UNLV | UNIVERSITY LIBRARIES

Honors College Theses

Honors College

4-2018

Efficient Image Coding and Transmission in Deep Space Communication

Reiner Dizon
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/honors_theses

Part of the Electrical and Computer Engineering Commons

Repository Citation
Dizon, Reiner, "Efficient Image Coding and Transmission in Deep Space Communication" (2018). *Honors College Theses*. 31.
https://digitalscholarship.unlv.edu/honors_theses/31

This Honors Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Honors Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Honors Thesis has been accepted for inclusion in Honors College Theses by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

EFFICIENT IMAGE CODING AND TRANSMISSION
IN DEEP SPACE COMMUNICATION

By

Reiner Dizon

Honors Thesis submitted in partial fulfillment

for the designation of Research and Creative Honors

Department of Electrical and Computer Engineering

Advisor: Dr. Emma Regentova

Committee Members: Dr. Andrew Hanson, Dr. Venkatesan Muthukumar

College of Engineering

University of Nevada, Las Vegas

April, 2018

Table of Contents

Abstract 2
I. INTRODUCTION 3
II. BACKGROUND 5
A. Data Compression and Source Coding 5
B. Error Resilience and Deep Space Communication Issues 6
C. Hardware Limitations 8
D. Huffman Coding 9
E. Rice Coding 10
F. Exponential-Golomb (Exp-Golomb) Coding 11
G. Channel Coding and Tradeoff with Source Coding 12
H. Field Programmable Gate Array 14
III. METHODS 15
A. Images and Buffer 15
B. Huffman Encoder Implementation 16
C. Rice Encoder Implementation 20
D. Exp-Golomb Encoder Implementation 24
E. Software Implementation and Introduction of Errors 25
F. Design Components and Power Usage 26
IV. RESULTS AND DISCUSSION 27
A. Compression Ratio 27
B. Error Resilience 30
C. Resource Utilization 33
D. Power Consumption 34
Discussion 34
Acknowledgement 40
REFERENCES 41
APPENDIX 46

Abstract

The usefulness of modern digital communication comes from ensuring the data from a source arrives to its destination quickly and correctly. To meet these demands, communication protocols employ data compression and error detection/correction to ensure compactness and accuracy of the data, especially for critical scientific data which requires the use of lossless compression. For example, in deep space communication, information received from satellites to ground stations on Earth come in huge volumes captured with high precision and resolution by space mission instruments, such as Hubble Space Telescope (HST). On-board implementation of communication protocols poses numerous constraints and demands on the high performance given the criticality of data and a high cost of a space mission, including data values. The objectives of this study are to determine which data compression techniques yields the a) minimum data volumes, b) most error resilience, and c) utilize the least amount and power of hardware resources. For this study, a Field Programmable Gate Array (FPGA) will serve as the main component for building the circuitry for each source coding technique. Furthermore, errors are induced based on studies of reported errors rates in deep space communication channels to test for error resilience. Finally, the calculation of resource utilization of the source encoder determines the power and computational usage. Based on the analysis of the error resilience and the characteristics of errors, the requirements to the channel coding are formulated.

Keywords—

Source Coding

Error Resilience

Channel Coding

Deep Space Communication

I. INTRODUCTION

One of the most important feature of today's digital devices is their ability to communicate to and from other devices with the correct data. Digital communication enables two or more devices, such as computers and phones, to exchange information in the form of bits (1's and 0's) over a medium, such as wires, fiber optic cables, and air. A device that sends data is the source or the transmitter, and the destination or the receiver obtains that data. To exchange data between these two devices, the channel connects these devices and deliver information from one device to another [1]. Since digital communication relates to data transfer, its efficiency also comes from ensuring data from a source arrives to its destination quickly and correctly. In other words, the receiver must obtain the data in a reasonable amount of time, and when processed, the received data is an exact copy of the transmitted data [1]. This growing need for rapid and accurate communication relies upon efficient and reliable data transmission. To ensure the efficient use of channel bandwidth and send data at the maximum rates, the data acquired for scientific instruments/sensors are encoded to ensure their compression to the smallest possible volume. Data transmission usually takes place over several relays or interconnection circuits as well as through many heterogeneous physical channels which can cause corruption or loss of data due to the limitation of storing or buffering all the data or due to the noise in the physical channel. Therefore, in data and communication networks, protocols prescribe verification of data and acknowledgment from the receiving end which, if needed, can request for retransmission.

Depending on the application and the criticality of the data, certain algorithms can tolerate a partial loss of information or transmission errors in varying degrees. For example, when a user watches a video on a website, this online video is only a fraction of the size of the original raw video due to algorithm used to compress it, but it still resembles the original video

after losing a significant amount of information. The human visual system and intelligence can "interpolate" data and extract information even from a low-quality video [2]. For the everyday communication, such as in the previous example, this partial data loss is acceptable for streaming data over multiplexed channel, used for several simultaneous transmissions. Therefore, so called lossy compression methods are widely employed, such as JPEG and MPEG compression standards [1]. However, this tolerance for transmission error is especially narrow for critical and costly scientific experiments, a slight alteration of the transmitted information can result in either a loss of precision or in the complete loss of the original data. One prominent example of this is deep space communication.

In deep space communication, information received from satellites to ground stations on Earth come in huge volumes, such as hyperspectral and multispectral data of planets and images of deep sky, captured with a high precision and resolution equipment, such as telescope cameras or spectrometers. Analog-to-digital converters (ADC) turn the raw inputs from these components into digital information which the satellites will transmit back to Earth [3]. Often, these deep space satellites "never return to Earth," so the preparation for these missions is paramount which is costly to space agencies, such as NASA [4]. At a high cost for each mission, every received data from these satellites are of an utter importance. Although, in modern digital communication, the probability of error, as, for example, alteration of bits (0 to 1 and vv, or an erasure) is rather low, the deep space communication may experience noise from photons and cosmic rays, electromagnetic interference which can compromise the accuracy of representation of valuable scientific or control data [5]. Although the communication protocols include means for checking the accuracy of data by providing a control checksum, a.k.a. Cyclic Redundancy Codes (CRC), it does not completely cover all possible errors that can occur [6]. Furthermore, retransmission due

to inaccurate receipt of data is accompanied with a retransmission of data. Given a length of the satellite link and buffer limitations which could be a payload constraint, these retransmissions lead to inefficient bandwidth and on-board storage usage [5]. To avoid retransmission and to secure fast delivery of volumes of critical data accurately, on-board of satellites communication protocols employ lossless data compression called *source coding* and add mathematically formulated and accordingly derived redundancy for error detection or/and correction that is called *channel coding*. Various algorithms have been proposed for lossless and near-lossless data compression. In this work, we implement and analyze a standard method developed by Consultative Committee for Space Data Systems (CCSDS), specifically CCSDS 121.0-B-1 method for lossless generic data compression and test it for channel error resilience [7]. Furthermore, we will evaluate it further in terms of complexity, power usage and compare to other lossless compression methods.

II. BACKGROUND

A. Data Compression and Source Coding

Data compression distill information into its most compact representation which means it reduces the number of bits to represent the original digital data. There are two types of data compression—lossy and lossless—which corresponds to the information content preserved in data [2]. Lossy compression algorithms reduce the original information past the point at which the compressed data cannot be reconstructed back to its original after decoding such data. Under certain applications, these compression codes do not result in the complete loss of the original data especially when they become slightly altered or they do not contain critical information [8]. The earlier example of the compressed video at the viewing end uses a lossy compression

algorithm to reduce the video size, specifically its resolution, for quick data transmission. Some of the most prevalent application of lossy compression is with storage of multimedia data, such as images, videos, and music [2]. However, this is not suitable for scientific data as the incurred loss would affect their precision. Lossless compression algorithms, however, keeps the original information intact after they reduce the data down to entropy, or its most fundamental bits. This type of compression allows the decoder of a receiver to reconstruct losslessly compressed data back to its original state [9]. Because of this property, this type of compression is appropriate for NASA space science missions exploring deep space.

Using data compression, the goal of source coding is to produce data from the source with the minimum number of bits. In this study, the focus is on lossless source coding, also called entropy coding, with the aim of delivering "the digital sequence... with the shortest sequence of symbols... [that] guarantee the perfect reconstruction of initial sequence" [1]. The "initial sequence" in this context denotes the original digital information with "symbols" represented by a single bit or by a sequence of bits, called codewords. In deep space communication, the use of entropy coding not only maintains the precision of their data but also reduces their memory storage needs before transmission.

B. Error Resilience and Deep Space Communication Issues

Before discussing the source coding algorithms in this study, the next sections will lay out the criteria for comparing these algorithms in the context of deep space communication. One of them is the error resilience of the produced code. Error resilience describes how compressed data keeps its original content when an error occurs in the communication channel. One of the models for error-prone channel is the Binary Symmetric Channel (BSC) which assumes that the

bits can be flipped from 1 to 0 or vice versa with a small probability [10]. There could occur also a burst of errors, that is the destination receives several contiguous bits in error due to multiple bit-flips errors or changing the information randomly [10]. When the compression or coding algorithm assigns an equal number of bits to every symbol generated by the source, the "damage" caused by such "noisy" communication is confined within a single codeword or just a few of them [11]. However, high efficiency compression is attained mostly by a variable length coding, and thus any error in decoding can propagate far causing a long sequence to be decoded incorrectly.

Because this study focuses on deep space communication, there are certain errors encountered in the data storage and during data transmission between deep space satellites and ground stations on Earth. Despite reinforcements to protect circuits, some of the components on these satellites can lack certain protections from energetic electromagnetic waves, such as gamma rays, or single events like photon hits which alter the information through physically changing the bits [12]. Because of this effect, the affected data become subject to bit-flip and burst errors which is unavoidable before transmission to ground stations. Also, like most communication channels, transmission from deep space satellites are also susceptible to environmental noise because of their physical location [5]. Noises include interference and electromagnetic effects on the transmitted signal, causing even more errors before the data arrives at the ground station. In satellite communication, radio frequencies have a typical bit error rate between 5×10^{-3} to 1×10^{-7} [13]. Because of these issues, the information may become irrecoverable which will affect its decoding at ground stations on Earth invoking retransmission assuming the error occurred in transmission. Additionally, retransmission assumes the preservation of data in the buffer assuming the receiver which confirms the delivery of unaltered

data packets demands on the memory resource. Testing for the error resilience of some source coding algorithms will offer insight on how much data is recoverable after transmission and how much these algorithms demand on correction and what is the need for data retransmission.

C. Hardware Limitations

Finding some efficient, error-resilient source coding algorithms is a great start, but if it uses onboard resources intensively, its implementation is unlikely on the actual hardware with limited resources. This section will discuss some of the hardware specifications and limitations of some deep space satellites with imaging capabilities. Some of the most important hardware on these satellites are the hyperspectral imaging instruments or the cameras which capture pictures of the deep sky [14]. The imagers use high precision analog-to-digital converters (ADCs) to convert the raw analog data from the imagers into digital information, known as pixels. Along with these instruments, there are other onboard sensors on these satellites which measure a variety of data, such as position and temperature [8]. To process all these data, the onboard processors, which all processes and devices must share, handle this task and stores the processed data into a fixed size for the buffers before transmission. Also, some satellites have redundant circuits of the same type, meaning engineers create one or more copies of the same circuit. For example, in deep space which is a “high-radiation environment,” Triple modular redundancy, or TMR, attempts to alleviate the issue of one circuit breaking down with the implementing two more of the same circuit and verifying results from at least two circuits [12]. Because of these limitations and redundancies, satellites cannot afford to use a resource intensive algorithm which can hog the satellite’s computational resources and should be reproducible in the same system.

