Project Number: SJB-VHDL

VHDL Implementation of the JPEG-LS Algorithm

A Major Qualifying Project Report:

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Joshua Raines
godsguy@wpi.edu _____

William Tolli
willyman@wpi.edu _____

Joseph Holmes
Jack7890@wpi.edu _____

Date: October 29th, 2004

Thomas Flatley
Tom.flatley@gsfc.nasa.gov

Professor Fred J. Looft
fjlooft@ece.wpi.edu _____

Moses McCall
Moses.mccall@gsfc.nasa.gov

Professor Stephen J. Bitar
sjbitar@ece.wpi.edu _____

Phyllis Hestnes
Phyllis.Hestness@gsfc.nasa.gov

Executive Summary

The purpose of this project was to implement the JPEG-LS image compression algorithm in hardware using VHDL in order to transfer images from NASA satellites and autonomous vehicles more efficiently. To date, these images have been transmitted back to Earth in their entirety, without any form of compression. Due to large file size, transmission times continue to be relatively large compared to the time it takes to acquire an image.

File compression is an important concern for files that need to be sent over any form of network or placed in storage. Images fall into this category, however restoring an image to its original quality can present a problem. Images used in situations such as biomedical research or astrophysical categorization must retain all the quality of the original image because machine analysis can be performed on these images to find very specific details that the human eye can not. If these images are to be transmitted or stored, some form of lossless image compression is needed.

The National Aeronautics and Space Administration (NASA), has need for such an algorithm when taking, sending, and storing pictures. Satellites such as the Landsat series take pictures of the earth which are then transmitted to a ground station where computer analysis is performed. In the case of the Landsat satellite, the Automatic Cloud Cover Algorithm (ACCA) is run to determine if a picture is too obscured by clouds to be of use. The transmission time of these pictures is exceedingly long due to the large size of each file, varying in size from 35 to over 130 megabytes. If these images could be compressed on board the satellite, transmitted, then decompressed on earth, the transmission times could be cut, thus reducing power consumption of the satellite, and preventing a backlog of images that need to be sent.

The JPEG-LS algorithm, created in the 1990's, fills this need. Designed by Hewlett Packard for a competition put on by the Joint Photographic Experts Group (JPEG), it is a simple algorithm based on prediction modeling, which produces a file that can be decompressed later to its original quality.

The objective of this project was to design, construct, and test the JPEG-LS algorithm in VHDL to be implemented on a Xilinx Virtex 2000 Field Programmable Gate Array (FPGA). By implementing the JPEG-LS algorithm on an FPGA, this can reduce processor load and theoretically speed up the compression time versus a standard software design. This design also provides the flexibility to adapt this compression for many different applications.

In order to reach this goal we needed to study Golomb coding, prediction and context modeling, and statistical analysis procedures to understand how the software algorithm

processes the data and subsequently compresses it. After this, we were required to show that the Annapolis Microsystems Wildstar Firebird board that had been given to us was operational. This was accomplished by following a tutorial to design a program to increment a counter and flash three LED's that are on the board. After this, a Direct Memory Access (DMA) program was created to test communication between the host system and the onboard memory associated with the Wildstar card. Once these programs successfully ran, the task of getting the JPEG-LS algorithm to work on the Wildstar board became feasible.

While researching the JPEG-LS algorithm on the World Wide Web, we found a graduate student from RIT, Michael Piorun, had already completed a working JPEG-LS compression algorithm in VHDL for simulation as his thesis project. We were able to obtain his thesis documentation as well as all of his working code. As this code had been written in 1999, and the synthesis tools for VHDL have changed in the five years since the completion of his project, many libraries and specific lines of code needed to be changed and updated for the current compilers.

Once the VHDL had been updated to current standards, a "makefile", provided by Annapolis Microsystems, converted this project into the binary file that is needed to program the FPGA. It is at this point that much of the optimization is done to the design, components that are not used are removed by the "makefile", and the program is condensed down to a functional minimum.

Our accomplishments during this project resulted in the determination that the Wildstar FPGA Firebird card is functional. This was done with the help of two sample programs. One determined that the main functions of the card were accessible (Active Matrix display and LEDS). The second program tested that the DMA functions of the card worked properly. The code was then complied and run through the "makefile" where we feel that important components were optimized out. This was due to time constraints that prevented us from being able to implement modules such as the memory interface and register control. We also feel that there is a problem with the numerical system that Michael Piorun used due to it being non standard VHDL syntax.

At this point we had to end our project due to time constraints. At the end of the project we focused on making sure next year's group would not have to work against the obstacles that slowed our progress throughout the term. Our final product is a good start on the complete implementation and a fully functional system with set up instructions for next year's project group.

Abstract

The need to compress a digital image is a growing concern due to the ever-increasing resolution at which these pictures are recorded. This fact, combined with pixel depth, creates images that require more storage space. NASA has many projects that currently take images at high quality of the earth or various objects in space. These high-resolution images take up a large amount of storage space when stored in bmp, TIFF, or any uncompressed format. A newer format, JPEG-LS, has an improved compression ratio, improved encoding and decoding time, as well as retaining all of the original quality of the uncompressed image. The goal of this project is to implement the JPEG-LS algorithm through a hardware description language (HDL) in response to NASA's need to compress images.

Table of Contents

Table of Figures

Table of Tables

Table of Equations

1 Introduction

The Joint Photographic Experts Group (JPEG) was formed in 1986 to define standards for image compression algorithms. The well-known JPG file extension has become the international standard for Internet image compression. By compression of the image, the overall file size decreases. To the casual viewer however, or even through a more detailed inspection of the image, there may not appear to be any loss of information from the original picture.

There are many applications where a picture is taken at one place then sent somewhere else to a base station for further processing or analyzing. NASA uses this concept on a daily basis when receiving images from satellites, such as Landsat 7, or the GOES satellites that are used to catalogue the Earth's surface. Images are taken aboard the satellite and then transmitted down to earth for further processing. The typical personal computer takes up significant amount of power for any compression algorithm. Therefore any method of software compression would not be feasible on a satellite or any autonomous vehicle where power consumption must be kept to an absolute minimum.

In addition, transmission of a full size, uncompressed image from a satellite takes up quite a significant amount of bandwidth due to the picture's size and complexity. It also takes a significant amount of time to transmit the image down to earth for processing. If the JPEG conversion could take place on board the satellite the transmitted information could be reduced by a factor of between 1.5 and 2.

In order to compress an image on-board a satellite, NASA has requested that the JPEG-LS algorithm be implemented in hardware. To this end, our goal was to realize the JPEG-LS algorithm to optimal efficiency through a hardware description language and create a working prototype for use at the Goddard Space Flight Center.

1.1 Summary

The transmission time and storage space required for images causes a need for compression onboard satellites or autonomous vehicles. The decision was made to use something from the JPEG committee standards because the JPEG committee is one of the leading members in the field of compression. JPEG-LS is the compression algorithm of choice because machine analysis is required on certain images and JPEG-LS is a lossless format. JPEG-LS has the advantage of not being complicated as well as having one of the best compression ratios on average. Machine analysis helps to determine certain features, such as landmass and cloud analysis, that the human eye cannot perceive. If data were lost, such as with JPEG, the machine analysis would be inaccurate and therefore useless.

2 Background

In order to thoroughly understand the principles, concepts, and need for utilization, background information had to be gathered for further analysis. The background research in this section includes information about NASA's need for image compression, the JPEG algorithm and subsequent development, and the Annapolis Micro Systems "FIREBIRD for PCI" FPGA board.

2.1 NASA and GSFC

2.1.1 NASA

The National Aeronautics and Space Administration (NASA) was formed in 1958 with the explicit intent of developing space exploration. Over the past 45 years NASA has distributed its research load over several different facilities worldwide. These facilities specialize in all forms of engineering with the dual goals to benefit everyday human life and to further the advance of the space program.

NASA began with absorbing all the government agencies, such as the National Advisory Committee for Aeronautics (NACA), which dealt with anything related to space. With the launch of Project Mercury, to see if humans could survive in the harsh confines of space, NASA was established in the public eye. In addition to human travel in space, NASA has launched numerous orbital and exploratory devices with the intent of documenting things both terrestrial and celestial. (NASA – About NASA)

Well-known spacecrafts such as Pioneer and Voyager have explored everything from the moon, to planets and have left the solar system. Others, such as the mars pathfinder have landed on planets in order to scientifically document the conditions. NASA has also launched orbital satellites that take images of both the earth, and places far out of reach of ground based telescope ability.

The importance of satellite observation cannot be overstated. Information about our earth, its weather patterns, our solar system, and beyond is invaluable to scientists around the world.

2.1.2 GSFC

The Goddard Space Flight Center (GSFC) is dedicated to the expansion of our knowledge of the Earth and its environment, as well as the solar system and the universe. Much of this knowledge is gained through observations made from orbit. Managing all of this data is such a daunting task that Goddard has declared it part of its mission statement;

"Develop and maintain advanced information systems for the display, analysis, archiving and distribution of space and Earth science data." (GSFC Mission Statement)

The Maryland facility also conducts space science studies on stars, galaxies, black holes, and dark matter. The scientists at Goddard also study the Earth's atmospheric conditions, such as the ozone layer, greenhouse warming, and oceanic properties. The information gained from satellites is instrumental to the development of sound scientific conclusions about our universe.

2.2 Image Formats

2.2.1 Introduction

Many satellites generate pictorial data that can be interpreted by humans and computers. When a satellite generates data it needs to be transmitted to the ground before any analysis can be performed. Ideally, only data that is deemed useful would have to be transmitted, or if the data could be compressed it would reduce processor time and power consumption.

When an image is first captured, be it in a digital camera, in a paint program or a scanned image, it normally consists of plain, raw data bytes that represent all of the color data to construct the image; generally there is no other information attached until the file is saved. When the file is saved onto some form of medium (hard disk, NVRAM, floppy disk), it can be saved in one of two formats, uncompressed or compressed. Currently the most popular compressed file format to save image data in is JPEG, which is discussed in section 2.4. There are also several uncompressed formats, of which the most prevalent forms are RAW, BMP and TIF.

2.2.2 RAW

The RAW file format is a very simple, very basic format where there is no extra information attached to the file. This format consists of only the pixel and image size information, normally in a 24-bit RGB format. However 8, 16, and 32 bit formats also exist. The RGB format consists of three values, each ranging from 0 and 255 in decimal representing gradually increasing color intensities. The RAW format contains no compression and no other information about the image such as size, bit-depth, resolution, and a host of other image parameters. This is how most images begin, but is not a very useful format to save images in because many image readers need at least the bit-depth and color mode to be able to display the image correctly.

2.2.3 BMP

The bitmap, also known as BMP, is another uncompressed image format. It is a very simple way to store the information needed to open and read the file. This format is often used on the Internet to store small images for quick and easy loading by the Internet browser, as it involves no decompression time. Bitmap files can be stored in generally the same format as RAW files, using a 24-bit RGB format, however there are also 8, 16, and occasionally 32 bit formats, although the later is rarely used. Bitmap files differ from RAW by including tags in the header of the file, storing information about the image such as bit-depth, size, even what program created the file.

2.2.4 TIF

The TIF format is also a very common way to store uncompressed or slightly compressed image data, however this format has some advantages over RAW and BMP. TIF stands for Tagged Information File (The TIFF Image File Format) because within the image data other information about the image can also be stored for record keeping. Information, such as time and date of image capture, compression used, image width and height, and a host of other data can be stored as tags inside the file. NASA is currently using this format to store their satellite image, so this is the format that will be discussed in more depth below.

2.3 TIF File Format

2.3.1 Introduction

Each TIF image has been designed in a specific format to allow programmers to develop TIF readers on any computer operating system. As a result the TIF format can be considered a “cross-platform” file format. It has also been designed with the purpose of being adaptable to future needs, as well as use for proprietary reasons. The TIFF 6.0 specification details the entire format of the file as well as how to manipulate the tags within the file.

2.3.2 Image File Header

Figure 1 shows an overall view of the TIF format, from the specification sheet, as well as how the data is arranged. Every TIF file starts off with an Image File Header (IFH). This IFH tells the TIF reader what byte order the file is in, and where the first Image File Directory (IFD) is located.

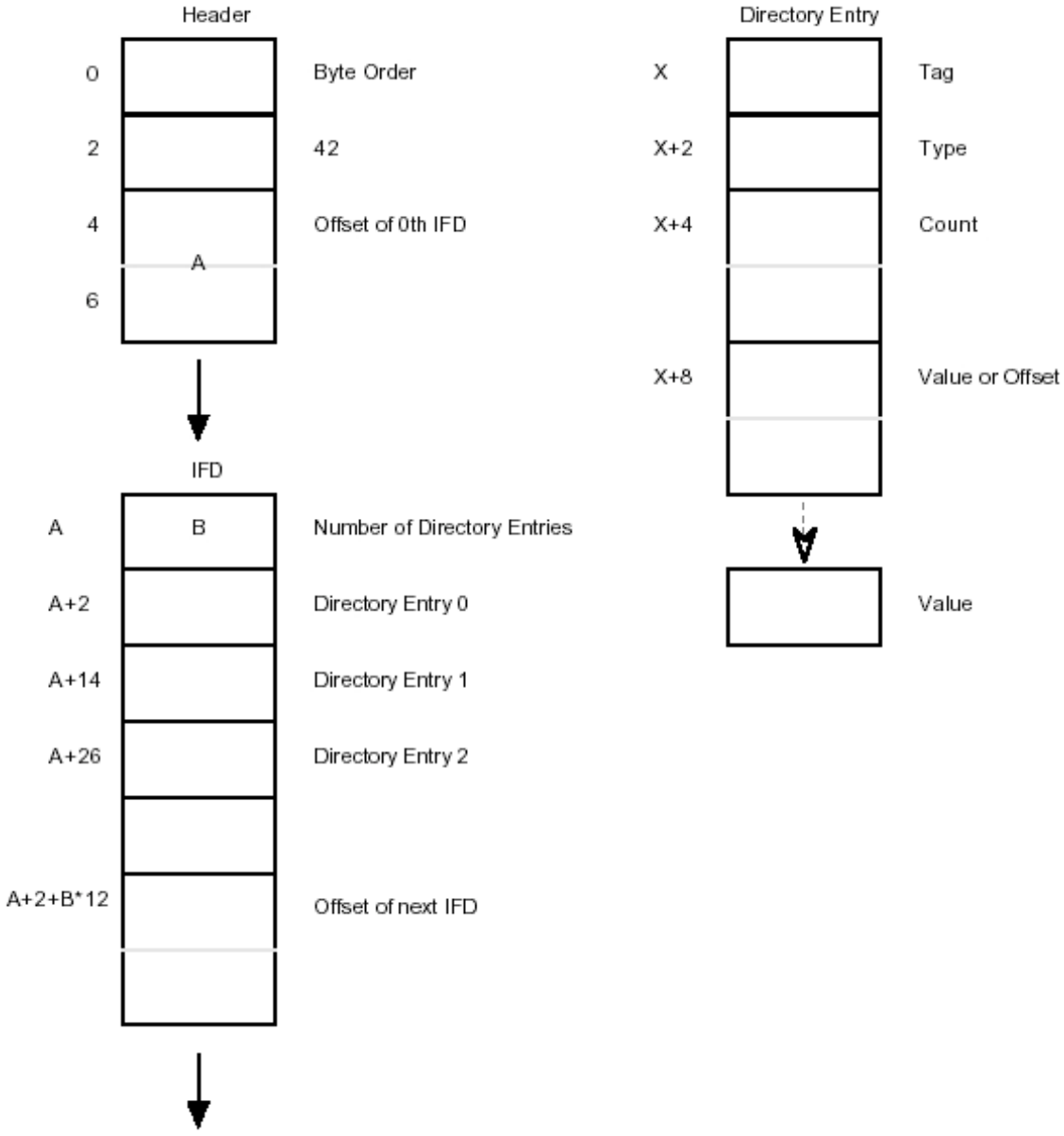

Figure 1: TIF file header layout (TIF 6.0 Specification, 13)

The first two bytes of the IFH determine what the byte order is, either Big-Endian or Little-Endian. For the purposes of this report, all future values used in this section are shown in hexadecimal format. If the first two bytes of the file are 4949, then the file has been written in Little-Endian, meaning the least significant byte is located in the lowest byte space. If the first two bytes are 4D4D then the file has been written using the Big-Endian format, with the most significant byte being placed in the lowest byte space and then progressing to the least significant byte in the highest byte space. (TIF 6.0 Specification, 13) An example of Little-Endian versus Big-Endian is given in Figure 2.

