Funny Soundboard

W4840 Project Report
Xiaoliang (Lee) Zhu, Dawei Liu

May 13, 2011

1 Goal

We planned to create an embedded system that plays back various sound clips when the user presses different buttons. The purpose is to generate laughs, and the potential application is for the system to be placed in theme parks, restaurants, or anyplace that people wait in line.

2 Description

Originally we intended for the system to display a grid of buttons along with text and animation representing the theme of the soundboard. There is a mouse connected to the device and a cursor outputted on screen. When a user clicks a button the corresponding sound file is output to the speakers. Figure 1 Shows a vision of the original plan. The final product consisted of a DE2

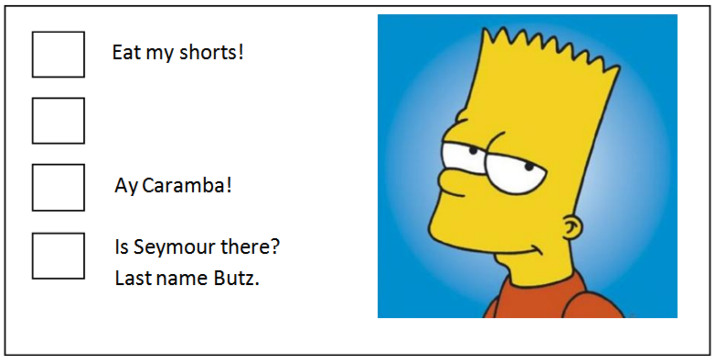


Figure 1: Original Proposal of Design

board with external mouse, speaker, and an SD card. The system reads images and audio files from the SD card and output to the screen and speakers. To simply implementation, we removed the grid of buttons and the text, and made the animation full screen at 640 x 480 pixels. For the demonstration we chose an animation consisting of 3 frames, and used a circular cursor to represent the mouse. When the user clicks the mouse, one of 7 audio clips is played through the speakers in stereo. Figure 2 shows a screen shot of the demonstration.

The number of frames for the animation can easily be increased to more than 10 frames at the current resolution and color depth. The color depth we used for the system is 16 -bits, which imply the demonstration system can display cinematic and life-like images. If the user chooses to decrease the color depth and animation resolution, the length of the animation can

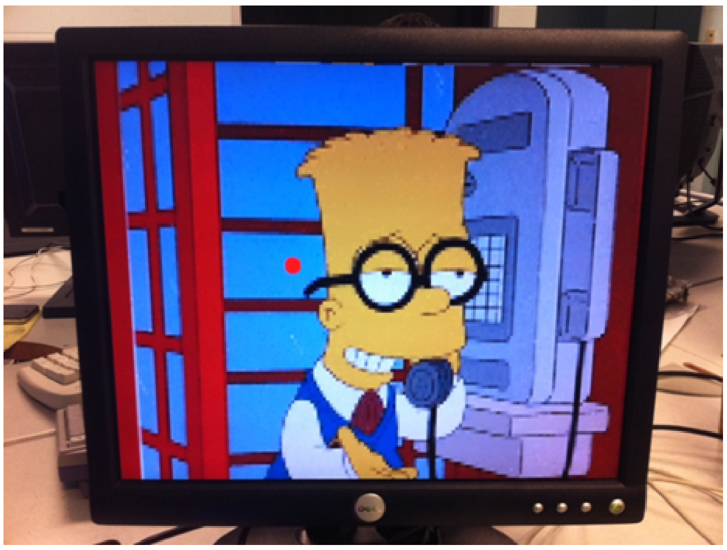


Figure 2: Demonstration System Screenshot

be increased substantially. The audio played by the system is stored on the SD card, and the only limit to the number of audio files is the capacity of the SD card.

3 Implementation

The system is based on the Altera Cyclone II FPGA and NIOS II microprocessor. A block diagram is shown in Figure 3. Important to note that while all the components are Avalon slaves, the SD Card controller is mostly software based, and that the VGA controller has an Avalon master interface that can access the SDRAM in the system directly. The corresponding system architecture is shown in Figure 4. Circles in the picture represent software driven components, and arrows represent data flow.

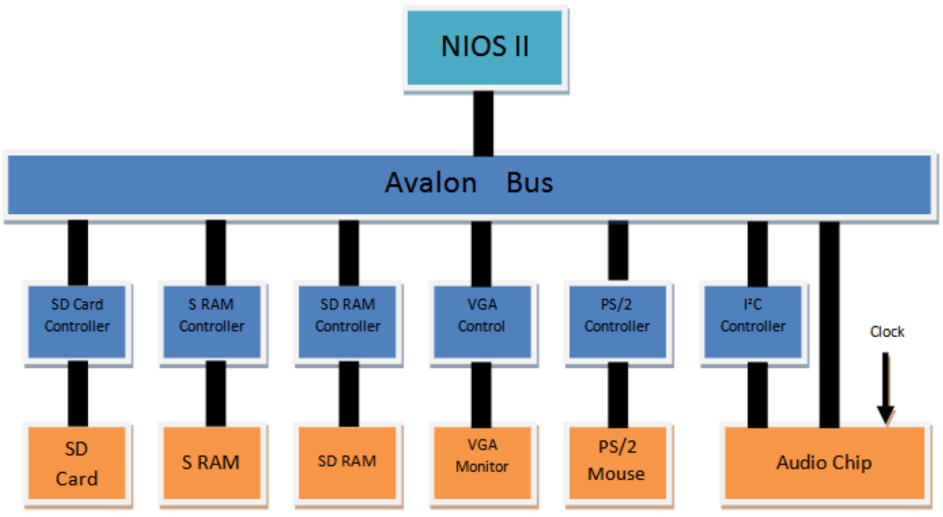


Figure 3: Block Diagram

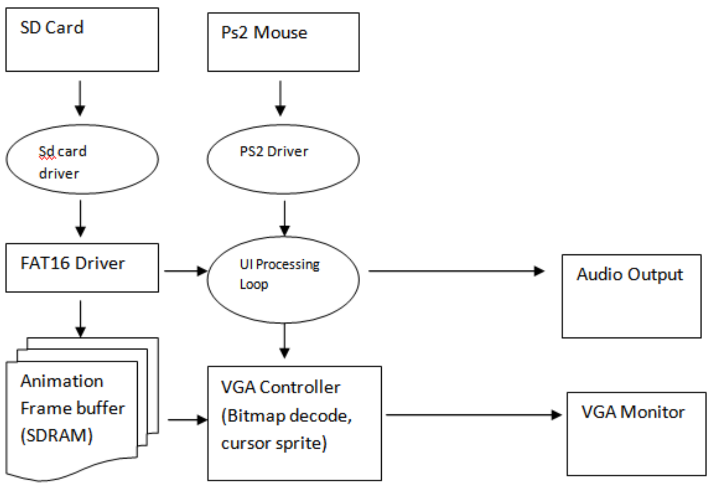


Figure 4: System Architecture

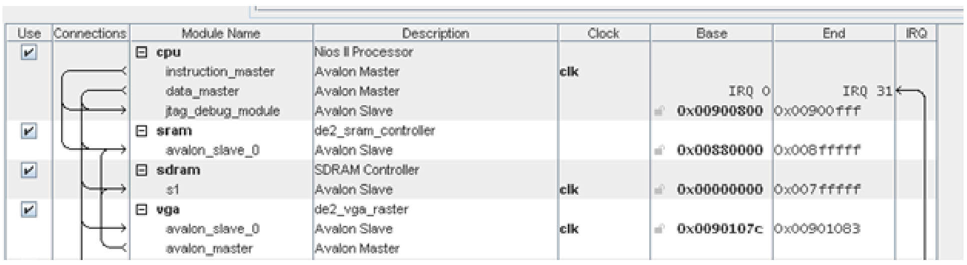
3.1 VGA Controller

The VGA controller used in the project is based on the simple VGA controller used in lab3. The raster signal generation parts from that implementation and the added circular sprite in lab3 was kept intact. The major additions were an Avalon Master interface, dual port memory to buffer pixel data, and bitmap decoding circuitry.