D. Huffman Coding

Huffman devised an optimum source coding algorithm that gives a “minimum average number of bit per symbol” based on the use of probabilities of occurrences of those symbols [15]. With applications to images, the symbols in this context are the levels of gray intensity or color. The algorithm assigns shorter codewords (length in bits) for symbols that occur more frequently and longer ones for less probable symbols. In this fashion, the average length of the code is closest to the entropy value where entropy is the lower bound for average length, and entropy plus one is its upper bound [2]. Hence, this algorithm produces variable length codes which are instantaneous parseable prefix code. This code means that as soon as the destination receives the last bit of a codeword, the decoder can map the code into its corresponding value. This fast decoding is possible since none of the codewords are prefix to another. Nevertheless, the codeword lengths have a minimum standard deviation that is an important property from a practical point of view which allows for a manageable buffer [2]. However, the major drawback of Huffman coding in practical application is the need to obtain the statistic of the source first, and the algorithm devises an optimum code for the data afterwards. Then, the receiver gets the coding table, so it knows how to decode the received sequence. Therefore, with known pros and cons of these algorithms, many others sprang up using the property of Huffman’s algorithm. These algorithms include Adaptive Huffman and Reversible Variable Length Codes which extends the Huffman algorithm to increase its performance [2], [16]. Non-uniformity of lengths exacerbates the application difficulties: first, data packing introduces an additional effort, and due to the nature of decoding of such codes, errors can propagate through the course, that is decoding would lead to altered data. According to Lelewer and Hirschberg’s paper on the analysis of various data compression algorithm, they observed the “self-correcting” nature of

Huffman codes, meaning propagation of transmission does not extend for too long [17]. This observation assumes the use of static version of Huffman's algorithm and quick resynchronization between transmitter and receiver, so the algorithm itself is not immune to transmission error, but it can self-synchronize to a certain extent. However, many applications use the adaptive Huffman code since it does not need the entire data and probabilities in designing the code. This paper also remarked on the effect of error in adaptive codes because they saw no evidence to suggest "adaptive methods are self-synchronizing" and the lack of attention in this research area [17].

E. Rice Coding

Robert F. Rice published an extension to the Huffman algorithm in his 1979 report to further improve upon the source coding algorithm [18]. The Rice algorithm coder uses two discrete parts: "pre-processor [with a] symbol mapper" and "adaptive symbol coding" [19]. After converting analog signals to their digital form, they enter the pre-processor block which find the "difference between adjacent data" and to then "map all difference values" into a new "sequence of... symbols" called blocks which becomes the input to the next functional block. The adaptive symbol coding or "variable length coder" uses different options of coding for specific level of "source entropy" [19]. Most of the coding options in this algorithm uses the principle of Golomb code of "the larger an integer, the lower its probability of occurrence," but these options are characterized as special types of "adaptive Golomb code" [1]. The output of this coder becomes transmitted to the receiver which has similar functional blocks to decode the incoming compressed data. Pen-Shu Yeh from Goddard Space Center lays out the algorithm for each of these options in two of her reports along with rest of Rice coding algorithm [19], [20]. This

algorithm has become then a standard by Consultative Committee for Space Data Systems (CCSDS), specifically CCSDS 121.0-B-1 for lossless generic data compression [7]. What makes the Rice coding algorithm a type of Huffman coding is the equivalence of its "variable length codes" to Huffman codes [20]. According to the method data are packed such that the description of the coding mode is incorporated into the header field, and thus any error affecting those bits would lead to the complete loss of the fixed length sequence. However, it suggests a fixed maximum error propagation by design.

In their 2008 paper, James Meany and Christopher Martens studied the error resilience of split-field source coding algorithm which includes Rice coding [21]. For their experimental study, they included a "wavelet transform" module to convert an image to a series of coefficients that corresponds to the transform before performing the compression algorithm. They observed that the "utility of split field coding... depends... on the proportion of suffix bits in the compressed" data [21]. The suffix, generated through the variable length encoder, may therefore be less susceptible to transmission error, especially those that have fixed length, which resembles the Rice coding property.

F. Exponential-Golomb (Exp-Golomb) Coding

Another type of Golomb codes was proposed by Jukka Teuhola in his 1978 journal called Exponential-Golomb coding [22]. As the algorithm's name points out, the generated codeword of this type of coding grows exponentially based on the value of the original datum which characterizes this algorithm as a type of Golomb code because larger values produce longer codewords [1]. Each codeword has three parts: padded zeroes, a separator '1' bit, and the remaining information [23]. The number of '0' bits for the first part of the codeword is

calculated based on the minimum number of bits to represent the original data plus one. The number of bits of the remaining information (original data plus one) is also the same number of the ‘0’ bits in the first part. With these codeword generation steps, the hardware implementation is simpler to design than that of Huffman or Rice coding. There is no need to accumulate statistical data on an image as in Huffman or to run multiple options to find the best codeword as in Rice coding. This ease of hardware implementation makes this a good candidate for study.

G. Channel Coding and Tradeoff with Source Coding

Channel coding algorithms add data redundancy to the data, either compressed or not, to detect or correct transmission errors [1]. This redundancy allows for error detection at the receiver which helps the receiver decide if there is corruption of the received data before processing them further. For example, a simple parity check code appends an extra bit at the end of the digital data to indicate whether the number of 1’s in the data are even or odd [24]. When the destination receives data with a parity check bit, the receiver will first count the number of 1’s in the data excluding the parity bit, generates the appropriate bit based on the parity check, and checks it against that last bit. When the bits do not match that indicates an error in the received data. More sophisticated and efficient codes have been developed from Hamming codes to cyclic, convolutional, Raptor, trellis, low density parity check, etc. [1], [25] With a higher redundancy the codes are able also to correct errors, however the trade-off between compressing source coding and redundant channel coding limits the usage of correcting codes. Many networking protocols use so called Cyclic Redundancy Check (CRC) bits which are calculated using special polynomials and append them to the bare data (compressed or uncompressed) [6]. At the receiver, the same algorithm is used to re-calculate the CRC and compare to one sent

along with the data. The mismatch indicates that there could be an error in either data or CRC. The network protocols can either send a negative acknowledgement, as in X.25, or remain silent, i.e. no acknowledge is sent which after some timeout period (ARQ protocol) is an indication that data were either not received or received in error (TCP/IP) [26]. In either protocols the transmitter would retransmit data.

Evidently, the correcting codes are preferable. A simplest example is a repetition code. Each bit of the codeword is duplicated k times. [13]. This property uses the idea of Hamming distance, or how many bits are different between any "good" code and the received code to determine what bit did the transmitter intended to send over in the presence of error [27]. If the parameter $k = 3$, then a single bit error can be corrected (a minimum number). The receiver will look for the minimum Hamming distance to correct for a one-bit error. Thus, whenever the received code is '000' or '111', the receiver can easily determine that no error was present in the packet. Otherwise, for example if '010' was received, then the error corrected bit is '0'. This code is highly redundant, but explain the main idea of introducing a distance between codewords for attaining the error correcting capability. For detecting e number of errors, the distance is at least e+1, and for correcting c errors, the distance is 2c+1 [13].

In deep space communication, adding redundancy using channel coding algorithms will increase the data transmission time. However, channel coding may eliminate the need for retransmissions because they can correct for transmission error at the receiver. Because of the given tradeoffs, self-synchronization of source codes and their error-resilience is of a paramount importance.

H. Field Programmable Gate Array

Field Programmable Gate Array, or FPGA, is an integrated circuit which enables the end user to configure its logic blocks through various means, such as block diagrams and hardware description languages (HDL) [28]. Without the need to reconfigure logic circuits physically using this device, FPGAs are flexible circuits for most hardware implementation needs as they allow hardware designers to test out their circuit design before the actual implementation in its own dedicated circuit. Therefore, there is no need to find Application-Specific Integrated Circuits (ASICs) to test for different applications [29]. Rather, the user can configure a single FPGA for multiple applications and debug them in the board as well. Furthermore, in recent years, the consideration of using FPGAs for space applications garnered "great interest" especially because FPGAs allow for testing issues, including "harsh environments," in a relatively safe manner [29]. In this study, an FPGA will mimic the hardware of satellites in deep space, along with its computational and power limitations because there is no access to the actual hardware. Without the need to test these source coding algorithms in an actual high radiation environment, data manipulation on the FPGA's memory that hold data will emulate transmission errors. Circuits made for outer space are fabricated using "radiation hardening" technology. That is based on electronic components and systems that are resistant to ionizing radiation which can cause malfunctioning or even damage.

III. METHODS

A. Images and Buffer

First, the data by source is two-dimensional optical image data obtained by CCD camera. Due to its flexibility and programmability features, the hardware implementation of each source coding algorithms is done on the FPGA. To test the compression algorithms, the storage of the images was a necessary component for testing and circuit operating. Also, to operate on a single clock, the buffer memory is likely to be on FPGA representing a static memory (SRAM). For storing these images, a buffer was created in the FPGA enough to store a single picture. The images used in this study are in grayscale for ease of the algorithms' calculation in the FPGA and are characterized as either public domain or astronomical images.

Typical public domain images are 8-bit which means there are only 2^8 or 256 representation levels for grayscale values. There are standard images that was primarily used for testing 8-bit grayscale images [30]. To test these benchmark images in 16-bit mode, their image histograms are stretched from 256 possible grayscale values to 65,536 grayscale using MATLAB. This modification is necessary for a fair comparison among all the 16-bit images. For this study, five of the benchmark images were used which all have dimensions of 256 by 256.

On the other hand, astronomical images that comes from deep space satellites have more precision and often have larger representation levels than public domain images. They are often stored in a special format called FITS (Flexible Image Transport System), which supports 8-bit, 16-bit, 32-bit, and 64-bit integer and floating-point values [31]. The astronomical image used for this study is 16-bit, which have dimensions of 1024 by 1024. For most of the error resilience study, the original 16-bit image was used, but for Huffman study, the 8-bit version of the image

was used instead due to the software limitations of that algorithm. Each 16-bit value was mapped to an 8-bit value using the following formula:

$$\text{8-bit value} = \frac{\text{16-bit value}}{65{,}536} \cdot 256$$

B. Huffman Encoder Implementation

The hardware implementation of the Huffman encoder circuit consists of four major sub-modules. Because the statistics of the image are needed before generating the codeword, a histogram circuit is needed to collect the frequency or counts of each pixel value that signifies the probability of these values needed for the codeword generation. There are different circuits for the histogram, but a two-port memory circuit was used to store these frequency counts. The number of entries in this memory circuit is equal to the number of grayscale levels, where each entry can hold up to the maximum count for the image size. Therefore, each pixel data is the input to this circuit and addresses an entry in this memory circuit. Figure 1 shows the logical view of this histogram circuit with a sample 8-bit image and a portion of the memory.

