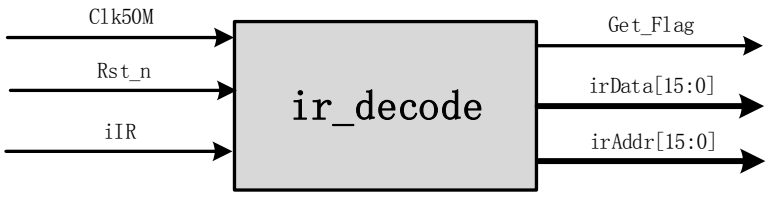
```
always@(posedge Clk or negedge Rst_n)
```

```
if(!Rst_n)  //复位时使 4 位 LED 全灭
    LED <= 4'b1111;
else if(Key_Flag)begin  //Key_Flag 高脉冲时有效
    case(Key_Value)
        4'd1:LED[0] <= ~LED[0];//按键 1，翻转 LED0
        4'd3:LED[1] <= ~LED[1];//按键 3，翻转 LED1
        4'd5:LED[2] <= ~LED[2];//按键 5，翻转 LED2
        4'd7:LED[3] <= ~LED[3];//按键 7，翻转 LED3
        default:LED <= LED; //其他按键，忽略，不对 LED 进行任何操作
    endcase
end
```

红外遥控解码驱动设计

红外遥控解码与驱动设计

本例中，红外遥控作为另一个输入设备，实现与矩阵键盘相同的功能，即用于实现人机交互的控制信号输入部分。通过红外遥控发射按键信号，并由 FPGA 解码红外接收器接收到的红外按键信号，得到按键信息，然后根据按键信息来实现控制数据采样率、数据采样触发方式，并调整波形显示位置等功能，红外遥控解码的具体设计思路和方案见《小梅哥 FPGA 设计思想与验证方法视频教程》“19_HT6221 红外遥控解码”一集视频教程，以及配套文档教程，见《芯航线 FPGA 数字系统设计教程+实例解析》“FPGA 设计思想与验证方法视频教程实验精讲手册”部分的“十九、 HT6221 红外遥控解码”小节，本实例直接使用该小节设计的红外遥控解码模块，因此不再重复介绍。设计完成的矩阵键盘驱动模块如下图所示。



模块每个信号的功能简介如下。

端口类型	端口名	描述
Input	Clk50M	系统时钟 50MHz
Input	Rst_n	系统复位，低有效
Input	iIR	红外遥控接收器信号脚
Output	Get_Flag	红外解码成功标志信号，每当解码成功，该信号产生一个时钟周期的高脉冲
Output	irData	红外解码得到的数据段,识别不同按键
Output	irAddr	红外解码得到的地址段，区分不同遥控器发出的数据。

其中，红外遥控解码得到的数据分为数据段和地址端，irData[15:0]为数据段信息，实际为 8 位有效数据，高 8 位为低八位的反码，例如，我们配套的红外遥控，按键 1 的键值为 0C，因此数据段的低八位则为 8'h0C，而高 8 位则为 8'h0C 的反码，即 8'hF3。同一个遥控，按下不同的按键，发出的信号传递的数据段是不一样的，因此我们对数据段进行解读，就能知道当前是哪个按键被按下，从而可以执行相应的响应。数据段的高 8 位和低 8 位进行比较，从而确定解码结果是否正确，如果高 8 位并非为低 8 位的反码，则表明当前解码结果有误。

irAddr[15:0]为地址段信息，不同厂家的遥控，通过设置不同的地址段，来进行区分，以避免在一个发射器的情况下发送指令被多个接收器同时接受。例如，使用海尔电视的专用的遥控器无法对海信电视进行控制，正是使用了不同的地址段来区分的。我们配套的遥控器，其地址段为 16' hFF00。

红外解码驱动使用示例

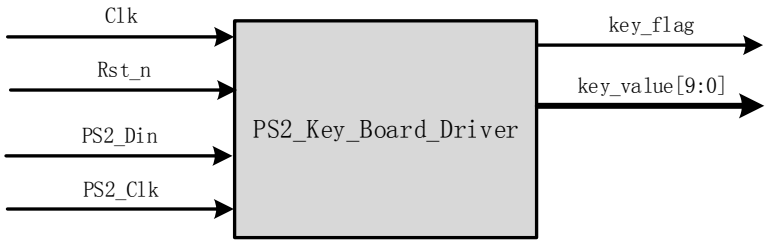
在使用本模块时，只需要将 Clk 连接到系统时钟输入，Rst_n 连接到系统复位引脚/信号，iIR 连接到红外接收器的数据输出脚即可。然后每次红外遥控按下一次按键，待本模块进行接收解码并确认得到了按键按下信息后，会在 irData 端口上输出当前检测到的数据段，irAddr 端口输出当前检测到的地址段，并驱动 Get_Flag 信号产生一个时钟周期的高脉冲，其他模块在使用红外解码模块的解码结果时，同矩阵键盘类似，只需要在 Get_Flag 信号有效时（高脉冲期间）读取 irData 和 irAddr 的值，并根据 irData 的值进行相应的操作即可，例如，使用地址码为 16' hFF00 的红外遥控的 1、3、5、7 键简单控制 4 个 LED 灯状态翻转的代码即可如下编写：

```
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)  //复位时使 4 位 LED 全灭
    LED <= 4'b1111;
else if(Get_Flag &&     //高脉冲时有效
       (irAddr == 16'hFF00) &&     //地址为 FF00
       (irData[7:0] == ~irData[15:8]))//高 8 位取反和低 8 位相等
begin
    case(irData[7:0])
        8'h0c:LED[0] <= ~LED[0];//键值 0c 对应按键 1，翻转 LED0
        8'h5e:LED[1] <= ~LED[1];//键值 5e 对应按键 3，翻转 LED1
        8'h1c:LED[2] <= ~LED[2];//键值 1c 对应按键 5，翻转 LED2
        8'h42:LED[3] <= ~LED[3];//键值 42 对应按键 7，翻转 LED3
       default:LED <= LED; //其他按键，忽略，不对 LED 进行任何操作
    endcase
end
```

PS2 键盘驱动设计

PS2 键盘驱动设计

本例中，PS2 键盘作为第三个可选输入设备，实现与矩阵键盘、红外遥控相同的功能，即用于实现人机交互的控制信号输入部分。使用时，如果开发板上外接了 PS2 键盘，那么按下 PS2 键盘上对应按键，则 FPGA 对 PS2 键盘产生的数据进行解码，从而得到按键信息，然后根据按键信息来实现控制数据采样率、数据采样触发方式，并调整波形显示位置等功能，PS2 键盘驱动的的具体设计思路和方案见《芯航线 FPGA 数字系统设计教程+实例解析》“芯航线 FPGA 数字逻辑设计精品教程”部分的“PS2 键盘解码驱动设计与验证”小节，本实例直接使用该小节设计的 PS2 键盘驱动模块，因此不再重复介绍。设计完成的 PS2 键盘驱动模块如下图所示。



详细端口名及其意义如下

端口说明	
端口名	端口功能或意义
Rst_n	全局复位
Clk	系统时钟输入端口，默认 50M
PS2_Din	PS2 接口数据线
PS2_Clk	PS2 接口时钟线
Key_Value	按键检测结果输出，总共 10 位，其中最高位为通/断码标志位，为 0 表示通码，为 1 表示断码（按键释放）；次高位为短码和长码（扩展码）标志位，为 0 表示短码，为 1 表示长码；低 8 位为数据位。
Key_Flag	按键检测成功标志信号，每当解码成功，该信号产生一个时钟周期的高脉冲

PS2 键盘驱动使用示例

在使用本模块时，只需要将 Clk 连接到系统时钟输入，Rst_n 连接到系统复位引脚/信号，PS2_Din 和 PS2_Clk 分别连接到 PS2 接口的数据脚和时钟脚即可。然后每次按下或释放键盘上的按键，PS2 键盘发送一次按键信息，待本模块正确解析确认得到了按键信息后，会在 Key_Value 端口上输出当前检测到的按键信息（键值、通断码、长短码），并驱动 Key_Flag 信号产生一个时钟周期的高脉冲，其他模块在使用 PS2 解码模块的解码结果时，同矩阵键盘类似，只需要在 Key_Value 信号有效时（高脉冲期间）读取 Key_Value 的值，并根据 Key_Value 的值进行相应的操作即可。

Key_Value 总共 10 位，其中最高位为通/断码标志位，次高位为短码和长码（扩展码）标志位，低 8 位为数据位。

9	8	7	6	5	4	3	2	1	0
通断码	长短码	数据码							

- 通断码：为 0 表示通码，代表此次为按键按下事件，为 1 表示断码，表明此次为按键释放事件；
- 长短码：为 0 表示短码，为 1 表示长码；键盘上部分按键的键码为短码，部分为长码，具体的按键码值，可以参见 PS2 键码对照表（见 PS2 教程小节）。
- 数据码：为当前按键的码值。具体的按键码值，可以参见 PS2 键码对照表（见 PS2 教程小节）。

因此，当我们使用 PS2 键盘的结果时，只需要根据 Key_Value 的最高位即可确定当前是按下按键还是释放按键。通过次高位的短码/长码标志位与低 8 位数据位共同作用，确定当前按下的按键键位是哪个，然后即可判断得到具体的按键信息，从而进行相应的操作。例如，每次按下 PS2 键盘上的 1、3、5、7 键，则对应控制 4 个 LED 灯状态翻转的代码即可如下编写：

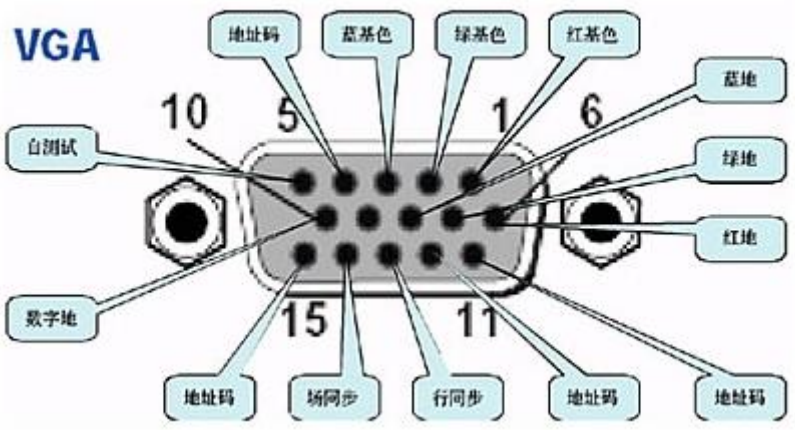
```
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)  //复位时使 4 位 LED 全灭
    LED <= 4'b1111;
else if(Key_Flag &&     //高脉冲时有效
       !Key_Value[9])//通断码，为 0 表示为通码，即按下.本例仅使用按下事件
begin
    case(Key_Value[8:0])
        9'h016:LED[0] <= ~LED[0];//键值 016 对应按键 1，翻转 LED0
        9'h026:LED[1] <= ~LED[1];//键值 026 对应按键 3，翻转 LED1
        9'h02E:LED[2] <= ~LED[2];//键值 02E 对应按键 5，翻转 LED2
        9'h03D:LED[3] <= ~LED[3];//键值 03D 对应按键 7，翻转 LED3
        default:LED <= LED; //其他按键，忽略，不对 LED 进行任何操作
    endcase
end
```

VGA 控制器设计

VGA 标准介绍

计算机显示器有许多现实标准，常见的有 VGA、SVGA 等，在这里我们用 VGA 接口来控制显示器，VGA 是 Video Graphics Adapter（Array）的缩写，即视频图形阵列。作为一种标准的显示接口得到广泛的应用。VGA 接口常使用 15 针的 DB15 接口，该接口引脚功能如下表所示：

引脚	名称	注释	引脚	名称	注释
1	RED	红基色 75R, 0.7V_{PP}	9	KEY	保留
2	GREEN	绿基色 75R, 0.7V_{PP}	10	SGND	同步信号地
3	BLUE	蓝基色 75R, 0.7V_{PP}	11	ID0	显示器标识位 0
4	ID2	显示器标识位 2	12	ID1/SDA	显示器标识位 1
5	GND	地	13	HSYNC/CSYNC	行同步或者复合同步
6	RGND	红色地	14	VSYNC	场同步
7	GGND	绿色地	15	ID3/SCL	显示器标识位 3
8	BGND	蓝色地			



VGA 扫描方式

在 VGA 标准兴起的时候。常见的彩色显示器一般由 CRT（阴极射线管）构成，色彩是由 RGB（红、绿、蓝）三基色组成。显示是用逐行扫描的方式解决。阴极射线枪发出电子束打在涂有荧光粉的荧光屏上，产生 RGB 三基色，合成一个彩色像素，扫描从屏幕的左上方开始，从左到右，从上到下进行扫描，每扫完一行，电子束都回到屏幕的左边下一行的起始位置。在这期间，CRT 对电子束进行消隐。每行结束时，用行同步信号进行行同步；扫描完所有行，用场同步信号进行场同步，并使扫描回到屏幕的左上方。同时进行场消隐，预备下一

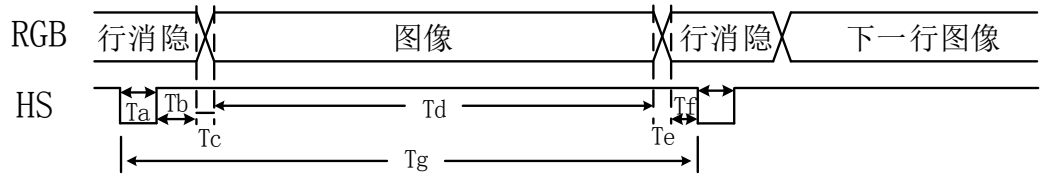
场的扫描。

随着显示技术的发展，出现了液晶显示器，液晶显示器的成像原理与 CRT 不同，液晶显示器是通过对液晶像素点单元施加电压与否，来控制液晶单元的透明程度，并添加三色滤光片、分别使 R、G、B 这 3 种光线透过滤光片，最后通过 3 个像素点合成一个彩色像素点，从而实现彩色显示。但是由于液晶显示技术后于 CRT 显示技术诞生，因此在液晶显示器诞生的时候，为了能够兼容传统的显示接口，因此液晶显示器通过内部电路实现了对 VGA 接口的完全兼容。因此，我们在使用显示器时，只要该显示器带有标准的 VGA 接口，我们就不用去关心其成像原理，直接使用标准的 VGA 时序即可驱动。

对于普通的显示器（无论是液晶还是 CRT），共有 5 个信号：R、G、B 三基色信号，行同步信号 HS，场同步信号 VS。对于时序驱动，VGA 显示器要严格遵循“VGA 工业标准”，即 640*480*60Hz 模式，否则可能会损害 VGA 显示器。

VGA 标准时序分析

通常我们所用的显示器都满足工业标准，因此我们设计 VGA 控制器时要参考显示器的技术规格，下图是 VGA 行扫描、场 4 扫描的时序图。



行扫描时序要求（单位：输出一个像素的时间间隔，即像素时钟）：

Ta（行同步头）：96

Tb：40

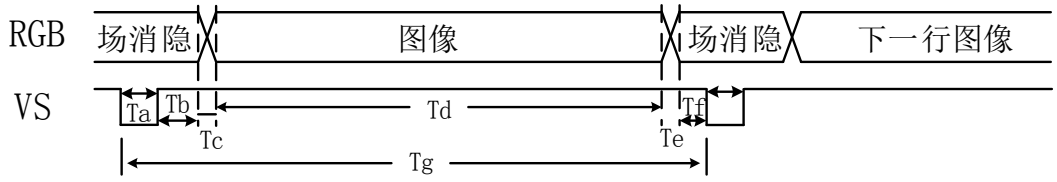
Tc：8

Td（行图像）：640

Te：8

Tf：8

Tg：800



场扫描时序要求（单位：输出一行 Line 的时间间隔）：

Ta（场同步头）：2

Tb：25

Tc：8

Td（场图像）：480

Te：8

Tf：2

Tg：525

VGA 工业标准所要求的频率如下：

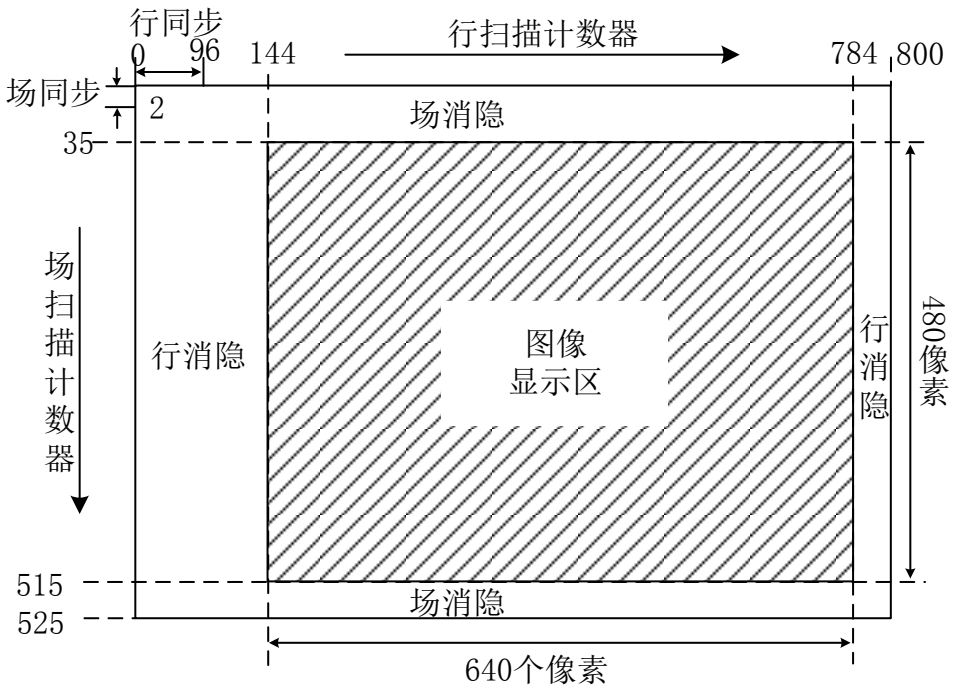
时钟频率 25.175MHz（像素输出的频率）

行频率 31469 Hz

场频率 59.94Hz（每秒图像刷频率）

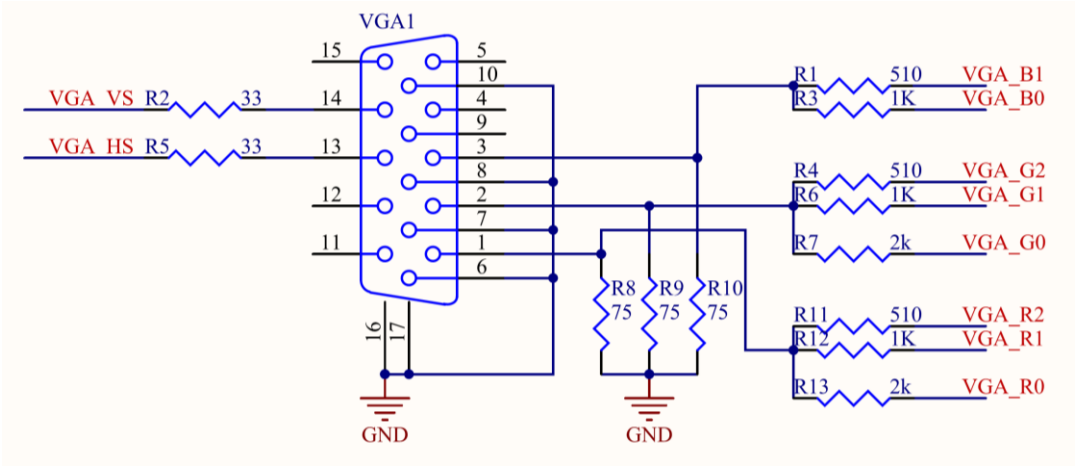
VGA 工业标准显示模式要求：行同步、列同步都为负极性，即同步脉冲要求是负脉冲。