The reason for using the Avalon Master interface is because we wanted to access the frame in SDRAM directly. The performance of the frame buffer is critical to animation performance, thus we needed to avoid overhead with accessing frame buffer with CPU then sending the data over to the VGA controller.

The Avalon Master interface is simple to use. One would add signals named avm_read_master_read, avm_read_master_address, avm_read_master_readdata, etc, similar to *chipselect*, write and data in the Avalon Slave interface. More signals are need to enable pipelining and burst reads, but 4 signals are all you need for the simplest read only master interface. When you add these signals and update the component in SOPC builder, an Avalon Master interface should show up in the new component, and one can edit wires to and from it very similar to other Avalon Slave interfaces.

Figure 5 shows the screen shots from SOPC builder with Avalon Master Interfaces added.



Use	Connections	Module Name	Description	Clock	Base	End	IRQ
✓		⊟ **cpu**	Nios II Processor				
		instruction_master	Avalon Master	**clk**			
		data_master	Avalon Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Slave		**0x00900800**	0x00900fff	
✓		⊟ **sram**	de2_sram_controller				
		avalon_slave_0	Avalon Slave		**0x00880000**	0x008fffff	
✓		⊟ **sdram**	SDRAM Controller				
		s1	Avalon Slave	**clk**	**0x00000000**	0x007fffff	
✓		⊟ **vga**	de2_vga_raster				
		avalon_slave_0	Avalon Slave	**clk**	**0x0090107c**	0x00901083	
		avalon_master	Avalon Master				

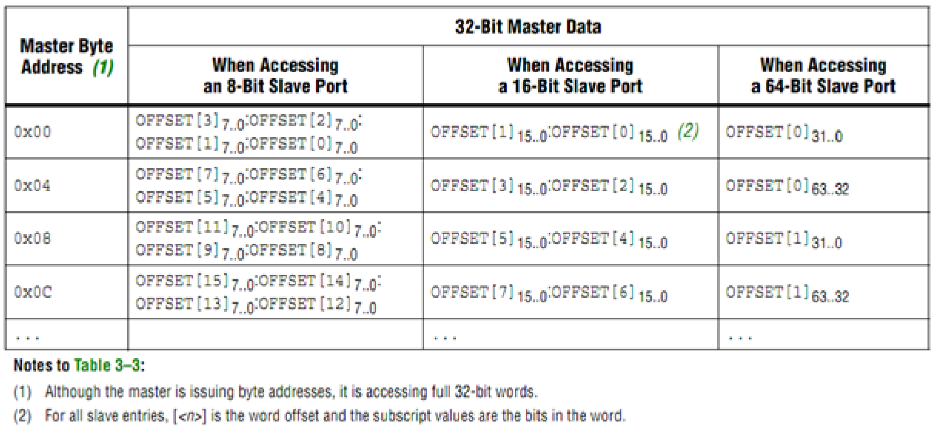
Figure 5: SOPC builder with Avalon Master Interfaces

To initialize a read, the master would drive the address bits onto the address bus, then assert the Avalon read signal. This sends the request to the Avalon arbitration fabric, and when the bus is free, the address will be sent to the peripheral and the read performed. When data is available, the *avm_read_master_wait_request* signal is deasserted back to the Avalon bus, and typically this signal is used as the data latch or clock enable.

The interval between asserting Avalon read signal and when data is valid is variable, thus to use the Avalon master interface a state machine is required to wait appropriately for the data. Different forms of the state machine are required to accommodate features such as pipeline and burst. Very good templates exist and can be found in the reference section.

In the VGA controller we use the simplest state machine, with no pipeline and no burst use. Typically a FIFO is used in conjunction with the state machine to save data, but this made the code more complicated for raster access. Thus a dual port memory block is used to save data from the SDRAM. At the start of each raster scan line, we start reading from the SDRAM, saving each pixel into the dual port memory. When the horizontal raster reaches the active screen area, it reads out the pixel information stored in the memory block.

It is very important to note that the Avalon Data Bus is 32-bits wide, and when it reads from 16-bit native device, such as SDRAM on the DE2, it will perform consecutive reads on adjacent addresses. Thus one must be careful in the software and the hardware to make sure the data words are matched up and no cycles are wasted accessing duplicate data. See Figure 6. In the VGA controller we take advantage of the 32-bit data access and in effect access two pixels at a time. Thus there are actually two dual-port buffers in use in the VGA controller, one for saving even pixels and one for saving odd pixels.



Master Byte Address (1)	32-Bit Master Data		
	When Accessing an 8-Bit Slave Port	When Accessing a 16-Bit Slave Port	When Accessing a 64-Bit Slave Port
0x00	OFFSET[3]$_{7..0}$:OFFSET[2]$_{7..0}$: OFFSET[1]$_{7..0}$:OFFSET[0]$_{7..0}$	OFFSET[1]$_{15..0}$:OFFSET[0]$_{15..0}$ (2)	OFFSET[0]$_{31..0}$
0x04	OFFSET[7]$_{7..0}$:OFFSET[6]$_{7..0}$: OFFSET[5]$_{7..0}$:OFFSET[4]$_{7..0}$	OFFSET[3]$_{15..0}$:OFFSET[2]$_{15..0}$	OFFSET[0]$_{63..32}$
0x08	OFFSET[11]$_{7..0}$:OFFSET[10]$_{7..0}$: OFFSET[9]$_{7..0}$:OFFSET[8]$_{7..0}$	OFFSET[5]$_{15..0}$:OFFSET[4]$_{15..0}$	OFFSET[1]$_{31..0}$
0x0C	OFFSET[15]$_{7..0}$:OFFSET[14]$_{7..0}$: OFFSET[13]$_{7..0}$:OFFSET[12]$_{7..0}$	OFFSET[7]$_{15..0}$:OFFSET[6]$_{15..0}$	OFFSET[1]$_{63..32}$
. . .		. . .	. . .

Notes to Table 3–3:

(1) Although the master is issuing byte addresses, it is accessing full 32-bit words.

(2) For all slave entries, [<n>] is the word offset and the subscript values are the bits in the word.

Figure 6: Dynamic Bus Sizing Master-to-Slave Address Mapping

3.1.1 Bitmap Decoding

We decided to use 16-bit bitmap images because of it is a good combination between size and color accuracy. A 16-bit bitmap is essentially an array of pixels, with each color represented by a block of 5 bits in the 16-bit word for a single pixel. The image file itself consists of a header section followed by a data section. The data section is in reverse raster order, which means the first pixel in the array is the lower leftmost pixel in the image.

The bitmap decoding was originally planned to be done in software, but then we realized that since the BMP format is very simple, it is a lot more efficient to just transfer the image data to SDRAM and do any decoding necessary in hardware. This is accomplished by mapping the 5-bit blocks in each pixel to the VGA DAC output pins. Scaling from the 5 bit resolution of the BMP image to the 10bit resolution of the DAC by shifting the bits left. On an ordinary monitor this loss of color resolution is negligible. The reverse raster order is adjusted by adding the appropriate offsets and conditions in the SDRAM address translation.

3.1.2 Performance issues

Due to lack of pipeline and burst mode support in our Avalon Master implementation, we incurred a significant performance penalty to access SDRAM using random access mode. The cycle count from address valid to data available for our implementation is approximately 10-12 clocks. The latency would be worst if there was bus arbitration involved.