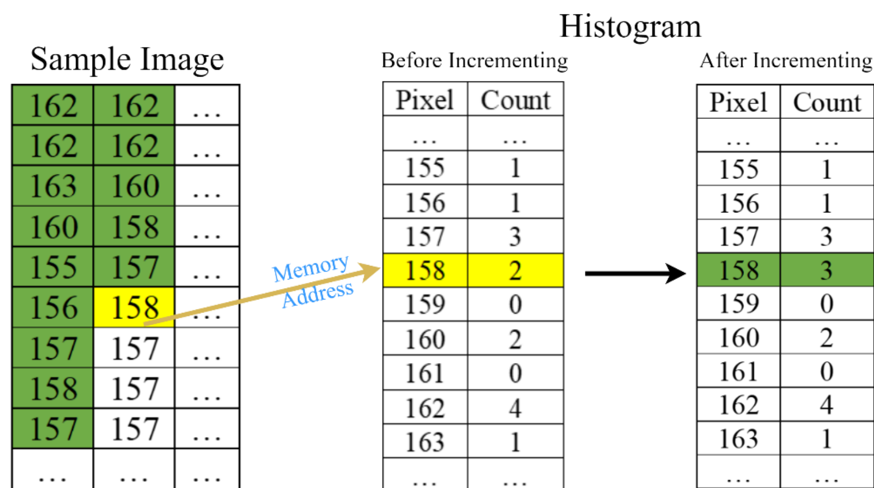

Figure 1. Logical View of the Histogram Circuit

After collecting the frequency of each pixel value, another circuit sorts these counts in descending order where the least probable count is placed in the last entry of the sorted histogram. The hardware implementation, introduced in the paper by Shengan Dong *et al.*, was used for this part of the encoder due to its quick parallel sorting algorithm [32]. Their circuit consists of multiple sub-modules of comparators and D flip-flops, which shares the same datum input as well as enable and load signals. The idea of this algorithm is to store the current value from the datum input to a specific D flip-flop if the flip-flop's content is less than or equal to the value, which gets cascaded to the next sub-module. Therefore, the values are fed to the circuit serially or one-by-one until all the values have been sorted. While the data (or, in this application, frequencies) are being sorted, the addresses (or pixel values) are also sorted using the same signals from the data sorting scheme. Figure 2 illustrates the components of the sorting circuit for both data and addresses, which comes from their paper [32].

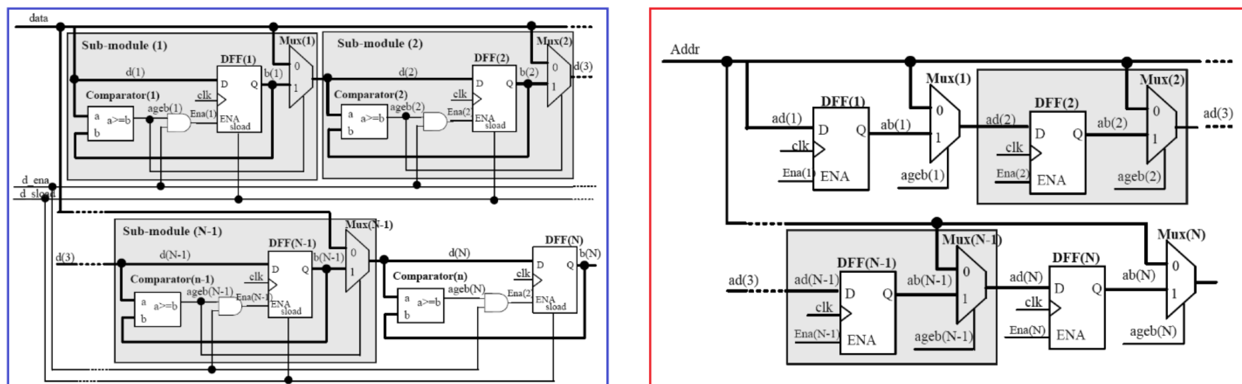

Figure 2. Sorting Circuit for Data and Addresses [32]

When the frequencies and their corresponding pixel values are finally sorted in descending order, the length of each codeword is generated based on how probable a specific pixel value is in the image histogram. The Huffman algorithm assigns a shorter codeword for those values with the larger probability or, for this implementation, count. With this principle in mind, the next module for this Huffman circuit is a finite state machine (FSM) which generates

the length of each codeword for each pixel value. This FSM consists of 5 states. The first one (START) loads the sorted counts into a buffer within this sub-module along with registers to keep track of their lengths, and their corresponding pixel values are maintained using a one-hot state registers. The sizes of these registers equal to the number of different pixel values in an image, and only one bit in each register is set to '1' initially before the next state. Figure 3 demonstrates the implementation of these registers.

One-Hot State Registers

Pixel	0	1	2	3	...	253	254	255
0	1	0	0	0	...	0	0	0
1	0	1	0	0	...	0	0	0
2	0	0	1	0	...	0	0	0
...					...			
...					...			
254	0	0	0	0	...	0	1	0
255	0	0	0	0	...	0	0	1

Figure 3. Memory View of One-Hot State Implementation

The COMBINE state combines the last two non-zero entries (least probable counts) of this buffer. This operation consists of performing OR operation between the one-hot registers and placing the result into the most probable entry among these two. The corresponding entry for the least probable pixel is cleared out. For example, after the first iteration, the last entry of the buffer is empty, and the one-hot register above it now has two '1's in its register. Afterwards, the UPDATE state increments the lengths of the pixel values that were combined in the previous state. The RE-SORT state then sorts the buffer using a nonstable sort which overrides the order of the pixel values, which means the combined pixel values' register is placed higher in the sorted buffer whenever two counts are equal. These three previous states are repeated until all symbols (or pixel values) are processed. Afterwards, the DONE state sorts the final lengths

buffer using bubble sort to maintain the order of the pixel values before the final codeword generation circuit. Figure 4 provides a summary of these states for this state machine.

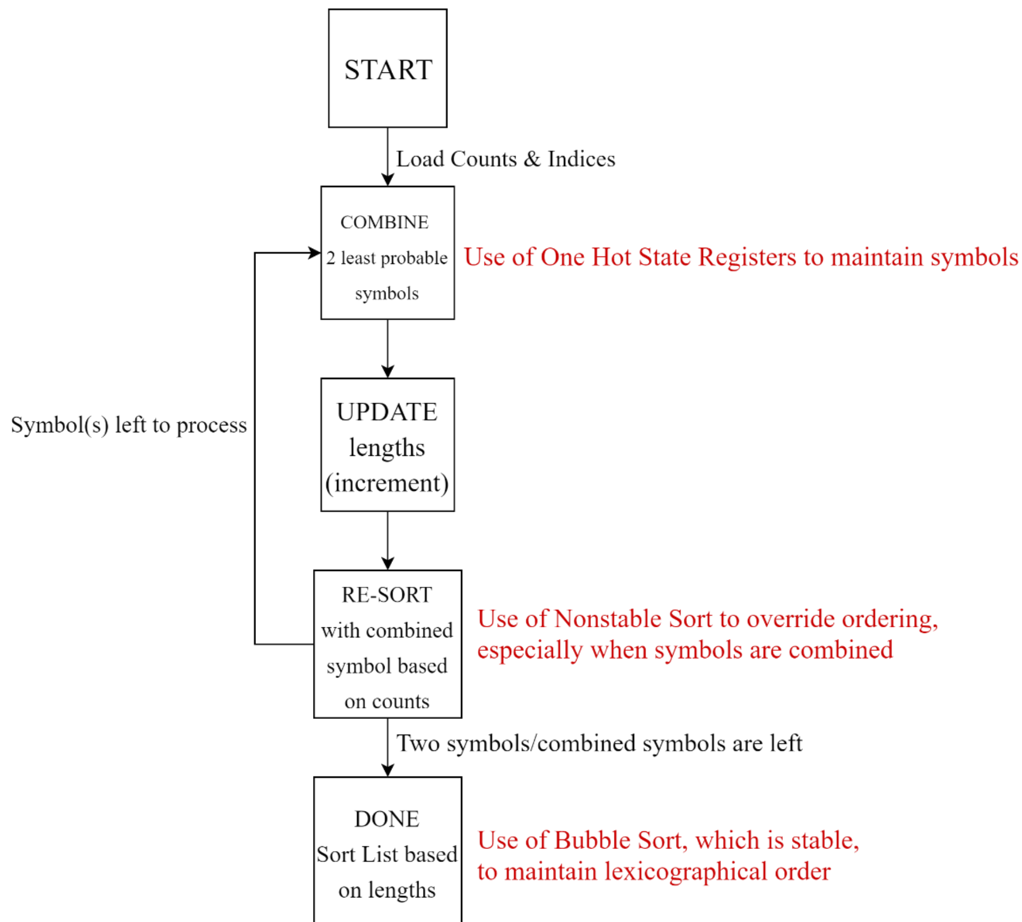

Figure 4. FSM of Length Calculation circuit

The last sub-module of this hardware implementation of the Huffman algorithm generates the final codeword for each pixel value using the calculated lengths from the previous circuit. These codewords are constructed using a canonical tree which ensures that each codeword is easily decodable. With this canonical tree, its maximum depth is equal to the largest calculated length. For the purposes of this study, this maximum depth is assigned to a constant which is appropriate for the benchmark images used in this study. Initially, a full binary tree is loaded into registers where each register contains one level of the tree. As the lengths are read, each register of the tree shifts out a codeword of different sizes, but the appropriate one comes

from the level number equal to that length. These codewords are then stored into a separate buffer for the actual encoding. Figure 5 illustrates the high-level picture of the Huffman encoder circuit. Afterwards, the image is then encoded using this buffer as the look-up table (LUT) before transmission to the receiver.

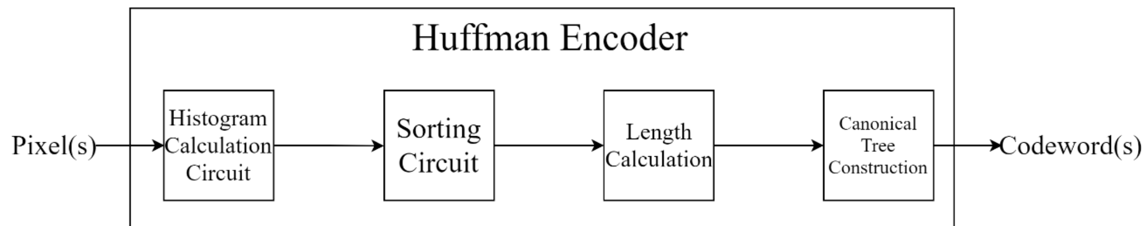

Figure 5. High-Level Schematic of the Huffman Encoder

C. Rice Encoder Implementation

For implementing the Rice encoder, the standard from CCSDS 121.0-B-1: Lossless Data Compression provided the explanation of the all the necessary pieces of circuitry to build the encoder [7]. This standard works on the data in groups called blocks (parameter *J* in the standard), which its size can be set to 8, 16, 32, or 64. For the hardware implementation, only block sizes of 8 and 16 were considered for this study. Along with the size of each pixel datum (parameter *n* in the standard), block size is an important parameter in building the Rice encoder, which dictated the need for parameterized modules when constructing the different configurations of the circuit. These parameterized modules allow for the implementation of any permissible configuration of the Rice parameters. To implement the other configurations, this same circuit is used, but the parameter variables can be changed which exhibits reusability of the code. As mentioned previously, the Rice encoder has two major functional parts: Pre-processor and Adaptive Entropy Coder—as illustrated in Figure 6.