下图为 VGA 图像显示扫描示意图，在设计时，可用两个计数器进行计数（行、场扫描计数器），行计数器的驱动时钟为 25MHz，场计数器的驱动时钟为行计数器的溢出信号。计数的同时控制行、场同步信号输出。并在适当的时候送出数据，就能显示相应的图像。注意消隐期间送出的数据应该为 0x00。显示器的刷新频率为 25MHz/800/525 = 59.52Hz，接近 VGA 工业标准场帧频 59.94Hz



芯航线 FPGA 学习套件 VGA 电路介绍

芯航线 FPGA 学习套件提供两种 VGA 接口输出，分别为 8bit 输出和高质量 24 位输出。24 位高质量的 VGA 视频输出模块使用专用视频 DAC 芯片 GM7123（兼容 ADV7123），实现高达 1600*1200 分辨率、60Hz 刷新频率，24 位动态色彩输出。该模块性能优异，价格较高，主要用于视频图像处理系统中。本例因为设计的逻辑分析仪不需要如此高的动态色彩范围，因此使用 8bit 输出型 VGA 模块。8 位 VGA 输出电路设计在“VGA_数码管_PS2”三合一模块上，下图为 VGA_数码管_PS2 模块图和 VGA 接口电路图：



该 VGA 接口三基色信号 R、G、B 共专用 8 位（分别是 R 为 3 位、G 为 3 位、B 为 2 位），因此可以显示 256 种颜色。RGB 数据的格式如下表所示：

D7	D6	D5	D4	D3	D2	D1	D0
R2	R1	R0	G2	G1	G0	B1	B0

以下为常见的几种颜色对应的数据编码：

颜色	黑	蓝	红	紫	绿	青	黄	白
R	0	0	1	1	0	0	1	1
G	0	0	0	0	1	1	1	1
B	0	1	0	1	0	1	0	1
数据编码	0x00	0x03	0xE0	0xE3	0x1C	0x1F	0xFC	0xFF

小结

通过以上介绍，我们了解了实现 VGA 驱动的行列扫描方法，即使用两个计数器分别进行行、场计数，根据计数值确定像素数据内容和行、场同步信号的电平状态。同时，也知道了要显示不同的颜色，只需要给 D0~D7 不同的数据，即可显示不同的颜色。

VGA 控制器设计

第一步，设计行扫描计数器

行扫描计数器即每个像素时钟自加 1，一旦加满到 799（刚好 800 个时钟周期），计数器清零并重新计数，该部分代码可如下设计：

```
    reg [9:0] hcount_r;     //VGA 行扫描计数器

    //**********************VGA 驱动部分**********************
    //行扫描计数器
    always@(posedge Clk25M or negedge Rst_n)
    if(!Rst_n)  //复位时，让行扫描计数器清零
        hcount_r<=10'd0;
    else if(hcount_r==10'd799)  //当一行数据扫完后，再次清零行扫描计数器
        hcount_r<=10'd0;
    else    //0~799 之间，每个像素时钟时行扫描计数器自加 1
        hcount_r<=hcount_r+10'd1;
```

第二步，设计场扫描计数器

由于场扫描计数器是在每次一行扫描完成后加 1 的，即场扫描计数器的自加条件是行扫描计数器溢出。所以，场扫描计数器的自加条件为行扫描完成，即“hcount_r==10'd799”，场扫描计数器代码如下所示：

```
    reg [9:0] vcount_r;     //VGA 场扫描计数器

    //场扫描
    always@(posedge Clk25M or negedge Rst_n)
    if(!Rst_n)  //复位时让场计数器清零
        vcount_r<=10'd0;
    else if(hcount_r==10'd799) begin    //每次一行扫描完成
        if(vcount_r==10'd524)   //每次一场扫描结束，清零计数器
            vcount_r<=10'd0;
```

```
        else
            vcount_r<=vcount_r+10'd1;//场计数器在 0~524，满足条件，自加 1
    end
    else //不满足行扫描结束条件器件，让场扫描计数器保持不变
        vcount_r<=vcount_r;
```

第三步，产生行同步信号和场同步信号

根据 VGA 工业标准时序，我们知道每一个完整的 VGA 帧都包含了数据段和消隐段，在消隐段期间，行同步信号和列同步信号有一段行同步头和场同步头，在同步期间，对应行同步信号或者场同步信号为低电平，因此我们可以根据行、场计数器的值来确定行、场同步信号的电平状态。对于行同步信号，其行同步头为一行扫描的前 96 个像素时钟周期，因此行同步信号可用如下的简单方式控制：

```
assign VGA_HS=(hcount_r>10'd95);
```

对于场同步信号，其场同步头为一行扫描的前 2 个像素时钟周期，因此行同步信号可用如下的简单方式控制：

```
assign VGA_VS=(vcount_r>10'd1);
```

第四步，输出数据

VGA 控制器的设计目的是为了驱动 VGA 显示器显示需求的图像内容，因此需要设计数据输出部分，这里，数据来源可以为其它部分产生的图像信号，如摄像头数据、BMP 图片数据。我们在驱动 VGA 时，只需要保证在扫描正确的像素点时，其它部分产生的图像信号能够与该像素点位置对应上，则不需要对图像数据再进行二次处理，但是，在行、场消隐期间，需要保证输出到 VGA 的 RGB 数据线上的数据全部为 0，因此可以设置一个二选一多路器，只有在非消隐期间，VGA 控制器才直接输出其他部分输入的图像数据，而消隐器件则强制输出全 0。

我们可以首先产生一个图像数据有效标志信号，然后使用该标志信号控制 VGA 输出数据的内容，即切换二选一多路器的通道，从而实现消隐器件数据全 0 的功能。

图像数据有效标志信号产生代码如下所示：

```
//数据、同步信号输出
assign dat_act=((hcount_r>=10'd143)&&(hcount_r<10'd783))
            &&((vcount_r>=10'd34)&&(vcount_r<10'd514));
```

dat_act 即为图像数据有效标志信号。

消隐强制输出 0 二选一多路器代码如下所示：

```
assign VGA_RGB=(dat_act)?data_in:8'h00;
```

其中，VGA_RGB 是输出到 VGA 接口上的数据，而 data_in 则是其他模块传递过来的正确的图像数据。

第五步，输出正确的行列扫描位置

为了使其他模块能够根据当前扫描位置正确的输出图像数据，因此需要将 VGA 控制器的实时扫描位置输出，以供其他模块使用。

```
    assign hcount=hcount_r-10'd143;
    assign vcount=vcount_r-10'd34;
```

完整 VGA 控制器设计

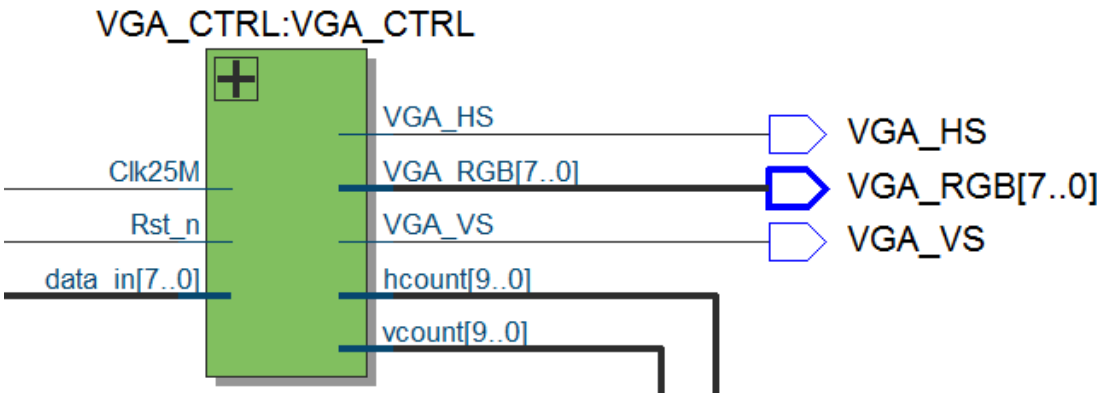
以上为我们根据直观思维设计的驱动电路，在代码中，直接使用了数字作为运算和比较的内容，这样不利于修改。因此，为了实现易于修改的控制器设计，方便后期简单修改后兼容其他分辨率，对代码进行优化，使用参数化设计。将代码中使用到的一些与时序相关的数字直接使用 parameter 这样的参数进行定义，这样在以后需要修改时间参数时，只需要修改 parameter 定义的内容即可，不需要再深入到代码中一个一个修改。这里不再一一介绍如何修改，只贴出最终设计修改完成的代码，请用户自行比对领悟。

```
module VGA_CTRL(
    Clk25M, //系统输入时钟 25MHZ
    Rst_n,  //复位输入，低电平复位
    data_in,    //待显示数据
    hcount,     //VGA 行扫描计数器
    vcount,     //VGA 场扫描计数器
    VGA_RGB,    //VGA 数据输出
    VGA_HS,     //VGA 行同步信号
    VGA_VS      //VGA 场同步信号
);

    //-----------------模块输入端口-----------------
    input  Clk25M;          //系统输入时钟 25MHZ
    input  Rst_n;
    input  [7:0]data_in;     //待显示数据

    //-----------------模块输出端口-----------------
    output [9:0]hcount;
    output [9:0]vcount;
    output [7:0]VGA_RGB;  //VGA 数据输出
    output VGA_HS;          //VGA 行同步信号
    output VGA_VS;          //VGA 场同步信号

    //-----------------内部寄存器定义-----------------
```

```
    reg [9:0] hcount_r;     //VGA 行扫描计数器
    reg [9:0] vcount_r;     //VGA 场扫描计数器
    //-----------------内部连线定义----------------
    wire hcount_ov;
    wire vcount_ov;
    wire dat_act;//有效显示区标定

    //VGA 行、场扫描时序参数表
    parameter VGA_HS_end=10'd95,
              hdat_begin=10'd143,
              hdat_end=10'd783,
              hpixel_end=10'd799,
              VGA_VS_end=10'd1,
              vdat_begin=10'd34,
              vdat_end=10'd514,
              vline_end=10'd524;

    assign hcount=hcount_r-hdat_begin;
    assign vcount=vcount_r-vdat_begin;

    //**********************VGA 驱动部分**********************
    //行扫描
    always@(posedge Clk25M or negedge Rst_n)
    if(!Rst_n)
        hcount_r<=10'd0;
    else if(hcount_ov)
        hcount_r<=10'd0;
    else
        hcount_r<=hcount_r+10'd1;

    assign hcount_ov=(hcount_r==hpixel_end);

    //场扫描
    always@(posedge Clk25M or negedge Rst_n)
    if(!Rst_n)
        vcount_r<=10'd0;
    else if(hcount_ov) begin
        if(vcount_ov)
            vcount_r<=10'd0;
        else
            vcount_r<=vcount_r+10'd1;
    end
    else
        vcount_r<=vcount_r;
```

```
    assign  vcount_ov=(vcount_r==vline_end);

    //数据、同步信号输出
    assign dat_act=((hcount_r>=hdat_begin)&&(hcount_r<hdat_end))
                  &&((vcount_r>=vdat_begin)&&(vcount_r<vdat_end));

    assign VGA_HS=(hcount_r>VGA_HS_end);
    assign VGA_VS=(vcount_r>VGA_VS_end);
    assign VGA_RGB=(dat_act)?data_in:8'h00;

endmodule
```

设计完成后，在 Quartus II15.1 中综合出来的电路符号如下所示：



每个端口的功能如下表所示：

端口名	端口功能
Clk25M	VGA 像素时钟，25MHz
Rst_n	系统复位，低电平复位
data_in[7:0]	待显示数据输入端口
VGA_HS	VGA 接口行同步信号
VGA_VS	VGA 接口场同步信号
VGA_RGB[7:0]	VGA 三元色数据输出
hcount[9:0]	图像区行扫描地址
vcount[9:0]	图像区场扫描地址

VGA 控制器仿真验证

本小节对设计的 VGA 控制器进行仿真验证，通过仿真查看行场同步信号是否满足设计需求。

Testbench 设计

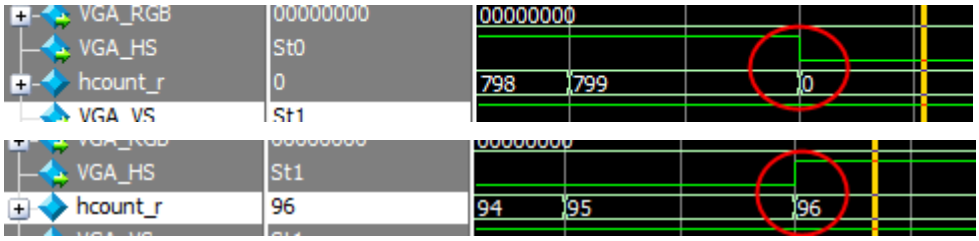
Testbench 的设计思路非常简单，只需要产生一个 25MHz 的时钟信号，然后在 data_in 端口上给一个固定的数据编码，为了与消隐时候的强制输出全 0 相区分，因此只需要使 data_in 上的数据不为 0 即可。testbench 内容如下所示：

```
`timescale 1ns/1ns

`define clk_period 40

module VGA_CTRL_tb;
    //-----------------模块输入端口----------------
    reg  Clk25M;          //系统输入时钟 25MHZ
    reg  Rst_n;
    reg  [7:0]data_in;     //待显示数据

    //-----------------模块输出端口----------------
    wire [9:0]hcount;
    wire [9:0]vcount;
    wire [7:0]VGA_RGB;  //VGA 数据输出
    wire VGA_HS;          //VGA 行同步信号
    wire VGA_VS;          //VGA 场同步信号

    reg [11:0]V_cnt = 0;//扫描行数统计计数器

    VGA_CTRL VGA_CTRL(
        .Clk25M(Clk25M),    //系统输入时钟 25MHZ
        .Rst_n(Rst_n),
        .data_in(data_in),  //待显示数据
        .hcount(hcount),        //VGA 行扫描计数器
        .vcount(vcount),        //VGA 场扫描计数器
        .VGA_RGB(VGA_RGB),  //VGA 数据输出
        .VGA_HS(VGA_HS),        //VGA 行同步信号
        .VGA_VS(VGA_VS)     //VGA 场同步信号
    );

    initial Clk25M = 0;
    always #(`clk_period/2) Clk25M = ~Clk25M;

    initial begin
        Rst_n = 0;
        data_in = 8'd0;
        #(`clk_period *20 +1);
        Rst_n = 1;
```

```
        data_in = 8'hff;
    end

    initial begin
        wait(V_cnt == 3);//等待扫描 2 帧后结束仿真
        $stop;
    end

    always @(posedge VGA_VS)//统计总扫描帧数
        V_cnt <= V_cnt + 1'b1;

endmodule
```

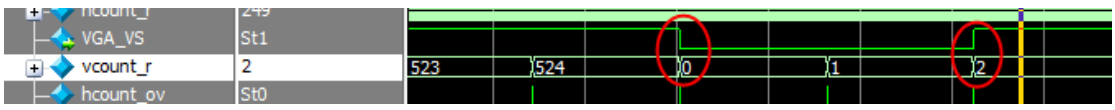
仿真结果分析

VGA_HS 信号：



由图可见，VGA_HS 在 0~95 这一行扫描段内为低电平，即行同步头，其他时间为高电平，行扫描一次，行扫描计数器计数最大值为 799，即刚好 800 个像素时钟周期，与设计一致，因此可知行扫描信号满足时序设计要求。

VGA_VS 信号：



由图可见，VGA_VS 信号在 0~1 这一段场扫描时间内为低电平，即场同步头，其他时间为高电平。场扫描一次，场扫描计数器计数最大值为 524，即刚好 525 个行扫描周期，与设计一致，满足 VGA 工业标准，因此可知场扫描信号满速时序设计要求。

其他信号本文不再进行详细分析比对，在进行板级调试中，如果发现显示效果不对，则可根据实际显示效果，判断错误位置，如行同步信号错误、场同步信号错误等。

VGA 控制器板级验证

在上一节，我们简述了 VGA 控制器的设计思路并给出了具体的 VGA 控制器设计过程，同时通过仿真验证了设计的合理性。本节，我们将对该 VGA 控制器进行板级验证，通过板级验证来进一步确定我们设计的正确性。

板级验证需求

VGA 的板级验证，主要验证以下三个方面：

1、 能够正确的全屏点亮屏幕，显示稳定
2、 能否正确的显示颜色，即按照需求制定需要显示的颜色
3、 能否正确的定位坐标，即实现在指定的位置显示对应的数据

板级验证电路设计

为此，我们设计一个测试工程，该工程中我们测试上述提到的 8 种颜色，通过颜色的位置，不但能确定是否能够正确输出指定颜色的图像，还能间接确定是否能够精确指定像素位置。

因此，我们对屏幕进行划分，将屏幕划分成 4 行 2 列总共八个像素阵列，每个阵列分别显示一种颜色。然后在屏幕的四角分别绘制一个线长为 11 个像素点的十字形光标。每个光标的颜色与当前位置显示的颜色相反。据此，我们可以首先定义每种颜色的具体数据编码，然后再定义每个像素阵列的基本显示颜色，这里首先使用 localparam 定义每种颜色的具体数据编码：

//定义颜色编码	//定义每个像素块的默认显示颜色值

localparam
BLACK = 0x00, //黑色
BLUE = 0x03, //蓝色
RED = 0xE0, //红色
PURPPLE = 0xE3, //紫色
GREEN = 0x1C, //绿色
CYAN = 0x1F, //青色
YELLOW = 0xFC, //黄色
WHITE = 0xFF; //白色
	localparam
	R0_C0 = BLACK, //第 0 行 0 列像素块
	R0_C1 = BLUE, //第 0 行 1 列像素块
	R1_C0 = RED, //第 1 行 0 列像素块
	R1_C1 = PURPPLE,//第 1 行 1 列像素块
	R2_C0 = GREEN, //第 2 行 0 列像素块
	R2_C1 = CYAN, //第 2 行 1 列像素块
	R3_C0 = YELLOW, //第 3 行 0 列像素块
	R3_C1 = WHITE; //第 3 行 1 列像素块

紧接着，我们需要知道 VGA 当前扫描的位置是在哪一个位置区间，换一种说法，我们需要通过 VGA 当前的扫描位置得到当前扫描的是哪一个像素阵列，然后给待显示数据赋予对应的颜色值即可。这里我们先定义每个像素块处于扫描中的条件。

1、 产生每一列的处于扫描状态标志信号，屏幕每行总共 640 个像素点，我们将屏幕划分成了 2 列，因此
   a) 当行扫描范围在 0~319 这一段像素内时，第 0 列处于活跃阶段；
   b) 当扫描范围在 320~639 这一段像素内时，第 1 列处于活跃阶段。

   因此可得：

```
wire C0_act = hcount >= 0 && hcount < 320; //正在扫描第 0 列
wire C1_act = hcount >= 320 && hcount < 640;//正在扫描第 1 列
```

2、 产生每一行的处于扫描状态标志信号，屏幕每列总共 480 个像素点，我们将屏幕划分成了 4 列，因此

a) 当行扫描范围在 0~119 这一段像素内时，第 0 行处于活跃阶段；

b) 当行扫描范围在 120~239 这一段像素内时，第 1 行处于活跃阶段；

c) 当行扫描范围在 240~359 这一段像素内时，第 2 行处于活跃阶段；

d) 当行扫描范围在 360~479 这一段像素内时，第 3 行处于活跃阶段，

因此可得：

```
wire R0_act = vcount >= 0 && vcount < 120;  //正在扫描第 0 行
wire R1_act = vcount >= 120 && vcount < 240;//正在扫描第 1 行
wire R2_act = vcount >= 240 && vcount < 360;//正在扫描第 2 行
wire R3_act = vcount >= 360 && vcount < 480;//正在扫描第 3 行
```

3、 产生扫描每一个像素块的标志信号：

```
wire R0_C0_act = R0_act & C0_act;   //第 0 行 0 列像素块被扫描中
wire R0_C1_act = R0_act & C1_act;   //第 0 行 1 列像素块被扫描中
wire R1_C0_act = R1_act & C0_act;   //第 1 行 0 列像素块被扫描中
wire R1_C1_act = R1_act & C1_act;   //第 1 行 1 列像素块被扫描中
wire R2_C0_act = R2_act & C0_act;   //第 2 行 0 列像素块被扫描中
wire R2_C1_act = R2_act & C1_act;   //第 2 行 1 列像素块被扫描中
wire R3_C0_act = R3_act & C0_act;   //第 3 行 0 列像素块被扫描中
wire R3_C1_act = R3_act & C1_act;   //第 3 行 1 列像素块被扫描中
```

