题　　目　基于 DE1 开发板的 VGA 图像显示

目录

一 简介

1 功能简介

VGA(Video Graphics Array)是 IBM 在 1987 年随 PS/2 机一起推出的一种视频传输标准，具有分辨率高、显示速率快、颜色丰富等优点，在彩色显示器领域得到了广泛的应用。本设计是基于 altera 公司的 Development & Education Board 1 上的 VGA 接口开发的一个图形显示模块。通过 VGA 协议，可以在显示屏上显示分辨率为 640*480，刷新频率为 60Hz 的彩条及彩色图片。图片数据可以从 24bit 宽的 bmp 图像通过一个 C 程序来转化成需要的格式。本设计可以通过 altera DE1 自带的软件来把图片数据下载到 sram 里，也可以通过串口把数据下载到 sram 里。

2 用途

该模块可以用于显示器工厂里对显示屏的坏点检测，也可以在大屏幕上显示需要的图像信息，由于系统简单，可扩展性好，图像更新方便，比传统用 PC 机控制的方法有成本上的极大优势。并且可以通过加大像素同步信号的频率或者并联几个同样的显示模块来达到控制更大显示屏的目的，有极大的灵活性。

3 难度部分

本设计的**难度部分**主要在图像产生模块与串口通信模块两部分，在图像产生模块里完成了对 ram 里压缩格式数据的解压缩，否则未压缩的数据无法完全放到 sram 里。串口通信模块在开始信号捕捉，数据计数以及数据和时钟的同步上逻辑关系不太好处理，特别是一个进程里只能捕捉一个时钟沿信号，并且对同一信号的赋值只能在同一个进程里，这样就造成捕捉到起始信号的下降沿后无法直接对数据进行计数，只能通过标志位传递来进行同步。

4 创新部分

创新部分在于串口通信模块没有采用常用的状态机加移位寄存器的结构，而是采用直接对号入座的方法，便于对接收到的数据进行变换，使软件和硬件可以共同完成数据的编码解码，增加了系统的灵活性。

二 设计思路

1 系统整体结构

系统的整体结构如图所示，本电路包含 7 个模块。像素时钟产生模块(pixel)，同步信号产生模块(sync)，显示控制模块(control)，图像产生模块(picture_style)，数码管显示模块(hex_display)，串口通信模块（RS232_RXD），端口选择模块（sram_latch）。

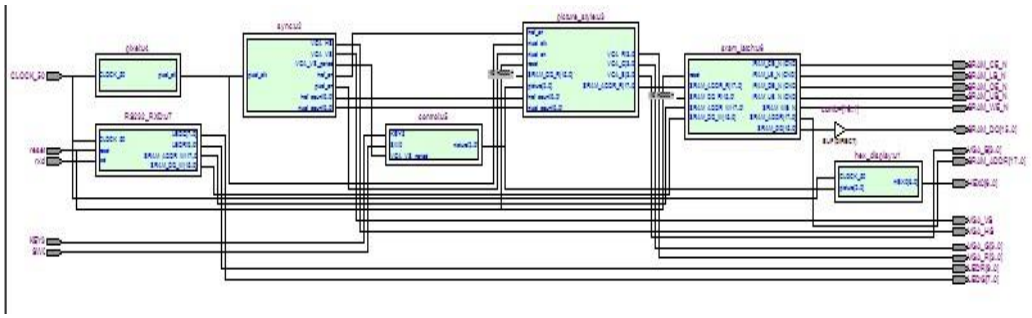

图 1 系统整体结构

像素时钟产生模块由系统时钟 50MHz 分频出 25MHz 的像素时钟。同步信号产生模块用来产生 VGA 相关的行同步时钟，场同步时钟以及奇偶场同步信号。显示控制模块定义了两种切换图片的模式，即手动模式和自动模式。图像产生模块里定义了 8 种显示图像，有从 sram 里读取的，也有直接产生的。通过计数，循环显示 8 种图片。数码管显示模块用来在数码管上显示图片的计数。串口通信模块完成与 PC 通过串口向 sram 下载图片的功能。端口选择模块用来对 sram 的数据源和地址源进行选取。

顶层实体的端口定义如下。

```
entity vga is
    port(
            CLOCK_50        :in         std_logic;
            KEY3            :in         std_logic;
            SW0             :in         std_logic;
            VGA_R           :out        std_logic_vector(3 downto 0);
            VGA_G           :out        std_logic_vector(3 downto 0);
            VGA_B           :out        std_logic_vector(3 downto 0);
            VGA_HS          :out        std_logic;
            VGA_VS          :out        std_logic;
            rxd             :in         std_logic;
            mode            :in         std_logic;
            SRAM_ADDR       :buffer     std_logic_vector(17 downto 0);
            SRAM_DQ         :inout      std_logic_vector(15 downto 0);
            HEX0            :out        std_logic_vector(6 downto 0);
            LEDG            :out        std_logic_vector(7 downto 0);
            LEDR            :out        std_logic_vector(9 downto 0);
            SRAM_WE_N,
            SRAM_OE_N,
            SRAM_UB_N,
            SRAM_LB_N,
            SRAM_CE_N       :out        std_logic
        );
end entity;
```

其中，CLOCK_50 为时钟输入引脚，输入 50MHz 的时钟信号作为系统时钟。KEY3 按键是手动切换按钮，每按一次就切换一次画面。SW0 开关是自动/手动切换开关，高电平时自动定时切换画面，低电平是用 KEY3 来控制切换。VGA_R，VGA_G 和 VGA_B 为 RGB 数据输入端，每个均为 4 位宽。VGA_HS 是行同步信号。mode 为工作模式选择端口，高电平时通过串口通信向 sram 里写数据，低电平时通过 VGA 在屏幕上显示图像。由 FPGA 按 VGA 时序产

