UNIVERSITY OF CALIFORNIA

Santa Barbara

JPEG Image Compression Using an FPGA

A Thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Electrical and Computer Engineering
by

James Rosenthal

Committee in charge:
Professor Steven Butner, Chair
Professor Michael Melliar-Smith
Professor Tim Cheng

December 2006

The thesis of James Rosenthal is approved.

Tim Cheng

Michael Melliar-Smith

Steven Butner, Committee Chair

December 2006

JPEG Image Compression Using an FPGA

James Rosenthal

ABSTRACT

JPEG Image Compression Using an FPGA

by

James Rosenthal

Image compression is an important topic in commercial, industrial, and academic applications. Whether it be in commercial photography, industrial imaging, or video, digital pixel information can comprise considerably large amounts of data. Management of such data can involve significant overhead in computational complexity, storage, and data processing. Typical access speeds for storage mediums are inversely proportional to capacity. Through data compression, such tasks can be optimized.

Image and video compressors and decompressors (codecs) are implemented mainly in software as digital signal processors have optimized instruction sets to manage the required operations. Hardware-specific codecs can be integrated into digital systems fairly easily, requiring work only in the areas of interface and overall integration. Improvements in speed occur primarily because the hardware can be tailored to the compression algorithm as well as the application. Using an FPGA to implement a codec combines the best of two worlds: significantly increased processing speed due to the use of customized hardware, and flexibility to make changes and tunings of the algorithm since FPGA-based designs are

easily modified.

The JPEG algorithm was chosen for this project as it is well defined and highly recognizable. JPEG provides a baseline compression algorithm that can be modified in numerous ways to fit any desired application. The JPEG specification, released initially in 1991, does not specify a particular implementation.

A programmable hardware platform, developed in the computer architecture laboratory at UCSB, was chosen as a substrate for this project. The baseline JPEG compression algorithm was tailored to fit this board, using custom hardware pipelining, as well as parallel data paths. The core compression design was created using the Verilog hardware description language. THe supporting software was written in C, developed for a DSP and the PC.

The implementation of this project was successful on achieving significant compression ratios. The sample images chosen showed different degrees of contrast and fine detail to show how the compression affected high frequency components within the images. The throughput of the design excelled in the FPGA core. However, inherent limitations in the interface to the FPGA limited the overall performance of the design.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Image Compression

Image compression is an important topic in the digital world. Whether it be commercial photography, industrial imagery, or video. A digital image bitmap can contain considerably large amounts of data causing exceptional overhead in both computational complexity as well as data processing. Storage media has exceptional capacity, however, access speeds are typically inversely proportional to capacity. [8] Compression is important to manage large amounts of data for network, internet, or storage media. Compression techniques have been studied for years, and will continue to improve.

Typically image and video compressors and decompressors (CODECS) are performed mainly in software as signal processors can manage these operations without incurring too much overhead in computation. However, the complexity of these operations can be efficiently implemented in hardware. Hardware specific CODECS can be integrated into digital systems fairly easily. Improvements in speed occur primarily because the hardware is tailored to the compression algorithm rather than to handle a broad range of operations like a digital signal processor.

Data compression itself is the process of reducing the amount of information into a smaller data set that can be used to represent, and reproduce the information. Types of image compression include lossless compression, and lossy compression techniques that are used to meet the needs of specific applications. JPEG compression can be used as a lossless or a lossy process depending on the requirements of the application. Both lossless and lossy compression techniques

employ reduction of redundant data.

Work in standardization has been controlled by the International Organization for Standardization (ISO) in cooperation with the International Electrotechnical Commission (IEC). The Joint Photographic Experts Group produced the well-known image format JPEG, a widely used image format. [2] JPEG provides a solid baseline compression algorithm that can be modified numerous ways to fit any desired application. The JPEG specification was released initially in 1991, although it does not specify a particular implementation.

1.2 Redundancy Coding

To compress data, it is important to recognize redundancies in data, in the form of coding redundancy, inter-pixel redundancy, and psycho-visual redundancy. [3] Data redundancies occur when unnecessary data is used to represent source information. Compression is achieved when one or more of these types of redundancies are reduced. [3]Intuitively, removing unnecessary data will decrease the size of the data, without losing any important information. However, this is not the case for psycho-visual redundancy.

The most obvious way to reduce compression is to reduce the coding redundancy. This is referring to the entropy of an image in the sense that more data is used than necessary to convey the information. Lossless redundancy removal compression techniques are classified as entropy coding. Other compression can be obtained through inter-pixel redundancy removal. Each adjacent pixel is highly related to its neighbors, thus can be differentially encoded rather than sending the entire value of the pixel. Similarly adjacent blocks have the same property, although not too the extent of pixels.

In order to produce error free compression, it is recommended that only coding redundancy is reduced or eliminated. [3] This means that the source image will be exactly the same as the decompressed image. However, inter-pixel redundancies can also be removed, as the exact pixel value can be reconstructed from differential coding or through run length coding.

Psycho-visual redundancy refers to the fact that the human visual system will interpret an image in a way such that removal of this redundancy will create an image that is nearly indistinguishable by human viewers. The main way to reduce this redundancy is through quantization. Quantizing data will reduce it to levels defined by the quantization value. Psycho-visual properties are taken advantage of from studies performed on the human visual system.

1.3 The Human Visual System

The Human Visual System(HVS) describes the way that the human eye processes an image, and relays it to the brain. By taking advantage of some properties of HVS, a lot of compression can be achieved. In general, the human eye is more sensitive to low frequency components, and the overall brightness, or luminance of the image.

Images contain both low frequency and high frequency components. Low frequencies correspond to slowly varying color, whereas high frequencies represent fine detail within the image. Intuitively, low frequencies are more important to create a good representation of an image. Higher frequencies can largely be ignored to a certain degree.

The human eye is more sensitive to the luminance(brightness), than the chrominance(color difference) of an image. Thus during compression, chromi-

nance values are less important and quantization can be used to reduce the amount of psycho-visual redundancy. Luminance data can be quantized, but more coarsely to ensure that important data is not lost.

Several compression algorithms use transforms to change the image from pixel values representing color to frequencies dealing with light and dark of an image, not frequencies of light. Many forms of the JPEG compressions algorithm make use of the discrete cosine transform. Other transforms such as wavelets are employed by other compression algorithms. These models take advantage of subjective redundancy by exploiting the human visual system sensitivity to image characteristics. [1]

1.4 Transform Coding

Another form of compression technique aside from exploiting redundancies in data is known as transform coding or block quantization. Transform coding employs techniques such as differential pulse code modulation as well as other predictive compression measures. Transform coding works by moving data from spatial components to transform space such that data is reduced into a fewer number of samples. Transform coders effectively create an output that has a majority of the energy compacted into a smaller number of transform coefficients. [8]

The JPEG image compression algorithm makes use of a discrete cosine transform to move pixel data representing color intensities to a frequency domain. Most multimedia systems combine both transform coding and entropy coding into a hybrid coding technique. [5] The most efficient transform coding technique employs the Karhunen-Loeve-Hotelling (KLH) transform. The KLH transform has the best results of any studied transform regarding the best energy com-

paction. However, the KLH transform has no fast algorithm or effective hardware implementation. Thus, JPEG compression replaces the KLH transform with the discrete cosine transform, which is closely related to the discrete fourier transform.

Transform coders typically make use of quantizers to scale the transform coefficients to achieve greater compression. As a majority of energy from the source image is compacted into few coefficients, vector quantizers can be used to coarsely quantize the components with little to no useful information, and finely quantize the more important coefficients. A major benefit to transform coding is that distortion or noise produced by quantization and rounding gets evenly distributed over the resulting image through the inverse transform. [8]

1.5 Lossless Compression

Lossless compression techniques work by removing redundant information as well as removing or reducing information that can be recreated during decompression. [2] Lossless compression is ideal, as source data will be recreated without error. However, this leads to small compression ratios and will most likely not meet the needs of many applications. Compression ratios are highly dependent on input data, thus lossless compression will not meet the requirements of applications requiring a constant data rate or data size.

Lossless techniques employ entropy encoders such as Huffman encoders. Huffman produced an efficient variable length coding scheme in 1952. [2] Such encoders similar to PKZIP, a popular public domain compression program, make use of order and patterning within data sets. This property allows entropy coding, such as run length coding, to compress data without any loss.

Entropy encoders give a codeword to each piece of data. The codeword is of variable length to enhance compression. The varying length is typically determined by the frequency that a certain piece of data appears. Some algorithms generate codewords after analyzing the data set, and others use standard codewords already generated based off of average statistics. Shorter codewords are assigned to those values appearing more frequently, where longer codewords are assigned to those values that occur less frequently. This is the property that the Morse Alphabet was created off of. Applying Morse coding to the English language, vowels such as 'e' occur most frequently, so small codewords are assigned. Other letters such as 'z' and 'q' occur much less frequently and thus are assigned longer codewords. These same principles are the foundation for many entropy encoders.

Run length coding, another form of entropy coding, was created to exploit the nature of inter-pixel redundancy. As each pixel is highly correlated to its neighbors, it can be expected that certain values will be repeated in adjacent pixels. By encoding this data repetition as a run length, significant compression can be achieved. When employed in lossy compression systems, run length coding can achieve significant compression compared to lossless compression, especially after quantization.

The best known lossless image compression algorithm is the Compuserve Graphics Interchange Format (GIF). Unfortunately the GIF format cannot handle compression of images with resolutions greater than 8-bits per pixel. Another lossless format called the Portable Network Graphics (PNG) can compress images with 24-bit or 48-bit color resolutions.

1.6 Lossy Compression

The main benefit of lossy compression is that the data rate can be reduced. This is necessary as certain applications require high compression ratios along with accelerated data rates. Of course significant compression can be achieved simply by removing a lot of information and hence quality from the source image, but this is often inappropriate.

Lossy compression algorithms attempt to maximize the benefits of compression ratio and bit rate, while minimizing loss of quality. [2] Finding optimal ways to reach this goal is a severely complicated process as many factors must be taken into account. Variables such as a quality factor which is used to scale the quantization tables used in JPEG can either reduce the resulting image quality with higher compression ratios, or conversely improving image quality with lower compression ratios. JPEG, although lossy, can give higher compression ratios than GIF while leaving the human observer with little to complain of loss in quality. Minimizing the variance between source and compressed image formats, otherwise known as Mean Square Error, is the ultimate goal for lossy compression algorithms.

Compression induced loss in images can cause both missing image features as well as artifacts added to the picture. Artifacts are caused by noise produced by sources such as quantization, and may show up as blocking within the image. Blocking artifacts within an image can become apparent with higher compression ratios. The edges are representative of blocks used during compression, such as 8x8 blocks of pixels, used in JPEG. Strangely, artifacts such as noise or ringing may actually improve the subjective quality of the image. [2] Noise can cause contouring which shows up in gradual shading regions of the image, while ringing is apparent at sharp edges. JPEG unfortunately does not do well with computer-

generated graphics.

1.7 Color Space

Compression also can come from the way color is represented within pixel data values. Monochrome images simply use one number to indicate the luminance of the sample. [1] In order to represent color within an image several values are used. There are several formats used to represent color, the two most common being RGB and Luminance/Chrominance, which both employ three-value pixel representations.

The RGB format represents a colored image in three separate parts. Each of the three samples represent the same image, just the relative proportions of red, green, and blue for each given pixel. [1] These are primary colors so any color or shade can be produced by a combination of red, green, and blue intensities. This is known as additive coloring.

The other most commonly used format is luminance(Y), along with two chrominance components (Cr and Cb). Cr and Cb represent the chrominance, or color difference within the sample. Actually there are three components to chrominance, Cr, Cb, and Cg for red, blue, and green chrominance. However, Cg can be derived from Y, Cr, and Cb. Thus only two components need be compressed as the image translator can infer the third based upon the other two. The two chrominance values make up a two-dimensional vector, where the phase gives the hue, and the magnitude gives color saturation. Chrominance is the color quality of light defined by its wavelength.

Using luminance and chrominance has major advantages for compression in comparison to RGB. Most importantly luminance and chrominance values are

separated rather than being incorporated in the same value such as RGB. RGB includes both values in each respective color. As the human visual system is less sensitive to color, the chrominance values can be further reduced in compression techniques while not severely altering the image as seen by a human observer. Luminance values alone can recreate an accurate monochrome image.

1.8 JPEG Compression

JPEG compression is defined as a lossy coding system which is based on the discrete cosine transform. However, there are several extensions to the JPEG algorithm to provide greater compression, higher precision, and can be tailored to specific applications. [3] It can also be used as a lossless coding system that may be necessary for applications requiring precise image restoration. JPEG has four defined extension modes: sequential DCT based, progressive DCT based, lossless, and hierarchical mode.

1.8.1 Sequential DCT Based

The sequential DCT based mode of operation comprises the baseline JPEG algorithm. This technique can produce very good compression ratios, while sacrificing image quality. The sequential DCT based mode achieves much of its compression through quantization, which removes entropy from the data set. Although this baseline algorithm is transform based, it does use some measure of predictive coding called the differential pulse code modulation (DPCM). After each input 8x8 block of pixels is transformed to frequency space using the DCT, the resulting block contains a single DC component, and 63 AC components. The DC component is predictively encoded through a difference between the current

DC value and the previous. This mode only uses Huffman coding models, not arithmetic coding models which are used in JPEG extensions. This mode is the most basic, but still has a wide acceptance for its high compression ratios, which can fit many general applications very well.

1.8.2 Progressive DCT Based

Progressive DCT based JPEG compression actually uses two complimentary coding methods. [9] The goal of this extension is to display low quality images during compression which successively improve. The first method for such a technique is known as spectral-selection. This implies that data is compressed in bands. The first band contains DC components and a very few AC components to get an image that is somewhat discernable. The second method employed is known as successive approximation. This method at first will grossly quantize the coefficients after the DCT, which will result in a small data set, which will incur large amounts of blocky artifacts during decompression. The following scans will contain information about the difference between the quantized and non-quantized coefficients, using finer quantization steps. This will allow the image to slowly come into focus during decompression. Again, this method is used in applications where these features are desired.

1.8.3 Lossless Mode

Quite simply, this mode of JPEG experiences no loss when comparing the source image, to the reproduced image. This method does not use the discrete cosine transform, rather it uses predictive, differential coding. As it is lossless, it also rules out the use of quantization. This method does not achieve high compression ratios, but some applications do require extremely precise image reproduction.

1.8.4 Hierarchical Mode

The hierarchical JPEG extension uses a multi-stage compression approach, with prediction, and can use the encoding methods from the progressive, sequential or lossless modes of operation. The strategy is to down sample the image in each dimension. Then code this data set using one of the three methods discussed, lossless, sequential, or progressive. The resulting encoded data stream is to be decoded, and up-sampled to recreate the source image. Then the process encodes the difference between the recreated image and the source. This process can be repeated multiple times.

2 Baseline JPEG Compression

The baseline JPEG compression algorithm is the most basic form of sequential DCT based compression. By using transform coding, quantization, and entropy coding, at an 8-bit pixel resolution, a high-level of compression can be achieved. However, the compression ratio achieved is due to sacrifices made in quality. The baseline specification assumes that 8-bit pixels are the source image, but extensions can use higher pixel resolutions. JPEG assumes that each block of data input is 8x8 pixels, which are serially input in raster order. Similarly, each block is sequentially input in raster order.

Baseline JPEG compression has some configurable portions, such as quantization tables, and Huffman tables, which can individually be specified in the JPEG file header. By studying the source images to be compressed, Huffman codes and quantization codes can be optimized to reach a higher level of compression without losing more quality than is acceptable. Although this mode of JPEG is not highly configurable, it still allows a considerable amount of compression. Furthermore compression can be achieved by subsampling chrominance portions of the input image, which is a useful technique playing on the human visual system.

2.1 Level Shift

In order to make the data fit the discrete cosine transform, each pixel value is level shifted by subtracting 128 from its value. The result of this is 8-bit pixels that have the range of -127 to 128, making the data symmetric across 0. This is good for DCT as any symmetry that is exposed will lead toward better entropy

compression. Effectively this shifts the DC coefficient to fall more in line with value of the AC coefficients. The AC coefficients produced by the DCT are not affected in any way by this level shifting.

2.2 Discrete Cosine Transform

The discrete cosine transform is the basis for the JPEG compression standard. Ahmed, Natarajan, and Rao originally proposed use of the DCT in 1974, and it has become the most popular transform for image and video coding.[1] Many compression techniques take advantage of transform coding as it decorrelates adjacent pixels from the source image. For JPEG, this allows for efficient compression by allowing quantization on elements that are less sensitive. The DCT algorithm is completely reversible making this useful for both lossless and lossy compression techniques.