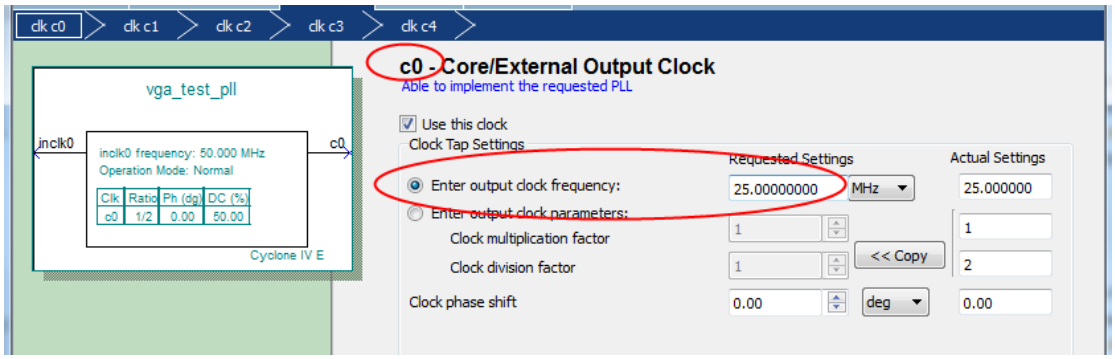
然后，我们就可以根据当前被扫描的像素块范围来确定需要给 VGA 输出什么颜色，这里采用一个多路器即可实现：

```
always@(*)
    case({R3_C1_act,R3_C0_act,R2_C1_act,R2_C0_act,
          R1_C1_act,R1_C0_act,R0_C1_act,R0_C0_act})
        8'b0000_0001:disp_data = R0_C0;
        8'b0000_0010:disp_data = R0_C1;
        8'b0000_0100:disp_data = R1_C0;
        8'b0000_1000:disp_data = R1_C1;
        8'b0001_0000:disp_data = R2_C0;
        8'b0010_0000:disp_data = R2_C1;
        8'b0100_0000:disp_data = R3_C0;
        8'b1000_0000:disp_data = R3_C1;
        default:disp_data = R0_C0;
    endcase
```

添加 PLL 时钟分频单元

通过以上步骤，我们就完成了简易 VGA 控制器测试电路的主要电路设计。在前面我们

曾经提到，VGA 控制器的像素时钟为 25MHz，而我们芯航线 FPGA 开发板设计的是 50MHz 的晶振，因此需要使用锁相环对时钟进行分频得到 25MHz 的时钟，以供 VGA 控制器使用。注意，虽然我们直接使用寄存器二分频也能从 50M 直接分频得到 25M 时钟，但是这样分频出来的时钟驱动能力是非常差的，抖动也非常大，不能再作为时序电路的时钟使用，因此这里必须使用 pll 来得到 25MHz 时钟。具体 PLL 配置请参考《芯航线 FPGA 数字系统设计教程+实例解析》“FPGA 设计思想与验证方法视频教程实验精讲手册”部分的“十六、 PLL 锁相环介绍与简单应用”小节。



完整的测试电路代码

实现完整的测试电路代码如下所示：

```
module VGA_CTRL_test(
    Clk,    //50MHZ 时钟
    Rst_n,
    VGA_RGB,//TFT 数据输出
    VGA_HS, //TFT 行同步信号
    VGA_VS  //TFT 场同步信号
);

    input Clk;
    input Rst_n;
    output [7:0]VGA_RGB;
    output VGA_HS;
    output VGA_VS;

    reg [7:0]disp_data;
    wire [9:0]hcount;
    wire [9:0]vcount;
    wire Clk25M;

    vga_test_pll vga_test_pll(
        .inclk0(Clk),
        .c0(Clk25M)
    );
```

```
    VGA_CTRL VGA_CTRL(
        .Clk25M(Clk25M),    //系统输入时钟 25MHZ
        .Rst_n(Rst_n),
        .data_in(disp_data),    //待显示数据
        .hcount(hcount),        //VGA 行扫描计数器
        .vcount(vcount),        //VGA 场扫描计数器
        .VGA_RGB(VGA_RGB),  //VGA 数据输出
        .VGA_HS(VGA_HS),        //VGA 行同步信号
        .VGA_VS(VGA_VS)     //VGA 场同步信号
    );

//定义颜色编码
localparam
    BLACK      = 8'h00, //黑色
    BLUE       = 8'h03, //蓝色
    RED    = 8'hE0, //红色
    PURPPLE = 8'hE3, //紫色
    GREEN      = 8'h1C, //绿色
    CYAN       = 8'h1F, //青色
    YELLOW  = 8'hFC, //黄色
    WHITE      = 8'hFF; //白色

//定义每个像素块的默认显示颜色值
localparam
    R0_C0 = BLACK,  //第 0 行 0 列像素块
    R0_C1 = BLUE,   //第 0 行 1 列像素块
    R1_C0 = RED,    //第 1 行 0 列像素块
    R1_C1 = PURPPLE,//第 1 行 1 列像素块
    R2_C0 = GREEN,  //第 2 行 0 列像素块
    R2_C1 = CYAN,   //第 2 行 1 列像素块
    R3_C0 = YELLOW, //第 3 行 0 列像素块
    R3_C1 = WHITE;  //第 3 行 1 列像素块

    wire R0_act = vcount >= 0 && vcount < 120;  //正在扫描第 0 行
    wire R1_act = vcount >= 120 && vcount < 240;//正在扫描第 1 行
    wire R2_act = vcount >= 240 && vcount < 360;//正在扫描第 2 行
    wire R3_act = vcount >= 360 && vcount < 480;//正在扫描第 3 行

    wire C0_act = hcount >= 0 && hcount < 320; //正在扫描第 0 列
    wire C1_act = hcount >= 320 && hcount < 640;//正在扫描第 1 列

    wire R0_C0_act = R0_act & C0_act;   //第 0 行 0 列像素块被扫描中
    wire R0_C1_act = R0_act & C1_act;   //第 0 行 1 列像素块被扫描中
```

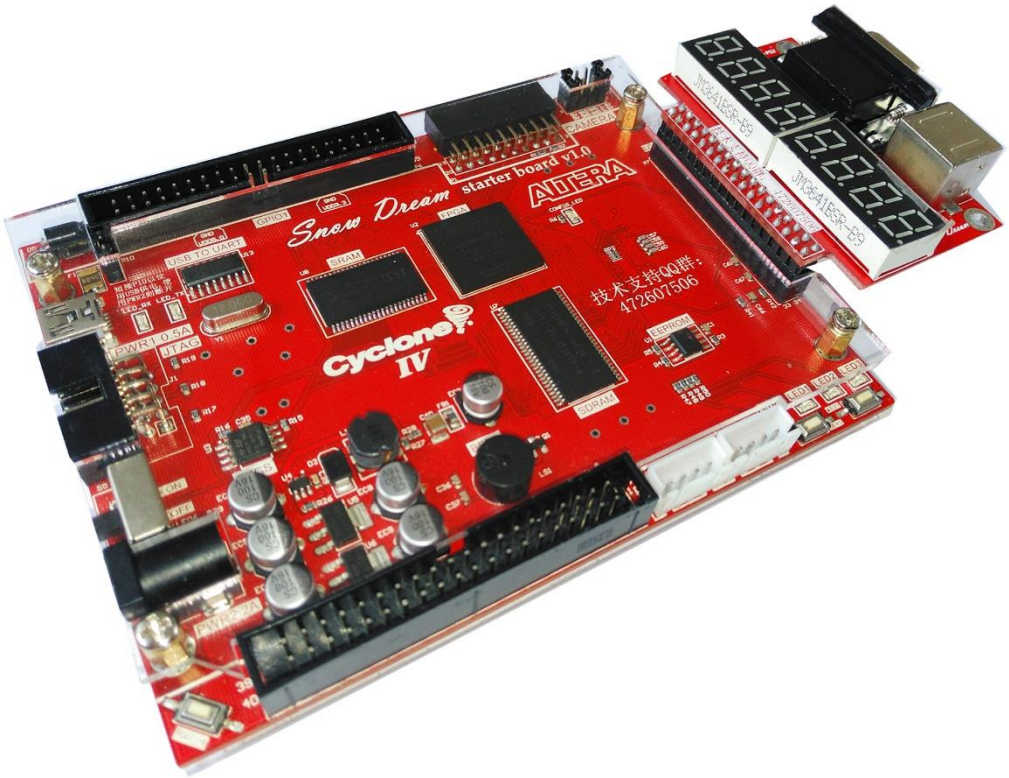
```
    wire R1_C0_act = R1_act & C0_act;   //第 1 行 0 列像素块被扫描中
    wire R1_C1_act = R1_act & C1_act;   //第 1 行 1 列像素块被扫描中
    wire R2_C0_act = R2_act & C0_act;   //第 2 行 0 列像素块被扫描中
    wire R2_C1_act = R2_act & C1_act;   //第 2 行 1 列像素块被扫描中
    wire R3_C0_act = R3_act & C0_act;   //第 3 行 0 列像素块被扫描中
    wire R3_C1_act = R3_act & C1_act;   //第 3 行 1 列像素块被扫描中

    always@(*)
        case({R3_C1_act,R3_C0_act,R2_C1_act,R2_C0_act,
              R1_C1_act,R1_C0_act,R0_C1_act,R0_C0_act})
            8'b0000_0001:disp_data = R0_C0;
            8'b0000_0010:disp_data = R0_C1;
            8'b0000_0100:disp_data = R1_C0;
            8'b0000_1000:disp_data = R1_C1;
            8'b0001_0000:disp_data = R2_C0;
            8'b0010_0000:disp_data = R2_C1;
            8'b0100_0000:disp_data = R3_C0;
            8'b1000_0000:disp_data = R3_C1;
            default:disp_data = R0_C0;
        endcase

endmodule
```

板级验证

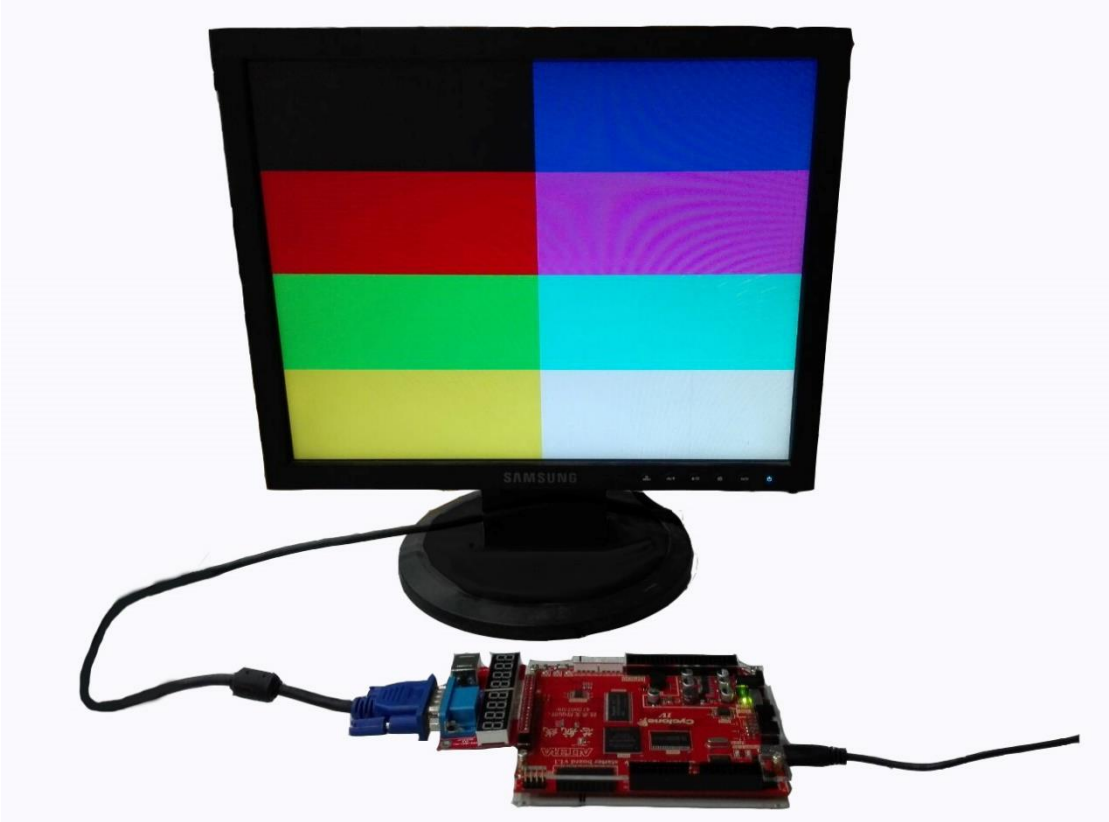
引脚分配，VGA 控制器测试工程引脚分配表如下所示：

VGA 模块信号名	对应 FPGA 引脚名称
VGA_RGB7	PIN_C14
VGA_RGB6	PIN_F11
VGA_RGB5	PIN_C11
VGA_RGB4	PIN_F9
VGA_RGB3	PIN_D11
VGA_RGB2	PIN_D9
VGA_RGB1	PIN_E11
VGA_RGB0	PIN_D12
VGA_HS	PIN_D14
VGA_VS	PIN_F10

芯航线 FPGA 学习套件主板与“VGA 数码管 PS2”三合一模块的连接如下所示：



最终测试效果如下图所示：



通过照片可知，VGA 控制器设计能够稳定正确的刷新 VGA 显示器并控制正确的显示位置，因此设计无误。

后续，我们就可以使用该控制器再结合一定的图像信号产生电路实现更多更负责的显示系统设计。当然，也可能根据具体的使用环境，再对本控制器进行设计微调。

无源蜂鸣器驱动设计

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型。

压电式蜂鸣器：压电式蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成。有的压电式蜂鸣器外壳上还装有发光二极管。

多谐振荡器由晶体管或集成电路构成。当接通电源后（1.5~15V 直流工作电压）,多谐振荡器起振,输出 1.5~2.5kHZ 的音频信号，阻抗匹配器推动压电蜂鸣片发声。

电磁式蜂鸣器：电磁式蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。

接通电源后，振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。

根据蜂鸣器本身是否集成了震荡源，蜂鸣器可以分为有源蜂鸣器与无源蜂鸣器。

有源蜂鸣器直接接上额定电源(新的蜂鸣器在标签上都有注明)就可连续发声；而无源蜂鸣器则和电磁扬声器一样，需要接在音频输出电路中才能发声。

有源蜂鸣器与无源蜂鸣器的区别：

注意：这里的“源”不是指电源，而是指震荡源。

也就是说，有源蜂鸣器内部带震荡源，所以只要一通电就会叫；

而无源内部不带震荡源，所以如果用直流信号无法令其鸣叫。必须用 2K-5K 的方波去驱动它

有源蜂鸣器往往比无源的贵，就是因为里面多个震荡电路。

无源蜂鸣器的优点是：

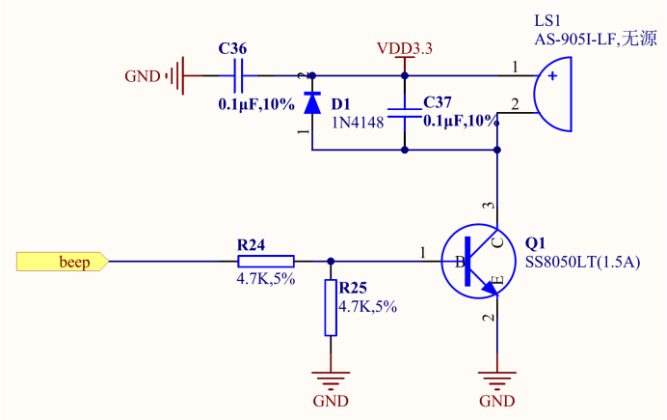
1. 便宜
2. 声音频率可控，可以做出“多来米发索拉西”的效果
3. 在一些特例中，可以和 LED 复用一个控制口

有源蜂鸣器的优点是：程序控制方便。

以上介绍了蜂鸣器的种类以及有源蜂鸣器、无源蜂鸣器的特点。接下来，我们将介绍芯航线 FPGA 学习套件主板上使用的蜂鸣器电路，并使用 Verilog 设计一个蜂鸣器驱动电路，来驱动蜂鸣器发声。

蜂鸣器电路介绍

芯航线 FPGA 学习套件主板上使用了一枚 3.3V 驱动的无源蜂鸣器，其电路如下所示：



电容 C37 用于提高电路抗干扰性能。 D1 起保护三极管的作用，当三极管突然截止时，无源蜂鸣器两端产生的瞬时感应电动势可以通过 D3 迅速释放掉，避免叠加到三极管集电极上从而击穿三极管。

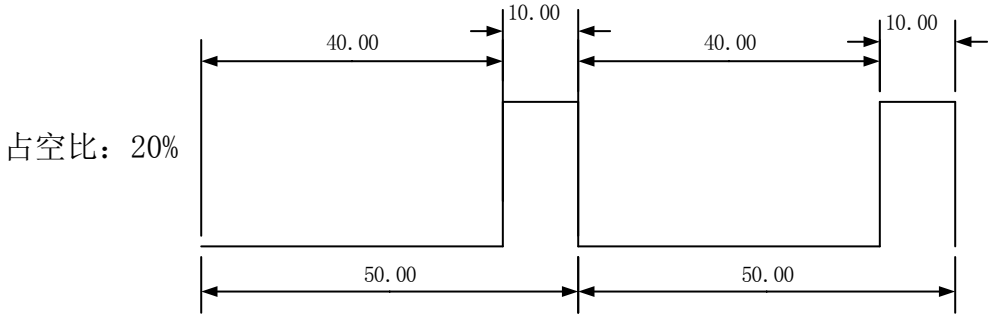
beep 端口接 FPGA 输出管脚，使用时，只需要在 beep 信号上输出 2~5KHz 的 pwm 波，就能驱动蜂鸣器发声。

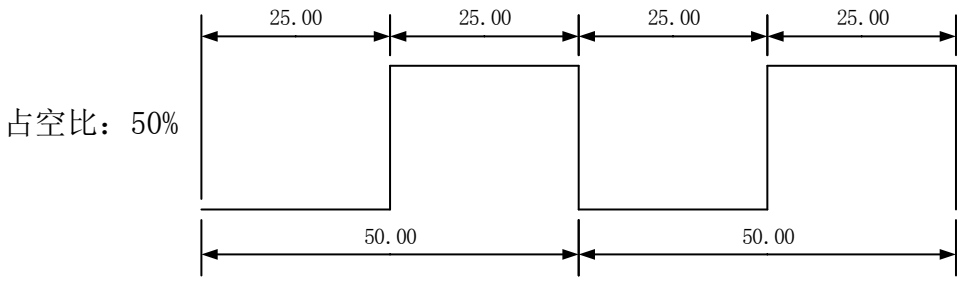
无源蜂鸣器控制器设计

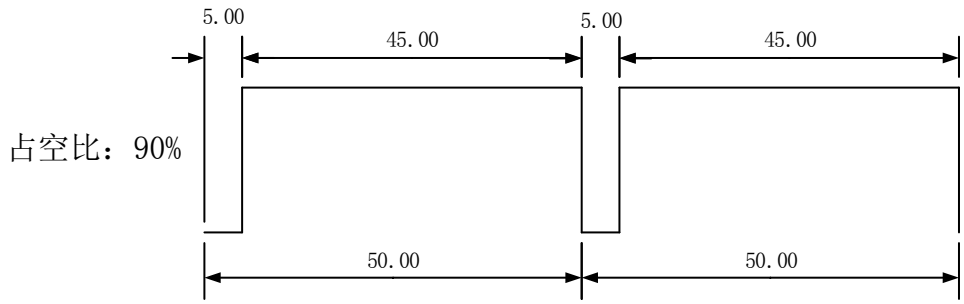
通过前面对无源蜂鸣器的特点介绍可知，要使无源蜂鸣器能够正常发声，需要在控制端 beep 给出相应频率的 PWM 波。因此，对于无源蜂鸣器的控制，就转化为了设计一个 PWM 波发生电路。因此，接下来我们将介绍 PWM 波发生电路的设计。