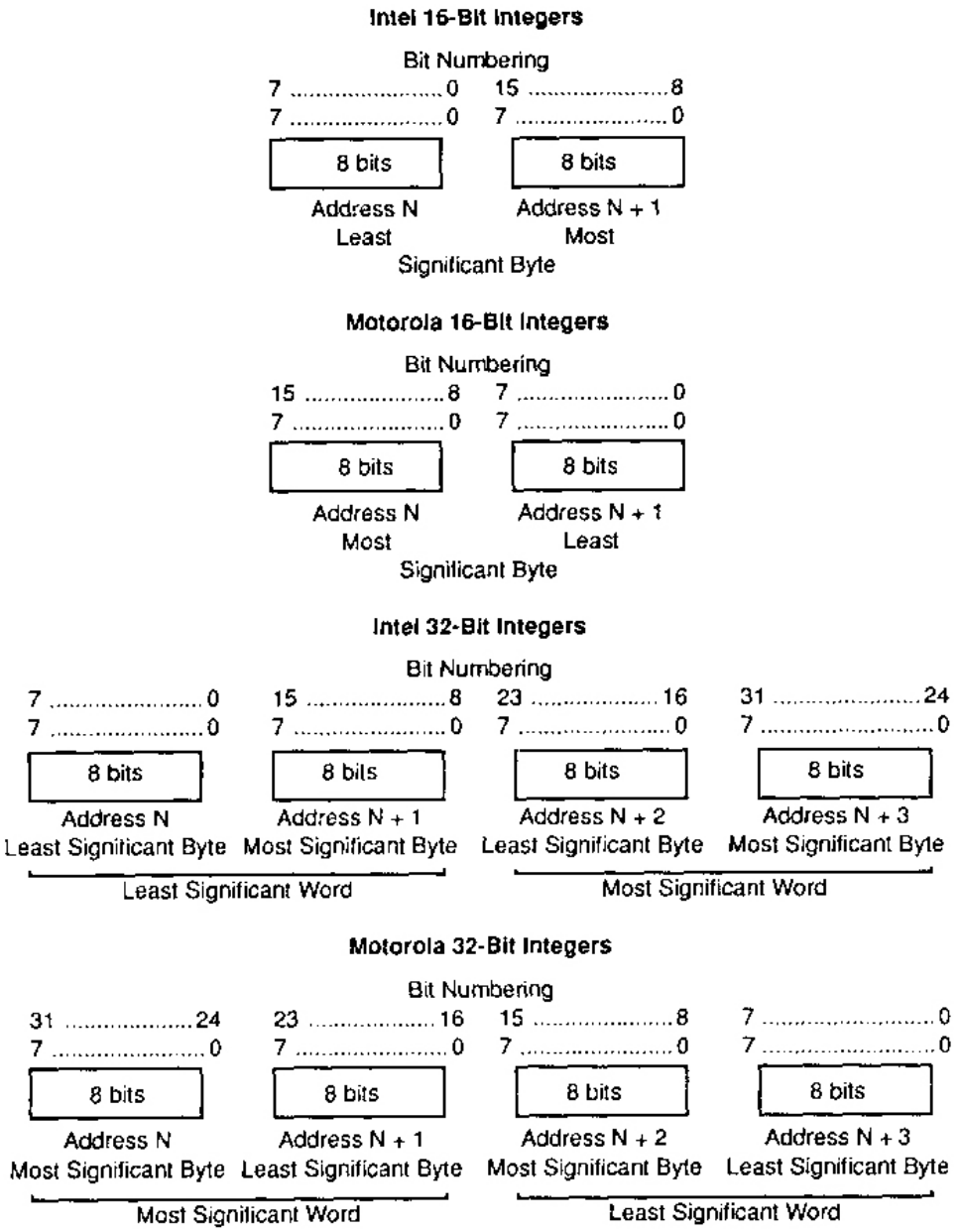

Figure 2: Little-Endian and Big-Endian layout (The TIFF Image File Format)

The next two bytes consist of a pre-assigned number, 2A, which designates this file as being a TIF file. Following that are four bytes that designate the offset in bytes of the first IFD. This may follow right after the IFH, or it may be located somewhere else in the file. After examining several NASA images taken directly from satellites, it is noted that the IFD has been placed at the end of the file, so the offset value is important if the tagged information is to be preserved. Each TIF file must have at least one IFD, and each IFD must have at least one Directory Entry (DE). The offset value is used with respect to the

beginning of the TIF file, meaning the first byte of the file has an offset of zero. (TIF 6.0 Specification, 13)

2.3.3 Image File Directory

The IFD can, and often does consist of many Directory Entries (DE), some of which may be standard entries, while others may contain proprietary information. The first two bytes contain the number of DE that will follow. Each DE consists of 12 bytes, arranged in a particular order that contain information that is pertinent to the image in question. Once the number of Directory Entries has finished, four bytes have been reserved at the end of the IFD to signal where the offset of the next IFD begins. If there is no other IFD, those four bytes are filled with zeros. (TIF 6.0 Specification, 14)

2.3.4 Directory Entries

Each directory entry consists of 12 bytes that contain a tag, a field type, the number of entries in that field, and either the value or an offset to the value for this entry. The first two bytes consist of a tag that identifies what the final value is representing. (TIF 6.0 Specification, 13) Some of these tags are standard, and have been defined, however a large number have been left open for companies, individuals, or organizations to places their own personal tags containing any form of information they need. Because there can be many tags, or Directory Entries for each IFD, they are sorted numerically by tag number.

The next two bytes indicate the type of data located in this entry. A few types of data that can be stored in the file are ASCII characters, 8, 16, or 32 bit unsigned integers, and even rational numbers. (The TIFF Image File Format)

Following this, the next four bytes are allocated to hold the number of values, also called the count, of the type of data. If the type of value is ASCII, and the value stored is HELLO, then the count would be 6, including the NUL character needed for ASCII code. If the type of value is SHORT, which represents a 16-bit word, and the value stored is FF, then the count would be 1. The count stores the number of values, not the number of bytes or bits in the value. (The TIFF Image File Format)

The final field, consisting of four bytes, contains the value or the value's offset. By reading the count and type, it is possible to determine if the data stored in this field will fit within four bytes. If this is possible, then the actual value is stored here. If the data is more then four bytes, then an offset is placed here, and the offset points to the start of the data in question. For instance, if the ASCII value HELLO were being stored, it would take up five bytes for the word and then another byte for the NUL character. These values would be placed somewhere else in the file, and an offset, pointing to the first value would be placed in

the four bytes reserved in the DE. If there is data in this field, it must be placed in the lowest bytes. (The TIFF Image File Format)

2.3.5 Image Data

Once the IFD has been identified, the image can be processed. The raster data that remains is cut into scan lines called strips. By doing this, the entire image does not have to be in memory at one time; only a single strip has to be buffered, which reduces massive amounts of memory when dealing with large, uncompressed files. The TIFF 6.0 specification suggests that strips be limited to 8,192 bytes, however the code in the TIFF function library has the ability to handle strips with lengths approaching 60,000 bytes. (The TIFF Image File Format)

2.4 JPEG

In the growing digital world, technologies emerge frequently that give us the ability to compress files at higher and higher ratios which in turn leaves more storage space for files and programs. Digital images are stored in several different formats, or compression styles. The best file format after uncompressed format such as TIF is that of the "lossy" file formats. One of these "lossy" file formats, and one of the most commonly used today is the Joint Photographic Expert Group (JPEG) (JPEG FAQ sec 1).

2.4.1 "Lossy"

"Lossy" refers to the fact that the encoder "throws out" unneeded pixels. This is done by comparing a pixel to the surrounding pixels and then seeing if any are similar. This format is based on the premise that within the picture, there will be pixels that are not needed and can be discarded. This means that if any given pixel is like another, then one of the two will be kept due to the fact that the human eye cannot notice minor color changes. "Lossy" file formats are meant to be viewed by humans. If machine analysis is done on a file that has been compressed using a "lossy" format then errors can occur because data is actually being thrown away, and the computer can distinguish between the slightest color variations (JPEG FAQ sec 2).

2.4.2 JPEG and "Lossy"

Although JPEG has its limitations due to it being a "lossy" file format, it has one advantage over others of its kind; the user has the ability to define the compression ratio of any given picture, in other words the user has the ability to change how much of the quality is lost and how much storage space is saved over the original image. If the user wants to

retain almost all of the pixels, then they must sacrifice some storage space that could be saved if a greater compression ratio were to be utilized. This ability to define the compression ratio is important (JPEG FAQ sec 2). As can be seen in Figure 3, which shows a JPEG image compared to a TIFF, JPEG can save a large amount of space, starting around 59%, even when the JPEG is compressed at the highest possible quality level. These images come to us from the SeaWiFS project at Goddard Space Flight center.

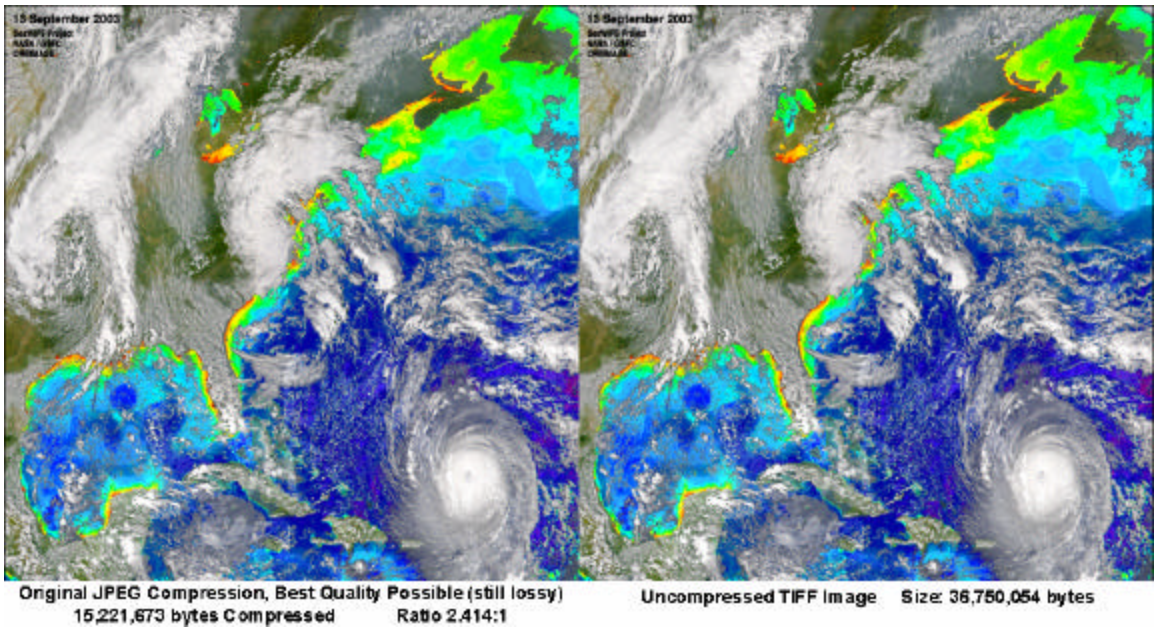

Figure 3: Comparison of storage size between JPEG and TIF

This image was taken at a resolution of 3500 X 3500. The term "resolution" stands for pixels by pixels, or rather how many pixels fill the picture (width by length). When the two images are subtracted from each other, what remains is the difference between the compressed "lossy" version, and the original picture. Figure 4 shows these two pictures, subtracted, and then the remaining data has been mathematically raised to the 6^{th} power for viewing purposes. The gray areas are where there has been a change in data from the original TIF image in the conversion to JPG.

