UNIVERSITY OF TORONTO

ECE532H1S - Digital Systems Design

Group #7: Ultimate Converter - Team Final Report

Authors:

Nikita Gusev

Anthony De Caria

Taylan Gocmen

Instructor:

Prof. Paul Chow

Teaching Assistant:

Daniel Rozkho

Thursday, April 13th, 2017

Table of Contents

1. Overview 1
  1.1. Background and Motivation 1
  1.2. Goals 1
  1.3. Block Diagram 1
  1.4. Brief Description of IP 3
    1.4.1. Reference Design 3
    1.4.2. Processor Code 3
    1.4.3. SD Card SPI 3
    1.4.4. UART Communication 4
    1.4.5. Conversion IP 4
2. Outcome 4
  2.1. Results 4
  2.2. Possible Further Improvements 5
3. Description of the System Components 5
  3.1. Conversion IP 5
  3.2. Processor 6
    3.2.1. uc_utils 7
    3.2.2. uc_image 7
    3.2.3. uc_bmp 8
    3.2.4. uc_png 8
    3.2.5. lodepng 9
  3.3. Internal Communication and External Ports 9
    3.3.1. SD card input 9
    3.3.2. UART to Host PC 10
4. Description of Design Tree 11
Bibliography 12
Appendix 13

1. Overview

1.1. Background and Motivation

Image conversion plays a large part of the current computer experience. We have all needed to convert images between formats, such as making an .ico image for a personal website favicon or to convert an .svg image to .jpeg for insertion into a Google Docs file. We wanted to develop a FPGA project that can perform this function.

While online image converters and image processing applications do exist, they are not practical for small batches of images, since one would need to download and set up a program. Furthermore, while many of these programs are free, they are limited to specific file formats, while others are expensive but support a wide variety of formats. Our project would be a free image conversion system that would encompass a variety of image formats.

1.2. Goals

The initial goal was to achieve image conversion to a few different standard formats as well as do required image processing for filtering and compression. Our project initially was going to take images from an SD card plugged into the Nexys 4 board and convert them into the required format. We also were going to allow the user to select and process all the images into all the given formats and save the images back to the SD card or the host computer.

1.3. Block Diagram

Figure 1.3.1 shows our original block diagram of the Ultimate Converter - reprinted in Figures 1.3.2 and 1.3.3 for clarity.