生。同理 VGA_VS 是场同步信号。rxd 是 RS232 串口通信的接收端，SRAM_ADDR 为与 sram 相连的 18 位地址总线，可 0 到 512K 随机寻址。SRAM_DQ 为 16 位数据总线，给出地址后就可以在数据总线上直接进行读写。SRAM_WE_N，SRAM_OE_N, SRAM_UB_N, SRAM_LB_N 和 SRAM_CE_N 为 sram 的控制总线，均为低电平有效。根据 sram 的真值表进行相应的控制，这里由于之需要进行读操作，故除 SRAM_WE_N 接可控制端，其他都直接接地就可以。

2 像素时钟产生模块

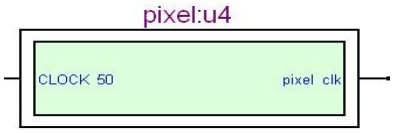

图 2 像素时钟产生模块

根据 VGA 协议，像素时钟为大约 25MHz 的信号，故只需对 50MHz 的系统时钟二分频就能得到。

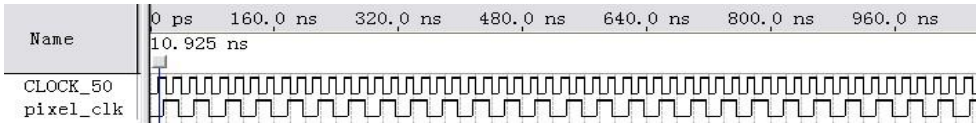

图 3 分频仿真图

```
entity pixel is
    port(
            CLOCK_50 :in      std_logic;
            pixel_clk:out std_logic
        );
end;

architecture arc of pixel is
begin
pixel:                                                  --generate 25MHz clock
    process(CLOCK_50)
    variable pixel_clk_t   :std_logic;
    begin
        if CLOCK_50'event and CLOCK_50='1' then
            pixel_clk_t:= not pixel_clk_t;
        end if;
        pixel_clk<=pixel_clk_t;
    end process;
end;
```

2.1 VGA 硬件电路

DE1 的 VGA 接口自带电阻阵列进行了简单的 DA 转换，电路图如图 4。
RGB 每个通道都是由 4 位宽的数据来控制，故能够显示 16*16*16=4096 种颜色，这就使得可以用此电路显示一些颜色相对复杂的图片。

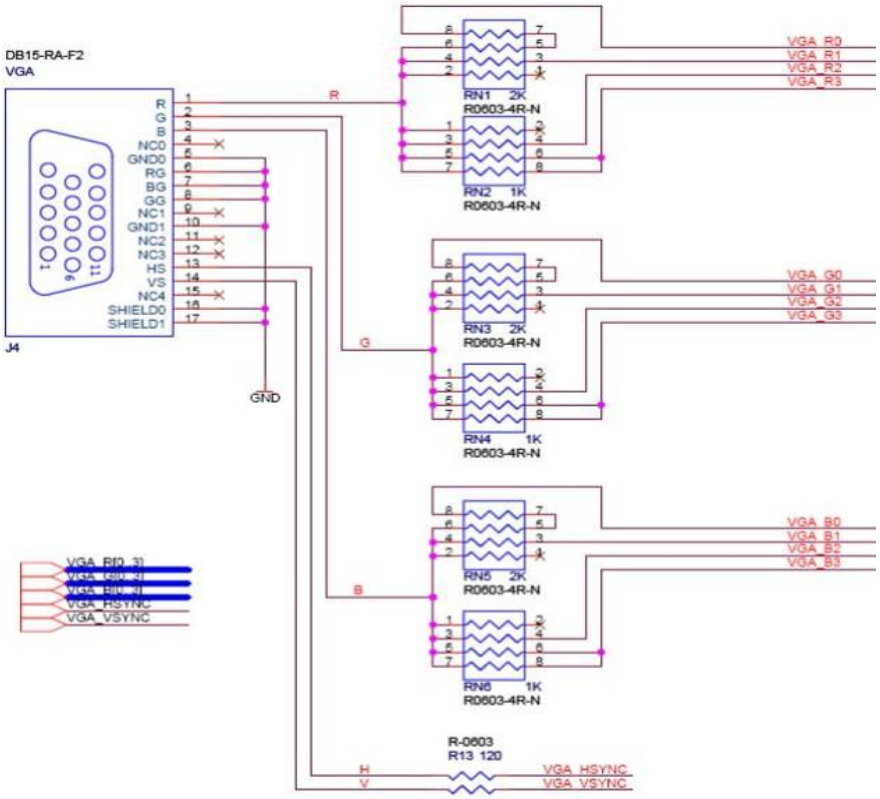

图 4 VGA 模块电路图

2.2 VGA 协议

VGA 的显示协议如图 2-2。要想在屏幕上显示图像，FPGA 需要输出 VGA 需要的 3 个同步信号，即像素同步，行同步和场同步。三个信号的时间关系如表 2-1 所示。具体来说。检测到每个像素同步的上升沿时对 RGB 数据端口的数据进行更新，即输出数据。行有效期间输出数据，行消隐期间停止输出数据。场有效期间正常输出数据，场消隐期间必须使 RGB 输出均为 0，行同步信号可以输出，也可以不输出。这里场消隐区如果有数据输出，就会造成显示器识别不出数据而显示频率超出范围。行同步和场同步为高电平脉冲和低电平脉冲都可以，因为显示器内部能够自动识别这两种规范，故只要脉冲持续时间及周期对就可以。

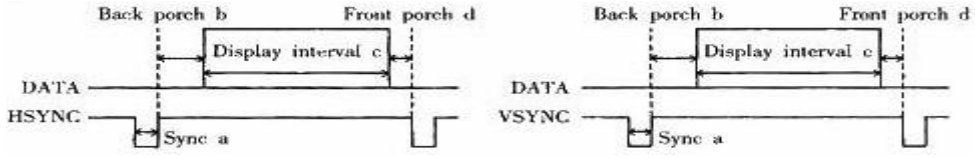

图 5 VGA 时序

Parameter	Vertical Sync			Horizontal Sync	
	Time	Clocks	Lines	Time	Clocks
Sync pulse time	16.7ms	416800	521	32us	800
Display time	15.36ms	384000	480	25.6us	640
Pulse width	64us	1600	2	3.84us	96
Front porch	320us	8000	10	640us	16
Back proch	928us	23200	29	1.92us	48