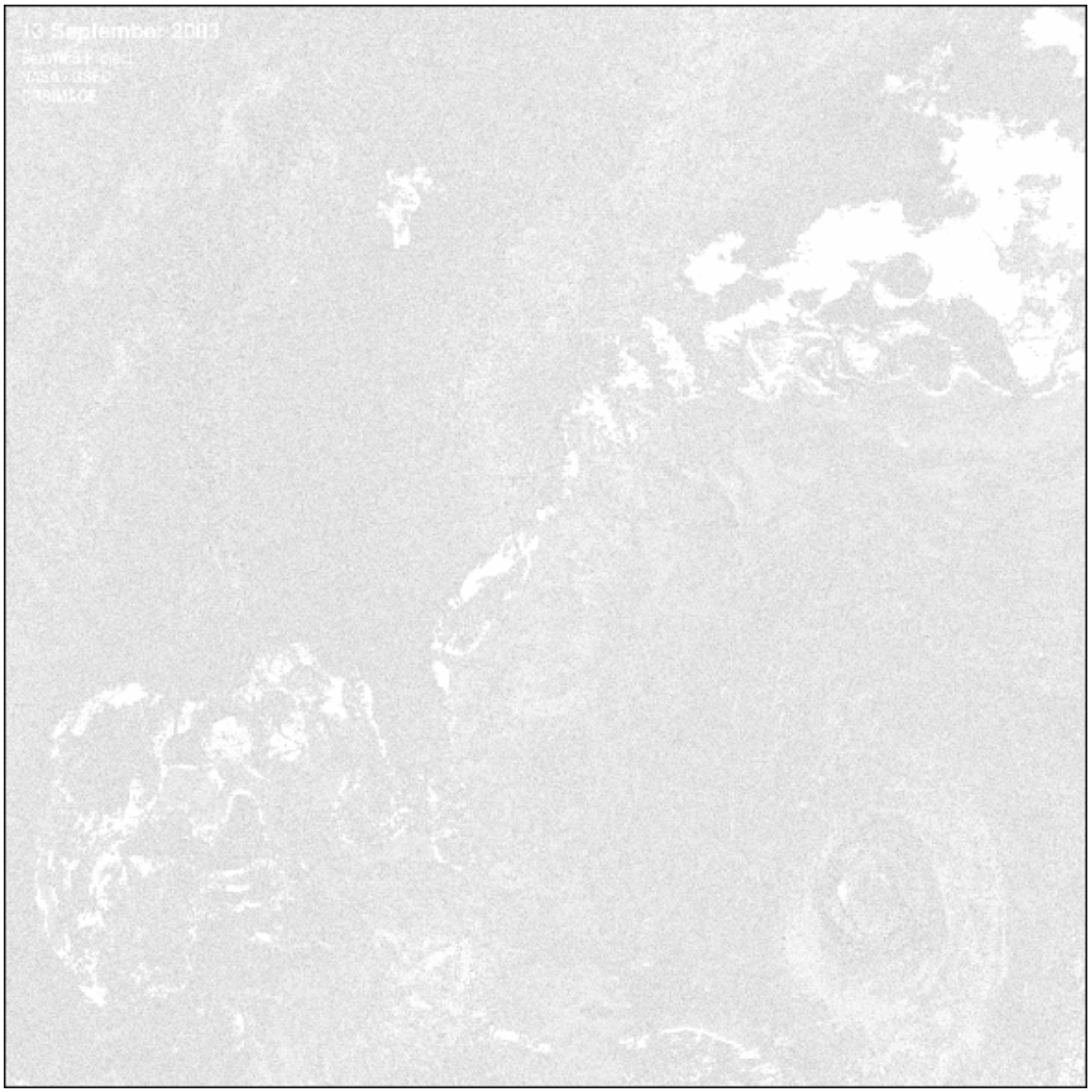

Figure 4: Comparison of data change between JPEG and TIFF

As can be seen, the gray sections are where data is different between the two pictures; the white sections are where the data remained the same. It is noted that there are not any sections that have massive data loss, however there is a difference between the two pictures that encompasses almost 90% of the picture.

2.4.3 Disadvantages to JPEG

Although Joint Photographic Expert Group (JPEG) is a widely used file format, especially in the area of online pictures and those taken by digital cameras, it does have some major disadvantages. The primary downfall to JPEG, which was mentioned in section 2.4.1,

is that JPEG has a "lossy" format and is therefore prone to many problems. As NASA plans to use the compressed file for multiple purposes, such as analyzing the images for specific anomalies, a loss of data of any type will cause the images to be unusable for proper analysis. Another issue with JPEG formatting is that if pixels are related, such as a large number of pixels are all the same color, then a blurring can occur, or more loss is occurring than originally expected. In some newer versions of the JPEG algorithm these issues have been compensated for, however it is still an inherent problem (JPEG FAQ sec 2).

2.5 JPEG-LS Algorithm

NASA's interest in image compression is for satellite image transmission. Sending data to earth requires power and system time in satellites where energy comes at a premium cost. Saving power and system time is of great importance. If an image is compressed on a satellite the only feasible way to do this is in hardware so processor time is minimized. Also, if an image is to remain useful to scientists, every detail must be precisely preserved. These details may not be necessary for humans to appreciate the differences, but in scientific analysis every pixel has useful information. This mandates a lossless compression algorithm that is simple and effective. JPEG is not capable of achieving this, and JPEG2000 is extremely complex.

Although JPEG-LS is not a new algorithm, it is still a very effective and useful compression standard. This algorithm was adapted from Hewlett-Packard's LOCO-I (LOw COmplexity LOssless COmpression for Images) compression scheme, which is focused on low complexity and lossless decompression. The compression ratios are comparable to many other high complexity lossless algorithms; however, some algorithms do better with some images than others. A continuous image is generally compressed best in JPEG-LS although it is still a competitive algorithm with any other type of image.

There are many compression algorithms available for compressing data. Some are specifically aimed at certain types of data. Many of these algorithms use extremely complex mathematical modeling to accomplish the goal of reduced file size. JPEG-LS is a very specialized format for lossless image compression. Although the human eye is capable of discerning extremely small imperfections, our brain filters most of this out allowing us to focus on the important data. This makes it possible to compress analog data such as still images and multi-frame images (video) into formats that discard some of the information that our brain would filter out under normal circumstances. The complexity of compressing an image to one-hundredth of its original size without loosing data to a point that the image becomes unrecognizable has been unachievable until JPEG2000 was introduced.

JPEG-LS is a relatively simple compression algorithm that is designed for lossless compression or near lossless compression. The format offers no resilience to data corruption, little room for future scaling and has no other additional features. For commercial applications this format is of little use because there aren't many applications when an image must be compressed while retaining all of the quality, however for NASA it is ideally suited. The algorithm is lossless, there will be no future changes that might complicate ground procedures, and it is no less resilient to noise than any raw image formats currently employed.

The JPEG-LS algorithm was developed using the LOCO-I algorithm that was developed at Hewlett-Packard Labs in the 1990's (Weinberger, 1). A more detailed description of the algorithm's components is covered in section 5.3. A basic block diagram of the algorithm's operation is shown in Figure 5 and described in detail in the following sections.

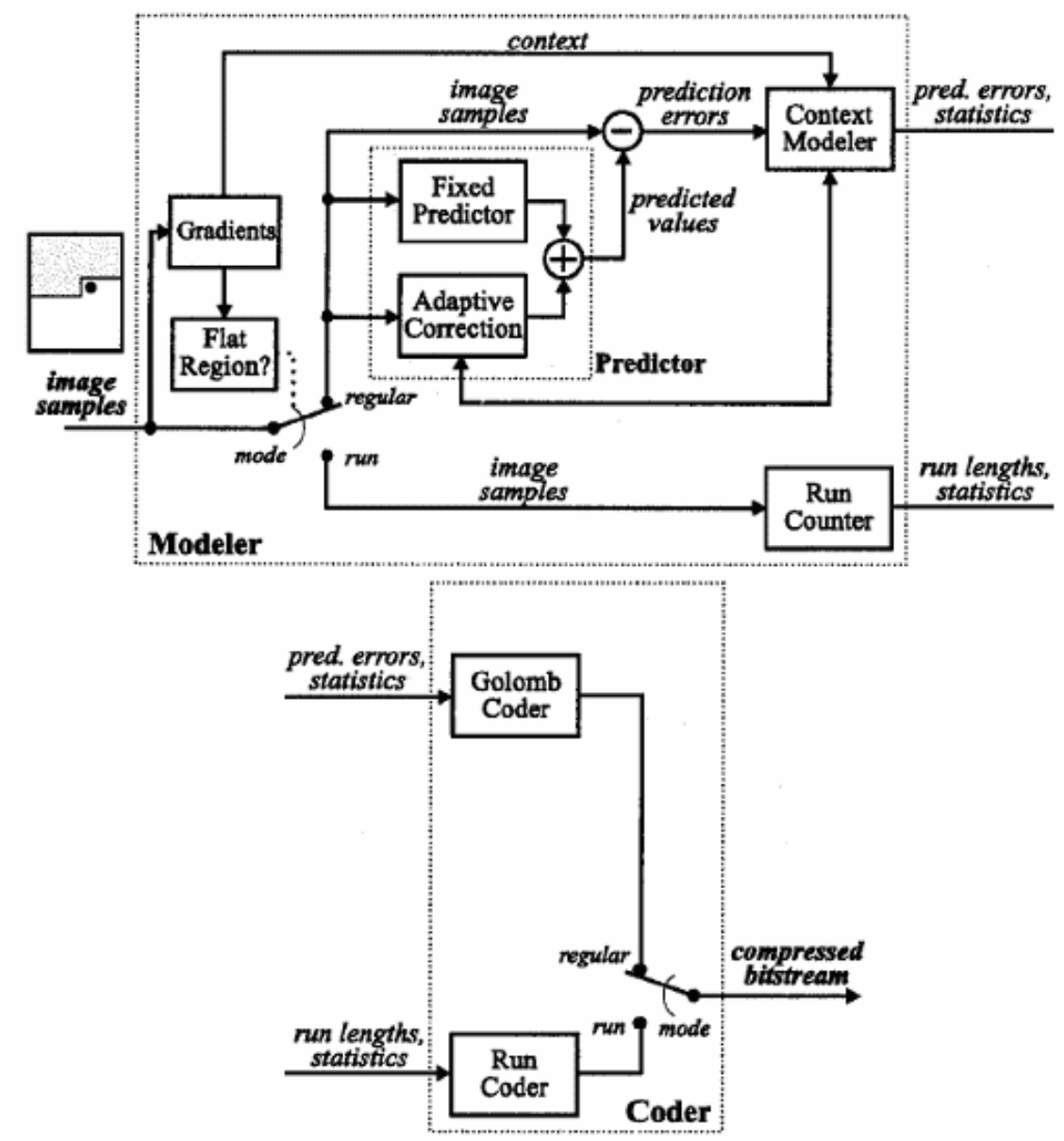

Figure 5: Block Diagram of JPEG-LS Algorithm (Rane 2)

2.5.1 Modeling, Prediction and Correction

The JPEG-LS modeling scheme can be described as an inductive inference problem. The initial portion of the algorithm is designed to formulate a prediction of what each neighboring pixel will be in terms of magnitude based upon neighboring pixels.

After a sequential scan of the image data, neighboring pixel data is used to formulate a probability distribution. This allows the decoder to generate a pixel based on likelihood of a certain color magnitude from surrounding pixels. This first step is divided up into several smaller components.

First, the initial predication step establishes what range a pixel should be in based upon probability from surrounding pixels. Secondly, the context of the pixel is determined and recorded. An example of "context," would be if an edge has occurred, decreasing magnitude or increasing magnitude. Of course, that depends on the direction of the data scan, which will be covered later. Thirdly, the prediction residual, known more simply as the model's error, is recorded.

Since only immediately adjacent pixels are pertinent to this first component, only a few lines need to be scanned into memory at a time for the algorithm to run. This will reduce memory requirements significantly.

2.5.2 Context Modeling

The JPEG-LS algorithm uses data on the context of a particular pixel to help the probabilistic model predict the actual pixel magnitude. These various contexts simultaneously deteriorate another function in the algorithm, but this problem is solved with some creative coding techniques involving the summation of errors, and counting the occurrence of various contexts.

2.5.3 Algorithm Coding

Two-Sided Geometric Distributions (TSGDs) are used to help describe the occurrence of prediction residuals (errors) in the initial probability calculations. This is used with something called a Golomb-Rice coder to elegantly describe the error distributions. These terms will be expanded upon in following chapters.

2.5.4 "Run" Mode

The determination to enter "run" mode is made when neighboring pixels are determined to be the same magnitude. This means that the prediction algorithm will most likely have a prediction residual of zero, so there is no need for further encoding. The data for this area is recorded in the probability equation and the algorithm progresses on.

2.6 JPEG2000

In 1996, the Joint Photographic Expert Group came together again and decided that a new algorithm was needed to compensate for the downfalls of the original JPEG compression algorithm. The idea of JPEG2000 was created in 1996 during one of Joint Photographic Expert Group meetings. JPEG2000, although having the same name as JPEG, is very different; for instance while JPEG is a "lossy" format, JPEG2000 is considered both

lossless and “lossy” (Marcellin 2). To fully understand the decisions made by the JPEG comity it is imperative to understand what lossless means.

2.6.1 Lossless versus “Lossy”

The most common form of algorithm today, due to its abundant use and ease, is the “lossy” JPEG algorithm. The major advantage to using a lossless algorithm is that a user can convert between uncompressed formats, such as bitmap and tiff, to the format and back again without any data being discarded. A “lossy” algorithm would continue to lose quality during each compression and decompression (JPEG FAQ 13). A comparison between TIF and JPEG2000 is shown in Figure 6.

Figure 6: Comparison of storage size between JPEG2000 and TIFF

As can be seen in Figure 6, the file size has been reduced significantly when compressed into JPEG2000 in “lossy” mode. To further stress this point, in Figure 7, a comparison between JPEG and JPEG2000’s file sizes at near to the same compression ratio can be seen.

Figure 7: JPEG and JPEG2000 file size comparison

In Figure 8, it is apparent that the JPEG2000 format retains more quality of the original image and therefore it can be seen why JPEG2000 is becoming a popular algorithm to implement for image compression. All JPEG2000 images were made using the free demo edition of Stardust® Image Encoder 2003. Unfortunately these images are implementing the "lossy" mode of JPEG2000. All images compressed in the lossless mode of JPEG2000 are not significantly reduced in size.

Figure 8: JPEG and JPEG2000 image quality comparison

2.6.2 JPEG2000 Features

JPEG2000 has many advantages over the original algorithm that have been created since its inception in 1996. In a paper from the *Proc. of IEEE Data Compression Conference,* An Overview of JPEG-2000, stated that:

- State-of-the-art low bit-rate compression performance
- Progressive transmission by quality, resolution, component, or spatial locality
- Lossy and lossless compression (with lossless decompression available naturally through all types of progression)
- Random (spatial) access to the bit stream
- Pan and zoom (with decompression of only a subset of compressed data)
- Compressed domain processing (e.g. rotation of cropping)
- Region of interest coding by compression
- Limited memory implementation

(Marcellin 1)

The compression performance noted above refers to the fact that by using the JPEG2000 algorithm, one can get a 20% improvement over current JPEG codices (JPEG2000 Resource par 7). Also, the progressive transmission gives the user complete control over all aspects of the compression.

