Embedded Systems Lab CSEE 4840 : Imagic Design Document

Abhilash Itharaju Nalini Vasudevan Vitaliy Shchupak Walter Dearing

Abstract

We have two goals for our project. The basic goal is to read the contents of SD card, using the SD Card reader on the DE2 board, decode and display all the JPEG images in it on the screen one after the other as a slideshow using the onboard VGA DAC. The next and more aggressive goal once this is achieved is to have effects in the slide show like fading, bouncing etc.

1 Introduction

The basic idea is to have two peripherals, 1) to control the onboard SD card reader and 2) to control the VGA DAC. The function of first peripheral is to talk to SD card reader and to read things from it. The other peripheral is to interact with VGA DAC and provide it with the proper signals (pixel data, H_SYNC, V_SYNC, BLANKING). The main function of the software is to initialize and control the peripherals and also to decode the JPEG image once it is read from the SD card. The top level idea is to have two memory locations. One is where the program sits (SDRAM) and the other (SRAM) is where the image buffer is kept so that the video peripheral can read from it.

2 Hardware Design

The hardware is broken into four main components:

1. SDRAM
2. VGA Peripheral
3. SD Card Controller
4. Nios II System

The four main components communicate over the Avalon Bus. The clock signal used is 50 MHz. The image is stored on the SD Card. The Nios II processor reads in this image and decompresses it. It then sends

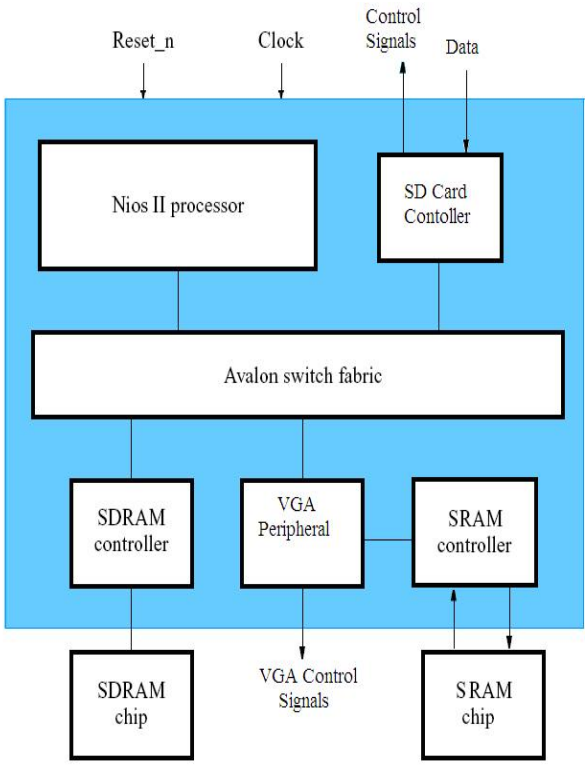

Figure 1: Block Diagram of Imagic

the image to the VGA peripheral. The VGA peripheral stores the data sent from the NIOS into SRAM. The VGA peripheral then reads the SRAM every 25 MHz for a new pixel to display on the screen.

Because each component of the system can be treated as a black box, each item will be discussed in detail separately.

2.1 Interface with MMC/SD Card

We will be communicating with the MMC/SD Card in the SPI mode. Since SD Card is backward compatible with MMC card, the interface is the same. This is how we can make the NIOS-II processor talk to the MMC card.

The following figure shows how the FPGA is connected to the SD Card slot.

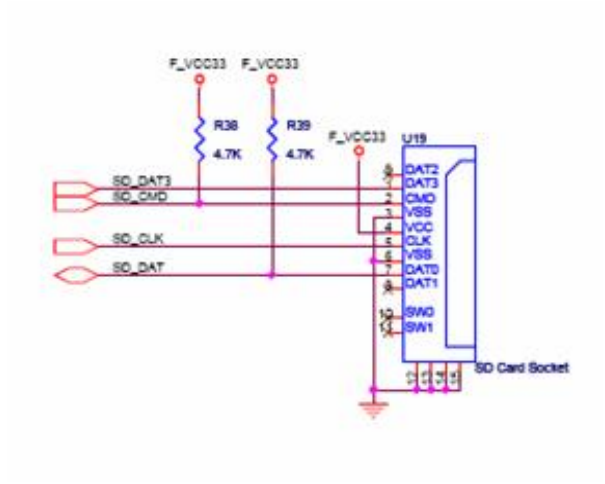

Figure 2: FPGA Connection to the SD Card Slot

For the SPI mode we need all the 4 pins.