The latency means that during the time to scan a line on the screen, the data needed for the line will not have all been loaded. What happens then is the raster will display all the newly loaded data, and then it will continue to display the data from the previous line for the following pixels. Since it take 10-12 cycles of the 50 Mhz system clock to read one word of pixel data, and the raster scans at 25 Mhz, during one line scan time we can read about 20% of the pixels needed. Reading in 32bit increments doubles that rate, so every 2.5 displayed lines will show the data intended for one line. The pixilation and data jumping behavior is shown in Figure 7. During development the origins of this effect was hidden for quite some time. When the number of pixels to display per row is reduced, the detrimental effect goes away. It was possible to get up to 100 pixels per row displayed perfectly.

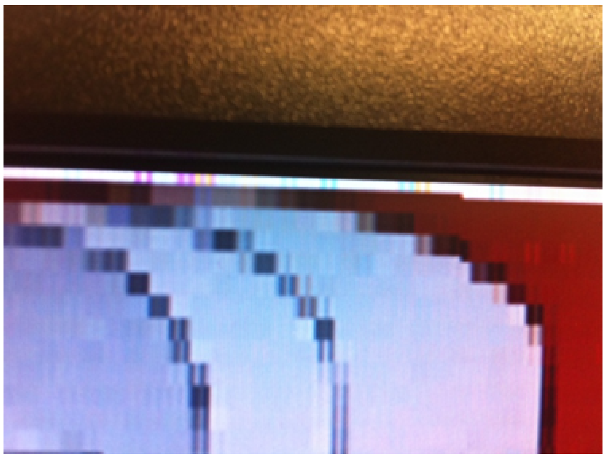


Figure 7: Pixilation and line glitch condition due to DRAM latency

Figure 7 shows the pixilation and line glitch condition due to DRAM latency. The white pixels at the top are supposed to be red, but they are white due to undefined data when the raster scan accessed uninitialized memory. The "step" near the upper left is when raster line 3 started reading the undefined data. This effect propagates to the rest of the picture and causes more pixilation.

3.2 FAT16 Driver

To access the Sd card file system we used a FAT16 driver implemented in software. We did not write the driver but ported a driver under free education license. The driver was originally developed for EECS5x embedded systems class taught at Caltech. Please see individual file for copy right details. This drive was chosen because it is very simple and well commented. Using this simple driver instead of a more comprehensive driver freed us from overheads involved in traditional file access functions.

Using this driver we can open and read any file. Long file names support is included. The reading function assumes files are in continuous blocks, and thus will not support files having defragmentation. The porting process took longer than expected because there were very slight differences in the SD card FAT structure and a FAT structure on an IDE. I had to debug the FAT file system initialization only to find out that the errors were cause by a constant being off by 1. Once the code was ported it worked very well in our context.

3.3 SRAM and SDRAM

The memory used in the project consisted of SRAM for instruction and SDRAM for data. We used the default SRAM and SDRAM controllers in SOPC builder, and then followed the instructions for interfacing to the SDRAM from the Altera literature linked on the class website.

3.4 SD Card

There are four pins of SD card interface: SD_CMD, SD_DAT, SD_CLK and SD_DAT3. We assign SD_DAT3 high to make sure SD card is always selected, and we use software control to make sure it works at SPI mode. The DE2 board has prepared the hardware circuit very well, so everything relating to protocol, like initialization, read and write of SD card is all realized in software.

Since there is no existing SD card core for Quartus 7.2 provided for DE2 board, we adopt the scheme used in SD Card Music Player in DE2 reference design. It requires adding 3 one-bit PIO in SOPC builder, of which SD_CMD and SD_DAT are set to be bidirectional pins, and SD_CLK the output. At the first stage of project, we simply follow the software control from the reference until we feel necessary to optimize data timing for audio output.

The performance of our audio output is actually affected by the software controlled SD card. Data read with uncertain timing from SD card results in jitter during sampling and thus distortion at the CODEC output. To resolve it, we come up with a few methods: decrease the latency during SD card reading, optimize the audio bit-stream sequence by buffering before sending them to DAC FIFO, or there are ways of walking it around by changing to lower sampling rate source files or modify the CODEF configuration. The reference design utilizes the fast CPU working at 100MHz in NIOS in pursuit of lower latency at SD card, but we also need to balance the clock and timing among some other components like VGA and SDRAM.

3.5 Mouse

The PS/2 protocol is relatively easy to understand after the fundamental exercise in lab2. But to make the mouse work well, we also need to understand how the ps/2 port is interfaced with FPGA. Thanks to Altera_UP_Avalon_PS2 core and its documents, what we need to do is just generating NIOS system by adding the above core into SOPC builder and then realize all its functions in software.

The mouse works in Stream Mode by default but we prefer Remote Mode because it enables us to acquire its movement regularly instead of waiting for interrupt to handle incoming data packets. Another trick is to set appropriate time interval for the three consecutive packets before retrieving data from data register. This could help avoid duplicate or missing data of mouse movement. Then we integrate the mouse capture part with VGA display so that it behaves on the screen like a real mouse. At last we optimize the codes to make the trace look smooth and employ a beautiful heart-shape cursor for fun.

3.6 Audio

Fancy audio output at the speaker depends on good understanding of CODEC principals as well as proper interfacing of WM8731 CODEC chip to FPGA. The interface is accomplished with two functions, configuration and clock generation. The configuration to wm8731 chip is done by writing to registers according to sampling rate and chip clock frequency, via an I2C interface from FPGA. We can realize this either by using reference file in lab2 or utilize a dedicated Altera core directly. Then based on configuration channel and sampling rate, certain clocks are generated and the stream is buffer in FIFO with timing alignment, which function is usually built up in an audio core. The audio core essentially interacts with the Audio CODEC chip on DE2 board and provides an interface for audio input and output.

There are two optional Audio cores for our project: Altera_UP_Avalon_Audio and Audio_DAC_FIFO. The former is the formal core version provided by Altera with complete interfacing function, and it can work together with another Altera IP core named Audio/Video Configuration Core to configure audio chip automatically. The latter is a simple version of the latter and features only DAC support, i.e. it assumes no audio line in or microphone. Nevertheless in its setting tab in SOPC Builder, we are still able to set critical parameters for CODEC like reference clock, sampling rate and bit width. Since we finally adopt the simpler core, we employ the two I2C-based files from lab 2 for CODEC configuration.

For both Audio cores, an individual reference clock is required to create certain necessary clock signals. It is firstly inverted and fed to CODEC as chip clock, then divided by a factor of 4 to generate bit stream clock for each bit of audio stream stored in FIFO. According to sampling rate, it is further divided down to channel clock indicating whether current bit is for left or right channel. This reference clock can usually be implemented by one PLL using MegaWizard in Quartus. For our project, we feed 18.4MHz (384fs) clock to audio core. Though not very accurate, it will suffice for audio demonstration. We also want to point out that there is another Altera IP core named Boards External Interface for DE2 /DE1 boards that doesn't behave as it claims. It is supposed to generate up to 6 kinds of the reference clock we mentioned above, but according to our tryout, there is actually no such clock output seeing from the HDL file generated by SOPC Builder. But it would be easy to be replaced by a DLL.

4 Initialization and user interface software

The project is controlled by a main program written in Nios2 C. At the start of the program it initializes the SD card into the appropriate mode, follow by initializing the FAT16 file system, and then it transfers all the animation frame files into SDRAM. Currently we do not have configuration file to specify the names of the animation frames.

Next, the software enables the PS2 mouse interface and begins loop processing of mouse input. It transfers the mouse coordinates to the VGA controller in the same way as the bouncing ball in lab3. When the user clicks a button, the main loop calls the audio output function, which blocks until audio data finishes playing.

5 Task division

Dawei and Lee worked jointly to setup the initial project architecture (including SRAM, SDRAM, and pins for SD card), and jointly ported the SD card interface and FAT16 driver for the project. Next Dawei focused on the mouse input processing, cursor tracking, and audio output capabilities. Lee worked on the VGA controller, adding Avalon Master interface, pixel data buffers, and bitmap pixel decoding. We also worked jointly on the initialization and main loop.