The DCT is a special case of the well known Fourier transform. Essentially the Fourier transform in theory can represent a given input signal with a series of sine and cosine terms. The discrete cosine transform is a special case of the Fourier transform in which the sine components are eliminated.[4] For JPEG, a two-dimensional DCT algorithm is used, which is essentially the one-dimensional version evaluated twice. By this property there are numerous ways to efficiently implement a software or hardware based DCT module.[1] The DCT is operated two dimensionally taking into account an 8 by 8 block of pixels. The resulting data set is an 8 by 8 block of frequency space components, the coefficients scaling the series cosine terms, known as basis functions. The first element at row 0 and column 0, is known as the DC term, the average frequency value of the entire block. The other 63 terms are AC components which represent the spatial

frequencies that compose the input pixel block, by scaling the cosine terms within the series.

There are two useful products of the DCT algorithm. First it has the ability to concentrate image energy into a small number of coefficients. Second, it minimizes the interdependencies between coefficients.[1] These two points essentially state why this form of transform is used for the standard JPEG compression technique. By compacting the energy within an image, more coefficients are left to be quantized coarsely, impacting compression positively, but not losing quality in the resulting image after decompression. Taking away inter-pixel relations allows quantization to be non-linear, also affecting quantization positively. DCT has been effective in producing great pictures at low bit rates and is fairly easy to implement with fast hardware based algorithms.[3]

Transform Coding

Transform coding is widely used among compression algorithms. By taking images from the spatial domain to the frequency domain, it makes the data more amenable to compression.[1] Different techniques in transform coding yield different results. Some have exceptional performance at the cost of computational complexity. Others are easy to compute, but lose in the compression aspect. For instance, a well known transform is the Karhunen-Loeve-Hotelling transform. This method gives the best performance by compacting energy efficiently into a minimal coefficient set. However, this technique is computationally inefficient especially during the reverse transform.[1]

The DCT is one of many transform functions that is widely used among image compression algorithms. This particular one was chosen for its ability to decorrelate image pixels within the spatial domain, as well as being an orthogo-

nal transform. An orthogonal transform such as the DCT has the good property that the inverse DCT can take its frequency coefficients back to the spatial domain at no loss. However, implementations can be lossy due to bit limitations, especially apparent in those algorithms in hardware. The DCT as determined by Ahmed, Natarajan, and Rao was discovered to be particulary close the KLH transform, in terms of performance.[1] The DCT does win in terms of computational complexity as there are numerous studies that have been completed in different techniques for evaluating the DCT.

The discrete cosine transform is actually more efficient in reconstructing a given number of samples, as compared to a fourier transform. By using the property of orthogonality of cosine, as opposed to sine, a signal can be periodically reconstructed based on a fewer number of samples. Any sine based transform is not orthogonal, and would have to take fourier transforms of more numbers of samples to approximate a sequence of samples as a periodic signal. As the signal we are sampling, the given image, there is actually no real periodicity. If the image is run through a fourier transform, the sine terms can actually incur large changes in amplitude for the signal, due to sine not being orthogonal. DCT will avoid this by not carrying this information to represent the changes.[2]

Basis Functions

The DCT is a one dimensional process that has 8 basis functions, which represent the frequency domain. Each basis function is the pixel pattern that results when that particular DCT coefficient is set to its maximum value and all the other coefficients are set to zero.[2] The DCT essentially correlates the input image with each of the basis functions. In the case of JPEG, a two-dimensional DCT is used, which correlates the image with 64 basis functions, shown in figure 2.1.

At this point the transform has taken the input image and changed it into values within that scale each of the basis functions, to represent the input image. In the frequency domain, the new frequency values are highly separable, and can be quantized more efficiently without losing the spatial correlations that would be susceptible to the human visual system.

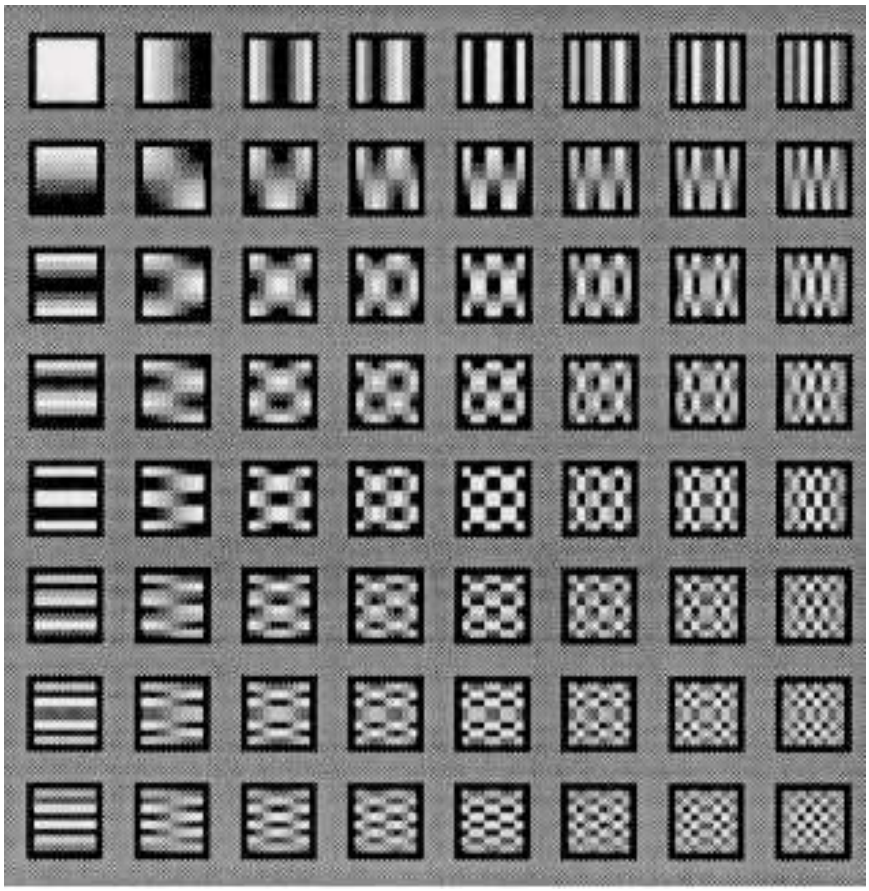

Figure 2.1: DCT Basis Functions

2.3 Zigzag Scanning

The zigzag process is an approximate ordering of the basis functions from low to high spatial frequencies. This process will give more compression by putting more order in the entropy.[2] Essentially our lower-frequency components, which describe the gradual luminance changes, are more important to the human visual system than the high frequency changes. By ordering the more important coefficients in the beginning of the 8x8 block, we can expect more runs of zeros later after quantization, toward the end of the 8x8 block. This will aid in further compression in the entropy encoding that will be discussed later.

2.4 Quantization

Quantization is an extremely important step in the JPEG compression algorithm, as it removes a considerable amount of information, thus reducing the entropy in the input data stream. Unfortunately, quantization is irreversible making Baseline JPEG lossy.[6] Quantization alone, is essentially a lossy compression technique. However, quantization does benefit the compression process, regardless of the lossy artifacts that are produced. The high frequency AC coefficients typically will become zero, which aids in entropy coding. Quantization is intended to remove the less important components to the visual reproduction of the image.[1]

Baseline JPEG allows for custom quantization tables to be defined within the encoded file headers. The quantization tables can be linear or non-linear. However, quantization is most effective when less important elements are quantized more coarsely. Larger quantization values will result in visual artifacts. Non-zero AC coefficients after quantization are worse for compression, but will suppress

blocking artifacts in the reconstructed image. Blocking artifacts indicate high spatial frequencies caused by an absence of AC coefficients.[6] Essentially, quantization, when used effectively, will result in high compression, with minimal loss in quality.

Types of Quantizers

Quantizers can either be linear or non-linear. Linear quantization is when input values map to a set of evenly distributed output values. This is adequate when a high level of precision is required across the entire range of possible input values.[1] Non-linear modes treat each input value differently.

Scale Factor

The scale factor is often a main parameter to control image quality and compression in an image codec.[1] The change in the factor will adjust the number of steps in the resulting quantized value. Larger quantization steps will lead to greater distortion and a smaller resulting data set. The smaller the steps will distort less, but will create a larger resulting data set.[6]

Tables

Tables essentially determine the quality of the output image. Fortunately, in the baseline JPEG algorithm, custom tables are allowed as long as they are defined in the compressed JPEG file headers. The tables should take into account that the human visual system is less sensitive to amplitude errors at higher frequency coefficients. Thus, those should be quantized coarsely.[2] For better performance, different tables are used for quantizing luminance and chrominance data sets. This is because the human visual system is more sensitive to brightness changes,

rather than color difference. The DCT process concentrates the energy of the data set into the upper left hand corner coefficients of the 8x8 block, and the quantization matrix will emphasize this.[2]

2.5 DC Differential Coding

To further add to the compression sequence, a process called differential pulse code modulation, or DC differential coding is used to reduce the entropy of the data set. Essentially, this process allows the first DC value of the first block to be passed through to the entropy coding module. Afterward, the value passed is a difference between the current block DC value and the previous block. This is done in raster order throughout all blocks of the image.

Adjacent blocks in a given image contain a high degree of similarity in both color and luminance. This makes the DC differential coding perform in a better way, by reducing more entropy in the data stream. After the first block, the DC terms after the difference module, will become zero or close to zero in many cases. This simple step actually adds to the compression substantially. Much like quantization, this step is used to reduce entropy in the data stream.

$$DIFF = DC_i - DC_{i-1} \tag{2.1}$$

2.6 Entropy Coding

Entropy is a measure of disorder and unpredictability. The degree of entropy can be used as a measure of the information carried by the message.[2] Up to

this point the input image has run through the DCT process, followed by zigzagging to approximately organize the data stream with more entropy toward the beginning of the stream. Then this stream was quantized, and run through the DC differencing in an effect to reduce entropy in the data stream. All of these processes both add to compression, and more importantly, make the data stream ready for entropy coding.

2.6.1 Run Length Coding

The first step in entropy coding is known as run length coding. This is a simple thought that is accomplished by assigning a code, run length and size, to every non-zero value in the quantized data stream. The run length is a count of zero values before the non-zero value occurred. The size is a category given to the non-zero value which is used to recover the value later. The DC value of the block is omitted in this process. Additionally, with every non-zero value a magnitude is generated which determines the number of bits that are necessary to reconstruct the value. It will indicate possible values in the size category that can be correct.[2] Run Length coding is a basic form of lossless compression.

Essentially, this process is a generalization of zero suppression techniques. Zero suppression assumes that one symbol or value appears often in a data stream.[5] After quantization the goal is that most high frequency components, which are less important to the human visual system, are set to zero. The zigzag process organized the sequence to have the lower frequency components which are less likely to be zero in the first part of the data stream. This effectively has organized the data to have larger runs of zeros, especially at the end, making the run length coding very efficient.

2.6.2 Huffman Coding

Huffman coding is a technique which will assign a variable length codeword to an input data item. Huffman coding assigns a smaller codeword to an input that occurs more frequently. It is very similar to Morse code, which assigned smaller pulse combinations to letters that occurred more frequently. Huffman coding is variable length coding, where characters are not coded to a fixed number of bits.[5]

This is the last step in the encoding process. It organizes the data stream into a smaller number of output data packets by assigning unique codewords that later during decompression can be reconstructed without loss. For the JPEG process, each combination of run length and size category, from the run length coder, are assigned a Huffman codeword.

2.7 Error

The baseline JPEG algorithm is a lossy compression algorithm. It is expected that decompression will lead to artifacts that are introduced by factors such as error in the JPEG compression process. Starting with the discrete cosine transform, the transform is in itself a lossless process. However, when implemented if there are a finite number of bits used, such as in hardware, there will be truncation in the output data. Although this does not explain all artifacts and errors in the reproduced image, it does add to the problem. Quantization adds a majority of the error in the reconstructed image, as the process is irreversible. The DC differential coding, and entropy coding are lossless as well.

Error in reconstructed images are mainly artifacts due to quantization. In quantization we give up more information on high frequency components, which

are less important to the human visual system, but still there is a non-zero amount of information lost. Errors in images typically come in the form of blocky artifacts around high frequency components, sharp edges, high amounts of color difference, and areas of fine detail. Fortunately the baseline JPEG algorithm still holds enough of the information to make the error between the source and resulting images acceptable to the human eye.

2.8 JPEG File Construction

The JPEG header format is necessary for several reasons. The headers both define the image type, size, quality, and more importantly it has information that the decoder must use to correctly reproduce the image. The headers are prescribed by the JPEG File Interchange Format specification, otherwise known as JFIF. The format has several header sections, each of which begins with a two-byte header, which is a unique symbol that will not be mistaken. The file structure allows for multiple scans of entropy coded segments, each which could use different Huffman tables, and such. There is a number of optional components to the header associated with JPEG files. Figure 2.2 depicts the minimum required set of headers which are used for Baseline JPEG.

Markers are used to define the header segments. A marker will always begin with the first byte as 0xFF, and the second byte defines which type of marker it is. The first byte is uniquely identified by the decoder as a header marker. To ensure that a marker is not mistakenly created in the entropy encoded bit stream, whenever a 0xFF is encountered, a byte 0 byte (0x00) is stuffed into the data stream after 0xFF. This is known as zero stuffing. The minimum required headers for Baseline JPEG Compression are shown in table 2.1.

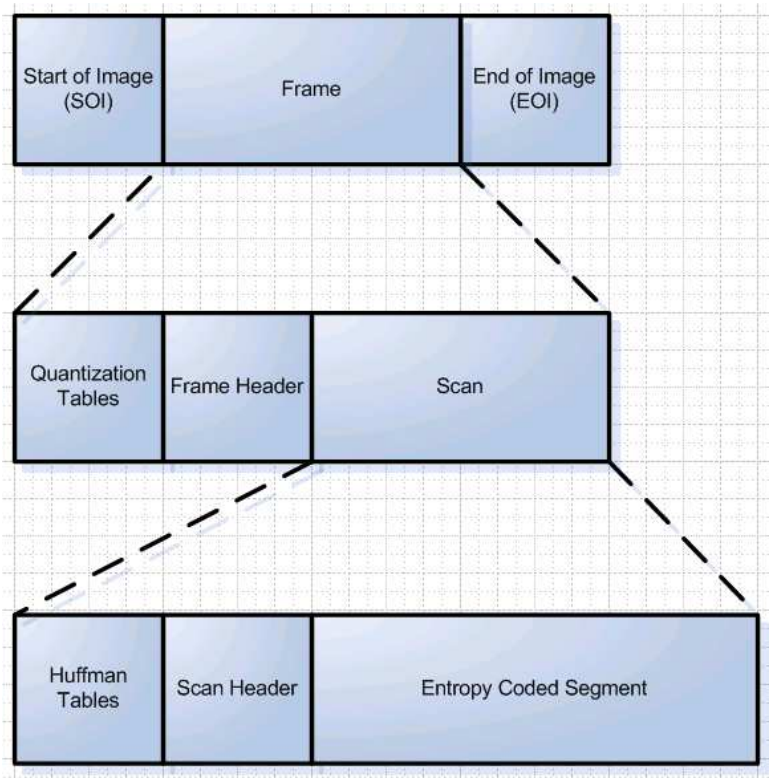

Figure 2.2: High Level File Structure

Marker	Symbol	Description
0xFFD8	SOI	Start of Image Marker
0xFFE0	APP0	Application Specific Marker
0xFFDB	DQT	Define Quantization Table Marker
0xFFC0	SOF	Start of Frame Marker
0xFFC4	DHT	Define Huffman Table Marker
0xFFDA	SOS	Start of Scan Marker
0xFFD9	EOI	End of Image Marker

Table 2.1: JPEG File Markers

2.8.1 Application Specific Data Header

The application specific data header, shown in figure 2.3, is used to define that this file is compliant to the JPEG File Interchange Format, and gives some specific image details. This header is useful as it gives a format that can allow the JPEG bitstreams to be exchanged between a wide variety of platforms and applications. Additionally, a thumbnail image can be defined and included in this header, but this is not a requirement of the Baseline JPEG specification.

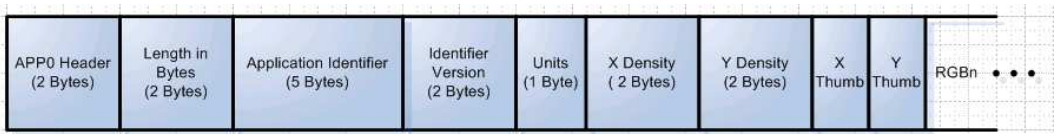

Figure 2.3: Application Specific Header

APP0 Header - 2 Bytes

Application specific marker APP0, 0xFFE0. This uniquely defines the application specific header.

Length - 2 Bytes

Total Length of header including two bytes used to specify length, but does not include header APP0.