表 1 VGA 时序与时钟关系

3 同步信号产生模块

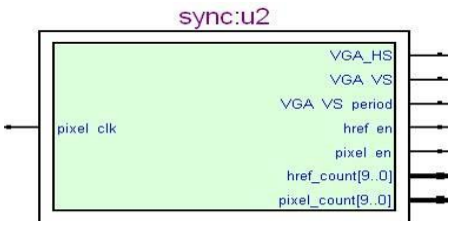

图 6 同步信号产生模块

根据 VGA 协议，需要通过对像素同步信号计数，产生行同步信号 VGA_HS 和场同步信号 VGA_VS 以及奇偶场同步信号 VGA_VS_period。同时，在本模块中也产生一些控制信号。

在 pixel 进程里从 50MHz 的系统时钟中分频出 25MHz 的时钟作为像素同步。在 href 进程里通过查像素同步的个数来确定行同步信号。并为行消隐区不显示数据定义 pixel_en 信号来进行控制。在 vsync 进程里通过查行同步脉冲的个数来确定场同步脉冲。在 scencewait 进程中通过查场的个数来确定自动图片切换时间，这里选择每隔 60 场自动切换一次图片。

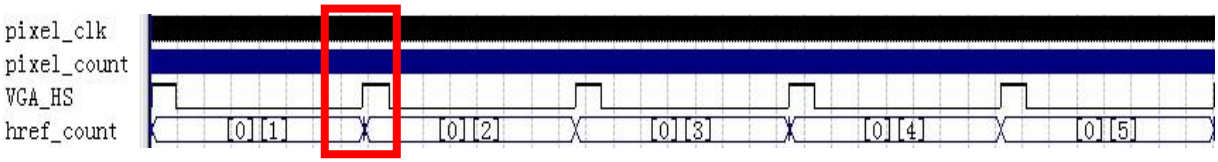

图 7 像素同步信号与行同步信号仿真

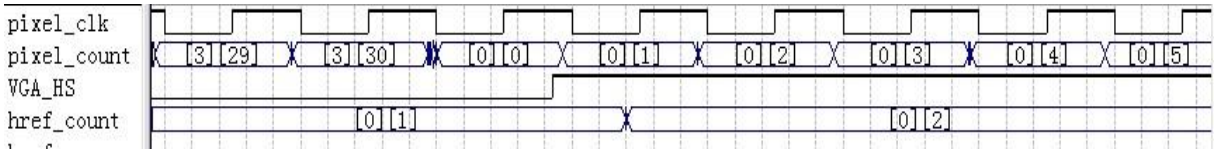

图 8 像素同步信号与行同步信号仿真的局部放大

仿真 1 显示了像素同步时钟与行同步信号的关系，其中 pixel_count 是对 pixel_clk 的计数，当数到 31E，即 798 时，VGA_HS 变为高电平，之后又数到 96 后变为低电平，然后重复，这就产生了行同步信号，href_count 是对行同步信号的脉冲计数。同理，行同步信号与场同步信号之间的关系也是如此。由于本模块的其他信号时间差距比较大，比如像素时钟为 25MHz，而场同步为 60Hz，故仿真起来很不方便，所以只是从示波器上观察产生的信号是否正确。

```
entity sync is
      port(
                  VGA_HS              :out      std_logic;
                  VGA_VS              :out      std_logic;
                  pixel_clk           :in       std_logic;
                  pixel_en            :out      std_logic;
                  VGA_VS_period       :buffer   std_logic;
                  href_count          :buffer   integer range 0 to 525;
                  pixel_count         :buffer   integer range 0 to 798;
                  href_en             :buffer   std_logic
            );
end;
architecture arc of sync is
constant    href_all_dot:integer:=798;   --800 pixel dots in a line including blanking zone
constant    href_sy_dot       :integer:=96;      --href high level time is 96 dot
constant    ver_all_line:integer:=525;
signal            VGA_HS_period:std_logic;
begin
href:                                                                  --generate Hsync -high level puls
      process(pixel_clk,href_en)
```

```
    begin
        if pixel_clk'event and pixel_clk='1' then
            if pixel_count<href_sy_dot then
                VGA_HS_period<='1';
                pixel_en<='0';
            elsif pixel_count<144 then
                VGA_HS_period<='0';
                pixel_en<='0';
            elsif pixel_count<784 then                    --pixel enable when 136-784
                VGA_HS_period<='0';
                pixel_en<='1';
            elsif pixel_count<href_all_dot then
                VGA_HS_period<='0';
                pixel_en<='0';
            end if;
            if pixel_count=href_all_dot then
                pixel_count<=0;
            else pixel_count<=pixel_count+1;
            end if;
        end if;
        if href_en='0' then
            pixel_en<='0';                                --stop pixel when href is disabled
        end if;
        VGA_HS<=VGA_HS_period;
    end process;
vsync:                                                    --generate Vsync
    process(VGA_HS_period)
    begin
        if VGA_HS_period'event and VGA_HS_period='1' then
            if href_count<2 then                          --pull down to 0 when Vsync
                VGA_VS_period<='0';
                href_en<='0';
            elsif href_count<35 then
                VGA_VS_period<='1';
                href_en<='0';
            elsif href_count<515 then
                VGA_VS_period<='1';
                href_en<='1';                             --href enable when 35-515
            elsif href_count<ver_all_line then
                VGA_VS_period<='1';
                href_en<='0';
            end if;
            if href_count=ver_all_line then
                href_count<=0;
            else href_count<=href_count+1;
            end if;
        end if;
    VGA_VS<=VGA_VS_period;
    end process;
end;
```