2.1.1 SPI Commands

Communications between the microcontroller and the MMC are initiated by different commands sent from the FPGA. All commands are 6 bytes long and are transmitted MSB first.

The following are the list of commands that can be sent to the MMC Card.

CRC CALCULATION

The CRC bit calculation is performed:

BYTE 1	7	6	5	4	3	2	1	0
FIELD	0	1	Command					

BYTES 2 - 5	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	Argument															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	Argument															

BYTE 6	7	6	5	4	3	2	1	0
FIELD	CRC						1	x

Figure 3: SPI Commands

CMD INDEX	ARGUMENT	RESPONSE	ABBREVIATION	COMMAND DESCRIPTION
CMD0	None	R1	GO_IDLE_STATE	Resets the MultiMediaCard
CMD1	None	R1	SEND_OP_COND	Activates the card Initialization process
CMD13	None	R2	SEND_STATUS	Asks the selected card to send its status register
CMD16	[31:0]block length	R1	SET_BLOCKLEN	Selects a block length (in bytes) for all following block commands (read and write).
CMD17	[31:0]data address	R1	READ_SINGLE_BLOCK	Reads a block of size selected by the SET_BLOCKLEN command
CMD24	[31:0]data address	R1	WRITE_BLOCK	Writes a block of the size selected by the SET_BLOCKLEN command
CMD32	[31:0]data address	R1	TAG_SECTOR_START	Sets the address of the first sector of the erase group
CMD33	[31:0]data address	R1	TAG_SECTOR_END	Sets the address of the last sector in a continuous range within the selected erase group, or the address of a single sector to be selected for erase.
CMD34	[31:0]data address	R1	UNTAG_SECTOR	Removes one previously selected sector from the erase selection
CMD38	[31:0]don't care	R1b	ERASE	Erases all previously selected sectors
CMD59	[31:1]don't care [0:0]CRC option	R1	CRC_ON_OFF	Turns the CRC option on or off. A '1' in the CRC option bit will turn the option on. A '0' will turn it off.

Figure 4: FPGA Connection to the SD Card Slot

7 bit CRC Calculation: G(x) = x7 + x3 + 1
M(x) = (start bit) x39 + (second bit) x38+ ...+ (last bit before CRC) x0
CRC[6..0] = Remainder[(M(x) x7)/G(x)]

RESPONSE FORMAT R1

This response token is sent by the card after every command with the exception of SEND_STATUS commands. It is 1 byte long; the MSB is always set to zero and the other bits are error indications. A 1 signals an error.

Here is the timing of various commands and responses:

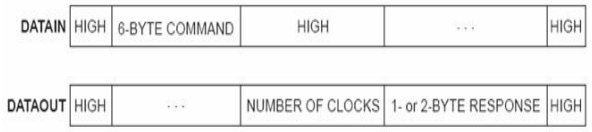

Figure 5: Command and Response when card is not busy

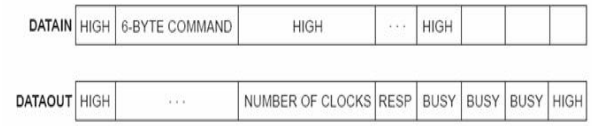

Figure 6: Command and Response when card is busy

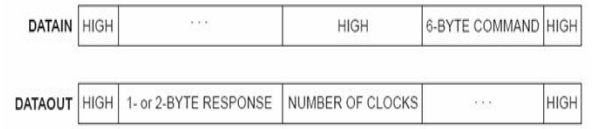

Figure 7: Card response to command from host

CLOCK CONTROL
The SPI bus clock signal can be used by the SPI host to set the cards to energy saving mode or to control data flow (to avoid under-run or over-run conditions) on the bus. The host is allowed to change the clock frequency or stop it altogether. There are a few restrictions the SPI host must follow:

- The bus frequency can be changed at any time, but only up to the maximum data transfer frequency, defined by the MultiMediaCards.
- It is an obvious requirement that the clock must be running for the MultiMediaCard to output