6 Advice for future groups

6.1 From Lee Zhu

Porting code from example projects and other people's code is very time consuming. Even for very well commented examples you will likely spend much time debugging. It is better to get a good understanding of what you're trying to do, start from a clean slate, and only consult examples as reference.

Integration is time consuming. It is worthwhile to sync up project designs along the way and not wait until the end when differences are large.

Build the system up slowly and carefully, this way you will be in control of VHDL's complexities. Have test cases on hand as you build up your design.

If you find Nios-2 is crashing, instruction memory is probably getting over written. This is a problem for multi-master systems. So it is good to connect other masters only to the memory they absolutely need to access.

6.2 From Dawei Liu

Life is short, so be wise to spend your time. So always try to understand the entire principal before starting to write codes, and always check your codes before compiling your design, because waiting will take up most of your project time. If you have problems, know how to find the answer.

7 Reference

1. http://www.cs.columbia.edu/ sedwards/classes/2008/4840/designs/Pelmanism.pdf
2. http://www.sdcard.org/developers/tech/sdcard/pls/simplified_specs/Part_1_Physical _Layer_Simplified_Specification_Ver3.01_Final_100518.pdf
3. http://home.teleport.com/ brainy/fat16.htm
4. http://wolverine.caltech.edu/eecs52/projects/188mp3/188mp3.htm
5. http://www.cs.columbia.edu/ sedwards/classes/2011/4840/ps2-keyboard.pdf
6. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

