2003

Near-Lossless Bitonal Image Compression System

Jeremy Pyle

Near-Lossless Bitonal Image Compression System

By

Jeremy Pyle

A Thesis Submitted

in

Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in

Computer Engineering

Approved by:

Principal Advisor_____
Dr. Andreas Savakis, Associate Professor and Department Head

Committee Member_____
Dr. Kenneth Hsu, Professor

Committee Member_____
Dr. Marcin Lukowiak, Visiting Assistant Professor

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
November 2003

Release Permission Form

Rochester Institute of Technology

Near-Lossless Bitonal Image Compression System

I, Jeremy Pyle, hereby grant permission to the Wallace Library and the Department of Computer Engineering at the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Jeremy C. Pyle

11/17/2003

Date

Abstract

The main purpose of this thesis is to develop an efficient near-lossless bitonal compression algorithm and to implement that algorithm on a hardware platform. The current methods for compression of bitonal images include the JBIG and JBIG2 algorithms, however both JBIG and JBIG2 have their disadvantages. Both of these algorithms are covered by patents filed by IBM, making them costly to implement commercially. Also, JBIG only provides means for lossless compression while JBIG2 provides lossy methods only for document-type images. For these reasons a new method for introducing loss and controlling this loss to sustain quality is developed.

The lossless bitonal image compression algorithm used for this thesis is called Block Arithmetic Coder for Image Compression (BACIC), which can efficiently compress bitonal images. In this thesis, loss is introduced for cases where better compression efficiency is needed. However, introducing loss in bitonal images is especially difficult, because pixels undergo such a drastic change, either from white to black or black to white. Such pixel flipping introduces salt and pepper noise, which can be very distracting when viewing an image. Two methods are used in combination to control the visual distortion introduced into the image. The first is to keep track of the error created by the flipping of pixels, and using this error to decide whether flipping another pixel will cause the visual distortion to exceed a predefined threshold. The second method is region of interest consideration. In this method, lower loss or no loss is introduced into the important parts of an image, and higher loss is introduced into the less important parts. This allows for a good quality image while increasing the compression efficiency. Also, the ability of BACIC to compress grayscale images is studied and BACICm, a multiplanar BACIC algorithm, is created.

A hardware implementation of the BACIC lossless bitonal image compression algorithm is also designed. The hardware implementation is done using VHDL targeting a Xilinx FPGA, which is very useful, because of its flexibility. The programmed FPGA could be included in a product of the facsimile or printing industry to handle the compression or decompression internal to the unit, giving it an advantage in the marketplace.

Acknowledgements

The author would like to acknowledge the members of the thesis committee for their guidance and assistance with the project: Dr. Andreas Savakis, Dr. Ken Hsu, and Dr. Marcin Lukowiak, and also family and friends for their support.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Image compression has been and continues to be a growing area of interest in the image processing field. As the capabilities of digital cameras, scanners and other optical recording devices improve and the recording resolution of these devices increase, the amount of disk storage space needed to store images increases. Therefore, a need for image compression exists.

There are two main categories of compression algorithms used today: lossless and lossy. A lossless compression algorithm is one in which, after the image has been decompressed each pixel value of the recovered image is identical to that of the original image [25-27]. When the application requires compression efficiency that is greater than what can be provided by lossless image compression algorithms, lossy image compression algorithms are used [28-31] [38]. A lossy compression algorithm is one in which the image data of the decoded image is not identical to that of the source. Near-lossless algorithms typically introduce loss in such a way that the compression efficiency increases sufficiently, but the amount of loss in the image is not noticeable by the human eye or is noticeable to an acceptable level.

Near-lossless compression algorithms continue to be a growing research interest, particularly for bitonal images. Introducing loss in a bitonal image is especially difficult. To introduce loss in a bitonal image, pixels are flipped meaning a pixel is either changed from a black pixel to a white pixel or vice versa. However, in grayscale or color images the loss can be introduced to a much lower degree, because the pixel value can be changed so that the difference is only a few gray levels or levels of color depending on the type of image. This pixel flipping introduces salt and paper noise on the bitonal image, greatly decreasing the visual quality if distortion control measures are not taken.

Compression algorithms can be very complex and, therefore, be costly when considering execution time, making it beneficial to implement the algorithm on a hardware platform [24] [32-36]. Also, bitonal images are typically used in faxing and printing, so implementing the compression in hardware would make it possible for a chip to be included in a printer or fax

machine to perform the decompression rather than requiring the printer or fax machine to be attached to a computer to handle the decompression.

The compression methods and hardware implementation developed and used in this thesis is beneficial, because it provides increased compression for bitonal images as well as the ability for the images to be compressed on a hardware chip. This would become useful in the faxing or printing industry. If an enormous document were going to be printed, rather than waiting for the entire lossless image to be sent to a printer, some loss could be introduced to the document. The document could then be compressed and sent to the printer where the hardware chip in the printer could decompress the image and then print it out saving valuable time. The same is also true for a fax machine.

Following the introduction in Chapter 1, the rest of this thesis is organized as follows: Chapter 2 provides background information in the image compression area; Chapter 3 discusses lossless bitonal image compression algorithms; Chapter 4 deals with the loss introduction algorithms; Chapter 5 discusses the hardware implantation for this thesis; Chapter 6 presents the results from the work of this thesis and Chapter 7 suggests possibilities for future work.

Chapter 2. Image Compression Background

This chapter discusses the fundamentals needed when dealing with image compression. It discusses terminology used throughout the image compression field, coding methods used to compress data and image halftoning methods.

2.1 Entropy

When performing image compression, the redundancy in the data is removed. The more redundancy that is removed from the image the more efficient the compression. For example, for a bitonal image, i.e. an image consisting of only black and white pixels, of size NxN, normally N^2 bits would be needed to store the image. However, if the image consisted only of ones, then it could be stored with only one bit, reducing all data redundancy.

Entropy is a well known metric, which defines the amount of information in a bit stream and represents the lower limit to the amount of bits that can be used with full data recovery. The equation for entropy, H, can be found below.

$$H = -\sum_{j=1}^{J} P(a_j) \log P(a_j) \qquad \textit{(Eqn. 2.1)}$$

where $P(a_j)$, is the probability of symbol a_j. The derivation of this equation can be seen below [5]. A quantity named self-information is the amount of information stored in an event E, and is shown below.

$$I(E) = \log \frac{1}{P(E)} = -\log P(E) \qquad \textit{(Eqn. 2.2)}$$

The above equation states that the amount of self-information stored in E, $I(E)$, is inversely proportional to the probability of E. If $P(E)$=1, meaning that the event is certain, then $I(E)$=0 and no information is stored in E. This is because, if it is known that the event will occur, then there is no uncertainty, so if it is communicated that the event has occurred, then no information has actually been transferred.

The base of the logarithms in Eqn. 2.2 determines the units that measure the information. If base-2 is used, then the resulting information is measured in bits, and for the rest of this thesis,

base-2 is used for all logarithms unless otherwise specified. Therefore, if the probability of an event occurring is ½, then the information stored in that event is $I(E) = -\log_2(½) = 1$ bit. When one of two possible events occurs, each with equal probability, then 1 bit of information is conveyed.

The set of symbols that are transmitted between a sender and receiver is called a source alphabet, for example A = $\{a_1, a_2, ..., a_J\}$. The probability that the information source will produce symbol a_j is defined as $P(a_j)$. Because the source alphabet is finite, the following equation holds true.

$$\sum_{j=1}^{J} P(a_j) = 1 \qquad \textit{(Eqn. 2.3)}$$

A vector z is used to represent the set of all source symbol probabilities, such that $z = \{P(a_1), P(a_2), ..., (P(a_J)\}$. The information source can be described completely using the finite ensemble (A,z). If k source symbols constitute the symbol alphabet, then according to the law of large numbers, for a sufficiently large value of k, symbol a_j will, on average, be output $kP(a_j)$ times. Therefore, the information obtained from observing k outputs is

$$I_{avg} = - kP(a_1)\log(P(a_1)) - kP(a_2)\log(P(a_2)) - \ldots \quad kP(a_J)\log(P(a_J))$$

or

$$-k\sum_{j=1}^{J} P(a_j)\log(P(a_j)) \qquad \textit{(Eqn. 2.4)}$$

Therefore, the amount of information in each source output, entropy, denoted as $H(z)$ is

$$H(z) = -\sum_{j=1}^{J} P(a_j)\log(P(a_j)) \qquad \textit{(Eqn. 2.5)}$$

For example, if an alphabet consists of three symbols a_1, a_2 and a_3, with probabilities 0.2, 0.3 and 0.5 respectively, then the entropy can be found using Eqn. 2.5 as shown below.

$$H(z) = -\sum_{j=1}^{J} P(a_j)\log(P(a_j)) = -(0.2*\log(0.2) + 0.3*\log(0.3) + 0.5*\log(0.5)) = .4472 bits$$

2.2 Image Encoding Techniques

There are three basic steps for image compression: mapping, quantization and encoding. In the mapping and quantization steps the pixels in the image are mapped to symbols that are encoded

in the encoding step. The quantization step is used for lossy compression, where similar pixels can be mapped to the same symbol. The mapping and quantization step decrease the size of the data by reducing the interpixel redundancies. For example, for an 8-bit image, if the image consists entirely of zeros, then only one symbol is needed to represent the image.

Several different types of encoders exist to compress image data, such as run-length coding [17], Huffman coding [7], arithmetic coding [17] and transform coding [17]. Some of these techniques are discussed in the following sections.

2.2.1 Huffman Coding

In 1952, D.A. Huffman developed a coding technique, which was later named Huffman coding. Huffman coding produces the shortest possible average code length given the source symbol set and the corresponding probabilities, only if the probabilities are exact powers of 2. The Huffman algorithm can be broken up into 3 steps:

1. Order the symbols according to their probabilities.
2. Combine the two smallest probabilities into one to form a parent node by adding the two probabilities together.
3. Repeat step 2 until there is only one root node with a value of 1.0

Each of the nodes are then given a value by traversing the binary tree and at each split appending a '1' or a '0' to the codeword depending on which path was taken. The algorithm can be best described by example. The example shows the codeword construction process of Huffman coding for the symbols and probabilities shown in the table below.

Symbol	Probability
a	0.05
b	0.05
c	0.2
d	0.25
e	0.45

Table 2.1 – Huffman Coding Symbol Probabilities

The figure below shows the binary tree created when doing the Huffman codeword construction using the probabilities shown in Table 2.1.

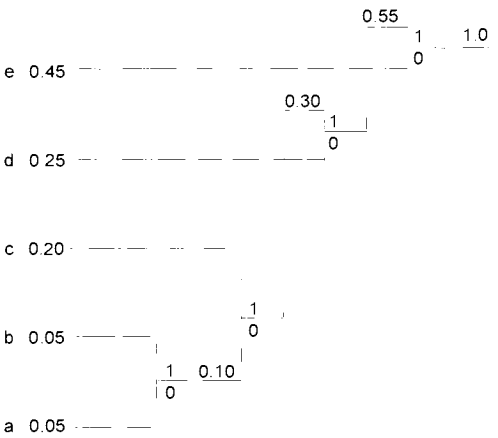


Figure 2.1 – Huffman Binary Tree

To create this tree first the symbols were listed in order of decreasing probabilities as can be seen along the left side. Next the two smallest probabilities, 0.05 and 0.05 were combined by adding them together and forming the 0.10 node. The probabilities were then reordered by decreasing probabilities and the next two smallest probabilities, 0.10 and 0.20, were combined to form the 0.30 node. The process repeats until the 1.0 root node was formed. Then at each combination a '1' was assigned to the top child node and a 0 to the bottom child node as can be seen in Figure 2.1. The codewords for each symbol are then found by traversing the tree from the root node to the corresponding child node and appending either the '0' or '1' digit depending on which path was taken. The table below shows each symbol and its corresponding codeword.

Symbol	Codeword
a	1100
b	1101
c	111
d	10
e	0

Table 2.2 – Huffman Coding Codewords

As can be seen in Tables 2.1-2.2 the symbols with the highest probabilities were given the smallest codewords and the symbols with the lowest probabilities were given the larger symbols, which provides efficient compression.

Huffman encoding is a very useful compression method when compressing data in which the probabilities are all integer powers of two. However one disadvantage is the decrease in compression efficiency as the difference between the probabilities and the integer powers of two increases.

2.2.2 Arithmetic Coding

An arithmetic coder is another means of reducing the redundancy in data. An arithmetic coder works by encoding a group of symbols as a real number in the range of 0 to 1. Several arithmetic coder implementations exist such as LZAP [8], the Lee and Park algorithm [9], the Z-coder [10], the IBM QM-coder[11], the IBM Qx-coder[12] and the Block Arithmetic Coder for Image Compression [1], which is used for this thesis.

There are two basic pieces of information that are needed for arithmetic coding: the probability of a symbol and its encoding interval range. Any number of symbols can be encoded using one real number as long as the precision of the real numbers is good enough. The example below illustrates how arithmetic coding works.

For this example, consider a system with a four-symbol alphabet as shown in the table below with their probabilities.

Symbol	Probability
a	0.1
b	0.2
c	0.3
d	0.4

Table 2.3 – Arithmetic Coding Example – Symbol Probabilities

The number of symbols that are going to be encoded in each codeword needs to predetermined and known by both the encoder and decoder. For this example, four symbols are encoded in each codeword. The encoding of the message "accd" can be seen below.

First each symbol is given a range according to its probability. The range starts from 0 to 1, so symbol 'a' is given the range 0 to 0.1, symbol 'b' is given the range 0.1 to 0.3. symbol 'c' is given the range 0.3 to 0.6 and symbol 'd' is given the range 0.6 to 1.0, as illustrated in Figure 2.2.

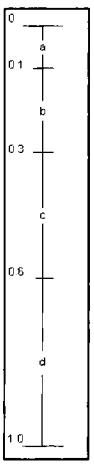


Figure 2.2 – Arithmetic Coding Example – Step 1

Because the first symbol in the message is 'a', the interval for the second step becomes 0 to 0.1. The range is then broken up again for each symbol using its probability. The range for symbol 'a' is now 0 to 0.01, the range for symbol 'b' is now 0.01 to 0.03, the range for symbol 'c' is now 0.04 to 0.06 and the range for symbol 'd' is now 0.06 to 1.0, as shown in Figure 2.3.

The next symbol in the message is 'c', so the interval now becomes 0.03 to 0.06. The same process is done two more times to encode the third and fourth symbols as can be seen in Figure 2.4. After the encoding of the third symbol 'c' the range becomes 0.039 to 0.048 and the range is again broken up into subdivisions. The last symbol in the message is 'd', so the message "accd" can be encoded as any real number between 0.0444 and 0.0480.

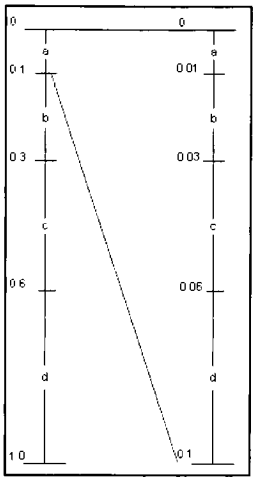


Figure 2.3 – Arithmetic Coding Example – Step 2

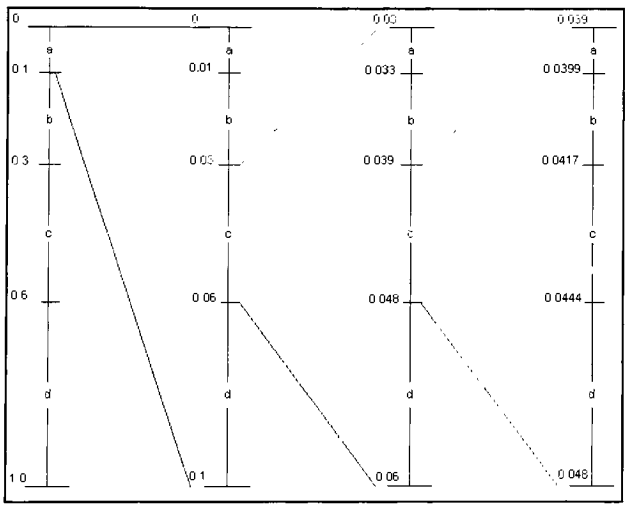


Figure 2.4 – Arithmetic Coding Example – Final Steps

There are problems inherent in arithmetic coding which are important to examine before attempting to implement an arithmetic coder. Since no machine can have infinite precision, underflow and overflow can be an issue. If insufficient precision is used considering the number of symbols being encoded in one message, then two symbols could potentially have ranges that overlap. Another setback of arithmetic coding is that each codeword is a real number in the range of 0 to 1, so decoding of the message cannot begin until the decoder

receives the entire codeword. Also, arithmetic coding is an error sensitive compression scheme, meaning a single bit error can corrupt the entire message.

There are two basic types of arithmetic coding: static and adaptive. In static arithmetic coding the probabilities of each of the symbols do not change. With adaptive arithmetic coding, the symbol probabilities are estimated during each step of the encoding process based on the changing symbol frequencies that have been seen in the previously encoded messages. Because in the real world the exact probabilities are impossible to produce, it cannot be expected that an arithmetic coder will achieve maximum efficiency when compressing a message. The best an arithmetic coder can do is to estimate the probabilities on the fly.

2.3 Image Halftoning

This thesis deals primarily with the compression of bitonal images. Several algorithms exist for obtaining a bitonal image from a grayscale image. The main goal of halftoning an image is to preserve the overall grayscale look of the image while changing each pixel from an 8-bit value (could also be 4-bit or higher than 8-bit) into a 1-bit bitonal value.

2.3.1 Ordered Dithering

There are several methods for converting an 8-bit grayscale image into a 1-bit bitonal image. Ordered dithering is one of the simplest methods for doing this and involves using a dither mask [5]. This rectangular mask can be any size, but is usually square. The mask consists of threshold values between 0 and 255, for an 8-bit image. The mask is then moved over the entire image and each pixel is then compared with its corresponding value in the dither mask using the equation below.

$$u_{i,j}' = \begin{cases} 0 & \text{if } u_{i,j} > m_{i,j} \\ 1 & \text{if } u_{i,j} \le m_{i,j} \end{cases} \qquad \textit{(Eqn. 2.6)}$$

where u_{ij} is the original pixel at location *(i,j)* with respect to the dither mask, m_{ij} is the value in the dither mask at position *(i,j)* and u_{ij}' is the new value of the pixel. The reason for the inverting of the bitonal pixel is that for grayscale images a black pixel has a value of 0 and a

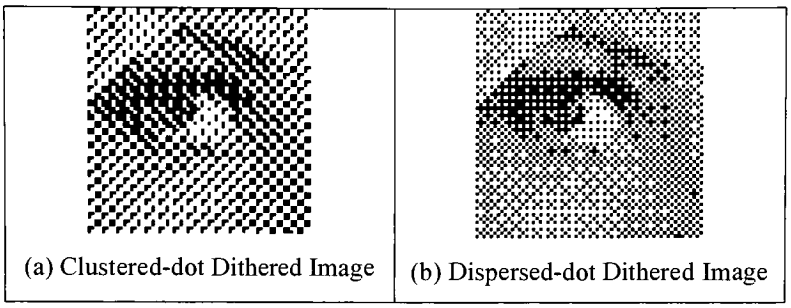
white pixel has a value of 255 (for 8-bit), while in a bitonal image a black pixel has a value of 1 and a white pixel has a value of 0. The dither mask is then moved around the entire image until all pixels have been halftoned.

Another thing to consider when performing an ordered dither is what type of dither mask to use. There are two basic types of dither masks: dispersed-dot and clustered-dot. A dispersed-dot mask would leave the black pixels spread out and not bunched together, while a clustered-dot mask would keep the black pixels grouped together and the white pixels clustered together. Dispersed-dot ordered dithers are designed to yield halftones, which convey the impression of gray. Clustered-dot ordered dithers are designed to yield halftones with few isolated pixels and are typically used with printing devices, which cannot properly display isolated pixels in an image. Figure 2.5 shows halftoned images using the two different types of dither masks, both masks are 4x4 pixels.

(a) Clustered-dot Dithered Image | (b) Dispersed-dot Dithered Image

Figure 2.5 – Dithered Images

As can been in the above figure at high resolution it is sometimes difficult to tell the difference between clustered-dot and dispersed-dot although differences can be seen. The figure below shows the three figures above except zoomed in on a specific area of the image to show the actual differences in the methods.



(a) Clustered-dot Dithered Image (b) Dispersed-dot Dithered Image

Figure 2.6 – Low Resolution Dithered Images

As can be seen when halftoning a low-resolution picture, such as the ones shown above in Figure 2.6, the visual quality decreases significantly. Even at this resolution the dispersed-dot dither still maintains a small illusion of grayscale, but the clustered-dot image has poor quality. As expected on the clustered-dot image the black pixels are clustered together.

2.3.2 Error Diffusion

Another method of halftoning is error diffusion [16]. As the name implies, in this method the error created by the thresholding is diffused to increase the quality of the halftone. The diagram below illustrates the error diffusion algorithm.

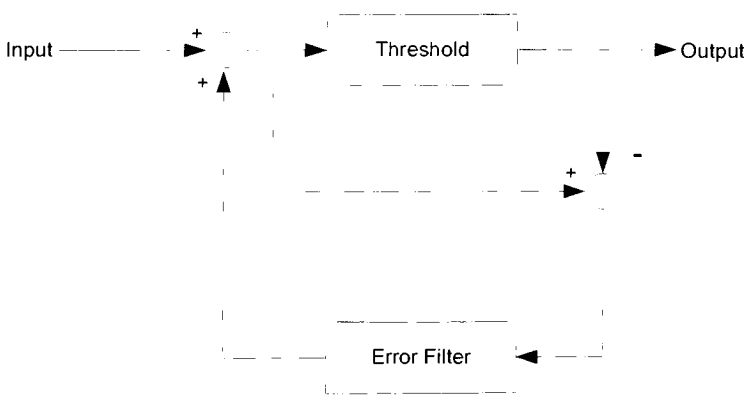


Figure 2.7- Error Diffusion Algorithm

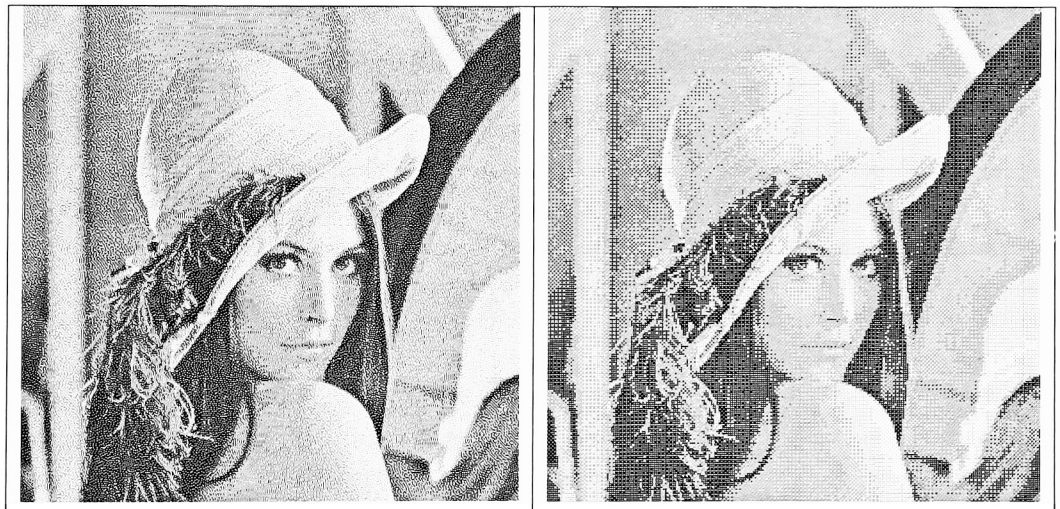
The algorithm works by traversing the image pixel by pixel, in raster scan order, and thresholding each pixel, usually with a value of 128 (for 8-bit images). The error for that pixel is then calculated, by either subtracting the pixel value from 255 or subtracting the pixel value

from 0, depending on the color of the corresponding bitonal pixel. This error is then diffused using the Floyd-Steinberg weights [16], shown in the figure below.

	x	7/16
3/16	5/16	1/16

Figure 2.8 – Floyd-Steinberg Weights

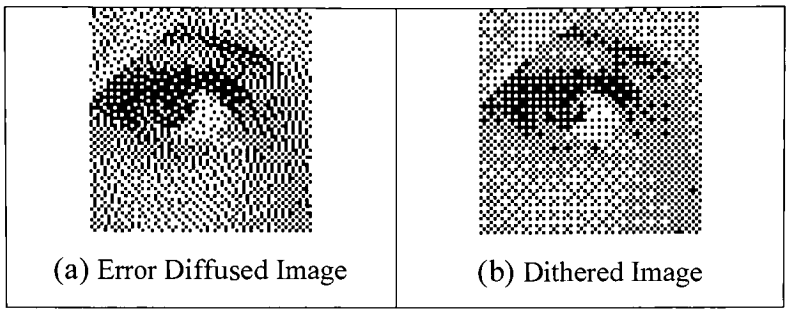
The 'x' in the above figure is the current pixel. The error created by halftoning the current pixel is then diffused to the neighboring pixels by multiplying the error by the value shown in Figure 2.8 and then adding that value to the grayscale value for that pixel. As can be seen in the above figure the Floyd-Steinberg weights only affect pixels that have not yet been processed. The figure below shows an image that has been halftoned using error diffusion.



(a) Error Diffused Image (b) Dithered Image

Figure 2.9 – Error Diffused and Dithered Images

As can be seen in the above figure, the overall grayscale look of the error diffused image is impressive. The error diffused image also looks better than the dispersed-dot dithered image, because it doesn't have the grid lines that show up in the dithered image. Below shows a low-resolution version of the above images by zooming in on a specific area.



(a) Error Diffused Image (b) Dithered Image

Figure 2.10 – Low Resolution Error Diffused and Dithered Images

As can be seen in the low resolution images in Figure 2.10, the error diffused image still has much better quality than the dithered image. Obvious differences now appear between the error diffused and the original image, but unlike the dispersed-dot dithered image the object in the image is easily recognizable.

2.4 Lossy Compression

Digital image compression algorithms can be classified into two categories: lossless and lossy. A lossless compression algorithm is one in which after the image has been decompressed each pixel value of the recovered image is identical to that of the original image. A lossy compression algorithm is one in which the image data of the decoded image is not identical to that of the source.

The goal of a lossy compression algorithm is to improve on the compression ratio while keeping the visual distortion to an acceptable level. Compression ratio is defined using the equation shown below.

$$Compression\ Ratio = \frac{Original\ image\ size}{Compressed\ image\ size} \qquad (Eqn.\ 2.7)$$

The level of acceptable visual distortion depends on the application. There are several metrics that can be used to measure the distortion, such as root mean-squared-error (RMSE) and signal-to-noise ratio (SNR). In image coding the peak signal-to-noise ratio (PSNR) is more commonly used in place of the signal-to-noise ratio [5].

The RMSE is defined using the equation shown below.

$$RMSE = \sqrt{\frac{1}{N}\frac{1}{M}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}(x_{i,j} - y_{i,j})^2} \qquad \textit{(Eqn. 2.8)}$$

where $x_{i,j}$ is the pixel in the original image at location (i,j), $y_{i,j}$ is the pixel in the compressed image at location (i,j), N is the height of the image and M is the width of the image.

The peak signal-to-noise ratio is defined in the equation below.

$$PSNR = 20\log_{10}\frac{255}{RMSE} \qquad \textit{(Eqn. 2.9)}$$

Because the focus of this thesis is bitonal image compression, the PSNR metric alone is not very useful. The reason is that each pixel can consist only of a 0 or 1, so the only error that can exist is 1. Another choice is to use 0 and 255 as the pixel values, then the error would be 255. However, this is a large error amount and could be misleading in that for only one pixel error in a region, the value of the error could be rather large. To better approximate the perceived grayscale error a metric called the modified peak signal-to-noise ratio (MPSNR) is used [21]. For this metric, the inverse halftoning of the bitonal image is done first, and then the PSNR is taken. In order to do the inverse halftoning a Gaussian Low Pass Filter is convolved with the original bitonal image, as can be seen in the equation below.

$$G = \frac{1}{100}\begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 16 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix} \qquad \textit{(Eqn. 2.10)}$$

$$H_{low} = H * G \qquad \textit{(Eqn. 2.11)}$$

Using Equations 2.10-2.11, the inverse halftone, H_{low}, of H is found. The MPSNR is then found using Equation 2.9, using the original grayscale image to do the comparison. The unit for Equation 2.9 is decibels (dB).

The images in Figure 2.11 show the values of the MPSNR for different levels of noise added to the Lena image. Image 2.11(a) is the lossless grayscale image. Image 2.11(b) is the lossless error diffused Lena image. Images 2.11(c) through 2.11(f) are the error-diffused image with increasing amounts of salt and pepper noise added. As can be seen in the figure, as the amount of noise increases, the MPSNR decreases, as expected.

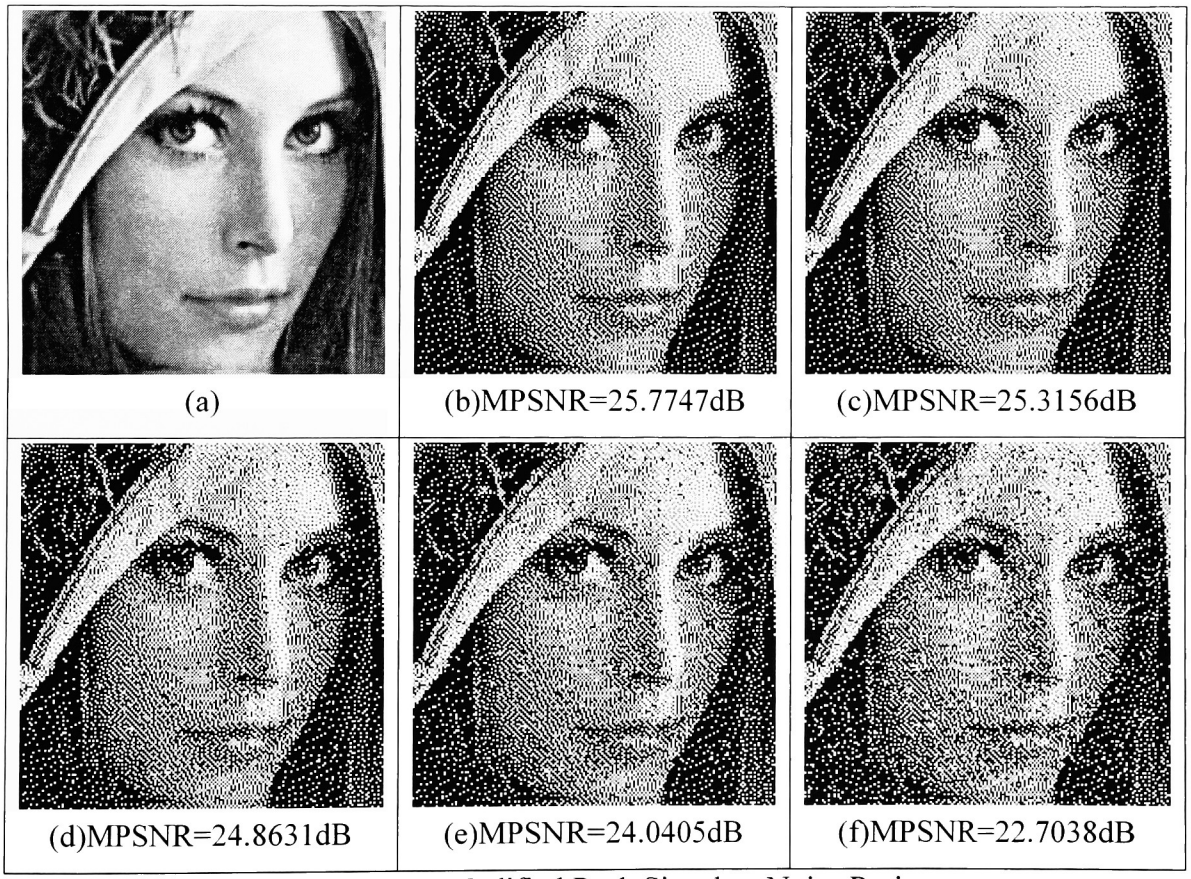


Figure 2.11 – Modified Peak Signal-to-Noise Ratio

Chapter 3. Image Compression Algorithms

There are several algorithms and standards that exist today for the compression of images. As was stated above, the goal of lossless compression algorithms is to reach a compression efficiency equal to the entropy. This chapter discusses several bitonal image compression standards and algorithms, one of which is used for this thesis. The other compression algorithms are discussed, because they show other means for bitonal image compression. The bitonal compression algorithm used for this thesis improves upon the deficiencies of those algorithms.

