Video Conference System
CSEE 4840 Project Report

Manish Sinha, Srikanth Vemula
Department of Electrical Engineering
Columbia University
{ms3766,sv2271}@columbia.edu

1 ABSTRACT

In this document, we describe all the details of the Video Conference System we designed and implemented as part of the CSEE 4840 Embedded Systems course. We first give an introduction of the system by noting the inputs, outputs, and connections of the system components. We then cover the hardware and software architecture of the system. Specifically, we explain the motivation, design and implementation behind each hardware block. We conclude with a section outlining the lessons we learned while doing this project. We also include some advice for students who take this course in the future.

2 INTRODUCTION

Our Video Conference System contains the following components: two cameras with composite output, two Altera DE2 boards, two LCD monitors with VGA interface, and one Ethernet switch. The figure below illustrates our setup:

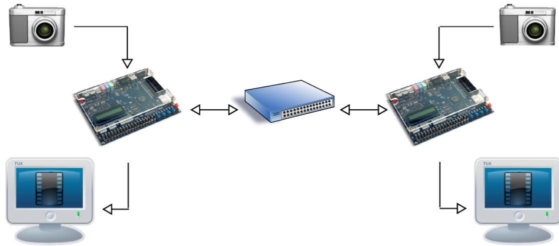

Figure 1: System layout [1] [2] [3]

The cameras interface with the Altera DE2 boards using the CVBS protocol [4]. The Altera DE2 boards interface with the LCD monitor using the VGA protocol. The Altera DE2 boards communicate with the switch using the IP protocol. The underlying transport layer protocol is UDP.

On the LCD monitor, the user sees two centered 320x240 size frames, one stacked on top of the other. The top frame displays the video coming from the local camera. The bottom frame displays the video coming from the network stream (from the other camera). All of the video will be in black and white with 4-bit pixel depth. The reasoning behind the choice of resolution, color, and pixel depth will be described later in the document.

3 HARDWARE ARCHITECTURE

The hardware architecture is as follows:

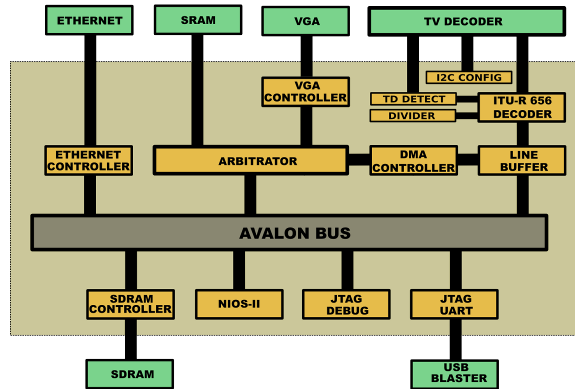

Figure 2: Block Diagram

3.1 ITU-R 656 DECODER

The ADV7181b decoder detects and converts composite video into digital ITU-R BT.656 format. The format embeds unique codes such as SAV(start of video) and EAV(end of video) within the video stream to avoid transmitting timing signals such as HSYNC and VSYNC. After each start of video (SAV) code, the stream

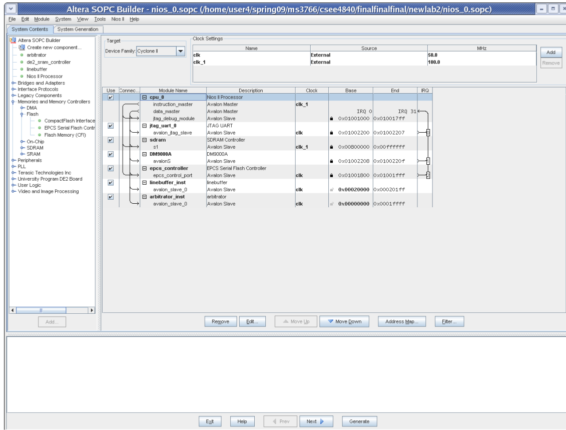

Figure 3: SOPC Builder Screenshot

of active data words always start with Chrominance but in our case we discard them and output only a four bit luminance value of each pixel. The ADV chip outputs at a resolution of 720 x 525 at 27Mhz.

The ITU Decoder takes 720 pixels data from the ADV Chip and outputs 640 pixels of data. This is done using a Divider(DIV) module. The ITU Decoder generates the pixel and line numbers which basically are counters. The input iSkip is used to skip pixels. If it is 1 the data valid output(oDVAL) is 0 signifying that the data is not valid. We have another counter(opixelno) which counts only to 640 that is when the data is valid. The field(oField) output represents the field of the pixel data. If field =0 it represents the odd field else if it is 1, it means even field. The output(oTV_Y) corresponds to the line numbers. We use the line numbers and the field outputs to calculate the SRAM address into which we write the pixel data. The outputs oTV_X gives the total number of pixels that are output by the ADV chip, which is 720. The pixel number (opixelno) output is the number of valid pixels that is 640. All the outputs are fed to the linebuffer. The iTD_DATA is the data input from the ADV chip which contains both luminance and chrominance values of each pixel. It also contains SAV and EAV information. SAV is checked to generate the TV_Y and TV_X outputs. The iSwap_CbCr is not of concern as we are not dealing with Chrominance values. The oYCbCr output corresponds to the 4 bit luminance values.

3.2 LINE BUFFER

The line buffer receives luminance information from the ITU 656 decoder at 27Mhz. It is housed in the block RAM and stores the luminance information of 1 line i.e. 320 pixels. The luminance information(Y) of each pixel is four bits. The line buffer is 80*16 bits in size. The luminance values of 4 pixels that is 16 bits of data is stored in each array index. The Line buffer is a RAM which stores the data from the ITU Decoder and transfers this data over the Avalon bus for network video and to the SRAM using the DMA controller for the local video. Both the transfers take place on receipt of one line of pixel data.The inputs xpos and ypos correspond to the pixel number and line number respectively and are driven by the opixelno and oTV_Y outputs of the ITU Decoder. Every other pixel and every other line is stored in the line buffer. This is done by checking the LSB of the inputs xpos and ypos i.e. the 0th bit . We pick up every other line to give the NIOS Processor enough time to make data transfers for the network video. Another reason for doing so is we are working with a resolution of 320 x 240. The input data valid gives an indication of whether the data is valid or not. Only after we receive 4 pixels of data do we write into an array index of the linebuffer (since we are skipping every other pixel it is 8 pixels). We can transfer 16 bits of data over the Avalon bus in one transfer, hence we write four pixels of data into each address of the block RAM so as to use the maximum capability of the Avalon bus and also to make as few data transfers as possible. The signal linebufferfull gives an indication of when the line buffer is full that is we have received 320 pixels of data from the ITU decoder. This is also an output to the DMA Controller. The data is transferred to the SRAM using a DMA controller only when the line buffer is full. The line buffer reads data from the ITU decoder at 27 Mhz. This ensures better video quality. The linenumber input is used to calculate the memory address in SRAM that the line buffer is written into once the line buffer is full. This calculation is important as the memory address of the SRAM into which the line buffer is written into determines where it is going to be printed on the VGA. The first line from the odd field that is output from the ITU Decoder is stored in the addresses 0-79 of the SRAM. The third line(because we are skipping the second line) of the odd field is stored in the addresses 160-239 and so on. The first line of the even field is written to the address 80-159 and the third line is written into the addresses 240-319 and so on.

The linebuffer also outputs the linebufferfull, luminance, linenumber and field data to the DMA controller. These outputs are useful for

calculating the SRAM addresses for the local video which is transferred via the DMA controller to the arbitrator which in turn transfers the data to the SRAM. The transfer of data happens to the DMA controller at 50 Mhz.

The line buffer is connected to the Avalon Bus and data is read from the line buffer to the NIOS Processor by providing the address in C which is twice the value of the array indices. The transfer of data over the Avalon bus happens only when the line buffer Is full. The data transfer over the Avalon bus to the NIOS Processor happens at 27Mhz.

3.3 DMA CONTROLLER

The motivation behind implementing a DMA (direct memory access) controller for our Video Conference System originates from the principles behind modular design. Each Altera DE2 board has to perform the following three tasks:

1. Display the video from the local camera onto the VGA.
2. Display the video from the network stream onto the VGA.
3. Stream the local video over the network.

A DMA controller enables us to decouple tasks (1) and (2) from each other atleast at the initial stage (ultimately, both tasks contend for the SRAM so they cannot be entirely separated). Formerly, we had the NIOS-II processor perform all three tasks. With a DMA controller, we relegate the NIOS-II to performing tasks (2) and (3) thus relieving its overall load. The benefits were obvious: with a DMA controller, we saw a significant increase in the local and network video framerate.

3.4 ARBITRATOR

The most critical component in our system is the arbitrator. It is needed to regulate the access to (write) and from (read) the SRAM. There are three resources contending for the SRAM. Access to the SRAM is prioritized in the following order inside the arbitrator (from highest priority to lowest priority): VGA controller, Ethernet controller, DMA Controller. The VGA controller gets the highest priority because the VGA controller needs pixels or else the screen output would become malformed and blotted. The reason why the Ethernet is given higher priority than the DMA controller is because the frequency of the incoming ethernet data is low relative to the DMA controller. If this order was switched, the DMA would take over the SRAM resulting in a poor network video framerate.
The Arbitrator works like an SRAM Controller by providing the control signals. The Arbitrator is connected to the Avalon bus, the VGA controller, DMA controller, SRAM.

The output busy to the DMA is used to indicate to the DMA Controller if the arbitrator is busy. At the rising edge of each 50 Mhz clock cycle and when 25 Mhz clock is 0

```
busy_to_dma <= '0';
```

which means that the DMA controller can go ahead and transfer data to the Arbitrator which in turn is put in the SRAM.

The HCOUNT and VCOUNT are inputs that are received from the VGA Controller. These indices signify what part of the VGA we are printing the data. We have a condition check as follows

```
if (HCOUNT+1 > (8+96+40+8+160-1) and
    HCOUNT+1 < (8+96+40+8+160+320) and
    VCOUNT > (2+2+25+8-1) and
    VCOUNT < (2+2+25+8+480) )
```

which is used to check if we are in the active region of the display or not.

```
HCOUNT+1 > (8 + 96 + 40 + 8 +160- 1)
```

checks to see if the VGA is printing in the active video. (8 + 96 + 40 + 8) corresponds to the blanking and sync signals [5]. We add 160 to center the video.