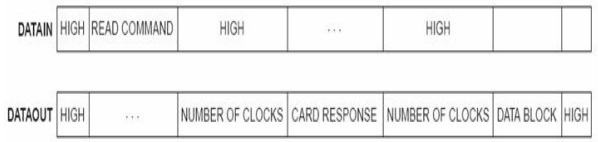

Figure 8: Data read command and response

data or response tokens. After the last SPI bus transaction, the host is required to provide 8 clock cycles for the card to complete the operation before shutting down the clock. During this 8-clock period, the state of the CS signal is irrelevant. It can be asserted or de-asserted. SPI BUS TRANSACTIONS

Here is a list of the various SPI bus transactions:

- A command/response sequence. Eight clocks must be output after the card response end bit. The CS signal can be asserted or de-asserted during these 8 clocks.
- A read data transaction. Eight clocks must be output after the end bit of the last data block.
- A write data transaction. Eight clocks must be output after the end bit of the last data block.
- A write data transaction. Eight clocks must be output after the CRC status token.
- The host is allowed to stop the clock of a BUSY card. The MultiMediaCard will complete the programming operation regardless of the host clock. However, the host must provide a clock edge for the card to turn off its BUSY signal. Without a clock edge, the MultiMediaCard (unless previously disconnected by deasserting the CS signal) will force the DataOut line LOW and hold it there.

2.1.2 MODE SELECTION

The MultiMediaCards SPI mode is the mode used for this Application Note. All transactions described in this Application Note are based on the SPI mode. The MultiMediaCard wakes up in the MultiMedia- Card mode. It will enter SPI mode if the CS signal (pin1 of the MMC) is asserted LOW during the reception of

the Reset command (CMD0). If the card is in MultiMediaCard mode, it will not respond to SPI-based commands.

If SPI mode is requested, the card will switch to SPI mode and respond with the SPI mode R1 response. To return to the MultiMediaCard mode, power cycle the card. In SPI mode, the MultiMediaCard protocol state machine is not observed. MultiMediaCard commands supported in SPI mode are always available. Since the card defaults to MultiMediaCard mode after a power cycle, Pin 1 (CS) must be pulled LOW and CMD0 (followed by a valid CRC byte) must be sent on the CMD (DataIn, Pin 2) line for the card to enter SPI mode. In SPI mode, CRC checking is disabled by default. However, since the card always powers up in MultiMediaCard mode, CMD0 must be followed by a valid CRC byte (even though the command is sent using the SPI structure). Once the card enters SPI mode, CRCs are disabled by default. CMD0 is a static command and always generates the same 7-bit CRC of 4Ah. Adding the 1 end bit (bit 0) to the CRC creates a CRC byte of 95h. The following hexadecimal sequence can be used to send CMD0 in all situations for SPI mode, since the CRC byte (although required) is ignored once in SPI mode. The entire CMD0 appears as: 40 00 00 00 00 95 (hexadecimal).

CMD0 is a static command and always generates the same 7-bit CRC of 4Ah. Adding the 1 end bit (bit 0) to the CRC creates a CRC byte of 95h. The following hexadecimal sequence can be used to send CMD0 in all situations for SPI mode, since the CRC byte (although required) is ignored once in SPI mode. The entire CMD0 appears as: 40 00 00 00 00 95 (hexadecimal).

RESET SEQUENCE

The initialization command is described in the following sequence:

1. Send 80 clocks to start bus communication
2. Assert nCS LOW
3. Send CMD0
4. Send 8 clocks for delay
5. Wait for a valid response
6. If there is no response, back to step 4
7. Send 8 clocks of delay
8. Send CMD1
9. Send 8 clocks of delay
10. Wait for valid response
11. Send 8 clocks of delay
12. Repeat from step 9 until the response shows READY.

It will take a large number of cycles for CMD1 to finish its sequence. After every power cycle, the MMC will be in Idle state (not active), the Idle bit in its response will be 1 if using CMD13 (SEND_STATUS) to check the status. Once the CMD1 process is finished, the Idle bit in the response is cleared. Only after MMC is fully up from Idle mode to Active, can it be read and written.