3.1 CCITT Group 3 Compression Standard

The Consultative Committee on Telephone and Telegraph (CCITT) developed the Group 3 Fax Algorithm (G3) in 1980 [18]. The G3 compression standard was developed for digital facsimile transmission on the public switched telephone network. The main goal of this bitonal compression algorithm was to be able to compress a document scanned at 100 dots per inch (dpi) and sampled at 1728 samples per line to be transferred at 4800 bits per second (bps) in an average time of about one minute. This means that in order to transmit a standard 8.5x11 inch page in the time needed, a compression ratio of about 6.6 is needed.

The G3 algorithm works by counting the size of blocks of either all ones or all zeros and encoding this count. This method takes advantage of the tendency of textual documents to include a lot of white space, because between each letter and between lines of a textual document is white space. There are two alternate coding schemes in the G3 algorithm: CCITT Group 3 one-dimensional and CCITT Group 3 two-dimensional.

3.1.1 CCITT Group 3 One-Dimensional Compression standard

The CCITT Group 3 one-dimensional coding scheme encodes an image as a series of variable length codewords, so that each codeword represents a horizontal block of either all white or all black pixels. The codewords used are the Modified Huffman (MH) code. The Modified Huffman code has two types of codewords: Terminating codewords and Make-up codewords. Terminating codewords are used to encode smaller blocks of black or white pixels, in the range

of 0 to 63. Make-up codewords are used with terminating codewords to encode blocks larger than 63 pixels. These make-up codewords can handle pixel blocks of size 64 to 1728. For run lengths larger than 1728 an optional set of make-up codewords can be used which can encode run lengths up to 2560 pixels, which would be used when transmitting documents at a higher resolution.

An End-Of-Line (EOL) codeword exists to signify either the start of an image or the end of a line. At the end of an image a special codeword is used called Return To Control (RTC) which consists of 6 EOL codewords. The flow diagram shown in Figure 3.1 illustrates the one-dimensional Group 3 compression standard.

As can be seen in the flow diagram, the algorithm starts by inserting an EOL character, as a rule with this algorithm, an EOL must exist before the encoding of the first line. Then the first run is counted and encoded accordingly. There are three tables that are used for this algorithm: Terminating codes table (T Table), Make-up codes table (M Table) and Additional Make-up codes table (AM Table). The T Table consists of 128 codewords for all different run-lengths, white or black, from 0 to 63 pixels. Therefore, if a run exists of length 0 to 63 it can be encoded with just one codeword from this table. The M Table consists of 55 codewords. It has a codeword for multiples of 64 from 64 to 1728 and an extra one defining EOL. So, if a run length is in the range of 64 to 1728, it is encoded as a codeword from the M Table and a codeword from the T Table. Finally, the AM Table is used to handle runs of length 1792 to 2560. It is just an extension of the M Table, so if a run were of length 1792 to 2560, it would be encoded using a codeword from the AM Table and a codeword from the T Table.

For example, if a run of 40 black pixels exists, then the T Table is used to find the codeword for a run of 40 black pixels (000001101100). So "000001101100" is the encoding of 40 black pixels. If a run of 100 white pixels exists first the M table would be used, using the codeword for 64 white pixels (11011), since 64 is the biggest multiple of 64 less than or equal to 100. Then the T Table is used to find the codeword for 36 white pixels (00010101), because 100 – 64 = 36. So the encoding of 100 white pixels would be "11011 00010101".

Looking at these tables, which can be found in [18], it can be seen that the encoding of white pixels has smaller codewords than for black pixels. This is because the G3 compression standard was made primarily for facsimile transmission. Facsimile transmission consists mostly of textual documents, which have a lot of white space in them. This allows for textual documents to compress efficiently.

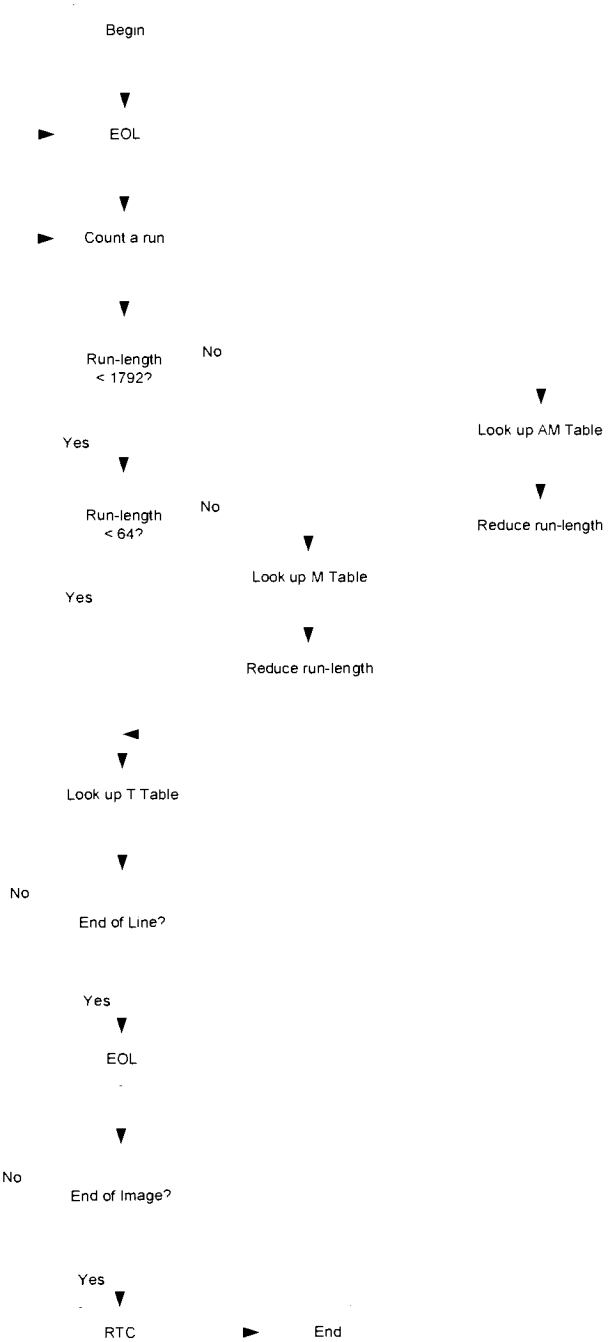


Figure 3.1 – Flow Chart for One-Dimensional G3 Compression standard [17]

3.1.2 CCITT Group 3 Two-Dimensional Compression Standard

The problem with the one-dimensional Group 3 algorithm is that it only takes advantage of runs in the horizontal direction. In a textual document runs exists in both the horizontal and vertical directions. Therefore, in order to get even greater compression efficiency the algorithm must take advantage of the vertical runs as well. The two-dimensional Group 3 algorithm explores correlation of pixels in the vertical direction as well as the horizontal direction [18].

Both the one-dimensional and two-dimensional Group 3 algorithms use a line-by-line coding method. In the two-dimensional algorithm, a reference element is used which determines the mode in which the current group of pixels will be coded. The reference element can either exist on the line currently being coded, called the coding line, or the previous line, called the reference line. After the entire coding line has been processed, it becomes the reference line and the next line down becomes the coding line. However, one problem with coding a line while taking into account the previous line is that if there is a transmission error on one line, then all the lines after that can be wrong. To avoid this problem the two-dimensional algorithm periodically sends a line using the one-dimensional G3 algorithm. This period is known as the K factor. K can be any positive integer. In other words for every K lines, 1 of them is encoded using the one-dimensional algorithm, and K-1 of them are encoded using the two-dimensional algorithm.

There are five pixels used to determine which mode and how each group of pixels is encoded which are listed in Table 3.1.

Changing Element	**Definition**
a_0	The reference element on the coding line.
a_1	The next element on the coding line to the right of a_0 and the opposite color of a_0.
a_2	The next element on the coding line to the right of a_1 and the opposite color of a_1.
b_1	The next element on the reference line to the right of a_0 and the opposite color of a_0.
b_2	The next element on the reference line to the right of b_1 and the opposite color of b_1.

Table 3.1 – Changing Elements for 2D G3 Algorithm

The element a_0 is set from the previous iteration of the algorithm, and at the beginning of encoding it is set to an imaginary white pixel before the first pixel of the image. The example in Figure 3.2 shows the placement of the 4 changing elements with respect to a_0.

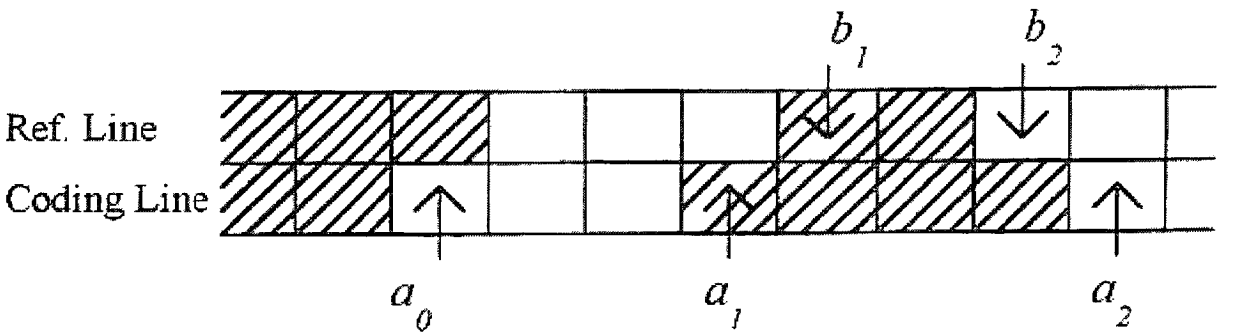


Figure 3.2 – Changing Element Placement [17]

There are three coding modes used in the two-dimensional G3 algorithm that are determined by the position of the changing elements shown in Figure 3.2: Pass Mode, Vertical Mode and Horizontal Mode. Pass Mode is used when the position of b_2 lies to the left of a_1. Vertical mode is used when the relative distance between a_1 and b_1, meaning the distance between a_1 and b_1 if they were on the same line, is less than or equal to 3. Horizontal mode is used when neither of the other two modes applies.

Each of the modes described above gets a set of distinct codewords. For Pass Mode there is only one codeword that is ever used 0001. The coding of the location of the other changing elements is not needed. For Horizontal Mode the codeword is $001+M(a_0a_1)+M(a_1a_2)$, where a_ia_j is the length and color of the run between a_i and a_j. For example in Figure 3.2, a_0a_1 is a white run of 3. $M(a_ia_j)$ is the codeword used in the one-dimensional G3 algorithm found in the M Table. There are 7 cases for the vertical mode, depending on where a_1 is with respect to b_1, each with its own codeword. Table 3.2 describes these different cases.

Figure 3.3 shows the flow diagram for the two-dimensional G3 algorithm.

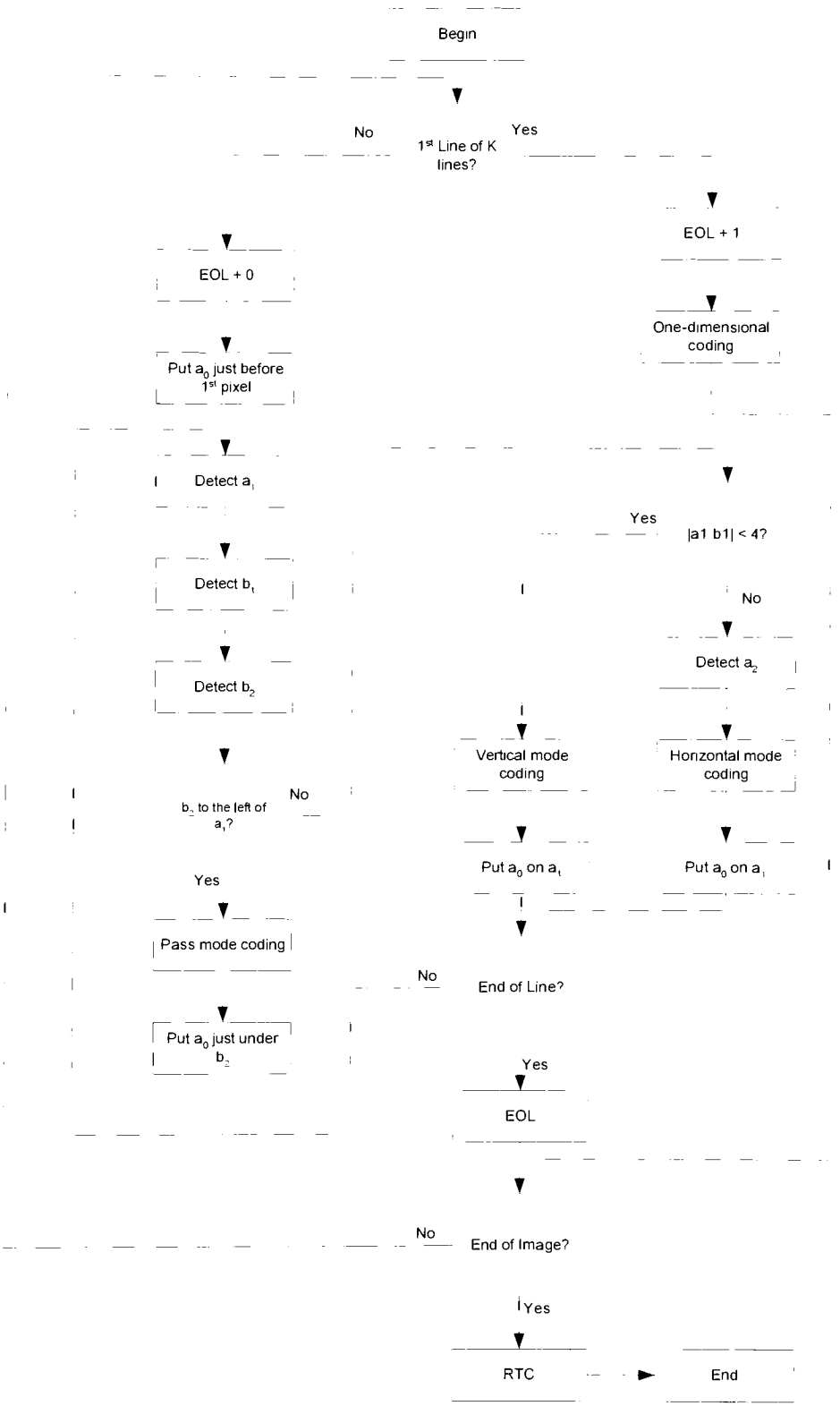


Figure 3.3 – Flow Diagram for Two-Dimensional G3 Algorithm [17]

Case	Description	Codeword
V(0)	a_1 is just under b_1	1
$V_R(1)$	a_1 is one pixel to the right of b_1	011
$V_R(2)$	a_1 is two pixels to the right of b_1	000011
$V_R(3)$	a_1 is three pixels to the right of b_1	0000011
$V_L(1)$	a_1 is one pixel to the left of b_1	010
$V_L(2)$	a_1 is two pixels to the left of b_1	000010
$V_L(3)$	a_1 is three pixels to the left of b_1	0000010

Table 3.2 – Vertical Mode Case Description

As shown in Figure 3.3, the algorithm starts by checking to see if the current line is the first of K lines, and if it is, it does one-dimensional coding. If it is not, then a_0 is placed just before the first pixel and a_1, b_1 and b_2 are found. After those three changing elements are found, the mode of coding, is found and the pixel group is encoded accordingly. Then a0 is repositioned and the whole process starts over again with the detection of a_1, b_1 and b_2. After the entire line has been coded, the algorithm checks to see if the image is done, if not, it starts over again with the checking to see if this is the first of K lines.

3.2 CCITT Group 4 Compression Standard

In 1984, the CCITT Group 4 (G4) compression standard was developed as a facsimile coding scheme for the compression of black and white images [19]. The standard is designed strictly for error-free digital facsimile transmission, and it assumes that all error correction is already handled on a lower level of the communication process.

The Group 4 standard works very similar to the two-dimensional G3 standard. The main difference between the two standards is that the two-dimensional G3 algorithm periodically encodes entire lines using the one-dimensional G3 method, but the G4 algorithm encodes all lines using the two-dimensional G3 algorithm. When the G4 algorithm begins, it inserts an imaginary white line above the first line of the image to be the first reference line.

The main advantage of this method over the two-dimensional G3 algorithm is the improved compression efficiency. The reason for this better efficiency is that the two-dimensional G3

algorithm does one-dimensional encoding every K lines, and one-dimensional coding lowers the compression efficiency, because it does not take into account runs in the vertical direction. However, robustness is a major problem with the G4 standard. The G4 standard assumes that the medium across which the data is sent is completely error free. However, if there is a bit transmission error, corruption of the entire image can occur. Therefore, the usability of the G4 compression standard lies primarily in the medium used to send the data.

3.3 JBIG

The Joint Bilevel Image Experts Group (JBIG) standard is a lossless image compression standard developed for bitonal images [20]. One major advantage of the JBIG standard is that it is capable of doing progressive or sequential encoding of an image and progressive or sequential decoding of an image independent of how the image was encoded.

Sequential coding is the type of coding used in the G3 and G4 algorithms. When using sequential coding an image is encoded in raster scan order. During progressive coding images are created from the original image but at lower resolutions than the original image. So, first a very low-resolution representation of the image is encoded, then a representation of the image with the next higher resolution and this continues until finally the original image is encoded. Progressive coding is very useful when transmitting a compressed image over a slow connection, because the receiver of the image can get a low resolution copy of the image and as more detail arrives decide if they would like the entire image or to cancel the transfer. Each resolution layer of the compressed image in JBIG is double that of the one below it.

One preprocessing non-trivial task that must be done is to create these lower resolution images. One option is to subsample the image by taking every other row and every other column, however this would leave a low quality representation of the image. The JBIG standard defines a method in which to acquire lower resolution copies of the image. The main purpose of the resolution reduction algorithm defined in the JBIG standard is the preservation of density by using a filter with exceptions. The exceptions for this filter are for occasionally overriding the output of the filter to preserve edges, lines, and periodic or dither patterns in the image.

The JBIG resolution reduction algorithm consists of two simple steps. The first step is to break up the image into groups of two by two blocks of pixels, padding the right or bottom side with zeros if needed. The second step consists of mapping each one of the two by two blocks of pixels into one low-resolution pixel. This mapping is done by using a table defined in the JBIG standard. Twelve surrounding high-resolution and low-resolution pixels are used as an index into the table to determine the value of the low-resolution pixel. The following equation is used to find the index for pixel X, and the figure below shows the pixel locations.

$$Index_X = \sum_{j=0}^{11} a(j)\ 2^j \qquad (Eqn.\ 3.1)$$

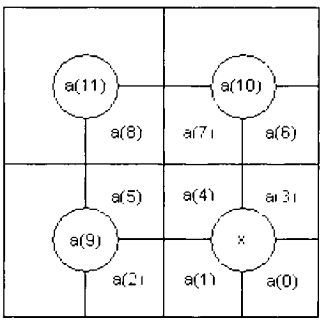


Figure 3.4 – Resolution Reduction Pixel Location

This process is done over and over until the number of resolution levels specified by the user is reached. Each group of four pixels in the higher resolution image is combined into one lower resolution pixel.

Compatibility between sequential and progressive coding is achieved by dividing the image into stripes. A stripe is a group of rows of an image, and is shown in the example below.

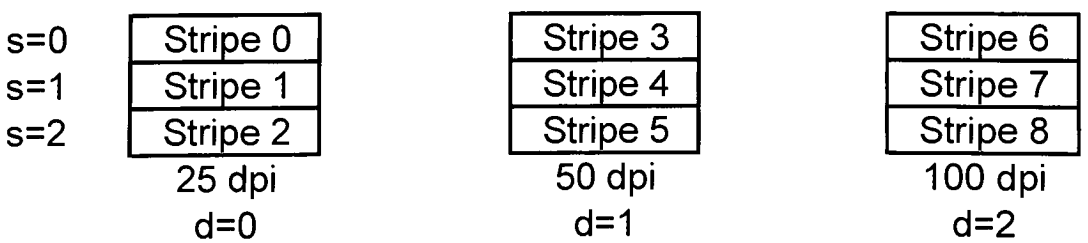


Figure 3.5 – Resolution Reduction and Data Striping

In the above example the original image has a resolution of 100 dpi, and two lower resolution images are created, the first with a resolution of 50 dpi and the second with a resolution of 25 dpi. Each of the three images is then broken up into 3 stripes as can be seen in Figure 3.5. The next step is called data ordering. From the 9 stripes in Figure 3.5, there are 4 ways they can be sorted according to their resolution layer, d, and stripe number, s. These four sorting variations can be seen in the figure below.

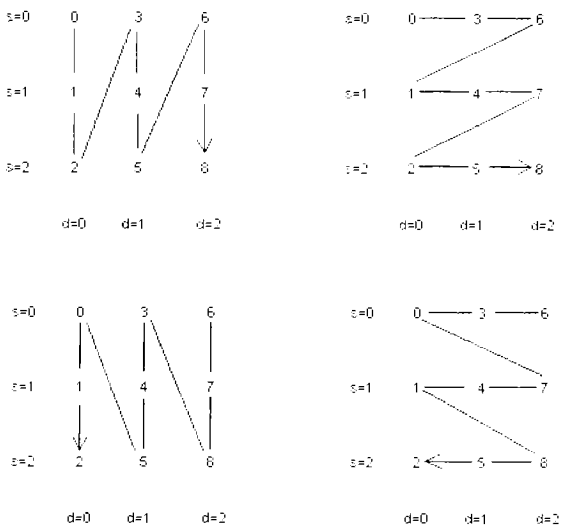
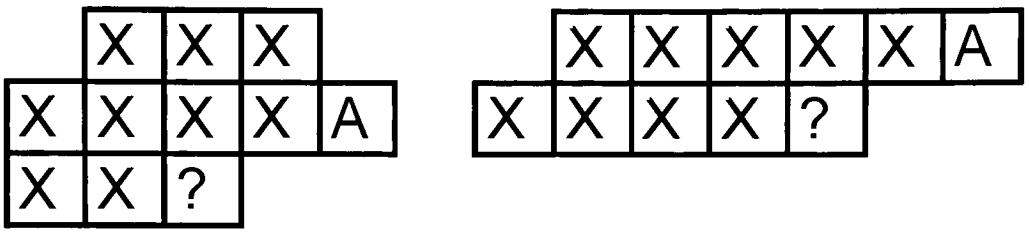


Figure 3.6 – Data Ordering

By breaking up the image into stripes, JBIG is able to provide compressed data, $C_{s,d}$, of the stripe image, $I_{s,d}$, for stripe s of resolution d, which is independent of stripe ordering. This is important, because it means that the amount of information describing an image is independent of the encoding and expected decoding method. Therefore, an image can be encoded either sequentially or progressively and then at the time of decoding it can either be decoded sequentially or progressively depending on what the application requires.

The actual compression of an image using JBIG is done using an arithmetic coder. In order to provide the arithmetic coder with the probabilities it needs, a template is used. The template is a group of neighboring pixels that provides the coder with a context, which is simply an integer. There are different templates depending on what resolution layer is being encoded, because the

higher resolution layer can include pixels of lower resolution, but the lowest resolution layer cannot include pixels from any other resolution layer. The three-line and two-line templates are shown below in Figure 3.7. These templates are used for both sequential coding and bottom-layer coding, which is the coding of the layer with the lowest resolution.



(a)Three Line Template (b)Two Line Template

Figure 3.7 – Lowest Resolution Layer Templates

In these templates the '?' represents the pixel currently being encoded. The pixels denoted by 'X' are ordinary pixels in the template, and the pixel labeled 'A' is an adaptive pixel. This pixel is special in that its location may change during the encoding process in order to adapt to the image attributes and provide for a better prediction. For example, adaptivity provides considerable gain with dithered images. The algorithm for determining when and where to move the adaptive pixel in the above templates is not defined in the JBIG standard and is instead left to the programmer. However the adaptive pixel can never be at the same spot as one of the pixels denoted by an 'X' nor can it be a casual pixel, which is one of the pixels ahead of the current pixel being encoded. This is to allow for decoding of the image. Each of the 10 pixels, both 'X' and 'A', make up a 10-bit number where each pixel is a bit of the number. This 10-bit number is the context of the pixel.

This context is then used by the arithmetic coder to produce a probability p_0 or p_1, the probability that the pixel denoted by the '?' is a 0 or 1 respectively. If the probability estimate is accurate and close to 0 or 1, then the compression ratio will be very good. The purpose of the JBIG templates is to make the value of the pixel being encoded highly predictable.

Figure 3.8 shows two of the templates that can be used for differential-layer coding, or coding in a layer other than the lowest resolution layer. Of course these templates can never be used

for sequential coding. As can be seen with these templates, there is still an adaptive pixel that is used as well as 5 ordinary pixels used in the template. The difference is that in each of these templates there is 4 pixels used from the lower resolution image. The reason that this can be done is when the decoder is processing it decodes the lower resolution layers first so the lower resolution pixels are available for the context.

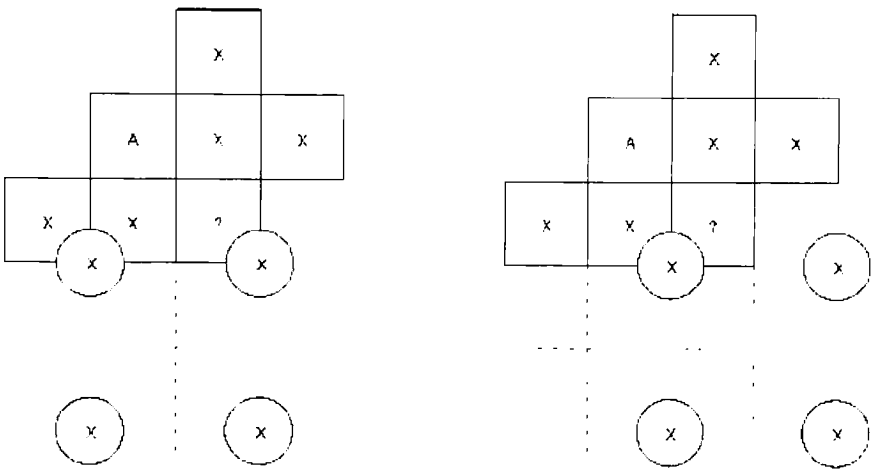


Figure 3.8 – Differential Layer Templates

Another method JBIG uses to improve compression efficiency is called prediction. In JBIG there are two types of prediction defined: typical prediction and deterministic prediction. Typical prediction can actually be broken up into two parts: bottom-layer typical prediction and differential-layer typical prediction.

Bottom-layer typical prediction is a very simple algorithm in which it checks for typical lines, which are lines that are identical to the one above it. In the case it finds a typical line, that line is not coded, therefore, this speeds up the algorithm as well as making it more efficient.

Differential-layer typical prediction works by attempting to predict the four higher resolution pixels that correspond to one low-resolution pixel. If this can be done then there is no reason to encode the four higher-resolution pixels. To do this, the algorithm searches regions of solid color. By looking at the color of the 8 low-resolution pixels surrounding the current pixel, X, the color of the 4 high-resolution pixels associated with X might be able to be determined. If

the 8 surrounding pixels have the same color as X then the 4 high-resolution pixels associated with X most likely have the same color as X. Of course, there are exceptions, but this is the general algorithm behind differential-layer typical prediction. The advantages to differential-layer typical prediction is that the encoding and decoding process can run faster, because they do not need to encode the higher resolution pixels that can be predicted. Also, the compression efficiency would be improved.

Deterministic prediction is similar to differential-layer typical prediction in that it attempts to predict the value of the 4 high resolution pixels associated with one low resolution pixel. The difference is in the way that it does this. The deterministic prediction is a table-driven algorithm. The values of certain pixels in the low-resolution image and pixels in the high resolution image are used as an index into a table to check for determinicity. Because the deterministic prediction tables depend heavily on the type of resolution reduction algorithm used, JBIG allows the user to define the deterministic prediction tables associated with a resolution reduction algorithm. The diagram below shows the pixels that are used for deterministic prediction and their position in the index.

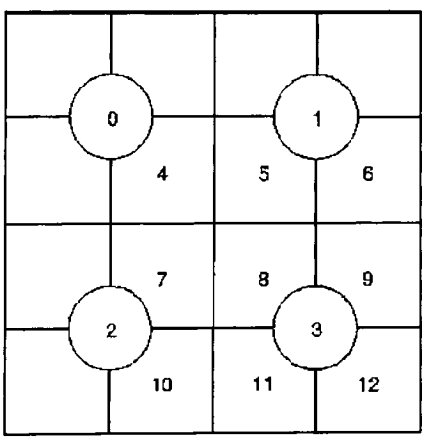


Figure 3.9 – Deterministic Prediction Pixels

As stated previously JBIG uses an arithmetic coder to do the actual encoding of the image. The block diagram below describes the encoding process using sequential coding.