何为 PWM 波？PWM 的英文全名叫 Pulse Width Modulation，即脉冲宽度调制。通过对一系列脉冲的宽度进行调制，来等效地获得所需要波形(含形状和幅值)。 PWM 广泛应用在从测量、通信到功率控制与变换的许多领域中。

以下为周期为 1KHz，脉冲宽度（占空比）分别为 20%、50%、90%的波形图：



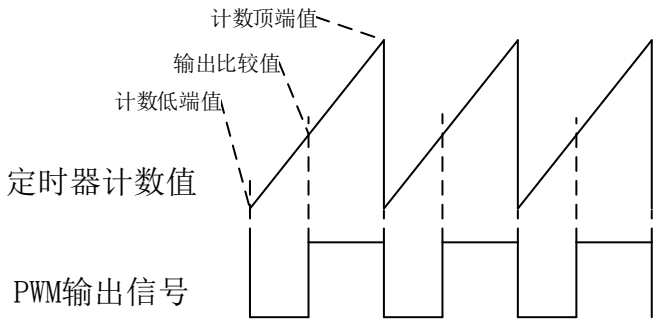




由图可知，当信号周期一定，信号高电平时间所占总时间的百分比不一样，即为不同占空比的 PWM 波。在逆变电路中，当使用这样的波形去驱动 MOS 管的导通时，因为一个周期内不同占空比的 PWM 信号其高电平持续长度不一样，因此使得 MOS 管的开通时间也不一样，从而使得电路中的平均电流也不一样，因此，通过调整驱动信号的占空比即可调整被控制电路中的平均电流。

而除了调整 PWM 信号的占空比，PWM 信号的周期也是可以调整的，例如，在逆变电路中，使用 IGBT 作为开关器件，常见开关频率为几 K 到几十 K，而使用 MOS 管作为开关器件，其开关频率则可高达几百 K。因此，对于不同的器件，对驱动信号的频率要求也不一样。所以，还需要能够对 PWM 波的频率进行调整。

通过以上分析，可以知道，要设计一个 PWM 发生电路，需要能够实现对信号的频率和占空比的调节。使用过单片机或者 DSP 产生 PWM 波的朋友应该知道，在单片机或者 DSP 中，产生 PWM 波的方法就是使用片上定时器进行循环计数，通过设定定时器的一个定时周期时长来确定对应输出 PWM 信号的频率，同时还有一个比较器，该比较器比较定时器的实时计数值与用户设定的比较值的大小，根据比较结果来控制输出信号的电平高低。通过设定不同的比较值，即可实现不同占空比的 PWM 信号输出。



对于 FPGA 来说，要产生 PWM 波，也可以借鉴单片机或 DSP 使用定时器产生 PWM 波的思路。

基于 FPGA 的 PWM 电路设计

根据 DSP 产生 PWM 波典型原理，在 FPGA 中设计 PWM 发生器时，也可提取出如下两个主要电路：

1、 定时器/计数器电路

2、 输出比较电路

定时器/计数器电路设计

定时器电路设计较为简单，在《小梅哥 FPGA 设计思想与验证方法视频教程》中，04 课“计数器设计与验证”介绍了最简单的计数器设计。参考各种 MCU 的计数器输出 PWM 波时的典型配置，可知该定时/计数器采用循环递减的计数方式，计数器循环从设定的计数初始值递减到 0，然后再回到计数初始值再次递减。这样，只需要设定一个计数初始值，并确定计数时钟源频率，即可确定计数一个完整周期的时间，也即 PWM 信号频率。

在本节中，我们设计定时/计数器的计数时钟源频率为芯航线 FPGA 学习套件主板上晶体振荡器的输出频率 50MHz，定时/计数器位宽为 32 位，则计数器代码如下所示：

```
reg [31:0]counter;//定义 32 位计数器
reg [31:0]counter_arr;//定义 32 位预重装寄存器
always@(posedge Clk50M or negedge Rst_n)
if(!Rst_n)
    counter <= 32'd0;
else if(cnt_en)begin
    if(counter == 0)
        counter <= counter_arr;//计数到 0，加载自动预重装寄存器值
    else
        counter <= counter - 1'b1;//计数器自减 1
end
else
    counter <= counter_arr; //没有使能时，计数器值等于预重装寄存器值
```

输出比较电路

输出比较电路通过比较计数器实时计数值与比较寄存器中的设定值，来确定最终 PWM 输出信号的电平状态。这里，我们可以定义，当计数器计数值大于等于比较值时，PWM 输出端输出低电平，当计数值小于比较值时，PWM 输出端输出高电平。因此输出比较电路设计代码如下：

```
reg o_pwm;//pwm 输出信号
reg [31:0]counter_ccr;//定义 32 位输出比较寄存器
always@(posedge Clk50M or negedge Rst_n)
if(!Rst_n)  //让 PWM 输出信号复位时输出低电平
    o_pwm <= 1'b0;
else if(counter >= counter_ccr)//计数值大于比较值
    o_pwm <= 1'b0;  //输出为 0
else    //计数值小于比较值
    o_pwm <= 1'b1; //输出为 1
```

通过以上设计，一个最简单的 PWM 产生电路主要电路就设计完成了，以下为 PWM 产生电路的完整代码：

```
module pwm_generator(
```

```
    Clk50M,
    Rst_n,
    cnt_en,
    counter_arr,
    counter_ccr,
    o_pwm
);
    input Clk50M;   //50MHz 时钟输入
    input Rst_n;    //复位输入，低电平复位
    input cnt_en;   //计数使能信号
    input [31:0]counter_arr;//输入 32 位预重装值
    input [31:0]counter_ccr;//输入 32 位输出比较值
    output reg o_pwm;   //pwm 输出信号

    reg [31:0]counter;//定义 32 位计数器
    always@(posedge Clk50M or negedge Rst_n)
    if(!Rst_n)
        counter <= 32'd0;
    else if(cnt_en)begin
        if(counter == 0)
            counter <= counter_arr;//计数到 0，加载自动预重装寄存器值
        else
            counter <= counter - 1'b1;//计数器自减 1
    end
    else
        counter <= counter_arr; //没有使能时，计数器值等于预重装寄存器值

    always@(posedge Clk50M or negedge Rst_n)
    if(!Rst_n)  //让 PWM 输出信号复位时输出低电平
        o_pwm <= 1'b0;
    else if(counter >= counter_ccr)//计数值大于比较值
        o_pwm <= 1'b0;  //输出为 0
    else    //计数值小于比较值
        o_pwm <= 1'b1; //输出为 1

endmodule
```

PWM 发生电路仿真验证

对本 PWM 发生电路的验证思路比较简单，只需要产生 50MHz 基准计数时钟源（其他频率也可以，只需要修正频率和占空比计算公式中的相关参数），然后给出预重装值和输出比较值，然后使能计数，即可启动 PWM 输出。在运行过程中，修改预重装值可以设置输出 PWM 信号的频率，并将同时影响输出占空比，而在预重装值确定的情况下，修改输出比较

值，则可以设置输出占空比。

最终输出 PWM 波的频率计算公式为：

$$f_{pwm} = \frac{f_{clk}}{counter\_arr + 1}$$

因此，当输出频率确定时，可计算得到预重装值，计算公式为：

$$counter\_arr = \frac{f_{clk}}{f_{pwm}} - 1$$

例如，当希望设置输出信号频率为 5KHz 时

$$counter\_arr = \frac{f_{clk}}{f_{pwm}} - 1 = \frac{50000000}{5000} - 1 = 9999$$

因此，我们只需要设置 counter_arr 值为 9999 即可使得最终输出信号频率为 5KHz。

当输出 PWM 频率确定后，其输出占空比计算则为输出比较值与预重装值之商。计算公式为：

$$\mathrm{PW} = \frac{counter\_ccr}{counter\_arr}$$

因此，当输出占空比确定时，可计算得到输出比较值，计算公式为：

$$counter\_ccr = \mathrm{PW} \times counter\_acr$$

例如，当输出频率为 5KHz，输出占空比为 70%时

$$counter\_ccr = \mathrm{PW} \times counter\_arr = 9999 * 0.7 = 6999$$

PWM 发生电路 testbench 设计

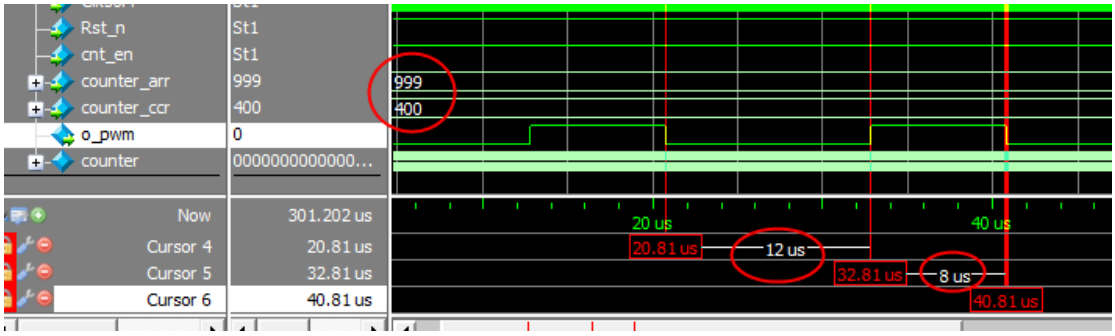
根据上述计算公式，可以设计 pwm_generator 模块的仿真文件如下所示：

```
`timescale 1ns/1ns
`define clk_period 20

module pwm_generator_tb;

    reg Clk50M; //50MHz 时钟输入
    reg Rst_n;  //复位输入，低电平复位
    reg cnt_en; //计数使能信号
    reg [31:0]counter_arr;//输入 32 位预重装值
    reg [31:0]counter_ccr;//输入 32 位输出比较值
    wire o_pwm; //pwm 输出信号

    pwm_generator pwm_generator(
        .Clk50M(Clk50M),
        .Rst_n(Rst_n),
        .cnt_en(cnt_en),
        .counter_arr(counter_arr),
        .counter_ccr(counter_ccr),
        .o_pwm(o_pwm)
    );
```

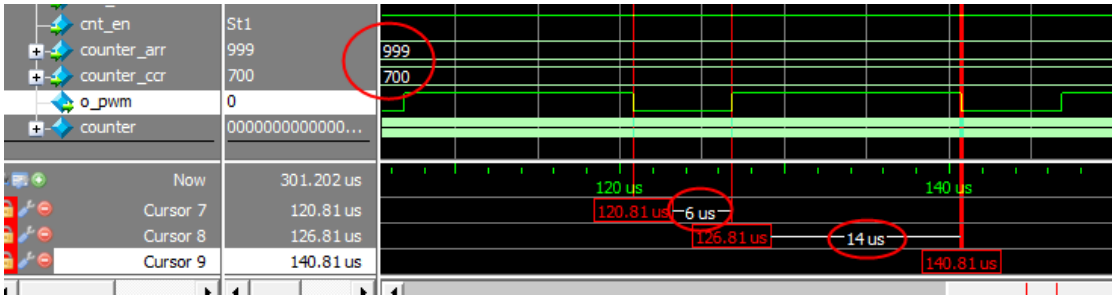
```
    initial Clk50M = 0;
    always #(`clk_period/2) Clk50M = ~Clk50M;

    initial begin
        Rst_n = 0;
        cnt_en = 0;
        counter_arr = 0;
        counter_ccr = 0;
        #(`clk_period*20 +1);
        Rst_n = 1;
        #(`clk_period*10 +1);
        counter_arr = 999;//设置输出信号频率为 50KHz
        counter_ccr = 400;//设置输出 PWM 波占空比为 40%
        #(`clk_period*10);
        cnt_en = 1; //启动计数以产生 PWM 输出
        #100050;
        counter_ccr = 700;//设置输出 PWM 波占空比为 70%
        #100050;
        cnt_en = 0; //停止计数以关闭 PWM 输出
        counter_arr = 499;//设置输出信号频率为 100KHz
        counter_ccr = 250;//设置输出 PWM 波占空比为 50%
        #(`clk_period*10);
        cnt_en = 1; //启动计数以产生 PWM 输出
        #50050;
        counter_ccr = 100;//设置输出 PWM 波占空比为 20%
        #50050;
        $stop;
    end

endmodule
```

仿真结果分析

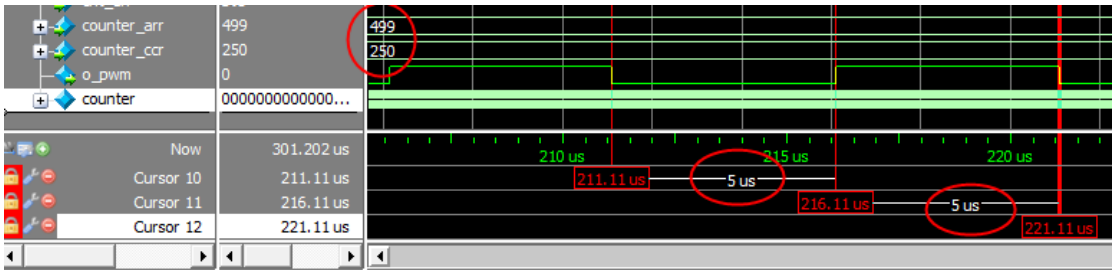
下图为设置输出 PWM 波频率为 50KHz（counter_arr 为 999）、占空比为 40%（counter_ccr 为 400）时的仿真波形，由图可知，低电平周期为 12us，高电平周期为 8us，整个信号周期为 20us，即频率为 50KHz。占空比为 8/20 = 0.4。



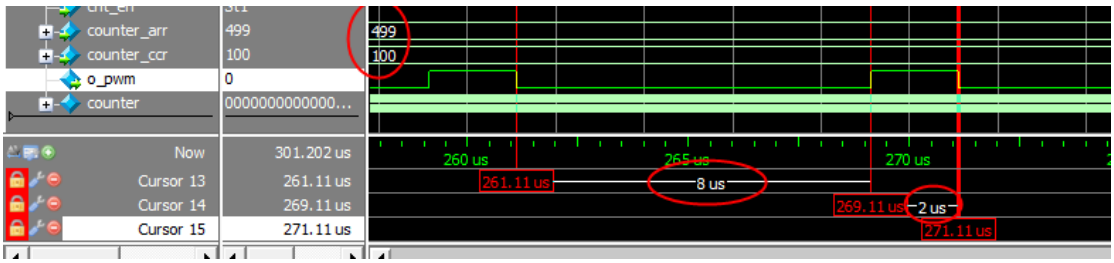
下图为设置输出 PWM 波频率为 50KHz（counter_arr 为 999）、占空比为 70%（counter_ccr 为 700）时的仿真波形，由图可知，低电平周期为 6us，高电平周期为 14us，整个信号周期为 20us，即频率为 50KHz。占空比为 14/20 = 0.7。



下图为设置输出 PWM 波频率为 100KHz（counter_arr 为 499）、占空比为 50%（counter_ccr 为 250）时的仿真波形，由图可知，低电平周期为 5us，高电平周期为 5us，整个信号周期为 10us，即频率为 100KHz。占空比为 5/10 = 0.5。



下图为设置输出 PWM 波频率为 100KHz（counter_arr 为 499）、占空比为 20%（counter_ccr 为 100）时的仿真波形，由图可知，低电平周期为 8us，高电平周期为 2us，整个信号周期为 10us，即频率为 100KHz。占空比为 2/10 = 0.2。



由此可知，该 PWM 生成电路能够正确的产生 PWM 输出信号。

PWM 驱动蜂鸣器板级验证

通过仿真验证，我们确认了该 PWM 发生电路理论设计正确，接下来，我们将使用该 PWM 发生模块来驱动芯航线 FPGA 开发板上的无源蜂鸣器，让无源蜂鸣器能够循环依次发出“哆来咪发梭拉西”的音调。（本想让蜂鸣器能够演奏一曲的，可是无奈本人音乐天赋为负数，学不会谱曲，因此只能把最基本的“哆来咪发梭拉西”放出来了，希望有音乐天赋的朋友能在此基础上谱写演奏出美丽的乐章）。

以下为查资料得知的每个乐调对应的频率。

音名	频率/Hz	音名	频率/Hz	音名	频率/Hz
低音 1	261.6	中音 1	523.3	高音 1	1045.5
低音 2	293.7	中音 2	587.3	高音 2	1174.7
低音 3	329.6	中音 3	659.3	高音 3	1318.5
低音 4	349.2	中音 4	698.5	高音 4	1396.9
低音 5	392	中音 5	784	高音 5	1568
低音 6	440	中音 6	880	高音 6	1760
低音 7	493.9	中音 7	987.8	高音 7	1975.5

根据每个音调的频率值，可以计算得出 PWM 发送模块的预重装值，以下为计算得出的音调频率与对应 PWM 发送模块输出相应频率的预重装值。

频率/Hz	预重装值	频率/Hz	预重装值	频率/Hz	预重装值
261.6	191130	523.3	95546	1045.5	47823
293.7	170241	587.3	85134	1174.7	42563
329.6	151698	659.3	75837	1318.5	37921
349.2	143183	698.5	71581	1396.9	35793
392	127550	784	63775	1568	31887
440	113635	880	56817	1760	28408
493.9	101234	987.8	50617	1975.5	25309

本例中，保持 PWM 波的占空比始终为 50%即可，而通过前面仿真验证可知，占空比为 50%时，输出比较值刚好为预重装值的一半，因此，我们只需要将预重装值除以 2（右移一位）的结果直接赋值给输出比较值即可，这样可以避免再重复计算输出比较值。

另外，为了保证音调的切换能够让我们容易分辨，因此设计一个 500ms 的定时器，每 500ms 切换一次音调。该部分电路非常简单，因此本板级验证部分将不再讲解代码的详细设计思路，只给出具体代码。

音调播放电路的代码如下所示：

```
module pwm_generator_test(
    Clk50M,
    Rst_n,
    beep
);

    input Clk50M;
```

```
    input Rst_n;
    output beep;

    reg [31:0]counter_arr;  //预重装值寄存器
    wire [31:0]counter_ccr; //输出比较值

    reg [24:0]delay_cnt;    //500ms 延时计数器
    reg [4:0]Pitch_num; //音调编号

    localparam
        L1 = 191130, //低音 1
        L2 = 170241, //低音 2
        L3 = 151698, //低音 3
        L4 = 143183, //低音 4
        L5 = 127550, //低音 5
        L6 = 113635, //低音 6
        L7 = 101234, //低音 7

        M1 = 95546, //中音 1
        M2 = 85134, //中音 2
        M3 = 75837, //中音 3
        M4 = 71581, //中音 4
        M5 = 63775, //中音 5
        M6 = 56817, //中音 6
        M7 = 50617, //中音 7

        H1 = 47823, //高音 1
        H2 = 42563, //高音 2
        H3 = 37921, //高音 3
        H4 = 35793, //高音 4
        H5 = 31887, //高音 5
        H6 = 28408, //高音 6
        H7 = 25309; //高音 7

    //输出比较值为预重装值一半
    assign counter_ccr = counter_arr >> 1;

    pwm_generator pwm_generator(
        .Clk50M(Clk50M),
        .Rst_n(Rst_n),
        .cnt_en(1'b1),
        .counter_arr(counter_arr),
        .counter_ccr(counter_ccr),
        .o_pwm(beep)
```

```
);

//500ms 延时计数器计数
always@(posedge Clk50M or negedge Rst_n)
if(!Rst_n)
    delay_cnt <= 25'd0;
else if(delay_cnt == 0)
    delay_cnt <= 25'd24999999;
else
    delay_cnt <= delay_cnt - 1'b1;

//每 500ms 切换一次音调
always@(posedge Clk50M or negedge Rst_n)
if(!Rst_n)
    Pitch_num <= 5'd0;
else if(delay_cnt == 0)begin
    if(Pitch_num == 5'd20)
        Pitch_num <= 5'd0;
    else
        Pitch_num <= Pitch_num + 5'd1;
end
else
    Pitch_num <= Pitch_num;

//根据音调编号给预重装值给相应的值
always@(*)
    case(Pitch_num)
        0 :counter_arr  = L1;
        1 :counter_arr  = L2;
        2 :counter_arr  = L3;
        3 :counter_arr  = L4;
        4 :counter_arr  = L5;
        5 :counter_arr  = L6;
        6 :counter_arr  = L7;
        7 :counter_arr  = M1;
        8 :counter_arr  = M2;
        9 :counter_arr  = M3;
        10:counter_arr  = M4;
        11:counter_arr  = M5;
        12:counter_arr  = M6;
        13:counter_arr  = M7;
        14:counter_arr  = H1;
        15:counter_arr  = H2;
        16:counter_arr  = H3;
```

```
            17:counter_arr  = H4;
            18:counter_arr  = H5;
            19:counter_arr  = H6;
            20:counter_arr  = H7;
            default:counter_arr  = L1;
        endcase

endmodule
```