DATA READ

The SPI mode supports single block read operations only. Upon reception of a valid Read command, the card will respond with a Response token followed by a Data token in the length defined by a previous SET_BLOCK_LENGTH command. The start address can be any byte address in the valid address range of the card. Every block however, must be contained in a single physical card sector. After the Data Read command is sent from FPGA to the card, the FPGA will need to monitor the data stream input and wait for Data Token 0xFE. Since the response start bit 0 can happen any time in the clock stream, its necessary to use software to align the bytes being read.

2.1.3 Implementation

Right now, there is peripheral for each of the pins. So, there are four peripherals in total connected to the NIOS-II processor. Everything else is software controlled. Each of the pins is either set high or low by the software directly and that is how the above interface protocol is followed.

2.2 SDRAM

The original plan was to use the SRAM to store the picture and the C code used to retrieve the image from the SD Card and decompress the image. While in development, it was realized early on that the C code needed for the project was going to be bigger

than expected. Therefore, the decision was made to use the SDRAM to store the C code, so that the code size would not be an issue. The SRAM would still be used to store the image.

The SDRAM was setup using the tutorial supplied by Altera [1]. The directions were straight forward. The setup specifies to instantiate the SDRAM controller in the SOPC builder when building the NIOS II System. The next step was to connect the SDRAM chip to the top level VHDL file, so that the SDRAM controller could be connected to the SDRAM. Using the top level design file and pin assignment file from Lab3, the only thing that had to be done was route the SDRAM to the NIOS II system in the corresponding port map. Most of the pin assignments were straightforward; however, the clock needed one special step. Because there may be potential clock skew between the SDRAM and NIOS system due to the physical characteristics of the board, a phase-lock loop is needed to so that the SDRAM clock signal leads the NIOS II System by 3 nano-seconds. Again, using the tutorial, this was straight forward. A separate phase locked loop component was produced that took the 50 Mhz clock as an input and returned the clock signal that could be routed to the SDRAM.

2.3 VGA Peripheral

The VGA peripheral is responsible for controlling the VGA(using the VGA controller) and is also responsible for reading/writing the SRAM. The peripheral accepts data from the NIOS system. The NIOS II System can write four different types of data to the peripheral. The VGA peripheral distinguishes the different type of data by what address the NIOS II writes to. Currently there are five bits to use for the address.

Address Written To	Data Type
0x00	Data for SRAM
0x01	Address to write to in SRAM
0x02	Address to write to in SRAM
0x03	Picture Width
0x04	Picture Length

Table 1: Specifying data to the VGA Peripheral

As already indicated, the VGA peripheral has sole control of the SRAM. The first thing that the NIOS system will send is the picture width and length. This will be stored in registers (using VHDL SIGNAL), so that it can be used later. Next the NIOS will send the first address that should be written in the SRAM. This data will also be latched to be used later. Finally the NIOS will send the first data value. This data will be written to the SRAM at whatever address was previously specified. The NIOS will then continuously send SRAM addresses then SRAM data until the entire image is written to SRAM.

The data sent from the NIOS II system is a 15 bit value that is used to specify one pixel point. The VGA controller expects RGB values where 9 bits specify each color. To save space, only five bits are used for each color. The bottom four bits are set to zero and the upper five bits of each color is specified as:

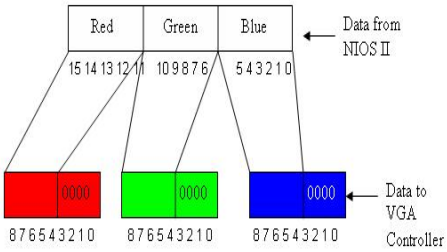

Figure 9: Block Diagram of Imagic

The image is arranged in the SRAM as follows:

SRAM Layout

Address	Data
0x00000	Pixel 1
0x00002	Pixel 2
0x00004	Pixel 3
.	.
.	.
.	.
.	Last Pixel
.	Invalid Data
.	.
0x3FFFB	.
0x3FFFD	.
0x3FFFF	Last Data Work (Junk)

Figure 10: SRAM Layout

There is 512K available in the SRAM. Therefore 256K pixels can be stored in the SRAM. Therefore approx a 546 x 408 size image can fit into SRAM max. The image will be centered and the rest of the screen will be blackened out.