Beyond the advantages listed above, JPEG2000 has been implemented by Internet Service Providers (ISPs) to streamline the Internet. The ISPs mentioned are the "high-speed" dialup providers that have become available in the last several years. Since graphical media is what slows the Internet down for most users, by compressing pictures further, Internet providers are able to increase the speed of their services. The other reason for its appearance in Internet use is that it is highly resistant to errors that may occur during the transmission of signals over lines or through the air (wireless) (JPEG2000 Resource par 5).

2.6.3 Creation of JPEG2000

JPEG2000 was built on the structure of other lossless algorithms. During the JPEG convention named WG1, in 1997, a challenge was made to all present to take a certain number of samples, all provided, and to compress them at different rates. The rates that were specified were in a range of 0.0625 to 2.0 bits per pixels (bpp). At the next convention all the samples were analyzed, via printing the images and then comparing image quality and also by quality analyzing algorithms that were developed. The top candidate for the basis of the JPEG2000 algorithm turned out to be the wavelet/trellis coded quantization (WTCQ), which

for every test that was run turned out to be the number one qualifier. The WTCQ algorithm, having won the day, was subject to further tests to find features that would be best to implement as the standards for JPEG2000. The basis of WTCQ and, therefore, JPEG2000 is the discrete wavelet transform (Marcellin 2). This discrete wavelet transform is a Fourier series and not easy to understand (Discrete Wavelet Transform par 1).

2.7 JPEG-LS to JPEG2000 – the Lossless Comparison

JPEG2000, the newest standard provided by the JPEG still image compression group is a highly sought after algorithm due to its compilation of features, as well as having a "lossy" and a lossless version. JPEG2000 has several features that make it ideal for a variety of reasons. However, the question must be asked, is it best to use such an algorithm for a purpose that only uses one of its many features? Looking at several lossless algorithms and comparing their compression sizes and compression time may best discover this. The reason for looking explicitly at lossless algorithms has been explained in section 1.

The algorithms that will be explored in the comparisons are that of JPEG2000, JPEG-LS, L-JPEG, BZIP2, and PNG. BZIP2 is a compression algorithm that is built into UNIX. Portable Network Graphics, or PNG, is based on a predictive scheme and entropy coding. For the purpose of testing several images were used. These tests were taken from the JPEG2000 still image coding versus other standards written by D. Santa-Cruz, T. Ebrahimi, J. Askelöf, M. Larsson and C. A. Christopoulos (Sanata-Cruz 5).

All testing was done on a set of test images that were gathered by the JPEG group. As mentioned in the JPEG2000 still image coding versus other standards:

> "The algorithms have been evaluated with seven images from the JPEG 2000 test set, covering various types of imagery. The images "bike" (2048x2560) and "cafe" (2048x2560) are natural, "cmpnd1" (512x768) and "chart" (1688x2347) are compound documents consisting of text, photographs and computer graphics, "aerial2" (2048x2048) is an aerial photography, "target" (512x512) is a computer generated image and "us" (512x448) an ultra scan. All these images have a depth of 8 bits per pixel." (Sanata-Cruz 4)

This explains that each of the pictures is of a different type. Each one contains different information, including everything from simple text in "cmpnd1" to full scenes in "aerial2". This leads to a wide range of test images and allows for a more comprehensive study of what type of image is best compressed by each algorithm. Tables 1 and 2 show the compression sizes of each of the individual pictures that were used as test samples (Sanata-Cruz 4).

	$J2K_R$	JPEG-LS	L-JPEG	BZIP2
bike	1.77	1.84	1.61	1.72
cafe	1.49	1.57	1.36	1.40
cmpnd1	3.77	6.44	3.23	5.65
chart	2.60	2.82	2.00	2.39
aerial2	1.47	1.51	1.43	1.63
target	3.76	3.66	2.59	7.34
us	2.63	3.04	2.41	3.22
average	**2.50**	**2.98**	**2.09**	**3.34**

Table 1: Lossless Compression Sizes (Santa-Cruz 4)

	$J2K_R$	JPEG-LS	L-JPEG	BZIP2
bike	1.77	1.84	1.61	1.72
cafe	1.49	1.57	1.36	1.40
cmpnd1	3.77	6.44	3.23	5.65
chart	2.60	2.82	2.00	2.39
aerial2	1.47	1.51	1.43	1.63
target	3.76	3.66	2.59	7.34
us	2.63	3.04	2.41	3.22
average	**2.50**	**2.98**	**2.09**	**3.34**

Table 2: Lossless Compression Sizes (Ebrahimi 3)

As can be seen from Tables 1 and 2 both PNG and BZIP2 are the best average compression ratios. Although, when taking a closer look at why they have a better average, it can be seen that the target image compression was high, it became an outlier that could be discarded. Overall the best lossless algorithm that was tested was that of JPEG-LS, due to it consistently achieving higher compression ratios then the other algorithms tested. Also, JPEG-LS has a noticeably high compression ratio for the cmpnd1 due to it being a continuous gray-scale image.

As with the comparison of compression ratios, the compression time was also measured for each algorithm. This measurement contributes toward a basic understanding of the complexity of the algorithm that is used. Faster compression is analogous to a simpler compression algorithm, because less time means less passes, less mathematical processing,

etcetera. The Tables that are shown in Table 3 show these compression times relative to JPEG-LS (Sanata-Cruz 5).

	$J2K_R$	L-JPEG	BZIP2	JPEG-LS abs.
bike	4.6	1.9	4.7	2.01 secs.
cafe	4.9	1.9	5.4	2.08 secs.
cmpnd1	7.1	3.6	3.4	0.07 secs.
chart	5.1	2.5	4.9	1.09 secs.
aerial2	4.8	1.9	4.6	1.64 secs.
target	5.5	2.8	5.0	0.06 secs.
us	5.3	1.0	3.8	0.06 secs.
average	**5.3**	**2.2**	**4.6**	-

Table 3: Decoding Times of Lossless Algorithms (Ebrahimi 3)

The compression times of Table 3 show that JPEG-LS is one of the simplest algorithms because it is the quickest to compress each of the algorithms. L-JPEG, being the closest, is on average 2 times slower to compress the images (except in the notable case of "us" where L-JPEG is able to match JPEG-LS).

Further testing was done at Goddard Space Flight Center using an image provided from one of NASA's satellites known as GEO. The test image, known simply as "test image" was saved in the TIF image file format. For the purposes of comparison it was converted only into JPEG-LS and JPEG2000. Figure 9 shows the test image that was used in the comparison. It is a picture of the earth.

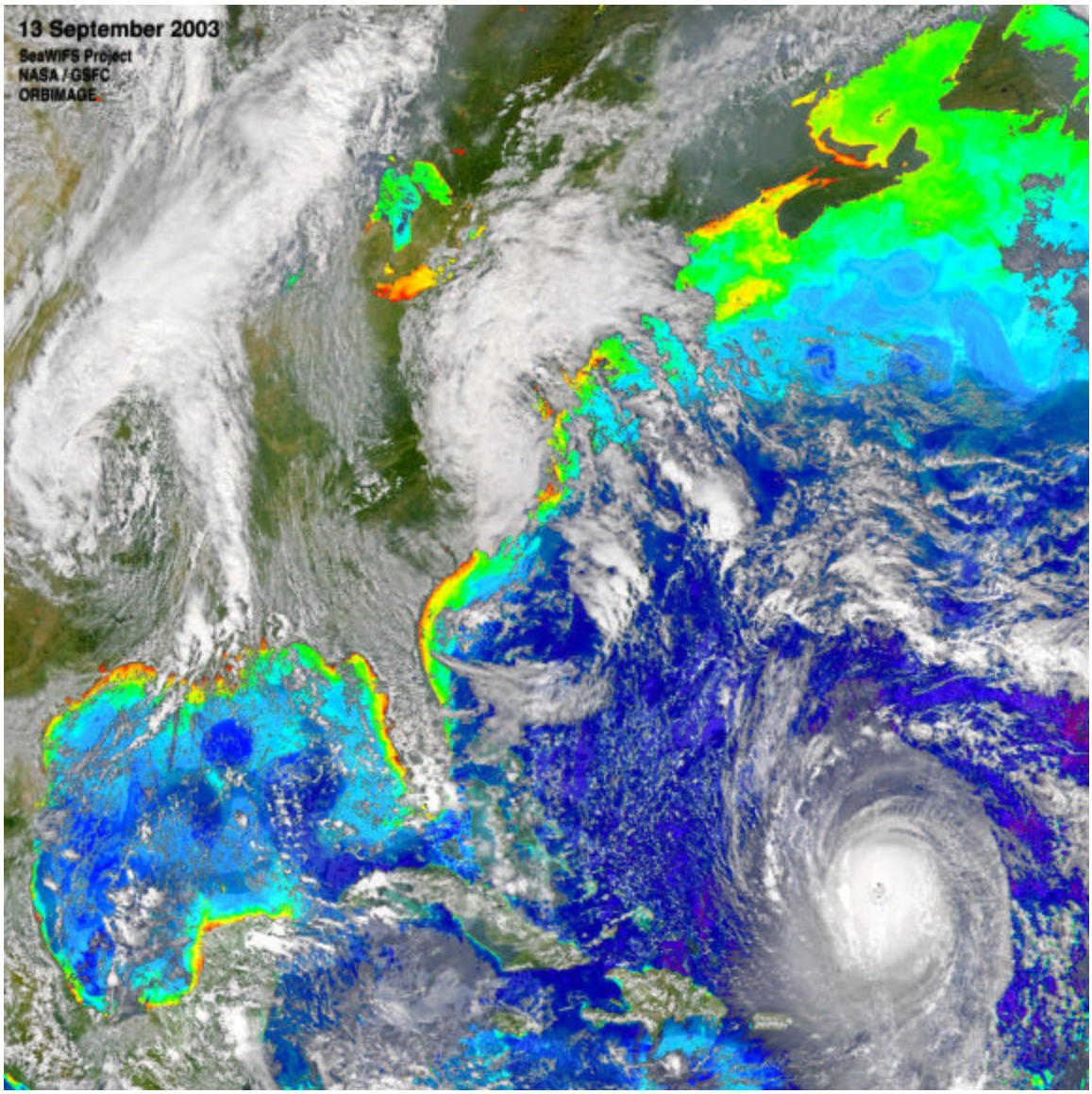

Figure 9: Test Image

The image was then converted into both JPEG2000 and then also in JPEG-LS. Table 4 shows the comparison between the two file sizes while using the original bmp as a comparison.

JPEG2000	JPEG-LS	BMP
20.4MB	20.1MB	35.0MB

Table 4: Comparison of encoding size of J2k to JLS

As can be seen, the JPEG-LS file is slightly smaller than the JPEG2000 image. Although this is only a small difference, when transmitting from space this can save several seconds, which will in turn save power and time. This example was also completed on a 35MB file. The raw data taken off satellites can range up to 130MB or more, which when compressed will only increase the reduction in file size over JPEG2000.

JPEG-LS is the better lossless compression algorithm, from both the results gathered by the study that was headed by Santa-Cruz and the test image data shown in Table 4. Beyond simply being better at compressing images, JPEG-LS happens to also be a simpler, faster, algorithm. Therefore, for the purposes given, JPEG-LS is the clear choice for the lossless compression in VHDL.

2.8 RIT code

While researching the JPEG-LS algorithm we found that the concept of this project has been done before. This was not a great surprise, in view of the fact that the algorithm has been used since it was made standard in 1995. This information was found online, in a paper by Dr. Savakas, who is the head of the Computer Engineering Department at RIT (Savakis). The document was fairly short and very general in scope. With only four pages it described more about the lossless verses lossy attributes of the algorithm than it did about the actual implementation. The document mentioned that this project had been done in VHDL, but nothing was said about an actual hardware testing. Around the same time we discovered that Jet Propulsion Laboratories (JPL) in California had also completed a similar project in Verilog. A very minimal amount of information was available online pertaining to that project.

Upon further investigation we were able to contact both Dr. Savakas and the necessary people at JPL to begin the process of acquiring some information for our work. It was agreed that obtaining this code would provide the most likely scenario for completing the project on schedule.

JPL responded quickly, and supplied us with their code and some documentation. This was a positive thing, but since we have no training or experience with Verilog code we were intimidated by this information and, in the few days spent reading through it, we did not gain any noteworthy information.

Just before we were ready to refocus our goals to understanding the Verilog code, we received the VHDL implementation from RIT. Dr. Savakas turned out to be the advisor to a graduate student who implemented this algorithm as part of his thesis (Piorun). This student, Michael Piorun, wrote an entire encoder and decoder in VHDL. He then simulated the whole

algorithm for timing and result checking. His results are well documented in his thesis and are very encouraging in the sense that his code will most likely work well on an FPGA.

The process of deciphering the algorithm from Michael Piorun's code got off to a good start and we began mapping out the logical function shortly after receiving it. We contacted him via email for some more information and continued reading though the code while awaiting a response.

While reading through his code a hierarchy of all the VHDL components was created to aid the process of understanding the algorithm. Each step of the algorithm is defined in a component. This project was done three years ago, so some of the VHDL coding has changed slightly (e.g. "wired_or"). This required going though all the signals in the code to eliminate any multiple drivers and unchanging signals.

Since several steps require variable amounts of time to complete their operation, a clever scheme of synchronous handshaking was used. All output signals are latched on a clock edge and the handshaking signal indicates that this module has completed its operation and the next module can take the data.

Michael Piorun's code for the JPEG-LS algorithm compiled and successfully mapped after some corrections were made. After this step we were able to view the RTL interconnect map in Synplicity. Now there is a visual structure that can be viewed to help us understand the flow of the algorithm.

2.9 "FIREBIRD" Board

To implement such a complex algorithm as the JPEG-LS compression in VHDL we will need a very capable device and interface. Fortunately, this has already been designed for us to use. The Annapolis Micro Systems "FIREBIRD™ for PCI" board has been assigned to us by our superiors to implement the JPEG-LS compression algorithm in VHDL.