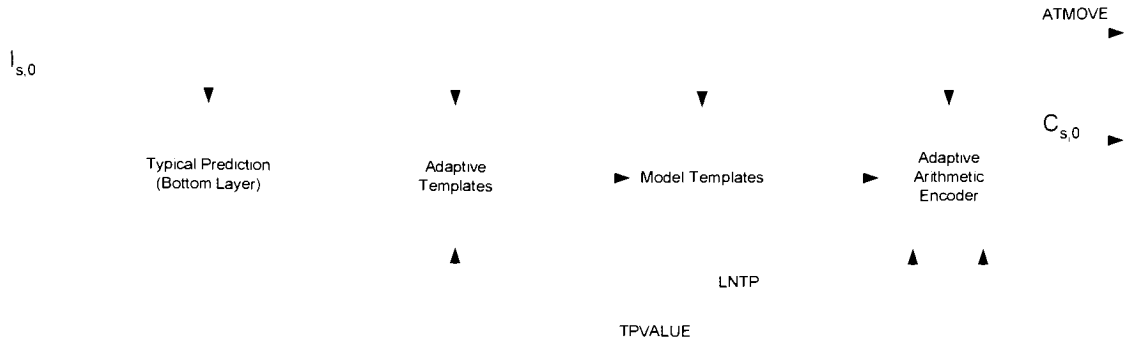


Figure 3.10 – Sequential Encoder Block Diagram

In Figure 3.10 $I_{s,0}$ is the original image data and $C_{s,0}$ is the compressed image. The first block in the diagram is the typical prediction block, which, for sequential coding, just checks to see if the previous line is equal to the current line and if so the current line is not encoded. The output of this block is TPVALUE meaning typical prediction value and the LNTP is the Line Not Typical variable. The middle two blocks are the adaptive template and model template blocks, which handle the moving of the adaptive pixel and providing the context for the encoder. The ATMOVE variable specifies when the adaptive pixel has moved so that this change can be known at the time of decoding. This way the decoder does not need to do any searching for the correct setting for adaptive templates.

Figure 3.11 shows the encoder for differential layer encoding. This encoder is used when doing progressive coding on a layer that is not the bottom layer.

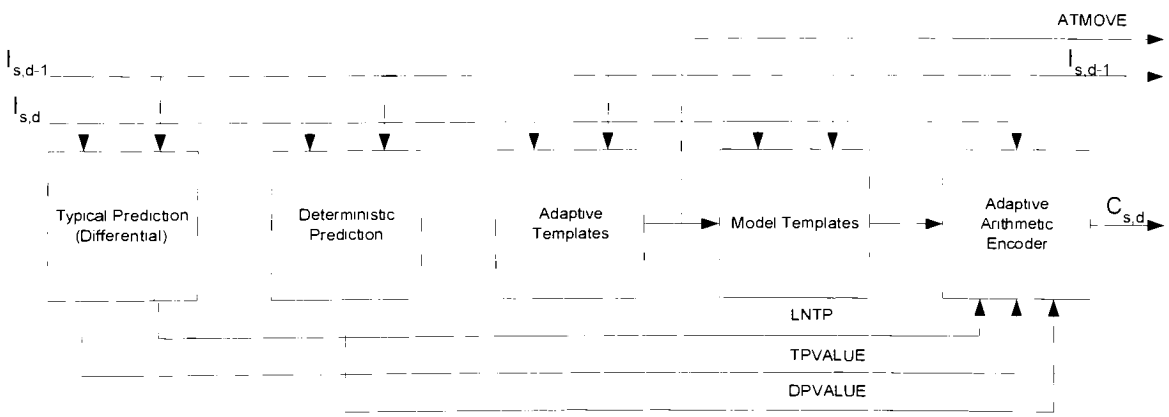


Figure 3.11 – Differential Layer Encoder

As can be seen in this block it uses both the current layer image, $I_{s,d}$, as well as the image from the previous layer, $I_{s,d-1}$, a lower resolution image. It performs both typical prediction and deterministic prediction, described above, and produces the TPVALUE and DPVALUE variables respectively. The two template functional blocks now use the templates that span the multiple layers. The three outputs of this encoder are ATMOVE which tells the decoder when an adaptive pixel was moved, $I_{s,d-1}$, which is the input image to the next stage as $I_{s,d}$, and $C_{s,d}$ which is the compression of the $I_{s,d}$ image from this stage.

The encoder for the entire system of JBIG when using progressive coding can be found below. This uses the sequential encoder and the differential encoder shown in Figures 3.10 and 3.11 as building blocks.

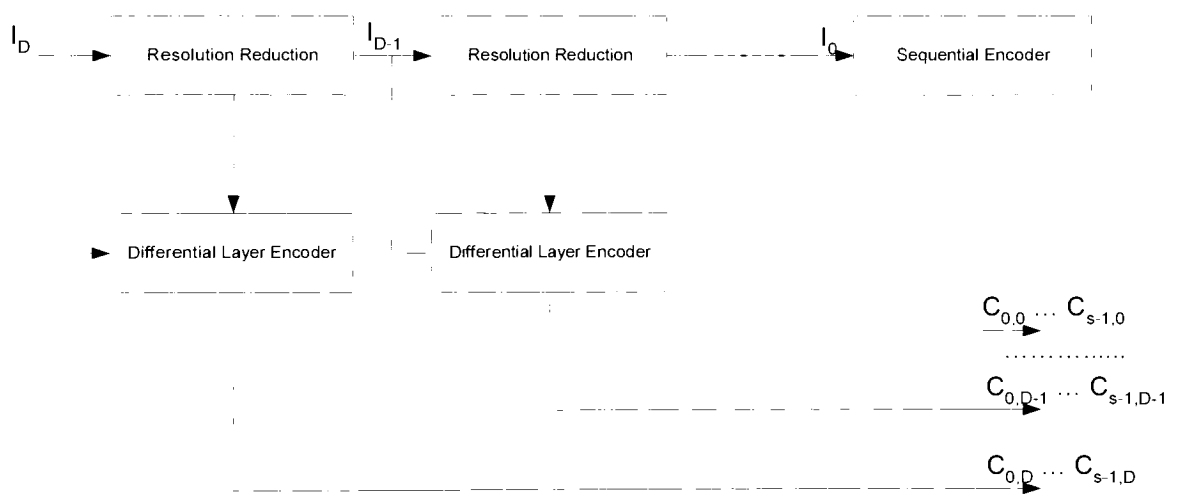


Figure 3.12 – JBIG Progressive Encoder

As can be seen in Figure 3.12 for progressive coding there is a variable number of stages, depending on the number of resolution layers used. With the exception of the bottom-layer, all of the stages consist of a differential layer encoder, shown in Figure 3.10, and a resolution reduction block, which is described above. The bottom-layer stage, I_0, consists of a sequential encoder, shown in Figure 3.9.

3.4 JBIG2

The Joint Bilevel Image Experts Group has developed a new standard, informally referred to as JBIG2, for lossless and lossy compression of bilevel images [6]. It supports model-based coding of text and halftones to permit higher compression ratios than JBIG, G3 and G4. JBIG2 also allows a preprocessing step for introducing loss that can be done to increase the compression ratio, hopefully without significantly affecting the visual quality of the image. The main goal when designing JBIG2 was to provide lossless compression performance better than that of existing standards as well as to provide lossy compression for bilevel images with almost no degradation in quality, which none of the current standards provide.

JBIG2 works by breaking up the image into segments and encoding each of these segments individually. For the most part, the images are broken up into textual regions and halftone regions. JBIG2 assumes it knows whether a certain image segment is a text region or a halftone region. For textual images a character-based pattem-matching technique is used. This pattern-matching technique is effective, because on a typical page of text, there are many repeated characters. So, instead of encoding all of the instances of the characters, only one instance of each character is encoded and entered into a dictionary. At the time of decoding this dictionary entry is then copied whenever another instance of that character appears.

JBIG2 provides for two pattem-matching algorithms, the first of which is called Pattern Matching and Substitution. Figure 3.13 shows the flow diagram for the pattern matching and substitution algorithm. As can be seen the algorithm starts by breaking up the image into pixel blocks where hopefully each pixel block is a character. Next each of the pixel blocks is checked to see if it is already in the dictionary, if it is, the index of the entry it matches is encoded. If it is not found in the dictionary, then the character is encoded and added to the dictionary. After a pixel block is encoded its position is encoded so that the decoder knows where to put it, usually this position is the distance from the previous character. For the actual format of the JBIG2 file, contrary to what Figure 3.13 illustrates, the dictionary data and the encoded data (the dictionary index and offset) is not interleaved. The dictionary is kept in a separate part of the file for easier decoding.

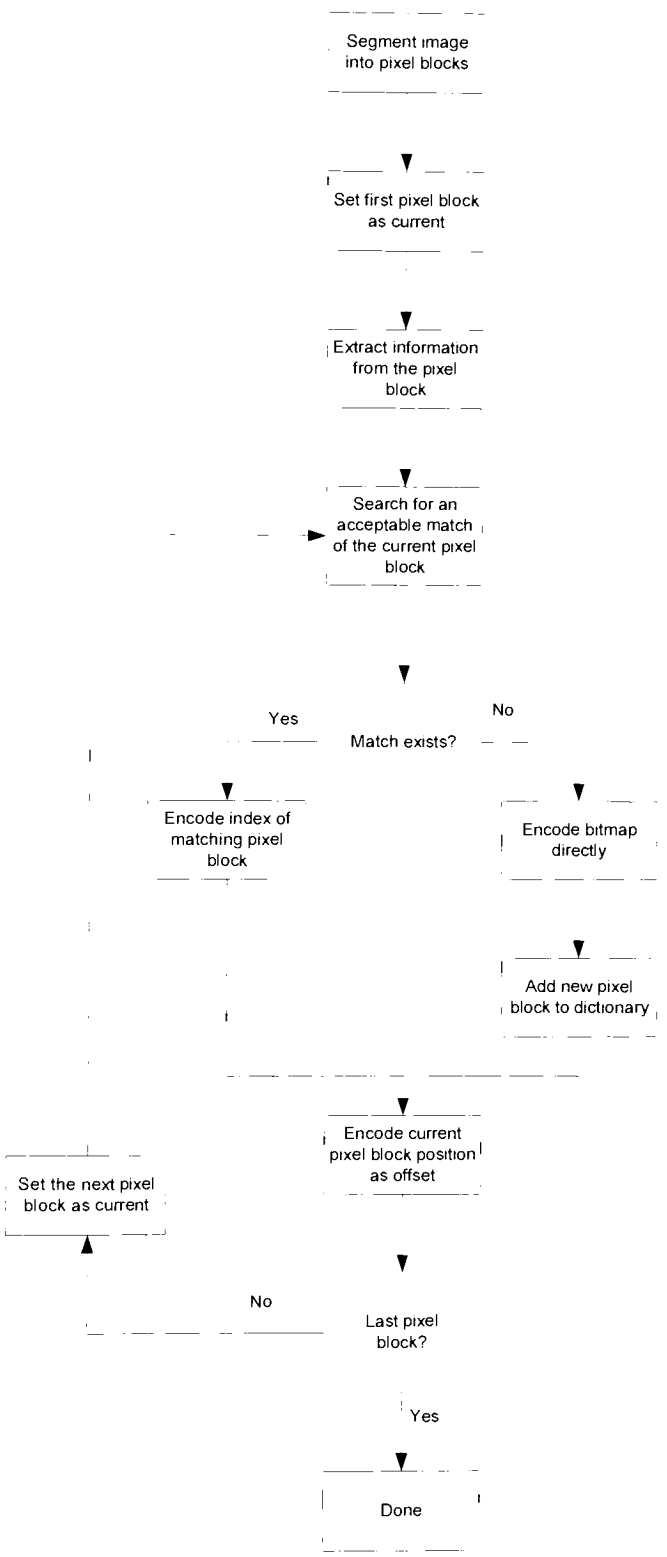


Figure 3.13 – JBIG2 Pattern Matching and Substitution Flow Diagram

An algorithm for the segmentation process described above is not given in the JBIG2 standard, so the implementation is left up to the programmer to use any segmentation technique. The next step in the pattern matching and substitution technique is to extract information about the pixel block: height, width, area, position, etc.

In a common scanned textual document two instances of the same character typically do not match pixel for pixel, even though the human eye sees them as the same. Therefore, the task of searching for an acceptable match in the dictionary is not a trivial one. This process is done in two steps. The first step consists of prescreening, where each of the entries in the dictionary is checked to see if its features, height, width, area, position, or the number of black pixels are close to those of the current pixel block. The second step consists of computing a match score for each of the potential matches that remain. A simple example of the match score would be the Hamming distance [23], which is found by first aligning the two blocks in comparison according to their geometric centers of their bounding blocks and then counting the number of pixels that differ. The dictionary entry with the highest match score is assumed to be identical to the pixel block being encoded, if its match score is above a predefined threshold.

In the case where a match is found the associated numerical data, the dictionary index and position, are either bitwise or Huffman-based encoded. In the case where there is no acceptable match and the new character must be added to the dictionary, the bitmap of the new character is encoded using a bitonal compression method, such as G4 or JBIG, discussed previously.

The pattern matching and substitution technique is a lossy compression algorithm that provides good compression ratios of textual data. One problem with this technique is that occasionally a wrong character is matched and then when the data is decoded, an incorrect character exists in the textual data.

The alternative to pattern matching and substitution for the compression of the textual regions using JBIG2 is called Soft Pattern Matching. This method differs from pattern matching and substitution in that in addition to encoding the dictionary entry and the position information, as

in the pattern matching and substitution technique, refinement data is included which can be used to recreate the original character in its lossless form.

The refinement data consists of the current desired character, encoded, making use of pixels from both the current character and the matching character. Because of the dictionary search that is described above, the correlation between the current character and the matching character should be rather high, so the prediction of the current pixel is now very accurate when encoding the current character. This allows for the current character to be encoded using a small number of bits. The Figure 3.14 shows the flow diagram for the soft pattern matching technique.

As can be seen when comparing Figures 3.13-3.14 the pattern matching and substitution technique is very similar to the soft pattern matching technique. The only difference between the two algorithms is that the lossy direct substitution of the matched character is replaced by a lossless encoding that uses the matched character in the coding context. The degree of refinement is left up to the encoder, so that the decoded character using the soft pattern matching technique does not necessarily have to be lossless, but it should be less lossy than the character produced by the pattern matching and substitution technique. The lossy soft pattern matching technique has one big advantage over the pattern matching and substitution technique described earlier in this section, which is that the lossy soft pattern matching technique does not need a very safe and intelligent matching procedure to avoid substitution errors. However, in any lossy substitution technique, substitution error can happen.

In order to encode the current character, an arithmetic coding such as the one used in JBIG is used. The bitmap of the current pixel block is losslessy encoded by aligning the geometric center of the current pixel block with the center of the matching pixel block and encoding each pixel in the current pixel block using the arithmetic coder. However, for this encoding, the standard JBIG templates are not used, instead a template that uses casual pixels (pixels already processed) from the current pixel block and neighboring pixels in the matched pixel block. An example of this template is shown in Figure 3.15.

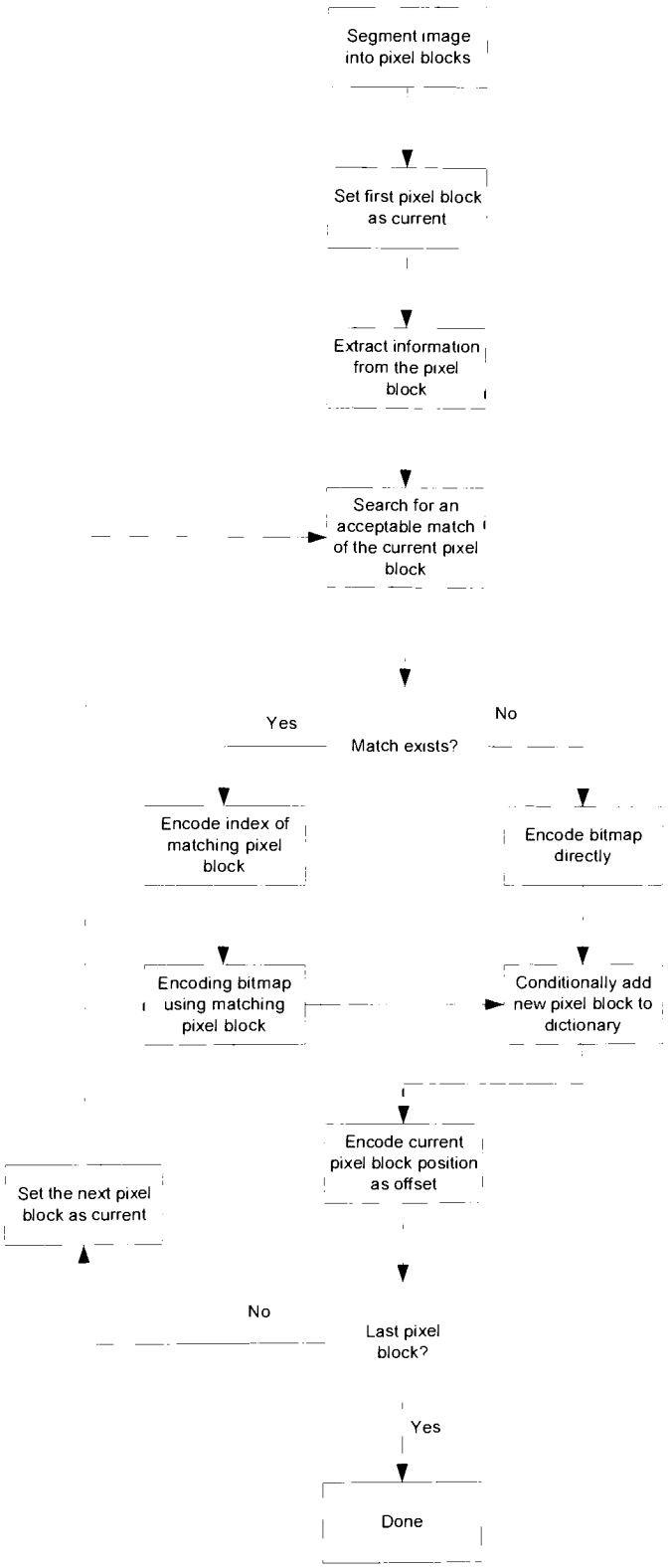


Figure 3.14 - JBIG2 Soft Pattern Matching

Figure 3.15 – JBIG2 Template for Refinement Coding

The template on the left is used for the current pixel block and the template on the right is used for the matched pixel block. The '?' is the pixel currently being encoded. The '?' matches up with the '7' pixel, meaning pixel '?' corresponds to pixel '7' when the pixel blocks are aligned according to their centers.

One big advantage to the soft pattern matching technique over the pattern matching and substitution technique is that in the pattern matching and substitution technique a substitution error can lead to the wrong character being inserted in the decoded image. However, in soft pattern matching, if a substitution error does occur the refinement coding takes care of this error, the worst that would happen is reduced compression efficiency.

As was stated before, the processing done prior to JBIG2 breaks up an image into textual portions and halftone portions. The compression of the textual portions was described already. JBIG2 offers two methods for the compression of the halftone portions of the image. The first method is similar to the arithmetic coding method used in JBIG. The main difference between this method and the one described in JBIG is the size of the template. In JBIG2 the template can be as big as 16 pixels with up to 4 adaptive pixels where in JBIG the template was 10 pixels with only one adaptive pixel. An example of this JBIG2 template can be found in Figure 3.16. This larger template is designed to provide a better prediction of the current pixel by exploiting specific types of redundancies that exist in halftone images.

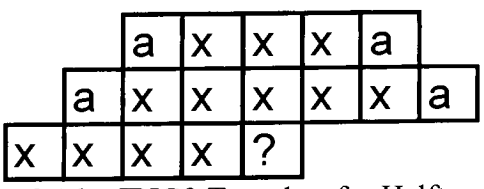


Figure 3.16 – JBIG2 Template for Halftone Encoding

The second method for encoding the halftone blocks in JBIG2 is done by taking the inverse halftone of the bitonal image to produce a grayscale image and compressing this grayscale image. To perform the inverse halftoning, first the bitonal image is broken up into m_b rows by n_b columns, padding the image if necessary with zeros on the right and bottom. So, for a bilevel image of m rows by n columns, a grayscale image is obtained of m_g rows by n_g columns. The equations for m_g and n_g can be found below.

$$m_g = \left\lfloor \frac{m + (m_b - 1)}{m_b} \right\rfloor \qquad \textit{(Eqn. 3.2)}$$

$$n_g = \left\lfloor \frac{n + (n_b - 1)}{n_b} \right\rfloor \qquad \textit{(Eqn. 3.3)}$$

A grayscale value is then produced for each m_b by n_b block. The grayscale value for each block could simply be found by using a dictionary with the various bit pattems and the corresponding grayscale values. The grayscale image is then Gray coded. Gray coded means that for each grayscale pixel level change of one value, only one bit can change in the Gray code value. Figure 3.17 shows the first 10 numbers in the Gray code. The bolded digit in the binary number represents the bit that changes.

After the image is Gray coded, it is encoded using context-based arithmetic coding as in JBIG. In order for the decoder to get the bitonal image back it takes the grayscale value and looks up the bitonal bitmap in the dictionary and substitutes that pattern. This method is somewhat similar to the two pattern matching methods of textual encoding in that a dictionary is used with substitution for the inverse halftoning. This method yields good compression results.

Although JBIG2 provides the opportunity for lossy compression, the types of loss that can be introduced are not defined in the standard. JBIG2 simply defines how the decoder works and how the JBIG2 compressed bit stream should look. The decoder is lossless with respect to the encoded image, however it might not be lossless with respect to the original image depending on what kind of loss is introduced in the encoding step. The original image could have loss introduced at the encoding stage in order to allow for higher compression efficiency. Of course, the pattern matching and substitution method for textual compression is inherently lossy.

Grayscale Value	Gray Code Binary Value
0	00000000
1	0000000**1**
2	0000001**1**
3	0000001**0**
4	0000011**0**
5	0000011**1**
6	00000101
7	00000100
8	00001100
9	00001101
10	00001111

Figure 3.17 – Gray Code Example

For bitonal images loss in an image is simply the flipping of pixels. For the pattern matching and substitution technique loss is introduced whenever there is a mismatch between the original pixel block and its match. A few examples of methods for introduction of loss are Quantization of Offsets, Noise Removal and Smoothing and Bit Flipping. Quantization of offsets and noise removal are described next and a bit flipping method is described in Chapter 4.

In the quantization of offsets the position of the characters is quantized. Quantizing the offsets basically means estimating the position of the character. For English text on a portrait-oriented page, character positions can be safely quantized to about 0.015 inches in the horizontal direction and 0.01 inches in the vertical direction, but any more quantization provides noticeable distortion and sometimes can make the resulting text unreadable. Unfortunately this method only provides an increase of about 1% in compression ratio so it is not a very useful procedure.

The noise removal and smoothing method involves images that consist of mostly text at a resolution proportional to the character sizes. In this method very small pixel blocks are removed from the image rather than trying to find a substitution match, because most likely this

small pixel block is just noise. One example of a smoothing method is to remove signal-protruding pixels along the edges of pixel blocks. Smoothing has the effect of providing a more accurate match for the substitution methods, which reduces the number of mismatching pixels between the original pixel block and its match. This method normally provides for about a 10% increase in compression efficiency, but only for textual documents.

JBIG2 is an efficient standard for multi-page mixed type documents. It provides compression ratios much greater than those of the G3, G4 and JBIG algorithms as will be seen in Chapter 6. However, one aspect in which the standard lacks is the introduction of loss. It basically provides the ability for the introduction of loss in preprocessing, but it does not give any methods for introducing this loss. The only lossy compression it really talks about is pattern matching and substitution, which is inherently lossy.

3.5 BACIC

Block Arithmetic Coding for Image Compression (BACIC) is an algorithm for the compression of bitonal images developed by Maire D. Reavy [1]. BACIC uses the Block Arithmetic Coder (BAC) in order to compress the image. BAC is a simple and efficient arithmetic coder that can be easily implemented on a variety of platforms. Because of the lack of compression efficiency when using G3 and G4 [3], and the fact that JBIG and JBIG2 both use IBM's patented QM-coder [11], this coding algorithm is used for this thesis.

As was stated earlier, there are two basic steps to arithmetic coding: probability estimation and encoding. Similar to the JBIG standard, BACIC uses templates to estimate the dynamic probability of p_0 and p_1, the probability that a pixel is a 0 or 1 respectively. In BACIC there are two templates that can be used to predict the value of a pixel, each of them is 12 bits. The first template is a three-line template and is used mostly for business-type documents and halftone images generated by error diffusion. The second template is a five-line template, which is used for halftone images generated by ordered dither. The two templates are shown in Figure 3.18.

These templates were designed to use the typical patters of different types of images. For example, business-type documents generally have patterns that occur as clusters around the

current pixel, so the three-line template provides a good prediction for this type of image. Dithered images, however, generally have patters that occur in horizontal, vertical or diagonal lines, so the five-line template provides a good prediction. As can be seen in the five-line template, it uses pixels directly above, directly to the left and diagonally up and to the right of the current pixel. Diagonal pixels above and to the left are not used, because it is assumed that the vertical and horizontal group of pixels covers those patterns. Also, many picture type images are naturally symmetric so the one diagonal line is sufficient. Because of the way error diffused images are created, using a feedback system, the value of a pixel is most influenced by the pixels above and to the left of it, and those in close proximity to it. For this reason, the three-line template works very well for prediction in error diffused images.

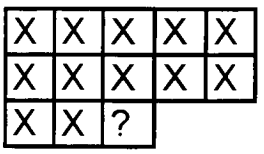


Figure 3.18(a)-Three-Line Template

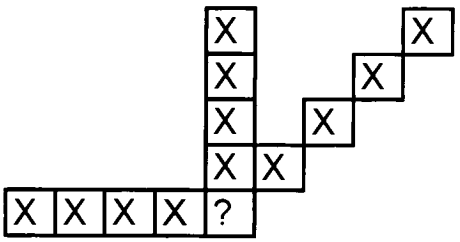


Figure 3.18(b)-Five-Line Template

In the BACIC algorithm, the BAC assumes that the method for halftoning the image being compressed is known, so choosing the template is done before encoding. This can either be an input parameter to the encoder or an algorithm could be used to detect the type of image. Because image type detection is not part of this thesis, the template to use is a parameter specified by the user to the encoder.

As with JBIG, the pixels of the template are grouped together to form an integer, which is a context into a table. The probability table for BACIC is different than the one used for JBIG. The table in BACIC consists of 2 real numbers, r_i and s_i. An example of this table can be seen in Figure 3.19.

Context	r_i	s_i
000000000000	r_0	s_0
000000000001	r_1	s_1
.		
.		
.		
111111111110	r_{4094}	s_{4094}
111111111111	r_{4095}	s_{4095}

Figure 3.19 – BACIC Probability Table

Using these two floating-point values p_1 is then computed using the formula

$$p_1 = \frac{r_i - \Delta}{s_i + 2\Delta} \qquad \textit{(Eqn. 3.4)}$$

where i is the context, and Δ is a constant, 0.006 for BACIC, used to keep from overestimating p_1, which is discussed below. r_i and s_i are computed using the equations

$$r_i(n+1) = u - \alpha r_i(n) \qquad \textit{(Eqn. 3.5)}$$

$$s_i(n-1) = 1.0 + \alpha s_i(n) \qquad \textit{(Eqn. 3.6)}$$

$$r_i(0) = 1.0 \qquad \textit{(Eqn. 3.7)}$$

$$s_i(0) = 2.0 \qquad \textit{(Eqn. 3.8)}$$

where u is the value of the current pixel and n is the number of previous pixels having the context *i*. The constant α is called the forgetting factor, and for BACIC has a value of 0.985. The forgetting factor insures that more recent pixels have a more significant effect on p_1 then pixels processed much earlier. Because of the nonstationary nature of images, recent pixels are typically predictors of the current pixel.

In images where a context occurs very frequently s_i may reach its upper limit, which is

$$\lim_{n\to\infty} s_i(n) = \frac{1}{1 \quad \alpha} = \frac{1}{1 - 0.985} = 66.67$$

This can cause the ratio of r_i and s_i to overestimate the true value of p_1, causing a decrease in performance. In order to compensate for this the constant Δ is used to bias the estimated p_1 slightly toward 0.5.

As stated previously, BACIC uses the block arithmetic coder. BAC uses a binary coding tree in order to determine the codeword for a variable set of pixels. The arithmetic coder uses two parameters in order to perform the encoding: p_0 and K. Because these are binary images, p_0 can be found using p_1. K is the size of the BAC codebook and has the restriction of being a multiple of 2. For this thesis K is equal to 4096.

Encoding consists of traversing the image in raster scan order and, for each pixel, computing p_1 and p_0. After computing the probabilities, K_0 and K_1 are found using the following equations.

$$K_0 = round(K * p_0) \qquad \textit{(Eqn. 3.9)}$$

$$K_1 = K \quad K_0 \qquad \textit{(Eqn. 3.10)}$$

A tree is then built using K as the parent node and K_0 and K_1 as its two child nodes. Then depending on what the current pixel is, either K_0 or K_1 is chosen as the new parent node and the process repeats until a value of 1 for the child node chosen is obtained. Exceptions to Equations 3.9 and 3.10 occur when K_0 or K_1 equal either 0 or the parent value. In the case where a 0 occurs, the node is set to 1, and in the case where the child node would equal the parent node a value of the parent node – 1 is used.

Figure 3.20 is an example of building the binary tree using a value $p_0 = 0.4$ and a value of $K = 16$. To create the tree, the root note was assigned a value of K, which is 16. Its two child nodes were then found using Equations 3.9 and 3.10. So the child nodes of 16 are 6 and 10. This

process is repeated recursively until all child nodes of 1 are obtained. Then each child node is assigned a codeword, 0 through K-1 as can be seen in the figure above. The codewords can now be obtained from this tree. For example if the input string is "1101" then the codeword would be 11 or "1011".

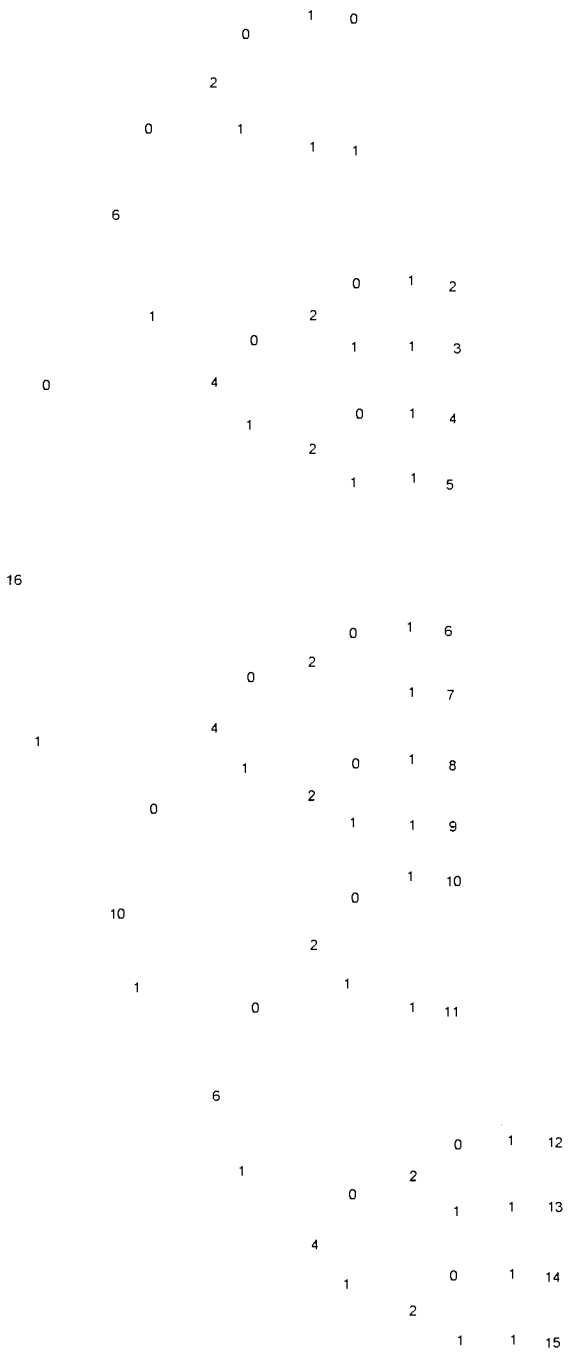


Figure 3.20 – BACIC Static Binary Coding Tree

However, when encoding an image, the probabilities are going to change as the template moves and the r_i and s_i values change. Therefore, an adaptive tree is used. The process is very similar to the example shown above in Figure 3.20. In the example below K is again 16, the input string is "0100....." and the respective p_0 values are {0.7, 0.55, 0.9, 0.2.....}.