The timing between the SRAM and VGA is of particular interest. Three timing diagrams are important when designing the timing diagram needed for

Imagic: the timing constraints needed to read from the SRAM[2], the timing constraints needed to write to the SRAM[2], and the timing constraints needed to force the slave-write transfer to stall one cycle[3].

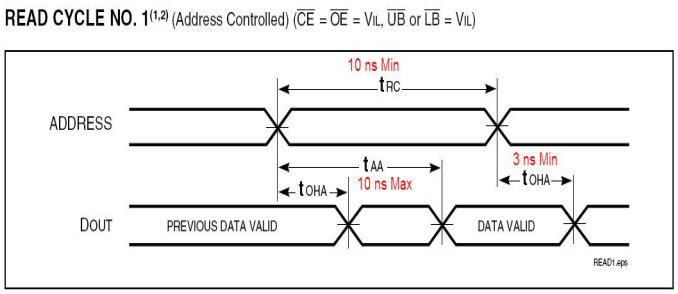

Figure 11: Timing constraints when reading from the SRAM

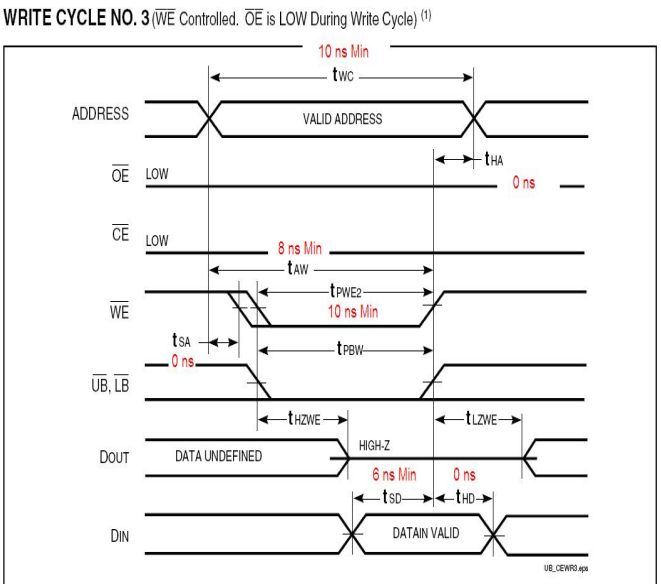

Figure 12: Timing constraints when writing to the SRAM

There are a few signals to consider when discussing the interaction between the VGA peripheral, the Avalon bus, and the SRAM:

- avs_s1_clk : 50 MHz clock
- Clk : 25 MHz clock (split from 50 Mhz clock)
- ADDRESS_TO_SRAM : When writing, what address data should be written. When reading, what address should be output from SRAM.
- RAM_READ_ADDR : What address should be output from SRAM.

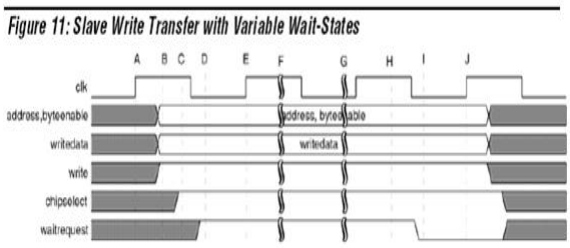

Figure 13: Timing constraints to force the slave write to stall

- RAM_WRITE_ADDR : What address should be written to SRAM.
- WRITEENABLE_TO_SRAM : Indicates when the SRAM should write data to SRAM.
- WRITEDATA_TO_SRAM : Image Data to be written to SRAM
- avs_s1_write = '1' : Signal received through Avalon protocol the master (NIOS II) wants to send the VGA slave data.
- avs_s1_chipselect : Specifies the NIOS II is selecting the VGA peripheral for I/O
- avs_s1_writedata : Data received from NIOS processor that is to be written to SRAM.
- avs_s1_address : Indicates what type of data is being received from NIOS
- avs_s1_waitrequest : Used to stall the Avalon interconnect when the VGA peripheral is not ready to retrieve and right data to the SRAM.
- READDATA_FROM_SRAM : Pixel Point read from SRAM
- IMAGE_RED : Internal SIGNAL used to store RED component of VGA signal
- IMAGE_GREEN : Internal SIGNAL used to store GREEN component of VGA signal
- IMAGE_BLUE : Internal SIGNAL used to store BLUE component of VGA signal

Using these guidelines and variables, basic components of the algorithm can be constructed.