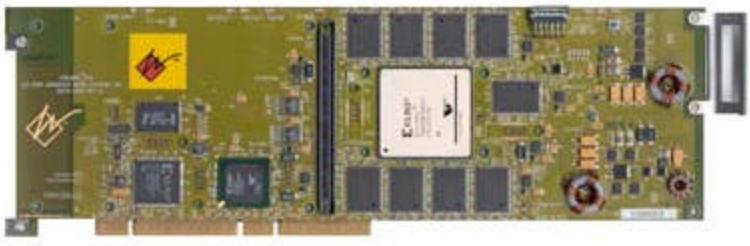

Figure 10: Firebird for PCI card taken from AMS website

This board features up to 2 million system gates depending on which XiLinx "Virtex™ E" chip is used. The board interfaces with a PC via a 64-bit PCI slot with a maximum theoretical bandwidth of 528 Mbytes per second. The FIREBIRD processing

element to I/O board is capable of 3 GBytes per second. The onboard FPGA and memory clock speed is 150MHz, allowing the memory to have a 6.6 GByte per second bandwidth in its banked configuration. The Virtex™ chip interfaces directly with onboard memory and the PCI controller, allowing for a faster interface and less VHDL code to implement. Figure 11 displays the Virtex™ chip and its surrounding support devices.

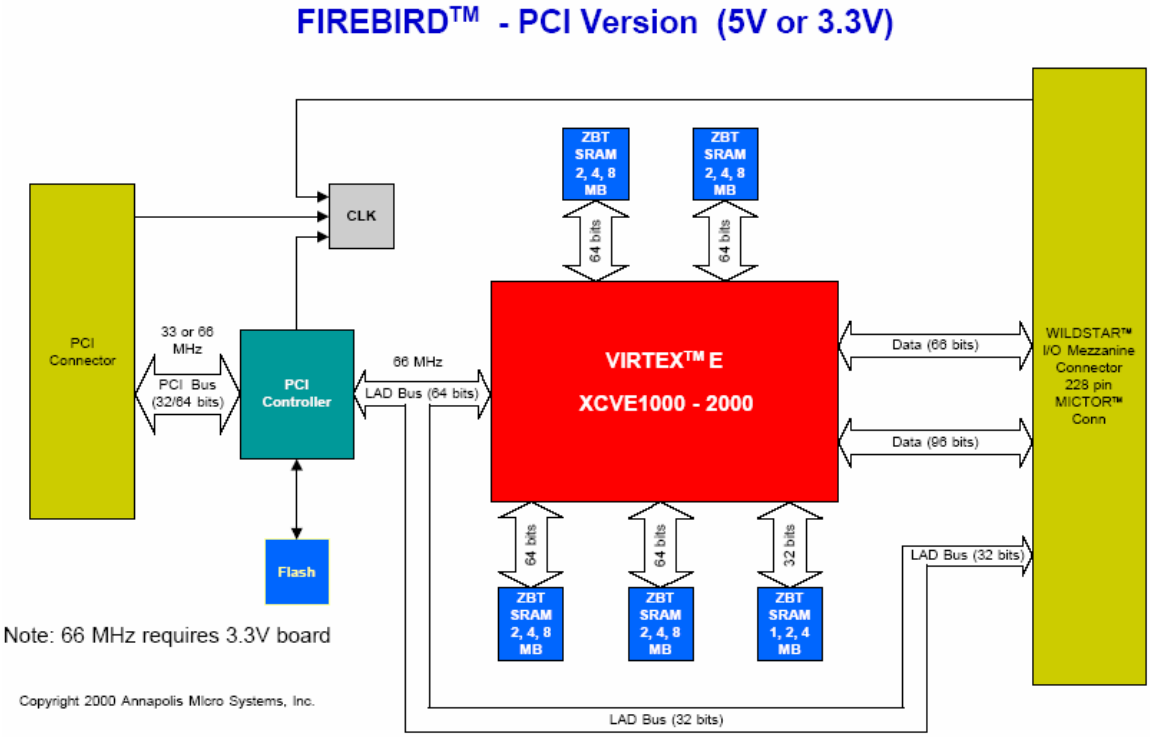

Figure 11: Firebird PCI Flowchart taken from AMS website

Also among this board's excess of power is its ability to program from Flash memory on power-up. The onboard memory may be configured from 9 to 36 Mbytes of synchronous ZBT SRAM in five different memory banks. Zero Bus Turnaround (ZBT) is a synchronous SRAM type that takes full advantage of available bandwidth. This can be accomplished by the memory module's ability to switch between write and read without needing idle clock cycles. This particular board also features voltage, current, and temperature monitoring capabilities.

The Virtex™ chips also have as many as 624 available I/O depending on model. These chips also have eight built in clock multipliers, up to 56 embedded 18x18 digital multipliers, and several different built in memory configurations. The internal multipliers may be clocked up to 210MHz. The Virtex™ devices are specifically optimized for DSP (Digital Signal Processing) applications. This will be very useful to help us implement the JPEG 2000 wavelet portion of the algorithm.

Physically the Virtex™ chips are built on a 0.15 micron 8-layer aluminum process with high-speed 0.12 micron transistors (Xilinx datasheet, 13). This technology supports operating I/O voltages of 1.2 volts to 3.3 volts. The device's outputs are even divided into four separate banks to allow for multiple I/O operating voltages. However, the core of the FPGA always uses 1.5 volts.

These Virtex™ FPGAs also utilize IP-Immersion (Intellectual Property) technology to further the usefulness of the device as a SoC (System on Chip). Different IP processor cores may be embedded into the FPGA for easy and fast access by the internal logic. IP processor cores may also be seamlessly integrated as a soft device in the FPGA to provide predictable interconnect timing. Any IP core may also be connected to external pins for maximum flexibility.

2.10 Summary

The need for this project directly relates to NASA's need for images to be transferred from space both faster and in a format that allows all the image data to be completely recovered. The reason for the recovery is that there is a need for machine analysis to be performed on the images that are obtained from space (whether satellites or other vehicles in space). This machine analysis can range from testing for clouds in pictures of earths landmass to that of detecting particles in space by identifying colors or spectral analysis. The need for an image to be completely recovered with no loss of data lead to the realization that something better than the tiff file format that is currently being used for compression of images from space is needed.

Research has been done to determine the best of the image file compression formats. This analysis explored many formats: JPEG, JPEG2000, and JPEG-LS. JPEG would not suffice do to the lossless standard being almost too simplistic and not achieving a high enough compression ratio (only having an average a little over 1, meaning there is no significant compression). JPEG2000 was the next one to be eliminated, but only due to it complexity compared to JPEG-LS. JPEG-LS was the decision for the compression algorithm, both for its fast compression time and its high compression ratio and for providing a lossless format. JPEG-LS has an average ratio of about 3 and an average compression time of about 1 sec on a Pentium III system. All test data and comparisons can be seen in further detail in section 2.7. The procurement of the JPEG-LS VHDL was a large help in moving this project along. This was carried forward thanks to Dr. Savakis, Head of the Computer Engineering Department of Rochester Institute of Technology, as well as Michael Piorun, the graduate student that wrote the code for his thesis requirement.

NASA also expressed an interest in this compression algorithm to be implemented in hardware, rather than in software. This could allow the hardware implementation to be installed in a new satellite as a permanent fixture. This would require less memory and would provide a degree of protection against corruption of the algorithm in a program's memory from radiation in space.

The test board for this project is the Annapolis Micro Systems Wildstar Firebird with a XiLinx Virtex 2000 FPGA. It is a capable test board that, for our version, is configured with 36 MBs of onboard memory. This board can support the JPEG-LS algorithm created in VHDL. This system should be able to achieve operating speeds far beyond any software implementation. Further testing of a working algorithm will be needed.

3 Problem Statement

3.1 Introduction

The purpose of this chapter is to present the goals, objectives, and tasks that were needed to complete this project. The objectives and tasks are presented in the order that each was to be accomplished to achieve the overall goal.

3.2 Problem Statement

The goal of this project was to implement the JPEG-LS algorithm in a hardware description language (HDL).

To accomplish our goal of implementing JPEG-LS it was necessary to identify certain objectives and the underlying tasks. These objectives and tasks are discussed in this section for the purpose of clearly setting out steps that were accomplished for the overall goal to be achieved.

3.2.1 Understand Mathematical Functions

The tasks that must be done as a part of the objective of developing an understanding of the mathematical functions pertaining to JPEG-LS were:

- Reviewed Calculus, Linear Algebra, and Fast Fourier Transforms
- Developed an understanding of prediction modeling, context modeling, and Golomb coding

3.2.2 Implementing VHDL

In order to implement the JPEG-LS algorithm in VHDL, we used the original code developed by Michael Piorun for his graduate thesis. Once the code had been obtained, the following tasks were performed.

- Developed flow charts and block diagrams in order to understand how the original code functions
- Synthesized the code and tested functionality

3.2.3 Operating the Firebird Board

The primary task of this objective was to discover the intricacies of operating the board through the help of individuals such as Dave Patrick and Mosses McCall, as well as

through the use of a training manual provided. The main portions of the board that must be explored were:

- Writing the makefile
- Design C code that operates the board
- Map our VHDL code to the Virtex chip

3.2.4 Optimizing VHDL

Because the code was obtained from Dr. Savakis, head of Computer Engineering at Rochester Institute of Technology, certain parts of the VHDL obtained were created generically, and are not needed when using the Firebird board. Due to these variations in project structure, the VHDL code requires streamlining and editing to allow it to properly function on the Fireboard FPGA board.

3.3 Summary

In this chapter the objectives and tasks associated with our goal of implementing the JPEG-LS algorithm in VHDL were presented. These objectives were to develop an understanding the mathematical functions pertaining to JPEG-LS, obtain the JPEG-LS algorithm from RIT, test and operate the Wildstar Firebird FPGA board, and to optimize the RIT VHDL code.

4 Methods

4.1 Introduction

In this chapter the methods of implementing the JPEG-LS algorithm in VHDL are explored in relation to the goals, objectives and underlying tasks that were discussed in section 3. Through research on the Internet, Library of Congress and other Goddard Space Flight Center facilities, we developed the methods that allowed us to gain a full understanding of the JPEG-LS algorithm.

4.2 Methods

4.2.1 Calculus, Linear Algebra, and Fast Fourier Transforms

The first required task was to refresh our mathematical skills in the areas that applied to the implementation of the JPEG-LS algorithm. The approach we took towards this end was:

- Reviewed the JPEG ISO standard
- Reviewed the JPEG-LS public draft
- Reviewed Calculus (MA1023 & MA1024) notes and book from WPI
- Reviewed Linear Algebra (MA 2071) notes and book from WPI
- Reviewed Statistics (MA 2611) notes and book from WPI

By reviewing these mathematical concepts, we achieved a major step in identifying the flow of the program and where pipelining would best apply to the system.

4.2.2 Prediction modeling, Context modeling, and Golomb coding functions

In order to gain an acute understanding of the detailed mathematical functions that the JPEG-LS algorithm uses, we needed to study:

- Golomb coding (Adaptive Coding and Golomb-Rice coding)
- Prediction Modeling (Adaptive Predictors)
- Context modeling (an Overview of JPEG-LS)

4.2.3 Flow Charts and Block Diagram

The next task that needed to be completed was to create flow charts and a general block diagram of the JPEG-LS algorithm. This task was completed after the RIT code was compiled and mapped using Synplicity. Synplicity created a logical map of the algorithm

including all interconnect wires and component names. From this map the flow of the algorithm was determined, which helped us gain a better understanding of the VHDL. We were also able to identify parts where further improvements could be made.

4.2.4 Synthesis

The next task was that of actually synthesizing the JPEG-LS algorithm in ModelSim. This task was important and all previous research was needed for completion. The synthesis in ModelSim required:

- Reviewed ModelSim simulation tools and timing layouts
- Reviewed Firebird Training Manual on synthesis in ModelSim
- Contacted ModelSim specialists at ModelSim

We decided that the best route to take was to follow the instructions provided for the Firebird board and to get sample code to synthesis in ModelSim. This required some work, as well as writing our own VHDL code and testing that first, as per the instructions of a training manual provided by Annapolis Micro Systems.

4.2.5 Operating the Firebird Board

The next step that was required is that of porting the VHDL code to the Firebird board. The approach we used to accomplish this task was:

- Reviewed Firebird Training Manual on sample programs and design layouts
- Reviewed ModelSim compilation requirements
- Wrote C Host Code
- Reviewed notes from Dave Petrick on previous projects

Through the use of these many resources we were able to gain a basic understanding of how to operate the Firebird board.

4.2.6 Optimizing VHDL

The original RIT VHDL was not designed to be implemented on any particular chip. The RIT code was altered because it had not been designed to use external memory. In this instance, the Firebird board actually contains 36 megabytes of on board memory that can be used for both data storage, and temporary arrays needed by the algorithm. Also, some VHDL was removed because all that is required for this project is the encoder; the RIT code

implements both encoder and decoder, as well as options for the near-lossless mode that is supported by JPEG_LS.

4.3 Summary

Everything from how we figured out the mathematics behind JPEG-LS to the actual implementation of the algorithm was discussed in this chapter. Our main resources were those of the RIT VHDL provided by Dr. Savakis (head of Computer Engineering at Rochester Institute of Technology), David Petrick, the JPEG ISO standard, the JPEG-LS public draft, and the Firebird training manual. Through these resources the understanding that was required to move on to the development of the system design was gained.

5 System Design

5.1 Introduction

In the following sections the how and the why of each of the individual steps that were taken to develop the overall system design for the JPEG-LS algorithm in VHDL are discussed. In specific the Software Design, Hardware Design, and then how the two, software and hardware, interface and work together as an overall system are discussed.

5.2 Software Design

In order for the VIRTEX 2000 chip to obtain the data needed for encoding, specific steps will need to be completed before the image is ready to be processed by the hardware. These steps will be performed in software, which can be scaled depending upon the application in which this design is being used. The particular language used is not of vital importance, as long as specific requirements are met for passing information to the FPGA. These requirements are described in section 5.4.

In order for the FPGA chip to start processing the data, certain information must be provided. Some of the information that must be provided is the number of columns, number of rows, and other specific bits that are needed to start the encoding. The image also needs to be loaded into the memory elements that are dedicated for the FPGA. This is best accomplished by using a direct memory access (DMA) method. DMA loads the data into memory much faster then regular memory access because DMA takes priority on the bus for the duration of the transfer.

5.2.1 Memory Paging

The environment in space is filled with radioactive particles that have the ability to disrupt the operation of electronic components. Memory modules are also subject to these vulnerabilities, and if a memory module is disrupted, it is possible for the data to be altered. Without error checking algorithms or redundant systems, this change in data could be irreversible. Due to these complications, it is standard practice to attempt to reduce all physical memory to minimal sizes to reduce the chance of data corruption.