Attachments: Codes

Main loop: de2_vga_raster.vhd

```
-------------------------------------------------------------------------------
--
-- Simple VGA raster display
--
-- Stephen A. Edwards
-- sedwards@cs.columbia.edu
--
-- Edited by Xiaoliang (Lee) Zhu and Dawei Liu
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity de2_vga_raster is

  port (
    -- Avalon slave interface for mouse and movie addresses
    reset : in std_logic;
    clk   : in std_logic;                    -- Should be 25.125 MHz
    chipselect : in  std_logic;
    write      : in  std_logic;
    address : in unsigned(0 downto 0);       -- don't really matter as we only have 1 thing to update
    writedata  : in unsigned(31 downto 0);

    -- Create our Avalon Master to access SRAM and SDRAM
    avm_read_master_read : out std_logic;
    avm_read_master_address : out std_logic_vector (31 downto 0);  -- use the same 32 bit addressing a
    avm_read_master_readdata : in std_logic_vector (31 downto 0);  -- the avalon bus always reads 32 b
    avm_read_master_waitrequest : in std_logic;

    -- Outputs for Actual VGA
    VGA_CLK,                        -- Clock
    VGA_HS,                         -- H_SYNC
    VGA_VS,                         -- V_SYNC
    VGA_BLANK,                      -- BLANK
    VGA_SYNC : out std_logic;       -- SYNC
    VGA_R,                          -- Red[9:0]
    VGA_G,                          -- Green[9:0]
    VGA_B : out unsigned(9 downto 0) -- Blue[9:0]
    );

end de2_vga_raster;
```

```
architecture rtl of de2_vga_raster is
  -- declare LineBuffer component
  -- the buffer (dual port RAM) component was generated using the Quartus II MegaWizard Plug-In Manage
  -- Output is not registered
  component LineBuffer
  port (
            clock : IN STD_LOGIC ;
            data : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            rdaddress  : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
            wraddress : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
            wren : IN STD_LOGIC;
            q     : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
  );
  end component;


  -- Video parameters
  constant HTOTAL       : integer := 800;
  constant HSYNC        : integer := 96;
  constant HBACK_PORCH  : integer := 48;
  constant HACTIVE      : integer := 640;
  constant HFRONT_PORCH : integer := 16;

  constant VTOTAL       : integer := 525;
  constant VSYNC        : integer := 2;
  constant VBACK_PORCH  : integer := 33;
  constant VACTIVE      : integer := 480;
  constant VFRONT_PORCH : integer := 10;

  -- Used to define the active screen area
  constant HSTART : integer := HSYNC + HBACK_PORCH;
  constant HEND   : integer := HSTART + HACTIVE;
  constant VSTART : integer := VSYNC + VBACK_PORCH;
  constant VEND   : integer := VSTART + VACTIVE;
  constant RADIUS_SQUARE : integer := 10 * 10;
  constant START_X : integer := HSYNC + HBACK_PORCH + (HACTIVE / 2);
  constant START_Y : integer := VSYNC + VBACK_PORCH - 1 + (VACTIVE / 2);
  constant SDRAM_BASE : unsigned (31 downto 0) := x"0000_0000";  -- SOPC dram address
  constant BMP_OFFSET : unsigned (31 downto 0) := x"0000_0036"; -- Standard 16bit bmp data offset

  constant Frame1_BASE : unsigned (31 downto 0) := x"0009_6000";  -- SOPC dram address
  constant Frame2_BASE : unsigned (31 downto 0) := x"0012_C000";  -- SOPC dram address

  -- Signals for the video controller
  signal clk25 : std_logic := '0';
  signal Hcount : unsigned(9 downto 0);  -- Horizontal position (0-800)
  signal Vcount : unsigned(9 downto 0);  -- Vertical position (0-524)
  signal EndOfLine, EndOfField : std_logic;
  signal vga_hblank, vga_hsync,
         vga_vblank, vga_vsync : std_logic;  -- Sync. signals

  signal circle : std_logic;  -- circle area

  -- Signals to communicate from outside
  signal center_x : unsigned(9 downto 0);
  signal center_y : unsigned(9 downto 0);
  signal pixel_base : unsigned(31 downto 0);     -- SOPC dram address
  signal pixel_offset : unsigned (31 downto 0);  -- Standard 16bit bmp data offset
  signal frame_count : unsigned(5 downto 0);     -- Use this to count to 20 frames -> 3 frames per sec

  -- state machine for Avalon master
  type read_states_T is (idle, running, stopping);
  signal read_state : read_states_T;

  -- extra signal for read master
  signal temp_read_address : std_logic_vector (31 downto 0);
  signal words_read : std_logic_vector (16 downto 0);            -- tracks the words read

    -- instiantiaton for lineBuffer
  signal buffer_write : std_logic;
  signal buffer_waddress, buffer_raddress : std_logic_vector(9 DOWNTO 0);

  -- signal to save color information
  signal bmp_pixel, bmp_pixel_even, bmp_pixel_odd : std_logic_vector(15 downto 0);

-------------------------------------------------------------------------------
-- Buffer instantiation
-------------------------------------------------------------------------------
begin
even_line_buffer: LineBuffer  -- 1024 WORD Buffer
port map (
      clock     => clk,
      data      => avm_read_master_readdata(15 downto 0),
      rdaddress => buffer_raddress,
      wraddress => buffer_waddress,
      wren      => buffer_write,
      q         => bmp_pixel_even
);

odd_line_buffer: LineBuffer  -- 1024 WORD Buffer
port map (
      clock     => clk,
      data      => avm_read_master_readdata(31 downto 16),
      rdaddress => buffer_raddress,
      wraddress => buffer_waddress,
      wren      => buffer_write,
      q         => bmp_pixel_odd
);

-- Match up the buffers outputs
bmp_pixel <= bmp_pixel_even when Hcount(0) = '0' else bmp_pixel_odd;


-------------------------------------------------------------------------------
-- Avalon master state machine
-------------------------------------------------------------------------------
  AvalonInit : process (clk)
  begin
    if reset = '1' then
       read_state <= idle;
       temp_read_address <= (others => '0');
       words_read <= (others => '0');
    else if rising_edge(clk) then
      case read_state is
        --IDLE, sit and wait until we can start loading data
        when idle =>
          -- Load the pixels at the start of each line.
          if (Hcount = 0) and (Vcount >= VSTART) and ((Vcount < VEND + 1)) then
          read_state <= running;
            temp_read_address <= STD_LOGIC_VECTOR(pixel_base + pixel_offset + ((VEND - Vcount - 1) * H
            words_read <= (others => '0');
          end if;

        --RUNNING
        --Hold all signals constant until waitrequest is not active
        when running =>
          if avm_read_master_waitrequest /= '1' then
              temp_read_address <= temp_read_address + 2;  -- Update address DMA style
              words_read <= words_read + 1;
              if words_read = (HACTIVE / 2) then  -- a whole lines worth in double words
                 read_state <= stopping;
              end if;

          end if;

        -- STOPPING, just a cycle delay for consistance, will not hurt performance
        when stopping =>
            read_state <= idle;
      end case;
    end if;
  end if;
  end process;
  -- combinational signal for avalon master
  avm_read_master_read <= '1' when read_state = running else '0';
  avm_read_master_address <= temp_read_address;

  -- simply write data into the buffer as it comes in (read asserted and waitrequest not active)
  buffer_waddress(9 downto 0) <= words_read(9 downto 0);
  buffer_write <= '1' when read_state = running and avm_read_master_waitrequest = '0' else '0';

  -- the read address is the same for either odd or even rows.
  buffer_raddress <= STD_LOGIC_VECTOR((Hcount - HSTART) srl 1);

-------------------------------------------------------------------------------
-- Software interfaces
-------------------------------------------------------------------------------
  SetCenter : process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
         center_x <= TO_UNSIGNED(START_X, 10);
         center_y <= TO_UNSIGNED(START_Y, 10);
      else
         if chipselect = '1' then
            if write = '1' then
                  center_x <= writedata(9 downto 0);
                  center_y <= writedata(25 downto 16);
               end if;
            end if;
         end if;
      end if;
  end process SetCenter;

-------------------------------------------------------------------------------
-- Animation generation
-------------------------------------------------------------------------------
  MoveFrame : process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        pixel_base <= SDRAM_BASE;
        pixel_offset <= BMP_OFFSET;
        frame_count <= (others => '0');
      else
        if (Hcount = 0) and (Vcount = 0) then
   frame_count <= frame_count + 1;
end if;
        if (frame_count = 0) then
           pixel_base <= SDRAM_BASE;
        end if;
```

```
if (frame_count = 20) then
           pixel_base <= Frame1_BASE;
        end if;
        if (frame_count = 40) then
           pixel_base <= Frame2_BASE;
        end if;
        if (frame_count > 60) then
            frame_count <= (others => '0');
        end if;
       end if;
    end if;
  end process MoveFrame;

-------------------------------------------------------------------------------
-- Circle generator, do it in four quadrants
-------------------------------------------------------------------------------

   CircleGen : process (clk25)
   variable x_distance, y_distance : unsigned(9 downto 0);  -- use for computation
   begin
     if rising_edge(clk25) then
        if reset = '1' then
           circle <= '0';
        else
       if Hcount < center_x then  -- left quadrants
              x_distance := center_x - Hcount;
           else                     -- right quadrants
              x_distance := Hcount - center_x;
           end if;

           if Vcount < center_y then -- top quadrants
              y_distance := center_y - Vcount;
           else                    -- bottom quadrants
              y_distance := Vcount - center_y;
           end if;

           if RADIUS_SQUARE > (x_distance * x_distance) + (y_distance * y_distance) then
              circle <= '1';
           else
              circle <= '0';
           end if;
         end if;
       end if;
   end process CircleGen;


-------------------------------------------------------------------------------
-- VGA related
```

```
-------------------------------------------------------------------------------
  -- Turn down the clock
  process (clk)  -- Let's try running the system at 25 MHz and see if it works
  begin
    if rising_edge(clk) then
      clk25 <= not clk25;
    end if;
  end process;

  -- Horizontal and vertical counters

  HCounter : process (clk25)
  begin
    if rising_edge(clk25) then
      if reset = '1' then
        Hcount <= (others => '0');
      elsif EndOfLine = '1' then
        Hcount <= (others => '0');
      else
        Hcount <= Hcount + 1;
      end if;
    end if;
  end process HCounter;

  EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

  VCounter: process (clk25)
  begin
    if rising_edge(clk25) then
      if reset = '1' then
        Vcount <= (others => '0');
      elsif EndOfLine = '1' then
        if EndOfField = '1' then
          Vcount <= (others => '0');
        else
          Vcount <= Vcount + 1;
        end if;
      end if;
    end if;
  end process VCounter;

  EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';

  -- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK
  HSyncGen : process (clk25)
  begin
    if rising_edge(clk25) then
      if reset = '1' or EndOfLine = '1' then
        vga_hsync <= '1';
      elsif Hcount = HSYNC - 1 then
        vga_hsync <= '0';
      end if;
    end if;
  end process HSyncGen;

  HBlankGen : process (clk25)
  begin
    if rising_edge(clk25) then
      if reset = '1' then
        vga_hblank <= '1';
      elsif Hcount = HSYNC + HBACK_PORCH then
        vga_hblank <= '0';
      elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
        vga_hblank <= '1';
      end if;
    end if;
  end process HBlankGen;

  VSyncGen : process (clk25)
  begin
    if rising_edge(clk25) then
      if reset = '1' then
        vga_vsync <= '1';
      elsif EndOfLine ='1' then
        if EndOfField = '1' then
          vga_vsync <= '1';
        elsif Vcount = VSYNC - 1 then
          vga_vsync <= '0';
        end if;
      end if;
    end if;
  end process VSyncGen;

  VBlankGen : process (clk25)
  begin
    if rising_edge(clk25) then
      if reset = '1' then
        vga_vblank <= '1';
      elsif EndOfLine = '1' then
        if Vcount = VSYNC + VBACK_PORCH - 1 then
          vga_vblank <= '0';
        elsif Vcount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
          vga_vblank <= '1';
        end if;
      end if;
    end if;
  end process VBlankGen;

  -- Registered video signals going to the video DAC
  VideoOut: process (clk25, reset)
  variable blue_bits, green_bits, red_bits : unsigned(9 downto 0);  -- use to read pixels from 16 bit
  begin
    if reset = '1' then
      VGA_R <= "0000000000";
      VGA_G <= "0000000000";
      VGA_B <= "0000000000";
    elsif clk25'event and clk25 = '1' then
      if circle = '1' then
        VGA_R <= "1111111111";
        VGA_G <= "0000000000";
        VGA_B <= "0000000000";
      elsif vga_hblank = '0' and vga_vblank ='0' then
        -- For 16bit bmp, bits are X-RRRRR-GGGGG-BBBBB, 5 bits small endian for red green blue
        -- Convert from 5 bit color to 10 bit for DAC by shifting left (multiply by 32)
        VGA_R(9 downto 5) <= UNSIGNED(bmp_pixel(14 downto 10));
        VGA_R(4 downto 0) <= "00000";
        VGA_G(9 downto 5) <= UNSIGNED(bmp_pixel(9 downto 5));
        VGA_G(4 downto 0) <= "00000";
        VGA_B(9 downto 5) <= UNSIGNED(bmp_pixel(4 downto 0));
        VGA_B(4 downto 0) <= "00000";
      else
        VGA_R <= "0000000000";
        VGA_G <= "0000000000";
        VGA_B <= "0000000000";
      end if;
    end if;
  end process VideoOut;

  VGA_CLK <= clk25;
  VGA_HS <= not vga_hsync;
  VGA_VS <= not vga_vsync;
  VGA_SYNC <= '0';
  VGA_BLANK <= not (vga_hsync or vga_vsync);

end rtl;
```