Since we have only 320 pixels we check for the condition

```
HCOUNT+1 < (8 + 96 + 40 + 8 + 160+320)
```

Then we check for pixelindex=-1 i.e. the next four pixels of data. This happens on every fourth pixel I.e. 0,4,8 th pixel and so on. We read from the SRAM once out of every fourth 25Mhz clock cycle as on one read of the SRAM we get luminance values of 4 pixels of data. When pixelindex is -1 we calculate the SRAM address from which data is read. The SRAM address calculation is based on the HCOUNT and VCOUNT inputs that we get from the VGA Controller. The line number that is VCOUNT is used to calculate the SRAM address from which to start the reading from. For example address 0-79 of the SRAM contain the pixel data of the first line that is displayed on the VGA. The second line that is displayed on the VGA is stored in addresses 80-159

and so on. So the SRAM address to start reading for the first line is 0 and for the second line it is 80 and so on. This is done as shown below

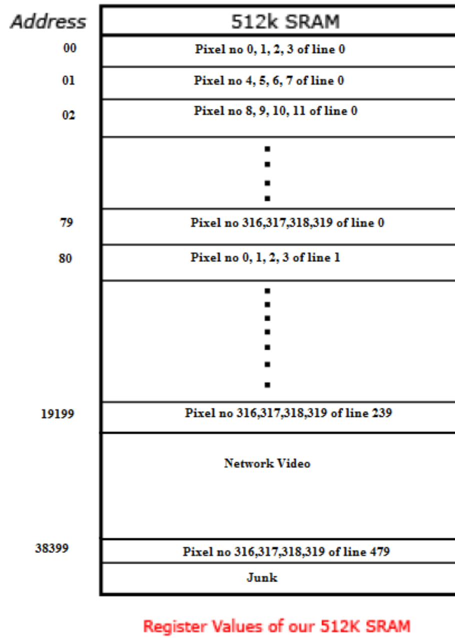

Figure 4: SRAM Memory Organization

```
tmp := ((VCOUNT - 37) * 80);
```

VCOUNT – 37 is done to make sure that we are printing active lines and to avoid blanking and sync signals.

HCOUNT is used to calculate the SRAM address to read data from for printing HCOUNT on the VGA.

HCOUNT should be divided by 4 because each address in the SRAM contains 16 bits of data which corresponds to 4 pixels.

```
tmp2 := "0000000000" & ((HCOUNT+1)-152);
```

This is done to take care of sync and blanking issues.

```
tmp2 := "00" & tmp2 (19 downto 2);
```

This is done to divide by 4 as discussed above. SRAM address calculations are done using tmp and tmp2 as shown below

```
tmp := tmp + tmp2;
SRAM_ADDRESS_BUFFER <=
  std_logic_vector(tmp) (17 downto 0);
```

After this we set readflag to 1 indicating that we can read data from the SRAM address. When reading from the SRAM it is very important to make timing considerations. For example if we give the SRAM address when 25Mhz clock is 0 then we can read data from the address only when clock 25 is 1. Hence we set readflag to 1 so that we can read data when 25Mhz clock becomes 1. But when writing into the SRAM we can specify the address and write data into that address instantly.

We set waitrequest to 1 to stall the NIOS and disable it from making calls to the arbitrator. This is done because the VGA is given top priority that is we provide data to the VGA constantly on every 25 Mhz clock cycle.

After this we check to see if we receive any data over the Avalon bus that is video that is transmitted through the network. This is second in the priority list. Then we check to see if we get any data from the DMA controller. If we do we transfer it to the SRAM and the address into which we write this data is calculated in the DMA Controller. Again we stall the NIOS when this happens to make sure that we have good quality local video. We set the writeflag to 1 indicating that we are writing to the SRAM.

Writing into SRAM from both processes happens when pixelindex=-1 and when pixelindex!=-1.

When 25 Mhz clock is 1 we again make calculations based on HCOUNT and VCOUNT as already described before to check for active video. Then we check for the Avalon bus write and read flag concurrently. If readflag is 1 then we read data from the SRAM and store it in a temporary buffer called fourpixels. This is done as each address in the SRAM contains 4 pixels of data and by storing in a temporary buffer we can ship data to the VGA even when we are not reading data from the SRAM address(we read from the SRAM on every fourth clock cycle of 25 Mhz clock). We also check to see if we receive any data over the Avalon bus at the same time that we check for read flag. If we dont receive any data over the avalon bus we check to see if we receive any data from the DMA controller. If true we transfer this data to the SRAM.

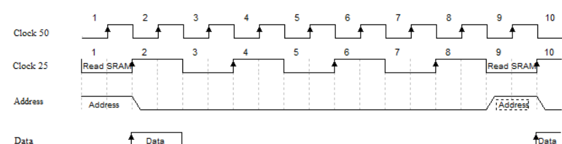

Figure 5: Arbitrator Timing Diagram

Next we check for the pixel index, if it is 0 we output the 4 MSB bits of fourpixels to the VGA. Pixelindex is 0 for 0,4,8th pixel and so on, it is 1 for 1,5,9th pixel and so on, it is 2 for 2,6,10th pixel and o on, 3 for the 3,7,11th pixel and so on. When writing into the SRAM we make sure that the 4 MSB bits correspond to the first pixel, the next 4 pixels correspond to the second pixel and so on.

3.5 VGA CONTROLLER

The VGA controller is adopted from lab 3 but modified to our needs. The rectangle drawing process and the blue background are stripped out. The controller is modified to accept and display an incoming stream of red, green, and blue; each 10-bits where the 6 MSB are zero. The controller is also modified to output HCOUNT (the current pixel index) and VCOUNT (the current line index) to the arbitrator so that the arbitrator can compute the correct SRAM read address when the VGA controller needs new data.

3.6 ETHERNET CONTROLLER

The Ethernet controller was adopted from lab 2. The controller doesn't perform any logic; rather it simply links the ethernet chip signals with the appropriate avalon bus signals.

3.7 SDRAM CONTROLLER

The SDRAM controller is generated by following Altera's tutorial [6]. The tutorial is straightforward and involves making a PLL that generates a clock signal that is delayed by 3ns with respect to the input reference clock. The input reference clock is 100 Mhz in our system. All the signals to the SDRAM chip are provided by the SDRAM controller except for the clock, which is generated by the PLL.

3.8 NIOS-II PROCESSOR

The NIOS-II processor is generated using SOPC builder. We use the NIOS-II/f processor, the most advanced processor SOPC offers. We also use embedded multipliers. Code and data are stored on the SDRAM. The NIOS and SDRAM operate at 100 Mhz. The 100Mhz clock signal was generated by making a PLL using Quartus' Megaplugin wizard. Switching from 50Mhz to 100Mhz significantly improved our network video framerate because the ethernet driver routines (handled by the NIOS) were performed more quicker. Instruction and data cache sizes were kept at their defaults (4 KB & 2 KB respectively).

3.9 JTAG DEBUG

This block is generated by SOPC builder and it helps to debug hardware with the use of "printf" statements.

3.10 JTAG UART

This block is generated by SOPC builder and it helps to debug hardware with the use of "printf" statements.

3.11 I2C CONFIG

This block was adopted from the Terasic DE2_TV [7] reference code along with the ITU decoder. The Analog Devices ADV7181 TV encoder chip is configured using the I2C protocol and this block performs that configuration.

3.12 TD DETECT

This module is used to stabilize the video. It provides the reset signal for the ITU decoder. This module Is required for a stable video.

3.13 DIVIDER

The DIV module takes as input the pixel number (oTV_X) from the ITU decoder, divides it by 9 and generates the quotient and remainder. If the remainder is 0 the signal mySkip is set to 1 in the top level entity(top.vhd). This signal is fed as input to the ITU decoder and helps to decide which of the pixels to discard.

4 SOFTWARE ARCHITECTURE

The pseudo-code of the main() function in our C code is as follows:

```
while(1)
{
    (1) Wait until the line buffer is
        full.
    (2) Read the line buffer and save
        it into a UDP packet buffer.
    (3) Check if 8 lines have been
        saved. If so, push the UDP
        packet buffer out to the
        ethernet.
    (4) Check if a packet has been
        received. If so, unwrap its
        contents and write the 8
        lines to the arbitrator.
}
```

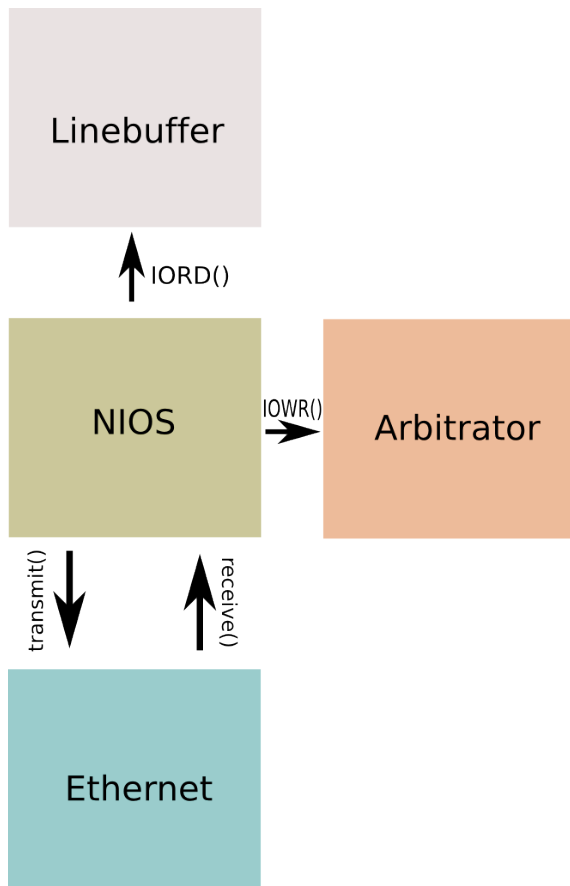

Figure 6: High-level software architecture diagram

Ethernet Activities

In our system, the NIOS both reads and writes from the DM9000A ethernet chip. This chip contains a 16 KB SRAM buffer; 13K for RX and 3K for TX. Receiving and sending a packet involves a sequence of register reads and writes. We found the DM9000A application notes [8] tremendously useful in understanding the ethernet chip. We adopted the ethernet driver code provided in lab 2 and heavily optimized it. We found that the standard delay of 1 microsecond used in the lab2 code too generous. Instead, we replaced them with "nop" instructions which dramatically helped the transmit speed of the ethernet chip. Another optimization we found was in the transmit routine when the SRAM is being written to. In the lab 2 code, a 1 microsecond delay is performed after each write to the ethernet SRAM. We found that the wait can be eliminated entirely after 50 successive writes to the SRAM. Before our modifications, the maximum transmit rate of the ethernet chip was 80 KB per second. After our modifications, we were able to transmit at 1.2 MB per second.

The transmit routine, outside of the optimization, is the same as in lab 2 [9]. The receive routine was re-written entirely because the lab 2 routine could not handle a continuous stream of incoming data. This process was straightforward due to the well written application notes of the ethernet chip.

The packet transmitted is a UDP packet. The UDP protocol was chosen because it is ideal for streaming applications. The user will be unable to detect a few lost lines if a UDP packet is dropped. The maximum size of an ethernet frame is 1518 bytes [10] so that is our upper bound. Each line consists of 320 4-bit pixels (or 160 bytes). Along with each line, its line number (2 bytes) and its field (1 byte) are sent. Thus, each line takes 163 bytes total. Limited by the upper bound of an ethernet frame, we can fit 8 lines into a ethernet frame.

We poll the receive routine to check if a new packet has arrived rather than using interrupts. The reason we did this was because we did not want the ethernet incoming data to thrash the transmit process.

We found two programs essential to helping us debug incoming and outgoing ethernet packets: Wireshark [11] and Colasoft Packet Builder [12]. To test the receive routine, we manufactured a packet consisting of 8 spaced out white lines in Packet Builder and then sent it to the Altera board. To test the transmit routine, we inspected the contents of the outgoing packets using Wireshark.