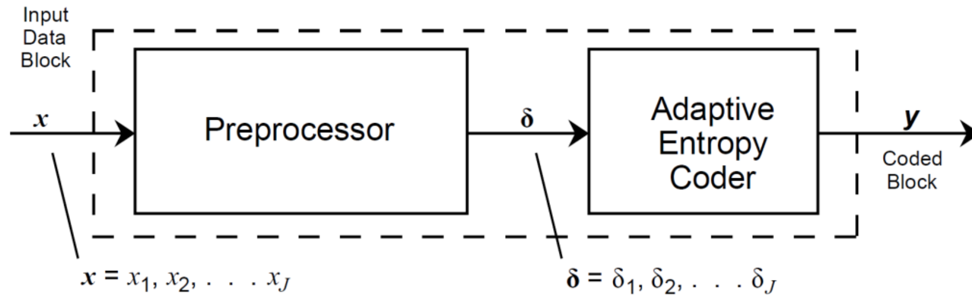

Figure 6. Main Functional Parts of Rice Encoder [7]

The Pre-processor module decorrelates the pixel data and maps only the difference between pixel data within a given block of them. The first pixel on any given block is transmitted in its entirety and not compressed because all other bits in the packet will rely on this reference. Because the pre-processor takes the “difference between adjacent data” (denoted as $\Delta_i = x_i - x_i$), a small buffer was necessary to store the current and previous pixel values [7]. This pre-processor buffer takes in the pixel from the image buffer and pushes the pixel datum into internal shift registers, shifting out the stored previous value. Both differences between the current and previous pixels are calculated using subtractors, and the most significant bit of the difference determines the sign. After obtaining the magnitude and the sign of the difference, it is then mapped using the function below, so the result becomes associated with a positive value.

$$\delta_i = \begin{cases} 2\Delta_i & 0 \le \Delta_i \le \theta \\ 2|\Delta_i| - 1 & -\theta \le \Delta_i \le 0 \\ \theta + |\Delta_i| & otherwise \end{cases}$$

where $\theta = \min\left(x_i - x_{\min}, x_{\max} - x_i\right)$

The mapping function is implemented using comparators and shift multiplier circuits because the calculation involves multiplication by two. Afterwards, this mapped difference (denoted as δ_i) is sent to the Adaptive Entropy Coder.

The Adaptive Entropy Coder consists of multiple sub-modules that all generate codewords for transmission, which all run in parallel. These sub-modules are a “set of code options” where each have their own way of compressing the mapped differences and packing them along with the reference value and an identification for the option used [7]. The selection among these options is based on whether the given block is all zeroes (Zero Block). If this is not the situation, then one of the remaining options with the lowest codeword is selected. Otherwise, the reference and mapped differences are sent without any compression. This code selection signals are implemented using comparators on the packet size generated by each option and a flag register for the Zero Block option. These signals control a final multiplexer that selects the codeword from the chosen option. Figure 7 summarizes the inner circuitry of the Adaptive Encoder with the code selection.

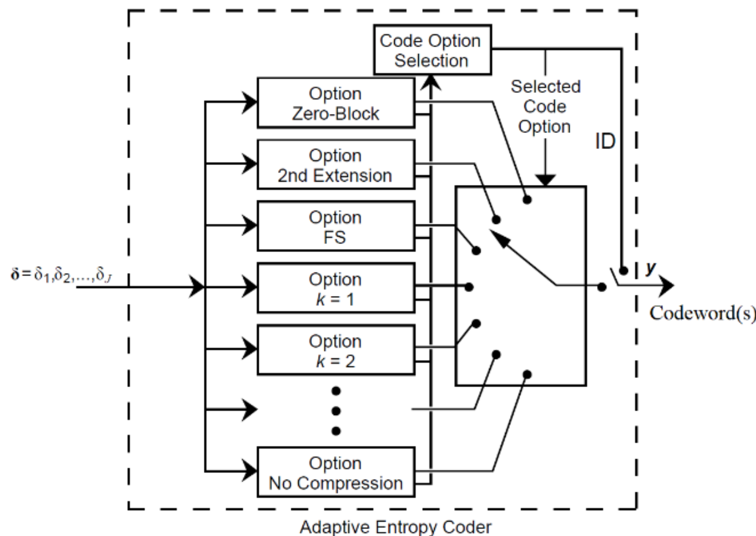

Figure 7. Main Functional Blocks of Rice Adaptive Encoder [7]

Most of these options base their coding scheme on the Fundamental Sequence (FS), including the Split-Sample options. A FS codeword composes of ‘0’ bits, which is equal to the value to be encoded, followed by a ‘1’ bit. For example, the value of 3 has a codeword of “0001.” The main component responsible for creating a FS codeword is a variable left shifter,

which shifts in a '1' bit into the appropriate place in the final packet based on the calculated shift amount. This shift amount is related to the mapped difference, which is the same value for FS and the most significant bits of the difference for the Split-Sample options. Therefore, for all split-sample options, one parameterized module was designed with the variable left shifter and was reused with changes to the parameter in the overall Adaptive Entropy Coder circuit.

Two of the options are of low entropy: Zero Block and Second Extension, which takes advantage of the smallest differences among pixel values in a block. These options also use the FS coding scheme and thus have the variable left shifter in their circuits. The Zero Block option also employs two comparators: one to check if all the mapped differences in a block are all zeroes and another to check if the current reference value is the same as the previous reference value. In addition to the variable left shifter, another shifter and multiplier was implemented for the Second-Extension option to calculate a new symbol between two adjacent mapped differences (δ_i and δ_{i+1}) using the Second-Extension formula:

$$\Upsilon = (\delta_i + \delta_{i+1})(\delta_i + \delta_{i+1} + 1)/2 + \delta_{i+1}$$

This algorithm was then extended to contain a simple error correction capability. Because the decoding of a given block starts with the header, damages to this portion of the packet causes a significantly set of different pixels to be decoded than the original. Therefore, protecting this header field was explored in this study. The header field is typically the smallest field in any given packet, so it can be protected using a simple redundancy code. This coding scheme simply repeats each bit of the header field two more times for a total of three bits per one bit of the original header field.

D. Exp-Golomb Encoder Implementation

Unlike the Rice encoder, the Exp-Golomb does not have a predetermined way of implementation as long as it generates the correct codeword as the algorithm dictates. For better compression, the hardware implementation of Exp-Golomb algorithm for this study uses the Pre-processor module from Rice circuit to decorrelate the pixel values within a row of pixels. Of course, this dimension must be known in both the source encoder and the transmitter's decoder. Therefore, one reference value is preserved while encoding all the differences using Exp-Golomb coding. Before generating a codeword, three values are determined in the circuit: (1) whether the current pixel is the reference, (2) the value for *M*, and (3) the value for *INFO*. The formula for *M* and *INFO* is as follows [23]:

$$M = \lfloor \log_2 (NUM + 1) \rfloor$$
$$INFO = NUM + 1 - 2^M$$

The variable *M* is the number of bits for the padded zero in the beginning of the codeword and for the bits to represent *INFO*. The variable *NUM* represents the value from the preprocessor circuit. The calculation for *M* in hardware is not trivial and can be resource intensive, so all values for *M* are generated based on the all positive values up to maximum permissible value for *NUM* and loaded into a look-up table (LUT). To access a specific entry on the LUT, the *NUM* variable becomes the address for the pre-calculated value of *M*. This method sacrifices some memory resources for the sake of using up more of the computational resources. An adder calculates the *INFO* value where one of the addends is the *NUM* value and the other is fixed to one. The output from the adder connects to a variable left shifter circuit which pushes the most significant bit of *INFO* to the most significant bit of a fixed size register, and the number of shifts comes from the value from the LUT. With the *M* and *INFO* values calculated, a final sub-module takes these values and generates the final codeword using another variable left

shifter. Because the circuit sends the reference value intact, a flag register from the Pre-processor module signals the final multiplexer whether to send either the generated codeword or original data intact (reference value).

E. Software Implementation and Introduction of Errors

In addition to the hardware implementation of source coding techniques, these algorithms are also programmed in software using C++ and MATLAB for functionality testing and simulation of error resilience. After the generation of codewords for each image, evaluating their compression power was the next step. Using a pseudorandom number generator, the errors are introduced based on the probability of error in deep space satellite communication channels mentioned in Chapter II Section B of this paper. Afterwards, the statistics are collected on the compressed, then corrupted by errors, data that will be used for the analysis of the performance of algorithms. Specifically, the compression ratio, that is the ratio of volumes of the original to the compressed data, the sum of absolute difference in pixel values, and lengths and magnitudes of damaged data are calculated as the total Hamming distance between the original and the damaged image. These equations for these values are summarized below:

$$\text{Compression Ratio} = \frac{\text{Compressed Size}}{\text{Original Size}}$$

$$\text{Sum of Absolute Difference} = \sum_{i=1}^{width} \sum_{j=1}^{height} \left| x_{ij} - x'_{ij} \right|$$

$$\text{Total Hamming Distance} = \sum_{i=1}^{width} \sum_{j=1}^{height} diff_bits\left(x_{ij} \oplus x'_{ij} \right)$$

$$\text{Damaged Pixels (\%)} = \frac{\text{Total Hamming Distance}}{\text{Original Size}} \cdot 100$$

where x_{ij} is the original pixel and x'_{ij} is the decoded pixel. The function *diff_bits* counts the number of 1's in the result of the XOR operation (⊕) between x_{ij} and x'_{ij}.

F. Design Components and Power Usage

The Computer-Aided Design (CAD) software, such as Quartus II, provides tools to analyze the number of components and the power usage after the compilation and synthesis of the circuit for each algorithm on the FPGA. For this study, the FPGA used is Cyclone IV. The component usage information is generated immediately after compilation, but to obtain the power usage information, a tool called PowerPlay Power Analyzer is invoked to generate this information. For this study, the default settings for this tool are used for fair comparison among all the encoder circuits. For the input and output signals, a default toggle rate of 12.5% is set with vectorless estimation for all remaining signals which estimates the "signal activity on nodes with no simulation" [33]. Based on these data, the determination of which algorithm uses either the fewer number of components or less amount of power is made.

IV. RESULTS AND DISCUSSION

A. *Compression Ratio*

Based on the dimensions and the range of pixel values per image, the uncompressed, i.e., original size is calculated as follows:

$$Actual\ Size = Width \cdot Length \cdot Data\ Width$$

Table I provides the compression ratios of 8-bit images encoded using the Huffman algorithm, and that includes an 8-bit remapped astronomical image.

TABLE I.
COMPRESSION RATIOS BY HUFFMAN CODING WITH 8-BIT IMAGES.

Image	Actual Size	Compressed Data Size	Compression Ratio
boat	524288	471098	1.113
cameraman	524288	427859	1.225
fingerprint	524288	500501	1.048
house	524288	459210	1.142
lena	524288	488650	1.073
fits1	8388608	2408658	3.483

Because the Rice encoder circuit allows for different parameters—data width (n) and block size (J)—all configurations of these parameters are tested for all the images. Tables II and III corresponds to the Rice coding's compression ratio without header redundancy for both 8-bit (II) and 16-bit (III) images in various configurations of the n and J parameters that are compressed using the Rice coding circuit. Furthermore, Tables IV and V list the compression ratio by Rice coding that includes the header redundancy.