Application Identifier - 5 Bytes

Specifically identifies JFIF through ASCII hex codes, 0x4A46494600.

Version Identifier - 2 Bytes

Specifies major and minor version numbers for JFIF, currently version 1.01 (0x0101).

Density Units - 1 Byte

Specifies units used to give pixel densities in the following entries. 0 signifies density given in pixels. 1 denotes density given in dots per inch. 2 specifies density given in dots per cm.

X Density - 2 Bytes

Horizontal pixel density in units specified.

Y Density - 2 Bytes

Vertical pixel density in units specified.

X Thumbnail

If a thumbnail is defined in this header segment, this specifies the horizontal pixel count of the thumbnail.

Y Thumbnail

If a thumbnail is defined in this header segment, this specifies the vertical pixel count of the thumbnail.

RGBn - 3n Bytes

24-bit RGB values given for thumbnail. (n) is given as X thumbnail * Y thumbnail. If X or Y Thumbnail entries are 0, this section is not necessary.

2.8.2 Define Quantization Table Header

The define quantization table header segment, shown in figure 2.4, is used to define both a quantization table for luminance components and chrominance components. This marker is only used in DCT based JPEG algorithms.

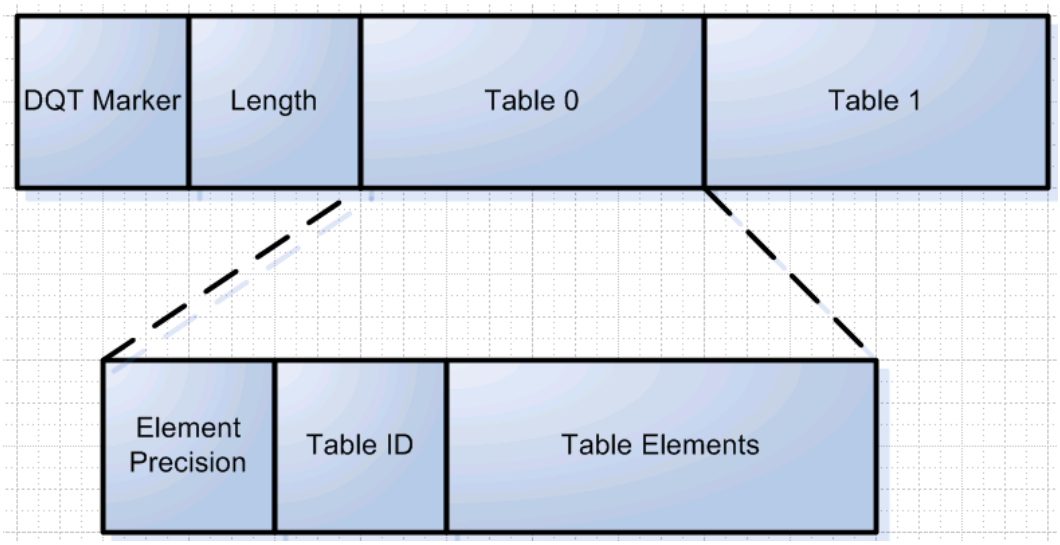

Figure 2.4: Define Quantization Table Segment

DQT Header - 2 Bytes

Define quantization table marker DQT, 0xFFDB. This uniquely defines the quantization table header segment.

Length - 2 Bytes

Length of quantization definition segment including the 2-byte length specification, but not including the two byte marker DQT.

Table Definition - Element Precision - 4 bits

For each table definition in this header segment, this defines the element precision. 0 = 8 bit precision, 1 = 16-bit precision.

Table Definition - Table Identifier - 4 bits

This gives each table definition a unique identifier. The decoder will use this info to apply the correct quantization table to the decoded data.

Table Definition - Quantization Element

There will be 64 elements defined per table, given in the precision specified above.

Next Table Definition

The next table definition follows the same format as the first definition, but will be assigned a different identifier, potentially different precision, and likely different quantization elements.

Tables

The quantization tables used for baseline JPEG compression can be customized, but there are prescribed tables in Annex K of Recommendation T.81, from the International Telecommunications Union. These tables for luminance and chrominance are shown in tables 2.2 and 2.3 respectively.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 2.2: Luminance Quantization Table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Table 2.3: Chrominance Quantization Table

2.8.3 Frame Header Segment

SOF is the start of frame marker, which is unique for each type of JPEG implementation. This header segment defines parameters that apply to all scans within the frame.

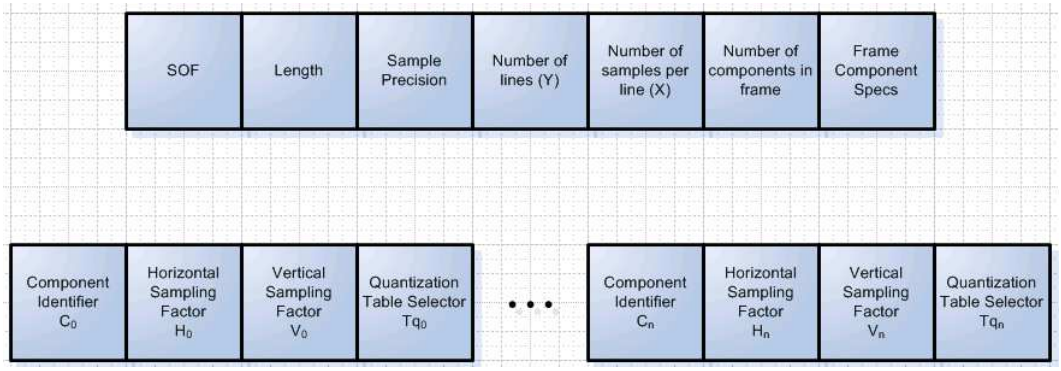

Figure 2.5: Start of Frame Header Segment

SOF Marker - 2 Bytes

The start of frame marker defined by the unique two byte segment 0xFFC0. This marker actually has several flavors which specify start of frame for the different modes of JPEG operation.

Length - 2 Bytes

This field defines the length for the frame header in bytes. It includes the 2 bytes used to define the length, but not the two bytes used for the marker.

Precision - 1 Byte

Defines the sample precision used for this type of JPEG compression algorithm. For baseline, this is 8-bit, specified by the value 0x08.

Dimension Y - 2 Bytes

Defines the aspect ratio in number of rows. The vertical aspect ratio.

Dimension X - 2 Bytes

Defines the number of samples per row. The horizontal aspect ratio.

Components - 1 Byte

Defines the number of components in this frame. This is typically 1 for grayscale images and 3 for color images, as we have luminance, and two chrominance components in each color scan.

Component Identifer - 1 Byte

This defines an identification number for this frame component that is going to be specified. The following entries relate to this component identification, including the quantization table identifier.

Horizontal Sampling - 4 bits

This applies to the component being specified, but can be used to give a factor which states a 1:1, 2:1, or other sampling factor. This is useful for chrominance components which can be subsampled to create a smaller data set to compress.

Vertical Sampling Factor - 4 bits

This applies to the component being specified, but can be used to give a factor which states a 1:1, 2:1, or other sampling factor. This is useful for chrominance components which can be subsampled to create a smaller data set to compress.

Quantization Table Selector - 1 Byte

Defines which quantization table will be used for this component. These were defined in the define quantization table header segment.

2.8.4 Huffman Table Definition Segment

DHT is the marker for the define Huffman table segment of the header, as shown in figure 2.8.4. This segment will define each component of the Huffman table for the particular frame components that are used. Huffman coding efficiency can be improved with custom tables, which are allowed to be specified within this header segment.

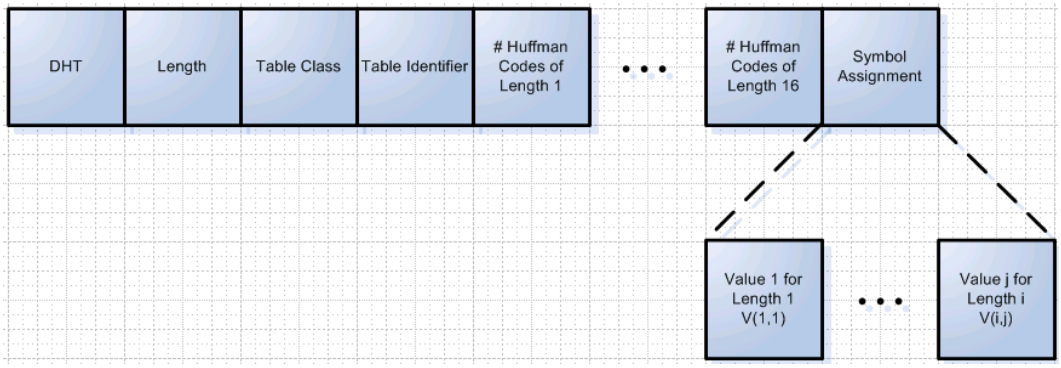

Figure 2.6: Define Huffman Table Segment

Define Huffman Table Header - 2 Bytes

Defines the Huffman table header segment uniquely with the marker DHT (0xFFC4). This segment will define all of the values that make up the Huffman codeword tables.

Length Definition - 2 Bytes

This defines the length of the Huffman table segment of the headers. It is a length in bytes, including the 2 bytes used for the length definition, and not including the two byte marker DHT.

Table Class - 4 bits

This defines the type of table, 1 for AC or 0 for DC.

Table Identification - 4 bits

This defines an identification for the table, 0 or 1. There will be two tables for luminance components, one for DC and one for AC. Similarly, the chrominance components together have two tables, one for AC and one for DC. Further elaborate table definitions can be used in custom applications.

Number of Huffman codes of Length i - 1 Byte

There are 16 length categories consisting Huffman codes. This defines the number of codes within category i.

Value Associated with Huffman Code - 1 Byte

This has a definition associated with each Huffman code within each length category. Each of these are the elements within the Huffman table.

Huffman Tables

DC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Table 2.4: DC Luminance Huffman Coefficients

DC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
0	2	00
1	2	01
2	2	10
3	3	110
Continued on the next page.		

DC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

Table 2.5: DC Chrominance Huffman Coefficients

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111110001100
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101
4/1	6	111011
4/2	10	1111111000
4/3	16	1111111110010110
4/4	16	1111111110010111
4/5	16	1111111110011000
4/6	16	1111111110011001
4/7	16	1111111110011010
4/8	16	1111111110011011
4/9	16	1111111110011100
4/A	16	1111111110011101
5/1	7	1111010
5/2	11	11111110111
5/3	16	1111111110011110
5/4	16	1111111110011111
5/5	16	1111111110100000
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
5/6	16	1111111110100001
5/7	16	1111111110100010
5/8	16	1111111110100011
5/9	16	1111111110100100
5/A	16	1111111110100101
6/1	7	1111011
6/2	12	111111110110
6/3	16	1111111110100110
6/4	16	1111111110100111
6/5	16	1111111110101000
6/6	16	1111111110101001
6/7	16	1111111110101010
6/8	16	1111111110101011
6/9	16	1111111110101100
6/A	16	1111111110101101
7/1	8	11111010
7/2	12	111111110111
7/3	16	1111111110101110
7/4	16	1111111110101111
7/5	16	1111111110110000
7/6	16	1111111110110001
7/7	16	1111111110110010
7/8	16	1111111110110011
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
7/9	16	1111111110110100
7/A	16	1111111110110101
8/1	9	111111000
8/2	15	111111111000000
8/3	16	1111111110110110
8/4	16	1111111110110111
8/5	16	1111111110111000
8/6	16	1111111110111001
8/7	16	1111111110111010
8/8	16	1111111110111011
8/9	16	1111111110111100
8/A	16	1111111110111101
9/1	9	111111001
9/2	16	1111111110111110
9/3	16	1111111110111111
9/4	16	1111111111000000
9/5	16	1111111111000001
9/6	16	1111111111000010
9/7	16	1111111111000011
9/8	16	1111111111000100
9/9	16	1111111111000101
9/A	16	1111111111000110
A/1	9	111111010
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
A/2	16	1111111111000111
A/3	16	1111111111001000
A/4	16	1111111111001001
A/5	16	1111111111001010
A/6	16	1111111111001011
A/7	16	1111111111001100
A/8	16	1111111111001101
A/9	16	1111111111001110
A/A	16	1111111111001111
B/1	10	1111111001
B/2	16	1111111111010000
B/3	16	1111111111010001
B/4	16	1111111111010010
B/5	16	1111111111010011
B/6	16	1111111111010100
B/7	16	1111111111010101
B/8	16	1111111111010110
B/9	16	1111111111010111
B/A	16	1111111111011000
C/1	10	1111111010
C/2	16	1111111111011001
C/3	16	1111111111011010
C/4	16	1111111111011011
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
C/5	16	1111111111011100
C/6	16	1111111111011101
C/7	16	1111111111011110
C/8	16	1111111111011111
C/9	16	1111111111100000
C/A	16	1111111111100001
D/1	11	11111111000
D/2	11	1111111111100010
D/3	16	1111111111100011
D/4	16	1111111111100100
D/5	16	1111111111100101
D/6	16	1111111111100110
D/7	16	1111111111100111
D/8	16	1111111111101000
D/9	16	1111111111101001
D/A	16	1111111111101010
E/1	16	1111111111101011
E/2	16	1111111111101100
E/3	16	1111111111101101
E/4	16	1111111111101110
E/5	16	1111111111101111
E/6	16	1111111111110000
E/7	16	1111111111110001
Continued on the next page.		

AC Luminance Huffman Coefficients		
Run/Size	Code Length	Code Word
E/8	16	1111111111110010
E/9	16	1111111111110011
E/A	16	1111111111110100
F/0 (ZRL)	11	1111111001
F/1	16	1111111111110101
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

Table 2.6: AC Luminance Huffman Coefficients

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
0/0 (EOB)	2	00
0/1	2	01
0/2	3	100
0/3	4	1010
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
0/4	5	11000
0/5	5	11001
0/6	6	111000
0/7	7	1111000
0/8	9	1111110100
0/9	10	1111110110
0/A	12	111111110100
1/1	4	1011
1/2	6	111001
1/3	8	11110110
1/4	9	111110101
1/5	11	11111110110
1/6	12	111111110101
1/7	16	1111111110001000
1/8	16	1111111110001001
1/9	16	1111111110001010
1/A	16	1111111110001011
2/1	5	111010
2/2	8	11110111
2/3	10	1111110111
2/4	12	111111110110
2/5	15	111111111000010
2/6	16	1111111110001100
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
2/7	16	1111111110001101
2/8	16	1111111110001110
2/9	16	1111111110001111
2/A	16	1111111110010000
3/1	5	11011
3/2	8	11111000
3/3	10	1111111000
3/4	12	111111110111
3/5	16	1111111110010001
3/6	16	1111111110010010
3/7	16	1111111110010011
3/8	16	1111111110010100
3/9	16	1111111110010101
3/A	16	1111111110010110
4/1	6	111010
4/2	9	1111110110
4/3	16	1111111110010111
4/4	16	1111111110011000
4/5	16	1111111110011001
4/6	16	1111111110011010
4/7	16	1111111110011011
4/8	16	1111111110011100
4/9	16	1111111110011101
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
4/A	16	1111111110011110
5/1	6	111011
5/2	10	1111111001
5/3	16	1111111110011111
5/4	16	1111111110100000
5/5	16	1111111110100001
5/6	16	1111111110100010
5/7	16	1111111110100011
5/8	16	1111111110100100
5/9	16	1111111110100101
5/A	16	1111111110100110
6/1	7	1111001
6/2	11	11111110111
6/3	16	1111111110100111
6/4	16	1111111110101000
6/5	16	1111111110101001
6/6	16	1111111110101010
6/7	16	1111111110101011
6/8	16	1111111110101100
6/9	16	1111111110101101
6/A	16	1111111110101110
7/1	7	1111010
7/2	11	11111111000
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
7/3	16	1111111110101111
7/4	16	1111111110110000
7/5	16	1111111110110001
7/6	16	1111111110110010
7/7	16	1111111110110011
7/8	16	1111111110110100
7/9	16	1111111110110101
7/A	16	1111111110110110
8/1	8	11111001
8/2	16	1111111110110111
8/3	16	1111111110111000
8/4	16	1111111110111001
8/5	16	1111111110111010
8/6	16	1111111110111011
8/7	16	1111111110111100
8/8	16	1111111110111101
8/9	16	1111111110111110
8/A	16	1111111110111111
9/1	9	111110111
9/2	16	1111111111000000
9/3	16	1111111111000001
9/4	16	1111111111000010
9/5	16	1111111111000011
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
9/6	16	1111111111000100
9/7	16	1111111111000101
9/8	16	1111111111000110
9/9	16	1111111111000111
9/A	16	1111111111001000
A/1	9	111111000
A/2	16	1111111111001001
A/3	16	1111111111001010
A/4	16	1111111111001011
A/5	16	1111111111001100
A/6	16	1111111111001101
A/7	16	1111111111001110
A/8	16	1111111111001111
A/9	16	1111111111010000
A/A	16	1111111111010001
B/1	9	111111001
B/2	16	1111111111010010
B/3	16	1111111111010011
B/4	16	1111111111010100
B/5	16	1111111111010101
B/6	16	1111111111010110
B/7	16	1111111111010111
B/8	16	1111111111011000
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
B/9	16	1111111111011001
B/A	16	1111111111011010
C/1	9	111111010
C/2	16	1111111111011011
C/3	16	1111111111011100
C/4	16	1111111111011101
C/5	16	1111111111011110
C/6	16	1111111111011111
C/7	16	1111111111100000
C/8	16	1111111111100001
C/9	16	1111111111100010
C/A	16	1111111111100011
D/1	11	11111111001
D/2	16	1111111111100100
D/3	16	1111111111100101
D/4	16	1111111111100110
D/5	16	1111111111100111
D/6	16	1111111111101000
D/7	16	1111111111101001
D/8	16	1111111111101010
D/9	16	1111111111101011
D/A	16	1111111111101100
E/1	14	11111111100000
Continued on the next page.		