蜂鸣器音调播放电路的引脚分配如下表所示：

无源蜂鸣器	对应 FPGA 管脚
beep	PIN_N8
Clk50M	PIN_E1
Rst_n	PIN_M1

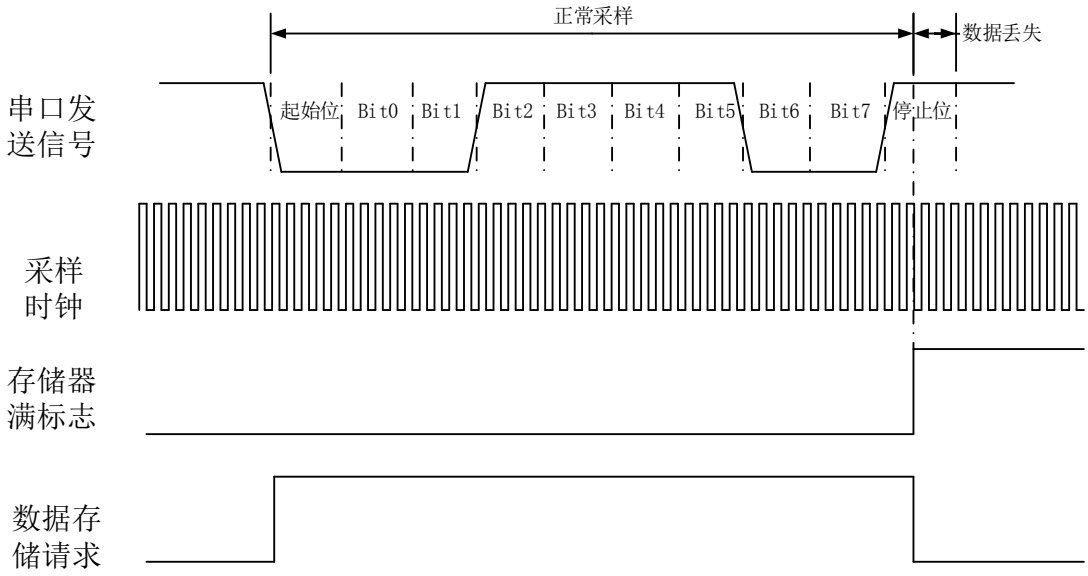
引脚分配完成后对工程全编译，然后下载到芯航线 FPGA 开发板上，下载完成后蜂鸣器即开始循环从低音 1 播放到高音 7。

采样时钟频率控制模块

采样速率与数据完整性

对于逻辑分析仪，数据采样速率是个关键的参数。可能一提到速率，大家就会认为，速度越快越好。然而对于逻辑分析仪，虽然采样速率越快，越能采集到波形的细节，如毛刺等。然而逻辑分析仪对于数据都是先存储后显示的，既然要存储，就涉及到一个存储器大小的问题，对于一个逻辑分析仪，其支持的存储容量一定是有限的，如果采用速度太快，那么在很短的时间内，存储器就会被存满，无法再存储新的数据。但是被采样的信号周期可能比较长，因此就会出现存储器已满，然而被采样信号周期还没结束的情况，从而使抓取到的数据不完整。以下以一个串口传输的具体例子来说明：

假设我们需要对一个串口发送一个完整字节数据的过程进行抓取，串口波特率为 9600bps、1 位起始位、8 位数据位、1 位停止位。那么串口完成一个字节传输所需时间为 1/9600 * 10 ，约为 1.05ms。而对于一个存储深度为 1024 的逻辑分析仪，如果使用 1MHz（周期 1us）的采样率来采样，则当采样被触发后，1024 个采样周期也即 1.024ms 后，存储器就已经存储满了，之后的数据，将被丢弃。因此导致采样数据不完整。



因此，为了保证采样结果的完整性，我们可以适当降低采样速率，例如，当我们将采样时钟从 1MHz 降低为 100KHz 后，从开始采样到最终存储器被存储满，就可以采样 10.24ms，远远大于串口一个字节的传输长度，因此就能保证完整的数据采集。而使用 100KHz 的采样时钟，每两个采样点间的间隔为 10us，而 9600 波特率下，串口传输一个数据位的时间为 104us 左右，即能够保证串口的每一位都能被采样 10 次，这对于分析串口传输波形来说，已经足够了，而之前使用 1MHz 的采样时钟，串口每一位被采样 100 次，这么高的采样密度对于分析一位数据来说，并没有任何实质性的效果提升。因此，合理选择采样率，能够在保证数据完整性的同时，降低对存储器容量的要求。本节，我们就设计一个采样时钟控制模块，通过控制信该模块产生不同频率的采样使能时钟信号，实现控制逻辑分析仪数据采样速率的功能。

采样速率控制模块功能分析

本采样控制模块支持产生 16 种采样速率的采样使能时钟，具体采样使能时钟的频率由一个 4 位宽度的采样速率选择信号“sel”控制。而具体实现不同频率的采样时钟，实质就是通过计数器来实现一个指定周期的循环计数。这一点，与我们讲述 PWM 生产模块的原理完全一致，本节，我们设计的采样率控制模块，实质就是一个分频器，分频器的核心电路是计数器，因此，我们可以首先设计一个计数器电路，由于计数器电路是 FPGA 设计中最最基本的电路，在之前的很多课程中我们都讲过，因此这里不再详解设计过程，只给出计数器部分代码：

```
    //-----------模块输出端口-----------
    output clken;       //时钟使能信号输出

    //------------寄存器定义------------
    reg [19:0]count;    //分频计数器
    reg [19:0]clk_div;  //时钟分频值

    always@(posedge Clk100M or negedge Rst_n)
    if(!Rst_n)
        count<=20'd0;
```

```
    else if(clken)
        count<=20'd0;     //计数清零
    else
        count<=count+1'b1;//计数累加

    assign clken=(count==clk_div);
```

这里，clk_div 即为计数器循环计数周期的设定值，通过设定不同的 clk_div 值，就能产生不同的采样使能时钟 clken。clken 的频率值与 clk_div 的计算关系为：

$$f_{clken} = \frac{f_{clk}}{clk\_div + 1}$$

则

$$clk\_div = \frac{f_{clk}}{f_{clken}} - 1$$

例如，系统时钟为 100MHz，我们希望采样使能时钟 clken 的频率为 100KHz，只需要设置 clk_div 的值为 999 即可。

采样速率选择信号(sel)的值与对应的采样时钟(clken)频率以及对应的分频值(clk_div)关系对应如下表所示：

频率选择信号值（sel）	采样频率	分频计数值(clk_div)
0	100Hz	999999
1	500Hz	199999
2	1KHz	99999
3	5KHz	19999
4	10KHz	9999
5	50KHz	1999
6	100KHz	999
7	200KHz	499
8	500KHz	199
9	1MHz	99
10	2MHz	49
11	5MHz	19
12	10MHz	9
13	20MHz	4
14	50MHz	1
15	100MHz	0

最后，我们只需要根据此表设置相应的分频值即可，该部分代码实质就是一个查找表，根据 sel 的值查找得出 clk_div 的值。代码如下所示：

```
    //-------------通过改变 sel 的值选择不同的时钟频率-------------
    always@(sel)
        begin
            case(sel)
                4'h0:clk_div=20'd999999; //100HZ
                4'h1:clk_div=20'd199999; //500HZ
```

```
                4'h2:clk_div=20'd99999;  //1KHZ
                4'h3:clk_div=20'd19999;  //5KHZ
                4'h4:clk_div=20'd9999;   //10KHZ
                4'h5:clk_div=20'd1999;   //50KHZ
                4'h6:clk_div=20'd999;    //100KHZ
                4'h7:clk_div=20'd499;    //200KHZ
                4'h8:clk_div=20'd199;    //500KHZ
                4'h9:clk_div=20'd99;     //1MHZ
                4'ha:clk_div=20'd49;     //2MHZ
                4'hb:clk_div=20'd19;     //5MHZ
                4'hc:clk_div=20'd9;      //10MHZ
                4'hd:clk_div=20'd4;      //20MHZ
                4'he:clk_div=20'd1;      //50MHZ
                4'hf:clk_div=20'd0;      //100MHZ
            endcase
        end
```

通过以上设计，我们就完成了分频模块的全部功能设计，以下为分频模块的全部代码：

```
module div_freq(
    Clk100M,//输入 100MHZ 时钟
    Rst_n,
    clken,//时钟使能信号输出
    sel   //频率选择
);

    //-----------模块输入端口-----------
    input Clk100M;        //输入 100MHZ 时钟
    input Rst_n;
    input [3:0]sel;     //频率选择

    //-----------模块输出端口-----------
    output clken;       //时钟使能信号输出

    //------------寄存器定义------------
    reg [19:0]count;    //分频计数器
    reg [19:0]clk_div;  //时钟分频值

    //-----------内部连线定义-----------
    wire div_100m;

    //sel==4'hf 时，输出 div_100m 为高电平
    assign div_100m=(sel==4'hf);

    always@(posedge Clk100M or negedge Rst_n)
    if(!Rst_n)
```

```
        count<=20'd0;
    else if(clken)
        count<=20'd0;     //计数清零
    else
        count<=count+1'b1;//计数累加

    //当 div_100m 为高电平，clken 也为高电平
    //否则只有当 count==clk_div 时，clken 为高电平

    assign clken=div_100m?1'b1:(count==clk_div);

    //-------------通过改变 sel 的值选择不同的时钟频率-------------
    always@(sel)
        begin
            case(sel)
                4'h0:clk_div=20'd999999; //100HZ
                4'h1:clk_div=20'd199999; //500HZ
                4'h2:clk_div=20'd99999;  //1KHZ
                4'h3:clk_div=20'd19999;  //5KHZ
                4'h4:clk_div=20'd9999;   //10KHZ
                4'h5:clk_div=20'd1999;   //50KHZ
                4'h6:clk_div=20'd999;    //100KHZ
                4'h7:clk_div=20'd499;    //200KHZ
                4'h8:clk_div=20'd199;    //500KHZ
                4'h9:clk_div=20'd99;     //1MHZ
                4'ha:clk_div=20'd49;     //2MHZ
                4'hb:clk_div=20'd19;     //5MHZ
                4'hc:clk_div=20'd9;      //10MHZ
                4'hd:clk_div=20'd4;      //20MHZ
                4'he:clk_div=20'd1;      //50MHZ
                4'hf:clk_div=20'd0;      //100MHZ
            endcase
        end

endmodule
```

采样时钟分频模块仿真验证

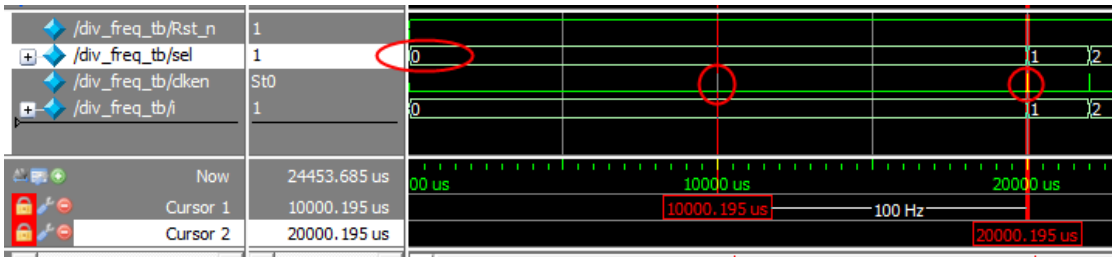
testbench 设计

对采样时钟分频模块的仿真验证思路非常简单，只需要产生系统时钟 clk，然后分别给出不同的 sel 值，然后查看 clken 的输出频率即可，完整 testbench 代码如下所示：

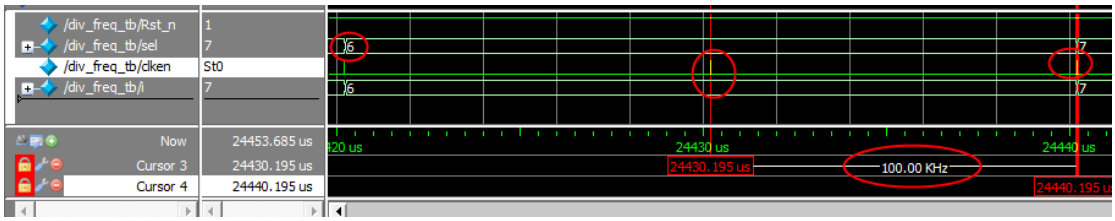
```
`timescale 1ns/1ns
`define clk_period 10

module div_freq_tb;

    reg Clk100M;        //输入 100MHZ 时钟
    reg Rst_n;
    reg [3:0]sel;     //频率选择

    wire clken;       //时钟使能信号输出

    div_freq div_freq(
        .Clk100M(Clk100M),//输入 100MHZ 时钟
        .Rst_n(Rst_n),
        .clken(clken),//时钟使能信号输出
        .sel(sel)   //频率选择
    );

    initial Clk100M = 0;
    always #(`clk_period/2)Clk100M = ~Clk100M;

    integer i;

    initial begin
        Rst_n = 0;
        sel = 0;
        #(`clk_period*20 + 1);
        Rst_n = 1;
        #(`clk_period*20 + 1);
        for(i=0;i<15;i=i+1)begin
        /*每个采样值出现 3 次采样使能时钟高电平后即切换到下一采
         *样频率，以使结果清晰方便观察。*/
            repeat(3)begin
                wait(clken);
                #(`clk_period);
            end
            sel = sel + 1;
        end
        #(`clk_period*200);
        $stop;
    end

endmodule
```

仿真结果验证

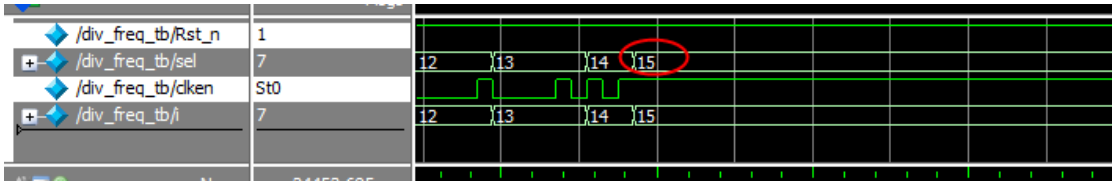
以下为 sel 为 0 时，输出的 clken 信号，频率为 100Hz。



以下为 sel 为 6 时，输出的 clken 信号，频率为 100KHz。



以下为 sel 为 15 时，输出的 clken 信号，可以看到 clken 一直为高电平，因此实际采样的时候该信号一直有效，采样率就等于系统时钟 100MHz。



其他频率的仿真结果，这里就不再贴图分析。如果大家在学习的过程中发现有任何问题，可以重复仿真过程，通过仿真查找问题并找到解决方案。

逻辑分析仪采样控制模块

逻辑分析仪最核心的功能就是对待测信号的采集。既然有数据采集，也就涉及到数据存储。对于本 DEMO，数据最终存储到 FPGA 片上 RAM 中。因此，我们只需要控制好将数据在正确的时间，以正确的方式存储到 RAM 中正确的位置即可。如何确保数据在正确的时间，以正确的方式存储到 RAM 中正确的位置呢，这就是采样控制模块的设计核心。

而如何在正确的时间，来存储数据到 RAM 中呢？这就涉及到一个触发的概念。以下以数字存储示波器的触发原理来进行介绍。

采样触发介绍

什么是触发?

任何示波器的存储器容量都是有限的，因此所有示波器都必须使用触发。触发是示波器

应该发现的用户感兴趣的事件。换句话说，它是用户想要在波形中寻找的东西。触发可以是一个事件(即波形中的问题)，但不是所有的触发都是事件。触发实例包括边沿触发、毛刺信号触发和数字码型触发。

示波器必须使用触发的原因在于其存储器的容量有限。例如，Agilent 90000 系列示波器具有 20 亿采样的存储器深度。但是，即便拥有如此大容量的存储器，示波器仍需要一些事件来区分哪 20 亿个采样需要显示给用户。尽管 20 亿的采样听起来似乎非常庞大，但这仍不足以确保示波器存储器能够捕获到感兴趣的事件。

示波器的存储器可视为一个传送带。无论什么时候进行新的采样，采样都会存储到存储器中。存储器存满时，最旧的采样就会被删除，以便保存最新采样。当触发事件发生时，示波器就会捕获足够的采样，以将触发事件存储在存储器要求的位置(通常是在中间)，然后将这些数据显示给用户。

逻辑分析仪触发方式

以上为示波器的触发理论原理介绍，而对于逻辑分析仪，由于采集的是数字信号，因此触发类型和示波器稍微有些区别。逻辑分析仪中常见的触发方式包括延迟触发、限定触发、组合触发、毛刺触发等。而本例作为学习型示例，主要目标为讲述一个完整数字系统的设计过程，重点不在于实现商用逻辑分析仪的完整功能，因此可以简化触发方式，本逻辑分析仪主要支持以下几种触发方式：

电平触发：低电平触发、高电平触发

边沿触发：上升沿触发、下降沿触发、边沿触发（上升沿或下降沿触发）

立即触发：实时触发

触发电路设计

触发条件检测

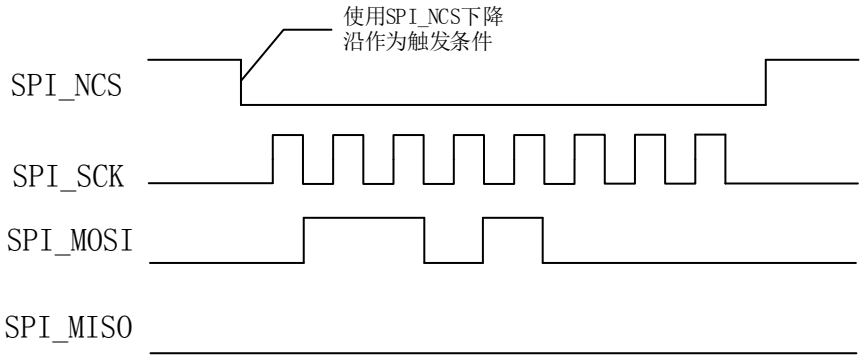
整个逻辑分析仪的触发模型就是一组被采样信号输入，根据被采样信号是否满足触发条件，如果被采样信号满足当前指定的触发条件，则开始存储被采样到的数据到 RAM 中。当 RAM 数据存储慢后，停止存储。等待再次满足触发条件，然后重新开始存储。

因此，要想实现正确的数据采样，首先我们需要能够正确的识别触发条件，电平识别非常简单，而边沿识别则相对复杂些。关于边沿识别的具体原理，在《小梅哥 FPGA 设计思想与验证方法视频教程》中，“09_A 按键消抖模块设计与验证”一课中有过介绍，关于边沿检测的具体原理，可参见该视频以及配套的视频精讲手册文档。以下为对输入的带采样数据进行边沿识别的代码：

```
reg [9:0]data_r1;        //输入信号同步寄存器
reg [9:0]data_r2;
wire [9:0] s_posedge;    //上升沿标志
wire [9:0] s_negedge;    //下降沿标志
```

```
    wire [9:0] s_edge;       //边沿标志

    //-----------------同步输入信号----------------
    always@(posedge Clk)
    if(clken) begin
        data_r1 <= data_in;//一级寄存器赋值
        data_r2 <= data_r1;//二级寄存器赋值
    end

    assign s_posedge = data_r1 & ~data_r2;//上升沿检测
    assign s_negedge = ~data_r1 & data_r2;//下降沿检测
    assign s_edge = data_r1 ^ data_r2;   //边沿检测
```