Writing to Arbitrator & Reading the Linebuffer

The arbitrator is connected to the avalon bus so we can use IOWR() commands to transmit data from the NIOS to the arbitrator. Similarly, we use IORD() calls to transmit data from the block RAM in the linebuffer to the NIOS. Each IORD() or IOWR() are 16-bit operations. We found that the addresses passed into these routines have to multiplied by 2 for alignment reasons.

4.1 Who Did What

Manish worked on the DMA controller and ethernet drivers. Srikanth worked on the linebuffer and made modifications to the ITU-decoder. Both group members worked on the arbitrator and VGA controller. Manish is more interested in software whereas Srikanth is more interested in hardware. This created great synergy in the group because we complemented each other well.

4.2 Lessons Learned

The most important lesson we learned was that timing diagrams are absolutely critical. Knowing what is happening at the cycle level and doing a proper timing analysis are essential techniques to uncovering potential problems. We also learned that simulations are helpful when possible. We used simulations to verify our SRAM calculations. Also, as Professor Edwards stats, having multiple clock domains in a system can lead to serious problems. Our system has three clock domains (27 Mhz, 50 Mhz, 100 Mhz) but we were able to carefully decouple each domain from the other. We also learned about the basic hardware-software balance/tradeoff. While implementing blocks in hardware can yield more precision and speed, it is much more costly than implementing it in software.

4.3 Advice for Future Projects

Our advice to future students of CSEE 4840 is to start their projects well in advance. These projects take much more time than one anticipates; in fact, the bulk of the time might be wasted waiting for the VHDL to compile. We would also advise groups to spend a thorough effort during the design phase to figure out all possible issues. This will reap rewards in the long term because less problems will be discovered during the implementation phase. We also suggest doing a comprehensive timing analysis on the system to ensure that there will not be any timing synchronization issues. We urge groups to use the simplicity of the SOPC builder to their advantage. Debugging hardware can be difficult at times because it can be difficult to read the data. SOPC builder allows the user to easily connect any component to the avalon bus so that printf() statements from the NIOS can be used to read values in hardware. Lastly, we suggest future students to distribute their work wisely so that work is done in parallel.

4.4 REFERENCES

[1] Network Switch, http://openclipart.org/people/rgtaylor_csc/rgtaylor_csc_net_switch.svg

[2] Altera DE2, http://faculty.lasierra.edu/~ehwang/digitaldesign/webpages/de2.jpg

[3] Computer Icon, http://www.gnome-look.org/content/show.php/SudUbuntu+?content=93010

[4] Video Basics, http://www.maxim-ic.com/appnotes.cfm/an_pk/734

[5] Video, http://www1.cs.columbia.edu/~sedwards/classes/2009/4840/video.pdf

[6] Using the SDRAM Memory on Altera's DE2 Board with VHDL Design, http://www1.cs.columbia.edu/~sedwards/classes/2009/4840/tut_DE2_sdram_vhdl.pdf

[7] Terasic DE2_TV, http://users.ece.gatech.edu/~hamblen/DE2/DE2_demonstrations/DE2_TV/

[8] DM9000A Application Notes, http://www1.cs.columbia.edu/~sedwards/classes/2009/4840/Davicom-DM9000A-Application-Notes.pdf

[9] CSEE 4840: Lab 2, http://www1.cs.columbia.edu/~sedwards/classes/2009/4840/lab2.tar.gz

[10] Ethernet, http://en.wikipedia.org/wiki/Ethernet

[11] Wireshark, http://www.wireshark.org/

[12] Colasoft Packet Builder, http://www.colasoft.com/packet_builder/

4.5 APPENDIX

4.6 VHDL - ARBITRATOR

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity arbitrator is

  port (
    reset : in std_logic;
    clk   : in std_logic; -- 50 Mhz

    -- These inputs & outputs make the component avalon bus
-- compliant.
read : in std_logic;
write : in std_logic;
chipselect : in std_logic;
address : in std_logic_vector(15 downto 0);
readdata : out std_logic_vector(15 downto 0);
writedata : in std_logic_vector(15 downto 0);
waitrequest: out std_logic;

-- SRAM Connection
SRAM_DQ   : inout std_logic_vector(15 downto 0);
    SRAM_ADDR : out std_logic_vector(17 downto 0);
    SRAM_UB_N, SRAM_LB_N : out std_logic;
    SRAM_WE_N, SRAM_CE_N : out std_logic;
    SRAM_OE_N            : out std_logic;

-- From the VGA Controller
HCOUNT, VCOUNT: unsigned (9 downto 0);

-- To the VGA Controller
RED, GREEN, BLUE : out std_logic_vector (9 downto 0);
pixel_clock: out std_logic;

-- TO DMA
busy_to_dma: out std_logic;

-- FROM DMA
address_from_dma: in std_logic_vector(17 downto 0);
data_from_dma: in std_logic_vector(15 downto 0);
data_ready_from_dma: in std_logic
    );

end arbitrator;

architecture behavior of arbitrator is

signal clk25 : std_logic := '0';

signal SRAM_ADDRESS_BUFFER : std_logic_vector(17 downto 0) := "000000000000000000";
signal SRAM_WRITEDATA_BUFFER : std_logic_vector(15 downto 0) := "0000000000000000";
signal SRAM_READDATA_BUFFER        : std_logic_vector(15 downto 0) := "0000000000000000";
signal SRAM_WRITE_ENABLE_BUFFER_N   : std_logic := '0';
```

```
signal read_flag : std_logic := '0';
signal write_flag : std_logic := '0';
signal RED_BUFFER, GREEN_BUFFER,
       BLUE_BUFFER : std_logic_vector (9 downto 0) := "0000000000";

signal sramWriteAddressFromNIOS     : std_logic_vector(15 downto 0) := "0000000000000000";
signal pixelDataFromNIOS            : std_logic_vector(15 downto 0) := "0000000000000000";

begin

process (clk)
begin
if rising_edge(clk) then
clk25 <= not clk25;
end if;
end process;

SRAM_DQ <= SRAM_WRITEDATA_BUFFER when SRAM_WRITE_ENABLE_BUFFER_N = '0' else (others => 'Z');
SRAM_READDATA_BUFFER <= SRAM_DQ;
SRAM_ADDR <= SRAM_ADDRESS_BUFFER;
SRAM_WE_N <= SRAM_WRITE_ENABLE_BUFFER_N;
SRAM_UB_N <= '0';
SRAM_LB_N <= '0';
SRAM_CE_N <= '0';
SRAM_OE_N <= '0';

pixel_clock <= clk25;


process(clk,clk25)

variable pixelIndex: integer := -1;
variable addr: std_logic_vector (19 downto 0) := "00000000000000000000";
variable tmp: unsigned (19 downto 0) := "00000000000000000000";
variable tmp2: unsigned (19 downto 0) := "00000000000000000000";
variable fourPixels: std_logic_vector(15 downto 0) := "0000000000000000";

begin
-- CLK_50 Positive Edge
if clk'event and clk='1' then
busy_to_dma <= '0';
-- CLK_50 Positive Edge, CLK_25 Negative Edge
if clk25 = '0' then
-- Are we about to be in the active region? Add 1 to HCOUNT because it's old
if (HCOUNT+1 > (8 + 96 + 40 + 8 + 160 - 1) and HCOUNT+1 < (8 + 96 + 40 + 8 + 160 + 320)
and VCOUNT > (2 + 2 + 25 + 8 - 1) and VCOUNT < (2 + 2 + 25 + 8 + 480) ) then

-- If our four pixel buffer is empty
if pixelIndex = -1 then
-- First, lets caculate the desired SRAM address based on HCOUNT & VCOUNT.
tmp := ((VCOUNT - 37) * 80);
tmp2 := "0000000000" & ((HCOUNT+1+160)-152);
tmp2 := "00" & tmp2 (19 downto 2); -- Divide by 4
tmp := tmp + tmp2;
SRAM_ADDRESS_BUFFER <= std_logic_vector(tmp) (17 downto 0);
SRAM_WRITE_ENABLE_BUFFER_N <= '1';
read_flag <= '1';
waitrequest <= '1';
```

```
elsif chipselect = '1' and write = '1' then
SRAM_ADDRESS_BUFFER <= "00" & address;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= writedata;
write_flag <= '1';
waitrequest <= '0';


elsif data_ready_from_dma = '1' then
SRAM_ADDRESS_BUFFER <= address_from_dma;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= data_from_dma;
write_flag <= '1';
waitrequest <= '1';
end if;

elsif chipselect = '1' and write = '1' then
SRAM_ADDRESS_BUFFER <= "00" & address;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= writedata;
write_flag <= '1';
waitrequest <= '0';


elsif data_ready_from_dma = '1' then
SRAM_ADDRESS_BUFFER <= address_from_dma;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= data_from_dma;
write_flag <= '1';
waitrequest <= '1';

end if;


-- CLK_50 Positive Edge, CLK_25 Positive Edge
elsif clk25 = '1' then
-- If we're in the active region
if (HCOUNT > (8 + 96 + 40 + 8 + 160 - 1) and HCOUNT < (8 + 96 + 40 + 8 + 160 + 320)
and VCOUNT > (2 + 2 + 25 + 8 - 1) and VCOUNT < (2 + 2 + 25 + 8 + 480) ) then

if chipselect = '1' and write = '1' then
SRAM_ADDRESS_BUFFER <= "00" & address;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= writedata;
write_flag <= '1';
waitrequest <= '0';

elsif data_ready_from_dma = '1' then
SRAM_ADDRESS_BUFFER <= address_from_dma;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= data_from_dma;
write_flag <= '1';
waitrequest <= '1';
end if;


-- Read the SRAM contents & buffer it locally *if* we just initiated a read
```

```
if read_flag = '1' then
fourPixels := SRAM_READDATA_BUFFER;
--SRAM_WRITE_ENABLE_BUFFER_N <= '0';
pixelIndex := 0;
read_flag <= '0';
waitrequest <= '0';
end if;

-- To the VGA Controller
case pixelIndex is
when 0 =>
RED <= fourPixels (15 downto 12) & "000000";
GREEN <= fourPixels (15 downto 12) & "000000";
BLUE <= fourPixels (15 downto 12) & "000000";
when 1 =>
RED <= fourPixels (11 downto 8) & "000000";
GREEN <= fourPixels (11 downto 8) & "000000";
BLUE <= fourPixels (11 downto 8) & "000000";
when 2 =>
RED <= fourPixels (7 downto 4) & "000000";
GREEN <= fourPixels (7 downto 4) & "000000";
BLUE <= fourPixels (7 downto 4) & "000000";
when 3 =>
RED <= fourPixels (3 downto 0) & "000000";
GREEN <= fourPixels (3 downto 0) & "000000";
BLUE <= fourPixels (3 downto 0) & "000000";

busy_to_dma <= '1';
when others =>
RED <= "0000000000";
GREEN <= "0000000000";
BLUE <= "0000000000";
end case;

-- If all four pixels have been read, time to signify that the pixel
-- buffer is empty so a new SRAM read can be initiated
if pixelIndex = 3 then
pixelIndex := -1;
else
pixelIndex := pixelIndex + 1;
end if;