4 显示控制模块

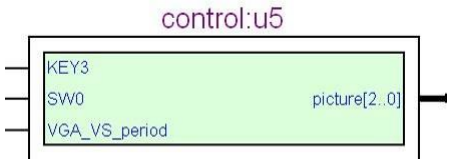

图 9 显示控制模块

数够 60 场的时间后 auto_picture_count_s 产生一个上升沿，按键 KEY3 按一下产生一个低电平脉冲。故定义 picture_count_s 为 auto_picture_count_s 与 KEY3 相异或，在 Style_control 进程里识别 picture_count_s 的上升沿，每个上升沿到来就让图片计数加一。这样就能达到既能手动控制又能自动控制的目的。

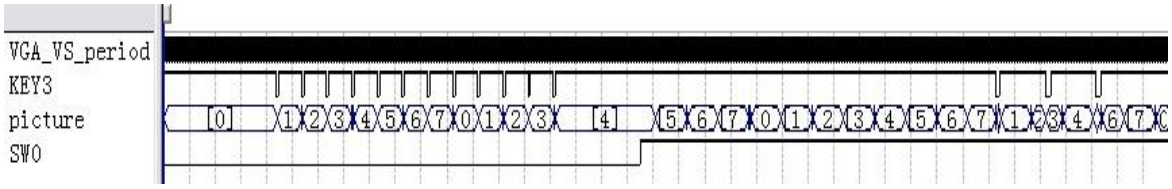

图 10 显示控制模块时序仿真

如图，SW0 为低电平时是手动模式，此时没按一次 KEY3，即 KEY3 段产生一个低电平脉冲，picture 计数加 1。不按 KEY3 时 picture 不变。当 SW0 为高电平时为自动模式，此时隔一定的时间 picture 自动加 1，并且在自动模式下，每按一次 KEY3，picture 也加 1。达到了手动与自动相结合控制的目的。

```
entity control is
	port(
		VGA_VS_period	:in	std_logic;
		SW0		:in	std_logic;
		KEY3		:in	std_logic;
		picture		:buffer	integer range 0 to 7
	);
end;
architecture arc of control is
signal		scence_wait	:std_logic;
signal		SW0_t		:std_logic;
signal		picture_count_s:std_logic;
signal		auto_picture_count_s:std_logic;
begin
scencewait:                                                    --change picture after some scences
	process(VGA_VS_period)
	variable scence_wait_t	:integer range 0 to 31;
	begin
		if VGA_VS_period'event and VGA_VS_period='1' then
			if scence_wait_t<30 then
				scence_wait_t:=scence_wait_t+1;
			else scence_wait_t:=0;
				scence_wait<=not scence_wait;
			end if;
		end if;
	end process;
	SW0_t<=SW0;
auto_count_s:                                                  --change picture after push SW0
	process(SW0_t,scence_wait)
	begin
		if SW0_t='1' then
			if scence_wait'event and scence_wait='1' then
				auto_picture_count_s<=not auto_picture_count_s;
			end if;
		else auto_picture_count_s<='1';
		end if;
	end process;
	picture_count_s<=auto_picture_count_s xor KEY3;		--two ways to control
Style_control:
	process(picture_count_s)
	begin
		if picture_count_s'event and picture_count_s='1' then
			picture<=picture+1;
		end if;
	end process;
end;
```

5 图像产生模块

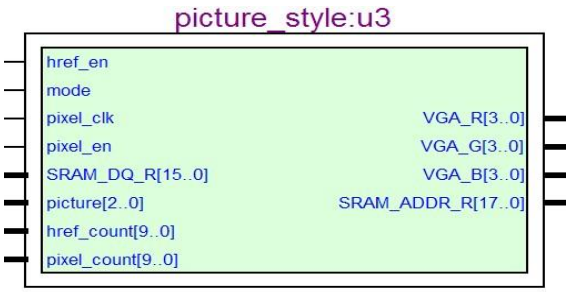

图 11 图像产生模块

Picture_style 进程是对显示的图片进行控制。这里定义了 8 种显示图像，0 为自定义的图片；1 为纯白；2 为纯黑；3 为纯红；4 为纯绿；5 为竖直彩条；6 为竖直渐变彩条；7 为水平彩条。每个像素时钟到来时，就按照图片计数里的数字显示相应的图片。其中第 0 种情况是读取 sram 里的数据来进行显示，由于 sram 的读取时间为 10ns，而 FPGA 的时钟周期为 20ns，故可以直接对 sram 进行读取。这里用到一个状态机是因为对原始图像数据进行了压缩，把空余的空间也用上了，即用 3 个存储单元储存 4 个像素的数据。故读取的时候需要读三个数据后空一次，并把每次读取的数据跟 RGB 三个通道对应好。

分辨率是 640*480,每个像素由 RGB3 个分量组成,每个分量由四位二进制数来表示,这样一幅彩色图片在不压缩的情况下要用 640*480*12bit,即大约 460K 字节.所以需要外部存储器来储存图像信息。开发板上有一片 512k 字节的 sram，并且 sram 的存取相对简单，故这里用 sram 作为存储器件。Sram 的硬件连接如图 9。由于 sram 的数据总线是 18 位，一个储存单元只放一个像素会空出四位，即加入一个像素的数据为 F(R)F(G)F(B),则在 sram 中储存格式为 0FFF。而且这样用存一张图片要用 400k 字节的空间，sram 无法提供这么大的空间。故只能考虑对数据进行压缩，即(FFF)(F FF)(FF F)(FFF),用三个存储单元储存 4 个像素的数据，这就要求在用 VHDL 实现的时候需要注意以下读取格式。