本系统逻辑分析仪总共有 10 个输入通道，对于逻辑分析仪的触发，除了触发条件外，另一个需要关心的就是触发通道。因为我们在使用逻辑分析仪的时候，往往需要根据一组信号中的某一个信号的关键状态进行数据采样，例如，在采集 SPI 通信中，我们往往使用 SPI 接口的 CS 信号的下降沿作为触发条件。而 SPI 信号接在哪个输入通道上，我们就需要设置使用这个通道作为触发通道。所以，我们在最终判断是否需要存储采样数据时，实际需要判断的是设定触发通道是否满足触发条件。



因此，我们需要对每个通道都设置一个触发条件判断标志，trigger_dat[9:0]。每一位存储每一个通道的触发状态（是否满足当前触发条件）。而具体触发条件判断标志应该实时保存哪一种状态，则由触发模式控制信号（mode_sel）进行控制。具体模式控制信号与触发模式的对应关系如下表所示：

mode_sel	0	1	2	3	4	5
触发条件	低电平	高电平	上升沿	下降沿	双边沿	立即触发

据此，我们就可以得出每一个通道触发条件判断标志信号的产生代码了：

```
    reg [9:0]trigger_dat;    //触发条件比较数据
    input  [2:0]mode_sel;    //模式选择
    //-------根据不同的触发模式选择触发数据（trigger_dat）------------
    always@(mode_sel or s_posedge or s_negedge or s_edge or
data_r2)
    begin
        case(mode_sel)
            3'd0:trigger_dat=~data_r2;  //低电平触发
```

```
            3'd1:trigger_dat=data_r2;   //高电平触发
            3'd2:trigger_dat=s_posedge; //上升沿触发
            3'd3:trigger_dat=s_negedge; //下降沿触发
            3'd4:trigger_dat=s_edge;    //边沿触发
            3'd5:trigger_dat=10'h3ff;   //立即触发
          default:trigger_dat=10'h3ff;
        endcase
    end
```

有了每个通道的触发状态，我们就可以根据设定的触发通道的触发条件是否有效来启动采样存储。所谓启动采样存储，即使能将输入数据写入到 RAM 中去。而具体写入 RAM 中的数据速度，则由采样时钟控制模块输出的采样使能信号（clken）控制。

首先，我们检测设置的通道是否满足触发条件，该部分代码如下所示：

```
    input  [3:0]channel_sel; //通道选择
    input  clken;            //单点采样使能，即采样时钟信号

    //-----------------检测是否满足触发条件-----------------
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        trigger<=0;
    //触发通道对于触发条件为真、且当前正在采样时钟节点
    else if(trigger_dat[channel_sel] && clken)
        trigger<=1; //标记触发有效信号为真
    else
        trigger<=0;
```

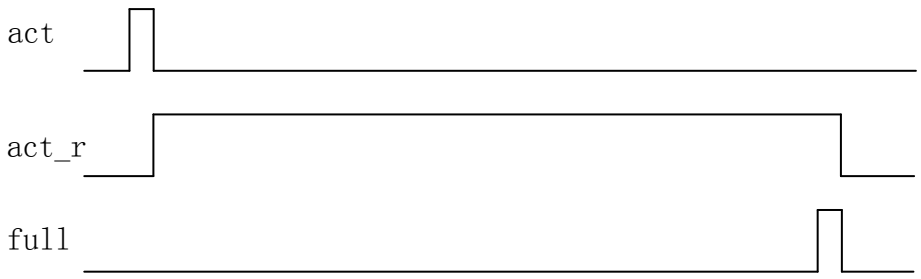
采样存储控制

指定通道的触发已经满足了，那么是否就可以马上进行采样了呢？未必，因为我们还需要一个最关键的采样控制——是否开始触发采样。什么意思呢？我们想象这样一个使用场景，我们使用该逻辑分析仪采样 SPI 通信的过程，我们的逻辑分析仪已经设置好了触发条件（假设为下降沿触发）、触发通道（通道 0），那么此时，一旦输入 0 通道出现下降沿，我们是不是就应该立即开始采样并存储数据呢？不一定，因为可能 SPI 整个通信过程中，我们只关心某一次传输的波形，而在这之前，可能有多次传输我们并不关心，例如，某一个数字系统带有 SPI 主机接口和两个按键，我们按下按键 1，该系统使用 SPI 对外传输系统的温度数据，按下按键 2，传输系统的湿度数据。我们希望通过逻辑分析仪抓取传输湿度数据（按下按键 2）的波形，因此，我们需要保证在系统在传输温度数据（按键 1）的时候，逻辑分析仪处于停止采样状态，这样，即使待采样通道满足了触发条件，因为我们没有使能采样，因此该触发信号并不启动采样，只有当我们在按下按键 2 之前，先打开采样使能，使逻辑分析仪处于就绪状态，然后按下按键 2，系统 SPI 接口传输湿度数据，该数据触发逻辑分析仪开始采样，这样采样结果才是我们需要的内容。这个采样控制信号也就是一个采样总开关的功能，只有当这个总开关打开了，一切触发条件才能够生效，否则系统并不执行采样。

以下为该总开关控制采样过程的代码：

```
input  act;              //启动触发
reg act_r;               //开始采集
wire full;               //写 RAM 满标志

//-----------------保持单次触发状态，直到数据完毕----------------
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    act_r <= 1'b0;
else if(act)    //使能采样开关有效
    act_r <= 1'b1;  //采集
else if(full)   //RAM 写满，停止采集
    act_r <= 1'b0;
else
    act_r <= act_r;
```

这里，act_r 作为一个采样控制信号，当每当检测到采样启动信号时，置 1，即使能采样，当 RAM 写满后，则置低，即停止采集。这样，我们就可以通过 act 这个采样使能信号，和 ram 写满信号 full 来确定采样区间。



接下来我们就可以使用上述得到的采样触发信号和采样使能信号来控制采样存储过程了。所谓控制存储，一个关键操作就是产生 RAM 的写使能信号和写地址。写使能信号在使能采样状态下，满足触发条件开始有效，直到 RAM 写满，然后被清零。写地址则是在写使能有效的情况下，每个采样时钟累加一次。如下为采样控制部分代码：

```
reg wren;                //写 RAM
reg act_r;               //开始采集
wire full;               //写 RAM 满标志

//-----------------检查写 RAM 条件是否满足-----------------
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    wren <= 1'b0;
else if(act_r && trigger && ~wren)//触发有效、已经使能采样
    wren <= 1'b1;   //RAM 写满，开始写数据到 RAM
else if(full)   //RAM 写满，停止写 RAM
    wren <= 1'b0;

reg [9:0]wr_addr;
```

```
    //----------------产生写 RAM 地址----------------
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        wr_addr <= 10'd0;
    else if(clken && wren)
        wr_addr <= wr_addr+1'b1;
    else if(~wren)
        wr_addr <= 10'd0;

    assign full = (wr_addr==10'h3ff); //写 RAM 满标志
```

完整采样控制模块设计

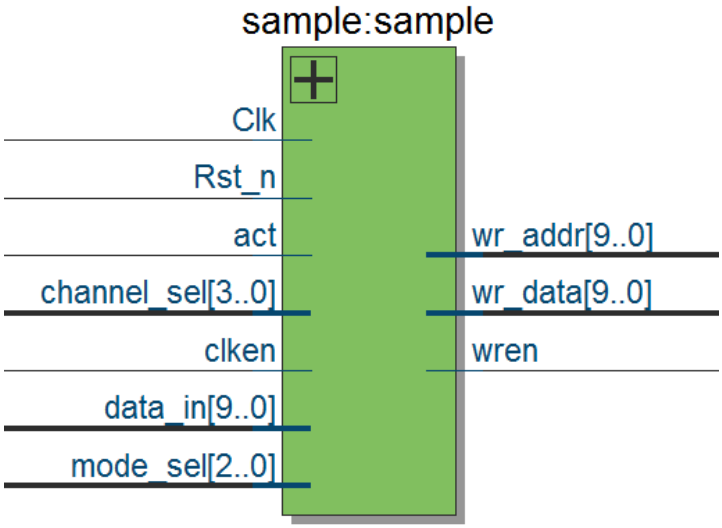
通过以上分析，我们就完成了采样控制模块的设计，该模块最终将 RAM 的写请求、写地址和写数据输出以实现对双口 RAM 进行写入数据。该模块完整代码如下所示：

```
module sample(
    Clk,        //系统时钟 100MHZ
    Rst_n,
    clken,      //采样时钟使能
    act,        //启动触发
    channel_sel,//触发通道选择
    mode_sel,   //触发模式选择
    data_in,    //被测信号输入
    wr_addr,    //写 RAM 地址
    wr_data,    //写 RAM 数据
    wren        //写使能
);

    //----------------模块输入端口----------------
    input  Clk;           //系统时钟 100MHZ
    input  Rst_n;           //
    input  clken;           //时钟使能
    input  act;             //启动触发
    input  [3:0]channel_sel; //通道选择
    input  [2:0]mode_sel;    //模式选择
    input  [9:0]data_in;     //被测信号输入

    //----------------模块输出端口----------------
    output [9:0]wr_addr;     //写 RAM 地址
    output [9:0]wr_data;     //写 RAM 数据
    output wren;            //写使能
```

```
    //----------------I/O 寄存器----------------
    reg [9:0]data_r1;        //输入信号同步寄存器
    reg [9:0]data_r2;
    reg [9:0]wr_addr;

    //----------------内部寄存器----------------
    reg trigger;             //触发标志
    reg wren;                //写 RAM
    reg act_r;               //开始采集
    reg [9:0]trigger_dat;    //触发条件比较数据

    //----------------内部连线----------------
    wire full;               //写 RAM 满标志
    wire [9:0] s_posedge;    //上升沿标志
    wire [9:0] s_negedge;    //下降沿标志
    wire [9:0] s_edge;       //边沿标志

    assign wr_data = data_r2;  //采集数据输出至 RAM

    //----------------同步输入信号----------------
    always@(posedge Clk)
    if(clken) begin
        data_r1<=data_in;//一级寄存器赋值
        data_r2<=data_r1;//二级寄存器赋值
    end

    assign s_posedge=data_r1&~data_r2;//上升沿检测
    assign s_negedge=~data_r1&data_r2;//下降沿检测
    assign s_edge=data_r1^data_r2;   //边沿检测

    //----------------根据不同的触发模式选择触发数据（trigger_dat）------
----------
    always@(mode_sel or s_posedge or s_negedge or s_edge or
data_r2)
        begin
            case(mode_sel)
                3'd0:trigger_dat=~data_r2;  //低电平触发
                3'd1:trigger_dat=data_r2;   //高电平触发
                3'd2:trigger_dat=s_posedge; //上升沿触发
                3'd3:trigger_dat=s_negedge; //下降沿触发
                3'd4:trigger_dat=s_edge;    //边沿触发
                3'd5:trigger_dat=10'h3ff;   //边沿触发
              default:trigger_dat=10'h3ff;
            endcase
```

```
        end

    //-----------------检测是否满足触发条件-----------------
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        trigger<=0;
    else if(trigger_dat[channel_sel]&&clken)
        trigger<=1;
    else
        trigger<=0;

    //-----------------保持单次触发状态，直到数据完毕-----------------
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        act_r <= 1'b0;
    else if(act)
        act_r <= 1'b1;
    else if(full)
        act_r <= 1'b0;
    else
        act_r <= act_r;

    //-----------------产生写 RAM 地址-----------------
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        wr_addr <= 10'd0;
    else if(clken && wren)
        wr_addr <= wr_addr+1'b1;
    else if(~wren)
        wr_addr <= 10'd0;

    assign full = (wr_addr==10'h3ff); //写 RAM 满标志

    //-----------------检查写 RAM 条件是否满足-----------------
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        wren <= 1'b0;
    else if(act_r && trigger && ~wren)
        wren <= 1'b1;
    else if(full)
        wren <= 1'b0;

endmodule
```

下图为该模块在 Quartus II 软件中综合处的模块符号图



各端口功能归纳如下：

端口名	端口功能
Clk	系统时钟 100MHZ
Rst_n	复位输入，低电平复位
act	采样使能开关信号，一个时钟周期长度的高脉冲即可使能采样
clken	采样使能时钟，采样模块在该时钟的高脉冲期间进行采样
data_in	待采样数据输入通道
channel_sel	触发通道选择端口
mode_sel	触发模式选择端口
wr_data	写 RAM 数据端口，采样到的有效数据通过此端口写入 RAM 中
wr_addr	写 RAM 地址端口，采样到的有效数据写 RAM 时地址
wr_en	写 RAM 请求信号，高电平对 RAM 执行写入操作

采样控制模块仿真验证

以上，我们完成了采样控制模块的设计，那么该模块是否能够满足设计功能呢？我们可以通过仿真来验证该设计。

testbench 设计

验证思路较为清晰，首先需要产生一个采样使能时钟信号 clken，该信号直接决定了对待采样信号的采样速率，这里我们设计采样时钟为 10MHz。模块工作时钟为 100MHz。然后就可以通过给 mode_sel 和 channel_sel 设置相应值来确定触发条件和触发通道。接着产生采样使能信号 act（单脉冲），即可启动采样。

另外，我们还需要产生待采样信号，这里，我们就模拟单次 SPI 传输，并将 SPI 传输的这四个信号接到采样模块的通道 0~通道 3。产生 SPI 单次传输信号可以使用一个任务（task）来实现。该 task 代码设计如下：

```
task SPI_TRANS;
    input [7:0]data_trans;  //待传输数据
    reg [7:0]data_tmp;  //传输数据
    begin
        data_tmp = data_trans;
        SPI_NCS = 1;
        #(`sample_clk_period *20);
        SPI_NCS = 0;
        repeat(8)begin
            SPI_SCK = 1;
            #(`sample_clk_period *10);
            SPI_SCK = 0;
            SPI_MOSI = data_tmp[7];
            #1;
            data_tmp[7] = data_tmp[7] << 1;
            #(`sample_clk_period *10 - 1);
        end

        #(`sample_clk_period *20);
        SPI_NCS = 1;
    end
endtask
```

单次采样的控制代码如下所示：

```
mode_sel = 1;   //设置触发模式为高电平触发
#(`clk_period*20);
//启动第 2 次采样
act = 1;    //产生采样使能脉冲
#(`clk_period);
act = 0;
#(`clk_period *10);
SPI_TRANS(8'h13);

wait(wr_addr == 1023);//等待 RAM 被写满
#(`clk_period*200);
```

我们只需要分别设置 model_sel 的数据，然后重复整个过程，就能够测试不通触发模式下的触发采样过程。

以下为 testbench 的完整代码

```
`timescale 1ns/1ns
`define sample_clk_period 100
`define clk_period 10

module sample_tb;
```

```
    //-----------------模块输入端口-----------------
    reg  Clk_100M;           //系统时钟 100MHZ
    reg  Rst_n;             //
    reg  clken;             //时钟使能
    reg  act;               //启动触发
    reg  [3:0]channel_sel; //通道选择
    reg  [2:0]mode_sel;    //模式选择
    wire  [9:0]data_in;     //被测信号输入

    //-----------------模块输出端口-----------------
    wire [9:0]wr_addr;     //写 RAM 地址
    wire [9:0]wr_data;     //写 RAM 数据
    wire wren;             //写使能


    reg SPI_NCS;    //模拟 SPI 接口 NCS 信号
    reg SPI_SCK;    //模拟 SPI 接口 SCK 信号
    reg SPI_MISO;   //模拟 SPI 接口 MISO 信号
    reg SPI_MOSI;   //模拟 SPI 接口 MOSI 信号

    assign data_in = {6'd0,SPI_MOSI,SPI_MISO,SPI_SCK,SPI_NCS};

    sample sample(
        .Clk(Clk_100M),
        .Rst_n(Rst_n),
        .clken(clken),
        .act(act),
        .channel_sel(channel_sel),
        .mode_sel(mode_sel),
        .data_in(data_in),
        .wr_addr(wr_addr),
        .wr_data(wr_data),
        .wren(wren)
    );

     initial Clk_100M = 0;
     always #(`clk_period/2)Clk_100M = ~Clk_100M;

     initial clken = 0;
     always begin
        clken = 0;
        #(`sample_clk_period - `clk_period);//clken 保持 9 个时钟周期的低
电平
```

```
        clken = 1;
        #(`clk_period);//clken 保持 1 个时钟周期的高电平
    end

    initial begin
        Rst_n = 0;
        act = 0;
        channel_sel = 0;
        mode_sel = 0;
        #(`clk_period*20 + 1);
        Rst_n = 1;
        #(`clk_period*20 + 1);
        channel_sel = 0;    //设置触发通道为通道 0
        mode_sel = 0;   //设置触发模式为低电平触发
        #(`clk_period*20);
//------------------------------------------------
        //启动第 1 次采样
        act = 1;    //产生采样使能脉冲
        #(`clk_period);
        act = 0;
        #(`clk_period *10);
        SPI_TRANS(8'h13);

        wait(wr_addr == 1023);//等待 RAM 被写满
        #(`clk_period*200);

//------------------------------------------------
        mode_sel = 1;   //设置触发模式为高电平触发
        #(`clk_period*20);
        //启动第 2 次采样
        act = 1;    //产生采样使能脉冲
        #(`clk_period);
        act = 0;
        #(`clk_period *10);
        SPI_TRANS(8'h13);


        wait(wr_addr == 1023);//等待 RAM 被写满
        #(`clk_period*200);

//------------------------------------------------
        mode_sel = 2;   //设置触发模式上升沿触发
        #(`clk_period*20);
        //启动第 3 次采样
```

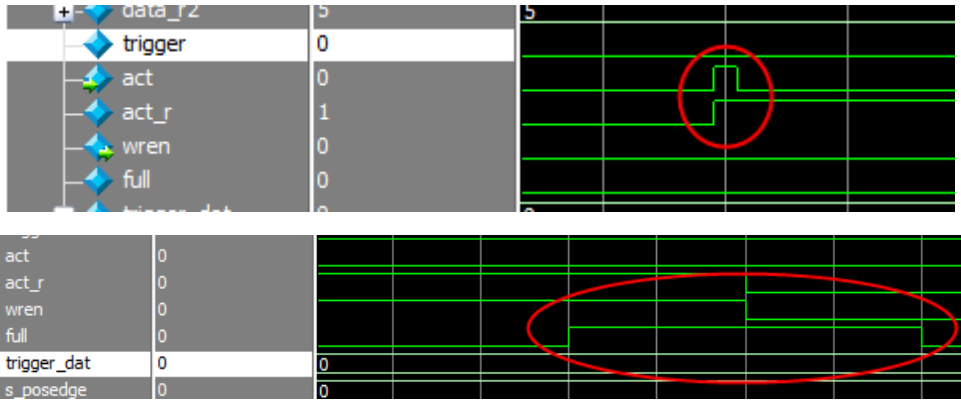
```
        act = 1;    //产生采样使能脉冲
        #(`clk_period);
        act = 0;
        #(`clk_period *10);
        SPI_TRANS(8'h13);

        wait(wr_addr == 1023);//等待 RAM 被写满
        #(`clk_period*200);

//-----------------------------------------------
        mode_sel = 3;   //设置触发模式下降沿触发
        #(`clk_period*20);
        //启动第 4 次采样
        act = 1;    //产生采样使能脉冲
        #(`clk_period);
        act = 0;
        #(`clk_period *10);
        SPI_TRANS(8'h13);

        wait(wr_addr == 1023);//等待 RAM 被写满
        #(`clk_period*200);

//-----------------------------------------------
        mode_sel = 4;   //设置触发模式双边沿触发
        #(`clk_period*20);
        //启动第 5 次采样
        act = 1;    //产生采样使能脉冲
        #(`clk_period);
        act = 0;
        #(`clk_period *10);
        SPI_TRANS(8'h13);

        wait(wr_addr == 1023);//等待 RAM 被写满
        #(`clk_period*200);

//-----------------------------------------------
        mode_sel = 5;   //设置触发模式立即触发
        #(`clk_period*20);
        //启动第 6 次采样
        act = 1;    //产生采样使能脉冲
        #(`clk_period);
        act = 0;
        #(`clk_period *10);
        SPI_TRANS(8'h13);
```

```
        wait(wr_addr == 1023);//等待 RAM 被写满
        #(`clk_period*200);

        $stop;
    end

    initial begin
        SPI_NCS = 1;
        SPI_SCK = 0;
        SPI_MISO = 1;
        SPI_MOSI = 1;
    end

    task SPI_TRANS;
        input [7:0]data_trans;  //待传输数据
        reg [7:0]data_tmp;  //传输数据
        begin
            data_tmp = data_trans;
            SPI_NCS = 1;
            #(`sample_clk_period *20);
            SPI_NCS = 0;
            repeat(8)begin
                SPI_SCK = 1;
                #(`sample_clk_period *10);
                SPI_SCK = 0;
                SPI_MOSI = data_tmp[7];
                #1;
                data_tmp[7] = data_tmp[7] << 1;
                #(`sample_clk_period *10 - 1);
            end

            #(`sample_clk_period *20);
            SPI_NCS = 1;
        end
    endtask

endmodule
```