-- If not, output black
else
RED <= "0000000000";
GREEN <= "0000000000";
BLUE <= "0000000000";

if chipselect = '1' and write = '1' then
SRAM_ADDRESS_BUFFER <= "00" & address;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
SRAM_WRITEDATA_BUFFER <= writedata;
write_flag <= '1';
waitrequest <= '0';

elsif data_ready_from_dma = '1' then
SRAM_ADDRESS_BUFFER <= address_from_dma;
SRAM_WRITE_ENABLE_BUFFER_N <= '0';
```

```
SRAM_WRITEDATA_BUFFER <= data_from_dma;
write_flag <= '1';
waitrequest <= '1';
end if;


end if;

end if;


-- CLK_50 Negative Edge
elsif clk'event and clk='0' then

end if;

end process;


-- Avalon stuff. WRITE = NIOS->HERE
-- READ  = HERE->NIOS
process (clk)
begin
if clk'event and clk='1' then
if reset = '1' then
readdata <= (others => '0');
else
if chipselect = '1' then
if read = '1' then
if address = "0000000000000000" then
readdata <= std_logic_vector("000000" & HCOUNT);
--readdata <= testSig;
elsif address = "0000000000000001" then
readdata <= SRAM_READDATA_BUFFER;
elsif address = "0000000000000010" then
readdata <= SRAM_WRITEDATA_BUFFER;
elsif address = "0000000000000100" then
readdata <= "000000000000000" & SRAM_WRITE_ENABLE_BUFFER_N;
elsif address = "0000000000001000" then
readdata <= SRAM_ADDRESS_BUFFER (15 downto 0);
end if;

end if;
end if;
end if;
end if;
end process;

end behavior;
```

4.7 VHDL - VGA CONTROLLER

```
-------------------------------------------------------------------------------
--
-- Simple VGA raster display - MODIFIED BY VIDEO CONFERENCE SYSTEM GROUP
--
-- Stephen A. Edwards
-- sedwards@cs.columbia.edu
```

```
--
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity de2_vga_raster is

  port (
    reset : in std_logic;
clk: in std_logic;
    pixel_clock: in std_logic;

-- Coming from the Arbitrator
RED, GREEN, BLUE : in unsigned (9 downto 0);

    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]

HCOUNT_OUT,VCOUNT_OUT: out unsigned(9 downto 0)
    );

end de2_vga_raster;

architecture rtl of de2_vga_raster is

  -- 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;

  constant RECTANGLE_HSTART : integer := 50;
  constant RECTANGLE_HEND   : integer := 110;
  constant RECTANGLE_VSTART : integer := 100;
  constant RECTANGLE_VEND   : integer := 160;

  -- Signals for the video controller
  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 rectangle_h, rectangle_v, rectangle : std_logic;  -- rectangle area
signal circle_h, circle_v, circle: std_logic;
signal final_output: std_logic;

signal xcenter: integer := 50;
signal xtmp: integer;
signal ytmp: integer;

signal ycenter: integer := 100;
signal CIRCLE_RADIUS: integer := 30;
signal CIRCLE_RADIUS_SQ: integer := 900;

signal clk25 : std_logic := '0';

begin

HCOUNT_OUT <= Hcount;
VCOUNT_OUT <= Vcount;

  HCounter : process (pixel_clock)
  begin
    if rising_edge(pixel_clock) 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 (pixel_clock)
  begin
    if rising_edge(pixel_clock) 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 (pixel_clock)
  begin
```

```
  if rising_edge(pixel_clock) 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 (pixel_clock)
begin
  if rising_edge(pixel_clock) 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 (pixel_clock)
begin
  if rising_edge(pixel_clock) then
    if reset = '1' then
      vga_vsync <= '1';
    elsif EndOfLine ='1' then
      if EndOfField = '1' then
xcenter <= xtmp;
ycenter <= ytmp;
        vga_vsync <= '1';
      elsif Vcount = VSYNC - 1 then
        vga_vsync <= '0';
      end if;
    end if;
  end if;
end process VSyncGen;

VBlankGen : process (pixel_clock)
begin
  if rising_edge(pixel_clock) 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 (pixel_clock, reset)
begin
```

```
    if reset = '1' then
      VGA_R <= "0000000000";
      VGA_G <= "0000000000";
      VGA_B <= "0000000000";
    elsif clk'event and clk = '1' then
      if final_output = '1' then
        VGA_R <= RED;
        VGA_G <= GREEN;
        VGA_B <= BLUE;
      elsif vga_hblank = '0' and vga_vblank ='0' then
        VGA_R <= "0000000000";
        VGA_G <= "0000000000";
        VGA_B <= "1111111111";
      else
        VGA_R <= "0000000000";
        VGA_G <= "0000000000";
        VGA_B <= "0000000000";
      end if;
    end if;
  end process VideoOut;

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

end rtl;
```

4.8 VHDL - DMA CONTROLLER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DMACONTROLLER is

  port (
clk_50: in std_logic;
linebufferfull_from_lb: in std_logic;
data_from_lb: in std_logic_vector(15 downto 0);
busy_from_arb: in std_logic;
field_from_lb: in std_logic;
lineNumber_from_lb: in std_logic_vector (9 downto 0);

address_to_lb: out std_logic_vector(6 downto 0);
address_to_arb: out std_logic_vector(17 downto 0);
data_to_arb: out std_logic_vector(15 downto 0);
data_ready_to_arb: out std_logic
    );

end DMACONTROLLER;

architecture behavior of DMACONTROLLER is

begin
process (clk_50)
```

```
variable state: std_logic := '0';
variable addressIndex: unsigned (6 downto 0) := "0000000";
variable tmp: unsigned(19 downto 0);

begin
if rising_edge(clk_50) then

if state = '0' then
data_ready_to_arb <= '0';
if linebufferfull_from_lb = '1' then
address_to_lb <= std_logic_vector(addressIndex);
state := '1';
end if;

elsif state = '1' then
if busy_from_arb = '0' then
data_to_arb <= data_from_lb;
data_ready_to_arb <= '1';

if field_from_lb = '1' then
tmp := (unsigned(lineNumber_from_lb) * 80) + addressIndex;
address_to_arb <= std_logic_vector(tmp) (17 downto 0);
else
tmp := ((unsigned(lineNumber_from_lb)+1) * 80) + addressIndex;
address_to_arb <= std_logic_vector(tmp) (17 downto 0);
end if;

if (addressIndex = 79) then
state := '0';
addressIndex := "0000000";
else
addressIndex := addressIndex + 1;
address_to_lb <= std_logic_vector(addressIndex);
end if;

end if;
end if;
end if;

end process;

end behavior;
```

4.9 VHDL - ITU 656 DECODER

```
module ITU_656_Decoder( // TV Decoder Input
iTD_DATA,
// Position Output
oTV_X,
opixelno,
oTV_Y,
// YUV 4:2:2 Output
oYCbCr,
oDVAL,
oField,
// Control Signals
iSwap_CbCr,
iSkip,
```

```
iRST_N,
iCLK_27 );
input [7:0] iTD_DATA;
input iSwap_CbCr;
input iSkip;
input iRST_N;
input iCLK_27;

// the output data
// MODIFIED
//output [15:0] oYCbCr;
output  oField;
output [3:0] oYCbCr;
output  [9:0] opixelno;
output [9:0] oTV_X;
output [9:0] oTV_Y;
output oDVAL;

// For detection
reg [23:0] Window; // Sliding window register
reg [17:0] Cont; // Counter
reg Active_Video;
reg Start;
reg Data_Valid;
reg Pre_Field;
reg Field;
wire SAV;
reg FVAL;
reg [9:0] TV_X;
reg [9:0] TV_Y;
reg [31:0] Data_Cont;

// For ITU-R 656 to ITU-R 601
// MODIFIED
reg [7:0] Cb;
reg [7:0] Cr;
//reg [15:0] YCbCr;
reg [7:0] YCbCr;
assign  oField  =   Field;
assign  opixelno= TV_X;
assign oTV_X = Cont>>1;
assign oTV_Y = TV_Y;
//assign oYCbCr = {YCbCr[7:4],4'b0};
assign oYCbCr = {YCbCr[7:4]};
assign oDVAL = Data_Valid;
assign SAV = (Window==24'hFF0000)&(iTD_DATA[4]==1'b0);
assign oTV_Cont= Data_Cont;

always@(posedge iCLK_27 or negedge iRST_N)
begin
if(!iRST_N)
begin
// Register initial
Active_Video<= 1'b0;
Start <= 1'b0;
Data_Valid <= 1'b0;
Pre_Field <= 1'b0;
Field <= 1'b0;
```

```
Window <= 24'h0;
Cont <= 18'h0;
// MODIFIED
// these are not used for B&W video
Cb <= 8'h0;
Cr <= 8'h0;
//YCbCr <= 16'h0;
YCbCr <= 8'h0;
FVAL <= 1'b0;
TV_Y <= 10'h0;
Data_Cont <= 32'h0;
end
else
begin
// Sliding window
Window <= {Window[15:0],iTD_DATA};
// Active data counter
if(SAV)
Cont <= 18'h0;
else if(Cont<1440)
Cont <= Cont+1'b1;
// Check the video data is active?
if(SAV)
Active_Video<= 1'b1;
else if(Cont==1440)
Active_Video<= 1'b0;
// Is frame start?
Pre_Field <= Field;
if({Pre_Field,Field}==2'b10)
Start <= 1'b1;
// Field and frame valid check
if(Window==24'hFF0000)
begin
FVAL <= !iTD_DATA[5];
Field <= iTD_DATA[6];
end
// ITU-R 656 to ITU-R 601
if(iSwap_CbCr)
begin
case(Cont[1:0]) // Swap
// return 00
// 0: Cb <=  iTD_DATA;
// first Luminance 8 bit
1: YCbCr <= iTD_DATA;//{iTD_DATA,Cr};
// return 00
//2: Cr <=  iTD_DATA;
// second Luminance
3: YCbCr <= iTD_DATA;//{iTD_DATA,Cb};
endcase
end
else
begin
case(Cont[1:0]) // Normal
//0: Cb <=  iTD_DATA;
1: YCbCr <= iTD_DATA;//{iTD_DATA,Cb};
//2: Cr <=  iTD_DATA;
3: YCbCr <= iTD_DATA;//{iTD_DATA,Cr};
endcase
```

```
end
// Check data valid
if( Start // Frame Start?
&&  FVAL // Frame valid?
&&  Active_Video // Active video?
&&  Cont[0] // Complete ITU-R 601?
&&  !iSkip ) // Is non-skip pixel?
Data_Valid <= 1'b1;
else
Data_Valid <= 1'b0;
// TV decoder line counter for one field
if(FVAL && SAV)
TV_Y<= TV_Y+1;
else if(!FVAL)
TV_Y<= 0;
// Data counter for one field
if(SAV)
TV_X <= 0;
else if(Data_Valid)
TV_X <= TV_X + 1'b1;
end
end

endmodule
```

4.10 VHDL - LINE BUFFER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--use ieee.std_logic_arith.all;

entity linebuffer is

  port (
    reset : in std_logic;
    clk_27   : in std_logic;
clk_50: in std_logic;
clk_100: in std_logic;

xpos: in std_logic_vector(9 downto 0);
ypos: in std_logic_vector(9 downto 0);
datavalid:in std_logic ;
y: in std_logic_vector (3 downto 0);
Field: in std_logic;