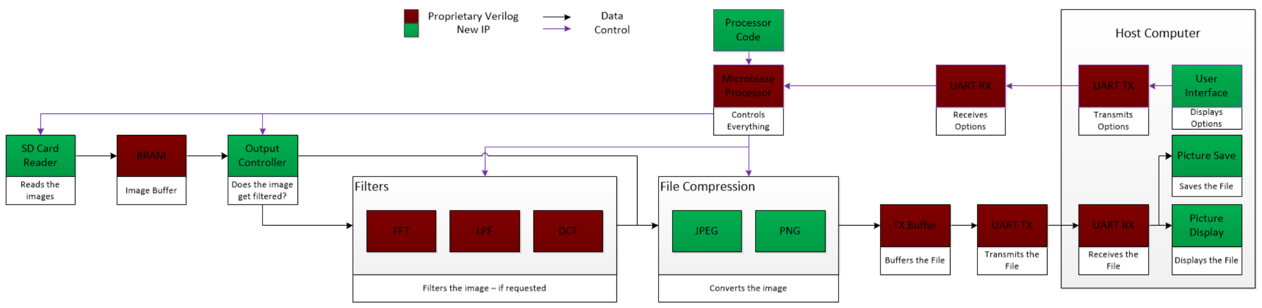

Figure 1.3.1: Our Original Block diagram

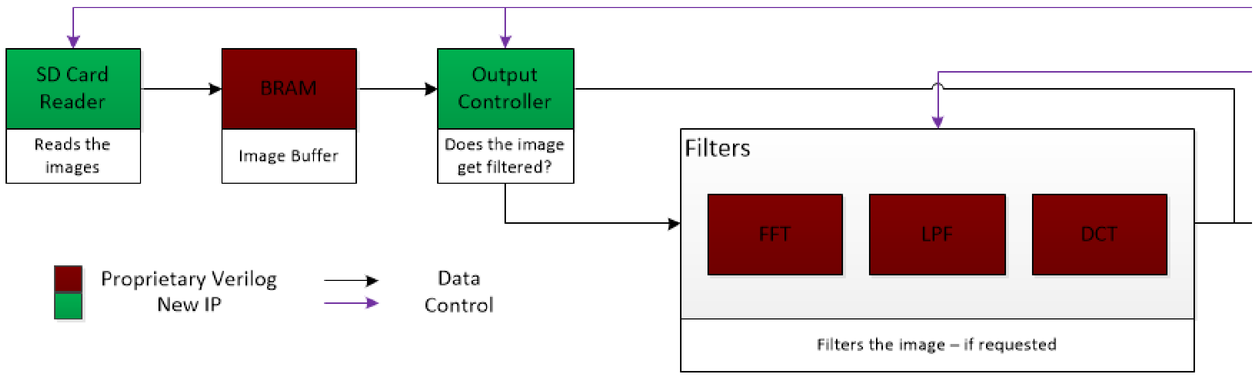

Figure 1.3.2: The left side of Figure 1.3.1 - Covering from the SD Card Reader to the Filters

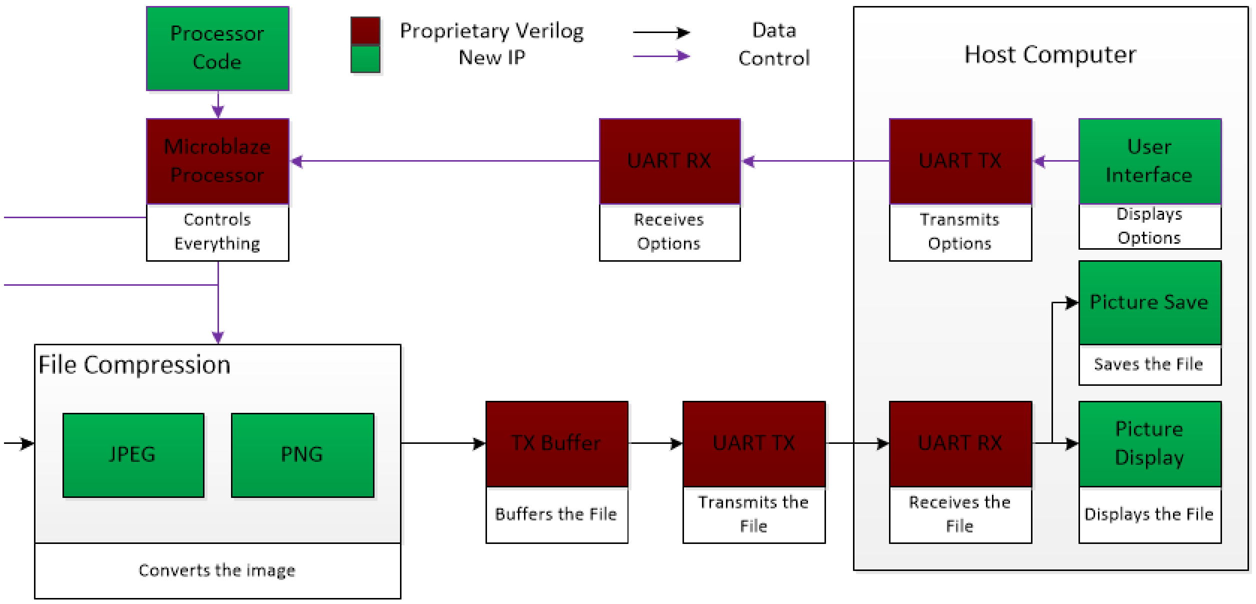

Figure 1.3.3: The right side of Figure 1.3.1 - Covering from the File Compression to the Host Computer.

Our final design, as detailed in our presentation, is shown in Figure 1.3.4. The pink blocks represent larger modules, green ones represent completed modules, the yellow partially completed modules, and the red incomplete modules.