AC Chrominance Huffman Coefficients		
Run/Size	Code Length	Code Word
E/2	16	1111111111101101
E/3	16	1111111111101110
E/4	16	1111111111101111
E/5	16	1111111111110000
E/6	16	1111111111110001
E/7	16	1111111111110010
E/8	16	1111111111110011
E/9	16	1111111111110100
E/A	16	1111111111110101
F/0 (ZRL)	10	1111111010
F/1	15	111111111000011
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

Table 2.7: AC Chrominance Huffman Coefficients

The Huffman tables shown in figures 2.4, 2.6, 2.5, and 2.7, define the tables prescribed by Annex K of the JPEG specification given by the International

Telecommunications Union. The tables defined prescribe the DC luminance Huffman table, AC luminance Huffman table, DC chrominance Huffman table, and the AC chrominance Huffman table, respectively.

2.8.5 Start of Scan

The scan header segment, shown in figure 2.7, defines parameters for the entropy coded data segment. The parameters define which quantization table to use, which Huffman tables to use, and how many components are used in the scan. For Baseline JPEG compression, this header will appear three times, one for each of the image components, luminance, while the other two times occur for the chrominance components.

SOS Header - 2 Bytes

This marker, SOS, uniquely defines the start of the scan. The scan it refers to is the entropy coded data that is from the JPEG encoder. The start of scan marker is defined by the hex 0xFFDA.

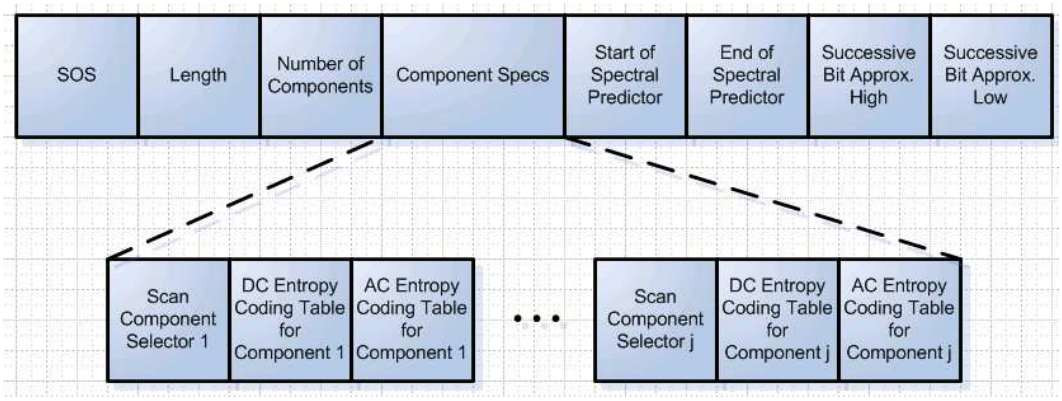

Figure 2.7: Start of Scan Segment

Length - 2 Bytes

As in other header segments, the length is two bytes which define how long the header segment is in bytes, including the 2 bytes used for specifying length, but not including the 2-byte marker SOS.

Number of Scan Components - 1 Bytes

This defines how many image components are contained within the current scan. This value is equal to the number of sets of scan specifications, which follow in this header segment. JPEG allows for component interleaving, in which Y, Cr, and Cb can be interleaved. For interleaving all 3 components, this byte would be set to 3. For single component scan, such as in Baseline JPEG, this value is set to 1.

Scan Component Selector - 1 Byte

This is the header for a specific component, which will be followed by that component's specified tables. This will match one of the components identified in the frame header.

DC Entropy Coding Table Selector - 4 Bits

This code will specify which of the DC entropy tables, specified in the define Huffman table segment are to be used by the decoder for the DC terms in this component.

AC Entropy Coding Table Selector - 4 Bits

This code will specify which of the AC entropy tables, specified in the define Huffman able segment are to be used by the decoder for the AC terms in this

component.

Start of Spectral or Predictor Selection - 1 Byte

This is used for other DCT based algorithms for selecting the first DCT coefficient to be coded in the scan. For the Baseline algorithm, all components, 0-63, of the DCT are encoded, so this value is set to 0.

End of Spectral Selection - 1 Byte

This as defines the last DCT coefficient in the block to be coded in the scan. For the Baseline algorithm, all components, 0-63, of the DCT are encoded, so this value is set to 63.

Successive Approximation Bit Position High - 4 Bits

This specifies the point transform used in the preceding scan. It is set to zero for the baseline algorithm. This is used for JPEG modes of operation such as progressive.

Successive Approximation Bit Position Low - 4 Bits

This specifies the point transform used before coding the band of coefficients specified by the spectral selection. It is set to zero for sequential DCT processes.

2.8.6 Entropy Coded Scan

Each entropy coded scan is preceded by a start of scan header segment. The scan will either be the luminance portion, or one of the two chrominance portions. Each entropy coded output of the encoder is variable length, composed of two parts. The two components are placed adjacent to one another in a stream. The

first portion of the data segment is the Huffman code which can be ranging in size from 2 bits to 16 bits. The second portion, which occurs after the Huffman code, is the magnitude value that comes from the run length encoder. The magnitude value ranges from 0 to 11 bits. So overall each of the combined variable coded output can be from 2 to 27 bits.

At any byte aligned boundary in the data stream, an occurrence of the 1 byte 0xFF can occur. As the 0xFF marker is specifically defined for a marker, the JPEG specification calls for zero stuffing in the following byte. This means, any occurrence of 0xFF in the entropy encoded data stream, will be followed by the byte 0x00. The decoder knows to ignore the 0xFF00, as this constitutes an illegal marker.

2.8.7 End Of Image

After the entropy encoded scan is complete. The end of image marker, EOI, specified by 0xFFD9, is appended. This marker is unique to the JPEG file format, and will be interpreted as end of image by the decoder.

3 System Overview

The system architecture for this project has been divided into three main areas, which can be seen in the high level block diagram in figure 3.1. The major portions of this design are easily divided into three sections, PC, DSP, and FPGA. The DSP and FPGA both reside on the PLogic PCI card, whereas the PC interfaces to the PLogic PCI card over the PCI bus. Although these sections are fairly separable, they interact heavily and are dependent on each other to fulfill the baseline sequential JPEG compression algorithm. The FPGA, is the heart, encoding input image data into a compressed output, used to create a JPEG image file. The DSP handles transactions between the PC and FPGA core, providing the interface to take raw pixel inputs from an input file, and command the FPGA to encode the input data. The DSP is also responsible to collect the

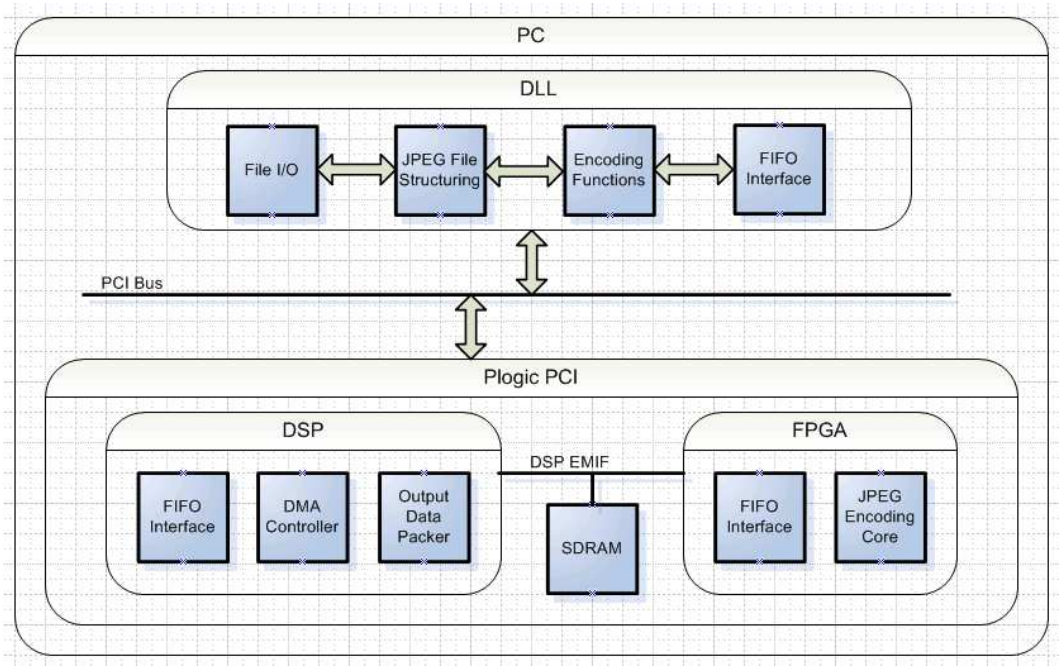

Figure 3.1: High Level System Overview

output and assemble an entropy encoded bit stream for use in the JPEG file structure. The PC is responsible for collecting input data, grabbing the output data stream, and assembling the output file.

3.1 FPGA Overview

The FPGA is composed of two main systems. First is the FIFO interface to the DSP. The FIFO interface is designed to allow for high speed data transactions in an asynchronous environment. Additionally an interrupt source state

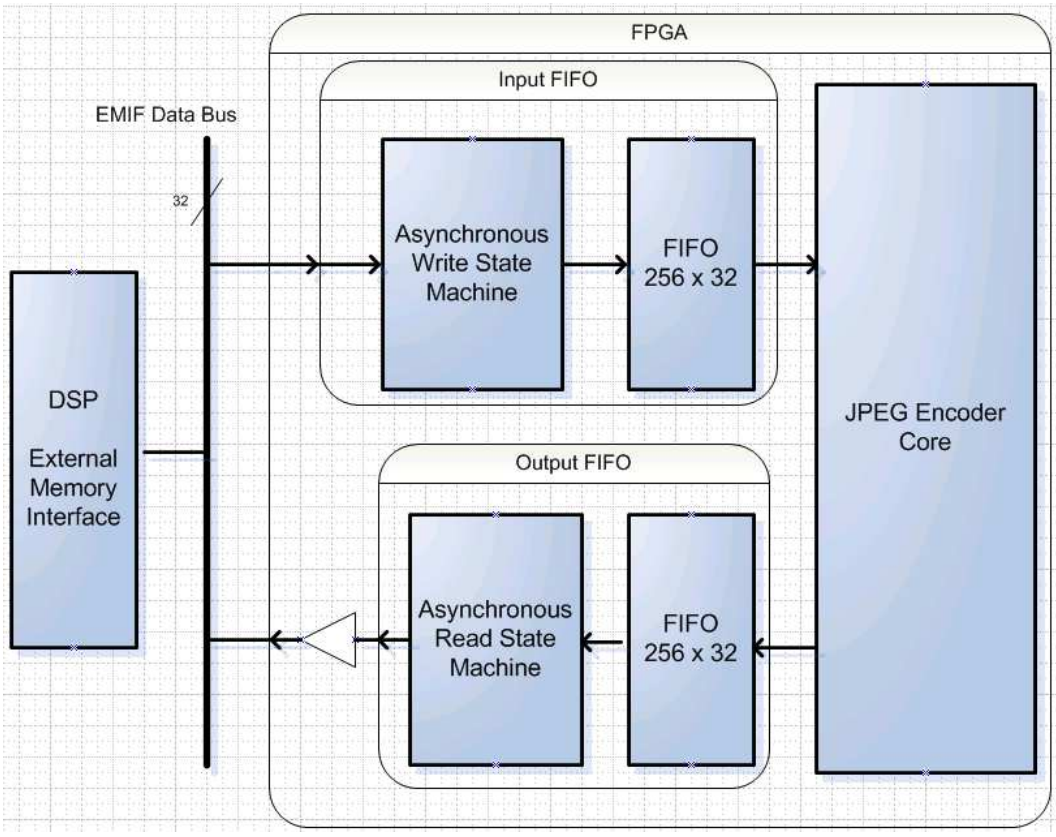

Figure 3.2: FPGA Core Overview

machine is associated with the FIFO interface to assist in a sort of handshaking between the DSP and FPGA. The second portion of the design is the JPEG encoding core. The JPEG encoder core runs on a 25 MHz clock and is fully synchronous, pipelined in stages, and employs parallel computation to help increase throughput.

3.1.1 Module Design

Each of the JPEG modules is designed with a common interface. The motivation behind this is to allow for dropping functional blocks into different codec designs.

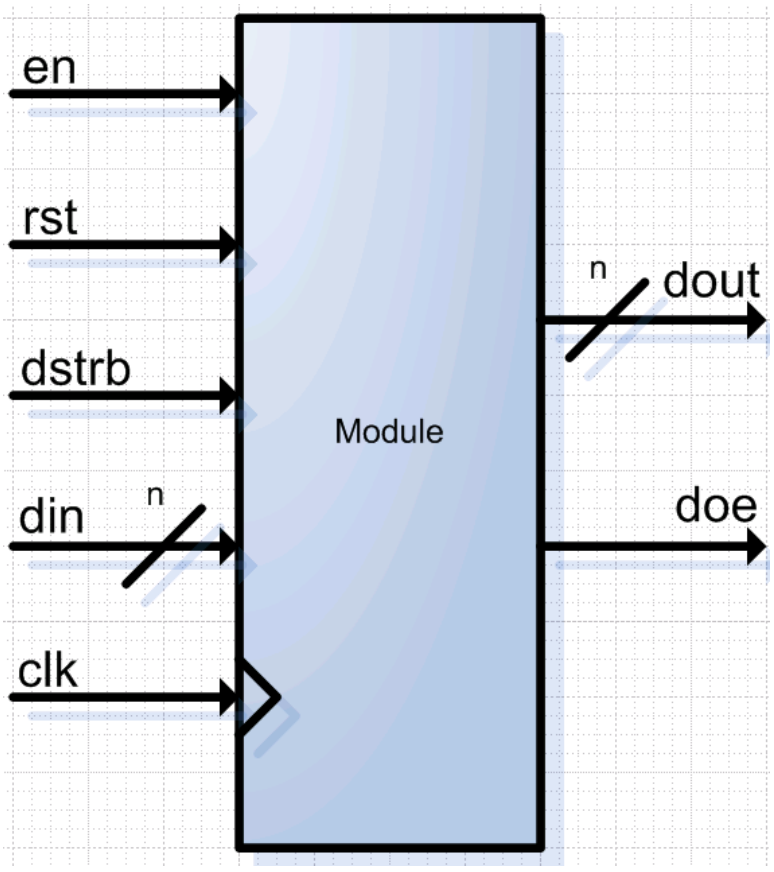

Figure 3.3: Module Design

Signal	Active Edge	In/Out	Description
en	High	Input	Global enable line to module.
rst	Low	Input	Asynchronous active low reset.
dstrb	High	Input	Data strobe active when data input.
din	High	Input	Data input.
clk	High	Input	Module clock.
doe	High	Output	Data output enable active at data output.
dout	High	Output	Data output.

Table 3.1: Module I/O

Additionally, it allows for ease of understanding how the blocks will communicate together. The interface consists of a clock, data strobe, data input, data output, and output data strobe, along with a global enable line. The enable line can allow the module to remain in its state regardless of clocks and inputs, but for this project, that effect is unused. The scheme consists of a clocked input which applies the data strobe on the same clock edge as the valid data input. By this scheme for streaming, or bursting, the data strobe is held high for every clock that there is valid data at the input of the module.

Figure 3.4 shows the timing for data input to any of the JPEG modules designed for this project. Enable (en) is an active high signal, data strobe (dstrb) is active high, and reset (rst) is active low. As figure 3.4 shows, enable is active, reset is inactive, and data strobe goes active in line with the positive edge of the clock (clk). On that same clock edge that the data strobe, the data input is active as well, allowing the module to latch the data.