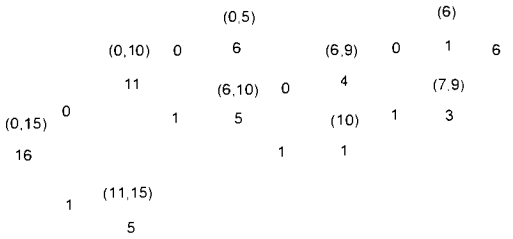


Figure 3.21 – BACIC Adaptive Binary Coding Tree

Identically to the static tree the root node is set to K and the two child nodes are found using Equations 3.9 and 3.10. Because the first pixel in the input stream is a '0', the 11 node now becomes the current node and its two child nodes are found. Also, while this is happening, codeword ranges are kept, so the 11 node has a codeword range of (0,10) and the 5 node has a codeword range of (11,15). The second pixel in the input string is a '1' so the new node becomes the 5 node and its range is (6,10). The third pixel in the input string is a '0' so the new node becomes 4 and its range is (6,9). The fourth pixel in the input string is a 0 so the new node becomes 1 and the process is over. The codeword for input string "0100" is 6 or "0110". This is the process used in the BACIC algorithm, with the exception that a much larger K value is used to yield better compression results.

Chapter 4. Loss Introduction Algorithms

The goal of a lossy compression algorithm is to improve on the compression ratio while keeping the visual distortion to an acceptable level. This is not a trivial task when dealing with bitonal images, because choosing which pixels to flip in order to improve compression efficiency to a maximum while sustaining the visual is a tricky process. In this chapter, algorithms are discussed for introducing loss to a bitonal image.

4.1 Background

In the BACIC algorithm, the equations used for the arithmetic coder can be found in Equations 3.4-3.8. As can be seen in these equations, BACIC uses a forgetting factor α. However, for the loss introduction part of this thesis, α must be ignored. The reason is that for the loss introduction algorithms the entire image is processed as a whole, collecting context statistics over the entire image. If the forgetting factor were included in these computations, then the loss introduction would work well at the very bottom portion of the image, but for the rest of the image it would not work very well. To handle this problem, for this chapter, the forgetting factor α is set to 1.0. Therefore, Equations 3.5 and 3.6 change as shown below.

$$r_i(n-1) = pix - \alpha r_i(n) = u - r_i(n)$$

$$s_i(n-1) = 1.0 - \alpha s_i(n) = 1.0 - s_i(n)$$

This means that r_i simply counts the number of previous pixels for each context that equal '1' while s_i counts the number of previous pixels for each context not considering their value. So for this chapter the two symbols n_0 and n_1 are used. $n_0(c)$ is the number of pixels equaling a '0' for context c and $n_1(c)$ is the number of pixels equaling a '1' for context c. Therefore, Equation 3.4 now becomes

$$p_1 = \frac{n_1 + \Delta}{n_0 + n_1 + 2\Delta} \qquad \textit{(Eqn. 4.1)}$$

Because these are binary images, the following equation also holds true.

$$p_0 = 1.0 - p_1 = \frac{n_0 \quad \Delta}{n_0 + n_1 + 2\Delta} \qquad \textit{(Eqn. 4.2)}$$

As shown in Equation 2.2, the number of bits needed to encode any pixel is

$$I(E) = -\log_2 p_u$$

where u is the value of the pixel. Therefore, when encoding an image the encoding length of the image for context c with n_0 and n_1 is

$$\lambda(n_0, n_1) = -\log_2 \frac{0 + \Delta}{0 + 2\Delta} - \log_2 \frac{1 + \Delta}{1 + 2\Delta} - \log_2 \frac{1 + \Delta}{2 + 2\Delta} - \log_2 \frac{2 + \Delta}{3 + 2\Delta} - \log_2 \frac{2 + \Delta}{4 + 2\Delta} - \ldots -$$
$$\log_2 \frac{n_u - 1 + \Delta}{n_0 + n_1 - 2 + 2\Delta} - \log_2 \frac{n_{\bar{u}} - 1 + \Delta}{n_0 + n_1 - 1 + 2\Delta} =$$
$$-\log_2 \left[\left(\frac{0 + \Delta}{0 + 2\Delta} \right) \left(\frac{1 + \Delta}{1 + 2\Delta} \right) \left(\frac{1 + \Delta}{2 + 2\Delta} \right) \left(\frac{2 + \Delta}{3 + 2\Delta} \right) \left(\frac{2 + \Delta}{4 + 2\Delta} \right) \ldots \left(\frac{n_u + \Delta}{n_0 + n_1 - 1 + 2\Delta} \right) \left(\frac{n_{\bar{u}} + \Delta}{n_0 + n_1 + 2\Delta} \right) \right]$$

where $\bar{u}$ is the opposite value of u [2]. Therefore

$$\lambda(n_0, n_1) = -\log_2 \frac{\prod_{j=0}^{n_0 - 1} (j + \Delta) \prod_{j=0}^{n_1 - 1} (j + \Delta)}{\prod_{j=0}^{n_0 + n_1 - 1} (j + 2\Delta)} \qquad \textit{(Eqn. 4.3)}$$

The exception to this formula is of course if either $n_0 = 0$ or $n_1 = 0$ or both equal 0. If they both equal 0 then no pixels are encoded, so the encoding length is 0. If either of them are 0 then that portion of the fraction is ignored. The encoding length of the entire image is

$$L = \sum_c \lambda(n_0(c), n_1(c)) \qquad \textit{(Eqn. 4.4)}$$

As stated before, the main goal of flipping pixels is to alter pixels in a manner that decreases the encoding length of the image. Because the values for p_0 and p_1 are found using templates,

if a pixel, u, were to be flipped it would not only change the encoding length of itself, but also change the encoding length of all the pixels in which u is a template pixel. This concept is better illustrated using the figure shown below.

12	11	10	9	8
7	6	5	4	3
2	1	?	1'	2'
3'	4'	5'	6'	7'
8'	9'	10'	11'	12'

Figure 4.1-BACIC Three Line Template Reflection

In this figure the pixels with apostrophes denote the template location where the current pixel, denoted by '?', would be for each of those pixels. For example, pixel 1' denotes that the current pixel would be template pixel number 1 when pixel 1' is processed.

If the quantity ΔL_q denotes the impact on the code length of altering the context of the pixel for which u is context pixel number q, then the change in the code length of the entire image, ΔL , if a pixel is flipped is

$$\Delta L = \sum_{q=0}^{k} \Delta L_q \qquad \textit{(Eqn. 4.5)}$$

where k is the number of pixels in the context, for BACIC $k = 12$. The direct gain from flipping u to $\overline{u}$ in context c is

$$\Delta L_0 = \log_2 \frac{n_u(c) - 1 - \Delta}{n_0(c) + n_1(c) - 1 + 2\Delta} -$$

$$\log_2 \frac{n_{\bar{u}}(c) + \Delta}{n_0(c) + n_1(c) - 1 + 2\Delta} = \log_2 \frac{n_u(c) - 1 + \Delta}{n_{\bar{u}}(c) + \Delta} \qquad \textit{(Eqn. 4.6)}$$

Using the notation u' as the value of the mirror pixel of context pixel number q the indirect terms in the sum are

$$\Delta L_q = \log_2 \frac{n_{u'}(v_q) - 1 + \Delta}{n_0(v_q) + n_1(v_q) - 1 + 2\Delta} - \log_2 \frac{n_{u'}(w_q) - \Delta}{n_0(w_q) + n_1(w_q) + 2\Delta} \quad \textit{(Eqn. 4.7)}$$

where v_q is the original context for pixel q and w_q is the context for pixel q if the pixel u is flipped. The remainder of this chapter discusses how to use the values λ and ΔL to decide which pixels should be flipped.

4.2 Greedy Rate-Distortion Flipping

One algorithm for the introduction of loss is called Greedy Rate Distortion Flipping [2]. This algorithm works by ranking pixels according to their ΔL values and then flipping the most profitable first. The algorithm can be summarized in 3 steps:

1. Collecting image statistics over all contexts.
2. Listing flip candidates.
3. Evaluating and flipping the candidates.

The first step, collecting the image statistics, consists of going through each pixel in the image, finding the context and then incrementing either n_0 or n_1 for that context. Once step 1 is completed, the coding length of the entire image can be found using Equation 4.3.

The second step consists of listing and sorting the flip candidates. In this step the image is again traversed and for each pixel the value of ΔL is computed. Because ΔL is the change in the encoding length of the image if the pixel were to be flipped, a negative value of ΔL is a profitable flip. The actual value used is $\frac{\Delta L}{\Delta d}$, where Δd is the change in distortion if the pixel were to be flipped, in most cases $\Delta d = 1$ bit/pixel. Therefore, those pixels with negative $\frac{\Delta L}{\Delta d}$ values are added to a list to be evaluated later. After all the pixels have been investigated, the list is then sorted. The flip candidates are then flipped one by one until a predetermined

threshold is reached. An example of this threshold is encoding length. In this case, the encoding length of the image is computed before any flipping is done and then after reach pixel is flipped ΔL is added to this length and checked against the threshold. By sorting the list it insures that the most profitable pixels are performed first, reducing the number of total pixels that need to be flipped in order to meet the demand.

It is possible, however unlikely, that a good flip candidate would be ignored. In some cases it may be beneficial to flip a pixel in which ΔL is greater than 0. One example of this can be shown using two pixels, a and b, such that a is a context pixel of b. It might be profitable to flip b if a were to be flipped, but since that is not taken into consideration in the calculation for b, the algorithm would not find that it is profitable to flip both a and b. However it should be seen that this case is unlikely to happen, because ΔL_0 should have the largest effect on ΔL, because the other factors in the summation are simply changing a context pixel in a relatively large context, so the prediction should still be accurate.

4.3 Low-Latency Greedy Flipping Utilizing Forgetful Error Diffusion

The rate distortion algorithm described above provides very good compression efficiency improvement over the compression of the lossless image, but it is a slow algorithm, because it requires three passes over the entire image and requires a sorting algorithm. The sorting algorithm is relatively trivial to implement but it can be very time consuming to run. Because most useful images contain such a large number of pixels, the list of the flip candidates for rate distortion could contain a good amount of candidates. Traversing this list and sorting the candidates could be a costly procedure even when using a quick sorting algorithm.

Another algorithm proposed, which is the one used for this thesis, is called Low-Latency Greedy Flipping Utilizing Forgetful Error Diffusion [2]. In this algorithm no sorting is required and only two passes over the image are needed. As will be seen soon, this algorithm also provides means for controlling image quality better than the rate-distortion algorithm. Not using a separate pass over the image to list flip candidates achieves this. The two steps for this algorithm are:

1. Collecting statistics over all contexts
2. Proceeding through the image and flipping pixels

The first step is the same as for the rate-distortion algorithm. Each pixel is visited, the context for that pixel is found and then either n_0 or n_1 is incremented depending on the pixel value. The difference between the two algorithms comes in step 2. This algorithm works with blocks. The image is broken up into blocks of size 8 x 8 pixels. Each block is then processed in raster scan order until all the blocks have been visited. For each block, ΔL is computed for each of the 64 pixels, and then if the distortion control allows it the most profitable pixel is flipped. If the distortion control does not allow it, the second most profitable pixel is flipped and so on. Below is a table that lists all the parameters used for this algorithm.

Name	**Description**
n_{max}	The maximum number of times a block can be processed.
g_{max}	The limit for the numerical grayscale error in a block.
s	Forgetting factor.
τ_λ	Threshold for the marginal length.
τ_L	Threshold for ΔL.

Table 4.1 – Error Diffusion Loss Introduction Parameters

The n_{max} parameter is the maximum number of times each block can be processed. Because only one pixel is flipped on each pass of a block, n_{max} is also the maximum number of pixels that can be flipped in each block. It is a good idea to make n_{max} an even number so that complementary flips are more likely to occur. If one pixel in a block is flipped from a 0 to a 1 and another is flipped from a 1 to a 0 then the perceived grayscale error of the block is still 0 and the noticeable loss when viewing the image should be at a minimum, if any.

The perceived grayscale error is the main distortion controlling quantity used in this algorithm. The grayscale error is recorded for each block of the image. Initially all blocks have a perceived grayscale error of 0. Then as each pixel is flipped the grayscale error for that block changes. If the pixel is flipped from 0 to 1 a value of 1.0 is added to the grayscale error and if a

pixel is flipped from a 1 to a 0 then a value of 1.0 is subtracted from the grayscale error. Before processing, a limit is set on the grayscale error per block, g_{max}, so that the actual grayscale error for a block, while processing, can be compared to this threshold to see if the flipping of a pixel is legal.

After the processing of a block is completed the grayscale error for that block is then diffused to the other surrounding blocks using the Floyd-Steinberg weights [16]. By diffusing the grayscale error into surrounding blocks that have not yet been processed the quality of the image can be sustained better. The main goal of keeping this grayscale error and diffusing it is to try to make complementary flips either in the same block or in neighboring blocks. This means that if a pixel is flipped from a 0 to a 1 in a block the next flip in that block or the surrounding blocks will hopefully be a 1 to a 0. This sustains the overall perceived grayscale quality of the image.

The s parameter is used when diffusing the error. It is a scaling factor similar to that of BACIC. It is only used when no flips could be done in a block, in which case the grayscale error for that block is multiplied by s before diffusing. The value of s assumes values between 0.0 and 1.0. This is also an important parameter for the quality of the image, because it can reduce the amount of recorded grayscale error diffused and, therefore, increase the actual grayscale error of the image, because more pixels can be flipped.

The thresholds τ_λ and τ_L are used to check against the marginal length and the value for ΔL. The marginal length is defined in equation shown below.

$$\lambda = -\log_2 p(u \mid c) \qquad \textit{(Eqn. 4.8)}$$

where u is the value of the pixel and c is the context. The marginal length is number of bits needed to compress the current pixel. If compression is performed this value will be less than 1, if negative compression is performed this value will be greater than 1. As was stated before a negative ΔL value would make the encoding length of the image less and, therefore, make this pixel a profitable flip and a positive value would not make this pixel a profitable flip.

Even though the compression efficiency of this algorithm is slightly worse than that of the rate-distortion algorithm the opportunity to control the visual distortion in the error diffusion algorithm is much greater and, therefore, makes this algorithm more useful for the introduction of loss. One method to get an even higher compression ratio for this algorithm, if that is required, is to pass over the entire image multiple times. Of course this would make for a longer running time for the algorithm as well as decrease the quality of the image.

4.4 Region of Interest

In most images there exists an object or objects that are the main focus or area of interest in the image. For example, a face of a person would be an example of a region of interest in an image of a person. When introducing loss into an image, one thing that can be considered would be this region of interest. Obviously a person would want much less loss or no loss in a region of interest and would want more loss outside of this region of interest in order to increase the compression efficiency while keeping detail in the important part of the image.

The low-latency greedy flipping utilizing forgetful error diffusion algorithm can be modified in order to accommodate this region of interest. As described above the error diffusion algorithm for loss introduction has several parameters associated with it. In order to accommodate a region of interest these parameters change as the image is traversed in order to allow for more loss to be introduced in some areas and less or none in others.

Four basic quality levels are defined for this thesis: perfect, high, medium and low. Each of these quality levels has different parameters associated with it, determined experimentally, as can be seen in Table 4.2.

	Perfect	**High**	**Medium**	**Low**
n_{max}	0	2	4	16
g_{max}	N/A	0.25	2.0	16
s	N/A	0.5	0.25	0.0

Table 4.2- Error Diffusion Loss Introduction Parameter Values

As can be seen in Table 4.2, the values chosen for n_{max} are even to hopefully promote complementary bit flipping. For the perfect quality there are no passes through the block, of course. For the high quality only 2 pixels can be flipped in each block and for the medium 4 pixels per block can be flipped. For low 16 pixels per block can be flipped, however it is unlikely that more than ¼ of the pixels would be profitable flips, so basically all the profitable flips are done. The g_{max} value for high quality was chosen so that two flips could not be done in the same block unless they were opposite flips and it would be difficult to do flips in the same direction for neighboring blocks, so that quality is sustained. The two parameters missing from this table, τ_λ and τ_L, are constant for all quality levels, 1.0 and 0.0 respectively. These parameters were chosen so that any flip that decreases the encoding length of the image can be done if the parameters in Table 4.2 allow it.

As will be seen in later chapters these parameter values provide good control over the loss introduction algorithm. It is useful to be able to determine at what quality level each part of an image should be compressed. Using this method maximum compression efficiency as well as maintaining quality can be achieved.

Chapter 5. Hardware Implementation

This chapter deals with the hardware implementation of the BACIC compression system and its modules. The prototyping board used, including the Field-Programmable Gate Array (FPGA) targeted for the implementation, is discussed. An FPGA was chosen for this implementation, because of its size. This small chip could be included in a printing or facsimile unit without requiring a significant amount of space.

5.1 Introduction

The BACIC codec (encoder and decoder) described in Section 3.5 was implemented on an FPGA platform. The FPGA targeted for this thesis was chosen for several reasons:

- Size: The FPGA contains 300,000 gates, which is enough for the BACIC codec.
- Speed: The FPGA has the ability to run on a 100MHz clock, which is fast when considering other FPGAs on the market.
- Availability: This FPGA and the Xess prototyping board were purchased by RIT and are available for student use in the Department of Computer Engineering.

5.2 Xess XSV-300 Prototyping Board

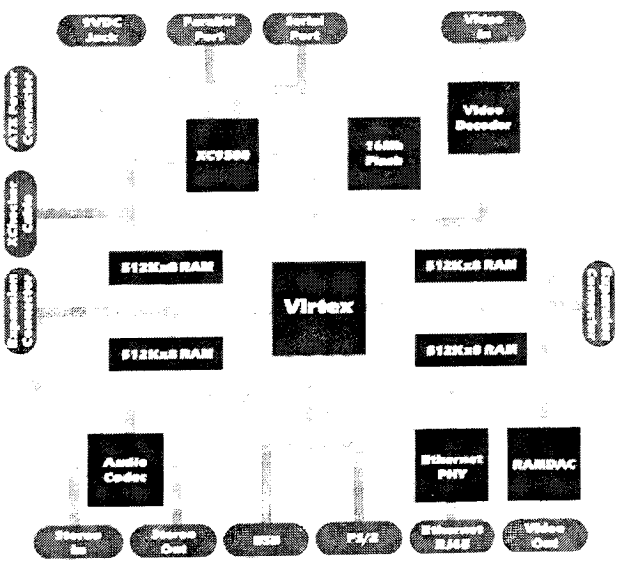


Figure 5.1 – XSV-300 Prototyping Board

The prototyping board targeted for this thesis is the Xess XSV-300 Prototyping Board [4], which uses the Xilinx XCV300 Virtex FPGA. VHDL code was used to model the BACIC encoder and decoder in hardware. The XSV-300 board has 2 Megabytes (MB) of memory in the form of Static RAM (SRAM). This memory is sufficient for storing the input image, the output image, the probability estimation tables and various parameters to the system. A block diagram of the XSV-300 prototyping board can be seen in Figure 5.1.

Because of its large memory and FPGA, this prototyping board provides a good target environment for the hardware development of the BACIC codec.

5.3 SRAM Model Module

Simulation is an important part of hardware development, because there are always problems with a design the first time around. Debugging the VHDL code after it has been programmed into the FPGA is almost impossible, because information inside the chip is not usually visible. To overcome this problem, VHDL simulators were created so that bugs could be found and fixed prior to synthesis.

In order to simulate the VHDL code using a simulator, a model of the prototyping board parts must be created e.g. the SRAM model must be created to mimic the SRAM on the prototyping board in order for testing to be successful. As can be seen in Figure 5.1 above, there are two banks of SRAM on the board, each consisting of 1 MB of storage space. It was decided that only two of these banks were required for this implementation, because only 1 MB of storage is needed and it is desired to reduce complexity of the VHDL memory controller.

The SRAM model handles all of the file input and output as well as provides the code to handle the requests to the SRAM, such as memory reads and writes. Upon system startup, the SRAM model reads in the input file, either the bitonal image or the BACIC compressed data depending on whether the system will be encoding or decoding. This model then waits for read or write requests to memory. After the system has finished executing, it writes the output to a file. The

output could either be a bitonal image or a BACIC compressed data file. The figure below shows the interface diagram for the SRAM Model.

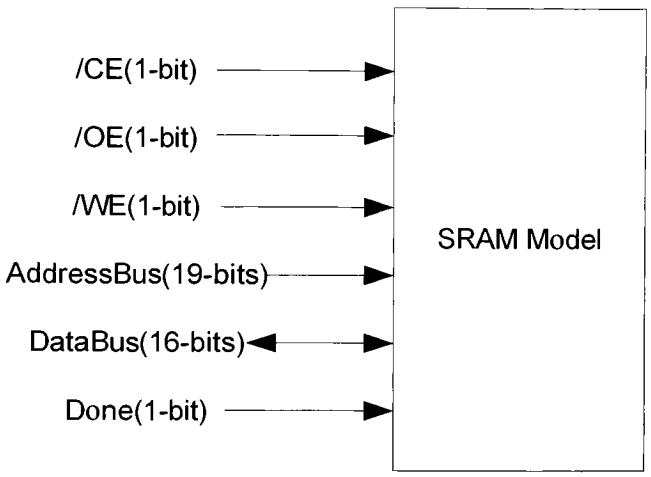


Figure 5.2 – SRAM Model Interface Diagram

The SRAM used on this board works on data as words (2 bytes), so that for every address there are 2 bytes stored in memory. To accomplish this, two of the memory chips are used in parallel. One of the chips is the high byte and the other chip is the low byte. Using the memory in this fashion provides for fast memory access.

As can be seen in Figure 5.2 above, there are three control bits used for this SRAM: Chip Enable (CE), Output Enable (OE), and Write Enable (WE). These three control lines are all active-low. The CE bit is used to enable the chip, meaning when low-voltage is applied, a memory operation can be performed. The OE bit is used for memory reads. When a low-voltage is applied on this line, then the data lines are set to whatever is at the memory location specified by the address lines. The WE bit is used to write memory. When a low-voltage is applied on the WE line, whatever is on the data bus is written to the memory location at the address on the address bus. The address bus is used to specify the address for reading or writing. The data bus is bidirectional in order to support both reads and writes. On a read the data bus is an output for the SRAM model and for a write the data bus is an input for the SRAM model. The Done bit is used only for simulation to inform the SRAM model that the operation is complete at which time it writes the data to a file.

The VHDL code for all components in this system can be found in Appendix A. Each memory block was modeled using an array of STD_LOGIC_VECTORs of size 8-bits. Since there are two memory blocks being used, there are two arrays of STD_LOGIC_VECTORs. The logic for the SRAM model was implemented using two processes. The first process reads in the file and sets all the memory prior to the algorithm starting. It then checks the CE, OE and WE lines and performs the necessary reads and writes. The second process performs the file write when the algorithm is finished. When the done signal is asserted, this process runs and writes the file.

Without this SRAM model, the creation of the hardware implementation of the BACIC encoder and decoder would not have been possible. This component provides a very important functionality, which is simulating the system running on the prototyping board.

5.4 Memory Controller Module

Another important module in this system is the memory controller. The memory controller allows communication between the SRAM and the rest of the system, and provides a synchronous interface to asynchronous memory. There are several different functions that have been implemented in the memory controller which are modeled around the needs of the BACIC system.

The first step was dividing the memory into blocks for all of the storage needed for this system. Each of the blocks and the respective memory ranges can be seen in Table 5.1. The image size is at the start of memory and occupies two words. This provides one word for the width and one word for the height. It was decided that the maximum size that this hardware implementation could handle would be 512 x 512 pixels. This reduces the hardware complexity, but also allows for large images to be processed.

The image data portion of memory is 16384 words. This allows for the storage of 262,144 bitonal pixels, which is the maximum image size. For encoding of the image this section will only be read by the system and for decoding an image this part of memory will be written.

	Size(words)	Start Address	End Address
Image Size	2	0x00000	0x00001
Image Data	16384	0x00002	0x04001
Probability Estimation Table	16384	0x4002	0x08001
Function/Template	1	0x08002	0x08002
Compressed Size	1	0x08003	0x08003
Compressed Data	N/A	0x08004	N/A

Table 5.1 – Hardware Memory Map

The probability estimation table section of memory stores the table shown in Figure 3.19. Because these were real numbers and not integers, consideration had to be given into how much precision to use for these fixed-point numbers. After experiments with different size fixed-point numbers it was decided that both r_i and s_i would be 32-bit numbers. Of the 32 bits, 7 bits are used for the integer portion of the fixed-point number and 25 bits are used for the fractional part. This provides for enough precision to let the algorithm efficiently compress, but does not take up too much data space or computation time. Therefore, the data portion of memory allocated to this table is 16,384 words which provides for 64-bits for each of the 4096 entries.

One word is allocated as a parameter into the system. The first byte of the word is the function the system is performing, 0 for encoding and 1 for decoding. The second byte of the word is which template the system is using, 0 for the 3-line template and 1 for the 5-line template (see Figure 3.18).

One word is allocated in order to store the size of the compressed data. When decoding an image, this number is written by the user prior to the system starting, and when encoding, this number is written by the encoder. The reason for this block in memory is that the amount of data needed to store the compressed image varies from image to image.

The final block of memory is the compressed image data. For encoding of the image, this block of memory is write-only and for decoding of the image, this block is read only for the system. The size of this block is listed as N/A in Table 5.1, because nothing comes after it, so it has as much room as it needs, except for the end of memory constraint, to store this data. The size of the compressed data should never be more than 16,384 words or else negative compression has occurred. The actual amount of memory allocated for this block is 491,516 words, which is more than enough.

Functions that the memory controller implements and their descriptions can be seen in the table below.

Function Name	**Function Description**
DONOTHING	Used when the memory controller should be idle.
GETIMAGESIZE	Returns the size of the image.
GETPIXELGROUP	Returns a group of 16 pixels.
GETRI	Returns r_i for a given context.
GETSI	Returns s_i for a given context.
GETFUNCTEMPLATE	Returns the function and template.
GETWORD	Returns a word from the compressed data memory block.
SETPIXELGROUP	Sets the value of a group of 16-bit pixels.
SETRI	Sets r_i for a given context.
SETSI	Sets s_i for a given context.
SETWORD	Sets a word in the compressed data memory block.
SETCOMPRESSEDSIZE	Sets the word in the compressed size memory block.

Table 5.2 – Memory Controller Functions

The DONOTHING function is used when the memory controller is not needed and other processing is being performed. The GETIMAGESIZE function is used only at the very beginning of execution, both for encoding and decoding, to retrieve the size of the image. The GETPIXELGROUP function is used both for encoding and decoding to read a portion of the image data. For decoding it is used to generate the context and for encoding it is used to generate the context and get the pixel values. The GETRI and GETSI functions are used to

retrieve the current values for r_i and s_i. These functions are used for both encoding and decoding. The GETFUNCTIONTEMPLATE function is used at the beginning of execution to check whether encoding or decoding is to be performed and which template to use. The GETWORD function is used to get a word from the compressed data region of memory and is used only for decoding. The SETPIXELGROUP function is used to set the value of a 16-bit group of pixels in the image data region of memory and is used only for decoding. The SETRI and SETSI functions are used to set the values of r_i and s_i in the table and are used both for encoding and decoding. The SETWORD function is used to set the value of a word in the compressed data region of data and is only used for encoding. The SETCOMPRESSEDSIZE function is only called once at the very end of encoding to set the size of the compressed data.

The block diagram below describes the interface for the memory controller.

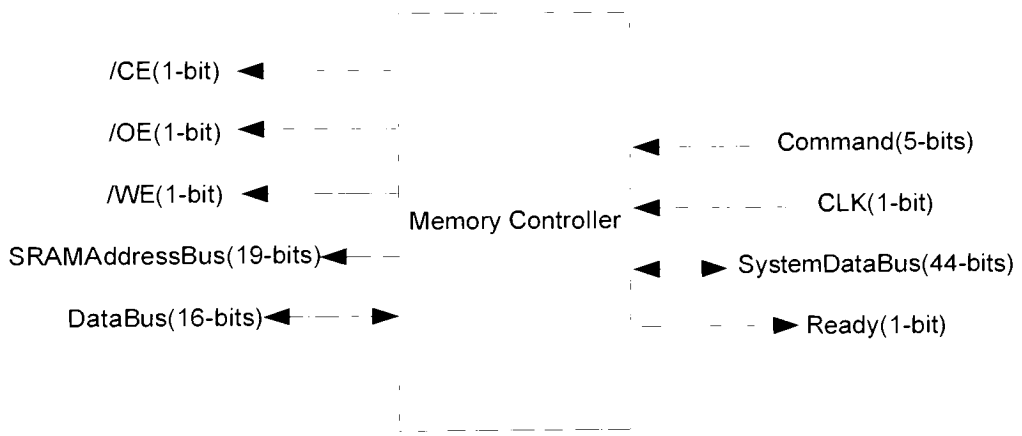


Figure 5.3 – Memory Controller Interface Diagram

The left side of the figure above shows the interface to the actual SRAM on the board or the SRAM simulator. The inputs and outputs on the right size of the interface diagram show the interface to the BACIC system. The Command is used to tell the memory controller which function to perform. The CLK is the global clock connected to all components except the asynchronous SRAM. The SystemDataBus is used to pass parameters into the memory controller as well as get return data from the memory controller. The length of 44 bits was chosen to allow for all commands to be done without complex parameter passing. The commands that need the largest parameter size and return value size are the GETRI, GETSI,

SETRI and SETSI functions. The GETRI and GETSI both need 12-bits for the context as an input and 32-bits as the return type. The SETRI and SETSI both need 12-bits for the context as an input and 32-bits as the value for an input. The Ready bit is used to tell the component using the memory controller that the read or write is finished.