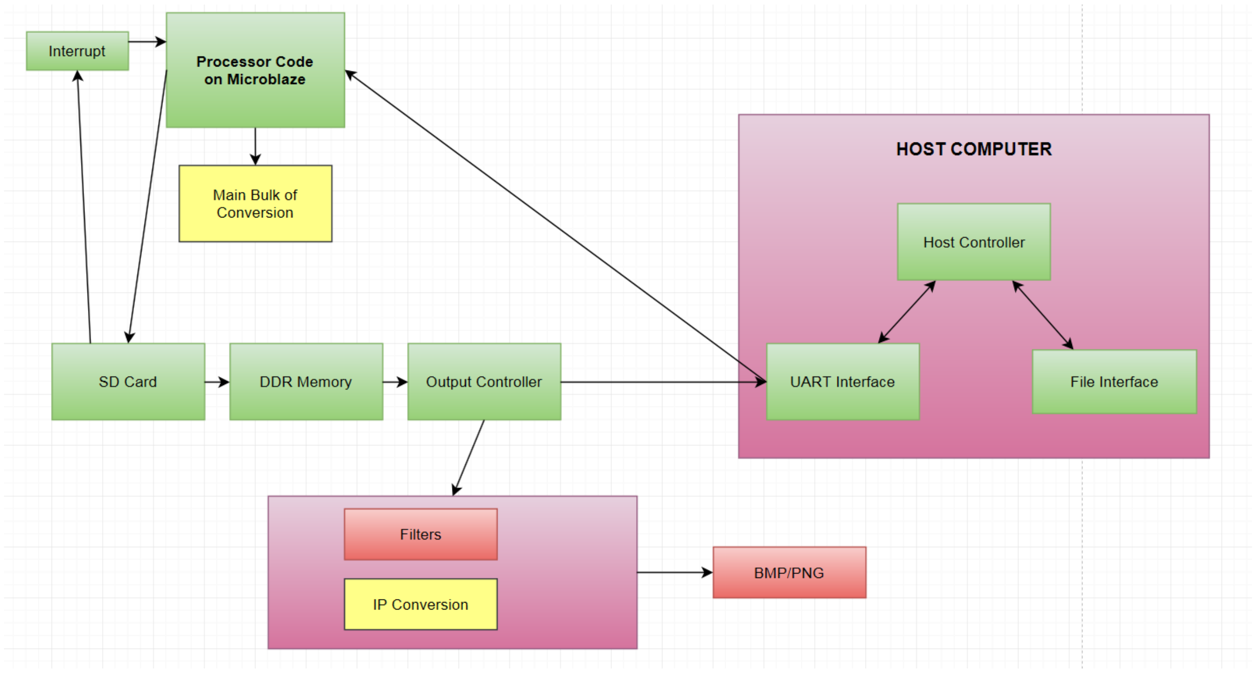

Figure 1.3.4: Our final design

1.4. Brief Description of IP

1.4.1. Reference Design

As a basis for the SD card SPI communication, we used the “SD card sample project” from Paul Chow in Piazza, which implemented all the low level FATfs commands necessary for communicating with the SD card.

1.4.2. Processor Code

The processor code performs the following functions:

- It takes in two pointers:
    - Where the original image is stored in the DDR memory
    - A DDR memory location for the converted image to be inserted
- Takes in two formats:
    - The format of the image
    - The format the image format it will be converted to
- Applies the image conversion
- Stores the converted image in the given location

1.4.3. SD Card SPI

This hardware module performs the following functions:

- Uses the AXI QUAD SPI configured in standard mode and a 10MHz clock to communicate with the SD card

- The sample project used for reference is “SD card sample project” which implemented all the low-level commands
- The low-level modules necessary for communication with the FAT file system on the SD card used the Elm Chan FATfs libraries **ff.c, and diskio.c** as the main source
- High level reading/writing functions are done using the macros provided by the FATfs

1.4.4. UART Communication

This AXI LITE hardware module:

- Was built using information from the UARTLite datasheet
- Collects data from the Host PC and stores it into DDR memory
- Retrieves data from the DDR memory and sends it to the Host PC
- Uses a simple protocol for sending file sizes between Host PC and FPGA
- Uses acknowledgement signals to determine if a piece of data is received - on both the FPGA and Host PC sides
- The Host PC code’s base was found on the Arduino Playground