    -- These inputs & outputs make the component avalon bus
-- compliant.
read : in std_logic;
write : in std_logic;
chipselect : in std_logic;
address : in std_logic_vector(7 downto 0);
readdata : out std_logic_vector(15 downto 0);
writedata : in std_logic_vector(15 downto 0);

-- TO DMA CONTROLLER
linebufferfull_to_dma : out std_logic;
data_to_dma: out std_logic_vector(15 downto 0);
```

```
field_to_dma: out std_logic;
linenumber_to_dma: out std_logic_vector (9 downto 0);

-- FROM DMA CONTROLLER
address_from_dma: in std_logic_vector(6 downto 0)

    );

end linebuffer;

architecture behavior of linebuffer is

type ram_type is array ( 80 downto 0) of std_logic_vector(15 downto 0);
signal RAM : ram_type;
signal linebufferfull: std_logic;
signal lineNumber: std_logic_vector(9 downto 0);

begin

process(clk_27)

variable tmpaddress: integer range 0 to 79:=0 ;
variable pixel_no: integer range 0 to 4:=0 ;
variable tmpdata: std_logic_vector(15 downto 0);

begin
if clk_27'event and clk_27='1' then
linebufferfull <= '0';

-- xpos(0) = '0' ensures every even pixel is taken (and thus every odd pixel is skipped)
-- ypos(0) = '0' ensures every even line is taken (and odd line is skipped)

-- The ITU decoder outputs 480 lines with 640 pixels per line. Using our sampling
-- technique decimates the resolution to 320x240
if (ypos(0)='0' and datavalid='1' and ypos > "0000010001" and
    ypos < "0011111110" ) then
if xpos(2 downto 0)= "000" then
tmpdata(15 downto 12) := y;
pixel_no := pixel_no + 1;
elsif xpos(2 downto 0)= "010" then
tmpdata(11 downto 8) := y;
pixel_no := pixel_no + 1;
elsif xpos(2 downto 0)= "100" then
tmpdata(7 downto 4) := y;
pixel_no := pixel_no + 1;
elsif xpos(2 downto 0)= "110" then
tmpdata(3 downto 0) := y;
pixel_no := pixel_no + 1;
end if;

-- If we've gotten four pixels, then: reset the pixel index, update the
-- write address of the local line buffer, clear the four pixel array, and
-- finally write the line buffer
if pixel_no = 4 then
pixel_no := 0;
RAM(tmpaddress)<= tmpdata;
tmpdata := "0000000000000000";
```

```
-- If we’ve filled addresses 0-79, it’s time to flush the buffer out to the
-- NIOS.
if (tmpaddress = 79) then
linebufferfull <= ’1’;
tmpaddress:=0;
lineNumber <= std_logic_vector(unsigned(ypos)-18);
RAM(80) <= Field & "0000" & lineNumber & ’1’;
else
tmpaddress := tmpaddress + 1;
end if;
end if;



end if;
end if;
end process;


process (clk_50)

begin
if rising_edge(clk_50) then
linebufferfull_to_dma <= linebufferfull;
data_to_dma <= RAM(to_integer(unsigned(address_from_dma)));
field_to_dma <= Field;
linenumber_to_dma <= lineNumber;
end if;
end process;

-- Avalon stuff. WRITE = NIOS->HERE
-- READ  = HERE->NIOS
process (clk_100)
begin
if rising_edge(clk_100) then
if reset = ’1’ then
readdata <= (others => ’0’);
else
if chipselect = ’1’ then
if read = ’1’ then
if address = "01010001" then
readdata <= datavalid & "00000" & xpos;
elsif address = "01010010" then
readdata <= datavalid & "00000" & ypos;
else
readdata <= RAM(to_integer(unsigned(address)));
end if;
elsif write = ’1’ then
-- Do not care
end if;
end if;
end if;
end if;
end process;

end behavior;
```

4.11 VHDL - TOP LEVEL ENTITY

```
--
-- DE2 top-level module that includes the simple VGA raster generator
--
-- Stephen A. Edwards, Columbia University, sedwards@cs.columbia.edu
--
-- From an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top is

  port (
    -- Clocks

    CLOCK_27,                                     -- 27 MHz
    CLOCK_50,                                     -- 50 MHz
    EXT_CLOCK : in std_logic;                     -- External Clock

    -- Buttons and switches

    KEY : in std_logic_vector(3 downto 0);        -- Push buttons
    SW : in std_logic_vector(17 downto 0);        -- DPDT switches

    -- LED displays

    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 -- 7-segment displays
       : out std_logic_vector(6 downto 0);
    LEDG : out std_logic_vector(8 downto 0);      -- Green LEDs
    LEDR : out std_logic_vector(17 downto 0);     -- Red LEDs

    -- RS-232 interface

    UART_TXD : out std_logic;                     -- UART transmitter
    UART_RXD : in std_logic;                      -- UART receiver

    -- IRDA interface

--    IRDA_TXD : out std_logic;                     -- IRDA Transmitter
    IRDA_RXD : in std_logic;                      -- IRDA Receiver

    -- SDRAM

    DRAM_DQ : inout std_logic_vector(15 downto 0); -- Data Bus
    DRAM_ADDR : out std_logic_vector(11 downto 0); -- Address Bus
    DRAM_WE_N,                                    -- Write Enable
    DRAM_CAS_N,                                   -- Column Address Strobe
    DRAM_RAS_N,                                   -- Row Address Strobe
    DRAM_CS_N,                                    -- Chip Select
    DRAM_CLK,                                     -- Clock
    DRAM_CKE : out std_logic;                     -- Clock Enable
```

```
DRAM_LDQM,                                   -- Low-byte Data Mask
DRAM_UDQM,                                   -- High-byte Data Mask
DRAM_BA_0,                                   -- Bank Address 0
DRAM_BA_1: out std_logic;                 -- Bank Address 0

-- FLASH

FL_DQ : inout std_logic_vector(7 downto 0);      -- Data bus
FL_ADDR : out std_logic_vector(21 downto 0);  -- Address bus
FL_WE_N,                                       -- Write Enable
FL_RST_N,                                      -- Reset
FL_OE_N,                                       -- Output Enable
FL_CE_N : out std_logic;                       -- Chip Enable

-- SRAM

SRAM_DQ : inout std_logic_vector(15 downto 0); -- Data bus 16 Bits
SRAM_ADDR : out std_logic_vector(17 downto 0); -- Address bus 18 Bits
SRAM_UB_N,                                   -- High-byte Data Mask
SRAM_LB_N,                                   -- Low-byte Data Mask
SRAM_WE_N,                                   -- Write Enable
SRAM_CE_N,                                   -- Chip Enable
SRAM_OE_N : out std_logic;                   -- Output Enable

-- USB controller

OTG_DATA : inout std_logic_vector(15 downto 0); -- Data bus
OTG_ADDR : out std_logic_vector(1 downto 0);    -- Address
OTG_CS_N,                                     -- Chip Select
OTG_RD_N,                                     -- Write
OTG_WR_N,                                     -- Read
OTG_RST_N,                                    -- Reset
OTG_FSPEED,                     -- USB Full Speed, 0 = Enable, Z = Disable
OTG_LSPEED : out std_logic;     -- USB Low Speed, 0 = Enable, Z = Disable
OTG_INT0,                                     -- Interrupt 0
OTG_INT1,                                     -- Interrupt 1
OTG_DREQ0,                                    -- DMA Request 0
OTG_DREQ1 : in std_logic;                     -- DMA Request 1
OTG_DACK0_N,                                  -- DMA Acknowledge 0
OTG_DACK1_N : out std_logic;                  -- DMA Acknowledge 1

-- 16 X 2 LCD Module

LCD_ON,                     -- Power ON/OFF
LCD_BLON,                   -- Back Light ON/OFF
LCD_RW,                     -- Read/Write Select, 0 = Write, 1 = Read
LCD_EN,                     -- Enable
LCD_RS : out std_logic;     -- Command/Data Select, 0 = Command, 1 = Data
LCD_DATA : inout std_logic_vector(7 downto 0); -- Data bus 8 bits

-- SD card interface

SD_DAT,                     -- SD Card Data
SD_DAT3,                    -- SD Card Data 3
SD_CMD : inout std_logic;   -- SD Card Command Signal
SD_CLK : out std_logic;     -- SD Card Clock

-- USB JTAG link
```

```
TDI,                        -- CPLD -> FPGA (data in)
TCK,                        -- CPLD -> FPGA (clk)
TCS : in std_logic;         -- CPLD -> FPGA (CS)
TDO : out std_logic;        -- FPGA -> CPLD (data out)

-- I2C bus

I2C_SDAT : inout std_logic; -- I2C Data
I2C_SCLK : out std_logic;   -- I2C Clock

-- PS/2 port

PS2_DAT,                    -- Data
PS2_CLK : in std_logic;     -- Clock

-- VGA output

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]

--  Ethernet Interface

ENET_DATA : inout std_logic_vector(15 downto 0);    -- DATA bus 16Bits
ENET_CMD,           -- Command/Data Select, 0 = Command, 1 = Data
ENET_CS_N,                                      -- Chip Select
ENET_WR_N,                                      -- Write
ENET_RD_N,                                      -- Read
ENET_RST_N,                                     -- Reset
ENET_CLK : out std_logic;                       -- Clock 25 MHz
ENET_INT : in std_logic;                        -- Interrupt

-- Audio CODEC

AUD_ADCLRCK : inout std_logic;                  -- ADC LR Clock
AUD_ADCDAT : in std_logic;                      -- ADC Data
AUD_DACLRCK : inout std_logic;                  -- DAC LR Clock
AUD_DACDAT : out std_logic;                     -- DAC Data
AUD_BCLK : inout std_logic;                     -- Bit-Stream Clock
AUD_XCK : out std_logic;                        -- Chip Clock

-- Video Decoder

TD_DATA : in std_logic_vector(7 downto 0);  -- Data bus 8 bits
TD_HS,                                      -- H_SYNC
TD_VS : in std_logic;                       -- V_SYNC
TD_RESET : out std_logic;                   -- Reset

-- General-purpose I/O

GPIO_0,                                    -- GPIO Connection 0
GPIO_1 : inout std_logic_vector(35 downto 0) -- GPIO Connection 1
```

```
    );

end top;



architecture datapath of top is

COMPONENT sdram_pll
PORT ( inclk0 : IN STD_LOGIC;
c0 : OUT STD_LOGIC;
c1: OUT STD_LOGIC );
END COMPONENT;

component I2C_AV_Config
port ( iCLK : in std_logic;
   iRST_N : in std_logic;
   I2C_SCLK : out std_logic;
   I2C_SDAT: inout std_logic
   );
end component;

component TD_Detect
port ( oTD_Stable: out std_logic;
   iTD_VS,iTD_HS,iRST_N : in std_logic
  );
end component;

component DIV
port ( aclr,clock : in std_logic;
denom: in std_logic_vector(3 downto 0);
numer: in std_logic_vector (9 downto 0);
quotient: out std_logic_vector (9 downto 0);
remain: out std_logic_vector (3 downto 0)
);
end component;

component Reset_Delay
port ( iCLK,
   iRST: in std_logic;
   oRST_0,
   oRST_1,
   oRST_2: out std_logic
 );
end component;

component ITU_656_Decoder
port ( iTD_DATA: in std_logic_vector (7 downto 0);
   oTV_X,opixelno,oTV_Y: out std_logic_vector (9 downto 0);
   oYCbCr: out std_logic_vector (3 downto 0);
   oDVAL: out std_logic;
   oField: out std_logic;
   iSwap_CbCr,iSkip,iRST_N,iCLK_27: in std_logic
  );
end component;

signal clk25 : std_logic := '0';
```

```
--TD_DETECT OUTPUT SIGNALS
signal myTD_Stable: std_logic;