TABLE II.
COMPRESSION RATIOS BY RICE CODING WITH 8-BIT IMAGES.

Image	*n*	*J*	Actual Size	Compressed Data Size	Compression Ratio
boat	8	8	524288	376675	**1.392**
	8	16	524288	362639	**1.446**
cameraman	8	8	524288	348834	**1.503**
	8	16	524288	333008	**1.574**
fingerprint	8	8	524288	496769	**1.055**
	8	16	524288	482604	**1.086**
house	8	8	524288	344424	**1.522**
	8	16	524288	325064	**1.613**
lena	8	8	524288	330110	**1.588**
	8	16	524288	311285	**1.684**
fits1	8	8	8388608	3341386	**2.511**
	8	16	8388608	2813535	**2.982**

TABLE III.
COMPRESSION RATIOS BY RICE CODING WITH 16-BIT IMAGES.

Image	*n*	*J*	Actual Size	Compressed Data Size	Compression Ratio
boat	16	8	1048576	909050	**1.153**
	16	16	1048576	891020	**1.177**
cameraman	16	8	1048576	880403	**1.191**
	16	16	1048576	861229	**1.218**
fingerprint	16	8	1048576	1029249	**1.019**
	16	16	1048576	1010988	**1.037**
house	16	8	1048576	876463	**1.196**
	16	16	1048576	853372	**1.229**
lena	16	8	1048576	862448	**1.216**
	16	16	1048576	839666	**1.249**
fits1	16	8	16777216	11518935	**1.456**
	16	16	16777216	10944396	**1.533**

TABLE IV.
COMPRESSION RATIOS BY RICE CODING WITH HEADER REDUNDANCY (8-BIT).

Image	*n*	*J*	Actual Size	Compressed Data Size	Compression Ratio
boat	8	8	524288	425853	**1.231**
	8	16	524288	387215	**1.354**
cameraman	8	8	524288	398102	**1.317**
	8	16	524288	357598	**1.466**
fingerprint	8	8	524288	545921	**0.960**
	8	16	524288	507180	**1.034**
house	8	8	524288	393644	**1.332**
	8	16	524288	349648	**1.499**
lena	8	8	524288	379296	**1.382**
	8	16	524288	335861	**1.561**
fits1	8	8	8388608	4134090	**2.029**
	8	16	8388608	3206607	**2.616**

TABLE V.
COMPRESSION RATIOS BY RICE CODING WITH HEADER REDUNDANCY (16-BIT).

Image	*n*	*J*	Actual Size	Compressed Data Size	Compression Ratio
boat	8	8	1048576	974586	1.076
	8	16	1048576	923788	1.135
cameraman	8	8	1048576	945951	1.108
	8	16	1048576	893997	1.173
fingerprint	8	8	1048576	1094785	0.958
	8	16	1048576	1043756	1.005
house	8	8	1048576	942003	1.113
	8	16	1048576	886140	1.183
lena	8	8	1048576	927984	1.130
	8	16	1048576	872434	1.202
fits1	8	8	16777216	12554583	1.336
	8	16	16777216	11464052	1.463

Tables VI and VII correspond to the compression ratio of images encoded using the Exp-Golomb algorithm for both 8-bit (VI) and 16-bit (VII) images.

TABLE VI.
COMPRESSION RATIOS BY EXP-GOLOMB CODING WITH 8-BIT IMAGES.

Image	Actual Size	Compressed Data Size	Compression Ratio
boat	524288	424934	**1.234**
cameraman	524288	376250	**1.393**
fingerprint	524288	679046	**0.772**
house	524288	376002	**1.394**
lena	524288	348908	**1.503**
fits1	8388608	2560088	**3.277**

TABLE VII.
COMPRESSION RATIOS BY EXP-GOLOMB CODING WITH 16-BIT IMAGES.

Image	Actual Size	Compressed Data Size	Compression Ratio
boat	1048576	1370524	**0.765**
cameraman	1048576	1269498	**0.826**
fingerprint	1048576	1711408	**0.613**
house	1048576	1282444	**0.818**
lena	1048576	1248634	**0.840**
fits1	16777216	16023610	**1.047**

B. Error Resilience

In this part of the study, the error resilience is evaluated based on the average absolute difference and average Hamming distance. Based on the error probability between 5×10^{-3} to 1×10^{-7}, the length of data (bitstream), and the number and the length of headers (as in the case for Rice coding), the number of bits in error are calculated accordingly [13]. With the Huffman algorithm, only the damage to 8-bit images were observed due to the software limitations of MATLAB in terms of encoding 16-bit images with Huffman encoding. With the given error probability mentioned, the best-case scenario happens when only one bit out of the entire compressed data was damaged and 12,056 bits in the worst-case. These values come from multiplying the compressed size of the astronomical image with the given probabilities, which is applied to all experiments for each source coding algorithm. Table VIII provides the error resilience analysis of the Huffman algorithm on 8-bit version of the astronomical image. The trial runs for the Huffman error resilience analysis is in the table XVII of the Appendix.

TABLE VIII.
ANALYSIS OF HUFFMAN DECODING ERRORS WITH 8-BIT FITS IMAGE.

Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)
1	1319270	310480	3.70%
12056	3880841	876081	10.44%

With the Rice coding algorithm, there are 64,817 total blocks in the compressed data with the 8-bit image and 64,487 blocks with the 16-bit image which is lower than the expected 65,536 blocks because there are blocks that are encoded using the low entropy options. In the best-case scenario, at most one bit of the header or data fields could be damaged in the compressed image, while only 324 (or 323 for 16-bit image) bits could be damaged in the worst case. The bit flipping according to BSC channel model is imposed both to the header and the data are investigated separately in these two cases to assess the fault tolerance of Rice coding. Similarly, the number of bits in error are calculated for selecting any bits in the compressed data at random. Tables IX to XII show the effect of errors that are an incorrect decoding by Rice coding algorithm with (XI-XII) or without (IX-X) header redundancy. For data gathered for the 8-bit Rice coding error resilience experiment, refer to tables XVIII to XXII and XXV to XXVI of the Appendix.

TABLE IX.
ANALYSIS OF RICE DECODING ERRORS WITH 8-BIT FITS IMAGE.

Location of Damage	Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)
Header	1	1332835	298599	3.56%
Header	324	7251211	1027525	12.25%
Data	1	1529699	344606	4.11%
Data	324	6567444	997905	11.90%
Random	1	1431267	321603	3.83%
Random	14082	14707923	1353914	16.14%

TABLE X.
ANALYSIS OF RICE DECODING ERRORS WITH 16-BIT FITS IMAGE.

Location of Damage	Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)
Header	1	190854336	1548298	9.23%
Header	323	698809977	3631052	21.64%
Data	1	2073	24	0.00%
Data	323	448789782	3447538	20.55%
Random	1	95428205	774161	4.61%
Random	54722	1070180613	3887672	23.17%

TABLE XI.
ANALYSIS OF RICE DECODING ERRORS WITH HEADER REDUNDANCY (8-BIT).

Location of Damage	Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)
Header	1	0	0	0.00%
Header	324	0	0	0.00%
Data	1	685928	155224	4.84%
Data	324	5410111	958569	11.43%
Random	1	342964	77612	0.93%
Random	14082	15861233	1509445	17.99%

TABLE XII.
ANALYSIS OF RICE DECODING ERRORS WITH HEADER REDUNDANCY (16-BIT).

Location of Damage	Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)
Header	1	0	0	0.00%
Header	324	0	0	0.00%
Data	1	1714	13	0.00%
Data	324	362385432	3323093	19.81%
Random	1	857	7	0.00%
Random	14082	988003116	3831081	22.84%

Similar calculations were made for the number of bits in error for the Exp-Golomb algorithm based on the size of the compressed images as with Huffman coding. Tables XIII and XIV display the effect of errors due to incorrect decoding by the Exp-Golomb coding algorithm. The individual trial runs for the Exp-Golomb study are in tables XXIII and XXIX.

TABLE XIII.
ANALYSIS OF EXP-GOLOMB DECODING ERRORS WITH 8-BIT FITS IMAGE.

Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)
1	2677964	256592	3.06%
12814	23588754	2194615	26.16%

TABLE XIV.
ANALYSIS OF EXP-GOLOMB DECODING ERRORS WITH 16-BIT FITS IMAGE.

Number of Bits in error	Avg Absolute Difference	Avg Hamming Distance	Avg Corrupted Pixels (%)

1	92868236	928164	5.53%
80119	641204648	4052670	24.16%

C. Resource Utilization

Table XI presents the breakdown of the resource utilization of each source coding algorithms. Because the Rice encoder circuit is constructed using parameterized modules, the parameter variables for data width and block size are changed accordingly before compilation to observe that Rice configuration's components usage. Similarly, different configurations of the Exp-Golomb circuit, specifically the data width, were observed. With the Huffman encoder circuit, the 8-bit version could not fit into the FPGA used in this study, so it has been modified. In this case, the 4-bit version was recorded below for comparison where its components usage extrapolated for comparing among the 8-bit versions of each source coding algorithm.

TABLE XV.
COMPONENTS USAGE OF EACH SOURCE CODING ALGORITHM.

Components	total logic elements	total combinational functions	total registers
Huffman (4-bit)*	308	300	177
Rice (n = 8, J = 8)	338	330	56
(header_red)	336	335	56
Rice (n = 8, J = 16)	426	426	58
(header_red)	434	434	58
Rice (n = 16, J = 8)	517	517	74
(header_red)	557	541	74
Rice (n = 16, J = 16)	1130	1130	75
(header_red)	1152	1136	75
Exp-Golomb (8-bit)	354	346	72
Exp-Golomb (16-bit)	759	758	89

D. Power Consumption

Using the PowerPlay Analyzer tool with the default settings in the Quartus software, the power consumption estimates for each lossless compression circuit is obtained. Table XII shows the breakdown of the power consumption per source coding algorithm.

TABLE XVI.
ESTIMATED POWER CONSUMPTION IN MILLIWATTS.

Circuit	Estimated Power Dissipation (mW)
Huffman (4-bit)*	215.25
Rice (*n* = 8, *J* = 8)	172.77
(header_red)	176.87
Rice (*n* = 8, *J* = 16)	181.20
(header_red)	177.28
Rice (*n* = 16, *J* = 8)	193.31
(header_red)	189.76
Rice (*n* = 16, *J* = 16)	214.94
(header_red)	214.98
Exp-Golomb (8-bit)	175.56
Exp-Golomb (16-bit)	190.11

Discussion

From Tables I, II, and VI, it can be concluded that the Rice coding method employed for development of CCSDS 121.0-B-1 standard scheme exhibits the highest performance in terms of compression ratio among the 8-bit images [7]. This result is mainly achieved by the low entropy options, i.e., Zero Block and Second Extension. The Zero Block is especially beneficial to the Rice's high compression ratio. For example, with $n = 8$ and $J = 8$, one zero block encoded using Rice compresses 64 bits of pixel data into a 13-bit codeword, including the option ID and reference pixel value. With two zero blocks, 128 bits of pixel data turns into a 14-bit codeword and so on. Likewise, the Exp-Golomb coding has a lower compression ratio than Rice coding.