1.4.5. Conversion IP

This AXI LITE hardware module performs the following functions:

- The processor calls this IP to speed up the conversion process
- Converts 32 bit RGB value representations of a pixel to an 8 bit color table index in a color table
- Converts an 8 bit color table index to a 32 bit RGB value
- Uses an FSM to assert done signals to signify when the calculations are finished
- Allows user to use one IP to do both compression/decompression of color values

2. Outcome

2.1. Results

The project did not meet many of the desired functionality goals, and as a result the converter did not convert any images. However, the peripherals necessary for image transfer, storage and manipulation were completed. The list of desired tasks and their status are given in Table 2.1.1 and Table 2.1.2.

Features	**Status**
User can select file and conversion standard from Host PC	Partial
User can select storage location (SD, DDR, Host PC)	Partial

Convert image from one format to another	Incomplete

Table 2.1.1: Status of Initially proposed features

Functional Requirements	**Status**
Read/Write SD card	Complete
UART communication with Host PC	Complete
Store Images to DDR memory	Complete
Speed up conversion using custom IP	Complete
Convert image and send to Host PC	Incomplete

Table 2.1.2: Status of Initially proposed functional requirements

2.2. Possible Further Improvements

The most pressing improvement should be to increase the number of file format conversions from the currently implemented PNG and BMP. This would give users the improved functionality we originally desired. After that, it would also be critical to increase the transmission size of the Host PC connection to allow larger images to be quickly transported to the Host PC.

To further improve the user experience, the user should be allowed to easily select the final image formats. This could be implemented by introducing a Host PC GUI or by using the FPGA's peripherals. In addition, a Bluetooth module should be integrated to allow users to send send and receive images from a mobile device.

3. Description of the System Components

3.1. Conversion IP

Inputs: slave registers 0 and 4
Status: slave registers 1 and 5
Outputs: slave registers 2 and 6

The IP FSM starts is in idle state and the status registers are 0 and no conversion is taking place. The Microblaze communicates with the IP through a pointer to its base address, and has the option of performing two types of conversions: index to RGB, or conversion RGB to index. They work as follows:

- RGB to index: The Microblaze sets slave register 0 with the RGB as input for conversion. An FSM makes sure that at least 7 cycles have occurred before slave register 1 is asserted symbolizing that the conversion is complete and the output can be read from slave register 2. Once slave register 2 has been read the state returns to idle allowing the write to slave register 0 or 4.
- index to RGB: The Microblaze sets slave register 4 with the Index as input for conversion. An FSM makes sure that at least 7 cycles have occurred before slave register 5 is asserted symbolizing that the conversion is complete and the output can be read from slave register 6. Once slave register 6 has been read the state returns to idle allowing the write to slave register 0 or 4.

These functions are found in the file **uc_bmp.c** as "bmp256_color_table_index_to_color" and "bmp256_color_table_color_to_index".

As a result of the above actions, the IP FSM contains five states: Idle, Performing Conversion (index to RGB), Done Conversion (index to RGB), Performing Conversion (RGB to index), and Done Conversion (RGB to index). After either Done Conversion, the FSM returns to Idle and the process can be repeated. This is process is detailed in Figure 3.1.1.