-- RESET DELAY OUTPUT SIGNALS
signal myDLY0,myDLY1,myDLY2: std_logic;

-- DIVIDER OUTPUT SIGNALS
signal myQuotient:std_logic_vector(9 downto 0);
signal myRemain: std_logic_vector(3 downto 0);

-- ITU DECODER OUTPUT SIGNALS
signal myTV_X: std_logic_vector (9 downto 0);
signal mypixelno: std_logic_vector (9 downto 0);
signal myTV_Y: std_logic_vector (9 downto 0);
signal myYCbCr: std_logic_vector (3 downto 0);
signal myDVAL: std_logic;
signal myField: std_logic;

-- SDRAM INTERMEDIARY SIGNALS
signal BA : STD_LOGIC_VECTOR(1 DOWNTO 0);
signal DQM : STD_LOGIC_VECTOR(1 DOWNTO 0);

-- FOR ITU
signal mySkip: std_logic;

-- ARBITRATOR OUTPUT / VGA INPUT
signal myRed, myGreen, myBlue: std_logic_vector (9 downto 0);
signal myPixelCLK: std_logic;

-- FROM VGA Controller
signal myHCount, myVcount: unsigned (9 downto 0);

signal enetClk: std_logic;

-- DMA Controller STUFF
signal busy_to_dma_from_arbitrator: std_logic;
signal data_to_dma_from_linebuffer: std_logic_vector(15 downto 0);
signal linebufferfull_to_dma_from_linebuffer: std_logic;
signal address_from_dma_to_arbitrator: std_logic_vector(17 downto 0);
signal address_from_dma_to_linebuffer: std_logic_vector(6 downto 0);
    signal data_from_dma_to_arbitrator: std_logic_vector(15 downto 0);
    signal data_ready_from_dma_to_arbitrator: std_logic;
signal field_from_linebuffer_to_dma: std_logic;
signal linenumber_from_linebuffer_to_dma: std_logic_vector (9 downto 0);

signal clk100: std_logic;

--signal myDVAL,myRESET,myCLK: std_logic;
--signal myY: std_logic_vector (7 downto 0);
--
--signal oYCbCr: std_logic_vector (3 downto 0);
--
--signal oTV_Cont: std_logic_vector (31 downto 0);
--signal oDVAL: std_logic;
--

--
--signal myRed, myGreen, myBlue: std_logic_vector (9 downto 0);
```

```
--signal myOVAL: std_logic;

--
--signal myTD_Stable,DLY0,DLY1,DLY2: std_logic;


begin

DRAM_BA_1 <= BA(1);
DRAM_BA_0 <= BA(0);

DRAM_UDQM <= DQM(1);
DRAM_LDQM <= DQM(0);

  process (CLOCK_50)
  begin
    if rising_edge(CLOCK_50) then
      clk25 <= not clk25;
    end if;
  end process;

ENET_CLK <= clk25;


u1: I2C_AV_Config port map (iCLK=> CLOCK_50,
    iRST_N => KEY(0),
I2C_SCLK => I2C_SCLK,
I2C_SDAT => I2C_SDAT);

u2: TD_Detect port map(myTD_Stable,
   TD_VS,
   TD_HS,
   KEY(0));

u3: Reset_Delay port map (CLOCK_50,
  myTD_Stable,
  myDLY0,
  myDLY1,
  myDLY2);

u4: DIV port map (not myDLY0,
  CLOCK_27,
  "1001",
  myTV_X,
  myQuotient,
  myRemain);

u5: ITU_656_Decoder port map (TD_DATA,
  myTV_X,
  mypixelno,
  myTV_Y,
  myYCbCr,
  myDVAL,
  myField,
  myQuotient(0),
  mySkip,
  myDLY1,
  CLOCK_27);
```

```
neg_3ns: sdram_pll PORT MAP (CLOCK_50, clk100, DRAM_CLK);
LEDG (7 downto 0) <= TD_DATA;
mySkip <= '1' when myRemain = "0000" else '0';
--
vga: work.de2_vga_raster PORT MAP (reset => '0', clk => CLOCK_50, pixel_clock => myPixelCLK, RED => 
  VGA_CLK => VGA_CLK, VGA_HS => VGA_HS,VGA_VS => VGA_VS,
  VGA_BLANK => VGA_BLANK,VGA_SYNC => VGA_SYNC,VGA_R => VGA_R,
  VGA_G => VGA_G,VGA_B => VGA_B, HCOUNT_OUT => myHCount, VCOUNT_OUT => myVCount);

dmactrl: work.DMACONTROLLER PORT MAP (clk_50 => CLOCK_50, linebufferfull_from_lb => linebufferfull_t
  data_from_lb => data_to_dma_from_linebuffer,
  busy_from_arb =>  busy_to_dma_from_arbitrator,
  address_to_lb => address_from_dma_to_linebuffer,
  address_to_arb => address_from_dma_to_arbitrator,
  data_to_arb => data_from_dma_to_arbitrator,
  data_ready_to_arb => data_ready_from_dma_to_arbitrator,
  field_from_lb => field_from_linebuffer_to_dma,
  lineNumber_from_lb => linenumber_from_linebuffer_to_dma);

  a1 : entity work.nios_0
    port map(
      ENET_CMD_from_the_DM9000A => ENET_CMD,
      ENET_CS_N_from_the_DM9000A => ENET_CS_N,
      ENET_DATA_to_and_from_the_DM9000A => ENET_DATA,
      ENET_RD_N_from_the_DM9000A => ENET_RD_N,
      ENET_RST_N_from_the_DM9000A => ENET_RST_N,
      ENET_WR_N_from_the_DM9000A => ENET_WR_N,
  ENET_INT_to_the_DM9000A => ENET_INT,
  --iRST_N_to_the_DM9000A => '1',
      BLUE_from_the_arbitrator_inst => myBlue,
      GREEN_from_the_arbitrator_inst => myGreen,
      RED_from_the_arbitrator_inst => myRed,
      SRAM_ADDR_from_the_arbitrator_inst => SRAM_ADDR,
      SRAM_CE_N_from_the_arbitrator_inst => SRAM_CE_N,
      SRAM_DQ_to_and_from_the_arbitrator_inst => SRAM_DQ,
      SRAM_LB_N_from_the_arbitrator_inst => SRAM_LB_N,
      SRAM_OE_N_from_the_arbitrator_inst => SRAM_OE_N,
      SRAM_UB_N_from_the_arbitrator_inst => SRAM_UB_N,
      SRAM_WE_N_from_the_arbitrator_inst => SRAM_WE_N,
  pixel_clock_from_the_arbitrator_inst => myPixelCLK,
      zs_addr_from_the_sdram => DRAM_ADDR,
      zs_ba_from_the_sdram => BA,
      zs_cas_n_from_the_sdram => DRAM_CAS_N,
      zs_cke_from_the_sdram => DRAM_CKE,
      zs_cs_n_from_the_sdram => DRAM_CS_N,
      zs_dq_to_and_from_the_sdram => DRAM_DQ,
      zs_dqm_from_the_sdram => DQM,
      zs_ras_n_from_the_sdram => DRAM_RAS_N,
      zs_we_n_from_the_sdram => DRAM_WE_N,
  Field_to_the_linebuffer_inst => myField,
      HCOUNT_to_the_arbitrator_inst => std_logic_vector(myHCount),
      VCOUNT_to_the_arbitrator_inst => std_logic_vector(myVCount),
      clk => CLOCK_50,
      clk_27_to_the_linebuffer_inst => CLOCK_27,
      datavalid_to_the_linebuffer_inst => myDVAL,
      reset_n => '1',
      xpos_to_the_linebuffer_inst => mypixelno,
```

```
      y_to_the_linebuffer_inst => myYCbCr,
      ypos_to_the_linebuffer_inst => myTV_Y,
clk_1 => clk100,
clk_100_to_the_linebuffer_inst => clk100,
      busy_to_dma_from_the_arbitrator_inst => busy_to_dma_from_arbitrator,
      data_to_dma_from_the_linebuffer_inst => data_to_dma_from_linebuffer,
      linebufferfull_to_dma_from_the_linebuffer_inst => linebufferfull_to_dma_from_linebuffer,
  address_from_dma_to_the_arbitrator_inst => address_from_dma_to_arbitrator,
      address_from_dma_to_the_linebuffer_inst => address_from_dma_to_linebuffer,
      data_from_dma_to_the_arbitrator_inst => data_from_dma_to_arbitrator,
      data_ready_from_dma_to_the_arbitrator_inst => data_ready_from_dma_to_arbitrator,
      field_to_dma_from_the_linebuffer_inst => field_from_linebuffer_to_dma,
  linenumber_to_dma_from_the_linebuffer_inst => linenumber_from_linebuffer_to_dma
    );

--  HEX7     <= "0001001"; -- Leftmost
--  HEX6     <= "0000110";
--  HEX5     <= "1000111";
--  HEX4     <= "1000111";
--  HEX3     <= "1000000";
--  HEX2     <= (others => '1');
--  HEX1     <= (others => '1');
--  HEX0     <= (others => '1');          -- Rightmost
  --LEDG     <= (others => '1');
  LEDR     <= (others => '1');
  LCD_ON   <= '1';
  LCD_BLON <= '1';
  LCD_RW <= '1';
  LCD_EN <= '0';
  LCD_RS <= '0';

  SD_DAT3 <= '1';
  SD_CMD <= '1';
  SD_CLK <= '1';

--  SRAM_DQ <= (others => 'Z');
--  SRAM_ADDR <= (others => '0');
--  SRAM_UB_N <= '1';
--  SRAM_LB_N <= '1';
--  SRAM_CE_N <= '1';
--  SRAM_WE_N <= '1';
--  SRAM_OE_N <= '1';

  UART_TXD <= '0';
--  DRAM_ADDR <= (others => '0');
--  DRAM_LDQM <= '0';
--  DRAM_UDQM <= '0';
--  DRAM_WE_N <= '1';
--  DRAM_CAS_N <= '1';
--  DRAM_RAS_N <= '1';
--  DRAM_CS_N <= '1';
--  DRAM_BA_0 <= '0';
--  DRAM_BA_1 <= '0';
--  DRAM_CLK <= '0';
--  DRAM_CKE <= '0';
  FL_ADDR <= (others => '0');
  FL_WE_N <= '1';
  FL_RST_N <= '0';
```

```
  FL_OE_N <= '1';
  FL_CE_N <= '1';
  OTG_ADDR <= (others => '0');
  OTG_CS_N <= '1';
  OTG_RD_N <= '1';
  OTG_RD_N <= '1';
  OTG_WR_N <= '1';
  OTG_RST_N <= '1';
  OTG_FSPEED <= '1';
  OTG_LSPEED <= '1';
  OTG_DACK0_N <= '1';
  OTG_DACK1_N <= '1';

  TDO <= '0';

--  ENET_CMD <= '0';
--  ENET_CS_N <= '1';
--  ENET_WR_N <= '1';
--  ENET_RD_N <= '1';
--  ENET_RST_N <= '1';
--  ENET_CLK <= '0';

  TD_RESET <= KEY(0); -- IMPORTANT !!!!!!