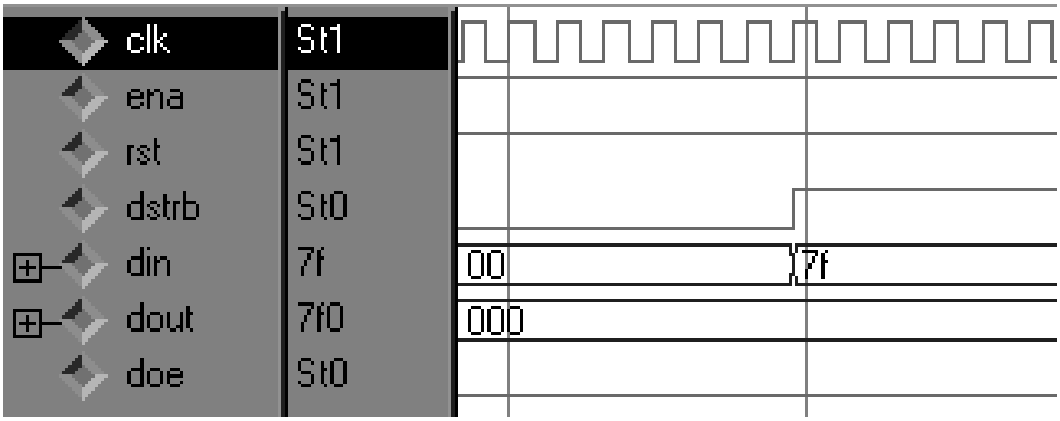

Figure 3.4: Module Input Timing

Figure 3.5 shows the data output of the module. The data output enable signal (doe) goes active on the same positive edge of the clock that valid data is output.

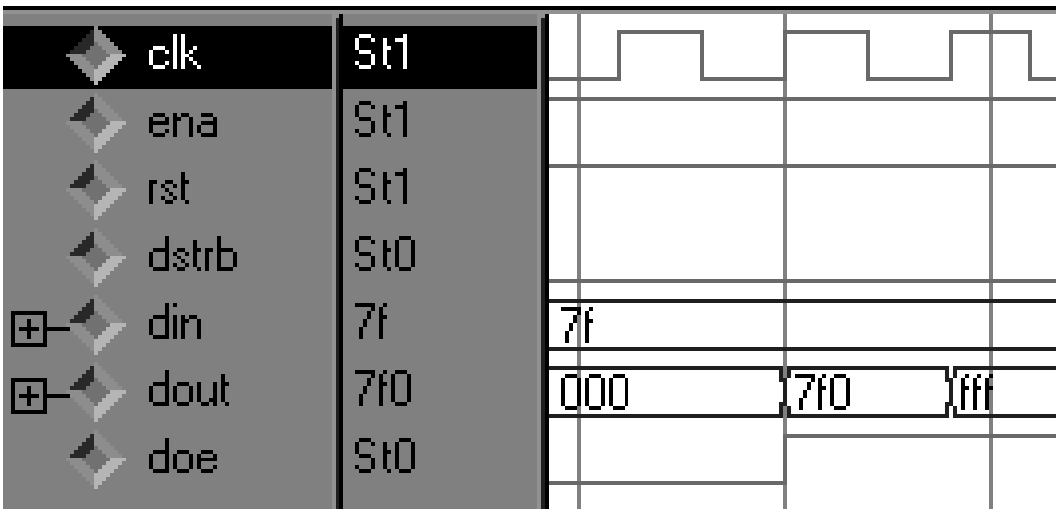

Figure 3.5: Module Output Timing

3.1.2 JPEG Encoder Core

The JPEG encoder core consists of four main modules which, although separable, are all necessary to work together to comply with the JPEG encoding specification. The units employed are for discrete cosine transform, quantization and rounding, differential pulse code modulation, and entropy coding. Some of these modules are pipelined and some are comprised of parallel data paths. Each module is fully synchronous, and are designed to have a common interface as described in section 3.1.1. At the output of the final stage of JPEG encoding, is a data stream assembler, which takes two variable length outputs from the entropy coder and combines them in a common format understood by the DSP.

The baseline JPEG compression algorithm requires an 8-bit pixel resolution for the data input. The other requirement is that the DCT based algorithm must produce an entropy coded output consisting of two variable length words. This data output must fit the JPEG file stream requirement. Quantization tables and Huffman tables can be specified within the JPEG file headers, and can be customized for better results based off of studies on a given data set. For this project, the tables used were prescribed by Annex K of the JPEG specification. The tables, stored as lookup tables that can easily be moved to block RAMs included within the Xilinx FPGA, and can be changed to fit certain applications. This exercise was left for future development.

Forward Discrete Cosine Transform

The forward discrete cosine transform module is designed with a high level of parallelism to allow high throughput transformation of 8x8 blocks of pixels. The DCT unit takes 8-bit inputs and can accept data in bursts or data streaming. This in fact allows for a degree of freedom in how this module is to be used.

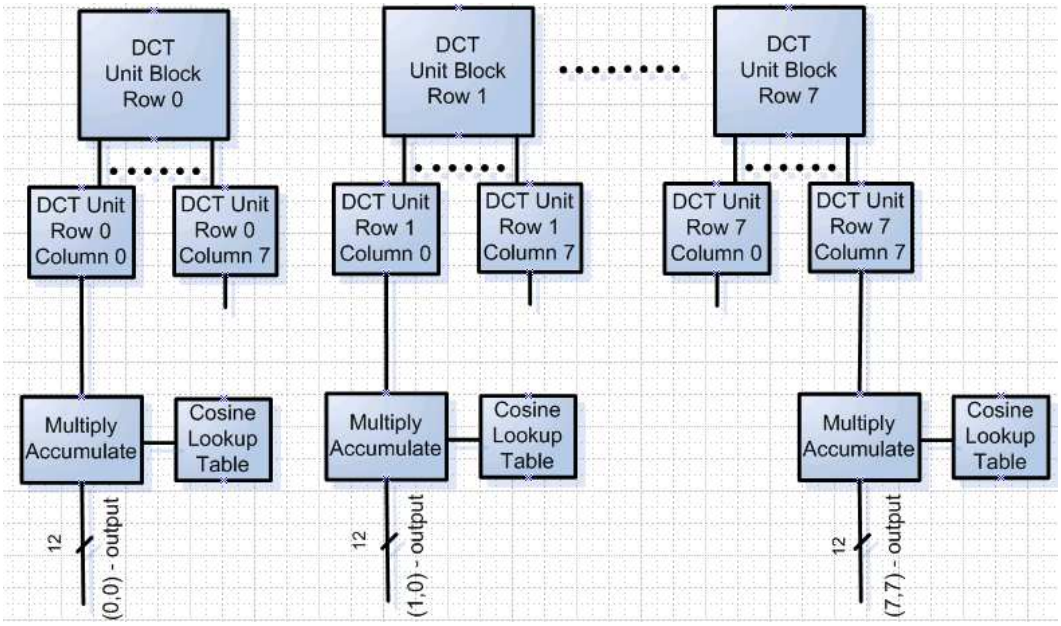

Figure 3.6: DCT High Level

The forward discrete cosine transform equation is shown in equation 3.1. An array of lookup tables were implemented such that the only unknown variable is the current pixel, f(x,y). There are 64 separate lookup table modules designed, each of which assume a fixed value for u and v, from (u,v) = (0,0) to (u,v) = (7,7). Each of the 64 lookup tables have 64 elements, one for each value of x and y, from (x,y) = (0,0) to (x,y) = (7,7). These lookup tables are all in parallel of one another, which helps increase the throughput of each block. Each lookup table is associated to a multiply-accumulate unit, which is used for computing the summation term of equation 3.1.

$$F(u,v) = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16} \quad (3.1)$$

$$C_u, C_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u, v = 0 \\ 1 & \text{otherwise} \end{cases}$$

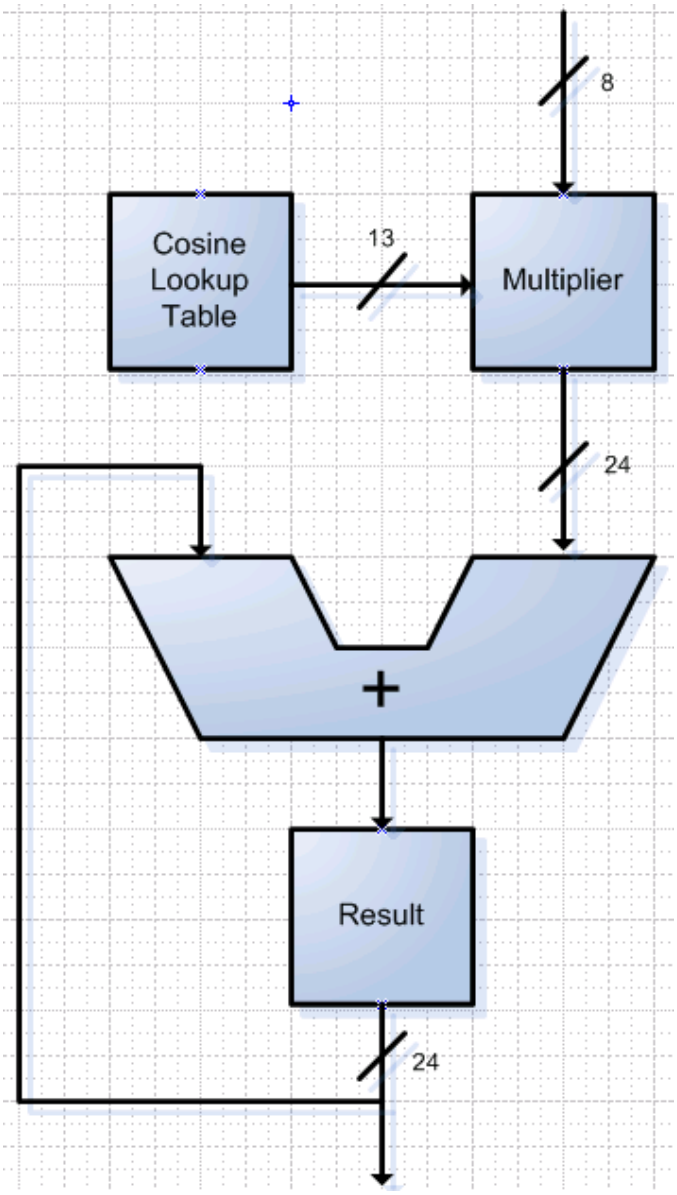

Figure 3.7: DCT Multiply Accumulate Module

There are 64 multiply accumulate modules working in parallel. At every data input the x and y count is incremented to pull out the appropriate cosine term for the multiplication. The data input is multiplied with the cosine term from the lookup table one clock cycle later. This design is shown in figure 3.7. The parallelism designed in the DCT unit allows for either bursty data inputs, or

data inputs serially input every clock cycle. After 64 data inputs, 64 outputs are produced in parallel.

Simulation Results

$$\text{Number of clocks} = 3 + 64 * n \tag{3.2}$$

Where n is the number of blocks.

The results of this module are quite impressive. As the DCT module overall has a high degree of parallelism, and can support streaming data inputs, a high throughput can be achieved. Assuming the best possible scenario of inputs streaming in constantly, a result of parallel outputs will be produced 67 clocks after the first data input of the 64 element block. As more data is streamed in, the next set of outputs is produced 64 clocks later. This scheme implies that parallel outputs are produced 64 clocks after the first element of the block is input, with an additional 3 clocks to fill the pipe on the first block, as described in equation 3.2. Due to rounding in the cosine tables, and the multiply-accumulate unit, the data outputs are off by at most 0.5 from the actual value. Fortunately this error is masked in the quantization unit.

Zigzag Unit

The zigzag unit takes 64 parallel inputs from the DCT unit, on a positive edge, active high data strobe signal. On this clock edge all 64 parallel inputs are stored into an array memory. Every subsequent clock cycle will output one of the values, in a zigzag order. The zigzag unit both serializes the DCT parallel output and reorders the data in a zigzag fashion, as shown in figure 3.8.

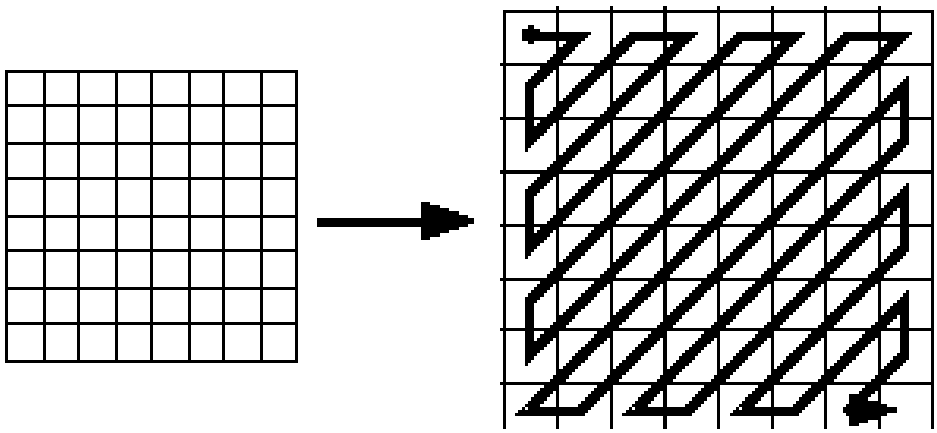

Figure 3.8: Zigzag Process

Quantization and Rounding

The quantization and rounding module will quantize each of the DCT zigzag coefficients, and round the result to the nearest whole number. This unit is pipelined with 12 stages, which supports 12-bit inputs, suitable for the DCT output. The division takes place by using a signed by unsigned non-restoring divider. The algorithm will initially use the dividend as the quotient, shift it and based on the sign, will add or subtract the divisor. This process is repeated to produce a quotient and remainder. The remainder is not important as the purpose of quantization is to scale the data element to be on a specific quantization level.

The signed by unsigned non-restoring divider, at the lowest level is an unsigned non-restoring divider. The module where this is instantiated has a shift register which stores the sign of the what the quotient will be. After the sign is applied to the quotient, the least significant bit is used to round the result. This is because the radix point of the result lies one bit left of the least significant bit. If the least significant bit is equal to one, the quotient is rounded up, and similarly rounded down if the least significant bit is zero. This architecture is depicted in figure 3.9.

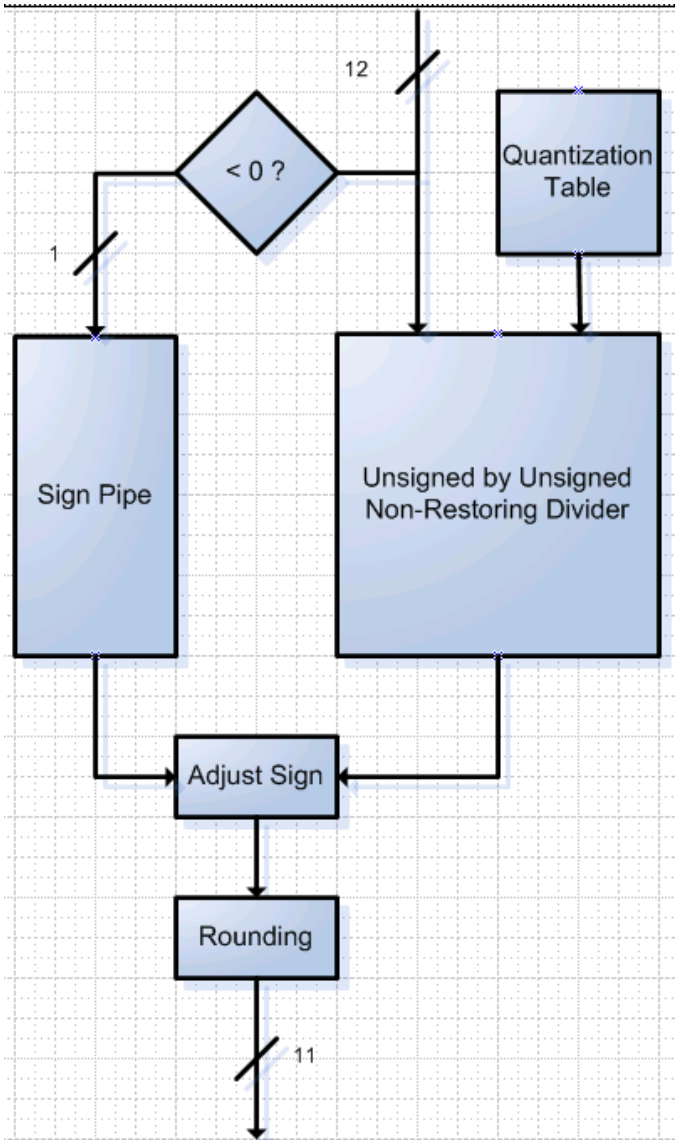

Figure 3.9: Quantization and Rounding

The design can accept data bursts or streaming data inputs. Once a valid data input is signaled by the data strobe signal, the twelve stage pipe will advance every clock until the output is produced. The data strobe signal is placed in its own pipe which also advances at every clock signal. The data output enable signal is produced at the end of the data strobe pipe. So when the quotient

is completed and output, the data output enable signal is also active. Using this design, streaming data can be accepted due to the pipeline, which advances every clock cycle. This allows a new quotient to be computed in each of the pipe stages. In the higher level module, the sign bit will align with the output of the unsigned divider, and apply the correct sign. This works well as the input to the module is from the DCT zigzag unit, which will output an element every clock.