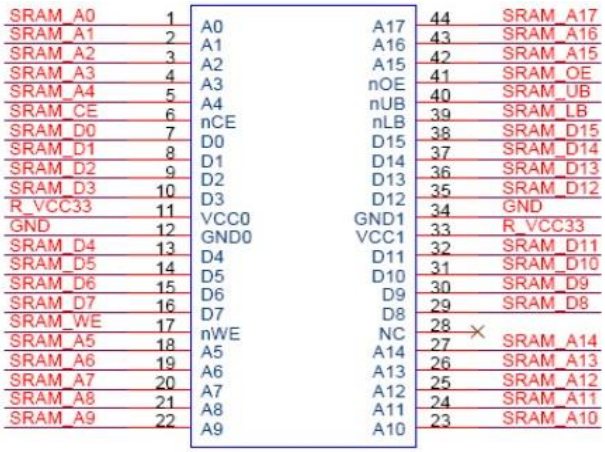

图 12 sram 硬件连接图

本部分代码比较长，故在附录 1 中给出。

由于本电路只能显示已经储存在 sram 里的特定格式的图片，故需要先用自己编写的 c 程序对 bmp 格式的图片进行转换，并用 altera 自带的软件把数据先烧写到 sram 中。图像处理的 C 程序见附录 2。

6 数码管显示模块

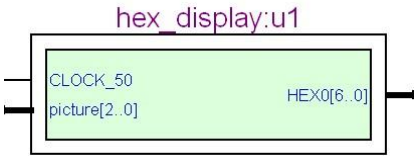

图 13 数码管显示模块

本模块的作用是把对显示图像的计数用数码管显示出来，即显示 0-7。

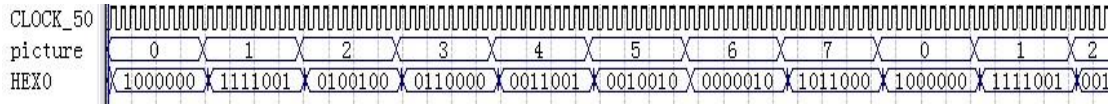

图 14 数码管显示模块仿真波形

```
entity hex_display is
        port(
                HEX0        :out        std_logic_vector(6 downto 0);
                CLOCK_50    :in         std_logic;
                picture     :in         integer range 0 to 7
            );
end;
architecture art of hex_display is
        begin
hex_display:
        process(picture,CLOCK_50)
            begin
            if CLOCK_50'event and CLOCK_50='1' then
                case picture is
                    when 0 =>HEX0<="1000000";
                    when 1 =>HEX0<="1111001";
                    when 2 =>HEX0<="0100100";
                    when 3 =>HEX0<="0110000";
                    when 4 =>HEX0<="0011001";
                    when 5 =>HEX0<="0010010";
                    when 6 =>HEX0<="0000010";
                    when 7 =>HEX0<="1011000";
                    when others=>NULL;
                end case;
            end if;
        end process;
end;
```

7 串口通信模块

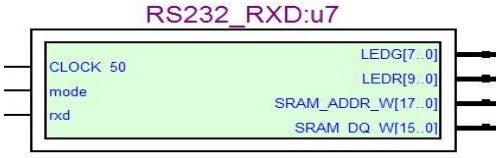

图 15 串口通信模块

该模块的功能是通过串口从 PC 机上读取图像信息，并把读到的图像数据写到 sram 里面，以供在图像输出模式时进行读取。在本模块中，包含一个子模块用于产生波特率为 115200 的时钟。

由前面可知一个图片的数据量为 450K byte，而传输速率为 115200/10=11.52K byte/s，故理想的最小传输时间为 450/11.52=39s。这是用 PC 机通过串口传输原始格式的图片数据的最

小时间。

捕捉 rxd 端在空闲状态下的下降沿，来识别数据的开始，由于数据传输格式是 8 位数据，无奇偶校验，一个停止位，故检测到开始信号后经过八个时钟可以读取八位数据，之后就是停止位。在读到停止位的时候把读到的完整数据写到 sram 里，由于 sram 的数据总线宽度为 16 位，而串口一次只能传 8 位，故需要连续采集两次然后一起写入 sram，模块里的 double_done 就是用来标志连续采集完两个字节，可以发送的标志位。LEDG 和 LEDR 总共 18 个 led 灯，用来以二进制形式显示写到的地址，可以大体检测传输数量是否正确。

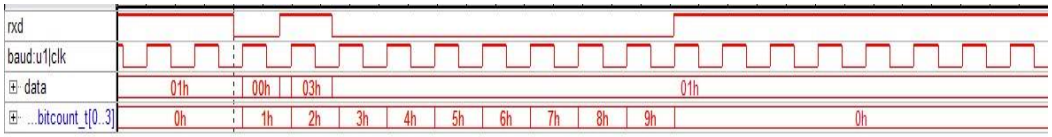

图 16 串口通信模块 SignalTapII 分析

上图为用 SignalTapII 进行逻辑分析的波形图。用串口工具在 rxd 端输入 0x01，clk 为波特率是 115200bps 的时钟信号。Data 储存串口读到的信号。Bitcount 是对读入的数据位进行计数。

无数据时 rxd 为高电平，此时模块处于 idle 状态，当 rxd 端有数据，首先捕捉起始信号，即下降沿，之后在每个时钟的上升沿读取 rxd 端的数据，并按位存到 data 中，每读一位 bitcount 就加 1，直到读完 1 个起始位，8 个数据位。由图可见读完 8 位数据位后 data 的数为 01h，与发送数据一致。