As can be seen in Figure 5.4 above, there are 11 total states, 4 of them for reading and 6 of them for writing and one common state. The left side of the state diagram shows the reading states. In the R_INIT_READ1 state the address is presented on the address bus and the OE line is applied. This data is read and then returned to the caller. For some functions, GETIMAGESIZE, GETRI and GETSI, two 16-bit reads need to be performed. If any of these commands are called then the second 16-bit word is read and the data is then returned.

In the R_PREPAREDATA state the address lines and the OE line are applied and the memory fetch occurs. In the R_RETURNDATA state the data is presented on the SystemDataBus to be returned to the caller. The R_INITREAD2 line is when a second read is needed by the system.

The writing functions are a little more complicated. The reason for this is that if either the address bus or the data bus changes while the write is being performed, then the wrong data could be written or the data could be written to the wrong address. This is not an issue in the reading process, because as long as the data is read at the correct time, then the address bus and data bus can change and nothing is going to happen to the memory. As a result, an extra state is used in the writing process where the WE line is desasserted before changing the address bus or data bus.

The functionality for the memory controller is implemented using a Moore State Machine. The diagram below shows the state diagram for this module.

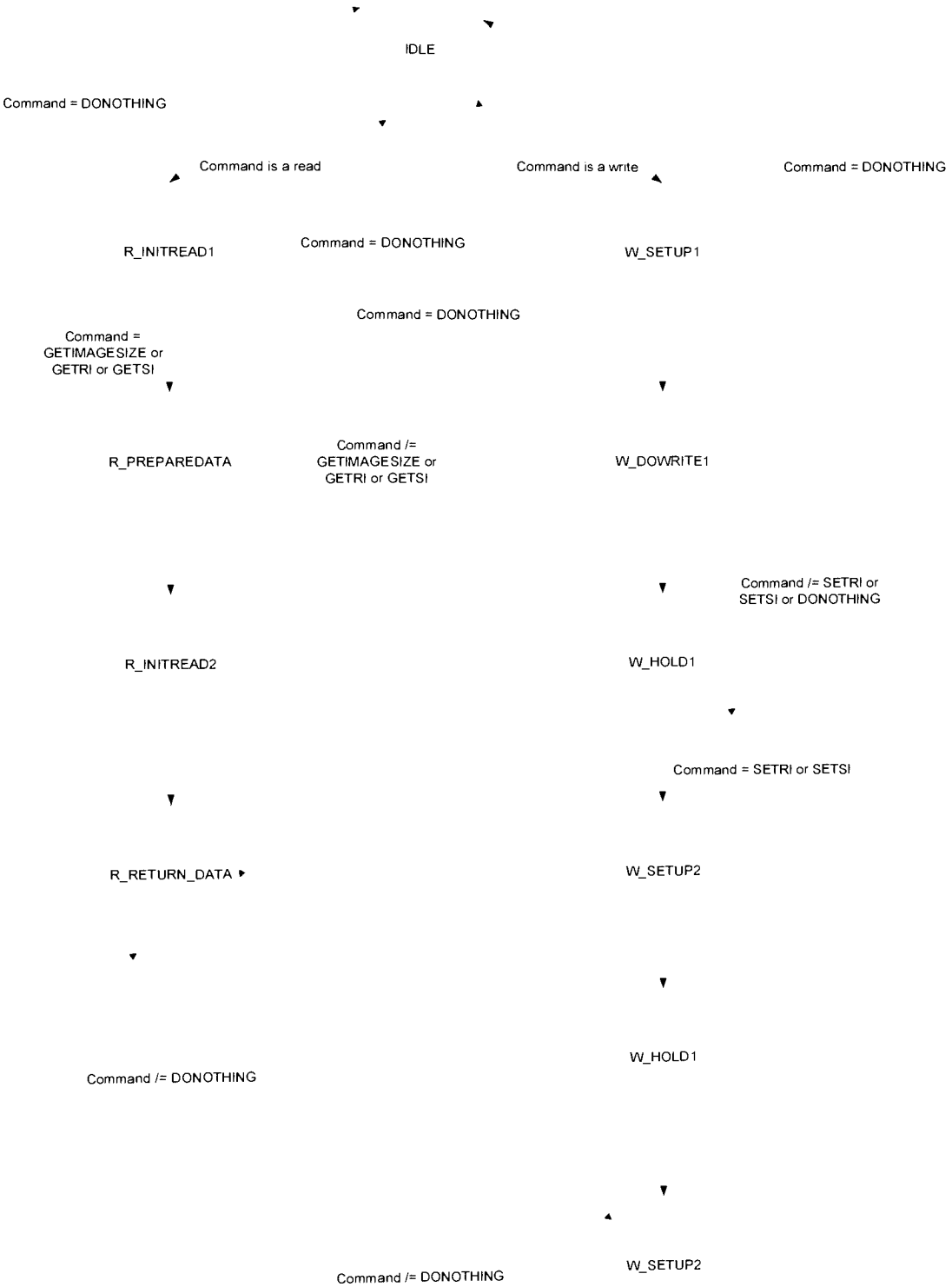


Figure 5.4 – Memory Controller State Diagram

The first state when performing a write function is W_SETUP1. In this state the address bus and data bus are written. In the W_DOWRITE1 state the WE line is applied and the actual write to memory is performed. In the W_HOLD state the WE line is deasserted, but the address and data lines are held to the same value as the previous state. The reason for this is described above. If the address and data bus were changed in the same state as the WE line, then the outputs could change in any order. Similar to the read functions, there are some write functions which take two writes instead of one: SETRI and SETSI. If either of these is the function, then the second write is performed.

When either the read or write function is completed the MemReady bit is set at which point the caller must set the command to DONOTHING and the memory controller returns to the IDLE state. The table below shows the number of clock cycles needed for each command.

Function Name	**Number of Clock Cycles Needed**
GETIMAGESIZE	5
GETPIXELGROUP	3
GETRI	5
GETSI	5
GETFUNCTEMPLATE	3
GETWORD	3
SETPIXELGROUP	4
SETRI	7
SETSI	7
SETWORD	4
SETCOMPRESSEDSIZE	7

Table 5.3 – Memory Controller Delays

As can be seen in the above table the longest memory read, where two 16-bit reads are necessary, takes five clock cycles. The longest memory writes, where two 16-bit writes are needed, takes seven clock cycles. These delays are adequate as will be seen in the

encoder/decoder section, because while the memory is reading or writing other processing can take place.

The functionality provided by the memory controller provides all the necessary tasks that the BACIC encoder/decoder and its sub-components need for memory access. These functions allow an adequate amount of processing in this module so that it does not have to be overly complex, but provides enough functionality so that a small amount of memory interface handling has to be done to use the memory controller.

5.5 Divider Module

The division operation is one aspect of BACIC, as well as other arithmetic coders, which makes it more difficult to implement in hardware and takes longer to execute. In order to compute p_l a division operation is performed, as shown in Eqn. 3.4. Because the Xilinx XCV-300 does not include a built in divider circuit, a divider needed to be implemented in VHDL.

Because p_l is always be in the range of 0.0 to 1.0, only a fractional part of the quotient is needed. This feature reduces the amount of time needed to complete a division as well as the complexity. Through simulation it was determined that the quotient would need to be 24 bits in length in order to provide the precision needed by the BACIC codec. Below is the interface diagram for the divider created for this system.

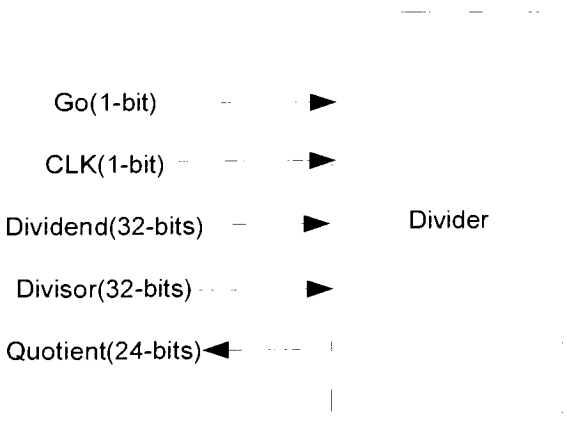


Figure 5.5 – Divider Interface Diagram

The CLK bit is used, because the divider was chosen to be synchronous so that the rest of the system would know when the division was completed and also so that the steps of the division could be broken up to allow it to run more efficiently. The Dividend and Divisor are the inputs to the divider block and the Quotient is the fractional output. The Go bit is used to start the divider. After the Dividend and Divisor have been loaded with their values, the Go bit is asserted and the divider starts. After 24 clock cycles the quotient is available.

The divider works very well for this system. It provides enough precision to allow the encoding to be efficient and reversible and it does not provide any costly overhead processing.

5.6 Context Generator Module

Other than the BACIC encoder/decoder the context generator module is the most complicated block in the system. Not only does it do all the memory reads necessary to get the context for a given pixel, but it can also return a pixel value. For decoding the context generator is also responsible for writing the pixel values to the memory.

The reason for giving the context generator so much functionality is speed. The context generator is going to need to do memory reads in order to retrieve the context for a pixel, so if it is already going to have the pixel values there is no reason to have another module do a read to obtain the pixel value. The reasoning for having the context generator handle the pixel writes when decoding follows the same lines. The context generator is going to have to know the values of the previous pixels to obtain the context for them, so it only makes sense for it to do the writes as well.

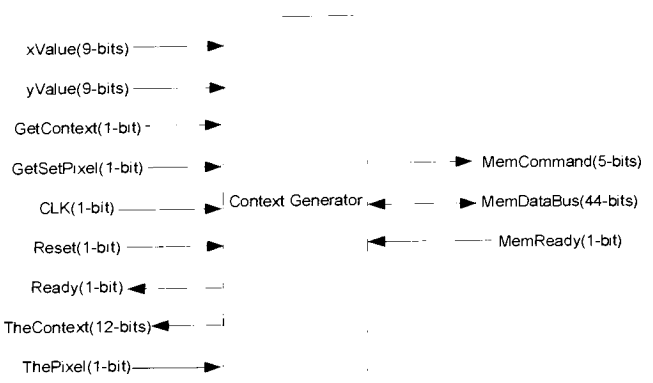


Figure 5.6 – Context Generator Interface Diagram

The interface diagram for the context generator module can be seen in Figure 5.6. The right side of the diagram connects to the memory controller as described above in Section 5.4. The left side of the interface diagram shown in Figure 5.6 connects to the BACIC encoder/decoder module.

The xValue and yValue parameter are used for getting a context, getting a pixel and setting a pixel. These two parameters provide the location of the current pixel. The getContext and getSetPixel inputs tell the context generator which function to perform. When the getContext input is asserted it gets the context for the pixel at the location given by the xValue and yValue parameters. The getSetPixel is used to either get or set the value of a pixel. If the function is encoding, then that input forces the context generator to get the value of a pixel, and if the function is decoding a pixel is written.

The Reset line is used only at the very beginning of simulation in order to reset the context generator. The reason why reset lines are used is because two of the components, the context generator and the BACIC encoder/decoder, are sharing a common memory controller. Therefore, their actions need to be controlled so that neither of them reset at the same time and then try to access memory at the same time. The Ready output tells the encoder/decoder that the value of the context or pixel has been returned or the write is complete and it is ready for the next command. The theContext output returns the value of the context upon a getContext command or returns the value of the pixel, in the lowest bit, for a getPixel command. The thePixel input is used to specify the value of the pixel to be written.

This module allows for the encoder/decoder to get both the current pixel and the context of that pixel. It also allows for a set pixel to occur in the case of decoding. This module provides enough functionality so that the encoder/decoder module does not have to be more complex than needed, but also keeps this component not overly complex.

5.7 BACIC Encoder/Decoder Module

The BACIC encoder/decoder module is by far the most complex module in the system. Not only does it have the ability to interface with all other modules in the system but it also does the

actual encoding or decoding of the data. Figure 5.7 below shows the interface diagram for this module.

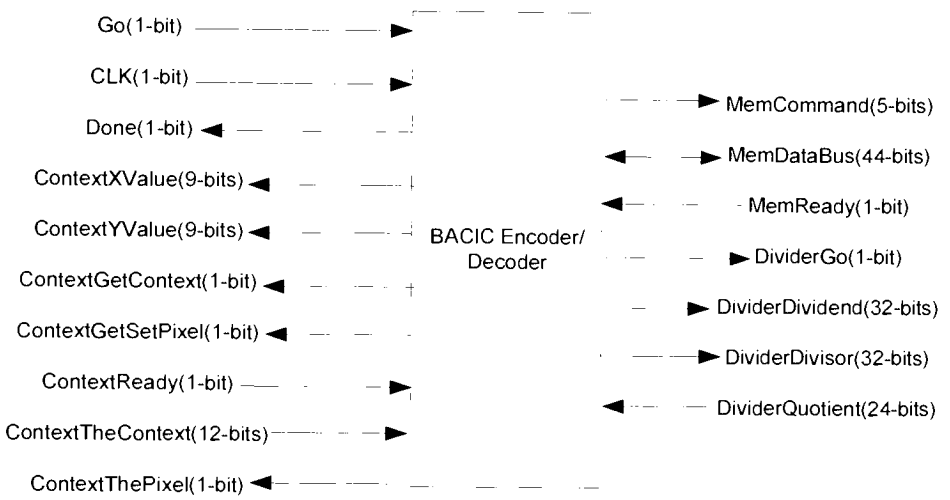


Figure 5.7 – BACIC Encoder/Decoder Interface Diagram

As can be seen in Figure 5.7 most of the inputs and outputs to this module are for interfacing with other modules. On the right side of the figure above is the interface to the memory controller and divider modules. On the left side is the interface to the context generator and the control lines for this module. The Go input tells the encoder/decoder to start processing. The CLK input is the global system clock and the Done output informs the system that the encoding or decoding process is complete. The majority of the functionality for this module is implemented using 2 Moore State Machines, one for encoding and one for decoding.

5.7.1 BACIC Encoder

Figure 5.8 shows the state diagram for the encoding state machine of this module. As can be seen, the state diagram of the BACIC encoder is quite complex. The system starts in the GETFIRSTCONTEXT state. The GETFIRSTCONTEXT, GETFIRSTPIXEL1 and GETFIRSTPIXEL2 are then executed once and only once at the beginning of execution. These three states together get the first pixel and the first context. The reason why no r_i or s_i fetch or division is needed for the encoding of the first pixel is because the p_1 value for the first pixel

Figure 5.8 – BACIC Encoding State Diagram

is always 0.5, because nothing is known about the image yet. The reason why a get context is done for the first pixel even though the context of the first pixel will always be 0 is that a requirement of the context generator is that a get context must always be done before a get pixel.

The division operation takes 24 clock cycles to complete. In order to speed things up and not let these 24 clock cycles become wasted time, the next context and pixel values are fetched as well as the current r_i and s_i values being written and the next r_i and s_i values being read during this period. The reason why the writes of r_i and s_i can be done ahead of time is the only thing that affects the new values of r_i and s_i are the pixel value, and, since that is already known for the current pixel, the values of r_i and s_i to be written are known. This speeds up the execution time.

Therefore, with the exception of the first pixel, the DISPATCH state would be the first state to run. In this state the division is started and the time until the completion of the division is computed. Next, while the division is occurring the next context and next pixel values are found in the states GETNEXTCONTEXT1, GETNEXTCONTEXT2 and GETNEXTPIXEL.

One big issue when doing the pre-fetch of the context, the pixel value, r_i and s_i is what happens when the same context occurs for two sequential pixels. If nothing is done to handle this, then the pre-fetch would have gotten the old values of r_i and s_i and the result would be incorrect. Therefore, in the case where two sequential pixels have the same context the read and write of r_i and s_i is skipped and the values for the current iteration are used in the next iteration.

However, in the normal case where the next context is not the same as the current context, after reading the pixel value and context value the next step would be to write the r_i and s_i values which is done in the WRITERI1, WRITERI2, WRITESI1, and WRITESI2 states. After the write is complete, the fetch of the next r_i and s_i values is done in the READRI1, READRI2, READSI1 and READSI2 states.

After the write and read of r_i and s_i is completed, the next step is the encoding. The encoding was broken up into two clock cycles in order to keep the clock period shorter, even though it could have been done in one cycle. After the encoding of the current pixel is complete, the next step is to see if a write needs to be done.

If the current node value is a 1 or this is the last pixel in the image then a write is required. If not, the system goes to the DISPATCH state and the process starts all over again. Because the memory is using 16-bit words for memory access and the value for K is 12, meaning that each codeword is 12-bits, there can potentially be a lot of wasted space. If each 12-bit value was written using a 16-bit word, that would be 4 bits of wasted space per codeword. In order to overcome this, each codeword write will actually write 4 codewords (48-bits), which is equivalent to 3 16-bit words.

Therefore, every time a leaf node is encountered it does not necessarily mean that a write has to occur. The WRITEDATA1 state checks to see if a write is required. In the WRITEDATA2 and WRITEDATA3 state the 16-bit words are actually written. In the WRITEDATA4 state the system checks to see if the writing of the codewords is done, if it is not the system goes back to the WRITEDATA2 state. If the writing of codewords is done then the system goes to one of two states. If the entire image has been encoded then the system goes to the WRITEDATA5 state and if the encoding is not completed it goes to the DISPATCH state where the process starts all over. In the WRITEDATA5 and WRITEDATA6 states the compressed size of the image is written and then the encoder stops and the encoding is complete.

The table below shows the number of states visited as well as the number of clock cycle needed in order to encode one pixel for several different scenarios. Note that the encoding of the first and last pixel is ignored in this table, because those only happen one time and in the big scheme of things do not really affect the execution time.

As can be seen in Table 5.4 the number of total clock cycles needed to encode one pixel can range anywhere from 8 to 46. One thing that was not taken into consideration is the case where

Scenario	Number of states visited	Number of clock cycles
No duplicated context, not a leaf node	15	31
No duplicate context, a leaf node and write required	24	46
Duplicate context, not a leaf node	8	8
Duplicate context, a leaf node and write required	17	23

Table 5.4 – Number of Clock Cycles Needed for Encoding

the context generator needs to do a read on a pixel group. This occur about every 16-pixels and adds between 3 and 15 clock cycles depending on which template is used and which image row is being processed. The most common case is the first one, no duplicate context and not a leaf node. So on average the number of clock cycles needed to encode one pixel is around 31. Keeping in mind that the encoding of most pixels requires two 32-bit memory reads and two 32-bit memory writes this encoder is reasonably fast.

5.7.2 BACIC Decoder

As will be seen shortly the BACIC encoder and decoder are similar and share a lot of the same states. Figure 5.9 shows the state diagram for the Moore State Machine used in the BACIC decoder.

Similarly to the encoder, the decoder starts by reading the first context. The first context is always 0, but this is done to initialize the context generator. The next state is the READDATA1 state. This state checks to see if more codewords are needed, and if they are, it goes to the READDATA2 and READDATA3 states where the codewords are read. Similarly to the encoder, three 16-bit words (four 12-bit codewords) are read at once.

Figure 5.9 – BACIC Decoder State Diagram

After the codewords are read, the r_i and s_i values are read in the GETRI1, GETRI2, GETSI1 and GETSI2 states. Once the memory is read the system goes to the DISPATCH state, which starts the division to compute p_1.

After the division is started the system goes to state DECODING1. Unfortunately nothing can be done while the division is taking place. This is because while the division is taking place the value of the current pixel is not known yet, so the next context cannot be found and because the r_i and s_i values to be written depend on the current pixel, they cannot be written. Therefore, while the division is performed, the system just waits the 24 clock cycles.

Once the division is done, the decoding of the current pixel is performed in three steps. Three steps are used in this case vs. the encoding having two steps, because things can be done ahead of time in the encoding, but they cannot for the decoding. After the three decoding states have finished the current pixel value is known as well as the r_i and s_i values to be written.

The next step is to get the next context to see if two sequential pixels have the same context. If they do, then the r_i and s_i values are not written for the current pixel or read for the next pixel. If there is not a duplicate context, then the system writes the r_i and s_i values in the WRITERI1, WRITERI2, WRITESI1 and WRITESI2 states. The system then returns to the READDATA1 state where the validity of the current codewords is checked and the process starts over again.

Unfortunately decoding an image requires a lot more time than encoding, because of the time wasted during the division. The table below shows the number of states visited as well as the number of clock cycles needed to decode a single pixel for several different scenarios.

Scenario	**Number of states visited**	**Number of clock cycles**
No duplicated context, not a leaf node	17	56
No duplicate context, a leaf node and read required	23	65
Duplicate context, not a leaf node	11	33
Duplicate context, a leaf node and read required	15	42

Table 5.5 - Number of Clock Cycles Needed for Decoding

One thing that was not taken into consideration is the clock cycles needed when the context generator needs to perform a SETPIXELGROUP. This should happen every 16 pixels and consumes 4 clock cycles. Also, the context generator needs to occasionally read pixel values in order to compute the context. This can take between 3 and 15 clock cycles depending on which template is being used and what image row is being processed. This should also occur every 16 pixels.

As can be seen when comparing Table 5.4 and Table 5.5, the decoder takes a lot more clock cycles to run than the encoder. However, as explained earlier there is no way around this. One setback of decoding is that the value of the current pixel is not known ahead of time so no pre-fetching can be done.

5.8 System Module

The final module in the BACIC hardware system is the system module. This module contains all the other modules and provides the interconnections between them. It also provides the high level logic for the system. It synchronizes the resets of all the components that need them as well as starting the encoder/decoder module and recording when it finishes. Figure 5.10 shows the system diagram for the entire BACIC encoder/decoder.

As can be seen, the system module instantiates all the other modules in the hardware system. It also makes the connections between all the modules. There are some input/outputs (I/O) for the system, a majority of which goes to the SRAM on the prototype board. As expected the outputs include all the control lines to the SRAM: CE, OE and WE. The I/O includes the data bus and the address bus for the SRAM module and control inputs and outputs that come to the system module. The CLK input comes from the oscillator on the prototype board and provides a synchronized clock signal to all the synchronous components. The EncodingDone output allows a user see when the encoding or decoding is complete. The Start input instructs the system to begin processing. The reason why this input, as well as the Reset for the context generator and the Go for the BACIC encoder/decoder, is not attached to anything in this figure is because it is controlled and used by internal circuitry for this module.

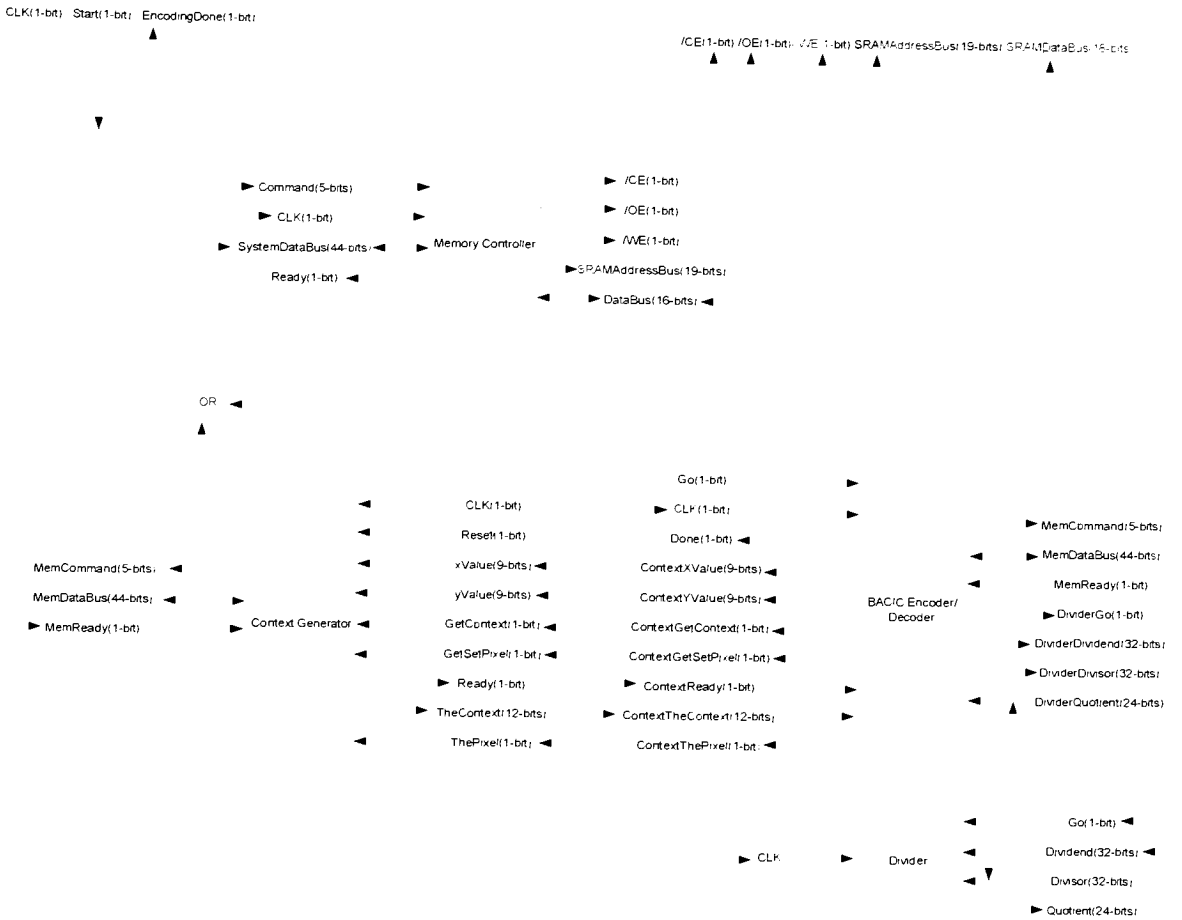


Figure 5.10 – System Module Architecture

Because there are two modules that need to use the memory controller, the context generator and the BACIC encoder/decoder, logic is needed to allow the sharing of this resource. The DONOTHING command for the memory controller has a command value of "00000". Therefore, the memory command output from the context generator and the memory command output from the BACIC encoder/decoder can be OR'd together in order to share the memory command bus. The other input to the memory controller that must be shared is the SystemDataBus. To implement this bus, 44 tri-state buffers are used. This way when either of the modules is not using the memory controller it can set the SystemDataBus to high impedance.

The logic behind this module is very simple when compared to the other modules. This module needs to do only a few steps. The first step is to reset the context generator. Then it tells the encoder to begin and waits until it finishes. This module is crucial to the system, however, because it instantiates all of the other modules as well as connects them together. It also provides logic that the other modules need such as the OR gate on the Command lines to the memory controller.

5.9 Testbench Module

A testbench module was created for simulation purposes. This module would connect the system module to the SRAM model module and start the BACIC encoder/decoder doing either compression or decompression depending on the simulation. The block diagram below shows the internal connections and components of the testbench.

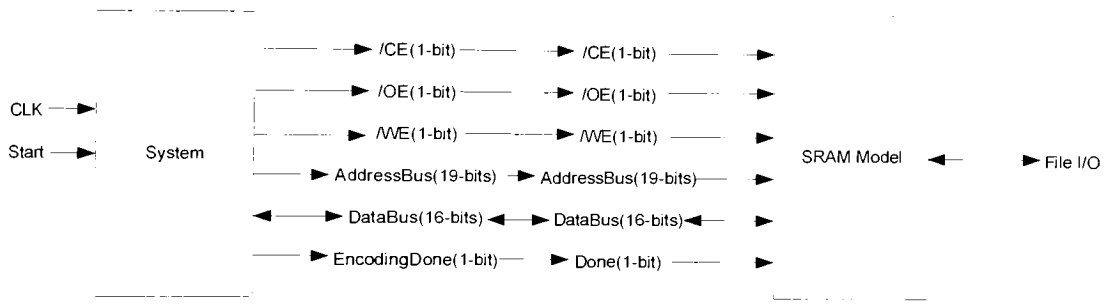


Figure 5.11 – Testbench Module Architecture

As can be seen Figure 5.11 the testbench module connects the inputs and outputs of the system module and SRAM module. The two unconnected inputs to the system, CLK and Start, are internal signals to this module. The CLK is implemented using a signal assignment statement with an after clause. The Start signal is asserted at the start of the simulation. This testbench provides a good environment for the simulation of the system in order to verify its functionality. Also, the file input and output from the SRAM Model is shown here.

Chapter 6. Software and Hardware Performance

This chapter presents results from both hardware and software implementations. First, results are presented on bitonal and grayscale lossless image compression ratios using a software implementation. Next, results are shown for bitonal lossy image compression with region of interest consideration. Finally hardware results are presented to show that the hardware implementation works correctly and efficiently.

6.1 Lossless Bitonal Image Simulation Results

Several sample images were used to determine algorithm performance. In order to provide valid results twelve common image processing images were used as well as the eight standard CCITT document-type images.

Image Name	Image Size
ccitt1	1728x2376
ccitt2	1728x2376
ccitt3	1728x2376
ccitt4	1728x2376
ccitt5	1728x2376
ccitt6	1728x2376
ccitt7	1728x2376
ccitt8	1728x2376

Table 6.1 – Test Documents

Image Name	Image Size
airplane	512x512
baboon	512x512
barbera	720x576
boat	512x512
bridge	256x256
café	816x1024
camera	256x256
couple	256x256
goldhill	512x512
lena	512x512
monarch	768x512
peppers	512x512

Table 6.2 – Test Images

Table 6.1 lists all of the document-type images used and Table 6.2 lists all of the pictorial images used. The size of the pictorial images ranged from 256x256 to 816x1024, while the CCITT documents were all 1728x2376 pixels. The 12 pictorial test images originally were grayscale images and then were halftoned in order to apply the bitonal image compression algorithms. The original grayscale images as well as the halftoned clustered-dot dithered images, dispersed-dot dithered images and the error diffused images can be found in Appendix B.

In order to provide valid results, all of the algorithms presented in Chapter 2 were used to compress the test images and each of their resulting compression ratios recorded. Tables 6.3(a-b) show the compression results when each of the CCITT documents were compressed. Table 6.3(a) shows the compression ratios. Table 6.3(b) shows the results in bits per pixel. Because these are bitonal images, if the bits per pixel value is less than 1.0, then compression has occurred.

Image	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
ccitt1	15.052	23.107	27.986	30.359	34.246	29.008
ccitt2	16.534	34.421	46.495	57.087	60.110	43.023
ccitt3	8.317	15.640	17.734	21.621	23.663	19.845
ccitt4	4.900	6.994	7.383	8.704	9.737	8.430
ccitt5	7.897	14.127	15.813	18.199	20.132	16.942
ccitt6	10.727	24.724	30.393	37.887	40.985	30.684
ccitt7	4.978	6.993	7.383	8.430	9.153	8.091
ccitt8	8.629	22.116	26.545	33.791	36.352	26.089
	9.629	**18.515**	**22.467**	**27.010**	**29.297**	**22.764**

Table 6.3(a) – Lossless Document Compression Ratios

Image	Raw Data	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
ccitt1	1	0.066	0.043	0.036	0.033	0.029	0.034
ccitt2	1	0.060	0.029	0.022	0.018	0.017	0.023
ccitt3	1	0.120	0.064	0.056	0.046	0.042	0.050
ccitt4	1	0.204	0.143	0.135	0.115	0.103	0.119
ccitt5	1	0.127	0.071	0.063	0.055	0.050	0.059
ccitt6	1	0.093	0.040	0.033	0.026	0.024	0.033
ccitt7	1	0.201	0.143	0.135	0.119	0.109	0.124
ccitt8	1	0.116	0.045	0.038	0.030	0.028	0.038
		0.104	**0.054**	**0.045**	**0.037**	**0.034**	**0.044**

Table 6.3(b) – Lossless Document Compression Results (bits per pixel)

As can be seen when inspecting Tables 6.3(a-b), the algorithm with the best compression ratio is JBIG2, the second highest is JBIG and the third highest is BACIC. When considering document-type images BACIC does not provide as good compression efficiency as JBIG or JBIG2. However, BACIC does perform better than the Group3 and Group4 algorithms. As expected the Group3 two-dimensional algorithm performs better than the one-dimensional algorithm and the Group4 algorithm performs better than both of the Group3 algorithms.