In parallel to the quantization divide and round unit, are two quantization tables stored statically in a memory array. The two modules are for luminance and chrominance values, stored in a zigzag fashion, to correctly account for the DCT elements that had already been zigzagged. An additional input of one bit to the quantization module specifies whether to use data elements from the luminance or chrominance tables. The divide and round unit provides a count value, which specifies which value to lookup from the table and output. This value is used as the divisor in the division calculation.

The output of the quantization module is an 11-bit number, with the radix point occurring directly after the LSB. In contrary, the input 12-bit number from the DCT unit has the radix point before the LSB. This is due to the rounding that the unit does.

Results After each input, the output will be available 18 clock cycles later. The throughput of this pipelined design is taken advantage of in the higher level JPEG module. The DCT output produces 64 parallel outputs to the zigzag unit on one clock cycle. Each subsequent clock cycle, an element is output from the zigzag module, into the quantization unit. After the first element is input to the clock cycle, the first output of the quantization unit is 18 clock cycles later. Each clock after that gives another output from the pipeline, until the entire block of 64 elements is quantized.

Differential Pulse Code Modulation

The differential pulse code modulation, or DPCM, module is fairly straightforward. The module expects 64 sequential inputs, in either a streaming mode or bursty. The module will take the first element, the DC offset term, and subtract the DC term from the previous block from the new term. All AC values are passed through. A counter internal to the module ensures that the DC differencing only occurs on the first element input to the module. This was shown earlier in equation 2.5.

Entropy Coding

The entropy coding module consists of two interrelated modules, as shown in figure 3.10. The run length encoder and the Huffman encoder modules are packaged together as the outputs are variable length, and need to be output in phase with one another. The data input initially goes to the run length encoder, which is looking for runs of zeros in the stream of AC data elements. There are two outputs of the run length encoder, one of which gets assigned a Huffman codeword, and another which is assembled in line with the Huffman output.

The entropy coding unit is not predictable on when outputs will be produced. The best case is to have only a DC term, and every AC term of the block is 0. This would output only one piece of data from the run length encoder, which only requires one Huffman code. There would be two outputs for the entropy unit, one for the DC term, and then one for the AC run of zeros. Worst case there are no zero terms in the entire block. Although this is worst for compression, it is best for throughput. No zero components would result in the fastest computation, as it would not have to calculate runs of zeros. The worst possible input would result in a 74 clock cycles from the first input to the final output.

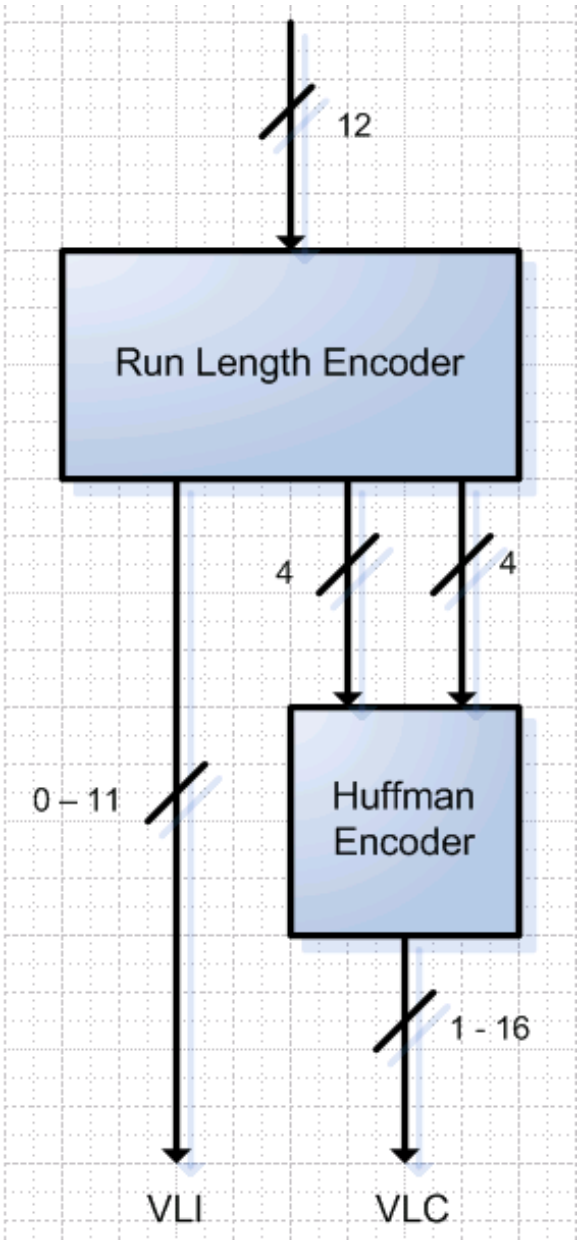

Figure 3.10: Entropy Encoder

Run Length Encoding The run length encoder is a fairly simple concept which looks for runs of zeros in the data stream. The run length encoder will output an amplitude value, a size category for the amplitude, and a run length of zeros. Every coefficient that equals zero, increments an internal counter, which

counts the run of zeros. Every nonzero coefficient input to the run length encoder will output a set of values. The run length value output specifies the number of zeros which occurred in the data stream before the nonzero value.

The run length encoder design consists of two modules, one which calculates the amplitude, size category, and an intermediate run length value, which may be passed on to the other module. The size category and run length are each 4 bits, and the amplitude output is 12 bits. The amplitude value is based on the data input. If positive, the MSB is cut off, and stored as the amplitude output. If the data input is negative, 1023 is added and stored as the amplitude output. The size category is based on the absolute value of the data input. The size category specifies how many bits from the LSB, are required to reproduce the original data input. It is a confusing process, but simply put the size value will specify the number of bits required to represent the original value. The amplitude is a way to determine which of the values within the size category is the original data element.

Quantization and zigzag preparation makes the data stream exceptionally well fit for run length encoding. The data toward the end of the stream contains less imporant AC coefficients which are heavily quantized and likely to contain runs of zeros. As the long run of zeros at the end of the stream will likely exceed 16, which is the maximum value represented by the 4 bit run length output. This is where the second module is used, the run length zero block detector. If there are greater than sixteen zeros before the next nonzero element, the modules will output a (run length, size) = (16, 0), and then output the nonzero value with its size and run length (without the 16 previously specified). This module will correctly output the end of block stream, when the remaining components are zero with a (run length, size) = (0,0).

Huffman Encoding The run length and size values from the run length encoder are input to the Huffman encoder and assigned a Huffman codeword. The Huffman codewords are stored in four separate lookup tables, for AC and DC components of luminance and chrominance. The appropriate lookup table is selected based on inputs to the module. The Huffman module is arguably the most simple module in this entire JPEG design. It will output the Huffman codeword, 1 clock cycle after the input, the output will be produced.

Output Formatting

The two important outputs of the entropy coding modules are the Huffman codeword, and the run length amplitude output. The amplitude output is known as the variable length integer, VLI. The Huffman module output a variable length codeword, VLC. This combination of variable length components need to be placed adjacent to one another in the entropy coded output stream. The variable length codeword comes first, as it is required to find the size category to see how many bits that follow in the data stream are required to recreate the amplitude value. The 5 MSB of this output specify the size in bits of the entropy encoded data, both the VLI and VLC. The output formatting procedure takes only 1 clock cycle.

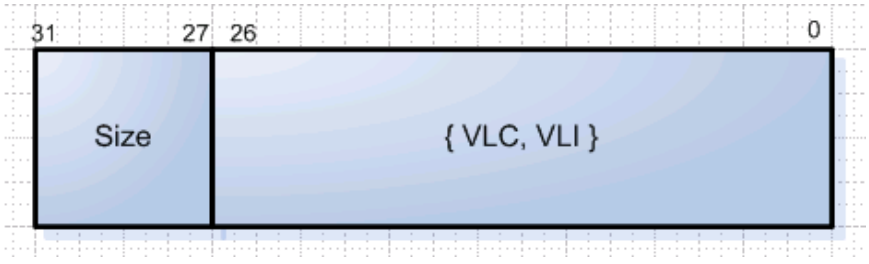

Figure 3.11: Output Format

3.1.3 Modular Addressing

As the JPEG design was modularized, there was a fair level of testability designed into it, as shown in figure 3.12. Through the control register, the input and output module can be selected. This was a very useful debugging tool. In fact, the entire module can be bypassed if a specific address is set in the control register.

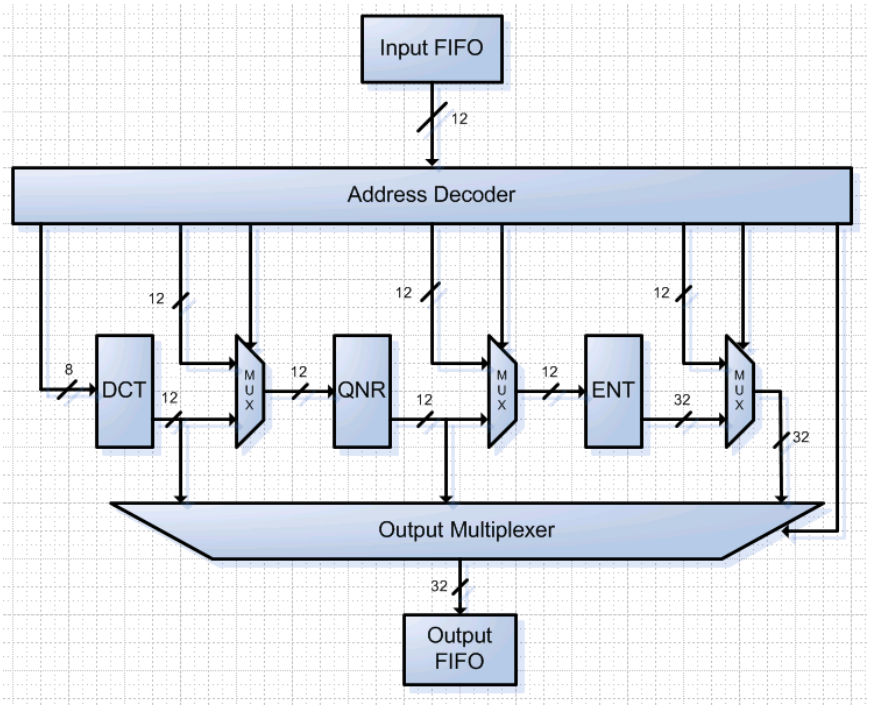

Figure 3.12: Modular Addressing

3.1.4 Status and Control Registers

For visibility and controllability into the design, status and control registers were added. The control register is a general purpose register that has numerous bits that are used for enabling and disabling certain functions. Status registers are used to represent the current state of the system.

Address	Input	Output
3'b000	FDCT	Entropy Encoder
3'b001	FDCT	FDCT
3'b010	FDCT	QNR
3'b011	QNR	QNR
3'b100	QNR	Entropy Encoder
3'b101	Entropy Encoder	Entropy Encoder
3'b110	INVALID	INVALID
3'b111	BYPASS	BYPASS

Table 3.2: Modular Addressing Table

Control Registers

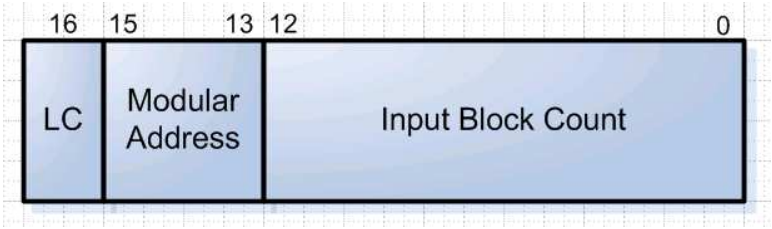

Figure 3.13: Encoder Control Register

Bit Position(s)	Description
16	Luminance(0) / Chrominance(1)
15:13	Modular Address
12:0	Total of blocks input

Table 3.3: Encoder Control Register

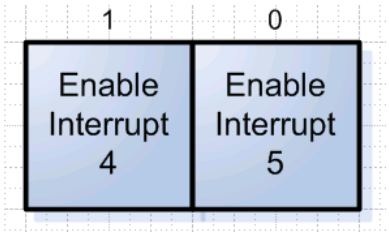

Figure 3.14: Interrupt Control Register

Bit Position	Description
1	Enable Interrupt 4 (Read Interrupt)
0	Enable Interrupt 5 (Write Interrupt)

Table 3.4: Interrupt Control Register

Status Registers

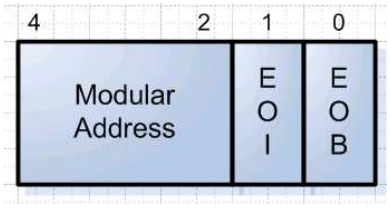

Figure 3.15: Encoder Status Register

Bit Position(s)	Description
4:2	Modular Address
1	Encoder End of Image
0	Encoder End of Block

Table 3.5: Encoder Status Register

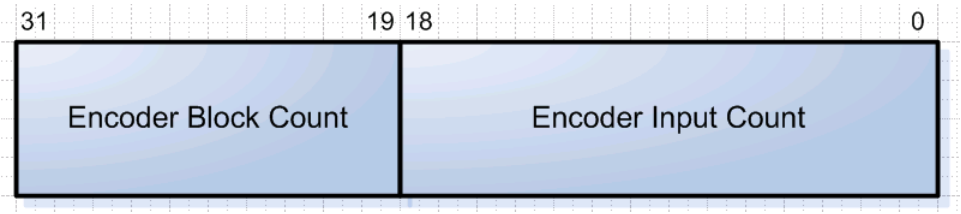

Figure 3.16: Encoder Count Status Register

Bit Position(s)	Description
31:19	Encoder Block Count
18:0	Encoder Input Count

Table 3.6: Encoder Count Status Register

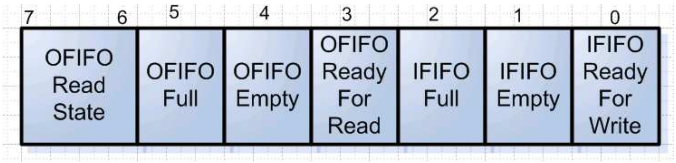

Figure 3.17: FIFO Status Register

Bit Position(s)	Description
7:6	Output FIFO Read State
5	Output FIFO Full
4	Output FIFO Empty
3	Output FIFO Ready for Read
2	Input FIFO Full
1	Input FIFO Empty
0	Input FIFO Ready for Write

Table 3.7: FIFO Status Register

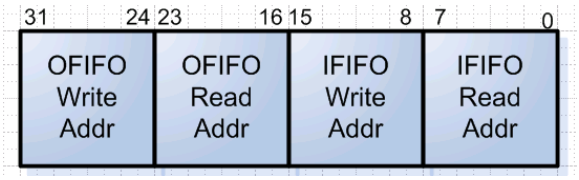

Figure 3.18: FIFO Address Register

Bit Position(s)	Description
31:24	Output FIFO Write Address
23:16	Output FIFO Read Addresses
15:8	Input FIFO Write Address
7:0	Input FIFO Read Address

Table 3.8: FIFO Address Register

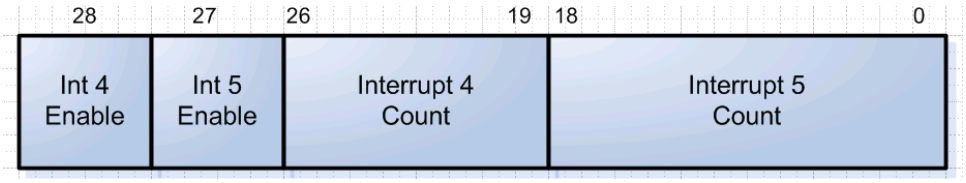

Figure 3.19: Interrupt Status Register

Bit Position(s)	Description
28	Interrupt 4 Enable
27	Interrupt 5 Enable
26:19	Interrupt 4 Count
18:0	Interrupt 5 Count

Table 3.9: Interrupt Status Register

3.1.5 FIFO Interface

The FIFO interface was designed for numerous reasons. First, the 100 MHz clock on the external memory interface of the DSP, is out of phase from the FPGA 25 MHz clock. This causes issues of metastability when latching data into registers. If a setup or hold time is violated, there is a potential for incorrect data being input to the encoder. The second reason for a FIFO interface, is to allow for streaming data as fast as possible into the encoder. As the FPGA clock is slower than the DSP EMIF clock, there is the ability to write faster than the FPGA JPEG encoder can accept data. Similarly, the DSP must read from the FPGA JPEG encoder, while servicing writes. Keeping the writes ahead of the reads can help performance by taking advantage of the pipelining in the design. Thus, the valid data may build up in the FIFO, so when reads commence they can be read in sequential clock cycles.