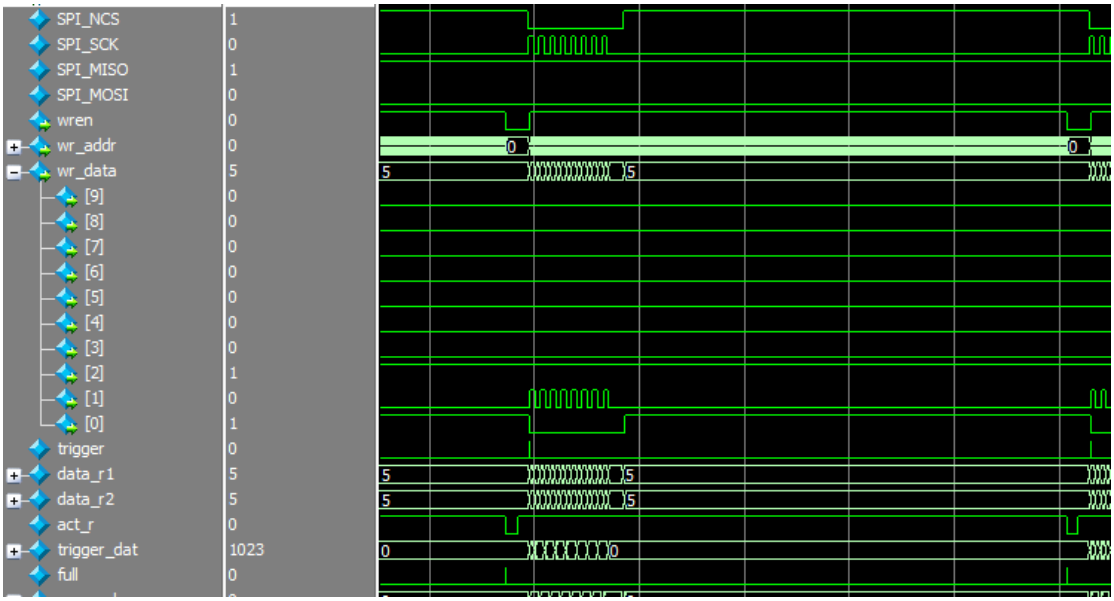
仿真结果分析

以下为采样使能信号 act 与开始采样信号 act_r 的关系，可以看到，当 act 出现高脉冲 ，

则 act_r 变为高电平，直到 full 信号出现高脉冲，act_r 才变为低电平。



以下为使用 SPI_NCS 下降沿信号触发采样的 RAM 写请求和写地址的信号变化波形，从波形可知，采样模块能够在 SPI_NCS 的下降沿时启动写 RAM，并开始控制地址增加，直到最后 RAM 写满后，才停止写 RAM。即整个模块设计正确无误。



波形显示控制

采集的波形数据存储在 FPGA 内部定义的双口 RAM 中，存储深度为 1024 位，数据宽度为 10 位，这样数据的每一位就会英语每一个通道的数据波形，采样到高电平就存 1，采样到低电平就存 0。例如其中一个波形数据为 10 位的 10_0011_1110，则表示第 1~5、9 通道采样为高电平，第 0、6~8 通道采样为低电平。

波形图像显示

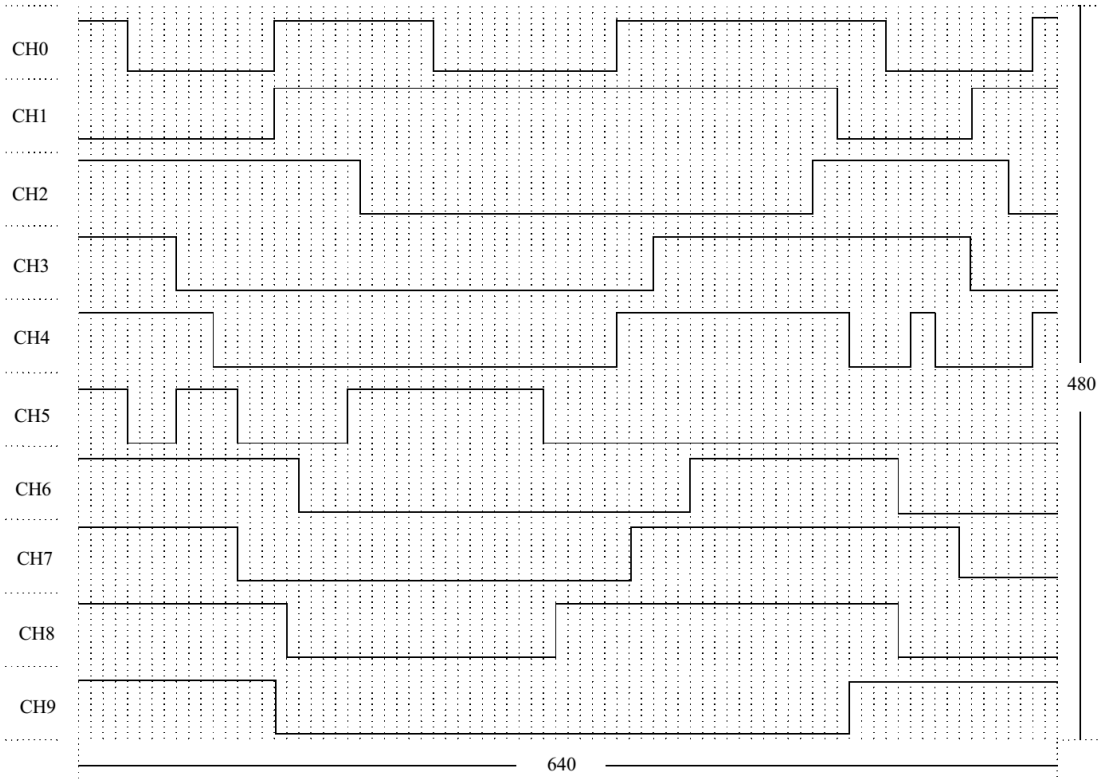
显示区域划分

本设计的 VGA 驱动程序驱动 VGA 显示器时显示分辨率为 640*480，因为总共有 10 个通道，所以垂直部分分为 10 行，每行 48 个像素点高度，波形显示的范围为 32 像素点，剩上下各 8 个像素点作为各个通道的隔离部分。各通道的显示区域范围如表 5.4 所列。

在水平部分显示栅格网络（栅格线用虚线显示），总共分 16 大格（列），每大格在细分 5 小格，每小格由 8 个像素点组成，总共就有 640 个像素点，这样显示器的显示区域就划分完毕。波形显示格式如图 5.12 所示，其中屏幕的背景颜色为浅蓝色，栅格线为黑色，波形

为绿色，时间标线为红色。

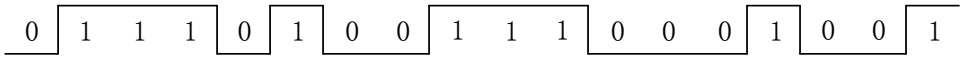
通道	CH0	CH1	CH2	CH3	CH4
范围	8~40	56~88	104~136	152~184	200~232
通道	CH5	CH6	CH7	CH8	CH9
范围	248~280	296~328	344~376	392~424	440~472



波形显示

在 VGA 显示驱动模块中有两个计数器，行扫描计数器 hcount 和场扫描计数器 vcount，经修正后从 VGA 模块输出，其输出显示有效范围分别对应与显示器的水平像素点 0~639 和垂直像素点 0~479，假设有如下表所列的一组波形数据，其对应的波形图如下图所示。现以 CH0 通道（D0）数据分析其显示原理。

地址		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…
波形数据	D0	0	1	1	1	0	1	0	0	1	1	1	0	0	0	1	0	0	1	…
	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
	D9	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	…

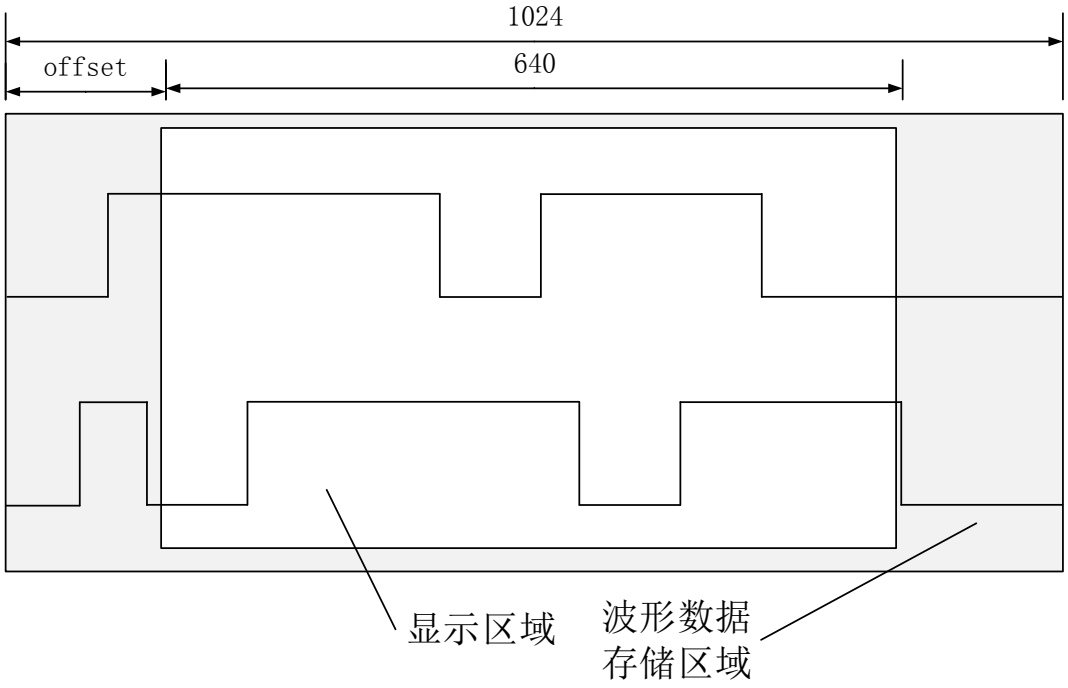


行扫描计数器和双口 RAM 的读取端地址是同步的，即每扫描一个像素点，双口 RAM 地址也加 1，假设 RAM 中的数据不变，则显示区域每列读出的数据都是相同的，现在要做的是在各个通道对应的位置显示对应通道的波形数据。当场扫描计数器 count 的值为 8 时，判

断 D0 值，所为“1”输出数据（指显示波形的颜色数据）显示，若为“0”则输出背景色或栅格（视栅格显示条件而定）；同理当 vcount 的值为 40 时，再次判断 D0 值，若为“0”输出数据，若为“1”则输出背景色或栅格。这样一来，就有高低电平的波形线显示出来了。但高电平与低电平跳变时的垂直线怎么显示呢？我们可以在 RAM 数据读取出来之后再加一级寄存器做个边沿检测电路，当 Din0（D0）的值产生变化时，在 mark 端即输出高电平。当 vcount 的值为 8~40 时判断 mark 的状态，若为 “1”则输出数据（波形颜色数据），若为 0 则显示背景颜色或栅格线。这样一来完整的波形就能显示出来。其他各通道的波形显示的方法相同。

栅格和时间标线的显示比较简单，只要在显示区域对应的位置（比如当 hcount 为 0、8、40 时）输出栅格颜色数据就可以了，若要显示虚线，则需要 vcount 值也做比较（比方当 vcount 为 1、3、5…时显示，vcount 为 0、2、4…不显示）。

移动显示区域，考虑到显示器的水平分辨率为 640 个像素，只能显示 640 个波形点，若波形数据存储器存储深度为 640，则比较浪费存储器资源（与 FPGA 存储器结构相关），所以一般存储器深度取 2^n，例如，取采样深度为 1024 则刚好满足 1K。这样一来。一次完整的采样是 1024 个点，但是屏幕只能显示 640 个点，因此会有一些波形数据显示不出来。这个问题我们可以通过改变双口 RAM 读取的起始位置来达到能查看所有波形数据的目的。下图，图中 offset 为起始地址的偏移量，范围为 0~384，offset 的值可以通过键盘改变。



显示控制部分代码如下所示：

```
module disp_controller(
    VGA_clk, //25MHZ 时钟
    disp_ena,//波形显示使能
    offset,  //显示区域偏移
    marker,  //显示垂直基准线
    vcount,  //VGA 行扫描计数器
    hcount,  //VGA 场扫描计数器
    rd_clk,  //波形存储器读时钟
    rd_data, //波形存储器读数据
    rd_addr, //波形存储器读地址
```

```
    disp_data//显示数据输出
);

    //------------模块输入端口---------------
    input VGA_clk;         //25MHZ 时钟
    input disp_ena;        //波形显示使能
    input [9:0]offset;     //显示区域偏移
    input [9:0]marker;     //显示垂直基准线
    input [9:0]vcount;     //VGA 行扫描计数器
    input [9:0]hcount;     //VGA 场扫描计数器
    input [9:0]rd_data;    //波形存储器读数据
    //------------模块输出端口---------------
    output rd_clk;         //波形存储器读时钟
    output [9:0]rd_addr;   //波形存储器读地址
    output [7:0]disp_data; //显示数据输出

    //-------------用户寄存器----------------
    reg [9:0] rd_data_r;
    reg [9:0] rd_addr_r;
    reg [7:0] disp_data_r;

    //------------内部连接信号线------------
    wire grid;                 //显示栅格标志
    wire mark;                 //显示垂直基准线标志
    wire [9:0]wave;            //显示波形标志
    wire [9:0]wave_edge;       //显示波形边沿标志

    //------------参数定义---------------
    parameter MarkColor=8'he0; //时间标志线颜色 红
    parameter WaveColor=8'h1c; //波形颜色 绿
    parameter GridColor=8'h2f; //栅格颜色 浅蓝
    parameter BackColor=8'h00; //背景颜色 黑

    //定义 10 个通道的波形显示范围
    //总的显示范围是 0-479
    parameter  CH0_H=9'd8,   CH0_L=9'd40,
               CH1_H=9'd56,  CH1_L=9'd88,
               CH2_H=9'd104, CH2_L=9'd136,
               CH3_H=9'd152, CH3_L=9'd184,
               CH4_H=9'd200, CH4_L=9'd232,
               CH5_H=9'd248, CH5_L=9'd280,
               CH6_H=9'd296, CH6_L=9'd328,
               CH7_H=9'd344, CH7_L=9'd376,
               CH8_H=9'd392, CH8_L=9'd424,
```

```
          CH9_H=9'd440, CH9_L=9'd472;

assign rd_clk=VGA_clk;              //双口 RAM 时钟
assign rd_addr=offset+hcount+1'b1;  //双口 RAM 地址
assign disp_data=disp_data_r;       //待显示数据

//------------------时间基准线显示计算------------------
assign mark=(hcount==marker)&&(vcount[1:0]!=2'd1);

//------------------波形竖线显示计算------------------
always@(posedge VGA_clk)
    rd_data_r<=rd_data;

assign wave_edge=rd_data^rd_data_r;//检测波形边沿

//------------------波形显示计算，共 10 个通道------------------
assign wave[0]=((vcount==CH0_H)&&rd_data_r[0])
           ||((vcount==CH0_L)&&~rd_data_r[0])
           ||((vcount>=CH0_H)&&(vcount<=CH0_L)&&wave_edge[0]);

assign wave[1]=((vcount==CH1_H)&&rd_data_r[1])
           ||((vcount==CH1_L)&&~rd_data_r[1])
           ||((vcount>=CH1_H)&&(vcount<=CH1_L)&&wave_edge[1]);

assign wave[2]=((vcount==CH2_H)&&rd_data_r[2])
           ||((vcount==CH2_L)&&~rd_data_r[2])
           ||((vcount>=CH2_H)&&(vcount<=CH2_L)&&wave_edge[2]);

assign wave[3]=((vcount==CH3_H)&&rd_data_r[3])
           ||((vcount==CH3_L)&&~rd_data_r[3])
           ||((vcount>=CH3_H)&&(vcount<=CH3_L)&&wave_edge[3]);

assign wave[4]=((vcount==CH4_H)&&rd_data_r[4])
           ||((vcount==CH4_L)&&~rd_data_r[4])
           ||((vcount>=CH4_H)&&(vcount<=CH4_L)&&wave_edge[4]);

assign wave[5]=((vcount==CH5_H)&&rd_data_r[5])
           ||((vcount==CH5_L)&&~rd_data_r[5])
           ||((vcount>=CH5_H)&&(vcount<=CH5_L)&&wave_edge[5]);

assign wave[6]=((vcount==CH6_H)&&rd_data_r[6])
           ||((vcount==CH6_L)&&~rd_data_r[6])
           ||((vcount>=CH6_H)&&(vcount<=CH6_L)&&wave_edge[6]);
```

```
assign wave[7]=((vcount==CH7_H)&&rd_data_r[7])
          |||((vcount==CH7_L)&&~rd_data_r[7])
          |||((vcount>=CH7_H)&&(vcount<=CH7_L)&&wave_edge[7]);

assign wave[8]=((vcount==CH8_H)&&rd_data_r[8])
          |||((vcount==CH8_L)&&~rd_data_r[8])
          |||((vcount>=CH8_H)&&(vcount<=CH8_L)&&wave_edge[8]);

assign wave[9]=((vcount==CH9_H)&&rd_data_r[9])
          |||((vcount==CH9_L)&&~rd_data_r[9])
          |||((vcount>=CH9_H)&&(vcount<=CH9_L)&&wave_edge[9]);

//栅格显示计算，共分 15 大格，每一大格分 4 个小格，每小格包括 8 个数据点
assign grid = (((hcount[9:0]==10'd0)
            ||(hcount[9:0]==10'd40)
            ||(hcount[9:0]==10'd80)
            ||(hcount[9:0]==10'd120)
            ||(hcount[9:0]==10'd160)
            ||(hcount[9:0]==10'd200)
            ||(hcount[9:0]==10'd240)
            ||(hcount[9:0]==10'd280)
            ||(hcount[9:0]==10'd320)
            ||(hcount[9:0]==10'd360)
            ||(hcount[9:0]==10'd400)
            ||(hcount[9:0]==10'd440)
            ||(hcount[9:0]==10'd480)
            ||(hcount[9:0]==10'd520)
            ||(hcount[9:0]==10'd560)
            ||(hcount[9:0]==10'd600))
            &&(vcount[1:0]!=2'd1))
            ||(vcount[0]==1'b0)
            &&(hcount[2:0]==3'd0);

//------------------显示数据输出------------------
always@(posedge VGA_clk)
if(mark&&disp_ena)           //显示基准线
    disp_data_r<=MarkColor;
else if(|wave&&disp_ena)      //显示波形
    disp_data_r<=WaveColor;
else if(grid)                //显示栅格
    disp_data_r<=GridColor;
else                         //显示背景
    disp_data_r<=BackColor;
```

```
endmodule
```

显示控制电路仿真验证，由于显示控制电路最终的输出数据是直接连接到 VGA 进行图像显示的，数据量大，而且通过仿真波形不易分析，因此对本模块暂不做仿真验证，如果后续发现无法实现功能，再对该模块的局部电路进行仿真验证，以查找问题。

系统控制模块

系统控制模块主要实现对输入设备输入的控制命令的解析，然后根据解析到的，命令控制系统相应参数，如调整触发通道、触发方式、启动采样、采样速率以及波形显示范围。这里，系统定义了 8 个功能按键，这 8 个功能按键分别实现不同的功能。而且，本系统支持多输入终端可配置，即在最终实现的时候，根据用户的实际系统配置，可以选择使用矩阵键盘、红外遥控、PS2 键盘等输入设备中的任意一种设备作为控制信号的输入。虽然输入设备不同，但因为矩阵键盘、红外遥控、PS2 键盘上都有数字 1~8，因此按键序号与对应功能并不改变。为了兼容这三种输入设备，我们只需要根据选择的数据设备，判断当前的输入键值是否与预期一致即可。下表为按键序号与对应的功能，同时，还给出了分别使用 PS2 键盘、红外遥控、矩阵键盘时这些按键对应的键值。