Tables 6.4(a-b) shown below present the compression results for the images shown in Table 6.2 using the clustered-dot dither method of halftoning.

Image	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
airplane	0.689	0.545	0.553	6.165	6.363	6.215
baboon	0.590	0.531	0.538	2.930	3.183	3.082
barbera	0.599	0.566	0.571	3.107	3.731	3.511
boat	0.621	0.571	0.579	4.600	5.042	4.811
bridge	0.599	0.542	0.557	2.606	2.917	3.023
café	0.665	0.622	0.628	2.565	2.977	2.902
camera	0.649	0.570	0.586	3.298	4.078	4.738
couple	0.589	0.541	0.555	3.660	4.186	4.746
goldhill	0.611	0.557	0.565	4.791	5.595	5.322
lena	0.603	0.547	0.554	4.515	5.697	5.974
monarch	0.585	0.539	0.543	5.037	5.968	5.700
peppers	0.609	0.555	0.562	4.599	5.649	5.679
	0.617	**0.557**	**0.566**	**3.990**	**4.615**	**4.642**

Table 6.4(a) – Compression Ratios for Lossless Clustered-Dot Dithered Images

Image	Raw Data	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
airplane	1	1.450	1.833	1.809	0.162	0.157	0.161
baboon	1	1.695	1.882	1.857	0.341	0.314	0.324
barbera	1	1.670	1.768	1.751	0.322	0.268	0.285
boat	1	1.610	1.752	1.728	0.217	0.198	0.208
bridge	1	1.668	1.845	1.796	0.384	0.343	0.331
café	1	1.504	1.608	1.592	0.390	0.336	0.345
camera	1	1.541	1.754	1.705	0.303	0.245	0.211
couple	1	1.697	1.850	1.801	0.273	0.239	0.211
goldhill	1	1.635	1.794	1.769	0.209	0.179	0.188
lena	1	1.660	1.828	1.804	0.221	0.176	0.167
monarch	1	1.710	1.857	1.841	0.199	0.168	0.175
pepper	1	1.643	1.802	1.778	0.217	0.177	0.176
		1.620	**1.798**	**1.769**	**0.270**	**0.233**	**0.232**

Table 6.4(b) – Lossless Clustered-Dot Compression Results (bits per pixel)

As can be seen in Tables 6.5(a-b), BACIC performs better than all the other algorithms for clustered-dot dithered images. The second best algorithm is JBIG2, which provides compression efficiency very close to what BACIC provides. The Group3 and Group4 algorithms do not perform well when compressing halftoned images; they actually expand the images instead of compressing them. This is understandable, however, because the Group3 and Group4 algorithms were modeled around compressing document-type images rather than halftoned images.

Tables 6.5(a-b) present results for images halftoned using the dispersed-dot dithering method.

Image	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
airplane	0.473	0.335	0.338	5.162	6.263	6.149
baboon	0.312	0.306	0.308	2.605	3.024	3.028
barbera	0.338	0.331	0.326	2.879	3.637	3.484
boat	0.331	0.315	0.318	4.122	4.848	4.748
bridge	0.339	0.313	0.318	2.426	2.879	3.016
café	0.416	0.370	0.372	2.382	2.955	2.900
camera	0.381	0.338	0.344	3.095	4.059	4.730
couple	0.287	0.302	0.307	2.851	3.986	4.657
goldhill	0.326	0.316	0.318	4.177	5.131	5.195
lena	0.325	0.307	0.309	4.484	5.305	5.868
monarch	0.316	0.303	0.304	4.776	5.737	5.736
peppers	0.349	0.311	0.314	4.566	5.566	5.790
	0.349	**0.321**	**0.323**	**3.627**	**4.449**	**4.608**

Table 6.5(a) – Compression Ratios for Lossless Dispersed-Dot Dithered Images

Image	Raw Data	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
airplane	1	2.116	2.983	2.958	0.194	0.160	0.163
baboon	1	3.203	3.271	3.247	0.384	0.331	0.330
barbera	1	2.956	3.025	3.066	0.347	0.275	0.287
boat	1	3.023	3.173	3.148	0.243	0.206	0.211
bridge	1	2.954	3.195	3.145	0.412	0.347	0.332
café	1	2.402	2.703	2.689	0.420	0.338	0.345
camera	1	2.626	2.960	2.909	0.323	0.246	0.211
couple	1	3.486	3.308	3.259	0.351	0.251	0.215
goldhill	1	3.067	3.169	3.144	0.239	0.195	0.192
lena	1	3.076	3.260	3.236	0.223	0.189	0.170
monarch	1	3.163	3.303	3.288	0.209	0.174	0.174
pepper	1	2.867	3.211	3.188	0.219	0.180	0.173
		2.862	**3.130**	**3.106**	**0.297**	**0.241**	**0.234**

Table 6.5(b) – Lossless Dispersed-Dot Compression Results (bits per pixel)

The results shown in Tables 6.5(a-b) match the results shown previously in Table 6.4. Again, the BACIC algorithm performs the best with JBIG2 having slightly lower compression efficiency. The compression ratio for JBIG is adequate, compressing the image to less than one-third of its original size. Similarly to the clustered-dot dithered image results, the Group3 and Group4 algorithms expand the image rather than compressing it. This property makes them unacceptable for compressing halftoned images.

Image	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
airplane	0.491	0.345	0.348	1.623	2.093	2.056
baboon	0.340	0.371	0.374	1.162	1.364	1.438
barbera	0.369	0.388	0.391	1.238	1.566	1.585
boat	0.336	0.390	0.393	1.400	1.867	1.872
bridge	0.371	0.367	0.374	1.406	1.284	1.427
café	0.464	0.440	0.443	1.227	1.503	1.524
camera	0.394	0.423	0.432	1.965	1.790	1.943
couple	0.298	0.370	0.377	1.789	1.649	1.812
goldhill	0.349	0.371	0.375	1.406	1.901	1.884
lena	0.342	0.367	0.371	1.389	2.012	1.990
monarch	0.350	0.355	0.357	1.389	2.022	1.925
peppers	0.374	0.368	0.372	1.380	2.012	1.973
	0.373	**0.380**	**0.384**	**1.448**	**1.755**	**1.786**

Table 6.6(a) – Compression Ratios for Lossless Error Diffused Images

Image	Raw Data	Group3 1D	Group3 2D	Group4	JBIG	JBIG2	BACIC
airplane	1	2.037	2.897	2.873	0.616	0.478	0.486
baboon	1	2.942	2.696	2.673	0.860	0.733	0.695
barbera	1	2.707	2.575	2.557	0.808	0.638	0.631
boat	1	2.979	2.565	2.541	0.715	0.536	0.534
bridge	1	2.698	2.723	2.675	0.711	0.779	0.701
café	1	2.156	2.270	2.255	0.815	0.665	0.656
camera	1	2.540	2.364	2.314	0.509	0.559	0.515
couple	1	3.355	2.700	2.653	0.559	0.606	0.552
goldhill	1	2.862	2.693	2.667	0.711	0.526	0.531
lena	1	2.925	2.722	2.699	0.720	0.497	0.502
monarch	1	2.854	2.817	2.801	0.720	0.495	0.519
pepper	1	2.675	2.715	2.690	0.725	0.497	0.507
		2.680	**2.645**	**2.617**	**0.706**	**0.584**	**0.569**

Table 6.6(b) – Lossless Error Diffused Compression Results (bits per pixel)

The final part of this section presents the compression results for images halftoned using the error diffusion method and can be found in Tables 6.6(a-b). As shown in these tables, BACIC again performs better than all the other algorithms with a compression ratio of 1.786 on average. Again JBIG2 provides the next highest compression efficiency and the third highest compression efficiency is given by JBIG. Similarly to the dithered images, the Group3 and Group4 algorithms expand the images rather than compressing them.

When comparing Tables 6.4-6.6, it can be seen that clustered-dot dithered images can be compressed more efficiently than dispersed-dot dithered images or error diffused images. Error diffused images compress the least efficiently out of the three. The reason why dithered images compress better than error diffused images is that the dither mask generates a pattern over the halftoned image, while the error diffusion process does not. This pattern makes the current pixel easier to predict, therefore, making the value of p_1 or p_0 higher, meaning the event is more certain. The reason why clustered-dot dithered images compress more efficiently than dispersed-dot dithered images is because with a clustered-dot dithered image the pixels close to the current pixel are more likely to have the same value as the current pixel and since those pixels close to the current pixel are used to predict the value of the pixel, it would be a better prediction.

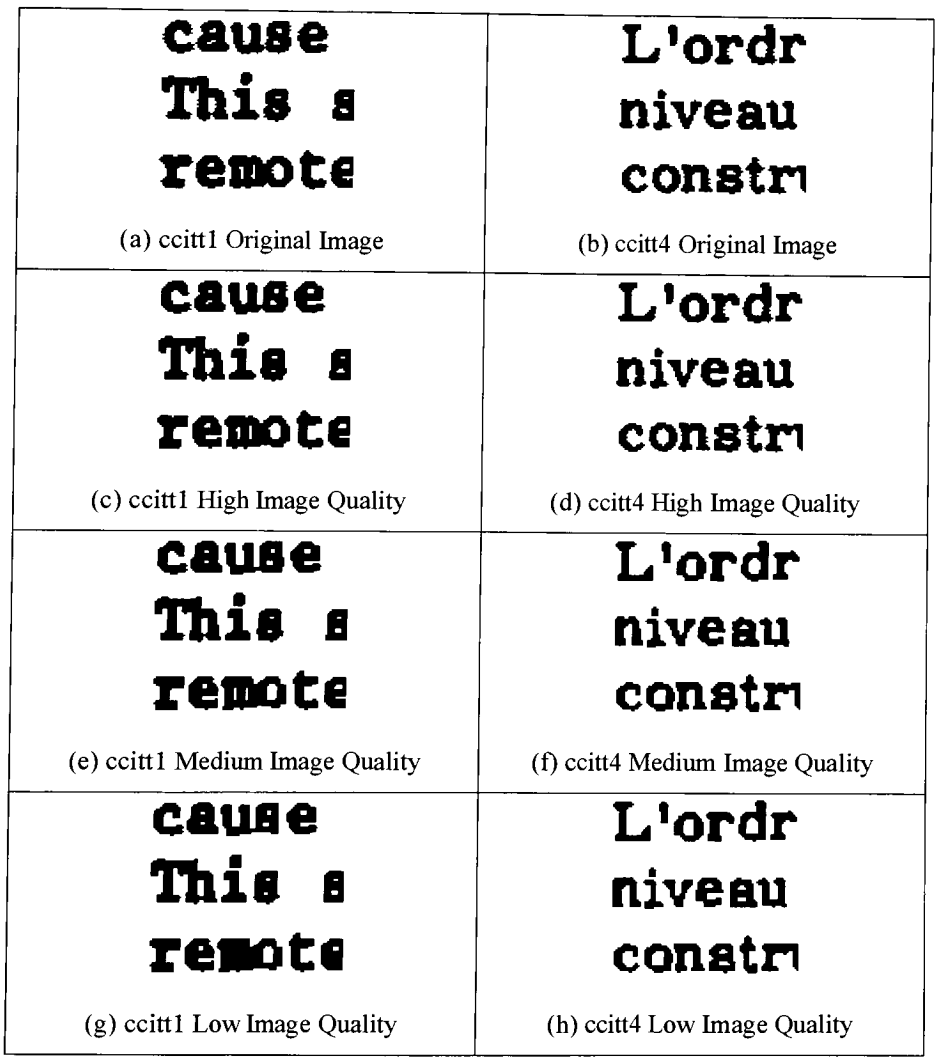
The results of this section illustrate that BACIC does not provide the best compression efficiency for document-type images, but still outperforms Group3 or Group4 algorithms, which are currently being used in the facsimile industry. For images halftoned using the dither method or the error diffusion method, the BACIC algorithm provides the best compression results.

6.2 Near-Lossless Bitonal Image Simulation Results

This next section presents compression results for near-lossless representations of the test images. The level of loss introduced was kept constant throughout the entire image, without any consideration for a region of interest. First results are presented for the document-type images, then for dithered images and finally for error diffused images.

Loss was introduced into each of the 8 document-type images using three levels of loss. The images below show near-lossless images. Figure 6.1 shows two of the document-type images with various quality levels.

The CCITT images were not shown in full, because of their size. Each of the images is 1728x2376 pixels, so even if the image took up the entire page details not could be seen. To overcome this problem image subsections are shown in Figure 6.1. As can be seen when comparing Figure 6.1(a) to Figure 6.1(c) and Figure 6.1(b) to Figure 6.1(d) the images look almost identical. In some cases it appears as though the loss introduced has cleaned up some of the protruding pixels of the images making it crisper. The text is easily legible on the high quality setting.



(a) ccitt1 Original Image

(b) ccitt4 Original Image

(c) ccitt1 High Image Quality

(d) ccitt4 High Image Quality

(e) ccitt1 Medium Image Quality

(f) ccitt4 Medium Image Quality

(g) ccitt1 Low Image Quality

(h) ccitt4 Low Image Quality

Figure 6.1 – Near-lossless Document-Type Images

The medium quality subsections can be found in Figures 6.1(e-f). As can be seen when looking at these images, the lossless image and near-lossless image at medium quality look almost identical and the near-lossless image is still easily legible. At this quality level, again some of the protruding pixels are removed, but bleeding from one character to the next has occurred.

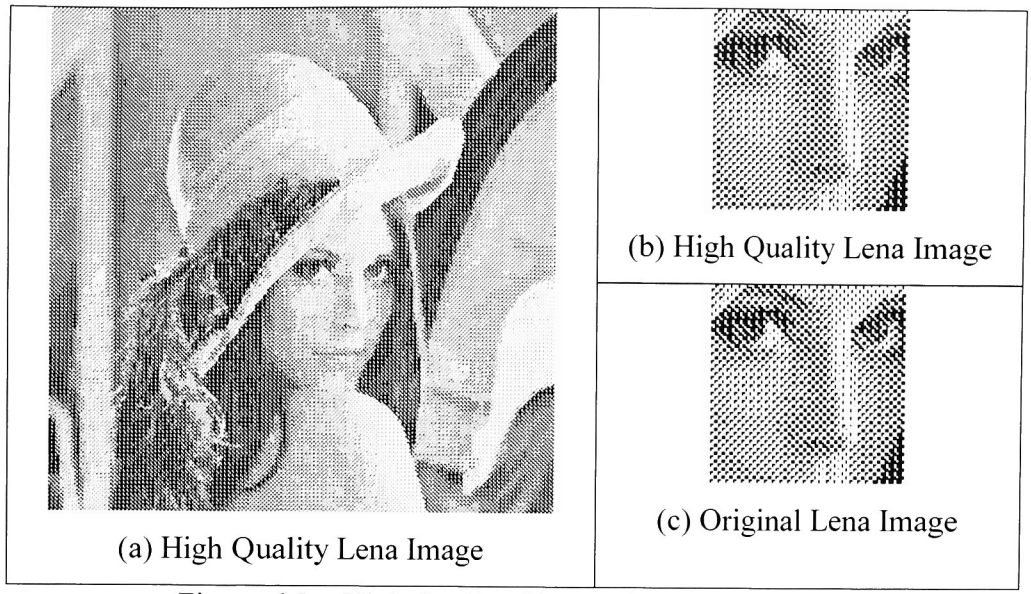
The images in Figures 6.1(g-h) show the CCITT document images at low quality level. At this quality level differences can be seen in the image. The 'm' in Figure 6.1(g) is an example of this loss. The loss introduction algorithm has filled in the white between the humps of the 'm' character. However, even at this quality level, the writing is still legible. Table 6.7 shows the PSNR and compression ratios for each of the different quality levels for the document-type images.

	Perfect	High	Medium	Low
Average PSNR (dB)		24.837	22.628	21.687
Average Compression Ratio	22.764	26.520	29.697	33.144
Bits Per Pixel	0.044	0.038	0.034	0.030
Compression Ratio Increase	-	16.50%	30.46%	45.60%

Table 6.7 – Near-Lossless Compression Results for Document-Type Images

The PSNR shown in Table 6.7 is used rather than using the MPSNR, because there is no grayscale image to compare. Therefore, the PSNR was computed by using pixel errors of 255, giving the PSNR values in Table 6.7. For each quality level it seems the PSNR decreased by almost 2 dB. Also, in Table 6.7 it can be seen that the loss introduction resulted in little loss that could be seen in the document-type images, while the compression ratios increase significantly. When lossless and near-lossless images were compressed using the BACIC algorithm the amount of compression ratio increase due to loss introduction ranged from 16.5% to 45.6%. This is a significant amount, considering that the document-type images were already compressed to less then 5% of their original size.

The next set of results are the near-lossless images created from the clustered-dot dithered images. Figures 6.2-6.3 show two of the clustered-dot dithered images at the high quality.

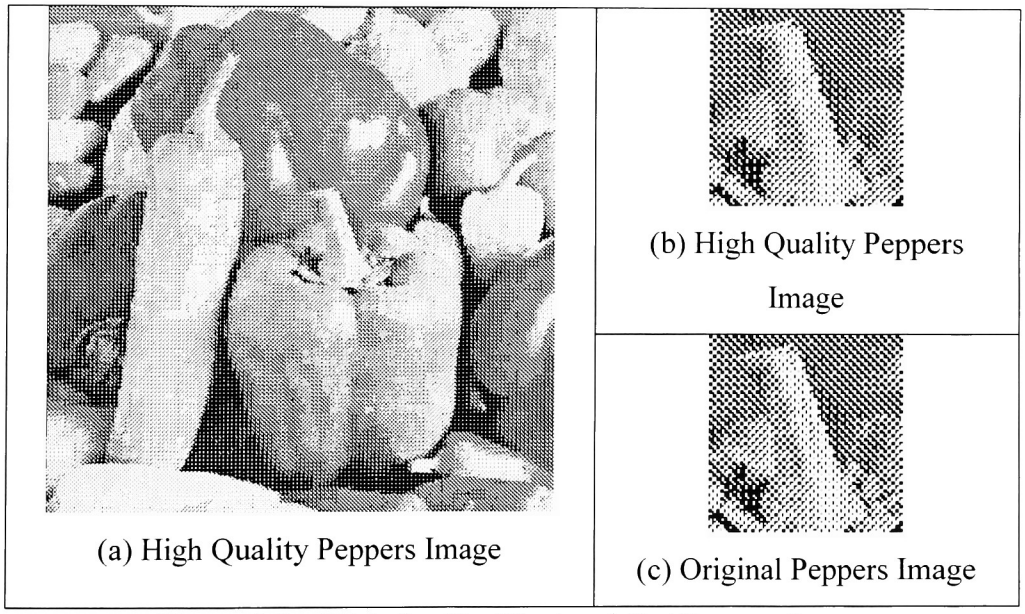


(a) High Quality Lena Image

(b) High Quality Lena Image

(c) Original Lena Image

Figure 6.2 – High Quality Clustered-Dot Lena Image



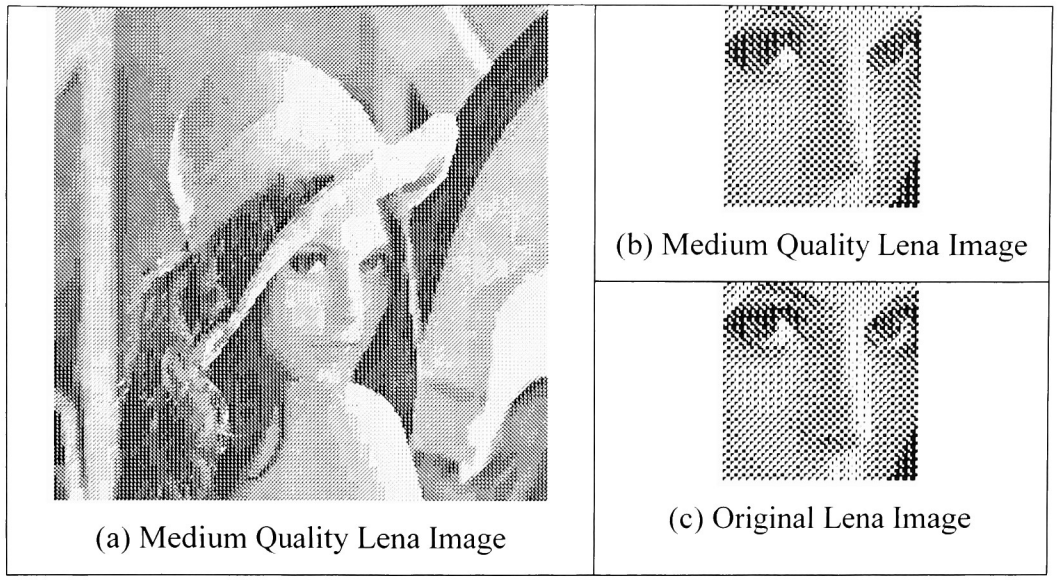
(a) High Quality Peppers Image

(b) High Quality Peppers Image

(c) Original Peppers Image

Figure 6.3 – High Quality Clustered-Dot Peppers Image

As can be seen when comparing the original lena and peppers image to the near-lossless ones, very few differences can be seen. This image quality setting is good when a small amount of loss is acceptable. Some detail has been removed, but the quality is still very good. Figures 6.4-6.5 are similar to Figures 6.2-6.3 except the near-lossless images are at the medium quality.

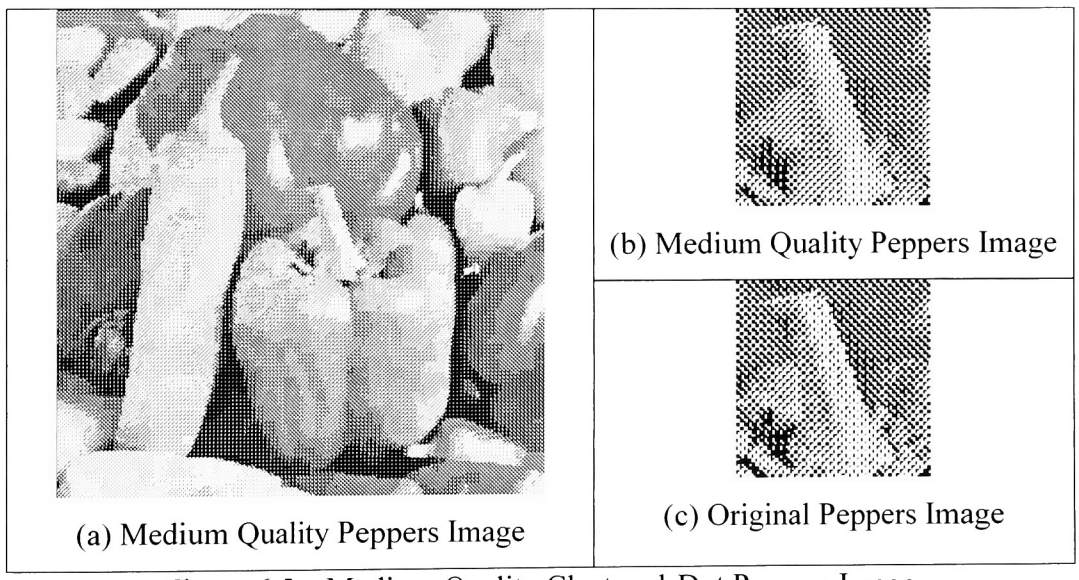


(a) Medium Quality Lena Image

(b) Medium Quality Lena Image

(c) Original Lena Image

Figure 6.4 – Medium Quality Clustered-Dot Lena Image



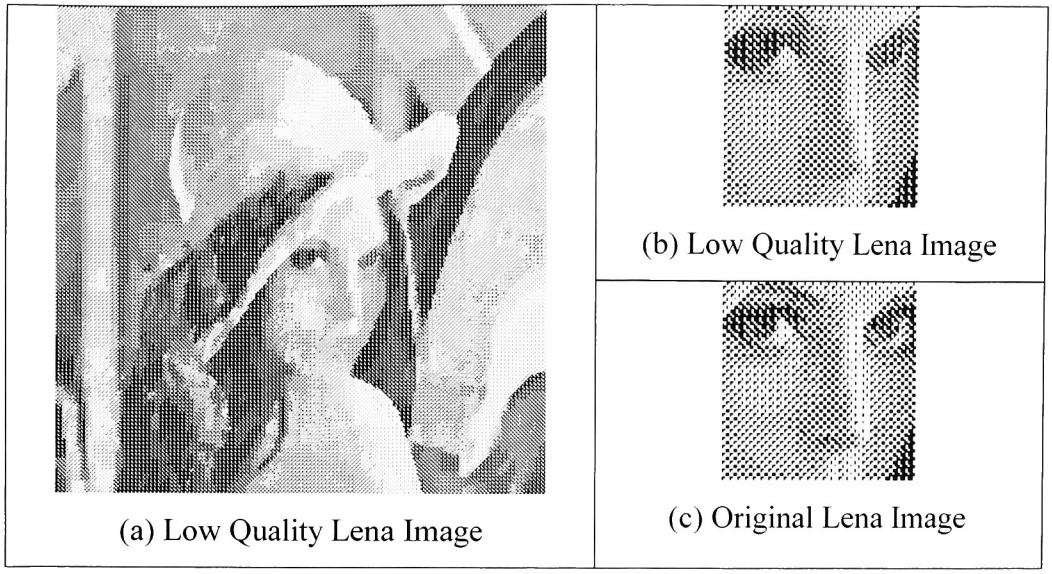
(a) Medium Quality Peppers Image

(b) Medium Quality Peppers Image

(c) Original Peppers Image

Figure 6.5 – Medium Quality Clustered-Dot Peppers Image

At the medium quality setting, the loss introduced into the image is noticeable. The objects in the near-lossless image are still easily recognizable, even though the loss in image quality is evident. Figures 6.6-6.7 below show the lena and peppers images at the low quality setting.

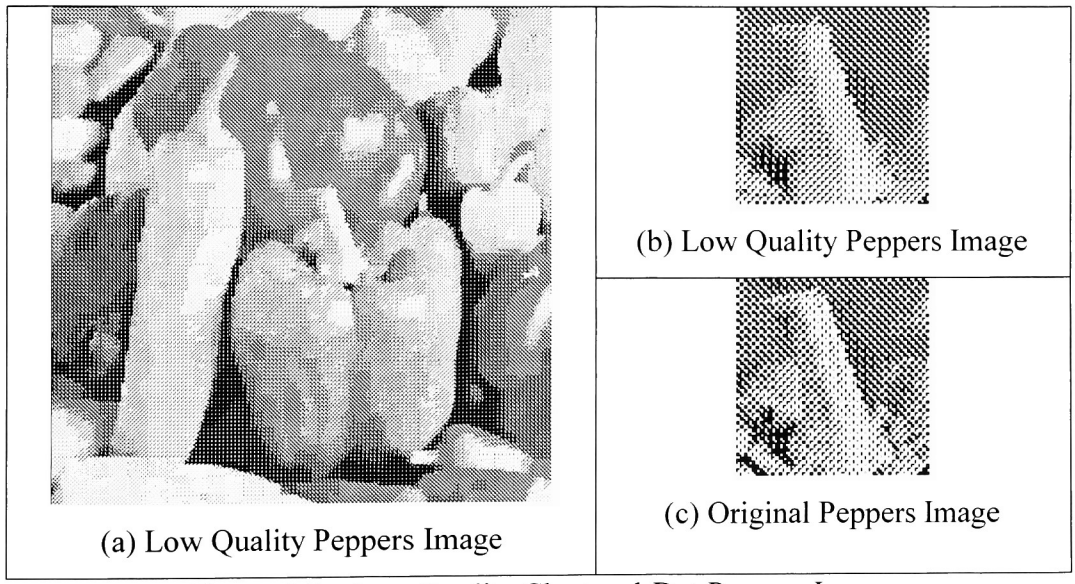


(a) Low Quality Lena Image

(b) Low Quality Lena Image

(c) Original Lena Image

Figure 6.6 – Low Quality Clustered-Dot Lena Image



(a) Low Quality Peppers Image

(b) Low Quality Peppers Image

(c) Original Peppers Image

Figure 6.7 – Low Quality Clustered-Dot Peppers Image

When looking at the images in Figures 6.6-6.7, obvious loss is noticeable. The details of the image have been lost. The objects in the image are still recognizable, but the loss introduced can be distracting. This quality level would only be used in blurry parts of an image that lack detail already or where detail is not important.

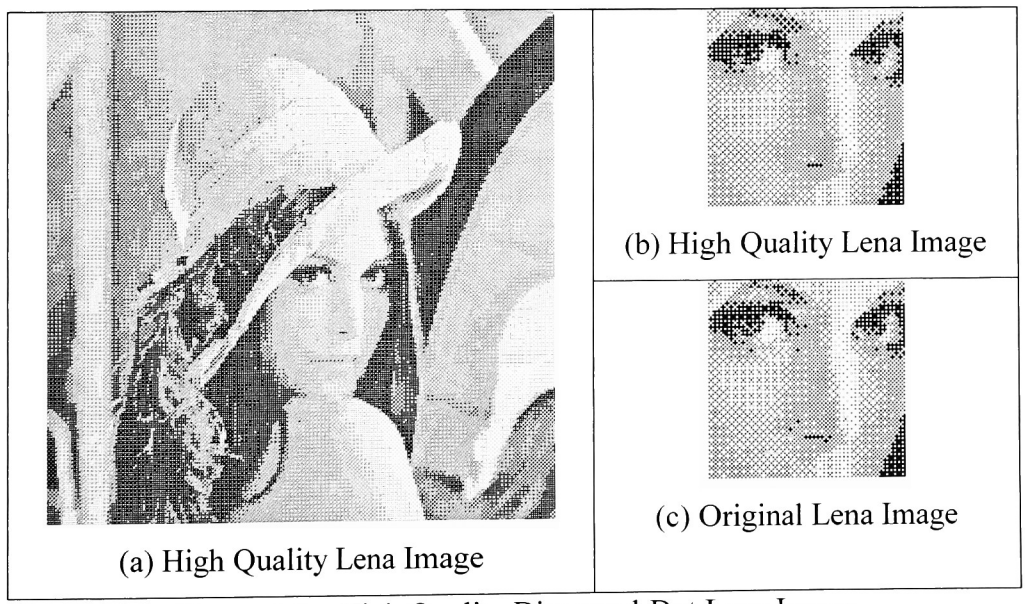
The table below shows the compression ratios achieved using the BACIC algorithm for compressing the images at different quality levels.