Main loop: hello_world.c

```
#include <io.h>
#include <system.h>
#include <stdio.h>
#include "SD_Card.h"
```

```
//--------------------------------------------------------------------------
BYTE SD_card_init(void)
{
    BYTE x,y;
    SD_CMD_OUT;
    SD_DAT_IN;
    SD_CLK_HIGH;
    SD_CMD_HIGH;
    SD_DAT_LOW;
    read_status=0;
    for(x=0;x<40;x++)
    NCR;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd0[x];
    y = send_cmd(cmd_buffer);
    do
    {
      for(x=0;x<40;x++);
      NCC;
      for(x=0;x<5;x++)
      cmd_buffer[x]=cmd55[x];
      y = send_cmd(cmd_buffer);
      NCR;
      if(response_R(1)>1) //response too long or crc error
      return 1;
      NCC;
      for(x=0;x<5;x++)
      cmd_buffer[x]=acmd41[x];
      y = send_cmd(cmd_buffer);
      NCR;
    } while(response_R(3)==1);
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd2[x];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(2)>1)
    return 1;
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd3[x];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(6)>1)
    return 1;
    RCA[0]=response_buffer[1];
    RCA[1]=response_buffer[2];
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd9[x];
    cmd_buffer[1] = RCA[0];
    cmd_buffer[2] = RCA[1];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(2)>1)
    return 1;
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd7[x];
    cmd_buffer[1] = RCA[0];
    cmd_buffer[2] = RCA[1];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(1)>1)
    return 1;
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd16[x];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(1)>1)
    return 1;
    read_status =1; //sd card ready
    return 0;
}


//-----------------------------------------------------------------------
BYTE get_blocks(UINT32 lba, UINT32 seccnt, BYTE *buff)
{
  BYTE c=0;
  UINT32  i,j;
  for(j=0;j<seccnt;j++)
  {
    {
      NCC;
      cmd_buffer[0] = cmd17[0];
      cmd_buffer[1] = (lba>>15)&0xff;
      cmd_buffer[2] = (lba>>7)&0xff;
      cmd_buffer[3] = (lba<<1)&0xff;
      cmd_buffer[4] = 0;
      lba++;
      send_cmd(cmd_buffer);
      NCR;
    }
    while(1)
    {
      SD_CLK_LOW;
      SD_CLK_HIGH;
      if(!(SD_TEST_DAT))
      break;
    }
    for(i=0;i<512;i++)
    {
      READ_BIT(c);  // bit 1
      READ_BIT(c);  // bit 2
      READ_BIT(c);  // bit 3
      READ_BIT(c);  // bit 4
      READ_BIT(c);  // bit 5
      READ_BIT(c);  // bit 6
      READ_BIT(c);  // bit 7
      READ_BIT(c);  // bit 8
      *buff=c;
      buff++;
    }

    // Toggle clock 16 times
    NCC;
    NCC;
  }
  read_status = 1;  //SD data next in


  return seccnt;
}


//--------------------------------------------------------------------------
BYTE get_blocks_32(UINT32 lba, UINT32 seccnt, unsigned short *buff)
{
  unsigned char cL, cH, hcL, hcH;
  unsigned int c;

  UINT32  i,j;
  int p = 0;
  for(j=0;j<seccnt;j++)
  {
    {
      NCC;
      cmd_buffer[0] = cmd17[0];
      cmd_buffer[1] = (lba>>15)&0xff;
      cmd_buffer[2] = (lba>>7)&0xff;
      cmd_buffer[3] = (lba<<1)&0xff;
      cmd_buffer[4] = 0;
      lba++;
      send_cmd(cmd_buffer);
      NCR;
    }
    while(1)
    {
      SD_CLK_LOW;
      SD_CLK_HIGH;
      if(!(SD_TEST_DAT))
      break;
    }
    for(i=0;i<512;i=i+4)
    {
      // Read each bit explcitly for speed.
      READ_BIT(cL);  // bit 1
      READ_BIT(cL);  // bit 2
      READ_BIT(cL);  // bit 3
      READ_BIT(cL);  // bit 4
      READ_BIT(cL);  // bit 5
      READ_BIT(cL);  // bit 6
      READ_BIT(cL);  // bit 7
      READ_BIT(cL);  // bit 8

      READ_BIT(cH);  // bit 1
      READ_BIT(cH);  // bit 2
      READ_BIT(cH);  // bit 3
      READ_BIT(cH);  // bit 4
      READ_BIT(cH);  // bit 5
      READ_BIT(cH);  // bit 6
      READ_BIT(cH);  // bit 7
      READ_BIT(cH);  // bit 8

      READ_BIT(hcL);  // bit 1
      READ_BIT(hcL);  // bit 2
      READ_BIT(hcL);  // bit 3
      READ_BIT(hcL);  // bit 4
      READ_BIT(hcL);  // bit 5
      READ_BIT(hcL);  // bit 6
      READ_BIT(hcL);  // bit 7
      READ_BIT(hcL);  // bit 8

      READ_BIT(hcH);  // bit 1
      READ_BIT(hcH);  // bit 2
      READ_BIT(hcH);  // bit 3
      READ_BIT(hcH);  // bit 4
      READ_BIT(hcH);  // bit 5
      READ_BIT(hcH);  // bit 6
```

```
      READ_BIT(hcH);  // bit 7
      READ_BIT(hcH);  // bit 8

      c = (hcH << 24 | hcL << 16 | cH << 8 | cL);

      //debug
      //printf("%x %x ", cL, cH);
      //TODO why is there a 24 byte offset?
      IOWR_32DIRECT(buff, p + 56, c);
      p = p + 2;

    }
    for(i=0; i<16; i++)
    {
        SD_CLK_LOW;
        SD_CLK_HIGH;
    }
  }
  read_status = 1;  //SD data next in


  return seccnt;
}

//-----------------------------------------------------------------------------
BYTE get_blocks_audio(UINT32 lba, UINT32 seccnt, unsigned short *buff)
{
  unsigned char cL, cH;
  unsigned short c;
  UINT32  i,j;
  int p = 0;
  for(j=0;j<seccnt;j++)
  {
    {
      NCC;
      cmd_buffer[0] = cmd17[0];
      cmd_buffer[1] = (lba>>15)&0xff;
      cmd_buffer[2] = (lba>>7)&0xff;
      cmd_buffer[3] = (lba<<1)&0xff;
      cmd_buffer[4] = 0;
      lba++;
      send_cmd(cmd_buffer);
      NCR;
    }
    while(1)
    {
      SD_CLK_LOW;
      SD_CLK_HIGH;
      if(!(SD_TEST_DAT))
      break;
    }
    for(i=0;i<256;i++)
    {
      BYTE j;
      for(j=0;j<8;j++)
      {
        SD_CLK_LOW;
        SD_CLK_HIGH;
        cL <<= 1;
        if(SD_TEST_DAT)
        cL |= 0x01;
      }
      for(j=0;j<8;j++)
      {
        SD_CLK_LOW;
        SD_CLK_HIGH;
        cH <<= 1;
        if(SD_TEST_DAT)
        cH |= 0x01;
      }

      c = (cH << 8 | cL);

      //debug
      //printf("%x %x ", cL, cH);

      IOWR_16DIRECT(AUDIO_BASE, 0, c);
      p++;

    }
    for(i=0; i<16; i++)
    {
        SD_CLK_LOW;
        SD_CLK_HIGH;
    }
  }
  read_status = 1;  //SD data next in


  return seccnt;
}

//-------------------------------------------------------------------------
BYTE response_R(BYTE s)
{
  BYTE a=0,b=0,c=0,r=0,crc=0;
  BYTE i,j=6,k;
  while(1)
  {
    SD_CLK_LOW;
    SD_CLK_HIGH;
    if(!(SD_TEST_CMD))
    break;
    if(crc++ >100)
    return 2;
  }
  crc =0;
  if(s == 2)
  j = 17;

  for(k=0; k<j; k++)
  {
    c = 0;
    if(k > 0)                      //for crc culcar
    b = response_buffer[k-1];
    for(i=0; i<8; i++)
    {
      SD_CLK_LOW;
      if(a > 0)
      c <<= 1;
      else
      i++;
      a++;
      SD_CLK_HIGH;
      if(SD_TEST_CMD)
      c |= 0x01;
      if(k > 0)
      {
        crc <<= 1;
        if((crc ^ b) & 0x80)
        crc ^= 0x09;
        b <<= 1;
        crc &= 0x7f;
      }
    }
    if(s==3)
    {
      if( k==1 &&(!(c&0x80)))
      r=1;
    }
    response_buffer[k] = c;
  }
  if(s==1 || s==6)
  {
    if(c != ((crc<<1)+1))
    r=2;
  }
  return r;
}

//--------------------------------------------------------------------------
BYTE send_cmd(BYTE *in)
{
  int i,j;
  BYTE b,crc=0;
  SD_CMD_OUT;
  for(i=0; i < 5; i++)
  {
    b = in[i];
    for(j=0; j<8; j++)
    {
      SD_CLK_LOW;
      if(b&0x80)
      SD_CMD_HIGH;
      else
      SD_CMD_LOW;
      crc <<= 1;
      SD_CLK_HIGH;
      if((crc ^ b) & 0x80)
      crc ^= 0x09;
      b<<=1;
    }
    crc &= 0x7f;
  }
  crc =((crc<<1)|0x01);
  b = crc;
  for(j=0; j<8; j++)
  {
    SD_CLK_LOW;
    if(crc&0x80)
    SD_CMD_HIGH;
    else
    SD_CMD_LOW;
    SD_CLK_HIGH;
    crc<<=1;
  }
  return b;
}
//--------------------------------------------------------------------------
```