按键序号	按键功能	PS2 键盘键值	红外遥控键值	矩阵键盘键值
按键 1	触发通道选择	10'h045	8'h16	4'd0
按键 2	触发模式选择	10'h016	8'h0C	4'd1
按键 3	mark 减	10'h01E	8'h18	4'd2
按键 4	mark 加	10'h026	8'h5E	4'd3
按键 5	采样频率选择	10'h025	8'h08	4'd4
按键 6	单次采样	10'h02E	8'h1C	4'd5
按键 7	连续采样选择	10'h036	8'h5A	4'd6
按键 8	停止采样	10'h03D	8'h42	4'd7

本例使用的是条件编译的方式来选择输入设备的，即在工程编译前，设置系统需要使用的输入设备，然后工程将自动根据设置条件将选择输入设备电路编译到工程中。

为了将设计和配置区分开，使得我们需要更改配置时只需要修改一下配置文件，就能对工程中所有用到这个参数的电路生效，我们设计了一个独立的配置文件：synas_cfg.v，该文件内容如下所示：

```
01  `define USE_Key4x4_Board 1
02  //`define USE_PS2_Key_Board 1
03  //`define USE_IR 1
04
05  `define user_ir_addr 16'hff00
06
07  `ifdef USE_Key4x4_Board
08
09      `define Key_Word0 4'd0
10      `define Key_Word1 4'd1
22      `define Key_Word1 10'h016
23      `define Key_Word2 10'h01E
24      `define Key_Word3 10'h026
25      `define Key_Word4 10'h025
26      `define Key_Word5 10'h02E
27      `define Key_Word6 10'h036
28      `define Key_Word7 10'h03D
29      `define Key_Word8 10'h03E
30
31  `elsif USE_IR
```

```
11      `define Key_Word2 4'd2
12      `define Key_Word3 4'd3
13      `define Key_Word4 4'd4
14      `define Key_Word5 4'd5
15      `define Key_Word6 4'd6
16      `define Key_Word7 4'd7
17      `define Key_Word8 4'd8
18
19  `elsif USE_PS2_Key_Board
20
21      `define Key_Word0 10'h045
```

```
32
33      `define Key_Word0 8'h16
34      `define Key_Word1 8'h0C
35      `define Key_Word2 8'h18
36      `define Key_Word3 8'h5E
37      `define Key_Word4 8'h08
38      `define Key_Word5 8'h1C
39      `define Key_Word6 8'h5A
40      `define Key_Word7 8'h42
41      `define Key_Word8 8'h52
42  `endif
```

第 1、2、3 行为使用的硬件选择条件，例如，当使用矩阵键盘时，将第一行的注释取消，第 2、3 行注释以屏蔽，那么系统就选择了使用矩阵键盘作为输入设备。由于本系统设置了 8 个按键，这 8 个按键使用不同的输入设备时，其键值是不一样的，因此从第 7 行到第 42 行，设计了一个条件编译部分，第 7 到 17 行，表示当选择使用矩阵键盘作为输入设备时，就分别定义按键 0 到 8 的键值为 0~8，第 19 行到第 29 行表示当选择使用 PS2 键盘作为输入设备时，分别定义按键 0 到按键 8 的键值为 10'h045……10'h03E。31 行到 41 行则是定义使用红外遥控时每个按键的键值。这样，当我们在使用按键时，就不需要再根据当前使用的输入设备再来分别判断不同的键值，只需要使用` Key_WordX 进行判断即可，下面就是使用`Key_Word5 来判断按键状态从而控制采样频率的代码示例。

```
    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)
        freq_sel <= 4'd0;
    else if((Key_Valve== `Key_Word5)&&Key_Flag)//按键 5，采样频率选择
        freq_sel<=freq_sel+1'b1;
    else
        freq_sel<=freq_sel;
```

另外，由于不同的输入设备，其键值位宽是不一样的，PS2 键盘的键值为 10 位，红外遥控的键值位宽为 8 位，矩阵键盘的键值为 4 位，因此在定义端口位宽的时候，也需要根据选择的输入设备分别定义键值位宽，以下为控制模块中根据条件编译选项定义键值位宽的代码：

```
`ifdef USE_Key4x4_Board
    input  [3:0]Key_Valve;       //4x4 矩阵键盘键值
`elsif USE_PS2_Key_Board
    input  [9:0]Key_Valve;       //PS2 键盘键值
`elsif  USE_IR
    input  [7:0]Key_Valve;       //红外遥控键值
`endif
```

至此，控制模块中的一些设计技巧就介绍完了，剩下的就是根据按键调整对应参数的设计了，这些内容没有详解的必要。大家看到代码就能理解。以下为系统控制模块的完整代码：

```
`include "synsz_cfg.v"
module sys_ctrl(
    Clk,        //输入时钟（100MHZ）
```

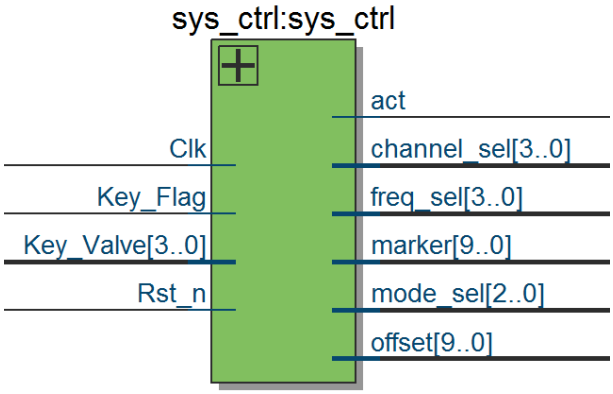
```
    Rst_n,
    Key_Valve,   //PS2 键值
    Key_Flag,    //PS2 按键动作标志信号
    act,        //启动采样
    offset,     //显示区域偏移值
    marker,     //基准线偏移值
    channel_sel, //触发通道选择
    mode_sel,    //触发模式选择
    freq_sel     //采样频率选择
);

    //------------模块输入端口---------------
    input  Clk;              //输入时钟（100MHZ）
    input   Rst_n;

`ifdef USE_Key4x4_Board
    input  [3:0]Key_Valve;       //4x4 矩阵键盘键值
`elsif USE_PS2_Key_Board
    input  [9:0]Key_Valve;       //PS2 键盘键值
`elsif  USE_IR
    input  [7:0]Key_Valve;       //红外遥控键值
`endif

    input  Key_Flag;            //按键动作标志信号

    //------------模块输出端口---------------
    output reg [9:0]offset;      //显示区域偏移值
    output reg [9:0]marker;      //基准线偏移值
    output reg [3:0]channel_sel; //触发通道选择
    output reg [2:0]mode_sel;    //触发模式选择
    output reg [3:0]freq_sel;    //采样频率选择
    output act;                 //启动采样

    //------------内部寄存器定义------------
    reg run;                    //连续采样

    //*************************按键功能处理*************************
    always@(posedge Clk  or negedge Rst_n)
    if(!Rst_n)
        channel_sel<=0;
    else if((Key_Valve== `Key_Word1)&&Key_Flag) begin//按键 1，通道选择
        if(channel_sel==4'd9)
            channel_sel<=0;
        else
```

```
        channel_sel<=channel_sel+1'b1;
end
else
    channel_sel<=channel_sel;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    mode_sel<=0;
else if((Key_Valve== `Key_Word2)&&Key_Flag)begin//按键 2，触发模式选择
    if(mode_sel==3'd5)
        mode_sel<=0;
    else
        mode_sel<=mode_sel+1'b1;
end
else
    mode_sel<=mode_sel;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    freq_sel <= 4'd0;
else if((Key_Valve== `Key_Word5)&&Key_Flag)//按键 5，采样频率选择
    freq_sel<=freq_sel+1'b1;
else
    freq_sel<=freq_sel;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    run<=1'b0;
else if((Key_Valve== `Key_Word7)&&Key_Flag)//按键 7，连续采样选择
    run<=1'b1;
else if((Key_Valve== `Key_Word8)&&Key_Flag)//按键 8，停止采样
    run<=1'b0;

assign act= ((Key_Valve== `Key_Word6)&&Key_Flag)|run;   //按键 6，单次采样

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)begin
    marker <= 10'd0;
    offset <= 10'd0;
end
else if(((Key_Valve== `Key_Word3)&&Key_Flag)) begin //按键 3，mark 减
    if(marker!=10'd0)
        marker<=marker-3'd5;
    else if(offset!=10'd0)          //offset 减
```

```
            offset<=offset-3'd5;
    end
    else if(((Key_Valve== `Key_Word4)&&Key_Flag)) begin //按键 4，mark 加
        if(marker<10'd639)
            marker<=marker+3'd5;
        else if(offset<10'd384)//offset 加
            offset<=offset+3'd5;
    end

endmodule
```

控制模块设计思路非常简单，不涉及到复杂的时序，可以暂不进行仿真，如果后续功能出现问题，再根据实际现象配合仿真来查找问题。

设计完成的系统控制代码如下所示：



至此，整个逻辑分析仪的关键部件设计就完成了，只需要将所有的模块按照预先设置好的架构连接起来，就能构成完整的系统了。本设计的精妙之处就在于对于功能模块的合理划分，通过将数据处理部分，显示部分，控制部分分开设计，并留出方便易用的接口，因此在最后组合系统时，不需要再进行任何的接口转换逻辑就能直接连接成为一个完整的系统，这也是笔者一直坚持的数字系统设计思想，即合理划分模块，通过让系统中的每一个模块既能独立设计验证，能够方便的与其他模块交互，实现灵活多变的应用，例如，本例中红外遥控、PS2 键盘、VGA 驱动、PWM 波模块等都可以直接提取出来作为一个独立的驱动用于其他设计中。希望读者仔细体会这种模块化设计的优势。

顶层代码设计

```
`include "synsz_cfg.v"

module synsz(
    input Clk50M,      //系统时钟
    input Rst_n,       //低电平复位信号
    input key_in0,       //独立按键输入

`ifdef USE_Key4x4_Board      //系统使用矩阵键盘作为输入设备
```

```
    input [3:0]Row_i,    //键盘行输入
    output [3:0]Col_o,    //键盘列输出
`endif

`ifdef USE_PS2_Key_Board    //系统使用 PS2 键盘作为输入设备
    input PS2_Din,
    input PS2_Clk,
`endif

`ifdef USE_IR   //系统使用红外遥控作为输入设备
    input iIR,
`endif

    output VGA_VS,    //VGA 场同步信号
    output VGA_HS,    //VGA 行同步信号
    output [7:0]VGA_RGB,//VGA 数据输出

    output Beep_o,  //蜂鸣器驱动

    input [9:0]s_in,    //逻辑分析仪待分析数据输入

    output [1:0]LED,    //LED 指示灯

    output [7:0]dig,    //数码管位选
    output [6:0]seg //数码管段选

);

    //PLL
    wire Clk25M;
    wire Clk_100M;

    //VGA
    wire [9:0]vcount;
    wire [9:0]hcount;

    //disp_controller
    wire [9:0]offset;
    wire [9:0]marker;
    wire rd_clk;
    wire [9:0]rd_addr;
    wire [9:0]rd_data;
    wire [7:0]disp_data;
```

```
    //div_freq
    wire clken;

    //sys_ctrl
    wire act;
    wire [3:0]freq_sel;
    wire [2:0]mode_sel;
    wire [3:0]channel_sel;

    //sample
    wire [9:0] wr_data;
    wire [9:0] wr_addr;
    wire wren;

`ifdef USE_Key4x4_Board
    //Key4x4
    wire   Key4x4_Flag;
    wire [3:0]Key4x4_Valve;

`elsif USE_PS2_Key_Board
    //PS2
    wire   PS2_Key_Flag;
    wire [9:0]PS2_Key_Valve;

`elsif USE_IR
    //红外遥控
    wire ir_Get_Flag;
    wire [15:0]irData;
    wire [15:0]irAddr;
    wire ir_Success;

    assign ir_Success = ir_Get_Flag && (~irData[7:0] ==
irData[15:8]) && (irAddr == `user_ir_addr);
`endif

    //独立按键
    wire key_flag0;
    wire key_state0;

    reg [9:0]cnt;

    wire [9:0]s;

    wire [31:0]HEX8_data;
```

```
    assign HEX8_data = {4'hc, 4'he, channel_sel, {1'b0,mode_sel},
4'hd, 4'hf, 4'hb, freq_sel};

    //-----------------模拟外部被测信号输入-----------------
    always@(posedge Clk50M or negedge Rst_n)
    if(!Rst_n)
        cnt<=0;
    else
        cnt<=cnt+1'b1;

    //-----------------调用 PLL 模块-----------------
    pll pll(
        .inclk0(Clk50M),//输入 50MHZ 时钟
        .c0(Clk_100M), //输出 100MHZ 时钟
        .c1(Clk25M)   //输出 VGA 时钟 25MHZ
    );

    VGA_CTRL VGA_CTRL(
        .Clk25M(Clk25M),    //系统输入时钟 25MHZ
        .Rst_n(Rst_n),
        .data_in(disp_data),    //待显示数据
        .hcount(hcount),        //VGA 行扫描计数器
        .vcount(vcount),        //VGA 场扫描计数器
        .VGA_RGB(VGA_RGB),  //VGA 数据输出
        .VGA_HS(VGA_HS),        //VGA 行同步信号
        .VGA_VS(VGA_VS)     //VGA 场同步信号
    );

    //-----------------调用波形显示控制模块-----------------
    disp_controller disp_controller(
        .VGA_clk(Clk25M),
        .disp_ena(1'b1),
        .offset(offset),
        .marker(marker),
        .vcount(vcount),
        .hcount(hcount),
        .rd_clk(rd_clk),
        .rd_data(rd_data),
        .rd_addr(rd_addr),
        .disp_data(disp_data)
    );

    //-----------------调用采样频率选择模块-----------------
```

```
    div_freq div_freq(
        .Clk100M(Clk_100M),//100MHZ 时钟输入
        .Rst_n(Rst_n),
        .clken(clken),   //时钟使能输出
        .sel(freq_sel)
    );

    //-----------------调用双口 RAM 模块-----------------
    dpram dpram(
        .data(wr_data),
        .wren(wren),
        .wraddress(wr_addr),
        .rdaddress(rd_addr),
        .wrclock(Clk_100M),
        .rdclock(rd_clk),
        .wrclocken(clken),
        .rdclocken(1'b1),
        .q(rd_data)
    );

    //-----------------调用信号采集触发模块-----------------
    sample sample(
        .Clk(Clk_100M),
        .Rst_n(Rst_n),
        .clken(clken),
        .act(act),
        .channel_sel(channel_sel),
        .mode_sel(mode_sel),
        .data_in(s),
        .wr_addr(wr_addr),
        .wr_data(wr_data),
        .wren(wren)
    );

     assign LED[1] = ~wren;//当进行采样时，LED1 亮，未进行采样，则 LED1 灭

    //-----------------系统控制模块-----------------
    sys_ctrl sys_ctrl(
        .Clk(Clk50M),        //输入时钟（50MHZ）
        .Rst_n(Rst_n),

`ifdef USE_Key4x4_Board
        .Key_Valve(Key4x4_Valve),    //4x4 矩阵键盘键值
        .Key_Flag(Key4x4_Flag),  //4x4 矩阵键盘检查成功标志
```

```
`elsif USE_PS2_Key_Board
      .Key_Valve(PS2_Key_Valve),   //PS2 键盘键值
      .Key_Flag(PS2_Key_Flag),     //PS2 按键动作标志信号
`elsif USE_IR
      .Key_Valve(irData[7:0]),     //红外遥控解码键值
      .Key_Flag(ir_Success),   //红外遥控解码成功标志信号
`endif

      .act(act),      //启动采样
      .offset(offset),        //显示区域偏移值
      .marker(marker),        //基准线偏移值
      .channel_sel(channel_sel), //触发通道选择
      .mode_sel(mode_sel),     //触发模式选择
      .freq_sel(freq_sel)  //采样频率选择
   );

   HXE8 HXE8(
      .Clk(Clk50M),
      .Rst_n(Rst_n),
      .En(1'b1),
      .disp_data(HEX8_data),
      .sel(dig),
      .seg(seg)
   );

`ifdef USE_Key4x4_Board
   //-----------------矩阵键盘模块-----------------
   Key4x4_Board Key4x4_Board(
      .Clk(Clk50M),
      .Rst_n(Rst_n),
      .Key_Board_Row_i(Row_i),
      .Key_Board_Col_o(Col_o),
      .Key_Flag(Key4x4_Flag),
      .Key_Value(Key4x4_Valve)
    );

`elsif USE_PS2_Key_Board
   PS2_Key_Board_Driver PS2_Key_Board_Driver(
      .Clk(Clk50M),
      .Rst_n(Rst_n),
      .PS2_Din(PS2_Din),
      .PS2_Clk(PS2_Clk),
      .Key_Flag(PS2_Key_Flag),
      .Key_Valve(PS2_Key_Valve)
```

```
    );

`elsif  USE_IR
        //-----------------红外遥控解码模块-----------------
    ir_decode ir_decode(
        .Clk50M(Clk50M),
        .Rst_n(Rst_n),
        .iIR(iIR),
        .Get_Flag(ir_Get_Flag),
        .irData(irData),
        .irAddr(irAddr)
    );
`endif

    //-----------------Beep 按键指示-----------------
    Beep Beep(
        .Clk(Clk_100M),
        .Rst_n(Rst_n),
`ifdef USE_Key4x4_Board
        .En(Key4x4_Flag),    //4x4 矩阵键盘检查成功标志
`elsif USE_PS2_Key_Board
        .En(PS2_Key_Flag),   //PS2 按键动作标志信号
`elsif USE_IR
        .En(ir_Success),     //红外遥控解码成功标志信号
`endif
//      .Beep_o()   //测试时，强制取消蜂鸣器鸣响
        .Beep_o(Beep_o) //正常系统蜂鸣器驱动
    );
//  assign Beep_o = 1'b0;//测试时，强制取消蜂鸣器鸣响

    key_filter key_filter1(
        .Clk(Clk50M),
        .Rst_n(Rst_n),
        .key_in(key_in0),
        .key_flag(key_flag0),
        .key_state(key_state0)
    );

//将逻辑分析仪的输入信号在内部测试信号和外部输入信号间切换
    reg EI_sel;

    always@(posedge Clk50M)
    if(key_flag0 && !key_state0)
        EI_sel <= ~EI_sel;
```

```
    assign s = EI_sel?s_in:cnt;

    assign LED[0] = EI_sel;//LED[0]灭：采样外部输入；LED[0]量：采样内部测试信号

endmodule
```

逻辑分析仪的使用

经过以上设计，逻辑分析仪系统就设计完成了，接下来我们可以编译工程并分配引脚，然后下载到开发板上，接上矩阵键盘或者使用红外遥控（需要在 synsz_cfg.v 中配置使用红外遥控）。就可以进行采样了，在顶层模块中，考虑到测试信号源可能不易获得，因此设计了一个计数器，并用按键 0 进行控制，通过按键 0 来选择是采样外部信号还是内部计数器的 10 位信号。

可以按照以下流程进行测试：

1. 下载 sof 文件到开发板中
2. 按下按键 0，将采样信号源切换为内部计数器信号（LED0 亮）
3. 连续按下 15 次按键 5，将采样率设置为 100MHz
4. 按下按键 7，启动连续采样。此时，我们可以看到屏幕上会出现密集的波形，表明已经采样到数据
5. 按下按键 4，可以看到光标右移，按下按键 3，光标左移，使用矩阵键盘时，保持按键按下不释放可以启动连按功能，当按下按键 4 或者按键 3 时可以看到光标快速右移或左移。当移动到屏幕最后边后继续按下，屏幕上的数据将开始更新，这就是在读取 640 点以后的数据并显示。

板级验证

这里我手头没有 VGA 显示器，因此先用 TFT 版的上个效果图。后续，我们还将使用该逻辑分析仪抓取 SPI、UART 等协议的波形。