这里需要注意的一点就是如果 FPGA 产生的 clk 比 PC 的波特率频率高，则会出现同一个数据位读两次的错误，而如果小一些，则会出现某一数据位无法读出的情况，即跳过了一位。如果二者频率相等，也会有很小的概率读取到不稳定的信号。故这种传输机制误码是无法避免的。

```
entity RS232_RXD is
	port(
		CLOCK_50  :in		std_logic;
		rxd		:in		std_logic;
		mode		:in		std_logic;		--0:reset    1:transmit
		LEDG		:out	std_logic_vector(7 downto 0);	--display the address
		LEDR		:out	std_logic_vector(9 downto 0);
		SRAM_ADDR_W :buffer	std_logic_vector(17 downto 0);
		SRAM_DQ_W	:out	std_logic_vector(15 downto 0)
	);
end;
architecture arc of RS232_RXD is
component baud
	port(
		CLOCK_50  :in		std_logic;
		clk		:out	std_logic
	);
end component;
signal	clk		:std_logic;
signal	rxdata_start:std_logic:='0';		--set to 1 when capture the falling edge of rxd when in idle state(rxd_done=0)
signal	rxd_done	:std_logic:='1';		--0 whne transmiting while 1 when transmission is done.
signal	double_done	:std_logic:='0';		--collect two byte then write in to ram
signal	bitcount	:integer range 0 to 15; --1 start, 8 data,1 stop.
signal	data		:std_logic_vector(7 downto 0);
begin
u1:baud port map(CLOCK_50,clk);
	process(rxd)
	begin
		if rxd_done='1' then
			if rxd'event and rxd='0' then
				rxdata_start<='1';
			end if;
		end if;
```

```
            if bitcount=10 then
                  rxdata_start<='0';
            end if;
      end process;

txdata:
      process(clk)
      variable    bitcount_t  :integer range 0 to 15;
      begin
            case bitcount_t is
                  when 0 =>         NULL;--SRAM_DQ_W<=(others=>'Z');--start
                  when 1 =>         data(0)<=rxd;rxd_done<='0';              --data
                  when 2 =>         data(1)<=rxd;
                  when 3 =>         data(2)<=rxd;
                  when 4 =>         data(3)<=rxd;
                  when 5 =>         data(4)<=rxd;
                  when 6 =>         data(5)<=rxd;
                  when 7 =>         data(6)<=rxd;
                  when 8 =>         data(7)<=rxd;
                  when 9 =>         rxd_done<='1';                           --stop
                  when others => NULL;
            end case;
            if mode='1' then
                  if rxdata_start='1' then
                        if clk'event and clk='1' then
                              bitcount_t:=bitcount_t+1;
                              if bitcount=9 then                                    --write data into ram
                                    double_done<=not double_done;
                                    if double_done='0' then
                                          SRAM_DQ_W(15 downto 8)<=data;
                                    elsif double_done='1' then
                                          SRAM_DQ_W(7 downto 0)<=data;
                                    end if;
                              end if;
                              if bitcount=8 then
                                    if double_done='0' then
                                          SRAM_ADDR_W<=SRAM_ADDR_W+1;
                                          LEDG<=SRAM_ADDR_W(7 downto 0);
                                          LEDR<=SRAM_ADDR_W(17 downto 8);
                                    end if;
                              end if;
                        end if;
                  else
                        rxd_done<='1';
                        bitcount_t:=0;
                  end if;
            else
                  SRAM_ADDR_W<=(others=>'0');
                  bitcount_t:=0;
                  double_done<='1';
            end if;
            bitcount<=bitcount_t;
      end process;
end;end;
```

8 端口选择模块

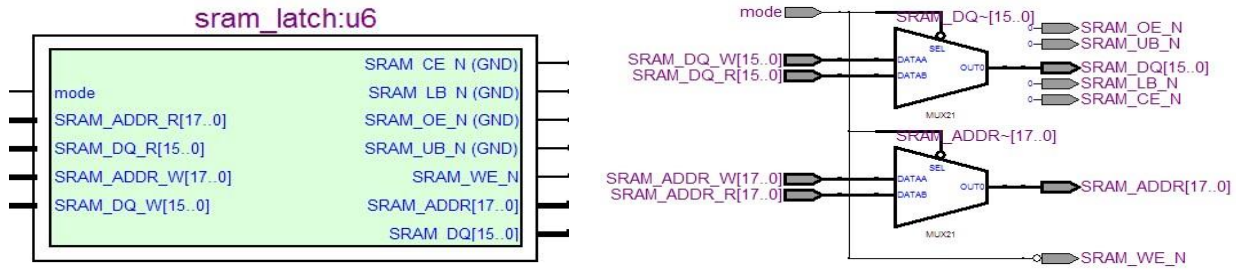

图 17 端口选择模块及其 RTL 视图

由于图像产生模块和串口通信模块都要对 sram 进行读写，故需要一个锁存器对端口进行锁存，以免信号混乱或者多驱动源的错误，同时，本模块也对 sram 进行一些端口设置。通过 mode 来选择 sram 的地址和数据的输入源，SRAM_ADDR_R 和 SRAM_DQ_R 与图像产生模块相连。SRAM_ADDR_W 和 SRAM_DQ_W 与串口通信模块相连。根据 mode 端选择从 SRAM_ADDR 和 SRAM_DQ 输出。向 sram 里写数据的时候 SRAM_WE_N 为低电平，读的时候需要为高电平。从 RTL 视图可以看出生成了 2 个数据选择器，mode 为控制端，与模块欲完成功能一致。下面是仿真波形。