The first thing that will be described is how the address that specifies what data should be read from

the SRAM is controlled. A register (VHDL SIGNAL) is created named RAM_READ_ADDR. This signal is 18 bits and initialized to zero. This variable is incremented every time a pixel from the image is read from SRAM. So every time the VGA peripheral displays a pixel in the rectangle defined by the width and height of the image, the RAM_READ_ADDR variable will be incremented. This ensures that the VGA peripheral reads a new value from the SRAM every 25 MHz. Once the VGA controller reaches the bottom right pixel (the end of the screen), the RAM_READ_ADDR variable is reset to zero, so that the start of the image can be read. This works because the image is stored continuously in SRAM.

Next the way the SRAM is read/written is described,

```
if avs_s1_clkevent and avs_s1_clk = 1
   if clk = 1   -- Read Data
        WRITEENABLE_TO_SRAM <= '1';
-- Tell SRAM to output data
        -- NOTE:  This also commits SRAM
                  data that may have been
                  specified during WRITE phase
        ADDRESS_TO_SRAM <= RAM_ADDR;
-- Specify what address to output
   else         -- Write Data
    if avs_s1_chipselect = 1 and avs_s1_write = '1'
     then
        if avs_s1_address = "00000" then
-- Data Type
          ADDRESS_TO_SRAM <= RAM_WRITE_ADDR;
--Tell SRAM where to write
          WRITEDATA_TO_SRAM <= avs_s1_writedata;
-- Write Data
          WRITEENABLE_TO_SRAM <= '0';
-- Tell SRAM to write data
        elsif avs_s1_address = "00001" then
-- Address Type
          RAM_WRITE_ADDR <= avs_s1_writedata;
-- Grab SRAM write ADDR
        elsif avs_s1_address = "00011" then
-- Width Type
           IMAGE_HEND <= avs_s1_writedata;
-- Grab Image Width
        elsif avs_s1_address = "00100" then
-- Height Type
           IMAGE_VEND <= avs_s1_writedata;
-- Grab Image Height
elsif avs_s1_clk'event and avs_s1_clk = '0' then
   if clk = '1' then - Latch data from SRAM
    IMAGE_RED <= READDATA_FROM_SRAM(14 downto 10);
    IMAGE_GREEN <= READDATA_FROM_SRAM(9 downto 5);
    IMAGE_BLUE <= READDATA_FROM_SRAM(4 downto 0);
    avs_s1_waitrequest <= 0;
-- Indicate that we can write data to
           SRAM next cycle
   else  clk = 0
      avs_s1_waitrequest <= 1;
-- Indicate that we cant write data
           to the SRAN next cycle and we need
           to stall.
```

Many aspects of this algorithm need to be analyzed. First it should be noted that avs_s1_clkevent is used to control when this code is evaluated. Also note that both the rising _s1_clk edge and the falling avs_s1_clk edge are important. This is the 50 MHz clock. Next, notice that clk is used to control what action is being performed on the SRAM. If clk is = 1, then the SRAM will be read from. If clk = 0, then the SRAM can be written do (if desired). Clk is the 25 MHz clock. Therefore, there are for distinct states available:

1. Rising edge of avs_s1_clk and Clk = 1
2. Falling edge of avs_s1_clk and Clk = 1
3. Rising edge of avs_s1_clk and Clk = 0
4. Falling edge of avs_s1_clk and Clk = 0

Now for a more detailed analysis of the read cycle (when clk = 1).

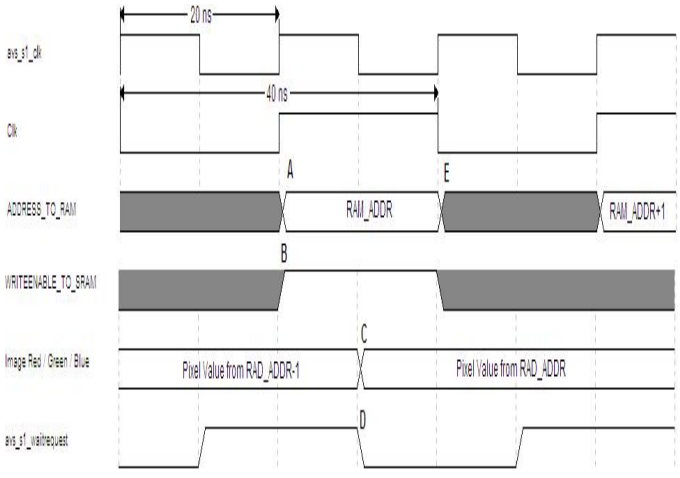

Figure 14: Read Cycle

Notes on Diagram:

- A : Output to the SRAM the address of the pixel point is to be read from the SRAM.
- B : Raise Write Enable Signal, since this is the read phase. This will cause SRAM to output data.
- C : As specified by the timing diagram from the SRAM datasheet [2], valid data will be output from the SRAM after 10 ns. At this point it has been 10ns since a valid address was supplied. Therefore, the data from the SRAM is latched and fed into the appropriate IMAGE_RED, IMAGE_GREEN, and IMAGE_BLUE registers.
- D : During the next rising avs_s1_clk edge, the VGA peripheral will be able to write the SRAM, so there is no need to raise the wait request signal.
- E : The RAM_ADDR and WRITENABLE_TO_SRAM values may change, since this is now the write phase.

Now for a more detailed analysis of the write cycle (when clk = 0).

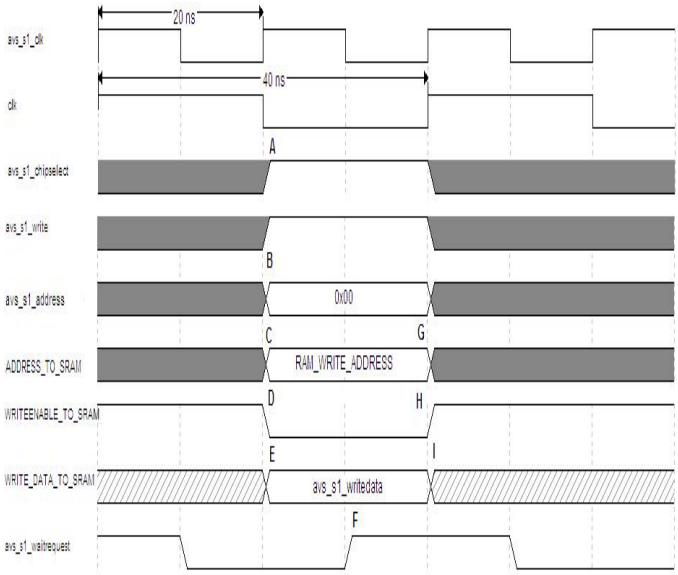

Figure 15: Write Cycle

Notes on Diagram:

- A : If chipselect signal and write signal is high, then the NIOS is writing data to VGA peripheral.
- B : This is used to distinguish what type of data is being sent to VGA peripheral. In this case its 0x00, so therefore its a pixel value to be written to the SRAM
- C : Set SRAM address that the pixel should be written to.
- D : Set WRITEENABLE_TO_SRAM to indicate that data should be written to SRAM.
- E : Write the pixel value to the SRAM. This is the value received over Avalon bus.
- F : During the next rising avs_s1_clk edge, the VGA peripheral will not be able to write the SRAM, so tell the Avalon interface to wait one clock cycle by raising the wait request signal.
- H : As specified by the timing diagram from the SRAM datasheet [2], the address for the SRAM can change immediately after valid data is written to the SRAM
- G : As specified by the timing diagram from the SRAM datasheet [2], the address and data must be valid for 10 ns before the data can be written to SRAM. At point H it has been 10 ns since there were valid address and data. Also note that the start of the read cycle will ALWAYS raise WRITEENABLE_TO_SRAM, so the data is guaranteed to be written to the SRAM.
- I : As specified by the timing diagram from the SRAM datasheet [2], the data for the SRAM can change immediately after valid data is written to the SRAM. This is let float, since the SRAM chip should be able to drive the data bus with its own data during the read phase.

Note that the timing diagram is not produced for the cases where the image width, image height, and SRAM write address are received from NIOS are produced, because these are simplified cases of the timing diagram above. Once the data is received from the Avalon bus, the data is immediately latched. It still occurs only during the write phase.