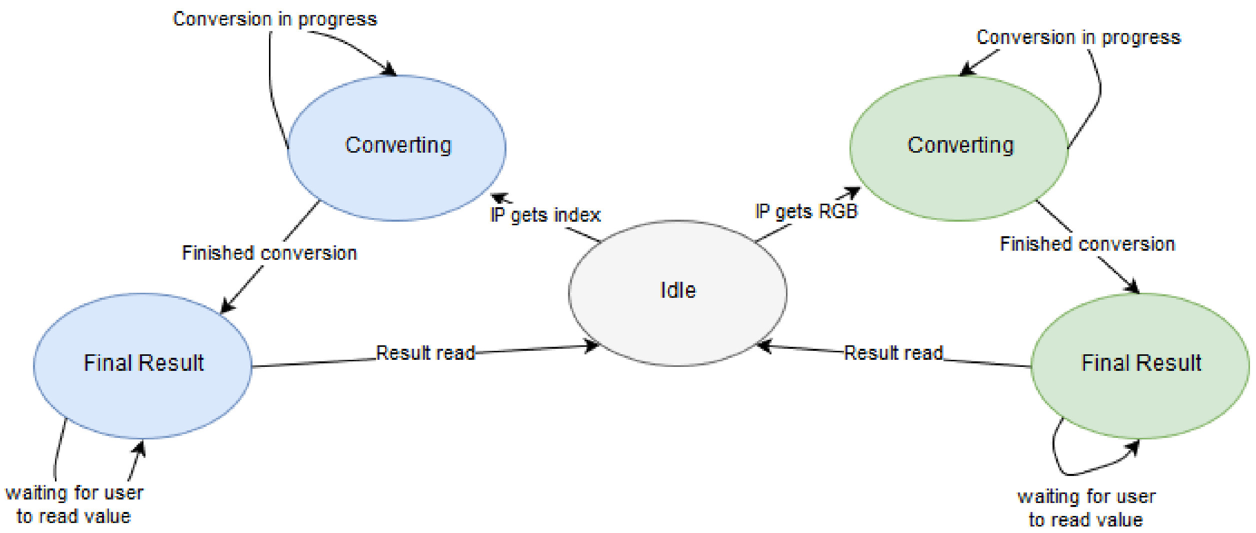

Figure 3.1.1: Conversion IP FSM. Note that "Final Result" refers to the Done Conversion mentioned above.

3.2. Processor

The processor portion of the project converts an existing image in DDR memory and returns it to another location. They are split into five header files according to functionality, to easily compartmentalize the code to different files maximize its utility. They are described in greater detail below.

3.2.1. uc_utils

Uc_utils contains the utility functions, enumerators and macros used across the four other uc files. It contains five functions that are explained below:

- abort_: This function takes a string and other variables causing errors to display and prints the given argument as an error string to stderr. It then aborts the process. This function was invaluable during the debugging process.
- to_power: This function takes two integers - a base and a power - and returns the base to that power. This was implemented to reduce our Microblaze overhead.
- print_binary: This function takes in the number of bytes to print and a starting pointer prints the contents of that array in binary. Like abort_, it was useful while debugging.
- get_bytes: This function takes two integers - the starting and ending indices - and a char buffer and returns the value of the buffer for the given indices with digits accounted for. For example, if 2 chars in the buffer was (EF)(AB) the function would return 61355. This then can be cast to either integer or char. This function is heavily used while reading and writing to addresses, as it is provides access to a buffer's contents.
- print_unsigned_bytes: This function takes the value of up to 4 bytes as an integer and prints it in hexadecimal form, usually used to test the content read with get_bytes. It was another helpful debugger tool.

3.2.2. uc_image

Uc_image contains the C level pseudo-class of the C++ UC_IMAGE methods. It contains the following structures and functions:

- UC_IMAGE: This structure is what the image file data gets converted into for processing and is the most important structure for the processor code. It contains:
    - pxArr: a 3D char array that contains the raw pixel array
    - pxHeight: the height of the image, also the first dimension of pxArr
    - pxWidth: the widht of the image, also the second dimension of pxArr
    - pxFormat: the number of components for each pixel, also the third dimension of pxArr
    - fBuffer: the pointer to the image file data
    - fSize: the size of the image file data
    - fFormat: the format of the image file data
- open_uc_image: This function takes in the the pointer to, size of and format of the image file data, creates a new UC_IMAGE and calls the corresponding image read function to convert the image file data to the UC_IMAGE format. It returns the newly created UC_IMAGE.

- write_uc_image: This function takes a UC_IMAGE pointer of the image to be converted to, the pointer to the address to write the converted image file to write to and the format to convert the UC_IMAGE to and calls the corresponding write function.
- close_uc_image: This function frees all the data structure created for the given UC_IMAGE.
- convert_image: This function is a easy to use wrapper that bundles the rest of the uc_image file functions, open_uc_image, write_uc_image and close_uc_image. It takes in the pointer, the file size for and the format of the image file data to be read, and the pointer and the format of the image data to be written and calls the aforementioned functions in the correct order.