FIFO Structure

There are two FIFOs used in the design, one for writing to the JPEG encoder, and one for reading from the JPEG encoder. Each FIFO has the same structure, but each is wrapped in a different higher level module. The higher level module is designed to help deal with the asynchronous interface to the DSP, for reading from and writing to the FPGA.

The FIFO is composed of a 256x32 dual port Block RAM, built into the Xilinx Virtex FPGA, located on the PLogic PCI card. There are several block RAMs available for use, in configurable sizes, which can be instantiated by macros. The FIFO design has an address pointer module, which keeps track of the read and write addresses during transactions. These address pointers are used to produce empty and full flags.

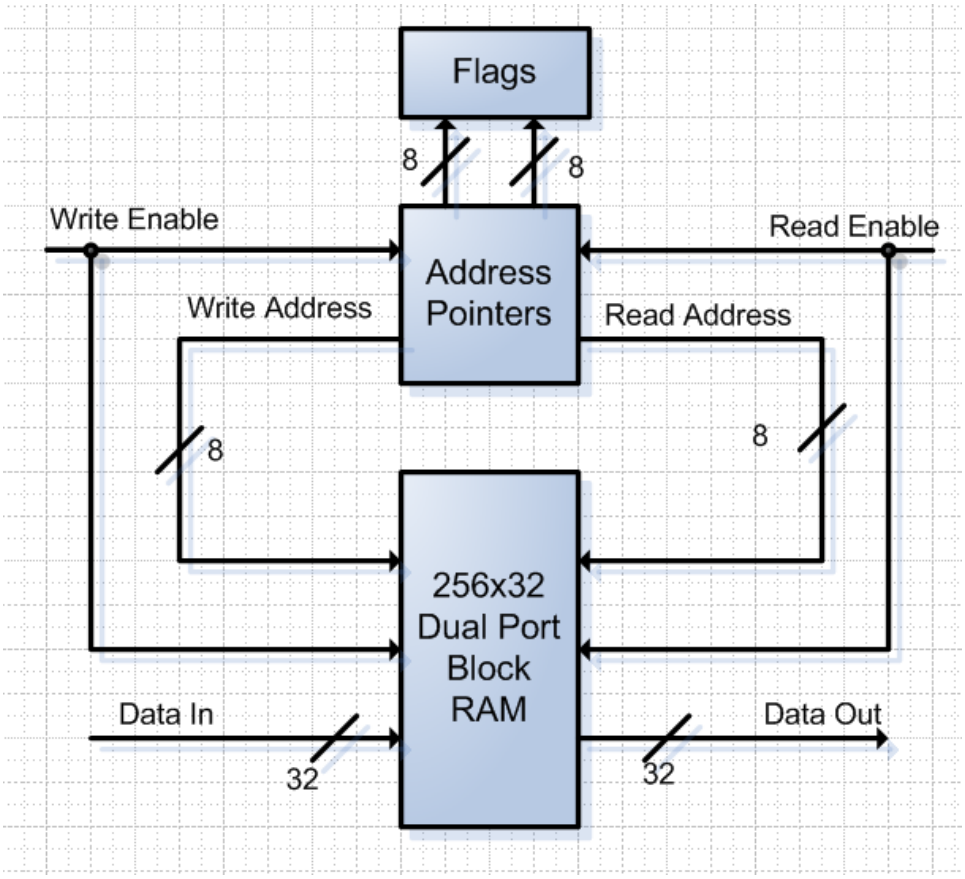

Figure 3.20: FIFO Structure

The address pointer module will increment the pointer for a write, after a write occurs. Similarly, the address will increment for reads, after a read cycle completes. These addresses are directly connected to the dual port block RAM. One port of this RAM is used for writing only, and the other is used for reading only. When the write pointer is at address 255, the last of the FIFO, another write will move the pointer back to address 0. Similarly for reads, this wraparound occurs.

The flags module easily can tell if the FIFO is full or empty. For the empty flag to be set active high, the addresses must be equal, otherwise, it is low, meaning the FIFO is not empty. Detection of the FIFO being full is a little

tricky as full would indicate the write pointer is 255 locations ahead of the read pointer. With the wraparound from location 255 to location 0, a full state could be achieved in two ways. First, if the read pointer is at address 0, and the write pointer is at location 255. If a wraparound has occurred, then the write pointer is potentially less than the read pointer. If it is less by 1, then we know that 255 levels are filled, thus requiring the full flag to be set active high. If 256 levels were written to, the address pointers would be equal, which the logic designed will show the FIFO is empty and not full. In order for ease of design, it was decided to make the FIFO 255 levels deep rather than 256, which would require more logic for determination of empty and full flags.

Input FIFO

The input FIFO is used for servicing writes to the JPEG encoder. As the DSP to FPGA interface is asynchronous, some care needs to be taken to ensure each write is correctly written to the FIFO. There are multiple ways errors can occur. First, there can be metastability in the latching of data caused by setup and hold times being violated. Asynchronously latching the data on the write enable signal allows the data to be latched without worry of any metastability cases. However, this data must be transferred to the synchronous portion of the design, before another data element is written.

At the same time the data is asynchronously latched, a bit is toggled. A synchronous register is in parallel with the asynchronous register. Along with the synchronous register, there is a synchronous toggle bit. Every edge of the FPGA clock, the toggle bits are compared. If the bits are not equal, then the data is transferred from the asynchronous register into the synchronous register. At this point the synchronous bit is then toggled. The data is then written to the FIFO on the next active edge of the FPGA clock.

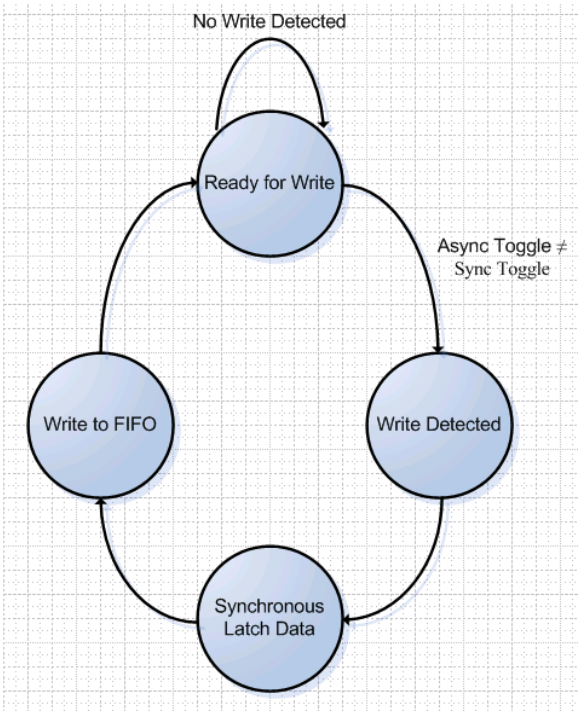

Figure 3.21: Asynchronous Write State Machine

Output FIFO

The output FIFO is used to store data outputs from the JPEG encoder, until they are read by the DSP. The reads are coming from an asynchronous interface and due to this, there must be care taken to not violate the timing parameters of the DSP. The read from the FIFO will take approximately one FPGA clock cycle, which is too long for the DSP timing to correctly adjust for. A read state machine was designed into the output FIFO to take care of this problem.

Every time the output FIFO is not empty, one data element is read and stored in a register, which can be accessed via the external memory bus. A synchronous toggle bit is associated with the read register. When the read from the DSP commences, an asynchronous toggle bit is inverted. At the next edge of the FPGA clock, the asynchronous toggle bit is compared to a synchronous

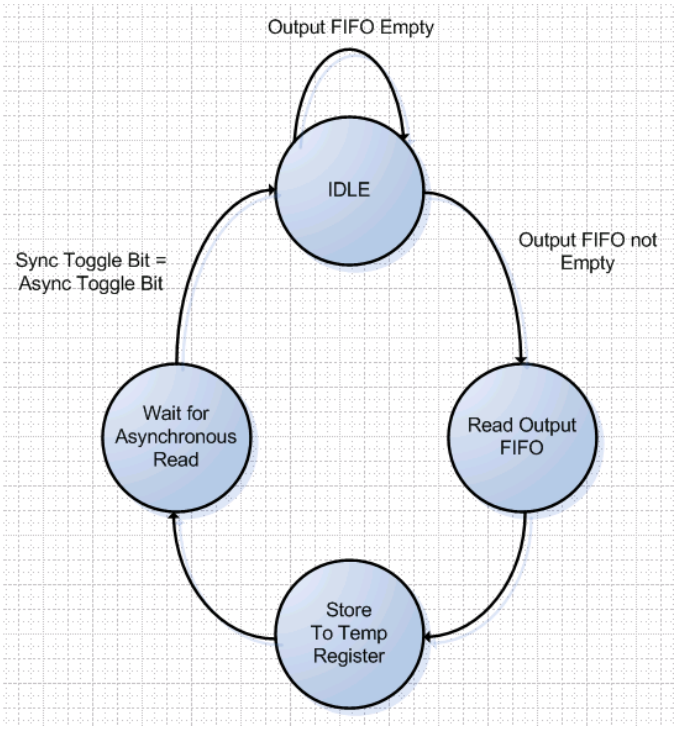

Figure 3.22: Asynchronous Read State Machine

toggle bit, and if equal, then a new element is read from the FIFO into the read register, as long as the FIFO is not empty.

3.1.6 Interrupt Driven Interface

In order to make best use of the FIFO interface, it would be important to get data to and from the FPGA as fast as possible. Of course, as fast as the asynchronous interface state machines on the input and output FIFOs will allow. The two ways explored in this design were using a programmed I/O method from the DSP, and the other is to utilize the DMA controller within the DSP. The DMA controller can be interrupt driven to transfer data to and from the DSP or other addressable locations.

The read state machine associated with the output FIFO produces a ready

for read signal, which when active states that another read can commence, and the current data has been read from the FIFO into the temporary read register. Similar to the ready for read signal, the write state machine associated with the input FIFO will generate a ready to write signal after the previous write is transferred into the FIFO.

Using a programmed I/O approach, the DSP will have to poll status registers in between reads and writes to make sure a valid read or write can safely occur. This process is very costly as after every read or write the processor must be involved. If using the DMA approach, with interrupts driving the DMA, the processor intervention can be avoided. There is an interrupt control register which can enable external interrupts 4 and 5, which correspond to the DMA read channel and DMA write channel respectively.

The two state machines associated with the input and output FIFOs are used to initiate interrupts. The input FIFO will issue a ready for write signal, which if interrupt 5 is enabled, will generate an interrupt on that channel. If the output FIFO ready for read signal is active, and interrupt 4 is enabled, an interrupt is generated on that channel.

The two external interrupts are assigned priorities. The read interrupt, is given the higher priority interrupt as there will be far less reads than writes. It is possible if writes were given higher priority, that the reads would never be serviced, and the FIFO would run the risk of being overfilled, which could result in lost data.

3.2 DSP Overview

The DSP program uses a real time operating system supported by the DSP called DSP BIOS. This realtime operating system has configurable interrupt channels with interrupt service routines, a configurable 4 channel DMA structure, as well as several other features that were not all used in this design. Some structures were defined to store status and control for the imaging functions that interact with the FPGA for configuration and for running the compression sequence.

3.2.1 DSP FIFO Server

The PC is able to communicate with the DSP over the PCI bus. There is a set of FIFOs on each side of the PCI bus which interact with each other for transferring commands and data between the PC and DSP. The PC can issue two commands via the FIFOs. The first command is called "put record", which allows the PC to place an specific code which defines an action, with arguments. The second command is called "get response", which attempts to read a response from the DSP via the other FIFO.

The "put record" commands have several valid action codes, some of which give a response, which the PC will get by issuing the "get response" command. There were numerous action and response codes used in this project, however, the ones in the tables listed are the only ones used for actual operation of the JPEG encoder design. Upon issuing an action code, there may be one to several arguments along with it. The arguments are used, for instance, to specify an element of a register to read or to modify.

Some action codes require a response code to be sent afterward to read the FIFO from the DSP to get the requested information. The response packets may

Action	Code	Description
ACT_RESET	1	Complete system reset, FPGA, and DSP structures.
ACT_WRCTRL	3	Write to image control structure, update FPGA control structure.
ACT_ENCODE	6	Run encode image function. Assemble FPGA output into data stream.
ACT_GETCTRL	8	Request read of image control structure.
ACT_GETSTAT	14	Request read of status registers.
ACT_TESTBLOCK	15	Run various test functions, used in development.
ACT_READFPGA	16	Read 1 word from FPGA, display on UART.
ACT_READADDR	18	Read FPGA FIFO addresses, display on UART.

Table 3.10: DSP FIFO Server Action Codes

include numerous data elements which could be an element from a control or status register, or data which gives information on the encoding process. If the response packet is not received, or an incorrect response code is received, an error is generated.

3.2.2 Action Codes

ACT_RESET

This action code is to complete a full system reset. This includes giving a hardware reset to the FPGA, re-initializing the DSP to PC FIFOs, clearing the image status structures, and initializing 512KBytes of SDRAM.

ACT_WRCTRL

This action code is used to change a single coefficient in the image control structure. There are two arguments given when this action code is issued. The first argument specifies the element within the structure to change. This is so a pointer to the structure can be adjusted, such that the write will occur to the correct element. The second argument is the data write into the control register element. The final thing this action will do is update the control register on the FPGA, with the new data written to the DSP control structure. This is followed up by updating the FPGA control register by calling the function update_creg().

ACT_ENCODE

This action code is to run the FPGA encode function, using DMA control, which is discussed later. The encode action has 3 arguments. The first and second provide the image aspect ratio in pixels. The third argument provides the image component type, either luminance or chrominance. These values are updated in the control structure, and then the control register on the FPGA. This action will first run a function called encodeImage(). A response is given to the PC with the number of 32-bit words in the encoded data stream.

ACT_GETCTRL

This action code will pass back the entire image control structure stored on the DSP. This is representative of what is on the FPGA control register, and what additional elements are used in configuring DSP portions of this design.

ACT_GETSTAT

This action will first read the FPGA status registers to update the DSP status structure, and then it will pass back the entire status structure to the PC via a response packet.

ACT_TESTBLOCK

This action code is general purpose used for development purposes only. It was used for debugging DMA transfers, using programmed I/O methods, and SDRAM tests.

ACT_READFPGA

This action will read one element from the FPGA address 0, which is the output FIFO address. The output will be displayed across the UART. This function is primarily used for debugging and development.

ACT_READADDR

This action is used to read the FIFO address pointers for both the input and output FIFO of the FPGA. The results are written across the UART. This action was used for development purposes.

3.2.3 Response Codes

There are only few response codes used in this project. Each are used for retrieving data regarding status, control, and

RSP_IMGSTRUCT

This response code contains the entire image control structure in it's packet.

Response	Code	Description
RSP_IMGSTRUCT	50	Response with image control structure.
RSP_GETSTAT	52	Response with status coefficient requested.
RSP_ENCODE	54	Response with number of 32-bit words in the assembled image stream.

Table 3.11: DSP FIFO Server Response Codes

RSP_GETSTAT

This response code contains the entire image status structure in its packet.

RSP_ENCODE

This response gives the number of 32-bit words to read from SDRAM, after encoding and the assembly of the image data stream.

3.2.4 Control Structures

The image control structure consists of 7 elements all of which will get set in the FPGA control register, upon running a function called update_creg(). Only five of these elements were actually implemented in the FPGA. There is an entry for quality factor, which is meant for scaling the quantization tables to finer or more coarse values. This was not implemented in the FPGA, but can be added in the future. The other element unused was a clock divide enable, which would instruct the FPGA clock DLL to use a slower division of the FPGA 25MHz clock. This ideally was for slowing down things to debug stages, but this proved unnecessary, even during development.

The elements used, are pixel aspect ratio in x, pixel aspect ratio in y, the image

type (luminance or chrominance), and two for selecting the module addresses for input and output of the JPEG encoder.