Some other considerations are the signals that have not been discussed regarding the SRAM chip. These pins are hard coded to the following values:

```
SRAM_UB_N <= '0';
SRAM_LB_N <= '0';
SRAM_CE_N <= '0';
SRAM_OE_N <= '0';
```

This will specify that the SRAM is always enabled, will always output data during the read phase, and every read/write will be a full word.

It has been described how to read and write an image to SRAM. Once the data is available for display, it is a trivial task to display it. Using the video display lab (lab 3) as a template, only minor modifications have to be made to display the image. Namely, instead of a constant color the RBG values will be updated every 25 MHz. Also, instead of a constant height or width for the rectangle, the height and width are now variable. Every other detail (involving syncs, porches, etc.) are exactly the same as lab3 and dont require any modification.

3 Software Design

3.1 Reading the FAT File system

The code to read the file system is an altered and simplified version of the FAT module that is part of FreeDos32.

The first 512 bytes of the SD card make up the BIOS Parameter Block, which contains the volume information, type of FAT (FAT16 is most common for removable media), location of root directory, location of the FAT table, as well as other information. Files on the disk are broken up into chunks of Clusters (2048 bytes), which contain 8 Sectors (512 bytes) each. The File Allocation Table contains the linked list of all Clusters that make up a complete file.

Since we are only interested in reading JPG files on the file system, we only implement the read functionality, and ignore Long File Names (we only read the Short File names in MSDOS 8.3 format). The root directory is read from the file system, and the JPEG files are retrieved. The functions that are used for our purpose are:

- fat_init() initialized the file system. Returns pointer to the file system structure.
- fat_nextfile() opens the next file in the root directory, returns the filename (8.3 format), file size, and pointer to access the file, or -1 when end of directory is reached.
- fat_read() reads the next n bytes from the given file pointer and fills a given buffer with its contents

3.2 JPEG Decoding

A JPEG image consists of a number of 8*8 pixel data block units known as Minimum Coded Unit or the MCU. The unit is converted to its frequency domain using Discrete Cosine Transform. The high frequency components are filtered. The low pass filter is characterized by the Quantization Tables which determines the quality and the compression ratio of the JPEG image. Finally the JPEG decoder is coded using Huffman codes to allow more frequent values to be stored as shorter codes.

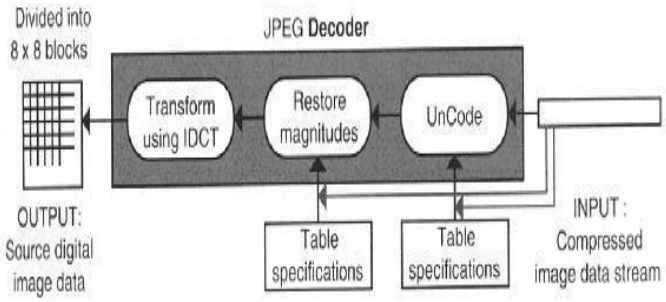

Figure 16: JPEG Decoding

The jpeg decoder is the inverse process. The 8*8 unit of information is retrieved from the encoded data. The coded data is decoded using the Huffmans algorithm using the data provided in the Huffman tables. The frequency components can be extracted using the quantization table. The result is a zig-zag (ZZ) vector which is reordered into an 8x8 block Finally, the inverse discrete cosine transform (IDCT) is applied to the frequency domain to get back the 8*8 MCU blocks.

The JPEG decoder will call the fat_nextfile() function and get a character array as input. It then outputs the image in raster format, and the corresponding RGB values are fed to the vga raster component.

References

[1] Using the SDRAM Memory on Alteras DE2 Board.

[2] 256K x 16 High Speed Asynchronous CMOS Static RAM With 3.3V Supply: Reference Manual

[3] Avalon Memory-Mapped Interface Specification

[4] Wikipedia - File Allocation Table

[5] FreeDOS-32: FAT file system driver project page from SourceForge

[6] J. Jones, JPEG Decoder Design, Sr. Design Document EE175WS00-11, Electrical Engineering Dept., University of California, Riverside, CA, 2000

[7] Jun Li, Interfacing a MultiMediaCard to the LH79520 System-On-Chip

[8] Engineer-to-Engineer Note Interfacing MultiMediaCard with ADSP-2126x SHARC Processors