	Perfect	High	Medium	Low
Average MPSNR (dB)	23.057	22.811	22.491	22.059
Average Compression Ratio	4.642	6.024	7.570	9.526
Bits Per Pixel	0.215	0.166	0.132	0.105
Compression Ratio Increase		29.76%	63.08%	105.21%

Table 6.8 - Near-Lossless Compression Results for Clustered-Dot Dithered Images

The compression ratios achieved by introducing loss into these dithered images is impressive. Even at the high-quality image where the loss was barely noticeable, the compression ratio increased by almost 30%. At the low quality level, were the loss was obvious in the image, the compression ratio more than doubled compared to the lossless case.

The next set of results are the near-lossless images created from the dispersed-dot dithered images. Figures 6.8-6.9 show two of the dispersed-dot dithered images at the high quality.



(a) High Quality Lena Image

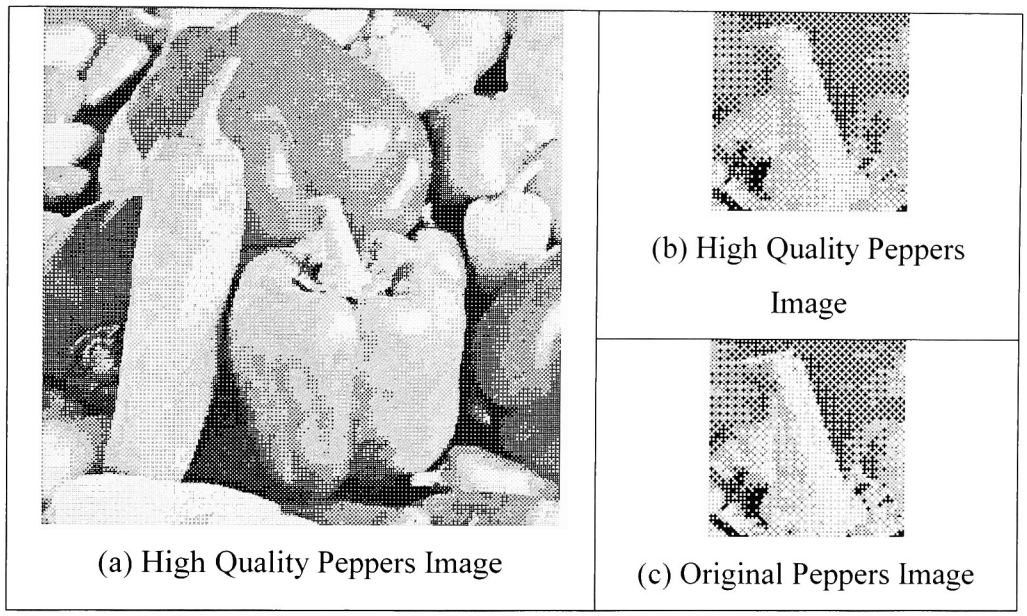
(b) High Quality Lena Image

(c) Original Lena Image

Figure 6.8 – High Quality Dispersed-Dot Lena Image

As can be seen when comparing the original images to the high quality near-lossless ones, differences can now be seen. It appears that more loss is visible in the image when compared

with the clustered-dot image at this quality setting. The reason for this is that the clustered-dot image is compressed more than the dispersed-dot image, therefore, with the dispersed-dot image there is a lot more that can be done to change the compression efficiency. Figures 6.10-6.11 display medium quality images.

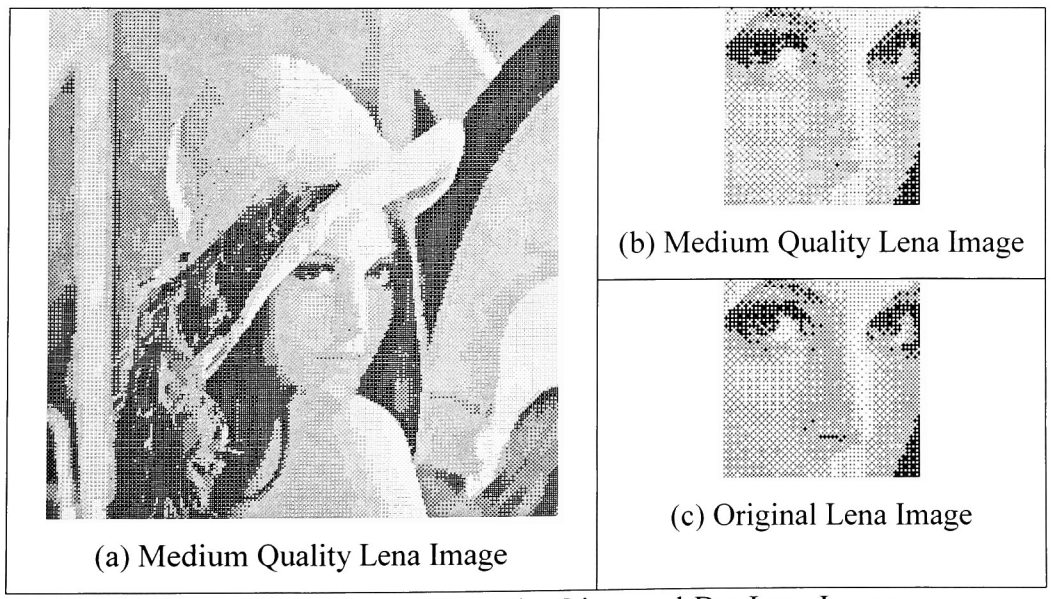


(a) High Quality Peppers Image

(b) High Quality Peppers Image

(c) Original Peppers Image

Figure 6.9 – High Quality Dispersed-Dot Peppers Image



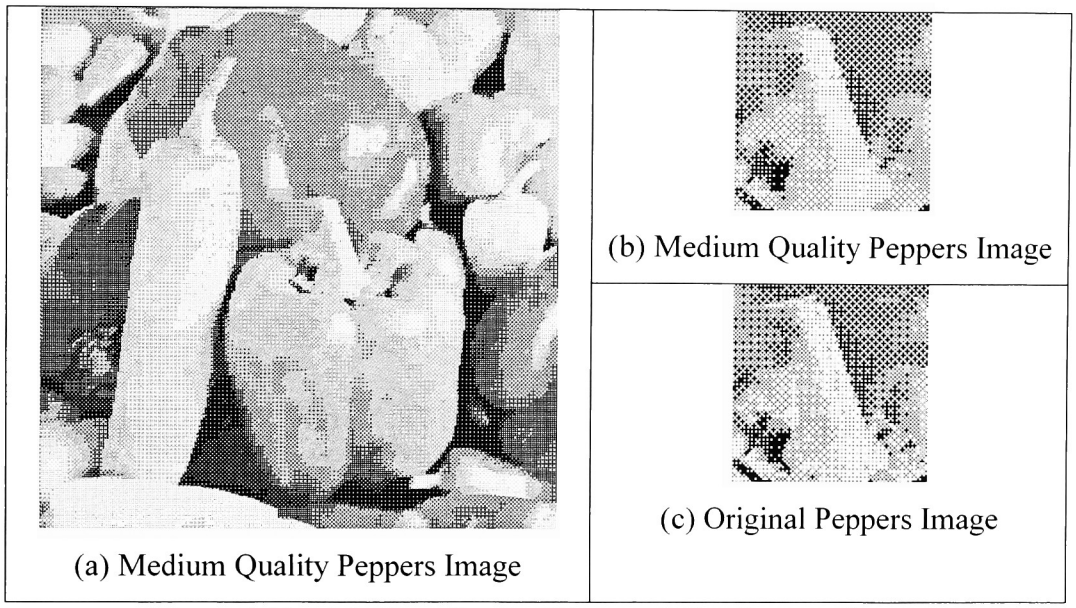
(a) Medium Quality Lena Image

(b) Medium Quality Lena Image

(c) Original Lena Image

Figure 6.10 – Medium Quality Dispersed-Dot Lena Image

As expected, the amount of noticeable loss increases for the medium quality when compared to the high quality. Both of these quality levels lack detail for dispersed-dot images. Figures 6.12-6.13 show images at the low quality setting.

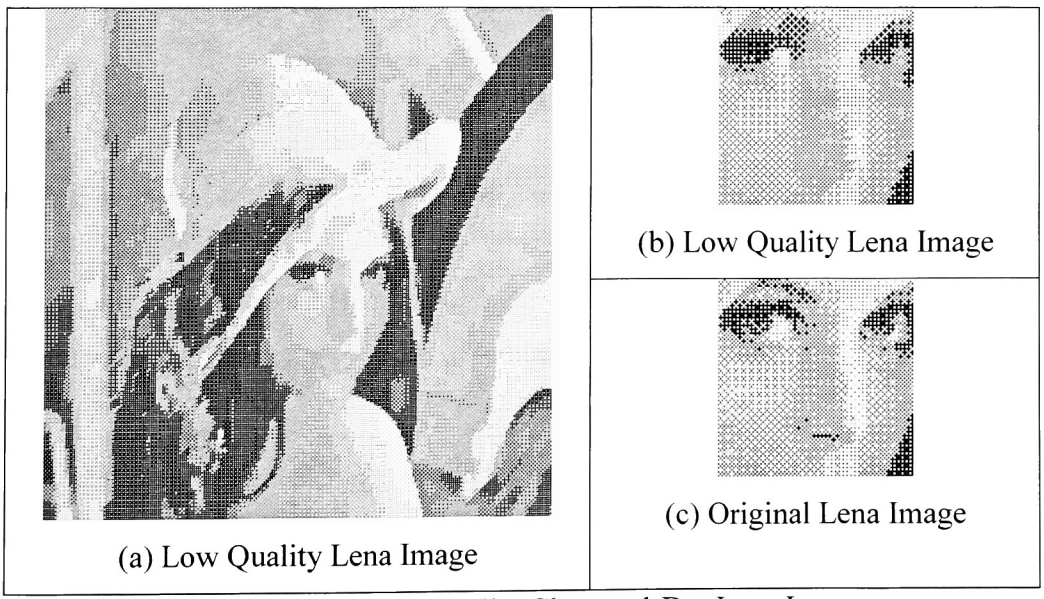


(a) Medium Quality Peppers Image

(b) Medium Quality Peppers Image

(c) Original Peppers Image

Figure 6.11 – Medium Quality Dispersed-Dot Peppers Image



(a) Low Quality Lena Image

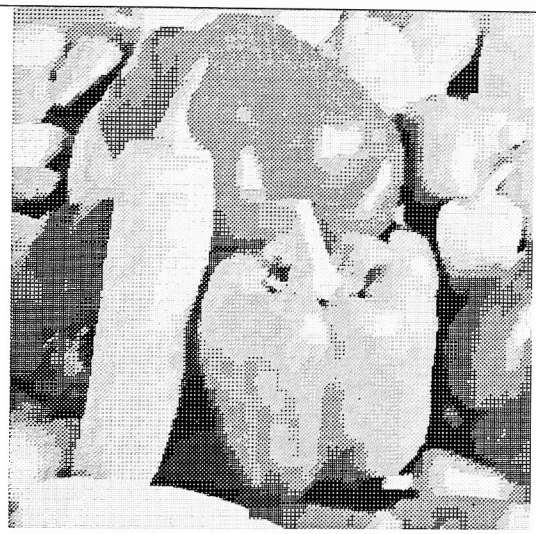
(b) Low Quality Lena Image

(c) Original Lena Image

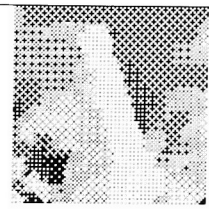
Figure 6.12 – Low Quality Clustered-Dot Lena Image

The visual quality of the medium and the low quality images is about the same. More loss was introduced in the low quality images when compared to the medium quality images, but the

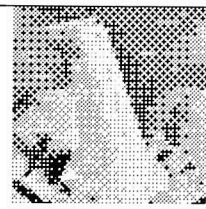
difference between the two is not as obvious as the clustered-dot images. The reason for this is because a lot of loss was introduced into the medium and high quality images for dispersed-dot dithered images, but less for the clustered-dot. As can be seen in Figures 6.2-6.11 the near-lossless clustered-dot images are of much higher quality than the dispersed-dot dithered images. The main reason why the clustered-dot looks better than the dispersed-dot images is because of the nature of the dither mask. For example, if four black pixels are clustered together, and one of them changes, then the difference would be less noticeable then if only one black pixel exists in the area and it is changed, because for the clustered-dot the other three pixels are still black.



(a) Low Quality Peppers Image



(b) Low Quality Peppers Image



(c) Original Peppers Image

Figure 6.13 – Low Quality Dispersed-Dot Peppers Image

The table below shows the compression ratios achieved using the BACIC algorithm for the dispersed-dot dithered images with different quality levels.

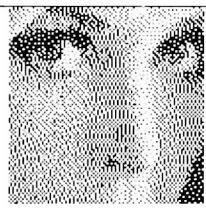
	Perfect	High	Medium	Low
Average MPSNR (dB)	22.562	22.331	22.057	21.694
Average Compression Ratio	4.608	6.267	7.595	9.440
Bits Per Pixel	0.217	0.160	0.132	0.106
Compression Ratio Increase		35.98%	64.80%	104.83%

Table 6.9 - Near-Lossless Compression Results for Dispersed-Dot Dithered Images

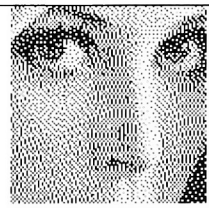
The compression ratios achieved by introducing loss into these dithered images is impressive. Comparing the compression ratios of the dispersed-dot dithered images and the clustered-dot dithered provides an interesting fact. For the lossless results, the compression ratio of the clustered-dot images is higher than that for the dispersed-dot images. However, once the loss is introduced, the dispersed-dot images are now compressed more than the clustered-dot images. The main reason for this is the nature of the dither mask. When using a dispersed-dot dither mask there are a lot of isolated pixels. For example, if there was a white area with only one black pixel in the middle, and this black pixel were to be flipped, then the prediction of the pixels in that white area would be a lot better. Of course this also has an adverse effect on the visual quality, which can be seen in Figures 6.10-6.15. This property of dispersed-dot dithered images makes it unfavorable for introducing loss into these images. The only time loss can be introduced is when detail is not important in an area of the image, so that the loss can be introduced only into that area, which is shown in the next section.

The next set of results present the near-lossless error diffused images. Figures 6.14-6.15 show the lena and peppers error diffused images at the high quality setting.

(a) High Quality Lena Image



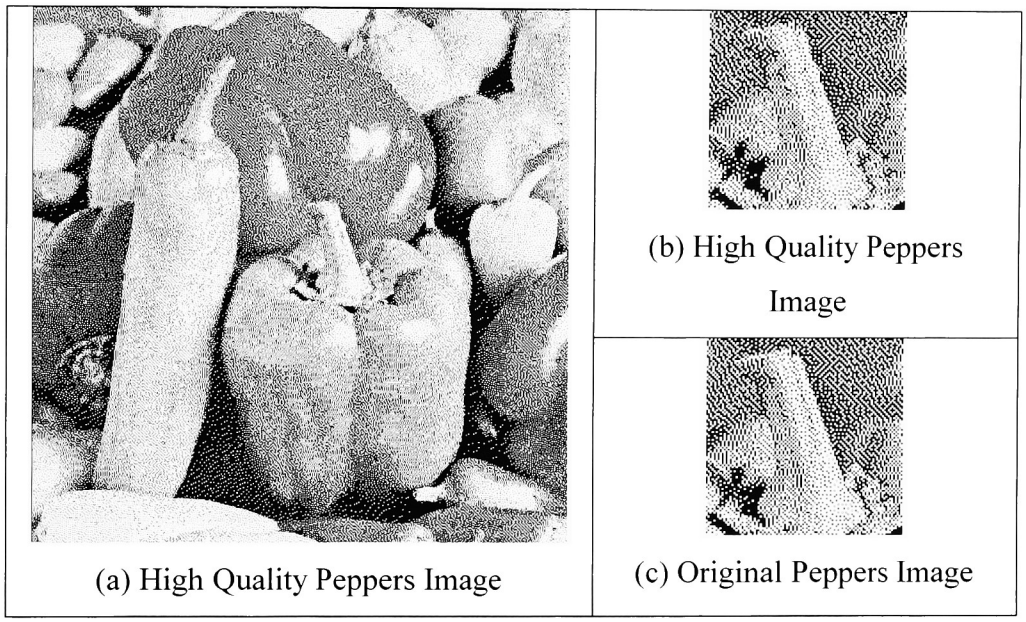
(b) High Quality Lena Image



(c) Original Lena Image

Figure 6.14 – High Quality Error Diffused Lena Image

As can be seen when comparing the original images to the high quality near-lossless ones a few differences can be seen. The loss introduced is noticeable, however, it does not degrade the image quality as much as for the dispersed-dot dithered images. This quality setting would be satisfactory for most applications. Figures 6.16-6.17 show the medium quality images.

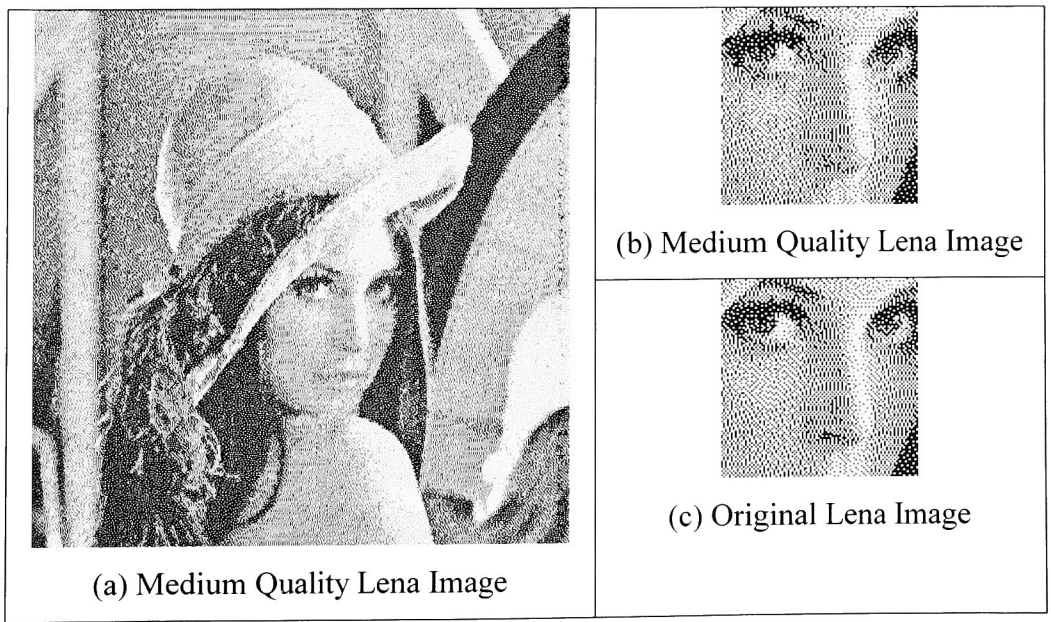


(a) High Quality Peppers Image

(b) High Quality Peppers Image

(c) Original Peppers Image

Figure 6.15 – High Quality Error Diffused Peppers Image



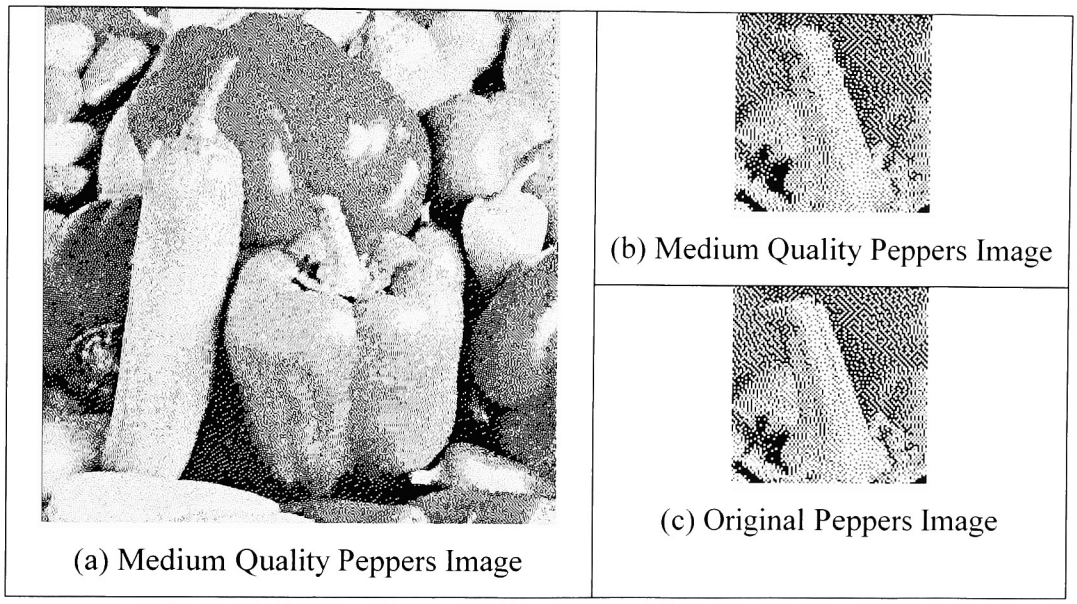
(a) Medium Quality Lena Image

(b) Medium Quality Lena Image

(c) Original Lena Image

Figure 6.16 – Medium Quality Error Diffused Lena Image

As can be seen in Figures 6.16 and 6.17 more loss is noticeable when compared to the high quality images. Some of the detail has now been removed from the image and the loss can be a little distracting. Figures 6.18-6.19 show the low quality setting.

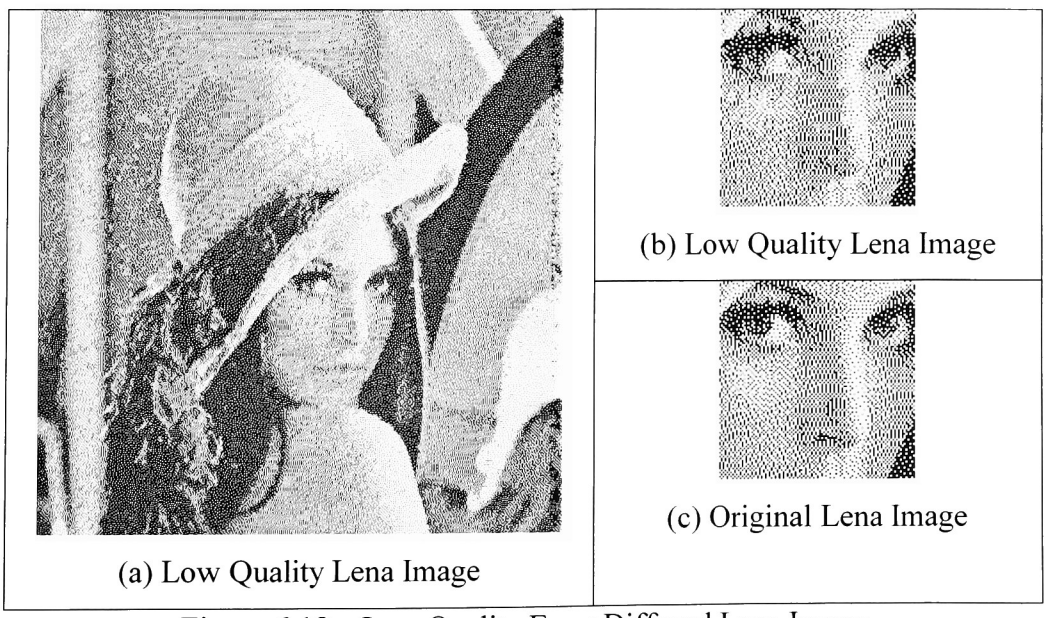


(a) Medium Quality Peppers Image

(b) Medium Quality Peppers Image

(c) Original Peppers Image

Figure 6.17 – Medium Quality Error Diffused Peppers Image



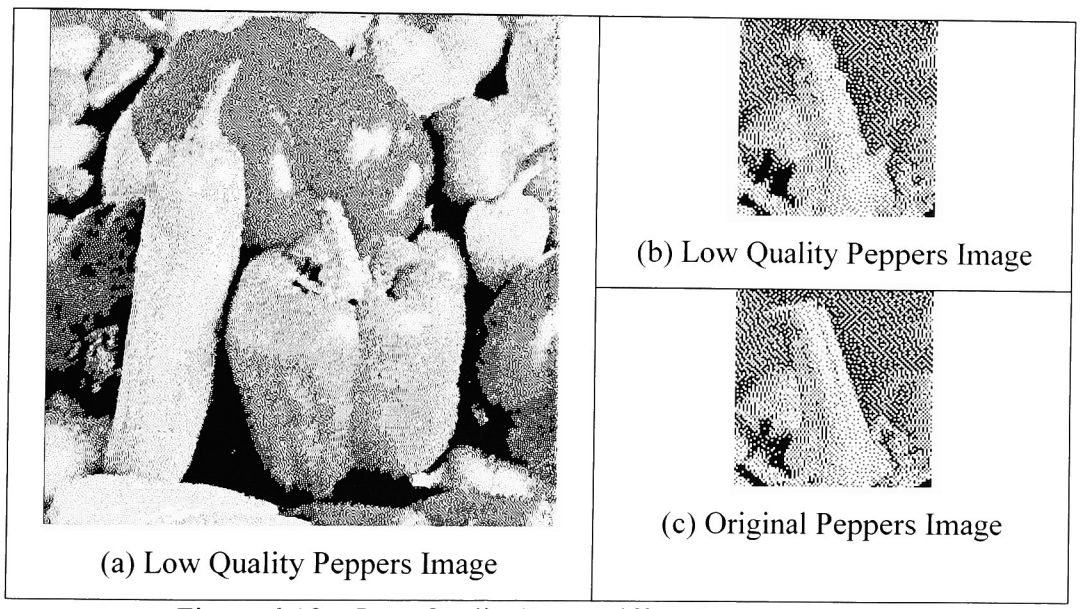
(a) Low Quality Lena Image

(b) Low Quality Lena Image

(c) Original Lena Image

Figure 6.18 – Low Quality Error Diffused Lena Image

As can be seen in Figures 6.18-6.19, the image quality degrades significantly for the low quality images. A lot of the detail has been removed from the image. An interesting fact to observe is that the edges of the image are now very fuzzy. The reason for this is that in the original image the edges of the image were dark, but when compressing with loss introduction, the images are padded with white pixels, so the loss introduction algorithm is trying to make the edges have a lighter color to match the padding. The low quality setting and probably the medium quality setting would most likely only be used when doing region of interest consideration on a part of the image where detail is not needed, when considering error diffused images. The reason for this is the removal of detail from the image.



(a) Low Quality Peppers Image

(b) Low Quality Peppers Image

(c) Original Peppers Image

Figure 6.19 – Low Quality Error Diffused Peppers Image

The table below shows the compression ratios achieved using the BACIC algorithm for the images with different quality levels.

	Perfect	High	Medium	Low
Average MPSNR (dB)	26.205	25.194	24.562	23.356
Average Compression Ratio	1.786	1.927	2.038	2.214
Bits Per Pixel	0.560	0.519	0.491	0.452
Compression Ratio Increase		7.92%	14.15%	24.00%

Table 6.10 - Near-Lossless Compression Results for Error Diffused Images

As can be seen when comparing the compression ratios for error diffused images, the increase in compression efficiency is not that high when compared to dithered images. Even at the low quality setting the amount compression ratio increase is less than 25%. The reason why compression efficiency does not increase as much is because of the nature of the error diffusion process. When an image is halftoned using a dither masking method patterns are generated throughout the image that help the compression process by improving the prediction process. For the error diffusion halftoned images, these patterns do not exist. Therefore, the pixels in the image are more random, making it harder to compress.

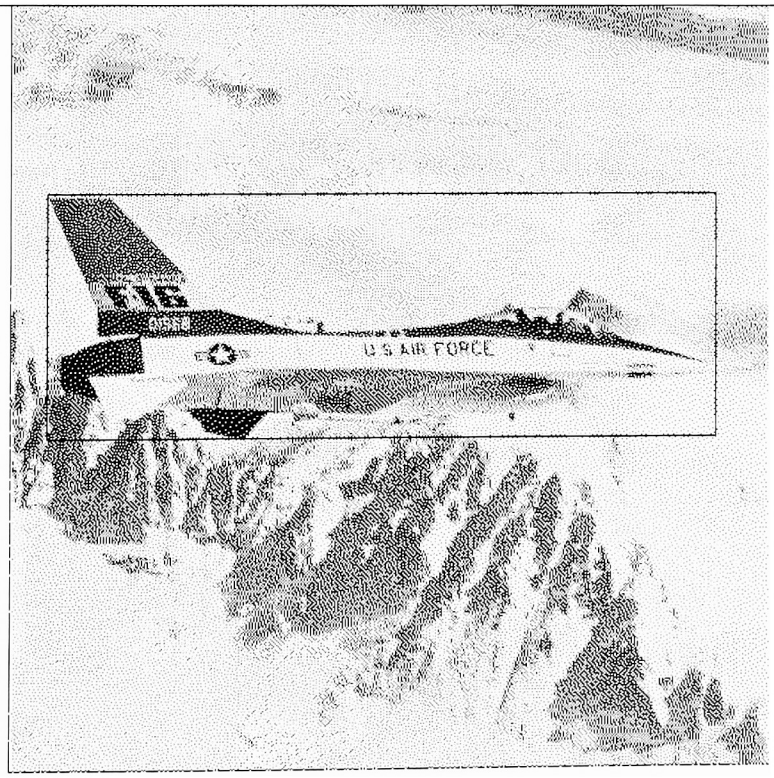
6.3 Near-Lossless With ROI Bitonal Image Simulation Results

This section presents compression results when considering a region of interest. Four images are used to demonstrate the advantages of the region of interest (ROI) consideration. Two of the images are dithered and two of the images are error diffused. The first image shown in Figure 6.20 is the airplane image, and it was created using the error diffusion method.

Figure 6.20(a) shows the near-lossless error diffused airplane image when considering an ROI. The ROI was outlined in the lossless image in Figure 6.20(b). The number in parenthesis for these images is the compression ratio when compressing using BACIC. The ROI was set at the high quality setting, and the area outside of the ROI was set at the low quality setting. As can be seen, there is more detail preserved in the ROI than outside the ROI.

As expected, the compression ratio of the near-lossless image with ROI is between the compression ratio of the lossless image (2.056) and the compression ratio of the low-quality image (2.329). When considering that the detail of the ROI of the image was sustained well, the compression ratio improvement of about 10% is reasonable. Figure 6.21 shows the same scenario except for the lena error diffused image.