However, there is one notable exception to this observation which is the 8-bit remapped version of the astronomical image. Its compression ratio was the highest when encoded using the Huffman algorithm followed by Rice and then Exp-Golomb. This anomaly comes from the remapping of a small subset of all possible 16-bit values onto an even smaller subset of the 8-bit values. Because the histogram of this remapped image is much smaller than the original, fewer pixel values are needed to build the Huffman coding dictionary. Therefore, this set of pixel values are encoded with fewer bits than if it was encoded using the mapped difference in either Rice or Exp-Golomb algorithms.

From Tables III and VII, the Rice coding algorithm performs with a highest compression ratio between itself and Exp-Golomb for 16-bit images. In fact, the compression ratios by Exp-Golomb of the benchmark images are below one as indicated with the higher compressed size than the actual size. However, it still performs compression on the astronomical image because the differences among its pixel values are much smaller than that for the benchmark images. Therefore, in the case of the benchmark images, it is better to send the pixel values as is rather than encode them using the Exp-Golomb algorithm.

Focusing on just the Rice coding compression ratios shows the variation in compression ratios for different configurations n and J. In Table II, the size of each compressed image when the block size is 16 is smaller than that for $J = 8$. Consequently, the compression ratio for the images that are decorrelated using larger block sizes are greater. Similarly, both the 16-bit FITS and the histogram-stretched benchmark images (Table III) also followed this trend, i.e., display a greater compression ratio when the block size of 16 was selected than that for the block size of 8. The larger block size option has a greater compression ratio because it captures more of the differences in one packet than that smaller block size. Furthermore, the compression ratio of the

images encoded with the header redundancy codes (Tables IV and V) are lesser than those from the original encoding scheme as expected because of those extra bits in the header. Most of the compressed benchmark images have compression ratios that are greater than one except for the *fingerprint* benchmark image. This anomaly was due to its compression ratio without header redundancy being very close to one, so the addition of the redundant bits made the effect of the compression insignificant.

As mentioned in previous sections, the Huffman software implementation is limited to encoding only 8-bit images, so just the 8-bit remapped version of the astronomical image rather than the original 16-bit image was used for the error resilience study. From tables VIII, IX, and XIII, in the best-case scenario where only one bit is flipped Exp-Golomb coding performs the best in terms of error resilience but only slightly better than Huffman or Rice coding. However, the worst-case scenario produces a different outcome where Huffman coding performs well over Rice or Exp-Golomb algorithm due to its ability to self-correct in a very short time (shorter erroneous sequence) which is beneficial in the presence of many bits in error [17]. This observation, therefore, disproves the hypothesis that Rice coding performs better than Huffman coding as shown in both extremes of error probabilities without the introduction of simple error-correcting codes. By introducing the redundancy codes (Table XI) in the header in Rice codes, the average percentage of corrupted pixels drops from 3.83% to 0.93% in the best-case scenario which is significantly better when compared to Huffman or Exp-Golomb coding. The header redundancy codes can easily correct a single bit error in the header for the best-case. However, it does not improve on the worst-case scenario where Huffman coding still dominates.

From tables X and XIV, the error resilience of Rice coding is higher than that of Exp-Golomb for the original 16-bit astronomical image, i.e., the case for deep space communication.

This observation is seen in both the best and the worst-case scenarios. Because the Rice algorithm allows for parallel calculation of all its options and selecting from the code of least length, the Exp-Golomb algorithm does not achieve the same performance as Rice coding. With the header redundancy for the Rice algorithm (Table XII), the error resilience in the best case becomes diminished while the performance in the worst case slightly improves. Therefore, the addition of redundancy codes in the header field in Rice coding provides a slight performance improvement which could be omitted as a mean to protect the header field.

In terms of decoding errors with the header errors encoded using Rice coding shown in Table X, the comparison suggests that the header errors introduce a significant issue. The first two rows of the table also show a variation in the length of error propagation of a single bit error in the header where a single bit flip event produces a more favorable outcome than multiple bit flip events. The difference in percentage of incorrectly decoded bits for the data field however tells a completely different story. With an average Hamming distance of 24 for the best-case, the error is contained within a block or two of pixel data because the minimum packet size in this configuration is 22, and the maximum is 261. Therefore, the percentage of incorrect bits in the data field of the packet is close to zero for the best-case scenario.

During the compilation and programming of the Huffman algorithm using the FPGA software, the implementation laid out in this study cannot fit within the hardware limitations of the FPGA used in this study, which was the same for all coding algorithms, for a fair comparison. Consequently, the Verilog code for the Huffman circuit was modified by changing the parameters in the code until it suited the hardware requirements, so the circuit can only handle 4-bit data given this implementation. Comparing this limited Huffman circuit with the lowest configuration of Rice and Exp-Golomb, the Rice encoder circuit uses the fewest

components, even with header redundancy incorporated into the encoder. When the data width is set to 16-bit, the Rice encoder circuit still has the least components usage only when the block size is set to 8. Conversely, the Exp-Golomb circuit uses fewer components the 16-bit Rice encoder when the block size is set to 16.

With the same issue as for the Huffman encoder circuit, only its limited 4-bit version was observed for the power consumption analysis. Looking at just the most limited version of each source encoder circuit, the Rice encoder consumes the least amount of power among the three implemented source coding algorithms even when comparing against the limited version of the Huffman encoder. In fact, this 4-bit Huffman circuit uses the most amount of power among all the hardware implementations in this study, so it performs the worst in terms of power dissipation. Comparing the 16-bit versions of Rice and Exp-Golomb circuits, the latter encoder outperforms the former one even against the two configurations of the block size in the Rice coding circuit. Given the compression ratio of 16-bit Exp-Golomb against the 16-bit Rice encoder in both block size configurations, the Rice encoder provides the best tradeoff between the compression ratio, component usage, and the power consumption over the Exp-Golomb encoder.

In terms of compressing the 8-bit benchmark images given in the results section, the Rice encoder circuit, specifically with $J = 8$, outperforms the Exp-Golomb method. The addition of the repetition code to the header in the Rice method does give a significant advantage in error resilience, components usage, and power consumption for the compression of the benchmark images, or more generally, public domain images. In fact, it only improves the error resilience in the best-case scenario while increasing the power consumption of the circuit. The summary of the analysis for 8-bit encoder circuits is shown in Table XVII. Therefore, among the source

coding methods in this study, the Rice encoder circuit with the block size of 8 gives a best configuration based on all the measurements performed in this study.

TABLE XVII.
SUMMARY OF COMPARISON OF METHODS FOR 8-BIT IMAGES.

	Huffman	**Rice**
Compression Ratio		X
Error Resilience	X	
Components Usage		X
Power Consumption		X

For astronomical images transmitted with 16-bit representation per pixel the Huffman encoder is not practical due to the huge number of components needed for implementation. Therefore, only the Rice and Exp-Golomb methods are the suitable candidates for comparison in terms of their feasibility for the application. Even though the Rice encoder with the block size of 16 provides a greater compression ratio than the one of smaller block size, the Rice circuit with $J = 8$ outperforms the Exp-Golomb encoder in all categories except in power consumption. This type of Rice encoder uses significantly fewer components than the larger Rice counterpart and Exp-Golomb, and it performs well in the presence of errors in both the best and worst-case scenarios. Furthermore, augmenting the Rice code with the repetition code of the header field, although it adds more components and reduces the compression ratio, it performs better than the Exp-Golomb coder with the added benefit of improving the error resilience, especially in the best-case scenario. The summary of the analysis among 16-bit encoder circuits is shown in Table XVIII. The Rice circuit with the block size of 8, augmented for error resilience with the repetition code represents a most favorable tradeoff by three categories: compression ratio, error resilience and the hardware complexity. Although the power usage is not of a lesser significance given the on-board operation, with the employment of efficient technologies and power

optimization methods for the circuit design, further improvements can be achieved. However, that is outside the scope of the current work and its goals in this pilot research of the author. The conclusions drawn in this study prove not only that this early CCSDS standard is a valid candidate for the use on board by NASA deep space exploration missions, but it also can be used for losslessly encoding the public domain 8-bit images. It would be interesting and useful to perform in the future the analysis of the recent CCSDS, such as 122.0-B-2 and include hyperspectral and multispectral datasets for the performance analysis. Also, the study in general is to be performed in association with the communication protocol and adopted channel coding methods, that is in an integral framework of operation.

TABLE XVIII.
SUMMARY OF COMPARISON FOR 16-BIT IMAGES.

	Rice	**Exp-Golomb**
Compression Ratio	X	
Error Resilience	X	
Components Usage	X	
Power Consumption		X

Acknowledgement

This work is partially supported by NASA through Nevada Space Grant #NNX15AI02H.

REFERENCES

[1] D. Le Ruyet and M. Pischella, *Digital Communications 1: Source and Channel Coding*, 1st ed. London: John Wiley & Sons, Inc., 2015. pp. 1, 71

[2] K. Sayood, *Introduction to Data Compression*, 4th ed. San Francisco, CA: Morgan Kaufmann, 2006. pp. 1, 3-5, 58

[3] H. Hihara, K. Moritani, M. Inoue, Y. Hoshi, A. Iwasaki, J. Takada, H. Inada, M. Suzuki, T. Seki, S. Ichikawa, and J. Tanii, "Onboard Image Processing System for Hyperspectral Sensor," *Sensors*, vol. 15, no. 10, pp. 24926–24944, Sep. 2015.

[4] J. Zumberge, L. Deutsch, and S. Townes (n.d.). "Deep Space Communications." [Website]. Available: https://scienceandtechnology.jpl.nasa.gov/research/research-topics-list/communications-computing-software/deep-space-communications. Accessed Mar. 17, 2018.

[5] J. Heller and I. Jacobs, "Viterbi Decoding for Satellite and Space Communication," in *IEEE Transactions on Communication Technology*, vol. 19, no. 5, pp. 835-848, October 1971. doi: 10.1109/TCOM.1971.1090711

[6] P. Koopman and T. Chakravarty, "Cyclic redundancy code (CRC) polynomial selection for embedded networks," *International Conference on Dependable Systems and Networks, 2004*, 2004, pp. 145-154. doi: 10.1109/DSN.2004.1311885

[7] CCSDS. (2012, Apr. 2). *Lossless Data Compression* [Online]. Available FTP: public.ccsds.org Directory: Pubs/ File: 121x0b2ec1.pdf

[8] Y. Erdem, A. M. N. Uyar, M. C. Soydan, M. S. Harmankaya, F. Alan and B. Akbulut, "Developing and modelling of satellite docking algorithm," 2017 8th International

Conference on Recent Advances in Space Technologies (RAST), Istanbul, 2017, pp. 465-471. doi: 10.1109/RAST.2017.8002987

[9] A. García, L. Santos, S. López, G. Marrero, J. F. López and R. Sarmiento, "High level modular implementation of a lossy hyperspectral image compression algorithm on a FPGA," *2013 5th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, Gainesville, FL, USA, 2013, pp. 1-4. doi: 10.1109/WHISPERS.2013.8080624

[10] M. Biskup. "Error Resilience in Compressed Data—Selected Topics". Ph.D. dissertation, Dept. of Math., Inform., and Mech., Univ. of Warsaw, Warsaw, Poland, 2008.

[11] S. T. Klein and D. Shapira, "Practical fixed length Lempel–Ziv coding," *Discrete Applied Mathematics*, vol. 163, no. 3, pp. 326-333, 2014.