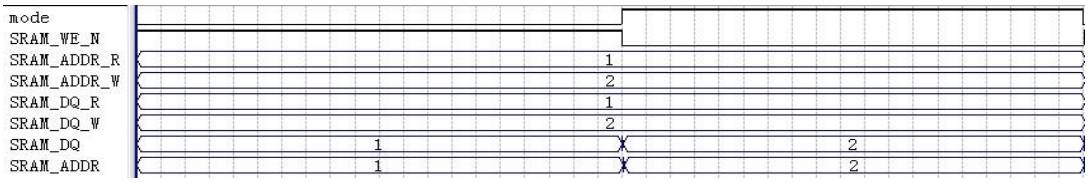

图 18 端口选择模块仿真波形

```
entity sram_latch is
	port(
		mode		:in	std_logic;
		SRAM_ADDR	:out	std_logic_vector(17 downto 0);
		SRAM_DQ		:out	std_logic_vector(15 downto 0);
		SRAM_ADDR_R	:in	std_logic_vector(17 downto 0);
		SRAM_DQ_R	:in	std_logic_vector(15 downto 0);
		SRAM_ADDR_W	:in	std_logic_vector(17 downto 0);
		SRAM_DQ_W	:in	std_logic_vector(15 downto 0);
		SRAM_WE_N,
		SRAM_OE_N,
		SRAM_UB_N,
		SRAM_LB_N,
		SRAM_CE_N	:out	std_logic );
end;
architecture arc of sram_latch is
begin
	SRAM_OE_N<='0';
	SRAM_UB_N<='0';
	SRAM_LB_N<='0';
	SRAM_CE_N<='0';
	process(mode)
		begin
		if mode='0' then			--display
			SRAM_WE_N<='1';
			SRAM_ADDR<=SRAM_ADDR_R;
			SRAM_DQ<=SRAM_DQ_R;
		else	SRAM_WE_N<='0';	--transmission
			SRAM_ADDR<=SRAM_ADDR_W;
			SRAM_DQ<=SRAM_DQ_W;
		end if;
	end process;
end;
```

三 实验结果

图 19 从 ram 里读取图像 0 信息

图 20 显示图像 1 纯白图像

图 21 显示图像 2 纯蓝图像

图 22 显示图像 3 纯红图像

图 23 显示图像 4 纯绿图像

图 24 显示图像 5 竖直彩条

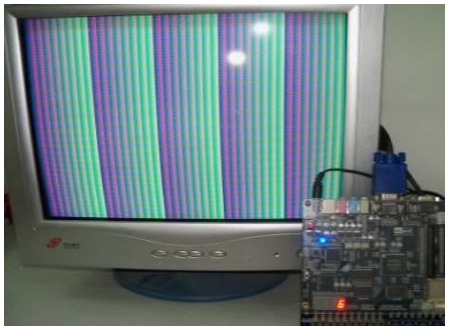

图 25 显示图像 6 颜色渐变

图 26 显示图像 7 水平彩条

四 总结

通过本电路的实现，我对 VGA 接口，sram 的读写控制以及图像的格式处理有了比较全面的了解。通过串口通信模块的编写，对通信协议的编写也有了一定的了解。但是本实验还存在一些遗留问题。

一是 sram 的存储空间毕竟还是有限，如果想储存几张照片来进行照片的切换时无法实现的，故考虑用 sdram 来实现，但是 sdram 的速度相对 sram 和 FPGA 比较慢，一般需要通过并联几个 sdram 的方法来弥补速度缺陷，但是在开发板只有一片 sdram 的情况下无法试验，另外就是采用动态加载的方式，即把几个图片的数据都存在 sdram 里，切换时从 sdram 加载到 sram 里，然后再对 sram 进行实时读取，这种方法有待于进一步实验。

二是想要显示动态图像的话一般是用双口 ram 来实现，如果用双口 ram 也可以做到串口下载与图像显示实时进行，并能在此基础上实现简单动画效果，但由于片内资源有限，无法做出足够大的双口 ram。单口 sram 由于速度比较快，在 FPGA 加入 PLL 的情况下理论上应该可以实现动态图像的显示，但是时序控制会复杂很多，需要再做相关实验。

三是虽然可以用串口下载图片，比用软件下载操作简单一些，但是现有的串口助手没有太合适的，虽然有可以以 HEX 格式传送文件的工具，但是由于没压缩的图片数据量比较大，串口工具无法一次加载所有数据，只能分段加载，造成比用 altera 自带软件操作更繁琐，而用超级终端还需要遵循它特定的协议，故理想状况下需要自己编写一个串口助手能一次性加载 900K 字节的数据并批量发送，计划以后用 QT 来实现。

四是如串口通信模块里所说，由于主机和从机的波特率时钟相互有三种关系，并且在本模块定义的传输机制下每种情况都有误码的概率，因此希望找到一种方法可以消除误码，或许加入奇偶校验可以在一定程度上降低误码率。