3.2.3. uc_bmp

Uc_bmp contains the four functions needed to manipulate a BMP image as well as the functions needed to access the Custom IP:

- read_bmp: This function takes a pointer to the UC_IMAGE and reads the data of size fSize from the pointer location of the fBuffer. It decodes the BMP's headers, color table and the pixel array before writing the storing that data in the appropriate UC_IMAGE fields.
- write_bmp: This function takes a pointer to the UC_IMAGE and a pointer to the DDR Memory. It takes the UC_IMAGE's raw pixel array and converts it to the BMP pixel array format, encodes the correct headers and the color table for the BMP file format and then writes the encoded data to the given pointer.
- bmp256_color_table_index_to_color: This function writes the index to the first input register of the custom IP for the first function implemented in the IP, waits for the IP to finish and then reads the color from the first output register of the IP. It is used to
- bmp256_color_table_color_to_index: This function writes the three colour components concatenated to the second input register of the custom IP for the second function implemented in the IP, waits for the IP to finish and then reads the index from the second output register of the IP.

3.2.4. uc_png

Like uc_bmp, uc_png contains the files needed to manipulate a PNG image:

- read_png: This function takes a UC_IMAGE pointer that contains data in its fBuffer. It then reads that fBuffer data using the lodepng_decode function. The lodepng_decode function extracts the width, height, pixelformat and stores them into the UC_IMAGE pointer.
- write_png: This function takes a pointer to the UC_IMAGE and a pointer to a location in DDR memory. It encodes the UC_IMAGE's raw pixel array using the lodepng_encode_memory function and writes the encoded data to the given pointer location in the DDR memory.

3.2.5. lodepng

Lodepng contains a lightweight, standalone libpng implementation of the libpng and zlib libraries[1]. It is developed by Lode Vandevenne for anyone's use.[2] It contains the lodepng_decode and lodepng_encode_memory needed in uc_png, as well as their auxiliary zlib functions inflate and deflate. These algorithms decompress and compress the IDAT, or image pixel data, of the PNG image file format.

3.3. Internal Communication and External Ports

3.3.1. SD card input

The low level FPGA to peripheral SD card communication uses the AXI QUAD SPI protocol for communication. An external interrupt pin is connected to the designated "sd_inserted" IO pin to check if the SD card is inserted or has been removed mid run.

The Microblaze used Elm Chan's FATfs to do the high-level for mounting the SD card and performing block read and write commands. These functions providing very useful macros are in **ff.c** and **diskio.c**.

Before any SPI communication can take place, the Microblaze checks if the card is inserted. Then, the card is initialized using the sequence described in Figure 3.3.1.1.

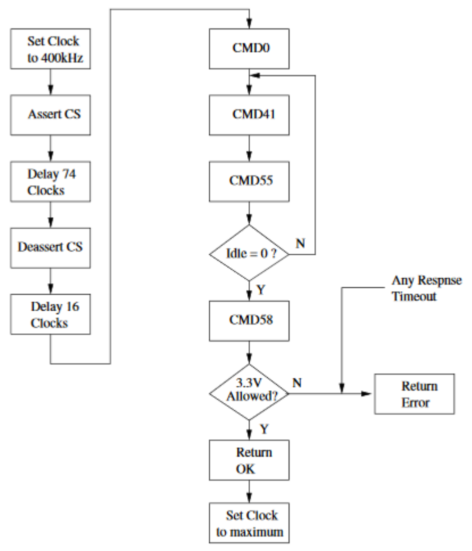

Figure 3.3.1.1: FSM for initializing the SD card [3]

If the SD card is properly initialized and all the commands return successful, then the SD card interaction can begin.

F_open is used to open a file with the specified name and give back its size in bytes. This is used when reading the image data. The F_write function is used to write image data to the SD card by providing a filename and size. The “read_files” function, written by the team, loops through each directory and folder of the SD card and outputs their names. This is done to give the user the option of selecting their desired folder through the UART interface.

Once a file is selected, its contents are moved from the SD card’s stored into DDR memory. A pointer is then returned so the rest of the Microblaze code can modify and convert this image independent of the SD card.

3.3.2. UART to Host PC

The Verilog portion of the FPGA-Host PC connection is basic; it simply involved invoking the UARTLite IP. As a result, the team decided to create its own Microblaze software suite to increase the connection’s complexity.