SD_Card.h

```
#ifndef   __SD_Card_H__
#define   __SD_Card_H__
//-------------------------------------------------------------------------
//  SD Card Set I/O Direction
#define SD_CMD_IN   IOWR(SD_CMD_BASE, 1, 0)
#define SD_CMD_OUT  IOWR(SD_CMD_BASE, 1, 1)
#define SD_DAT_IN   IOWR(SD_DAT_BASE, 1, 0)
#define SD_DAT_OUT  IOWR(SD_DAT_BASE, 1, 1)
//  SD Card Output High/Low
#define SD_CMD_LOW  IOWR(SD_CMD_BASE, 0, 0)
#define SD_CMD_HIGH IOWR(SD_CMD_BASE, 0, 1)
#define SD_DAT_LOW  IOWR(SD_DAT_BASE, 0, 0)
#define SD_DAT_HIGH IOWR(SD_DAT_BASE, 0, 1)
#define SD_CLK_LOW  IOWR(SD_CLK_BASE, 0, 0)
#define SD_CLK_HIGH IOWR(SD_CLK_BASE, 0, 1)
//  SD Card Input Read
#define SD_TEST_CMD IORD(SD_CMD_BASE, 0)
#define SD_TEST_DAT IORD(SD_DAT_BASE, 0)

//  SD clock macro, toggle clock 8 times
#define NCC \
{ \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
    SD_CLK_LOW;  \
    SD_CLK_HIGH; \
    SD_CLK_LOW;  \
    SD_CLK_HIGH; \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
}

// Command register macro, toggle command register
#define NCR \
{ \
  SD_CMD_IN; \
  SD_CLK_LOW; \
  SD_CLK_HIGH; \
  SD_CLK_LOW; \
  SD_CLK_HIGH; \
}

//  Deserialize macro
#define READ_BIT(in_byte) \
{ \
    SD_CLK_LOW; \
    SD_CLK_HIGH; \
    (in_byte) <<= 1; \
    (in_byte) |= SD_TEST_DAT & 0x0001; \
}

//------------------------------------------------------------------------------
#define BYTE    unsigned char
#define UINT16  unsigned int
#define UINT32  unsigned long
//------------------------------------------------------------------------------
void Ncr(void);
void Ncc(void);
BYTE response_R(BYTE);
BYTE send_cmd(BYTE *);
BYTE SD_read_lba(BYTE *,UINT32,UINT32);
BYTE SD_card_init(void);
//------------------------------------------------------------------------------
BYTE read_status;
BYTE response_buffer[20];
BYTE RCA[2];
BYTE cmd_buffer[5];
const BYTE cmd0[5]   = {0x40,0x00,0x00,0x00,0x00};
const BYTE cmd55[5]  = {0x77,0x00,0x00,0x00,0x00};
const BYTE cmd2[5]   = {0x42,0x00,0x00,0x00,0x00};
const BYTE cmd3[5]   = {0x43,0x00,0x00,0x00,0x00};
const BYTE cmd7[5]   = {0x47,0x00,0x00,0x00,0x00};
const BYTE cmd9[5]   = {0x49,0x00,0x00,0x00,0x00};
const BYTE cmd16[5]  = {0x50,0x00,0x00,0x02,0x00};
const BYTE cmd17[5]  = {0x51,0x00,0x00,0x00,0x00};
const BYTE acmd6[5]  = {0x46,0x00,0x00,0x00,0x02};
const BYTE acmd41[5] = {0x69,0x0f,0xf0,0x00,0x00};
const BYTE acmd51[5] = {0x73,0x00,0x00,0x00,0x00};

BYTE SD_card_init(void);
BYTE get_blocks(UINT32 lba, UINT32 seccnt, BYTE *buff);
BYTE get_blocks_32(UINT32 lba, UINT32 seccnt, unsigned short *buff);
BYTE response_R(BYTE s);
BYTE send_cmd(BYTE *in);

#endif
```