[12] B. Pratt, M. Fuller, M. Rice and M. Wirthlin, "Reduced-Precision Redundancy for Reliable FPGA Communications Systems in High-Radiation Environments," in *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 1, pp. 369-380, Jan. 2013. doi: 10.1109/TAES.2013.6404109

[13] W. Cary Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*, 1st ed. Cambridge: Cambridge University Press, 2015. pp. 4, 578

[14] F. Lansing, L. Lemmerman, A. Walton, G. Bothwell, K. Bhasin and G. Prescott, "Needs for communications and onboard processing in the vision era," *IEEE International Geoscience and Remote Sensing Symposium*, 2002, pp. 375-377 vol.1. doi: 10.1109/IGARSS.2002.1025044

[15] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” in *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, Sept. 1952. doi: 10.1109/JRPROC.1952.273898

[16] Wook-Hyun Jeong, Young-Suk Yoon and Yo-Sung Ho, “Design of reversible variable-length codes using properties of the Huffman code and average length function*,” Image Processing, 2004. ICIP '04. 2004 International Conference on*, 2004, pp. 817-820 Vol.2. doi: 10.1109/ICIP.2004.1419423

[17] D. Lelewer and D. Hirschberg, “Data Compression,” *ACM Computing Surveys (CSUR),* vol. 19, no. 3, pp. 261-296, 1987. doi: 10.1145/77556.77566

[18] R. F. Rice, “Some Practical Universal Noiseless Coding Techniques,” Jet Propulsion Lab., California Inst. of Tech., Pasadena, CA, Rep. NASA-CR-158515, Mar. 1979

[19] Pen-Shu Yeh, R. F. Rice, and W. Miller, “On the Optimality of Code Options for a Universal Noiseless Coder,” Jet Propulsion Lab., California Inst. of Tech., Pasadena, CA, Rep. NASA-CR-187973, Feb. 1991

[20] R. F. Rice and Pen-Shu Yeh, “Algorithms for a very high speed universal noiseless coding module,” Jet Propulsion Lab., California Inst. of Tech., Pasadena, CA, Rep. NASA-CR-187974, Feb. 1991

[21] J. J. Meany and C. J. Martens, “Split field coding: Low complexity error-resilient entropy coding for image compression,” in *Proc. of SPIE - The International Society for Optical Engineering*, San Diego, CA, 2008. doi: 10.1117/12.797357.

[22] J. Teuhola, “A compression method for clustered bit-vectors,” *Inform. Process. Lett.*, vol. 7, pp. 308–311, Oct. 1978.

[23] J. Chen, Y. Chen, H. Zhu, C. Sui, P. Wu and X. Cao, "The hardware design and implementation for CAVLC and Exp-Golomb in H.264/AVC," *2010 The 12th International Conference on Advanced Communication Technology (ICACT)*, Phoenix Park, 2010, pp. 1610-1613.

[24] J. Zhang (1994, May 23). "Parity Check Codes." [Website]. Available: http://www.rpi.edu/locker/75/000475/main/subsection3_7_1.html. Accessed Oct. 7, 2017.

[25] B. M.J. Leiner. (2005). *LDPC Codes – a brief Tutorial* [Online]. Available FTP: bernh.net Directory: media/download/papers File: ldpc.pdf

[26] M. Sarkar, K. K. Shukla, and K. S. Dasgupta, "A Survey of Transport Protocols for Deep Space Communication Networks," *International Journal of Computer Applications*, vol. 31, no. 8, pp. 25-32, Oct. 2011.

[27] Numberphile (2013, Sept. 10). "Error Correction." [YouTube Video]. Available: https://www.youtube.com/watch?v=5sskbSvha9M. Accessed Dec. 7, 2017.

[28] A. Pang and P. Membrey, *Beginning FPGA: Programming Metal: Your Brain on Hardware*, 1st ed. Springer, 2017.

[29] M. J. Wirthlin, "FPGAs operating in a radiation environment: Lessons learned from FPGAs in space," *Journal of Instrumentation,* vol. 8, no. 2, 2013.

[30] G. Mayer. (2009). "Image Repository." [Website]. Available: http://links.uwaterloo.ca/Repository.html. Accessed Dec. 10, 2017.

[31] NASA & GSFC. (2017, Dec. 4). "The FITS Support Office." [Website]. Available: https://fits.gsfc.nasa.gov. Accessed Jan. 10, 2018.

[32] S. Dong, X. Wang and X. Wang, "A Novel High-Speed Parallel Scheme for Data Sorting Algorithm Based on FPGA," *2009 2nd International Congress on Image and Signal Processing*, Tianjin, 2009, pp. 1-4. doi: 10.1109/CISP.2009.5302455

[33] Altera. (2013, Nov. 4). *PowerPlay Power Analysis* [Online]. Available FTP: altera.com.cn Directory: zh_CN/pdfs/literature/hb/qts File: qts_qii53013.pdf

APPENDIX

TABLE XIX.
TRIAL RUNS FOR 8-BIT HUFFMAN ERROR RESILIENCE EXPERIMENT.

Random Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	17	3	0.000%
2	817007	197597	8.204%
3	869312	209227	8.686%
4	2204295	513175	21.305%
5	2390769	558091	23.170%
6	1182951	279053	11.585%
7	1534106	359985	14.945%
8	1947049	453343	18.821%
9	614971	152563	6.334%
10	1826107	425851	17.680%
11	165837	51103	2.122%
12	27648	10319	0.428%
13	1086207	257505	10.691%
14	2307406	537837	22.329%
15	2261427	526837	21.873%
16	8	1	0.000%
17	2415310	564075	23.419%
18	2457929	574703	23.860%
19	1123971	265935	11.041%
20	1153066	272397	11.309%
Random Damage (12056 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	3882603	876160	36.375%
2	3869506	874269	36.297%
3	3846757	870546	36.142%
4	3901790	879977	36.534%
5	3872549	873986	36.285%
6	3902863	880488	36.555%
7	3890216	877906	36.448%
8	3890830	877117	36.415%
9	3851440	870923	36.158%
10	3916372	882493	36.638%
11	3884815	877983	36.451%
12	3881590	875514	36.349%
13	3859873	872636	36.229%
14	3912323	881254	36.587%
15	3861227	872749	36.234%
16	3891275	877759	36.442%
17	3886598	876842	36.404%
18	3923289	883617	36.685%
19	3840873	869020	36.079%
20	3850032	870386	36.136%

TABLE XX.
TRIAL RUNS FOR 8-BIT RICE ERROR RESILIENCE EXPERIMENT (HEADER).

Header Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	1819526	406951	4.851%
2	3755814	837277	9.981%
3	1731583	384321	4.581%
4	3924602	881518	10.509%
5	3492099	774619	9.234%
6	96	33	0.000%
7	1479338	331026	3.946%
8	1714670	390377	4.654%
9	111	38	0.000%
10	2631875	584548	6.968%
11	130	22	0.000%
12	29	8	0.000%
13	1979228	443881	5.291%
14	150	45	0.001%
15	527823	130307	1.553%
16	136	25	0.000%
17	38501	8914	0.106%
18	252	27	0.000%
19	2780040	609683	7.268%
20	780697	188358	2.245%
Header Damage (324 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	7182570	1028815	12.264%
2	7658829	1041075	12.411%
3	7392607	1034217	12.329%
4	6837700	1007315	12.008%
5	6846669	1008449	12.022%
6	7338415	1033589	12.321%
7	7257717	1028934	12.266%
8	7328980	1023189	12.197%
9	7401249	1036219	12.353%
10	7287985	1031122	12.292%
11	7095202	1026352	12.235%
12	7200146	1030722	12.287%
13	7214374	1024777	12.216%
14	7368017	1025569	12.226%
15	7197162	1025029	12.219%
16	7373443	1031683	12.299%
17	7465274	1039146	12.388%
18	7190939	1025438	12.224%
19	7306086	1022090	12.184%
20	7080860	1026770	12.240%

TABLE XXI.
TRIAL RUNS FOR 8-BIT RICE ERROR RESILIENCE EXPERIMENT (DATA).

Data Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	55	26	0.000%
2	3902900	874543	10.425%
3	3815215	854802	10.190%
4	1996942	450401	5.369%
5	1109788	257059	3.064%
6	3753552	835301	9.958%
7	959198	220327	2.627%
8	3335573	741524	8.840%
9	87	31	0.000%
10	2643642	585032	6.974%
11	84	19	0.000%
12	7814	676	0.008%
13	114	37	0.000%
14	701908	164669	1.963%
15	750947	178407	2.127%
16	1990467	447609	5.336%
17	532718	131675	1.570%
18	1716981	384994	4.589%
19	742356	177257	2.113%
20	2633647	587733	7.006%
Data Damage (324 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	6458775	988053	11.779%
2	6867325	1003783	11.966%
3	6640115	1006094	11.994%
4	6806181	1014552	12.094%
5	6701865	1009206	12.031%
6	6307349	974685	11.619%
7	6604492	995458	11.867%
8	6614416	1001797	11.942%
9	6681765	1008326	12.020%
10	6455344	997199	11.888%
11	6237473	986707	11.762%
12	6520065	996534	11.880%
13	6534512	999981	11.921%
14	6766183	1005211	11.983%
15	6696204	1004802	11.978%
16	6325043	990038	11.802%
17	6459167	984776	11.739%
18	6468418	992630	11.833%
19	6425249	994878	11.860%
20	6778931	1003390	11.961%

TABLE XXII.
TRIAL RUNS FOR 8-BIT RICE ERROR RESILIENCE EXPERIMENT (RANDOM).

Random Damage (14082 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	14637376	1345148	16.035%
2	14738036	1356845	16.175%
3	14848087	1357713	16.185%
4	14872157	1362057	16.237%
5	14596271	1350780	16.103%
6	14565187	1346154	16.047%
7	14639161	1352921	16.128%
8	14825113	1356047	16.165%
9	14626433	1351496	16.111%
10	14605141	1346561	16.052%
11	14887306	1355723	16.161%
12	14720539	1356311	16.168%
13	14586056	1348726	16.078%
14	14626263	1355970	16.164%
15	14718860	1355422	16.158%
16	14822796	1358815	16.198%
17	14524473	1344188	16.024%
18	14646250	1353241	16.132%
19	14711982	1354303	16.145%
20	14960972	1369864	16.330%

TABLE XXIII.
TRIAL RUNS FOR 8-BIT RICE ERROR RESILIENCE EXPERIMENT WITH REDUNDANCY (DATA).