If an image is larger then the allotted memory for the FPGA, a method of memory paging must be implemented. One section of data will be sent to the memory, and then once that data has been processed, will be overwritten with new data. Through the use of memory paging, a 30MB file could be processed using 4MB of memory. The first 4MB of the file would be loaded into memory, processed, and then sent into storage. The memory would

then be overwritten with the next 4MB of data, and the process would continue until the end of file is reached.

5.2.2 Image extraction

At some point in the image processing, the data needs to be removed and stored for transmission or archival purposes. This can be accomplished by using a dedicated memory bank to be read every time a compression cycle has completed, or the compressed data could be streamed out of the processor and written directly to a file, or streamed into a transmission signal to a ground base. Each of these options may work better for one application over another. Buffering the compressed data and then writing to a file all at once may be a better option for an autonomous vehicle over streaming the data to a ground base or overhead satellite. These decisions will need to be made by a project engineer responsible for the completion of this venture.

5.3 Hardware Design

The Hardware Design refers to that of the design of the HDL that was provided for this team by Michael Piorun, a graduate student from Rochester Institute of Technology. There were minor changes done to the original RIT VHDL because of the need only for the Encoder section of the HDL and due to the need to make it so that the HDL functions properly on the Firebird FPGA card that has been provided. This HDL will be utilized on the FPGA, therefore why this is considered the "Hardware design". This is referring to the design of the JPEG-LS Encoder parts of Piorun's VHDL in lossless mode of encoding only. The components that form the encoder of the JPEG-LS VHDL design each perform a specific task of the JPEG-LS algorithm. These components are:

5.3.1 Get Next Sample

In order to start encoding a given pixel, a few pieces of information must also be gathered, such as the four surrounding pixels that will be used to predict the value of the current pixel.

Ra	*Rb*	*Rc*
Rd	*x*	

Figure 12: Pixels for Modeling

Figure 12 shows the reconstructed values *Ra*, *Rb*, *Rc*, and *Rd* that have been retrieved from memory. The component GET_NEXT_SAMPLE retrieves these values from memory as shown in Figure 12 so that analysis can be performed on the pixels to determine the

predicted value of the pixel labeled as X. This prediction is actually handled in the ENCODE_RUN_LENGTH component as is discussed in section 5.3.8. Michael Piorun has created a table that defines a few special rules for what values are used when the image boundary has been encountered. This table can be found on page 49 of his masters thesis document.

Pixel	Position	Exception	Value Assigned
Ra	west	col = 0 and row = 0	0
		col = 0 and row ≠ 0	*Rb*
Rb	north	row = 0	0
Rc	northwest	row = 0	0
		col = 0 and row ≠ 0	*Ra* from previous time col = 0
Rd	northeast	row = 0	0
		col = N_COLS-1 and row ≠ 0	*Rb*

Table 5: Image Boundary Rules

These rules have been designed because there are instances when certin pixel data is not available, for instance at the top of the image, when there is no row above pixel X. "To follow these rules, flags were used in the code whether to retrieve certain values from memory. After the flags were set, values were retrieved one by one from the image memory in the following order: *x* (current pixel value), *Rb*, *Rc*, *Rd*, and *Ra*. The main reason for this ordering is to retrieve *Rb* before *Ra* and *Rd*, in the case that *Rb* needed to be assigned to *Ra* or *Rd*." (Piorun, 49)

5.3.2 Fill Image Row

Once the GET_NEXT_SAMPLE component acknowledges a start signal, the FILL_IMAGE_ROW block reads the values of the pixels in the next N_COLS, and loads them into the image memory. While this is happening, the GET_NEXT_SAMPLE component waits until he entire row has been filled in order to not drive the image address signals at the same time, causing a conflict in where the information is stored.

The new row replaces the "previous image row" by a handshaking signal called read_input that is connected to the input stream. Michael Piorun designed this to read a single byte at a time, overwriting the previous data. "A new row is read in by overwriting the previous 'previous image row' in an alternating fashion, by pulsing a read_input signal to the input stream block. In this design, the input stream reads a byte at a time from the input file. In a hardware chip, a separate device must conform to the specification of providing the next pixel value when this signal is asserted. The previous 'current image row' first becomes the

'previous image row' simply by toggling a bit instead of copying the values into the other row." (Piorun, 49)

5.3.3 Find Context

The primary function of FIND_CONTEXT is to find the conditioning state, or context, integer Q (Piorun 50). Comparing the differences of the pixel values of *Ra, Rb, Rc*, and *Rd* and then finding the corresponding threshold values accomplish this. *Ra*, *Rb*, *Rc*, and *Rd* are further explained in section 5.3.1. The threshold values, T1, T2, and T3, have a value by default of 3, 7 and 21 when in the context of an 8-bit grayscale image. These threshold values are used to find the quantized gradients (q1,q2, and q3) when the differences of *Ra*, *Rb*, *Rc*, and *Rd* are calculated (Piorun 34). The differences, d1, d2, and d3, are found by:

$$d1 = Rd - Rb$$
$$d2 = Rb - Rc$$
$$d3 = Rc - Ra$$

Equation 1 (Piorun Eqn 3.1)

Through the use of the threshold values these differences are then categorized into one of the quantized gradients. There are nine different possible values that the quantized gradient can assume, from -4 to 4. As Michael Piorun explained in his thesis, the input output function can be easily shown in the form of a graph, as seen in Figure 13 (Piorun 34).

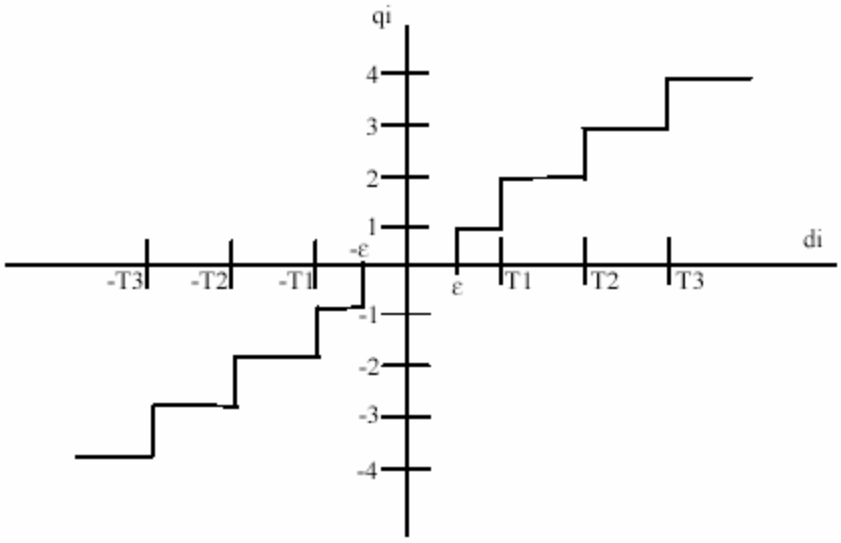

Figure 13 Quantized Gradient Vs. Threshold Graph (Piorun 34)

The way that quantized gradient handling is done by the FIND_CONTEXT component is that it calls upon a jpegls_pkg file called GetQi which handles everything from what is described in Equation 1 to the comparison shown in Figure 13. To indicate whether

the sign of the quantized gradients signs were flipped due to symmetry in the case where the first non-zero is negative, the variable SIGN was utilized. Once the sign and each individual quantized gradient is found, so can the context Q be solved for (Piorun 50). The equation that is used to find the context Q is:

$$Q = 81q_1 + 9q_2 + q_3$$

Equation 2 (Piorun Eqn 3.2)

Due to timing constraints, a multiplication function was not used to represent Equation 2, but rather indexing arrays were utilized to solve for the context Q, thus increasing speed. Q being computed completes the main process of this component.

The FIND_CONTEXT component is started by the GET_SAMPLE_DONE signal being asserted. The GET_SAMPLE_DONE signal is driven by the GET_SAMPLE component and is asserted when GET_SAMPLE has finished calculating the values of the reconstructed and current pixels. Once the GET_SAMPLE_DONE signal has been asserted, upon the next clock cycle the context will be found, unless the context is skipped by the SKIP_CONTEXT signal. The SKIP_CONTEXT signal is driven by the GET_NEXT_SAMPLE, and is asserted while processing future run mode pixels. If the context is found then it is possible to exit the run before properly encoding the pixel, which interrupted the run. This is due to the FIND_CONTEXT component being, that which asserts the REG_RUNB_MODE signal (Piorun 50).

5.3.4 Predictor

The PREDICTOR module is the heart of the whole algorithm. It is dependant on a simplified statistical model for error. Since the prediction model is standardized for the encoder and decoder, its operation is exactly copied between the two. Thus the resulting prediction value can be encoded while the rest of the raw input data can be discarded after the prediction.

The first step in the predictor's operation is finding an approximate value for pixel "x" to calculate its value based on the surrounding pixels. This value is then compared with the actual pixel value being indexed. The resulting error is all that needs to be encoded, since the decoder uses the same prediction scheme. After that, the decoder just adds the encoded error value to the result to obtain the original pixel value. This works well because error or difference values usually have lower entropy than the values themselves (Piorun 37).

The predictor module uses the following function to compute the predicted value.

$$Px = \begin{cases} \min(Ra, Rb) & \text{if } Rc \geq \max(Ra, Rb) \\ \max(Ra, Rb) & \text{if } Rc \leq \min(Ra, Rb) \\ Ra + Rb - Rc & \text{otherwise} \end{cases}$$

Equation 3: (Piorun Eqn 3.3)

This function will be sensitive to an edge in an image, and thus will be more accurate in the prediction. The resultant value "Px" is further corrected by a context correction factor, which is stored in "C_array(Q)." This correction variable is adapted throughout the full run of the algorithm, to help minimize repeated errors.

Since this error would follow a symmetrical distribution, the encoder can encode the resulting error value using Golomb-Rice coding. The first step in that process is to calculate the Golomb variable "k." This value is dependant on the number of occurrences of a given context, and the summation of the prediction errors for that context. These values are stored in N_array(Q) and A_array(Q) respectively.

5.3.5 Encode Register Error

The ENCODE_REG_ERROR component of the JPEG-LS Encoder handles the proper encoding of the prediction error (*MErrval*) into the Golomb coder module. The first task that this portion of the Encoder performs is that of finding the variable *k*. *k* is dependent upon the number of times a context is encountered (*N[Q]*) and the sum of magnitudes of prediction errors (*A[Q]*). The way that *k* equates to *N[Q]* and *A[Q]* is shown in Equation 4 (Piorun 51).

$$k = \log_2 \frac{A[Q]}{N[Q]}$$

Equation 4 (Piorun Eqn 3.8)

Equation 4, represented in HDL, is accomplished converting *N[Q]* from integer format to STD_LOGIC_VECTOR format. After the conversion, *N[Q]* is shifted until *N[Q]* is greater than or equal to *A[Q]*. The number of shifts that are done to accomplish this represents *k* (Piorun 51). From the variable *k*, *MErrval* (the prediction error) can be calculated by using *Errval* and Equations 5 and 6. *Errval* represents the error that is found between a single pixels predicted value and the correct value from the original pixel, or how far off the prediction is for a single pixel. *MErrval* is actually the overall mapped predicted error, or rather all the error of all the predicted pixels. Which equation to use is determined by the value of *k*, meaning if $k \neq 0$, then the normal equation (Equation 5) is used, if $k=0$ and

$B[Q] \leq -\frac{N[Q]}{2}$ then the special case (Equation 6) is then used (Piorun 38). The case where $k \neq 0$ is given by:

$$MErrvall = \begin{cases} 2 * Errval & \text{if } Errval \geq 0 \\ -2 * Errval - 1 & \text{if } Errval < 0 \end{cases}$$

Equation 5 (Piorun Eqn 3.9)

The special case equation for mapping the error values is:

$$MErrvall = \begin{cases} 2 * Errval + 1 & \text{if } Errval \geq 0 \\ -2 * Errval + 1 & \text{if } Errval < 0 \end{cases}$$

Equation 6 (Piorun Eqn 3.10)

With MErrval mapped, the next function of the ENCODE_REG_ERROR component is to encode this with the Golomb-Rice code on the next clock cycle and is then sent to the output file stream (Piorun 51).

5.3.6 Update Regular Variables

The UPDATE_REG_VAR component is the last step of the regular mode encoding of a pixel. This component does the updating of the context-specific variables. These variables are that of *A[Q]*, *B[Q]*, *C[Q]*, and *N[Q]* (Piorun 51). These variables, as discussed in Michael Piorun's thesis document on page 36, are the cumulative sum of the magnitudes of the prediction errors, the sum of prediction errors after correction, context-specific correction value to offset prediction bias, and the accumulated number of context occurrences in the image, respectively.

The UPDATE_REG_VAR component, once ENCODE_REG_ERROR is finished, will increment the *N[Q]* variable to designate that a sample has been processed with that context (Piorun 51). Once *N[Q]* has been updated, the magnitude of the quantized error is added to *A[Q]*; the signed, quantized error is added to the bias variable *B[Q]*; and then *B[Q]* allows an update to the prediction correction variable C[Q] by at most +/- 1 for each sample. Once the context variable, or rather *N[Q]*, reaches the default value of 64, or otherwise known as the RESET value, *A[Q]*, *B[Q]*, and *N[Q]* are halved so as to not cause a memory overrun error to occur for any of the variables (Piorun 39). To accomplish this, the component converts the integer values to STD_LOGIC_VECTOR and then performs a shift right by one. This procedure is still done even when the integer values are negative, although this is no longer a division by two in that instance (Piorun 51).

5.3.7 Find Run Count

The FIND_RUNCNT component is responsible for counting the numbers of pixels that are different in a given run. "A pixel is counted as long as the magnitude of the difference between that pixel and the reconstructed previous value *Ra* is less than the near-lossless threshold (or 0 for fully-lossless encoding)." (Piorun, 52) As this project is dealing with only lossless compression, the magnitude must be zero.

A "run" is the amount of data that will be encoded into a single block of data. This may be one pixel or it may be many pixels, but the length of a run depends on the image data or the end of a current line of data. With regards to this project, each pixel must be exactly the same to the pixel next to it. In a near-lossless system the tolerance value is also assessed when determining the end of the run length.