(a) High/Low Quality Airplane Image (Compression Ratio = 2.258)



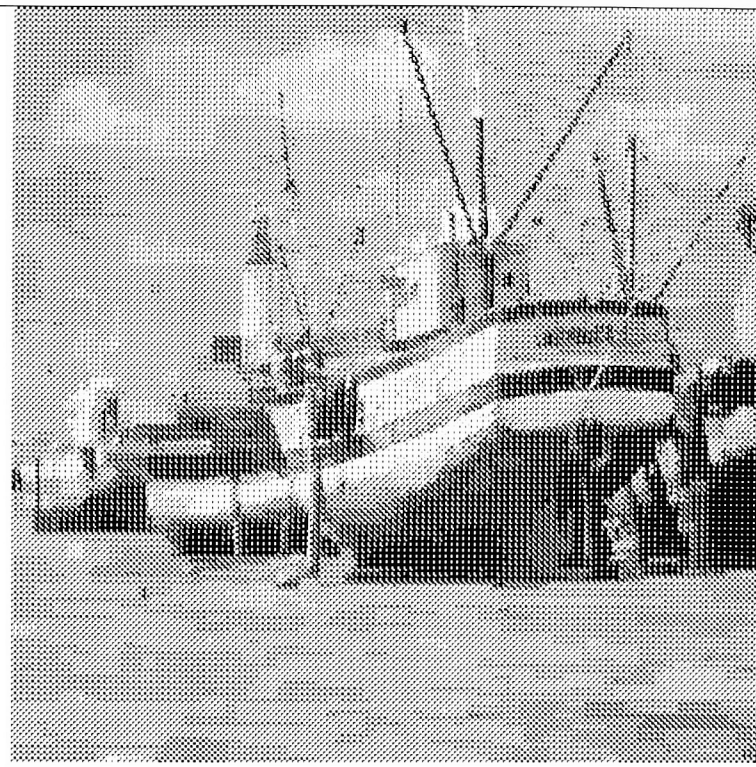
(b) Original Airplane Image With ROI Outlined (Compression Ratio = 2.056)

Figure 6.20 – Near-lossless(ROI) Airplane Image

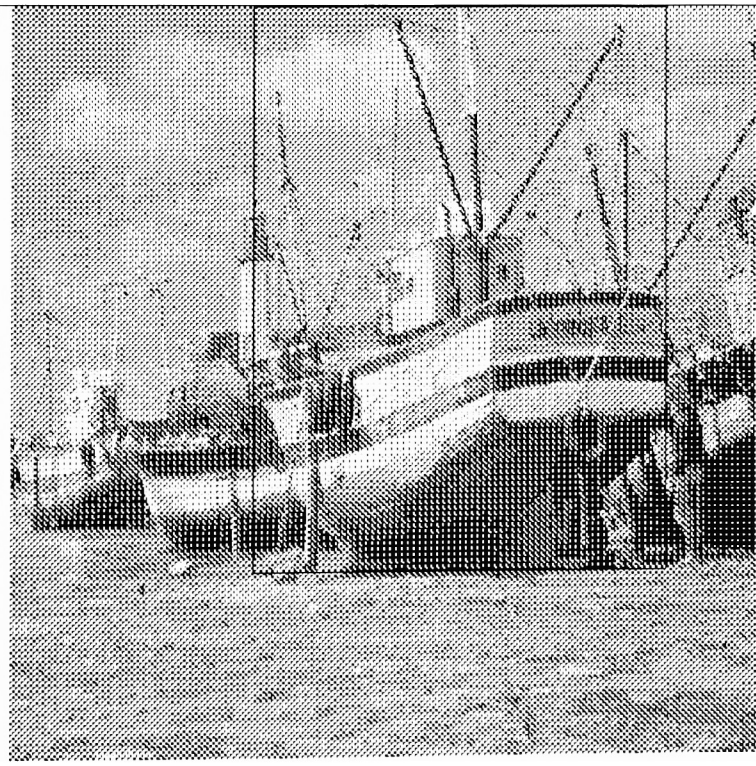
(a) Perfect/Low Quality Lena Image (Compression Ratio = 2.236)

(b) Original Lena Image With ROI Outlined (Compression Ratio = 1.990)

Figure 6.21 Near-lossless(ROI) Lena Image



(a) High/Low Quality Boat Image (Compression Ratio = 7.959)



(b) Original Boat Image With ROI Outlined (Compression Ratio = 4.811)

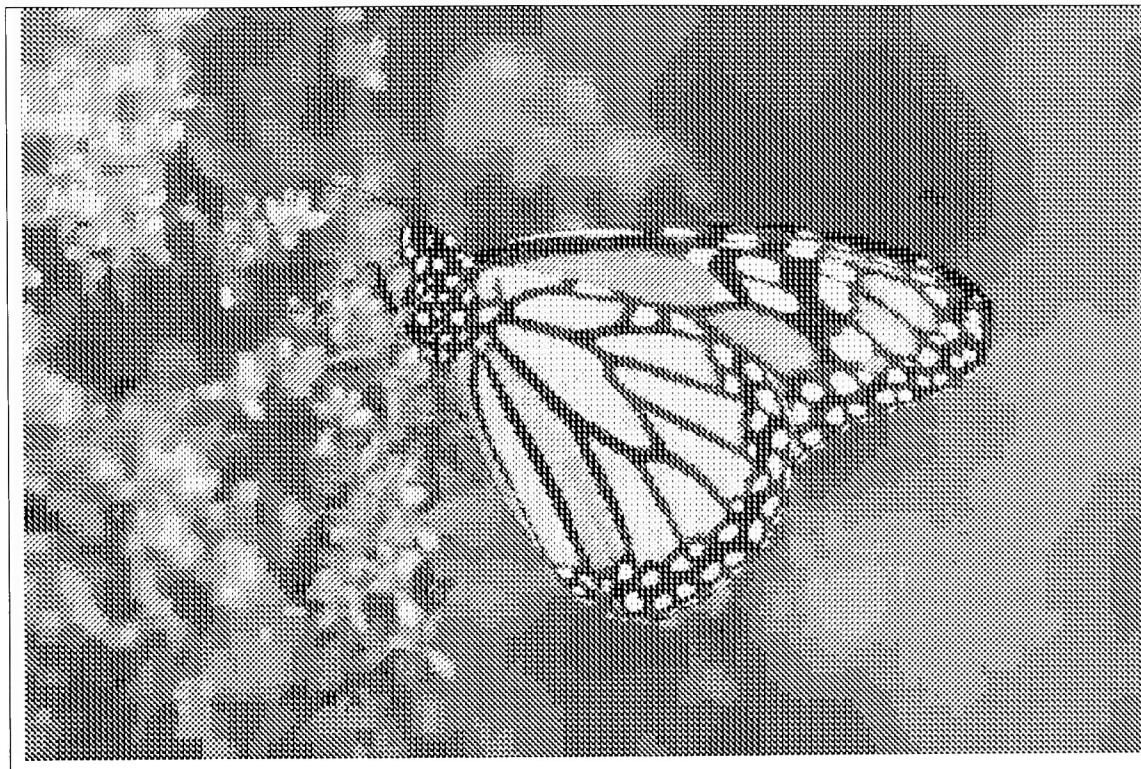
Figure 6.22 - Near-lossless(ROI) Boat Image

Figure 6.21 is the same as Figure 6.20 except for the lena image the ROI was set to the perfect quality. As can be seen the quality of the image is still impressive, and the compression ratio increased by a large amount, over 12%. The reason for making the ROI in Figure 6.21 perfect quality and the ROI for Figure 6.20 high quality is because generally a face has more important detail than an airplane, so the detail of the ROI in Figure 6.21 is fully preserved. As can be seen the quality of the image was sustained well even though the compression ratio increased by a significant amount. The next figure shows a near-lossless representation of the boat image.

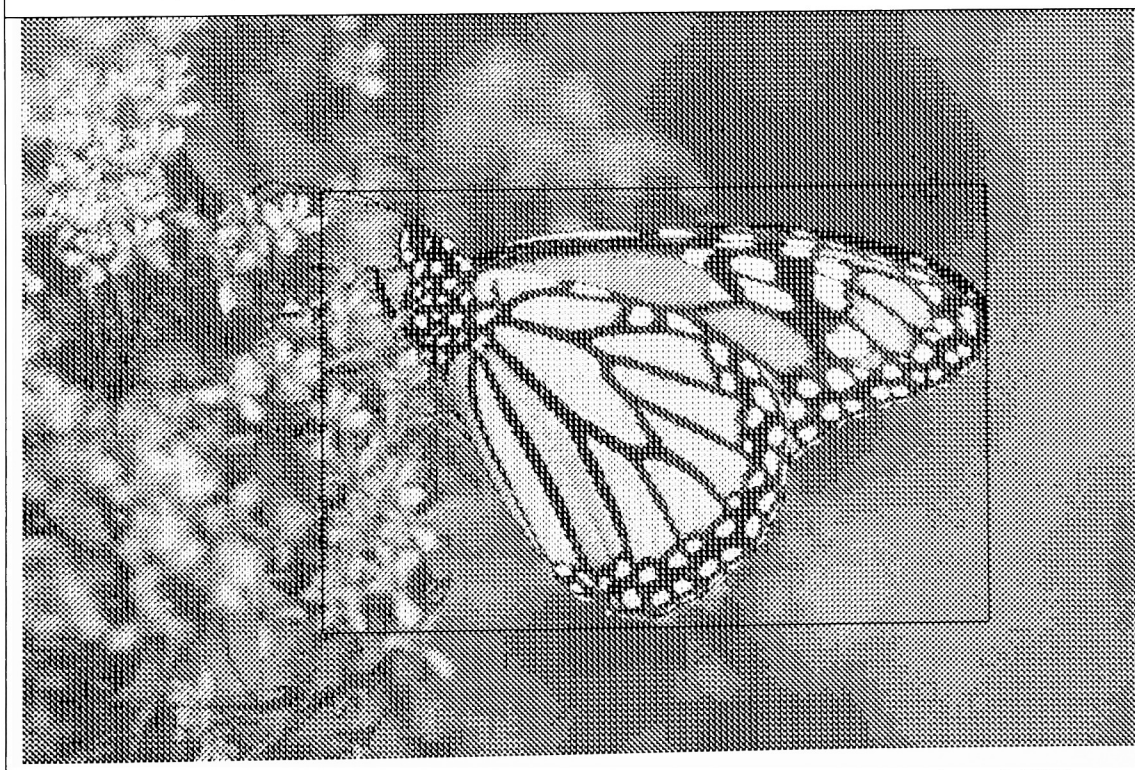
The images in Figure 6.22 are the boat image halftoned using the clustered-dot dither method. The ROI chosen for this image is the boat in the foreground as can be seen in Figure 6.22(b). For this image the ROI was compressed using the high quality setting and the rest of the image was compressed using the low quality setting. As can be seen in this case the loss introduced increased the compression ratio significantly, over 65%. The reason it works so well is that images halftoned using clustered-dot compress very well, especially when loss is added. Figure 6.23 shows the near-lossless monarch image.

For the monarch image shown in Figure 6.25 the ROI chosen was the butterfly, shown in Figure 6.23(b). For this image the ROI was set at the high quality level and the rest of the image was set at the low quality level. As can be seen the detail of the image was sustained very well, especially when looking at the most detailed parts, like the wings of the butterfly.

The monarch image is a very good example of how well this near-lossless compression with ROI works. Even though only small differences can be seen between the lossless and near-lossless image, the increase in compression efficiency is very good, over 40%.



(a) High/Low Quality Monarch Image (Compression Ratio = 8.003)



(b) Original Monarch Image With ROI Outlined (Compression Ratio = 5.700)

Figure 6.23 Near-lossless(ROI) Monarch Image

6.4 Grayscale Image Compression Results

One advantage of bitonal image compression algorithms is that they can easily be used to compress grayscale images [37]. In order to accomplish this each of the bit planes of the image is compressed individually. So, for an 8-bit grayscale image, the encoder would be run eight times. As shown below, the BACIC algorithm can compress a bitonal image efficiently.

Several images were used to compare the ability of the BACIC encoder to other image compression algorithms. The list of these images can be found in Table 6.11.

Image	Size
airplane	512x512
baboon	512x512
barbera	720x576
boats	512x512
bridge	256x256
café	816x1024
camera	256x256
couple	256x256
gccitt1	1728x2376
gccitt2	1728x2376
gccitt3	1728x2376
gccitt4	1728x2376
gccitt5	1728x2376
gccitt6	1728x2376
gccitt7	1728x2376
gccitt8	1728x2376
goldhill	512x512
lena	512x512
monarch	768x512
peppers	512x512

Table 6.11 – Grayscale Images

The pictorial grayscale images are the original images that were halftoned for the bitonal compression part of the results. The gccitt documents were used in [3] and were generated by printing and scanning the CCITT images.

Several algorithms, both bitonal and grayscale, were used in [3] to test out the compression efficiency of encoding grayscale images. The grayscale image compression algorithms used are: JPEG-Lossless (JPEG-LS) [14] [24] and context-based adaptive lossless image coding (CALIC) [15]. The bitonal compression algorithms used are: Ziv-Lempel-Welch (LZW) [13], Group 3 (1d and 2d) [18], Group 4 [19], JBIG [20], JBIG2 [6] and BACIC [1]. The results for the grayscale algorithms were taken from [3]. Except for the LZW encoder, all of the bitonal image encoders are described in Chapter 3.

Two different version of the BACIC encoder were used to generate the results. The first version is the one described in Section 3.5. The second version of BACIC is identical to the original except that it uses a template that spans multiple bit planes. This not only takes advantage of the patterns in a single plane, but also takes advantage of the influence that a pixel in a different bit plane has on the current pixel. The template used can be seen in the figure below.

Figure 6.24 – BACIC Multiplane Template

As can be seen in Figure 6.24 for context pixels 0 through 6 the template is identical to the original three-line template. The remaining five pixels are taken from the plane below the current plane, except when the current plane is already the least-significant bitplane. In order to study the interplanar dependencies further, the test images were Gray coded and compressed. When compressing one bit plane at a time, it can be beneficial to Gray code the image, because interplanar dependencies can be exploited.

Tables 6.12(a-b) below show the results for the bitonal image compression algorithms. The unit for Table 6.12(b) is bits-per-pixel. The RAW Data column is showing that there is 8 bits per pixel in the uncompressed image. The BACICm column is the one using the multiplanar template. As can be seen in this table BACIC does much better than the other bitonal

compression algorithms. Also, the Gray encoded images compress more efficiently than the other images.

	LZW	Group 3	Group 4	JBIG	JBIG2	BACIC	BACICm
GCCITT	1.52	1.15	1.12	2.01	2.05	2.10	2.30
PICS	0.98	0.73	0.73	1.29	1.29	1.36	1.52
Gray Encoded GCCITT	1.84	1.84	1.88	2.16	2.08	2.11	2.34
Gray Encoded PICS	1.10	0.86	0.87	1.49	1.34	1.38	1.58

Table 6.12(a) – Compression Ratios for Bitonal Image Compression on Grayscale Images

	RAW Data	LZW	Group 3	Group 4	JBIG	JBIG2	BACIC	BACICm
GCCITT	8	5.28	6.96	7.16	3.99	3.91	3.80	3.47
PICS	8	8.18	10.91	10.92	6.22	6.19	5.88	5.27
Gray Encoded GCCITT	8	4.35	4.35	4.26	3.7	3.84	3.80	3.42
Gray Encoded PICS	8	7.25	9.31	9.17	5.36	5.97	5.81	5.05

Table 6.12(b) – Bitonal Image Compression on Grayscale Images (bits per pixel)

One advantage to using a bitonal image compression algorithm to compress grayscale images is that when negative compression occurs on a bit plane, the original image data can be stored instead of the compressed data, making the compression more efficient. Negative compression typically occurs on the least-significant bit planes, because they tend to be more random. Tables 6.13(a-b) show the results when negative compression is not allowed.

	LZW	Group 3	Group 4	JBIG	JBIG2	BACIC	BACICm
GCCITT	1.82	1.75	1.81	2.08	2.18	2.23	2.30
PICS	1.19	1.18	1.22	1.36	1.38	1.54	1.54
Gray Encoded GCCITT	1.84	1.84	1.88	2.16	2.20	2.23	2.34
Gray Encoded PICS	1.30	1.29	1.35	1.56	1.42	1.55	1.60

Table 6.13(a) – Compression Ratios for Bitonal Image Compression on Grayscale Images Not Allowing Negative Compression

	RAW Data	LZW	Group 3	Group 4	JBIG	JBIG2	BACIC	BACICm
GCCITT	8	4.4	4.57	4.43	3.85	3.67	3.58	3.47
PICS	8	6.7	6.77	6.58	5.9	5.78	5.21	5.21
Gray Encoded GCCITT	8	4.35	4.35	4.26	3.7	3.63	3.58	3.41
Gray Encoded PICS	8	6.14	6.21	5.92	5.14	5.62	5.17	5.01

Table 6.13 – Bitonal Image Compression on Grayscale Images Not Allowing Negative Compression

As can be seen in this table, except for the multiplanar BACIC, the other algorithms improve significantly when not allowing for negative compression. The BACIC multiplanar algorithm still performs better than the other bitonal image compression algorithms. Tables 6.14(a-b) show the results for grayscale image compression algorithms.

	JPEG-LS	CALIC	BACICm
GCCITT	2.37	2.44	2.34
PICS	1.75	1.79	1.60

Table 6.14(a) – Compression Ratios for Grayscale Image Compression Algorithms

	RAW Data	JPEG-LS	CALIC	BACICm
GCCITT	8	3.38	3.28	3.41
PICS	8	4.58	4.47	5.01

Table 6.14(b) – Grayscale Image Compression Algorithms (bits per pixel)

As can be seen in Tables 6.13-6.14 the best algorithm for the compression of grayscale images is CALIC. CALIC compresses about 4% better than multiplanar BACIC for the GCCITT document-type images and about 12% better for the pictoral images. This is expected, because the CALIC encoder was designed to work on grayscale images while the BACIC algorithm was designed to work on bitonal images.

6.5 Hardware vs. Simulation Performance

This section compares the hardware implementation of the BACIC algorithm to the software (C) implementation. It is demonstrated that the hardware implementation can correctly compress

any image within the codec's size limit and decompress that same image. It also compares the execution times of the two implementations.

The hardware implementation was verified using the Modelsim VHDL simulator [22]. Several images, both error diffused and ordered dithered, were compressed. The resulting data was compared to the data output from the software implementation. Not only were the two compressed files for each image the same size, but also their contents were identical. In addition, for each of the compressed images, the hardware implementation was run for the decoding process, and all of the images were decompressed without loss.

The functionality of the VHDL code was proven, however getting it to fit on the target FPGA was still a challenge. After a great deal of optimization the BACIC encoder and decoder fit on the Xilinx XCV-300, using up around 90-95% of the chip. The maximum clock frequency that can be used with the implementation was found to be 41.259 MHz, as provided by the synthesis tools, which is reasonably fast considering the amount of computation that needs to be done.

The next study was the effect on execution time. Several images were compressed, both error diffused and dithered images, using both the hardware implementation and the C implementation, and the resulting execution times were recorded. Table 6.15 shows the average execution time for both the hardware implementation and the C implementation of the BACIC encoder and decoder. The C implementation was run on a personal computer (PC) running the Microsoft Windows 2000 operating system and utilized an AMD 450 MHz CPU. The hardware simulation was run using a clock frequency of about 40 MHz. Table 6.13 shows the execution time on the two platforms.

Image Type	Action	C Execution Time(ms)	Hardware Execution Time(ms)	Hardware Speedup
Error Diffused	Compress	462.4	203.06	2.28
Error Diffused	Decompress	471.9	369.30	1.28
Dithered	Compress	438.6	206.06	2.13
Dithered	Decompress	442.6	371.92	1.19

Table 6.15 – BACIC Encoder and Decoder Execution Times

As can be seen in Table 6.15, the hardware implementation runs faster than the C implementation for both the encoding and decoding process. However, considering the speed of processors today reaching above 2 GHz, if the performance was measured for a much faster CPU than the hardware system would have been slower for both cases.

This section proves that it was beneficial to implement the compression algorithm in hardware. The hardware implementation with limited precision can compress the image as efficiently as the C implementation with virtually unlimited precision. This chip that was synthesized using the VHDL code would be very useful to include in a scanning, printing or facsimile product to handle the compression or decompression. This hardware implementation provides a flexible and low power solution for image compression.

Chapter 7. Future Work and Discussion

In this thesis, a method was developed for introducing loss into bitonal images. This algorithm is especially useful, because it can introduce low amounts or no loss into regions of the image that are of special interest. This field of region of interest consideration is very useful in the image compression field, because it allows for an increase in compression efficiency over lossless image compression, while sustaining the quality of the important parts of the image. This thesis also provides a hardware implementation of the BACIC codec, which could be useful in situations where a stand-alone product requires the ability to perform compression or decompression of bilevel images.

Additional work that could be done to expand upon the work completed in this thesis would be to take the hardware implementation of the BACIC algorithm and parallelize it for grayscale images. Using the idea presented in Section 6.4, this scenario would compress an 8-bit grayscale image using eight of the BACIC encoders created in this thesis, each of them compressing one bit plane, the same being true for decoding. This would be beneficial, because an entire 8-bit image could be compressed to less than half its size, in most cases, in the time it takes to compress a bitonal image.

Also work could be done to look into a hardware implementation using 32-bit memory access rather than 16-bit memory access. The memory organization on the Xess prototyping board allows for two 16-bit words to be read in parallel. This would speed up the memory access when 32-bit words are needed, making the implementation faster.

As was demonstrated in this thesis, the BACIC algorithm is a very efficient bitonal image compression algorithm that provides compression efficiency better than that of the current standard, JBIG2, for most pictorial images. The results presented in this thesis do match the compression results for BACIC provided in [1]. Also, the loss introduction algorithm proved to be extremely useful when higher compression efficiency is needed without significant loss in image quality. As was shown in Chapter 6, the quality of the region of interest could be sustained while removing detail from the less important regions of the image and improving the

compression efficiency. Therefore, the work completed in this thesis was a success and would be beneficial to use in several products in the scanning, printing and facsimile industry.

Glossary

Arithmetic Coding: A coding technique that maps a sequence of source symbols to a floating point interval.
BAC: Block Arithmetic Coder
BACIC: Block Arithmetic Coder for Image Compression. A bitonal image compression algorithm.
Bitonal Image: An image consisting of only white and black pixels.
Bitplane: A binary image representing the same bit of each pixel value.
CALIC: Context-based Adaptive Lossless Image Codec
CCITT: Consultative Committee on Telephone and Telegraph
Codec: A combination of an encoder and decoder.
Codeword: A sequence of information units (bits) that map to a value in the source data set when decoded.
Compression Ratio: Size of the input image divided by the size of the output image.
Decoder: An implementation that reconstructs a representation of the original data set from the encoder compressed output data set.
ΔL: The change in encoding length if a pixel were to be flipped.
Encoder: An implementation that translates an input data set into a recoverable compressed output data set.
Entropy: The lower bound on number of information units per symbol in information theory.
Error Diffusion: A method of halftoning where the error created by thresholding a pixel is diffused to neighboring pixels.
FPGA: Field Programmable Gate Array.
g_{max}: The threshold for the grayscale error.
Gray Code: A binary code where each number in succession differs by only 1 bit.
Grayscale Image: An image, which has pixels that are each a discrete shade of gray.
Halftoning: The process of converting a grayscale image into a bitonal image.
Huffman Coding: A coding technique that is based upon the probability of occurrence of particular symbols in a source data set. Fewer bits are used to construct the codes for the more probable symbols.
JBIG: Joint Bilevel Image Experts Group
JPEG: Joint Photographic Experts Group
JPEG-LS: A standard produced by the JPEG committee for lossless and near-lossless image compression.
K: The size of the BAC codebook.
λ: The marginal length.
Lossless Compression: A compression technique where the exact original may be reconstructed.
Lossy Compression: A compression technique where it is possible to reconstruct the original data approximately, not exactly.
MPSNR: Modified Peak Signal to Noise Ratio. A benchmark for comparing lossy bitonal images.
n_0: The number of pixels that equal 0.
n_1: The number of pixels that equal 1.
n_{max}: The maximum number of times each block is processed.

Ordered Dither: A method of halftoning where a threshold mask is used.
p_0: The probability that a pixel is a 0.
p_1: The probability that a pixel is a 1.
Pattern Matching and Substitution: A method for compression of textual documents where only a single copy of each character is encoded.
PSNR: Peak Signal to Noise Ratio. A benchmark for comparing lossy image compression algorithms.
Progressive Coding: Coding in which lower resolution representations of an image are encoded with the original image.
Q-Coder: An arithmetic coder implemented by IBM in hardware that uses fixed-precision arithmetic.
Region of Interest: The object or objects that are the main focus of an image.
r_i: A count of the number of previous pixels equaling a 1, weighted by a forgetting factor.
RMSE: Root Mean Squared Error. This is a benchmark for comparing lossy image compression algorithms, and is inversely proportional to the PSNR.
Sequential Coding: Coding in which only the original image is encoded.
s: The scaling factor.
s_i: A count of the number of previous pixels weighted by a forgetting factor.
Soft Pattern Matching: Similar to pattern matching and substitution except a refinement image is encoded as well.
SRAM: Static Random Access Memory
Synthesis: The process by which a design written in a hardware description language is translated into logic gates.
Template: A group of surrounding pixels used to generate a context.
VHDL: VHSIC Hardware Description Language.
VHSIC: Very High Speed Integrated Circuits.

References

[1] M. Reavy and C. Boncelet, "An Algorithm for Compression of Bilevel Images", *IEEE Transactions Image Processing*, pp.669-676, May 2001.

[2] B. Martins and S. Forchhammer, "Lossless, Near-Lossless and Refinement Coding of Bilevel Images", *IEEE Transactions Image Processing*, pp.601-613, May 1999.

[3] A. Savakis, "Evaluation of Lossless Compression Methods for Gray Scale Document Images", *Journal of Electronic Imaging*, pp.75-86, January 2002.

[4] Xess, "XSV-300 Virtex Prototyping Board with 2.5V, 300,000-gate FPGA", http://www.xess.com/prod014_3.php3.

[5] R.C. Gonzalez and R.E. Woods, *Digital Image Processing*, Addison-Wesley Publishing Company, New York City, NY (1992).

[6] JBIG, JBIG2 Final Committee Draft Version 1, ISO/IEC International Standard 14492, 1999.

[7] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE*, Vol. 40, No. 9, pp. 1098-1101, 1952.

[8] A. Barbir, "A New Fast Approximate Arithmetic Coder", *IEEE Transactions Image Processing*, pp. 482-486, April 1996.

[9] C. Lee and H. Park, "Multisymbol Data Compression Using a Binary Arithmetic Coder", *Electronics Letters*, Vol. 38, Issue 3, pp. 125-125, January 2002.

[10] L. Bottou, P.G Howard and Y. Bengio, "The Z-Coder Adaptive Binary Coder", *Data Compression Conference*, pp. 13-22, April 1998.

[11] W. Pennebaker and J. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

[12] M. Slattery and J. Mitchell, "The Qx-Coder", *IBM J. Res. Develop*, Vol. 42, pp. 767-784, 1998.

[13] T. Bell, J. Cleary and I. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, NJ (1990).

[14] M. Weinberger, G. Seroussi and G. Shapiro, "LOCO-I: A Low Complexity, Context-Based Lossless Image Compression Algorithm", *IEEE Data Compression Conference*, 1996.

[15] X. Wu, "Lossless Compression of Continuous-Tone Images via Context Selection, Quantization and Modeling", *IEEE Transactions Image Processing*, pp. 656-664, 1997.

[16] R. Floyd and L. Steinberg. "An Adaptive Algorithm for Spatial Gray Scale", *Proc. Soc. Inf. Display*, Vol. 17, pp. 75–77, 1976.

[17] W. Kou, *Digital Image Compression Algorithms and Standards*, Kluwer Academic Publishers, Boston, MA (1995).

[18] CCITT, Recommendation T.4, Standardization of Group 3 Facsimile Apparatus for Document Transmission, ISO CCITT, 1980.

[19] CCITT, Recommendation T.6, Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus, ISO CCITT, 1988.

[20] JBIG. Progressive Bi-Level Image Compression. ISO/IEC International Standard 11544, 1993.

[21] M. Fu, "Data Hiding Watermarking for Halftone Images", *IEEE Transactions Image Processing*, pp. 477-484, April 2002.

[22] Xilinx, "ModelSim Xilinx Edition II", http://www.xilinx.com.

[23] R. Hamming, *Coding Information and Theory*, Prentice Hall, Englewood Cliffs, NJ (1986).

[24] M. Piorun, "Hardware Implementation of a JPEG-LS Codec", Rochester Institute of Technology, September 2001.

[25] R. van der Vleuten, "Low-Complexity Lossless and Fine-Granularity Scalable Near-Lossless Compression of Color Images", *Data Compression Conference*, pp. 477, April 2002.

[26] A. Bilgin, G. Zweig and M. Marcellin, "Efficient lossless coding of medical image volumes using reversible integer wavelet transforms", *Data Compression Conference*, pp. 428 –437, April 1998.

[27] C. Sim, W. Wong, and K. Ong, "Segmented approach for lossless compression of medical images", *Proceedings of IEE*, Vol. 2, pp. 554-557, September 1993

[28] C. Constantinescu and J. Storer, J, "Application of single-pass adaptive VQ to bilevel images", *Data Compression Conference*, pp. 423, March 1995.

[29] Y. Chen, H. Peterson and W. Bender, "Lossy compression of palletized images", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 5, pp. 325-328, April 1993.

[30] J. Saghri and A Tescher, "Feature-based lossy compression of multispectral data", *IGARSS Proceedings*, pp. 2046-2050, July 1999.

[31] J. Wilson, "Wavelet-based lossy compression of turbulence data", *Data Compression Conference*, pp. 578, March 2000.

[32] H. Grosse, M. Varley, T. Terrell and Y. Chan, "Hardware implementation of versatile zigzag-reordering algorithm for adaptive JPEG-like image compression schemes", *Sixth International Conference on Image Processing and Its Applications*, Vol. 1, pp. 184-188, July 1997.

[33] J. Singh, A. Antoniou and D. Shpak, "Hardware implementation of a wavelet based image compression coder", *IEEE Symposium on Advances in Digital Filtering and Signal Processing*, pp. 169-173, June 1998.

[34] W. Turri, W. Smari and F. Scarpino, "Integer division for quantization of wavelet-transformed images, using field programmable gate arrays", *Proceedings of the 44th IEEE Midwest Symposium on Circuits and Systems*, Vol. 1, pp. 176-179, August 2001.

[35] Y. Pen-Shu, J. Venbrux, P. Bhatia and W. Miller, "A real-time high performance data compression technique for space applications", *IEEE Geoscience and Remote Sensing Symposium*, Vol. 2, pp. 612-614, July 2000.

[36] L. Chang; C. Cheng; T. Chen, "An efficient adaptive KLT for multispectral image compression", *IEEE Southwest Symposium Image Analysis and Interpretation*, pp. 252-255, April 2000.

[37] S. Yu and N. Galatsanos, "Binary decompositions for high-order entropy coding of grayscale images", *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 21-31, February 1996.

[38] R. Ansari, N. Memon and E. Ceran, "Near-lossless Image Compression Techniques", *Journal of Electronic Imaging*, Vol. 7, No. 3, pp. 486-494, July 1998.