Data Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	1075755	252179	7.864%
2	140	28	0.001%
3	784	26	0.001%
4	2980650	664822	20.733%
5	107	26	0.001%
6	149	44	0.001%
7	10866	1178	0.037%
8	3003685	669762	20.887%
9	252	27	0.001%
10	1279	114	0.004%
11	208	61	0.002%
12	2203245	498096	15.533%
13	3985	244	0.008%
14	1005666	230421	7.186%
15	1354	76	0.002%
16	60668	24564	0.766%
17	8	1	0.000%
18	2425407	540437	16.854%
19	944278	222355	6.934%
20	71	14	0.000%
Data Damage (324 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	5377307	952132	11.350%
2	5416287	964883	11.502%
3	5280640	954784	11.382%
4	5300028	953878	11.371%
5	5478421	969348	11.556%
6	5295709	950032	11.325%
7	5484504	964798	11.501%
8	5396984	949182	11.315%
9	5249051	951525	11.343%
10	5412640	952313	11.352%
11	5453159	967689	11.536%
12	5476798	965184	11.506%
13	5413777	956190	11.399%
14	5524393	958372	11.425%
15	5489761	967237	11.530%
16	5488894	960100	11.445%
17	5386920	958648	11.428%
18	5247023	947819	11.299%
19	5445503	958979	11.432%
20	5584415	968286	11.543%

TABLE XXIV.
TRIAL RUNS FOR 8-BIT RICE ERROR RESILIENCE EXPERIMENT WITH REDUNDANCY (RANDOM).

Random Damage (16050 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	15836345	1508453	17.982%
2	15760477	1500320	17.885%
3	15933129	1513087	18.037%
4	16012892	1516667	18.080%
5	15798478	1508253	17.980%
6	15925106	1507758	17.974%
7	15734249	1502472	17.911%
8	15918325	1517782	18.093%
9	15870590	1507130	17.966%
10	15926694	1511543	18.019%
11	16003146	1508107	17.978%
12	15597190	1502704	17.914%
13	15754823	1510094	18.002%
14	16005732	1518057	18.097%
15	16054229	1520822	18.130%
16	15690786	1498027	17.858%
17	16030433	1512360	18.029%
18	15624898	1499480	17.875%
19	15867938	1517229	18.087%
20	15879201	1508553	17.983%

TABLE XXV.
TRIAL RUNS FOR 8-BIT EXP-GOLOMB ERROR RESILIENCE EXPERIMENT.

Random Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	337	73	0.001%
2	9495	2640	0.031%
3	19768458	1946306	23.202%
4	77382	5324	0.063%
5	21663648	2009250	23.952%
6	0	0	0.000%
7	1138	643	0.008%
8	6526	2584	0.031%
9	1069394	136861	1.632%
10	2586	1072	0.013%
11	397270	25838	0.308%
12	1902	690	0.008%
13	1875	738	0.009%
14	2424	901	0.011%
15	349786	40739	0.486%
16	97815	8589	0.102%
17	1991	752	0.009%
18	1117	242	0.003%
19	10101761	947338	11.293%
20	4373	1263	0.015%
Random Damage (12814 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	23528195	2205192	26.288%
2	24303568	2234688	26.640%
3	23859326	2209337	26.337%
4	23325496	2170150	25.870%
5	23187270	2185088	26.048%
6	24665144	2248616	26.806%
7	23671731	2196789	26.188%
8	22823063	2147591	25.601%
9	22725932	2152531	25.660%
10	23575525	2195797	26.176%
11	23397314	2185185	26.049%
12	23344354	2171705	25.889%
13	24802977	2237365	26.671%
14	24053457	2217177	26.431%
15	23065888	2158950	25.737%
16	23158881	2177966	25.963%
17	24177871	2234178	26.633%
18	23762810	2221282	26.480%
19	22688727	2152243	25.657%
20	23657557	2190462	26.112%

TABLE XXVI.
TRIAL RUNS FOR 16-BIT RICE ERROR RESILIENCE EXPERIMENT (HEADER).

Header Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	303744930	2452810	14.620%
2	331188125	2678033	15.962%
3	183268796	1514728	9.028%
4	319472929	2584609	15.405%
5	43797302	348101	2.075%
6	285004911	2311080	13.775%
7	37707	200	0.001%
8	128668612	1046713	6.239%
9	74569971	568385	3.388%
10	81945077	641781	3.825%
11	262675869	2148200	12.804%
12	292245935	2392739	14.262%
13	248098001	2010837	11.986%
14	322058382	2608486	15.548%
15	60169	124	0.001%
16	209817837	1712486	10.207%
17	144861427	1189982	7.093%
18	137828195	1116005	6.652%
19	267307689	2168707	12.927%
20	180434857	1471962	8.774%
Header Damage (323 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	706924182	3623763	21.599%
2	688613318	3603515	21.479%
3	671577919	3607249	21.501%
4	704974727	3650935	21.761%
5	670804584	3618486	21.568%
6	702603714	3641269	21.704%
7	672248342	3600178	21.459%
8	716332614	3644672	21.724%
9	723667152	3668140	21.864%
10	671381510	3611432	21.526%
11	687264320	3630078	21.637%
12	717637771	3645357	21.728%
13	681060935	3624259	21.602%
14	683864892	3610687	21.521%
15	682423193	3631810	21.647%
16	719922524	3638707	21.688%
17	712758200	3602869	21.475%
18	728754242	3663934	21.839%
19	722786079	3636605	21.676%
20	710599323	3667100	21.858%

TABLE XXVII.
TRIAL RUNS FOR 16-BIT RICE ERROR RESILIENCE EXPERIMENT (DATA).

Data Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	63	25	0.000%
2	3541	31	0.000%
3	98	6	0.000%
4	4012	48	0.000%
5	12686	47	0.000%
6	248	24	0.000%
7	4	1	0.000%
8	0	0	0.000%
9	3538	38	0.000%
10	1701	16	0.000%
11	4464	39	0.000%
12	320	64	0.000%
13	1659	25	0.000%
14	88	13	0.000%
15	958	7	0.000%
16	1482	16	0.000%
17	1113	17	0.000%
18	793	9	0.000%
19	2282	15	0.000%
20	2413	30	0.000%
Data Damage (323 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	443967673	3460995	20.629%
2	445641274	3475262	20.714%
3	436859835	3463491	20.644%
4	460517540	3485632	20.776%
5	473660528	3506924	20.903%
6	443823660	3472236	20.696%
7	473539867	3502567	20.877%
8	440788909	3448354	20.554%
9	428333157	3411949	20.337%
10	491801340	3512497	20.936%
11	436780249	3464074	20.647%
12	435933820	3356790	20.008%
13	431900236	3447659	20.550%
14	497686854	3497432	20.846%
15	413461865	3305859	19.704%
16	453766148	3470524	20.686%
17	454995324	3464052	20.647%
18	462253855	3349866	19.967%
19	419843120	3425653	20.418%
20	430240378	3428939	20.438%

TABLE XXVIII.
TRIAL RUNS FOR 16-BIT RICE ERROR RESILIENCE EXPERIMENT (RANDOM).

Random Damage (54722 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	1061355563	3883136	23.145%
2	1061489112	3890170	23.187%
3	1081689247	3898802	23.239%
4	1067667640	3881912	23.138%
5	1067890547	3884411	23.153%
6	1074392056	3894437	23.213%
7	1065611836	3884058	23.151%
8	1072481462	3896976	23.228%
9	1071142203	3886484	23.165%
10	1068929954	3888009	23.174%
11	1073280514	3884300	23.152%
12	1066707676	3884978	23.156%
13	1080764427	3894890	23.215%
14	1066292947	3881540	23.136%
15	1077648463	3891357	23.194%
16	1075925812	3884923	23.156%
17	1070087494	3886198	23.164%
18	1065048964	3885811	23.161%
19	1075138999	3892368	23.200%
20	1060067340	3878682	23.119%

TABLE XXIX.
TRIAL RUNS FOR 16-BIT RICE ERROR RESILIENCE EXPERIMENT WITH REDUNDANCY (DATA).

Data Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	0	0	0.0000%
2	937	7	0.0000%
3	3372	27	0.0002%
4	917	8	0.0000%
5	917	7	0.0000%
6	122	6	0.0000%
7	887	8	0.0000%
8	19888	114	0.0007%
9	0	0	0.0000%
10	117	8	0.0000%
11	0	0	0.0000%
12	1777	11	0.0001%
13	0	0	0.0000%
14	4	4	0.0000%
15	999	18	0.0001%
16	839	14	0.0001%
17	1868	17	0.0001%
18	1640	11	0.0001%
19	1	1	0.0000%
20	0	0	0.0000%
Data Damage (324 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	351548062	3212510	19.148%
2	341369517	3038032	18.108%
3	355131605	3316822	19.770%
4	356444620	3325764	19.823%
5	355374872	3299467	19.666%
6	348883344	3216564	19.172%
7	381288746	3391493	20.215%
8	375338105	3384181	20.171%
9	374048766	3422972	20.403%
10	348835954	3273820	19.513%
11	363966179	3354732	19.996%
12	359375436	3350904	19.973%
13	364391375	3384432	20.173%
14	371497177	3385920	20.182%
15	378058231	3435189	20.475%
16	362930633	3376494	20.125%
17	368861905	3296500	19.649%
18	354867424	3268665	19.483%
19	371759467	3358156	20.016%
20	363737222	3369250	20.082%

TABLE XXX.
TRIAL RUNS FOR 16-BIT RICE ERROR RESILIENCE EXPERIMENT WITH REDUNDANCY (RANDOM).

Random Damage (57321 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	988722825	3822795	22.786%
2	985580745	3835798	22.863%
3	974104914	3820003	22.769%
4	976766792	3815359	22.741%
5	995751087	3846657	22.928%
6	992056436	3832683	22.845%
7	983740546	3828682	22.821%
8	980498668	3822570	22.784%
9	978145332	3828440	22.819%
10	992660822	3835301	22.860%
11	999630120	3840467	22.891%
12	989083296	3830931	22.834%
13	995596350	3835668	22.862%
14	999332483	3834867	22.858%
15	989490256	3833132	22.847%
16	994010068	3839841	22.887%
17	981717612	3828803	22.821%
18	980102324	3826708	22.809%
19	996558599	3839468	22.885%
20	986513036	3823444	22.790%

TABLE XXXI.
TRIAL RUNS FOR 8-BIT EXP-GOLOMB ERROR RESILIENCE EXPERIMENT.

Random Damage (1 bit)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	326102501	3183852	18.977%
2	11539	416	0.002%
3	141517	1384	0.008%
4	713896	3122	0.019%
5	199409848	2032568	12.115%
6	51949	607	0.004%
7	44931	1090	0.006%
8	706271	5089	0.030%
9	154420184	1599970	9.537%
10	17366	759	0.005%
11	226472220	2309115	13.763%
12	199283	2251	0.013%
13	313838681	3271388	19.499%
14	48043	1536	0.009%
15	269458679	2513362	14.981%
16	61317741	530512	3.162%
17	213419	1718	0.010%
18	303658716	3100496	18.480%
19	521443	3591	0.021%
20	16484	463	0.003%
Random Damage (80119 bits)			
Trial #	**Avg Absolute Difference**	**Avg Hamming Distance**	**Avg Corrupted Pixels (%)**
1	649173770	4056517	24.179%
2	614663999	4057349	24.184%
3	643417814	4060392	24.202%
4	624855380	4050042	24.140%
5	607307598	4029324	24.017%
6	630442915	4060636	24.203%
7	621145589	4056240	24.177%
8	613870012	4029944	24.020%
9	638248445	4039606	24.078%
10	640834283	4086853	24.360%
11	638279469	4060004	24.200%
12	609364735	4049210	24.135%
13	634689863	4068798	24.252%
14	616044168	4047317	24.124%
15	621817176	4065864	24.234%
16	643287223	4045014	24.110%
17	928597285	4063117	24.218%
18	611443553	4050636	24.144%
19	621526336	4044267	24.106%
20	615083354	4032278	24.034%