  --I2C_SCLK <= '1';

  AUD_DACDAT <= '1';
  AUD_XCK <= '1';

  -- Set all bidirectional ports to tri-state
  DRAM_DQ     <= (others => 'Z');
  FL_DQ       <= (others => 'Z');
  --SRAM_DQ     <= (others => 'Z');
  OTG_DATA    <= (others => 'Z');
  LCD_DATA    <= (others => 'Z');
  SD_DAT      <= 'Z';
  I2C_SDAT    <= 'Z';
  ENET_DATA   <= (others => 'Z');
  AUD_ADCLRCK <= 'Z';
  AUD_DACLRCK <= 'Z';
  AUD_BCLK    <= 'Z';
  GPIO_0      <= (others => 'Z');
  GPIO_1      <= (others => 'Z');


end datapath;
```

4.12 C - hello_world.c

```
#include <io.h>
#include <system.h>
#include <stdio.h>
#include "DM9000A.h"
#include "basic_io.h"

#define LINEBUFFER_LENGTH 80

#define UDP_PACKET_PAYLOAD_OFFSET 42
```

```
#define EIGHT_LINE_DATA_LENGTH 1304

short lineBuffer[LINEBUFFER_LENGTH];
short lineCheck[240];

unsigned char mac_address[6] = { 0x11, 0x60, 0x6E, 0x11, 0x02, 0x0F  };
unsigned int receive_buffer_length;
unsigned char receive_buffer[MAX_PACKET_SIZE];
unsigned int interrupt_number = 0;

unsigned char transmit_buffer[] =
{
  // Ethernet MAC header
  0x01, 0x60, 0x6E, 0x11, 0x02, 0x0F, // Destination MAC address
  0x11, 0x60, 0x6E, 0x11, 0x02, 0x0F, // Source MAC address
  0x08, 0x00,                         // Packet Type: 0x800 = IP

  // IP Header
  0x45,                // version (IPv4), header length = 20 bytes
  0x00,                // differentiated services field
  0x05,0x34,           // total length: 20 bytes for IP header +
                       // 8 bytes for UDP header + 1304 bytes for payload
  0x3d, 0x35,          // packet ID
  0x00,                // flags
  0x00,                // fragment offset
  0x80,                // time-to-live
  0x11,                // protocol: 11 = UDP
  0xa3,0x43,           // header checksum: incorrect
  0xc0,0xa8,0x01,0x01, // source IP address
  0xc0,0xa8,0x01,0xFF, // destination IP address

  // UDP Header
  0x67,0xd9, // source port port (26585: garbage)
  0x27,0x2b, // destination port (10027: garbage)
  0x05,0x20, // length (136: 8 for UDP header + 1304 for data)
  0x00,0x00, // checksum: 0 = none

  /* UDP PAYLOAD STARTS HERE */

  // Line Data (160 bytes)
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

// Line Number (2 bytes)
0x00, 0x00,

// Field the line belongs in (1 bit)
0x00,

// Line Data (160 bytes)
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

// Line Number (2 bytes)
0x00, 0x00,

// Field the line belongs in (1 bit)
0x00,

// Line Data (160 bytes)
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
// Line Number (2 bytes)
0x00, 0x00,

// Field the line belongs in (1 bit)
0x00,

// Line Data (160 bytes)
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

// Line Number (2 bytes)
0x00, 0x00,

// Field the line belongs in (1 bit)
0x00,

// Line Data (160 bytes)
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

// Line Number (2 bytes)
```

```
0x00, 0x00,

// Field the line belongs in (1 bit)
0x00,

// Line Data (160 bytes)
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

// Line Number (2 bytes)
0x00, 0x00,

// Field the line belongs in (1 bit)
0x00,

// Line Data (160 bytes)
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

// Line Number (2 bytes)
0x00, 0x00,
```

```
  // Field the line belongs in (1 bit)
  0x00,

  // Line Data (160 bytes)
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

  // Line Number (2 bytes)
  0x00, 0x00,

  // Field the line belongs in (1 bit)
  0x00
};

int main()
{
    int i=0;
    short read = 0;
    int currentLineNumber = -1;
    int sramAddress = 0;
    int field = 0;   // 0 = even, 1 = odd
    unsigned int z = 0;
    int cnt = 0;
    unsigned int receive_status;
    int transmitBufferLineCount = 0;
    int transmitBufferIndex = UDP_PACKET_PAYLOAD_OFFSET;

    printf("%d\n",DM9000_init(mac_address));
    msleep(1500);

    // White out the SRAM Frame Buffer. 19199 is the address that contains the
    // the last four pixels of the 320th line.
    while (z != 19199*2)
    {
         IOWR_16DIRECT(ARBITRATOR_INST_BASE, z*2, 0x0000);
         z++;
    }

    while(1)
    {
```

```
do
{
    read = IORD_16DIRECT(LINEBUFFER_INST_BASE, 160);


}
while ((read & 0x0001) != 1);

currentLineNumber = ((read & 0x07FE) >> 1);
field = (read & 0x8000) >> 15;

    for (i=0; i<LINEBUFFER_LENGTH; i++)
    {
        //Read the line buffer
        lineBuffer[i] =IORD_16DIRECT(LINEBUFFER_INST_BASE, i*2);

        transmit_buffer[transmitBufferIndex] =  (lineBuffer[i] & 0xF000) >> 8;
        transmit_buffer[transmitBufferIndex] += (lineBuffer[i] & 0x0F00) >> 8;
        transmit_buffer[transmitBufferIndex+1] = (lineBuffer[i] & 0x00F0);
        transmit_buffer[transmitBufferIndex+1] += (lineBuffer[i] & 0x000F);
        transmitBufferIndex += 2;

        // end of the line...no pun intended
        if (i==LINEBUFFER_LENGTH-1)
        {
            transmitBufferLineCount++;
            transmit_buffer[transmitBufferIndex] = 0x00;
            transmit_buffer[transmitBufferIndex+1] = currentLineNumber;
            transmit_buffer[transmitBufferIndex+2] = field;
            transmitBufferIndex += 3;
            transmitBufferLineCount++;
        }

        if (transmitBufferLineCount == 8)
        {
            transmitBufferLineCount = 0;
            TransmitPacket(transmit_buffer,UDP_PACKET_PAYLOAD_OFFSET+EIGHT_LINE_DATA_LENGTH,
            cnt++;
            transmitBufferIndex = UDP_PACKET_PAYLOAD_OFFSET;
        }
    }

receive_status = ReceivePacket(receive_buffer);
int i=0;
int j=0;
int leftBound = 0;
int rightBound = 0;

if (receive_status == DMFE_SUCCESS)
{
    // UDP check
    if (receive_buffer[23] == 0x11)
    {
        // Eight lines per packet
        for (i=0;i<8;i++)
        {
            if (i==0)
                leftBound = UDP_PACKET_PAYLOAD_OFFSET;
```

```
                    else
                        leftBound = rightBound + 3;

                    rightBound = 160 + leftBound;
                    int lineNumber = receive_buffer[rightBound+1];
                    int field = receive_buffer[rightBound+2];

                    lineNumber += 240;
                    int lineIndex = 0;

                    for (j=leftBound; j<rightBound; j+=2)
                    {
                        unsigned short line = (receive_buffer[j] << 8) + receive_buffer[j+1];

                        // Calculate the SRAM address
                        if (field == 1)
                        {
                             // Even line
                            sramAddress = (lineNumber)*80 + lineIndex;
                        }

                        else
                        {
                           sramAddress = (lineNumber+1)*80 + lineIndex;
                        }

                        lineIndex++;
                        IOWR_16DIRECT(ARBITRATOR_INST_BASE, (sramAddress*2), line);
                    }
                }
            }
        }

     }

    return 0;
}
```

4.13 C - DM9000A.c

```
#include <stdio.h>
#include "DM9000A.h"
#include "basic_io.h"
int warmup  = 0;

void dm9000a_iow(unsigned int reg, unsigned int data)
{
  IOWR(DM9000A_BASE, IO_addr, reg);
  asm("nop");
  asm("nop");
  asm("nop");

  IOWR(DM9000A_BASE, IO_data, data);
}

unsigned int dm9000a_ior(unsigned int reg)
{
  IOWR(DM9000A_BASE, IO_addr, reg);
```

```
  asm("nop");
  asm("nop");
  asm("nop");
  return IORD(DM9000A_BASE, IO_data);
}

void dm9000a_iow_init(unsigned int reg, unsigned int data)
{
  IOWR(DM9000A_BASE, IO_addr, reg);
  //usleep(STD_DELAY);
usleep(STD_DELAY);
  IOWR(DM9000A_BASE, IO_data, data);
}

unsigned int dm9000a_ior_init(unsigned int reg)
{
  IOWR(DM9000A_BASE, IO_addr, reg);
 usleep(STD_DELAY);
  return IORD(DM9000A_BASE, IO_data);
}


void phy_write(unsigned int reg, unsigned int value)
{
  /* set PHY register address into EPAR REG. 0CH */
  dm9000a_iow_init(0x0C, reg | 0x40); /* PHY register address setting,
              and DM9000_PHY offset = 0x40 */

  /* fill PHY WRITE data into EPDR REG. 0EH & REG. 0DH */
  dm9000a_iow_init(0x0E, ((value >> 8) & 0xFF));   /* PHY data high_byte */
  dm9000a_iow_init(0x0D, value & 0xFF);            /* PHY data low_byte */

  /* issue PHY + WRITE command = 0xa into EPCR REG. 0BH */
  dm9000a_iow_init(0x0B, 0x8);                     /* clear PHY command first */
  IOWR(DM9000A_BASE, IO_data, 0x0A);  /* issue PHY + WRITE command */
  usleep(STD_DELAY);
  IOWR(DM9000A_BASE, IO_data, 0x08);  /* clear PHY command again */
  usleep(50);  /* wait 1~30 us (>20 us) for PHY + WRITE completion */
}

/* DM9000_init I/O routine */
unsigned int DM9000_init(unsigned char *mac_address)
{
  unsigned int  i;

  /* set the internal PHY power-on (GPIOs normal settings) */
  dm9000a_iow_init(0x1E, 0x01);  /* GPCR REG. 1EH = 1 selected
             GPIO0 "output" port for internal PHY */
  dm9000a_iow_init(0x1F, 0x00);  /* GPR  REG. 1FH GEPIO0
             Bit [0] = 0 to activate internal PHY */
  msleep(5);        /* wait > 2 ms for PHY power-up ready */