Once a pixel has been assessed and determined to fall within the run, then the value *Ra* is reconstructed and written back to memory and a new pixel is requested for analysis from the GET_NEXT_SAMPLE component. Once the FIND_RUNCNT has determined the end of the run length, then the information is passed on to the ENCODE_RUN_LENGTH block for processing.

5.3.8 Encode Run Length

The functional goal of run length encoding is to mass a line of similar or same colored pixels into one pixel, while effectively encoding the length of that line so that it may be reconstructed by a decoder. Run length encoding is triggered by a series of pixels that have identical values for lossless encoding or are within a certain tolerance of each other for lossy encoding. A run may be interrupted by a pixel that is out of the tolerance range or if the end of a row is encountered.

Once the count is obtained it is broken down into groups of quantities equal to a power of two. A look up table is consulted for the encoded value to cut down on complex calculations. The value to be encoded is encoded with binary '1's while any remaining pixel counts that are not divisible by the power of two are encoded with a series of binary '0's. Each time a run length encoding is found with an identical length, the corresponding variable in the table is incremented. Similarly, if an encoding is found to be less than the previous run length, then the corresponding table value is decremented. This provision is designed to make the table dynamic, to adjust to the particular image's characteristics.

The primary function of the VHDL component implemented in the encoder is to encode the run length into individual segments, of a certain magnitude as described above. When the encoding values need to be parsed, two lookup tables are used. One is for the code

order and the other is for the code length. This eliminates the need to calculate the two values on the fly, which would take more time. (thesis, 52)

When a run of length power of two is encoded, the index of a look up table is increased by one, moving towards greater length values in the next run. The value "rk" represents the actual run length. In the next run length, encoding a binary value of '1' represents a longer run, thus increasing the compression. The associated value in the "rm" column gets encoded in binary to minimize size. This algorithm adapts nicely to different characteristics of the image, since the incrementing and decrementing of the index is adaptive. Michael Piorun's look-up table is represented below in Table 6.

Index	rk	$rm = 2^{rk}$	Index	rk	$rm = 2^{rk}$
0	0	1	16	4	16
1	0	1	17	4	16
2	0	1	18	5	32
3	0	1	19	5	32
4	1	2	20	6	64
5	1	2	21	6	64
6	1	2	22	7	128
7	1	2	23	7	128
8	2	4	24	8	256
9	2	4	25	9	512
10	2	4	26	10	1024
11	2	4	27	11	2048
12	3	8	28	12	4096
13	3	8	29	13	8192
14	3	8	30	14	16384
15	3	8	31	15	32768

Table 6: Run length look up table

The values in this table are stored as a constant array in the VHDL code. These values are referenced in the algorithm and compiled as part of the physical hardware implementation.

If the next run ends at a value that is less than the previously indexed code length, then the index is decremented. This has a similar effect in the case of a trend, since it moves the indexing towards smaller values, in which a single '1' would represent a shorter run. This also adapts to the image. If the run is interrupted before the length is equal to the code

order, a single '0' is used to indicate this, followed by the actual run count and the value that interrupted the run. In this event the "ENCODE_RUN_INTERRUPTION" component is given a signal to properly encode the run interruption sample.

When an end of a row occurs in the image it is not necessary to encode the run interruption sample, since there was none. In this case, control is transferred to the component responsible for obtaining the next sample in the image ("GET_NEXT_SAMPLE"), as soon as the length encoding is finished.

5.3.9 Encode Run Interruption

The ENCODE_RUN_INTERRUPTION component has two main functions; that of encoding the run interruption sample as well as to update the run interruption variables. Michael Piorun grouped these two functions together due to the update function being so small and such an insignificant process that it was actually easier to make one component that took care of two main functions. This was much easier then dealing with all the added port mapping and other similar formalities that would have needed to be done had the updates been done in another component (Piorun 53).

The first function of the ENCODE_RUN_INTERRUPTION component is to actually encode the run interruption sample. This is done by finding the context *RItype* and generating *Q* using Equation 7, which is defined (Piorun 41).

$$Q = RItype + 365$$

Equation 7 (Piorun Eqn 3.11)

RItype is the type information provided in both the hardware format and software format of JPEG-LS. There are two types; type 0 where the prediction for the next pixel is based on *Rb* and type 1 where the prediction is based on *Ra*. Type 1 represents a smooth image while type 0 generally represents a sharp edge. The reason that 365 is added to RItype is because Q represents the location in memory where the N and A array (memory locations) variables are stored. The first 365 locations are used for regular mode encoding as discussed in section 5.3.5 and the single run mode. When in type 0 context, the *k* is then calculated as in regular mode (Equation 5) for Golomb-rice encoding. If type 1 is being utilized then a special equation must be used to calculate a temp variable that is used in place of *A[Q]* in the solution of *k* (Equation 6) (Piorun 41). This temp variable is calculated by the equation given below:

$$Temp = A[Q] + (N[Q] >> 1)$$

Equation 8 (Piorun Eqn 3.12)

Through the use of Equation 8 the error value can be mapped and then be encoded much the same as in the regular mode encoding (section 5.3.5). The error must first be computed to a positive value before it can be encoded into Golomb-Rice coding (Piorun 41).

The ENCODE_RUN_INTERRUPTION component also has the function of updating the run interruption variables. This is done very similarly to that of the UPDATE_REG_VAR component in that when *N[Q]* reaches the default (RESET) value (64); *N[Q]*, *A[Q]*, and *Nn[Ritype]* are reset. The major differences between this function of the ENCODE_RUN_INTERRUPTION and that of the component that updates the regular variables is that *Nn[Ritype]* is an internal signal used only for this component. As Michael Piorun explains in his thesis, *Nn[Ritype]* is the variable used to "indicate the number of occurrences of that context with a negative error value (Piorun 53)."

5.3.10 Encoder Memory Control

The ENC_MEM_CNTRL block deals with managing the memory arrays designed by Michael Piorun. Because so many different components need to access memory, this block regulates which component access any given array. The components that will need to access memory consist of FILL_IMAGE_ROW, GET_NEXT_SAMPLE, PREDICTOR, and FIND_RUNCNT. ENC_MEM_CTRL has been designed in such a way that component A can access memory bank 1 at the same time that component B accesses memory bank 2, however component B can not access memory bank 1 until component A has completed its' operation. In essence, a queue, or stack has been made for array access. Depending upon final implementation of the memory, this component may be removed.

5.3.11 DMA Controller

In order to encode an image, the image data will need to be taken off the host system and placed in the Firebird memory banks. There are two ways of doing this operation. The first involves taking a 64-bit word, and writing it to a specific address, increasing that address, then retrieving another 64-bit word, writing that to the following address, and so on. This method is simple, however can be very slow. The second method involves Direct Memory Access (DMA). In this case, a starting point is set, and the length of data to be read and written is recorded. Then a direct read and write function is enabled which takes directly from one memory to another, bypassing any buffers or registers. Also with a DMA function, the PCI bus used to interface with the card is dedicated to this transfer, where as before the PCI bus is not dedicated to the operation, it may be used for other operations intermittently. The DMA method is significantly faster, possibly by a factor of ten, however this would have to be tested in detail to be proved.

The firebird card does not have a dedicated DMA controller, so one will have to be constructed. Annapolis Micro Systems has provided different function calls to be used specifically for DMA control, which made it easier to implement a DMA controller inside the Virtex 2000 chip.

5.4 Hardware/Software Interface

In order for the FPGA to operate correctly, several pieces of information must be passed to and from the main system. One of the main pieces of information to be passed to the FPGA is the image data, however this will be stored in external memory until the FPGA specifically request it. Once the image data has been loaded, several identifying bits must be sent from the main system. The number of rows (N_ROWS) and the number of columns (N_COLS) that the image contains must be sent in order for the JPEG-LS encoder to calculate the end of a row, or where the *Ra*, *Rb*, *Rc* and *Rd* pixels are located. The encoder knows that the end of a row has been reached by indexing the pixel it is processing by the number of columns. Likewise, by indexing the number of rows, the end of file can be determined.

The FPGA also needs a signal that indicates that the image data has been loaded and the encoding process needs to start. This will be sent from the host code once the image buffer is full, or the entire image has been loaded into the available memory.

Once the encoding process has been completed, the only information that will be sent out to the host code is a signal signifying that the encoding process is done and the size of the file. The size must be output because the encoding process is adaptive, the final product will not be a standard size, and the host code must recognize this in order to read the correct values from memory. At this point the compressed data can be retrieved from memory by the host code and either sent to long-term storage or streamed to a ground base for analysis.

5.5 Summary

This chapter presented the system design of the JPEG-LS algorithm in VHDL. The system design was broken down into three sections; the Software Design, Hardware Design, and the Hardware/Software interaction sections. Each section was covered the intricacies of the design of the JPEG-LS algorithm being ported to VHDL. The main portion of the Software Design discussed the reason for a need for software in the design of the overall system and how the software is meant to protect the hardware from corruption due to particle interference. The Hardware design presneted each individual component of the JPEG-LS Encoder and how each interacted with the other and how the system as a whole created the compression. The Hardware/Software Interface discussed how the software and hardware

communicated with each other and why there was a need for such communication. The main reason for this communication was the transmission of the image to be encoded and then, once encoded, for the acknowledgement of the size of the compressed image and image transmission by the host code. The overall system design can be clearly seen by understanding each of Hardware, Software, and Hardware/Software Interfaces individual components.

6 Results

6.1 Introduction

In this chapter the results of the testing and functionality of the Firebird board, including the test programs LEDBlink and the DMA controller testing, are explained in detail. This chapter also includes the discussion of the results of Michael Piorun's tests in ModelSim and predictions for the VHDL running on the board. The results of the "Place-and-route" are also to be discussed, which explains the size of the overall design. This design is based on the 8-bit LOCO-I version of JPEG-LS, as Michael Piorun had originally implemented.

6.2 Results

6.2.1 Operational Board

A primary task in being able to make the code functional on the FPGA is to make sure that we had the ability to put the program on the board. Through the use of a tutorial given to us by David Petrick we were able to construct a program that would increment a counter, flash LED's on the Firebird board, and scroll a message on the small display screen also attached to the board. Figure 14 shows a screen shot taken from the LEDBlink program that we designed.

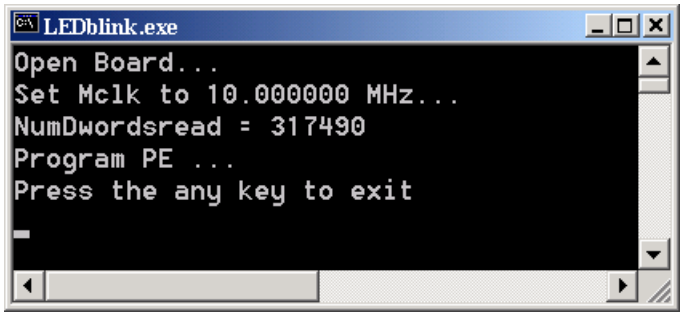

Figure 14: LEDBlink Screen shot

By creating this program, and successfully getting it to run, we also were able to learn how to program and deprogram the Firebird board. This will be important in later exercises. A video of the board has been included in the attached Compact Disc.

6.2.2 DMA control

In order for the image data to be loaded and removed from the Firebird's system memory, a DMA function needed to be written. It is also important that specific information

be sent to the card in a register, indicating when to start the encoding process as well as how many rows and columns the picture contains. A program was designed to take an image from the computer's hard drive, send it to the Firebird board with a DMA request, along with a register file of data. Once the information was on the card, a comparison was done on the register file, and if the information matched, a status bit was sent back to the host program, and the computer read the image file back from the card with another DMA request. This code will be useful once the whole package has been put together, and data must be sent to and received from the card. Figure 15 shows a screen shot from the DMA Test that we created.

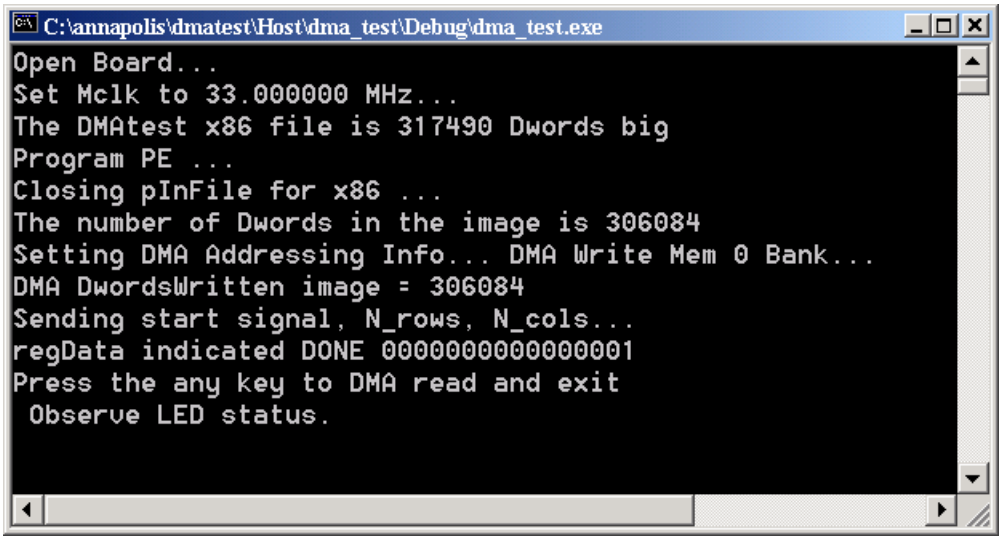

Figure 15: DMA Test Screen Shot

Once the final code has been compiled, and appears to work, it will be placed inside the DMA code, which will take care of the data transfer on and off the card.

6.2.3 Place and Route

Since we obtained Michael Piorun's JPEG-LS code, the relative amount of VHDL that we needed to write was greatly reduced. Theoretically, his implementation should not need to be edited in order to work with our version of ModelSim. Due to errors in library declarations, the code was not able to be compiled correctly by ModelSim. Since this is a perquisite to place and route, running the makefile to produce the *.x86 file was not possible. These errors may simply be due to the differences in compilers used by Michael Piorun and those used in this lab.

6.3 Summary

This chapter presented the results that were gathered throughout the project in accordance to implementing JPEG-LS in VHDL. The most important result that was accomplished was that the Firebird board was operational and that the DMA controller functioned properly. In specific, the LEDblink VHDL that displayed "Hello World" showed that the firebird board functioned properly. The DMA was proven to work properly by a test image being sent onto the memory of the Firebird card and then being successfully transferred out, with the ability to be opened and recognized as an image file.