0.1 SD_Card.c

```
#include <io.h>
#include <system.h>
#include <stdio.h>
#include "SD_Card.h"

//---------------------------------------------------------------------------
BYTE SD_card_init(void)
{
    BYTE x,y;
    SD_CMD_OUT;
    SD_DAT_IN;
    SD_CLK_HIGH;
    SD_CMD_HIGH;
    SD_DAT_LOW;
    read_status=0;
    for(x=0;x<40;x++)
    NCR;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd0[x];
    y = send_cmd(cmd_buffer);
    do
    {
      for(x=0;x<40;x++);
      NCC;
      for(x=0;x<5;x++)
      cmd_buffer[x]=cmd55[x];
      y = send_cmd(cmd_buffer);
      NCR;
      if(response_R(1)>1) //response too long or crc error
      return 1;
      NCC;
      for(x=0;x<5;x++)
      cmd_buffer[x]=acmd41[x];
      y = send_cmd(cmd_buffer);
      NCR;
    } while(response_R(3)==1);
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd2[x];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(2)>1)
    return 1;
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd3[x];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(6)>1)
    return 1;
    RCA[0]=response_buffer[1];
    RCA[1]=response_buffer[2];
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd9[x];
    cmd_buffer[1] = RCA[0];
    cmd_buffer[2] = RCA[1];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(2)>1)
    return 1;
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd7[x];
    cmd_buffer[1] = RCA[0];
    cmd_buffer[2] = RCA[1];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(1)>1)
    return 1;
    NCC;
    for(x=0;x<5;x++)
    cmd_buffer[x]=cmd16[x];
    y = send_cmd(cmd_buffer);
    NCR;
    if(response_R(1)>1)
    return 1;
    read_status =1; //sd card ready
    return 0;
}


//---------------------------------------------------------------------------
BYTE get_blocks(UINT32 lba, UINT32 seccnt, BYTE *buff)
{
  BYTE c=0;
  UINT32  i,j;
  for(j=0;j<seccnt;j++)
  {
    {
      NCC;
      cmd_buffer[0] = cmd17[0];
      cmd_buffer[1] = (lba>>15)&0xff;
      cmd_buffer[2] = (lba>>7)&0xff;
      cmd_buffer[3] = (lba<<1)&0xff;
      cmd_buffer[4] = 0;
      lba++;
      send_cmd(cmd_buffer);
      NCR;
    }
    while(1)
    {
      SD_CLK_LOW;
      SD_CLK_HIGH;
      if(!(SD_TEST_DAT))
      break;
    }
    for(i=0;i<512;i++)
    {
      READ_BIT(c);  // bit 1
      READ_BIT(c);  // bit 2
      READ_BIT(c);  // bit 3
      READ_BIT(c);  // bit 4
      READ_BIT(c);  // bit 5
      READ_BIT(c);  // bit 6
      READ_BIT(c);  // bit 7
      READ_BIT(c);  // bit 8
      *buff=c;
      buff++;
    }

    // Toggle clock 16 times
    NCC;
    NCC;
  }
  read_status = 1;  //SD data next in


  return seccnt;
}



//----------------------------------------------------------------------
BYTE get_blocks_32(UINT32 lba, UINT32 seccnt, unsigned short *buff)
{
  unsigned char cL, cH, hcL, hcH;
  unsigned int c;

  UINT32  i,j;
  int p = 0;
```

```
for(j=0;j<seccnt;j++)
{
  {
    NCC;
    cmd_buffer[0] = cmd17[0];
    cmd_buffer[1] = (lba>>15)&0xff;
    cmd_buffer[2] = (lba>>7)&0xff;
    cmd_buffer[3] = (lba<<1)&0xff;
    cmd_buffer[4] = 0;
    lba++;
    send_cmd(cmd_buffer);
    NCR;
  }
  while(1)
  {
    SD_CLK_LOW;
    SD_CLK_HIGH;
    if(!(SD_TEST_DAT))
    break;
  }
  for(i=0;i<512;i=i+4)
  {
    // Read each bit explcitly for speed.
    READ_BIT(cL);  // bit 1
    READ_BIT(cL);  // bit 2
    READ_BIT(cL);  // bit 3
    READ_BIT(cL);  // bit 4
    READ_BIT(cL);  // bit 5
    READ_BIT(cL);  // bit 6
    READ_BIT(cL);  // bit 7
    READ_BIT(cL);  // bit 8

    READ_BIT(cH);  // bit 1
    READ_BIT(cH);  // bit 2
    READ_BIT(cH);  // bit 3
    READ_BIT(cH);  // bit 4
    READ_BIT(cH);  // bit 5
    READ_BIT(cH);  // bit 6
    READ_BIT(cH);  // bit 7
    READ_BIT(cH);  // bit 8

    READ_BIT(hcL);  // bit 1
    READ_BIT(hcL);  // bit 2
    READ_BIT(hcL);  // bit 3
    READ_BIT(hcL);  // bit 4
    READ_BIT(hcL);  // bit 5
    READ_BIT(hcL);  // bit 6
    READ_BIT(hcL);  // bit 7
```

```
      READ_BIT(hcL);  // bit 8

      READ_BIT(hcH);  // bit 1
      READ_BIT(hcH);  // bit 2
      READ_BIT(hcH);  // bit 3
      READ_BIT(hcH);  // bit 4
      READ_BIT(hcH);  // bit 5
      READ_BIT(hcH);  // bit 6
      READ_BIT(hcH);  // bit 7
      READ_BIT(hcH);  // bit 8

      c = (hcH << 24 | hcL << 16 | cH << 8 | cL);

      //debug
      //printf("%x %x ", cL, cH);
      //TODO why is there a 24 byte offset?
      IOWR_32DIRECT(buff, p + 56, c);
      p = p + 2;

    }
    for(i=0; i<16; i++)
    {
        SD_CLK_LOW;
        SD_CLK_HIGH;
    }
  }
  read_status = 1;  //SD data next in


  return seccnt;
}

//---------------------------------------------------------------------------
BYTE get_blocks_audio(UINT32 lba, UINT32 seccnt, unsigned short *buff)
{
  unsigned char cL, cH;
  unsigned short c;
  UINT32  i,j;
  int p = 0;
  for(j=0;j<seccnt;j++)
  {
    {
      NCC;
      cmd_buffer[0] = cmd17[0];
      cmd_buffer[1] = (lba>>15)&0xff;
      cmd_buffer[2] = (lba>>7)&0xff;
      cmd_buffer[3] = (lba<<1)&0xff;
      cmd_buffer[4] = 0;
      lba++;
      send_cmd(cmd_buffer);
      NCR;
    }
    while(1)
    {
      SD_CLK_LOW;
      SD_CLK_HIGH;
      if(!(SD_TEST_DAT))
      break;
    }
    for(i=0;i<256;i++)
    {
      BYTE j;
      for(j=0;j<8;j++)
      {
        SD_CLK_LOW;
        SD_CLK_HIGH;
        cL <<= 1;
        if(SD_TEST_DAT)
        cL |= 0x01;
      }
      for(j=0;j<8;j++)
      {
        SD_CLK_LOW;
        SD_CLK_HIGH;
        cH <<= 1;
        if(SD_TEST_DAT)
        cH |= 0x01;
      }

      c = (cH << 8 | cL);

      //debug
      //printf("%x %x ", cL, cH);

      IOWR_16DIRECT(AUDIO_BASE, 0, c);
      p++;

    }
    for(i=0; i<16; i++)
    {
        SD_CLK_LOW;
        SD_CLK_HIGH;
    }
  }
  read_status = 1;  //SD data next in
```

```
  return seccnt;
}

//---------------------------------------------------------------------------
BYTE response_R(BYTE s)
{
  BYTE a=0,b=0,c=0,r=0,crc=0;
  BYTE i,j=6,k;
  while(1)
  {
    SD_CLK_LOW;
    SD_CLK_HIGH;
    if(!(SD_TEST_CMD))
    break;
    if(crc++ >100)
    return 2;
  }
  crc =0;
  if(s == 2)
  j = 17;

  for(k=0; k<j; k++)
  {
    c = 0;
    if(k > 0)                      //for crc culcar
    b = response_buffer[k-1];
    for(i=0; i<8; i++)
    {
      SD_CLK_LOW;
      if(a > 0)
      c <<= 1;
      else
      i++;
      a++;
      SD_CLK_HIGH;
      if(SD_TEST_CMD)
      c |= 0x01;
      if(k > 0)
      {
        crc <<= 1;
        if((crc ^ b) & 0x80)
        crc ^= 0x09;
        b <<= 1;
        crc &= 0x7f;
      }
    }
    if(s==3)
    {
      if( k==1 &&(!(c&0x80)))
      r=1;
    }
    response_buffer[k] = c;
  }
  if(s==1 || s==6)
  {
    if(c != ((crc<<1)+1))
    r=2;
  }
  return r;
}

//---------------------------------------------------------------------------
BYTE send_cmd(BYTE *in)
{
  int i,j;
  BYTE b,crc=0;
  SD_CMD_OUT;
  for(i=0; i < 5; i++)
  {
    b = in[i];
    for(j=0; j<8; j++)
    {
      SD_CLK_LOW;
      if(b&0x80)
      SD_CMD_HIGH;
      else
      SD_CMD_LOW;
      crc <<= 1;
      SD_CLK_HIGH;
      if((crc ^ b) & 0x80)
      crc ^= 0x09;
      b<<=1;
    }
    crc &= 0x7f;
  }
  crc =((crc<<1)|0x01);
  b = crc;
  for(j=0; j<8; j++)
  {
    SD_CLK_LOW;
    if(crc&0x80)
    SD_CMD_HIGH;
    else
    SD_CMD_LOW;
    SD_CLK_HIGH;
    crc<<=1;
  }
  return b;
}
//---------------------------------------------------------------------------
```