  /* software-RESET NIC */
  dm9000a_iow_init(NCR, 0x03);   /* NCR REG. 00 RST Bit [0] = 1 reset on,
             and LBK Bit [2:1] = 01b MAC loopback on */
  usleep(20);       /* wait > 10us for a software-RESET ok */
  dm9000a_iow_init(NCR, 0x00);   /* normalize */
  dm9000a_iow_init(NCR, 0x03);
```

```
usleep(20);
dm9000a_iow_init(NCR, 0x00);

/* set GPIO0=1 then GPIO0=0 to turn off and on the internal PHY */
dm9000a_iow_init(0x1F, 0x01);  /* GPR PHYPD Bit [0] = 1 turn-off PHY */
dm9000a_iow_init(0x1F, 0x00);  /* PHYPD Bit [0] = 0 activate phyxcer */
msleep(10);       /* wait >4 ms for PHY power-up */

/* set PHY operation mode */
phy_write(0,PHY_reset);   /* reset PHY registers back to the default state */
usleep(50);               /* wait >30 us for PHY software-RESET ok */
phy_write(16, 0x404);     /* turn off PHY reduce-power-down mode only */
phy_write(4, PHY_txab);   /* set PHY TX advertised ability:
                 ALL + Flow_control */
phy_write(0, 0x1200);     /* PHY auto-NEGO re-start enable
                 (RESTART_AUTO_NEGOTIATION +
                 AUTO_NEGOTIATION_ENABLE)
                 to auto sense and recovery PHY registers */
msleep(5);                /* wait >2 ms for PHY auto-sense
                 linking to partner */

/* store MAC address into NIC */
for (i = 0; i < 6; i++)
  dm9000a_iow_init(16 + i, mac_address[i]);

/* clear any pending interrupt */
dm9000a_iow_init(ISR, 0x3F);  /* clear the ISR status: PRS, PTS, ROS, ROOS 4 bits,
            by RW/C1 */
dm9000a_iow_init(NSR, 0x2C);  /* clear the TX status: TX1END, TX2END, WAKEUP 3 bits,
            by RW/C1 */

/* program operating registers~ */
dm9000a_iow_init(NCR, NCR_set); /* NCR REG. 00 enable the chip functions
          (and disable this MAC loopback mode back to normal) */
dm9000a_iow_init(0x08, BPTR_set); /* BPTR  REG.08  (if necessary) RX Back Pressure
            Threshold in Half duplex moe only:
            High Water 3KB, 600 us */
dm9000a_iow_init(0x09, FCTR_set);  /* FCTR  REG.09  (if necessary)
             Flow Control Threshold setting
             High/ Low Water Overflow 5KB/ 10KB */
dm9000a_iow_init(0x0A, RTFCR_set); /* RTFCR REG.0AH (if necessary)
             RX/TX Flow Control Register enable TXPEN, BKPM
             (TX_Half), FLCE (RX) */
dm9000a_iow_init(0x0F, 0x00);      /* Clear the all Event */
dm9000a_iow_init(0x2D, 0x80);      /* Switch LED to mode 1 */

/* set other registers depending on applications */
dm9000a_iow_init(ETXCSR, ETXCSR_set); /* Early Transmit 75% */

/* enable interrupts to activate DM9000 ~on */
dm9000a_iow_init(IMR, INTR_set);   /* IMR REG. FFH PAR=1 only,
             or + PTM=1& PRM=1 enable RxTx interrupts */

/* enable RX (Broadcast/ ALL_MULTICAST) ~go */
dm9000a_iow_init(RCR , RCR_set | RX_ENABLE | PASS_MULTICAST);
/* RCR REG. 05 RXEN Bit [0] = 1 to enable the RX machine/ filter */

/* RETURN "DEVICE_SUCCESS" back to upper layer */
```

```
  return  (dm9000a_ior_init(0x2D)==0x80) ? DMFE_SUCCESS : DMFE_FAIL;
}

unsigned int TransmitPacket(unsigned char *data_ptr, unsigned int tx_len, int cnt)
{
  unsigned int i;

  /* mask NIC interrupts IMR: PAR only */
  dm9000a_iow(IMR, PAR_set);

  /* issue TX packet's length into TXPLH REG. FDH & TXPLL REG. FCH */
  dm9000a_iow(0xFD, (tx_len >> 8) & 0xFF);  /* TXPLH High_byte length */
  dm9000a_iow(0xFC, tx_len & 0xFF);         /* TXPLL Low_byte  length */

  /* wirte transmit data to chip SRAM */
  IOWR(DM9000A_BASE, IO_addr, MWCMD);  /* set MWCMD REG. F8H
                    TX I/O port ready */
  for (i = 0; i < tx_len; i += 2) {
    if (cnt < 50)
    {

    int z = 0;
    for (z=0; z<3; z++)
    {
        usleep(STD_DELAY);
    }
    }

    IOWR(DM9000A_BASE, IO_data, (data_ptr[i+1]<<8)|data_ptr[i] );
  }

  /* issue TX polling command activated */
  dm9000a_iow(TCR , TCR_set | TX_REQUEST);  /* TXCR Bit [0] TXREQ auto clear
                     after TX completed */

  /* wait TX transmit done */
  while(!(dm9000a_ior(NSR)&0x0C))
  {
    if (cnt < 50)
    {
        int z=0;
        for (z=0; z<3; z++)
        {
            usleep(STD_DELAY);
        }
    }
  }

  /* clear the NSR Register */
  dm9000a_iow(NSR,0x00);

  /* re-enable NIC interrupts */
  dm9000a_iow(IMR, INTR_set);

  /* RETURN "TX_SUCCESS" to upper layer */
  return  DMFE_SUCCESS;
}
```

```
unsigned int ReceivePacket(unsigned char *dataPtr)
{
    unsigned char status = dm9000a_ior(0xFE);
    int GoodPacket = FALSE;

    if (status & 0x01)
    {
        dm9000a_iow(0xFE,0x01);
        dm9000a_ior(0xF0);

        unsigned char rx_ready = IORD(DM9000A_BASE,IO_data);

        if ( (rx_ready & 0x01) == 1 )
        {
    IOWR(DM9000A_BASE,IO_addr,0xF2);

            usleep(STD_DELAY);

            unsigned int rx_status = 0;
            unsigned int rx_length = 0;

            rx_status = IORD(DM9000A_BASE,IO_data);
             usleep(STD_DELAY);
            rx_length = IORD(DM9000A_BASE,IO_data);

            int i=0;
            short data = 0;

            for (i=0; i<rx_length;i+=2)
            {
                data = IORD(DM9000A_BASE,IO_data);
                dataPtr[i] = data & 0xFF;


                dataPtr[i+1] = (data >> 8);
            }

            GoodPacket = TRUE;
        }

        else
        {
            dm9000a_iow(0xFF,0x80);
            dm9000a_iow(0xFE,0x0F);
            dm9000a_iow(0x05,0x00);

            dm9000a_iow(0x00, 0x01);
            usleep(STD_DELAY);

            dm9000a_iow(0x00, NCR_set);
            dm9000a_iow(0xFF, 0x80);
            dm9000a_iow(0x05, RCR_set | 1);

            GoodPacket = FALSE;
        }
    }

    return GoodPacket ? DMFE_SUCCESS : DMFE_FAIL;
```

```
}
```

4.14 C - DM9000A.h

```
#ifndef   __DM9000A_H__
#define   __DM9000A_H__

#define IO_addr     0
#define IO_data     1

#define NCR    0x00  /* Network  Control Register REG. 00 */
#define NSR    0x01  /* Network  Status Register  REG. 01 */
#define TCR    0x02  /* Transmit Control Register REG. 02 */
#define RCR    0x05  /* Receive  Control Register REG. 05 */
#define ETXCSR 0x30  /* TX early Control Register REG. 30 */
#define MRCMDX 0xF0  /* RX FIFO I/O port command: READ a byte from RX SRAM */
#define MRCMD  0xF2  /* RX FIFO I/O port command READ  from RX SRAM */
#define MWCMD  0xF8  /* TX FIFO I/O port command WRITE into TX FIFO */
#define ISR    0xFE  /* NIC Interrupt Status Register REG. FEH */
#define IMR    0xFF  /* NIC Interrupt Mask   Register REG. FFH */

#define NCR_set    0x00
#define TCR_set    0x00
#define TX_REQUEST 0x01  /* TCR REG. 02 TXREQ Bit [0] = 1 polling
    Transmit Request command */
#define TCR_long   0x40  /* packet disable TX Jabber Timer */
#define RCR_set    0x30  /* skip CRC_packet and skip LONG_packet */
#define RX_ENABLE  0x01  /* RCR REG. 05 RXEN
    Bit [0] = 1 to enable RX machine */
#define RCR_long   0x40  /* packet disable RX Watchdog Timer */
#define PASS_MULTICAST 0x08 /* RCR REG. 05 PASS_ALL_MULTICAST
    Bit [3] = 1: RCR_set value ORed 0x08 */
#define BPTR_set   0x3F  /* BPTR REG. 08 RX Back Pressure Threshold:
    High Water Overflow Threshold setting
    3KB and Jam_Pattern_Time = 600 us */
#define FCTR_set   0x5A  /* FCTR REG. 09 High/ Low Water Overflow Threshold
    setting 5KB/ 10KB */
#define RTFCR_set  0x29  /* RTFCR REG. 0AH RX/TX Flow Control Register
    enable TXPEN + BKPM(TX_Half) + FLCE(RX) */
#define ETXCSR_set 0x83  /* Early Transmit Bit [7] Enable and
    Threshold 0~3: 12.5%, 25%, 50%, 75% */
#define INTR_set   0x81  /* IMR REG. FFH: PAR +PRM, or 0x83: PAR + PRM + PTM */
#define PAR_set    0x80  /* IMR REG. FFH: PAR only, RX/TX FIFO R/W
    Pointer Auto Return enable */

#define PHY_reset  0x8000  /* PHY reset: some registers back to
      default value */
#define PHY_txab   0x05e1  /* set PHY TX advertised ability:
      Full-capability + Flow-control (if necessary) */
#define PHY_mode   0x3100  /* set PHY media mode: Auto negotiation
      (AUTO sense) */

// #define STD_DELAY       20      /* standard delay 20 us */
#define STD_DELAY 40

#define DMFE_SUCCESS    0
#define DMFE_FAIL       1
```

```
#define TRUE            1
#define FALSE           0

#define DM9000_PKT_READY  0x01  /* packets ready to receive */
#define PACKET_MIN_SIZE   0x40  /* Received packet min size */
#define MAX_PACKET_SIZE   1522  /* RX largest legal size packet
   with fcs & QoS */
#define DM9000_PKT_MAX    3072  /* TX 1 packet max size without 4-byte CRC */
//-----------------------------------------------------------------------
extern void          dm9000a_iow(unsigned int reg, unsigned int data);
extern unsigned int  dm9000a_ior(unsigned int reg);
extern void          phy_write(unsigned int reg, unsigned int value);

/* DM9000_init I/O routine */
extern unsigned int  DM9000_init(unsigned char *mac_address);

/* Transmit One Packet TX I/O routine */
extern unsigned int  TransmitPacket(unsigned char *data_ptr,
    unsigned int tx_len, int cnt);

/* Receive One Packet I/O routine */
extern unsigned int  ReceivePacket(unsigned char *data_ptr);
//-----------------------------------------------------------------------

#endif
```