7 Summary and Conclusion

7.1 Introduction

This chapter presents the overall conclusions of the project. In this chapter the problems that slowed progress, the comparison of results to our goal, and work that still needs to be or could be done are discussed.

7.2 Problems Encountered

7.2.1 Synplicity

Synplicity turned out to be a very useful program once we figured out its subtle formatting requirements. These minor requirements turned out to be fatal errors in ModelSim. The process of importing all the VHDL into a project file was a challenge since we had never used this particular program before. However, having used several project/workspace centric programs we were eventually able to figure it out.

7.2.2 Annapolis Libraries

The next problem we encountered was actually several problems with Annapolis Micro Systems' code. Annapolis Micro Systems provided all the necessary libraries and packages with the Firebird PCI card. These libraries and packages provide access to the controls for the various features on the Firebird board.

The Firebird board actual uses two FPGAs to accomplish a user's design. One is the primary unit on which the user design is implemented. The second is a configurable PCI bridge that is set up at the time of programming for the user's application. This configuration is not accessible by the user directly, but is built by the Annapolis software depending on what parts are used by the user's hardware configuration. The functionality of this unit is dependant on correct library definitions so the compiler can associate all the correct signals. The same dependency exists for all other board functions (memory, expansion card configuration, etc). Our project requires the use of all the memory and its associated direct memory access components. We also need to use the Local Address Data (LAD) bus for register communication back to the program that is executing on the PC.

Some of the necessary components are dependent on other components (e.g. DMA requires the LAD bus and memory bridges). The basic functional units of the Annapolis software would not compile. Upon further investigation we realized that this was a result of numerous errors in the library declarations in the Annapolis VHDL. We determined that many of these errors could be related to older versions still existing on the computer that we

were using or errors that were overlooked by Annapolis. Eliminating all of the errors took a significant amount of time, but eventually yielded a working interface.

7.2.3 ModelSim

ModelSim compiles the files for mapping to the specific device using a macro denoted by the pneumonic "vcom." This file basically sets up the compiler for executing all the necessary operations. ModelSim is an extremely powerful simulation tool, and can be used to simulate the operation of complete designs. This is basically as far as Michael Piorun ever got with his thesis work.

The formatting of this macro was completely new to us. Also, ModelSim is extremely sensitive to exact syntax and library linking. Small problems associated with libraries consumed an entire week of digging through a relatively small amount of code. The Annapolis Micro Systems libraries combined to about 10,000 lines of code. Beyond the 10,000 lines was user design and interfacing. If an error occurred while compiling, a significant amount of time was spent determining whether it was in the user code or the provided libraries. In some instances the same error occurred on multiple lines, therefore taking even more time finding each occurrence of the error.

The "vcom" macro defines user code paths, associated libraries and packages. This must be arranged in the correct order, leading up to the top-level architecture file. Nothing can be repeated, or cross-linked. A large program can present a daunting task here, or at the very least a great amount of frustration.

7.2.4 Makefile

The "make-file" is another macro used for the place and route of a design. The ultimate output from this operation is a *.x86 file to be burned to the card. The changes that are required in this file are minimal compared to the "vcom" macro. However, we still ran into a tremendous amount of trouble here. Everything was in perfect order, but the "make" operation was still exiting prematurely. The problems turned out to be a cross-linked file and the some definitions in the operating system's environmental variables. These problems were due to files from a different FPGA board that had previously been used in the computer we were using still being associated with the previous FPGA.

7.2.5 Host code

The host programming language of choice for the firebird is C. The C code programs the board with the *.x86 file, runs a set of selected tests, and finally closes the board. This is

necessary to use the board. Annapolis Micro Systems provides some sample executables and accompanying *.x86 files, but little documentation.

Upon further investigation we discovered how powerful the host interface could be. Virtually every imaginable resource on the board, from temperature to clock speed, can be monitored and/or controlled. Access to these resources is not well documented, and the functions are already compiled into DLL files that were supplied with the board software. Some documentation is available from Annapolis Micro Systems, but many of the special functions are not explained in detail, or at all. Basic special register access is presented and the variable placement in the supplied functions is explained, but the options supported by various functions are not explained at all.

7.2.6 Host Compiler

Compiling the C code lead us to more difficulties related to libraries. These problems were encountered because the required include files and compiler libraries are not well documented by Annapolis Micro Systems. The various function calls used for communication and control of the board are hidden in DLL files that are not noted. We had to add these files to the compiler's configuration before we could successfully compile the code. Some of the options had to be entered in text switches for the compiler since the options were not presented in the GUI.

7.2.7 Full operation

The process of understanding the operation of the WildStar Firebird FPGA board should not to be underestimated. We started off by trying to get the onboard LEDs to flash using a counter in the FPGA. A printed tutorial was used to help us though the design process. This tutorial was designed to supplement a class on the board and much detail was excluded from the text. Eventually this problem gave way to persistence and we gained an even greater understanding of the board. After this first hurdle had been cleared, we had a great deal of information that helped us to move more quickly though the next implementation.

Next we pieced a DMA controller together from sample code, and Dave Petrick's guidance. Once this module was working the logic of the whole thing became apparent. Control of the board seems relatively simple now, however, special functions still require a lot of research to understand their use.

The task of implementing JPEG-LS on the Firebird FPGA board will be dependant on DMA to and from the board, and register access to and from the board. Temperature and

current monitoring will be extremely useful for testing the efficiency of the algorithm running at higher speeds, but this information may be difficult to acquire.

7.3 Project Performance Analysis

For the purposes of this project we had the goal of implementing JPEG-LS on an FPGA in VHDL. Unfortunately the final goal could not be fully realized. The objectives have been discussed in greater detail in chapter 3 and our progression on the objectives as well as how well these objectives were met is discussed in this section.

In order to implement the JPEG-LS algorithm, we needed to understand mathematical functions associated with the algorithm. This was accomplished by reading books about Golomb coding, prediction and context modeling, Michael Piorun's thesis, and ISO standards as well as various online resources.

The process of making our code functional required first proving the Wildstar card to be in working order. By fixing several of the included libraries supplied by Annapolis Microsystems, as well as by implementing two small programs to test functionality. Once these two programs ran successfully, then the JPEG-LS algorithm was examined in greater detail.

The ultimate goal of implementing the JPEG-LS algorithm in VHDL was partially accomplished by performing the place and route of the JPEG-LS VHDL. Although this is extremely close to the overall goal, the actual VHDL could not be run due to the unfortunate outcome of the place and route. Michael Piorun's VHDL includes several implementations of numerical values that are not a standard part of the VHDL libraries and therefore caused the synthesis tool to optimize out a large portion of the VHDL. Due to time constraints more testing and debugging was not possible, and therefore the problem was not solved.

7.4 Future Work

7.4.1 VHDL

Since the core of this algorithm has already been developed and simulated, the task remains to make it work on an FPGA. The algorithm currently compiles and maps, although the place and route was unable to be run due to problems with the RIT VHDL. Since Michael Piorun implemented both decoder and encoder in his complete algorithm, the task remains to map memories and controls to their proper places.

The original code is very well organized and his thesis gives detailed descriptions of all components. We assume he shared the memory modules between the encoder and decoder for the reason of saving space on the design. Since these modules are accessed

through the top level component in the original design there are three possible mapping schemes that we can see.

First, the memory modules can all be mapped to one bank and accessed whenever needed. Unfortunately, this otherwise simple solution is complicated by the fact that multiple modules need to access the memory banks at the same time. Clocking the onboard RAM up to its maximum 150MHz would limit the algorithm's clock to 30MHz. This would probably not be optimal for the overall performance of the algorithm.

The second option we considered was to allocate a small portion at the top or bottom of each memory bank for direct access at any time. Since five arrays are needed by the algorithm and five arrays exist on the board this would work, but it could interfere with accessing memory to retrieve raw image data and save compressed data. A small buffer for the memory banks associated with reading and writing data to the card via DMA may be needed to allow access to continue during a DMA. It may ultimately be necessary to pause the compression for the duration of any DMA action. Since this is a restriction based on hardware, such a problem could be avoided in a system designed specifically for this application.

As a third option, it is also possible to implement Michael Piorun's original memory array design. The components can be added to the architecture code and linked to the encoder by signals, or they can be added to the encoder itself and linked to their respective signals, thus eliminating the memory IO from the entity of the encoder.

We believe the second option is the best, however timing issues may make this an overly complex task for something that will ultimately not be implemented. However, since the point is proof of concept, the slow alternatives may be contradictory to the purpose of this project.

Controls for the encoder algorithm include a start bit, a done bit, and two words describing the width and height of the image. Since a continuous bank of memory large enough for a typical whole image doesn't exist on the card, memory paging will have to be used. This will mandate extra controls for stopping, indexing, and restarting.

The same problem exists for getting the compressed data off of the card. If the total compressed size of the image is too large for a single memory bank, processing must be stopped for a DMA read to complete, or the result bank must be switched to an empty bank while the DMA read takes place. These actions will need control signals and address lines to notify the software of the full memory bank. This would initiate a read and then a write to refill raw data into the vacant bank.

One area of great concern is the method of handshaking used by the components in Michael Piorun's JPEG-LS encoder. All outputs from the various components are

synchronized with the rising edge of the clock signal, but since many components require variable amounts of time to complete; this was not enough. A clever method of synchronous handshaking and internal signaling was created between components. This is meant to maximize efficiency, allowing components to run as soon as the data is available instead of waiting for it or executing a delay such that the amount of time until completion is always constant. This method worked very well in Michael Piorun's simulation results; however, it may not be so functional in real life. It is possible that the handshaking system could actually limit the maximum speed that the algorithm can run at, thus crippling the implementation. This will need to be thoroughly tested.

7.4.2 Software

The software for communicating with the card is relatively simple. Although there are many things that need to be completed in a specific order to make the card work, it is not a complicated process. The Annapolis Micro Systems "Firebird" board has drivers that allow the programmer to call certain functions to control the card. Just like any other C program, the functions have variables in them and a return code is sent back from the function, indicating a successful operation.

The first operation that must be completed is programming the card. The *.x86 file created by the place and route utility will need to be written to the FPGA on the card. The file must be opened and read into memory, just like any other data that is read in a C program. Next the program calls a function for the Firebird board that sends the data to the card for programming the FPGA.

Now that the card is programmed, the raw image data can be written using DMA. The code has to read in a TIFF image, strip the header off, and output the raw image data to the card. If memory paging is used then a routine will have to monitor the state of the algorithm constantly to determine when to send the next data and to which memory bank.

As the algorithm is running and filling up memory banks with compressed image data, this data will have to be recovered using DMA. The data will need to be compiled and output to system memory. When the compression is done the header should be assembled, and the image can be stored on the hard drive.

There are many things that can be implemented in software relating to the card. For example, obtaining results on how much power is dissipated by timing the compression run and monitoring the card's operating voltage and current draw. These values are available to the programmer though special functions in the Firebird documentation.

References

Discrete Wavelet Transform. *Digital ImageProcessing Documentation*. <http://documents.wolfram.com/applications/digitalimage/UsersGuide/8.6.html> April 18th 2004

Ebrahimi, Touradj and Santa-Cruz, Diego. A Study of JPEG2000 Image Coding Versus Other Standards. <http://jj2000.epfl.ch/jj_publications/papers/004.pdf> Aug 18th 2004

Firebird for PCI. *Annapolis Micro Systems, Inc* <http://www.annapmicro.com/firebird_pci.html> April 22nd 2004

JPEG image compression FAQ, part ½.<http://www.faqs.org/faqs/jpeg-faq/part1/> April 18th 2004

The JPEG2000 Resource Web Page. *Video Image Processing Laboratory*. <http://stargate.ecn.purdue.edu/~ips/tutorials/j2k/> April 18th 2004

LANDSAT 7. *National Aeronautics and Space Administration.* <http://landsat.gsfc.nasa.gov/> March 17th 2004.

Marcellin, Michael W. and others. An Overview of JPEG-2000. *Proc. Of IEEE Data Compression Conference*. <http://rii.ricoh.com/~gormish/pdf/dcc2000_jpeg2000_note.pdf> April 18th 2004

Memory Solution Center – ZBT RAM. *ALTERA*. <http://www.altera.com/technology/memory/sram/zbt/mem-zbt.html> April 22nd 2004

Mission Statement. *Goddard Space Flight Center.* <http://www.gsfc.nasa.gov/about_mission.html#content> April 21st 2004

NASA – About NASA. *NASA*. <http://www.nasa.gov/about/highlights/index.html> April 21st 2004

Piorun, Michael. Hardware Implementation of a JPEG-LS Codec. Rochester Institute of Technology – Computer Engineering. September 2001.

Rane, Shantanu D. Evaluation of JPEG-LS, the New Lossless and Controlled-Lossy Still Image Compression Standard, for Compression of High-Resolution Elevation Data . <http://www.stanford.edu/~srane/sdr_pubs_files/jpeglspaper.pdf> Aug 20th 2004

Santa-Cruz, Diego; Ebrahimi, Touradj; Askelöf, J; Larsson, M; Christopoulos, C. A. JPEG 2000 still image coding versus other standards. <http://jj2000.epfl.ch/jj_publications/papers/003.pdf> Aug 18th 2004.

Savakis, Andreas. Benchmarking and Hardware Implementation of JPEG-LS. http://www.ce.rit.edu/~savakis/papers/ICIP02_savakis_piorun.pdf. Aug 20th 2004.

Stardust® Software. http://www.stardustsoftware.com/products/imgenc3/ April 20th 2004.

TIFF Image File Format, The. Cooper Union Electrical and Computer Engineering. <http://www.ee.cooper.edu/courses/course_pages/past_courses/EE458/TIFF/> Aug 20th 2004.

TIFF Revision 6.0. *Adobe Developers Association.* <http://partners.adobe.com/asn/developer/pdfs/tn/TIFF6.pdf>. Aug 20th 2004.

USGS Landsat Project. *United States Geological Survey*. < http://landsat.gsfc.nasa.gov/ >. March 24th 2004.

Virtex-II Platform FPGA Features. *Xilinx® Home.* <http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=v2_features#> April 22nd 2004.

VirtexTM-II Q&A. http://www.xilinx.com/products/virtex2/v2qa.pdf April 22nd 2004.

Weinberger, Marcelo J. Seroussi, Gadiel. and Sapiro, Guillermo. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. http://www.hpl.hp.com/loco/dcc96copy.pdf. Aug 18th 2004.