另外就是在系统优化方面速度和面积优化都没有考虑，还需要进行进一步优化。

附录 1

```
entity picture_style is
	port(
		VGA_R		:out	std_logic_vector(3 downto 0);
		VGA_G		:out	std_logic_vector(3 downto 0);
		VGA_B		:out	std_logic_vector(3 downto 0);
		SRAM_ADDR_R	:buffer	std_logic_vector(17 downto 0);
		SRAM_DQ_R	:in	std_logic_vector(15 downto 0);
		picture		:in	integer range 0 to 7;
		pixel_en	:in	std_logic;
		pixel_clk	:in	std_logic;
		href_en		:in	std_logic;
		mode		:in	std_logic;
		href_count	:in	integer range 0 to 525;
		pixel_count	:in	integer range 0 to 798
	);
end;
architecture art of picture_style is
begin
picture_style:
	process(picture,pixel_en,pixel_clk,href_en)
	variable rgb_state	:integer range 0 to 3;
	variable RGB_add	:std_logic_vector(11 downto 0);
	variable rgb_temp1b,rgb_temp2b,rgb_temp3g,rgb_temp4b,rgb_temp5g,rgb_temp6r:std_logic_vector(3 downto 0);
	begin
		if mode='0' then
			if pixel_en='1' then
				if pixel_clk'event and pixel_clk='1' then
					case picture is
						when 0 =>                                                    --picture in sram
							if href_count<515 then
								if rgb_state=3 then
									rgb_state:=0;
								else rgb_state:=rgb_state+1;
								end if;
								case rgb_state is
								when 0 =>                                                    --read 4 byte one
time,RGBR GBRG BRGB .so read 3 times while display 4 pixels
                                                                SRAM_ADDR_R<=SRAM_ADDR_R+1;
                                                                VGA_R<=SRAM_DQ_R(11 downto 8);
                                                                VGA_G<=SRAM_DQ_R(7 downto 4);
                                                                VGA_B<=SRAM_DQ_R(3 downto 0);
                                                                rgb_temp1b:=SRAM_DQ_R(15 downto 12);
                                                        when 1 =>
                                                                SRAM_ADDR_R<=SRAM_ADDR_R+1;
                                                                VGA_R<=SRAM_DQ_R(7 downto 4);
                                                                VGA_G<=SRAM_DQ_R(3 downto 0);
                                                                VGA_B<=rgb_temp1b;
                                                                rgb_temp2b:=SRAM_DQ_R(11 downto 8);
                                                                rgb_temp3g:=SRAM_DQ_R(15 downto 12);
                                                        when 2 =>
                                                                SRAM_ADDR_R<=SRAM_ADDR_R+1;
                                                                VGA_R<=SRAM_DQ_R(3 downto 0);
                                                                VGA_G<=rgb_temp3g;
                                                                VGA_B<=rgb_temp2b;
                                                                rgb_temp4b:=SRAM_DQ_R(7 downto 4);
                                                                rgb_temp5g:=SRAM_DQ_R(11 downto 8);
                                                                rgb_temp6r:=SRAM_DQ_R(15 downto 12);
                                                        when 3 =>
                                                                VGA_R<=rgb_temp6r;
                                                                VGA_G<=rgb_temp5g;
                                                                VGA_B<=rgb_temp4b;
                                                        when others => NULL;
                                                        end case;
                                                else SRAM_ADDR_R<=(others=>'0');--b"000100101011111111";--4AFF";--
                                                        VGA_R<="0000";                --black
                                                        VGA_G<="0000";
                                                        VGA_B<="0000";
                                                        rgb_state:=0;
                                                end if;
                                        when 1 =>
                                                        VGA_R<="1111";                --white
                                                        VGA_G<="1111";
                                                        VGA_B<="1111";
                                        when 2 =>
                                                        VGA_R<="0000";                --blue
                                                        VGA_G<="0000";
                                                        VGA_B<="1111";
                                        when 3 =>
```

```
                                VGA_R<="1111";                  --red
                                VGA_G<="0000";
                                VGA_B<="0000";
                        when 4 =>
                                VGA_R<="0000";                  --green
                                VGA_G<="1111";
                                VGA_B<="0000";
                        when 5 =>                                       --vertical colour bar
                                if pixel_count<224 then
                                        VGA_R<="1111";
                                        VGA_G<="1111";
                                        VGA_B<="1111";
                                elsif pixel_count<304 then
                                        VGA_R<="0000";
                                        VGA_G<="1111";
                                        VGA_B<="0000";
                                elsif pixel_count<384 then
                                        VGA_R<="0000";
                                        VGA_G<="0000";
                                        VGA_B<="1111";
                                elsif pixel_count<464 then
                                        VGA_R<="1111";
                                        VGA_G<="1111";
                                        VGA_B<="0000";
                                elsif pixel_count<544 then
                                        VGA_R<="0000";
                                        VGA_G<="1111";
                                        VGA_B<="1111";
                                elsif pixel_count<624 then
                                        VGA_R<="1111";
                                        VGA_G<="0000";
                                        VGA_B<="1111";
                                elsif pixel_count<704 then
                                        VGA_R<="1111";
                                        VGA_G<="0000";
                                        VGA_B<="0000";
                                else
                                        VGA_R<="0000";
                                        VGA_G<="0000";
                                        VGA_B<="0000";
                                end if;
                        when 6 =>                                       --colour changed gradually
                                if href_count<515 then
                                        RGB_add:=RGB_add+1;
                                        VGA_R<=RGB_add(3 downto 0);
                                        VGA_G<=RGB_add(7 downto 4);
                                        VGA_B<=RGB_add(11 downto 8);
                                else RGB_add:=(others=>'0');
                                end if;
                        when others =>                                  --horizontal colour bar
                                if href_count<155 then
                                        VGA_R<="1111";
                                        VGA_G<="1000";
                                        VGA_B<="0100";
                                elsif href_count<270 then
                                        VGA_R<="1000";
                                        VGA_G<="0000";
                                        VGA_B<="1111";
                                elsif href_count<390 then
                                        VGA_R<="1111";
                                        VGA_G<="1111";
                                        VGA_B<="0000";
                                else
                                        VGA_R<="1111";
                                        VGA_G<="0000";
                                        VGA_B<="1111";
                                end if;
                        end case;
                end if;
        else VGA_R<="0000";                                             --else must output 0,or there won't any
picture on the screen
                VGA_G<="0000";
                VGA_B<="0000";
        end if;
    end if;
  end process;
end;
```

附录 2

```
#include "stdio.h"
unsigned char change(unsigned char t)
{
	unsigned char ch;
	if (t<=9)
	{
		ch=48+t;
		return(ch);
	}
	else
	{
		ch=t-10+97;
		return(ch);
	}
}
void main()
{
	FILE * fpr,* fpw;
	//char fr[3],fw[2],temp[3];
	int i,j,k;
	unsigned char fr[6],fw[6],temp[6];
	char name[20];
	printf("please input the file's name:");
	scanf("%s",name);
	fpr=fopen(name,"rb");
	fpw=fopen("picture16.txt","wt+");
	printf("Dealing\n");
	for(i=0;i<480;i++)
	{
	fseek(fpr,0x36+(479-i)*3*640,0);
		for(j=0;j<320;j++)
		{
			fread(fr,sizeof(char),6,fpr);
			temp[0]=fr[0]>>4;// blue
			temp[1]=fr[1]>>4;// green
			temp[2]=fr[2]>>4;// red
			temp[3]=fr[3]>>4;// blue
			temp[4]=fr[4]>>4;// green
			temp[5]=fr[5]>>4;// red
			for(k=0;k<6;k++)
			fw[k]=change(temp[k]);
			fwrite(fw,sizeof(char),6,fpw);
		}
	}
	fclose(fpw);
	fclose(fpw);
	printf("finished!\n");
}
```