This software suite is split into two levels - the lower axi_uartlite.h code and the higher ECE532.h code. Axi_uartlite.h handles the communication with the UARTLite, doing so by writing to and reading from the UARTLite’s four slv registers. If the status register shows that valid data exists in the RX FIFO, or that data can be added to the TX FIFO, data is handled accordingly, using a buffer provided by ECE532.h. The size of the buffer is sent by ECE532.h as well, and is assumed to be within the UARTLite size limit of 16 bytes.

While the UARTLite does support interrupts, we decided to poll the RX and TX data valid bits due to time constraints.

The ECE532.h code handles the higher protocols; these being the Packet Acknowledgement, the End of File, and the File Size. The Packet Acknowledgement protocol is simple - once either side sends a 16 byte packet, it then immediately waits to receive an ACK character. Once it has received that character, it will then move on to the next packet, reducing the possibility of file corruption.

For the End of File protocol, once the Host PC is finished sending the file, it sends a small character string to the FPGA. This would allow the FPGA to switch from receiving the picture file to converting that file. Originally this string was simply the EOF character; however, this became problematic when sending picture data as “EOF characters” represented pixel colour values. As a result, we changed to a larger string structure.

The File Size system protocol is the same for both sides - a packet would be sent starting with the characters “FSIZE=” (excluding the quotations). The rest of the packet until the NULL character would then contain the decimal representation of that size in characters. For example, if the Host PC was sending a 160 byte file, it would send “FSIZE=160” to the FPGA. This required us to create our own itoa function on the FPGA, in order to reduce our Microblaze overhead. While this protocol does limit us to a 10 digit file size, we felt that was acceptable, especially with the 16 byte packet limitation reducing the file size more drastically.

The Host PC code handles the reading and saving of files. It uses code from the Arduino Playground to create a serial connection with the FPGA. Once a connection exists, it will store the file into a perfectly sized char array. It will then read 16 bytes of the char array and send it to the FPGA - following the Acknowledgement and File Size protocols listed above. Once the Host PC has received the new file from the FPGA and stored it into a new char array, it will then use this string to save a new file on the Host PC. In its current form, you have to manually change the file strings in the fread and fwrite functions in order to change which file, and what type of file, is being read and saved. The serial connection code uses functions from the Windows libraries; as a result the Host PC code will only work on a Windows machine.

4. Description of Design Tree

Our repository design tree is as follows:

- doc - Contains our Group Report and Presentation Slides
- hostPC - Code that was running on the Host PC side
- legacy - Old code that is no longer in use. This includes old main files and Verilog IPs attempted.
    - SPIforSD - Our original attempt to make a SD Card connection
    - TestingModules - Contains old main files that tested our separate components before integration
        - CustomIP
        - Conversion
        - SDCard
        - UART
- src - Our Source code
    - C - All the C Code. testparams.c contains the main function for the system that was running at the Final Demo.
        - Conversion - The C Code that can convert images
        - SD Card - The C Code that will access the SD Card
        - UART - The C code that uses the UART
    - Verilog - All the Verilog code
        - IP - The Custom IP we created. The folders underneath this are the generic folders created when an IP is generated.

- Project - The Project. It contains the main project file, and the srcs needed to build the project.

Bibliography

[1] Lode Vandevenne, 'LodePNG for C (ISO C90) and C++', 2016, [Online]. Available: http://lodev.org/lodepng/

[2] Lode Vandevenne, 'LodePNG', 2016, [Online]. Available: https://github.com/lvandeve/lodepng

[3] F. Foust, 'Application Note Secure Digital Card Interface for the MSP430', 2004. [Online]. Available: http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf.

Appendix

Contributions:

M = main contributor, E = editor

Section	Nikita	Anthony	Taylan
1.1		E	M
1.2			M
1.3		M	
1.4.1	M		
1.4.2		E	M
1.4.3	M		
1.4.4		M	
1.4.5	M		
2.1	M	E	E
2.2	E		M
3.1	M	E	
3.2		E	M
3.3.1	M		
3.3.2		M	
4.0		M	
Whole Document	E	E	E