3.2.5 Status Structures

The status structure contains 19 elements. These are all updated upon the ACT_GETSTAT action code being issued by the PC. Also, various programmed I/O functions that were used for debugging, call this function. The elements in this structure are listed in the table 3.12

3.2.6 External Memory Interface

The DSP external memory interface is where both the FPGA and SDRAM are located. This memory interface shares a common bus amongst all the elements. Each element is byte addressable across the memory bus. This means a single byte can be written to if desired, but this has repercussions as the data bus is 32-bits wide. What is done for this project is to start at an address, and for each successive write, the address is incremented by 4.

FPGA

The FPGA is located on the CE1 memory space of the DSP external memory interface. The address is location 0x01400000, which translates to address 0 on the FPGA. Address 1 on the FPGA would be 0x01400004, and so on. The address space extends from 0x01400000 onto 0x017FFFFF.

SDRAM

The SDRAM is located at address 0x02000000 for CE2, and another SDRAM is located at address 0x03000000 for CE3. The CE3 memory space is unused for

Element	Description
0	Module address for testability in JPEG encoder. Mirrors control register.
1	End of image, specified by JPEG encoder.
2	End of block, specified by JPEG encoder.
3	Number of blocks output of encoder.
4	Number of elements input to encoder.
5	Output FIFO read state machine state.
6	Output FIFO full flag.
7	Output FIFO empty flag.
8	Output FIFO ready for read.
9	Input FIFO full flag.
10	Input FIFO empty flag.
11	Input FIFO ready for write flag.
12	Output FIFO write address.
13	Output FIFO read address.
14	Input FIFO write address.
15	Input FIFO read address.
16	Interrupt enable flags for read and writes.
17	Number of interrupts generated on channel 4.
18	Number of interrupts generated on channel 5.

Table 3.12: DSP Status Structure Elements

the JPEG project. The CE2 address space from 0x02000000 to 0x02FFFFFF, is split in two, for source image pixel data, and for the encoded data stream. From location 0x02000000 to 0x023FFFFF, the source data can be stored. Currently, storing one 8-bit pixel per 32-bit location in SDRAM, there is still plenty of room. Similarly, from location 0x02400000 to 0x02FFFFFF, the output data stream is stored. The output data stream is continuous data across the 32-bit locations. This provides more than enough room for a large output data stream.

3.2.7 Programmed I/O Interface

The programmed I/O interface functions were programmed mainly for debugging and development purposes. However, there are a few functions that are called from the FIFO server, and are still very useful in the actual mode of operation. Programmed I/O methods are in fact slower than the DMA implementation for reasons such as utilizing the processor rather than the DMA unit which is separate, and having to run by polling status registers, rather than interrupt driven.

Function - void write0_sdram()

This function is intended to clear out SDRAM. It was implemented early on in development to ensure that only valid data from the current encoding procedure is stored in SDRAM. This is done at system resets to ensure that SDRAM is cleared prior to encoding processes.

Function - void write_fpga(int addr, int data)

This function has two arguments given, for an address and for data to write. The address supplied is for which location on the FPGA the data is to be written.

Function	Description
write0_sdram()	Clear 512Kbytes of SDRAM
write_fpga()	Write a single 32-bit value to an FPGA address.
read_fpga()	Read a single 32-bit value from an FPGA address.
write_mult_fpga()	Write an array to an FPGA address.
read_mult_fpga()	Read multiple 32-bit words from the FPGA.
write_src()	Write a single 32-bit element to SDRAM addressed from 0x02000000
read_src()	Read a single 32-bit element from SDRAM addresses starting at 0x02000000.
write_mult_src()	Write multiple 32-bit words to the SDRAM addresses offset from 0x02000000.
read_mult_src()	Read multiple 32-bit words to SDRAM address offset from 0x02000000.
write_enc()	Write 1 32-bit word to an FPGA address.
read_enc()	Read 1 32-bit word from an FPGA address.
write_mult_enc()	Write multiple 32-bit words to FPGA.
read_mult_enc()	Read multiple 32-bit words from FPGA.
update_status()	Update DSP status structure, by reading FPGA status registers.
update_creg()	Update FPGA control registers from DSP control structure.
assemble_image()	Assemble data stream from variable length encoder outputs
encode_image()	Encode image using DMA architecture.

Table 3.13: DSP Programmed I/O Functions

Function - void read_fpga(int addr, int *data)

This function has two arguments supplied, one for an address to read from, and the other is a pointer to where to store the data. The address given is assumed to be offset from the FPGA base address 0x01400000.

Function - void write_mult_fpga(int addr, int *data, int length)

This function has 3 arguments given. One is an address to write every value to. The second argument is a pointer to the array of data to be written. The final argument is how many data elements to write. Simply this function runs a loop calling write_fpga() a number of times specified by the length argument.

Function - void read_mult_fpga(int addr, int *data, int length)

This function has 3 arguments given. The first argument is the address from which to read the values from the FPGA. This address is assumed to be offset from the FPGA address 0x01400000. The second argument given is a pointer to the array on which to write the data read from the FPGA. The third argument is the number of data elements to read. This function simply runs the read_fpga() function in a loop a number of times specified by the third argument given, length.

Function - void write_src(int addr, int data)

This function has two arguments provided. The first is the address to write to in the first half of SDRAM CE2 space, offset from 0x02000000. The second argument is the data to write to that location.

Function - void read_src(int addr, int *data)

This function has two arguments given. The first is the address to read from in SDRAM CE2 space, offset from 0x02000000. The second argument is a pointer to the location where the data read will be stored.

Function - void write_mult_src(int addr, int *data, int length)

This function has three arguments provided. The first argument is the starting address to write, offset from 0x02000000, SDRAM CE2 space. Each incremental write increases to the next 32-bit location. The second argument given is a pointer to the array of data to write into SDRAM. This function calls write_src in a loop, which will run a number of times specified by the third argument length.

Function - void read_mult_src(int addr, int *data, int length)

This function has three arguments given. The first argument is the starting address to write, offset from 0x02000000, SDRAM CE2 space. Each subsequent read in this function will be incremented to the next 32-bit location in SDRAM. The second argument is a pointer to the array of data to write to these locations. This function will run a loop calling the read_src() function a number of times specified by the third argument length.

Function - void write_enc(int addr, int data)

This function will write a 32-bit value to the second half of SDRAM CE2, offset from address location 0x02400000. The first argument given is the address to write to offset from that location. The second argument is the data to write.

Function - void read_enc(int addr, int *data)

This function is meant to read from the second portion of SDRAM CE2, offset from the address location 0x02400000. The first argument given is the address to read from, offset from that location. The second argument is a point to the data to store the value read.

Function - write_mult_enc(int addr, int *data, int length)

This function is to write multiple values to the second half of SDRAM CE2, offset from address 0x02400000. The first argument given in the starting address to write to. Each subsequent write will increment by 4 bytes. The second argument is a pointer the array of data to write. This function will run the write_enc() function in a loop a number of time specified by the third argument length.

Function - read_mult_enc(int addr, int *data, int length)

This function is to read multiple data elements from the second half of SDRAM CE2, offset from address 0x02400000. The first argument given is the starting address to begin reading. Each subsequent read will increment by 4-bytes from the previous read address. The second argument is a pointer to the data array to write to. This function runs the read_enc() function in a loop a number of times specified by the third argument length.

Function - void update_status()

This function will poll the FPGA status registers to determine the current state of the system. The status registers read are the encoder status register at location 1, the encoder counts register at location 2, the FIFO flags register at address 3, the FIFO address pointer register at address 4, and the interrupt status register

at register 5. These reads will update the DSP status structure described in section 3.12.

Function - void update_creg()

This function will use the DSP control register to update the FPGA control register. The modular address, image type, and image block count values are updated on the FPGA.

Function - int assemble_image()

The purpose of this function is to assemble an image stream into a sequence of 32-bit words stored into SDRAM CE2. Returned from this function is the number of 32-bit words in SDRAM that were used for this image stream. The function will go through each variable length encoded data segment from the JPEG encoder output, and find the size of the data output specified by the 5 most significant bits. The size will determine how many bits to shift over a temporary 32-bit word which is storing the stream. Once the size exceeds the 32-bit boundary of the temporary integer, that 32-bit integer is written to SDRAM, the address is incremented, and the rest of the entropy encoded data not yet in the bitstream is appended to the next temporary integer being prepared to write. This function executes in a loop based on a value from the DMA read structure. This function will not work for programmed I/O methods for this reason. The value in the DMA structure is the final address written to in SDRAM. This value is used to determine how many reads should be done, until the entire stream is assembled.

Function - int encode_image()

This function will enable the two interrupts on the FPGA for driving the DMA unit transfers. A semaphore is then posted to wait until the encoding is complete. Once the DMA channel interrupt service routine posts a semaphore, signifying completion of encoding and transferring data to SDRAM, the assemble_image() function is run. A time stamp is collected before interrupts are enabled, and after the semaphore is posted. This value is returned, and signifies the number of high resolution clocks it took to encode the image component.

3.2.8 DMA Interface

The DSP has a DMA controller in parallel with its core. The DMA unit can operate on the external memory interface bus, to transfer data elements to and from the DSP internal memory, or between external memory devices. Each of the four DMA channels have nine level FIFOs associated to buffer memory transfers. These channels can be configured to operate on a wide number of interrupts. This works well for this design, allowing the external memory interrupts driven by the FPGA to trigger memory transfers between the DSP and FPGA. Each of these DMA channels labeled zero through three. These channels operate where channel zero is highest priority for transfers, and channel three is lowest priority.

The DMA channels can be configured to increment the source and or destination address, within its configuration registers. These addresses can be configured to increment based upon an element size for 8-bits to 32-bits. Otherwise the address can be configured to remain static.

The DMA unit also has a transfer count register which keeps track of the number of elements, and frames of elements transferred. The transfer register gives 16-bits for frames, and 16-bits for elements. Once the element count reaches

0, the frame count is decremented, and the element count is reloaded by one of the global reload registers, which specifies how many elements are within a frame.

Each individual DMA channel can be setup to transfer an element or a frame on a synchronizing event. These elements can be an interrupt generated by another DMA channel, an external interrupt, a timer interrupt, as well as other sources. This is particularly helpful in transferring elements based on interrupts generated on the external interrupt line from the FPGA. These synchronization events can be tied to the DMA transfer operating between the FPGA and SDRAM.

The DMA channels can also be configured to generate an interrupt based upon different events. For instance, the last transfer in a frame, defined in the transfer counter register, can generate a DMA channel interrupt. There can be an interrupt generated on the completion of the second to last frame. The ability to use different conditions to trigger interrupts gives a higher level of configurability for these DMA channels.

The general approach taken for this DMA strategy is to give the reads from the FPGA higher priority than writes, as writes will happen much more frequently, and we do not want to risk overfilling the output FIFO. The interrupt state machines associated with the input and output FIFOs generate external interrupts that trigger the DMA channels for data transfers.

DMA Channel 1 - Write FPGA

DMA channel 1 is configured to take care of writing data stored in SDRAM, into the FPGA. As described before, the source data is stored in the first half of the CE2 memory space, address locations 0x02000000 - 0x023FFFFF. This DMA channel is configured to transfer 32-bit elements, incrementing the source

address, but not changing the destination address. Unfortunately, there are two set backs to the DMA implementation, inherent to the PLogic PCI card design. As the design has an imperfect interface with two asynchronous clocks, we need to be sure of each element being transferred successfully. This facilitated the design of the read and write state machines on the FPGA. By this notion, the write DMA channel is configured to only transfer 1 element at every interrupt trigger. In the DMA channel interrupt service routine, it will calculate how many elements are left to transfer, and as long as that does not equal zero, it will load the transfer register with the value to make one element transfer. This channel is triggered off of external interrupt 5, generated by the FPGA input FIFO write state machine. Upon a trigger, the DMA channel will transfer one element from SDRAM to the FPGA input FIFO.

DMA Channel 0 - Read FPGA

DMA channel 0 is configured to take care of reads from the FPGA output FIFO. The reads are stored to the second half of the CE2 address space from 0x02400000 to 0x02FFFFFF. The DMA channel is configured to transfer 32-bit elements, incremenenting the destination address, but not the source address. The DMA channel is set to transfer a maximum of 65,535 (0xFFFF) 32-bit elements, but additional work could allow for a greater number to be handled. This was determined as acceptable as 640x480 images tested, used significantly less transfers. The channel is higher priority than the write channel, and is triggered to transfer off of a higher priority interrupt, external interrupt 4, produced by the FPGA output FIFO read state machine.

3.3 PC Overview

The PC level of this design can be separated into three sections, all of which interact with each other. The lowest level section interacts with the DSP via a FIFO interface, as well as direct addressing to DSP registers, and the DSP external memory interface. The level above this deals with constructing the output JPEG image file, and similarly the input file. The highest level is a TCL interface to the C libraries. TCL was chosen for its ability to create a simple command driven, or graphical interface through buttons, and response windows.

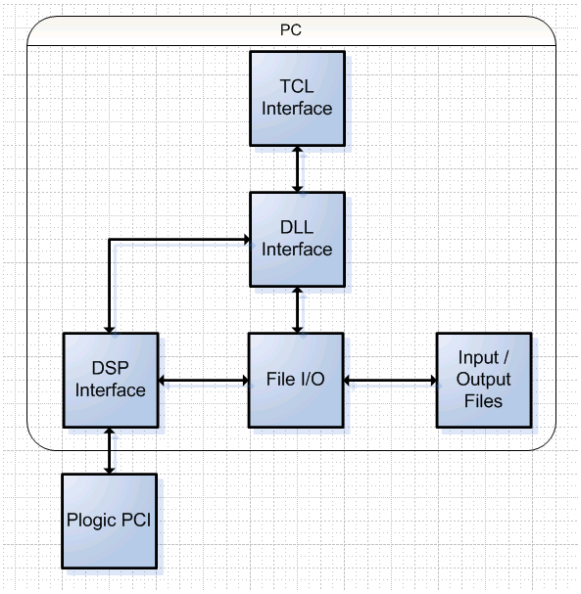

Figure 3.23: PC Overview

3.3.1 PLogic Interface

The PLogic FIFO interface functions mirror the FIFO interface on the DSP. The PC is meant to issue a "put record" command which send the action codes from table 3.10, which will be interpreted by the DSP FIFO server. The PC will then issue a get response command and wait for the DSP to send a response packet.

In addition to the FIFO interface, there are low level DSP related functions for accessing DSP registers and the DSP external memory interface. These functions all operate through the DSP memory map. Through functions read_DSPreg(), write_DSPreg, read_DSPblock, and write_DSPblock(). These functions are useful in determining status of the DMA channels, such as checking the transfer register, and the source and destination registers. The block transfer functions are useful for storing source image data into SDRAM, and reading the encoded variable length data, or the assembled encoded image bitstream from SDRAM.

3.3.2 Imaging

There were several functions implemented to handle the understanding of the input image, sending it to the encoder, calling the encode process, retrieving the output, assembling the JPEG headers, and producing the output file. All of these functions were fairly simple to put together as they are high level functions within this design. The tricky part really has to do with the JPEG headers. Getting the JPEG headers correct is an important part of the design, as the entropy coded bitstream could be correct, but incorrect headers will result in an erroneous image output stream read by a decoder.

Image Input

The first step for the PC interface is to open a known type of input file, parse it, and send it to SDRAM. The supported input image format for this design was chosen to be the PPM format, which stands for portable pixel map. This is a fairly easy to understand format. There are four parameters which describe the file type. The first parameter is either P5 or P6, which defines a grayscale (luminance only), or a color image, respectively. The second number defines the

pixel resolution. The values used for baseline JPEG compression in this design are 8-bit, so the value in the input file here is 255, to represent 256 values starting from 0. The last two values specify the horizontal and vertical pixel resolutions. The values for horizonal and vertical pixel ratios are stored to the DSP control register, through a "put record" command.

The grayscale images, denoted by the P5, will have one value per pixel, to specify the luminance intensity, where 0 is white, and 255 is black. If this is a P6 image, for color, there are three values used; one for luminance, and the other two for chrominance. When parsing the input file, a temporary output file is created to store a list of hex pixel values. There will be one created for luminance in both cases of color and grayscale images. For color images, there are also temporary files created for the two chrominance components. This was necessary as the three components are each encoded in their own steps.

Encoding Process

For color images the encoding procedure is run three times. This is to handle each of the input components separately, rather than being interleaved. The encoding process will read 64 pixel components in a loop, from one of the files created during the parsing operation discussed in section 3.3.2. Using one of the block transfer functions, these values are stored to SDRAM. This process goes on until the file is read completely, and transferred into SDRAM. At this point, the DSP control register is updated with the pixel aspect ratios, and the image component type, luminance or chrominance.

Prior to running the encoder on the FPGA the image headers are assembled. Just as described in section 2.8, the headers are stored to an output file. After the header portion is completed, the actual encoding cycle is run, and the output

bitstream is stored into the output file.

The actual encode cycle on the FPGA is run once the "put record" command is given with the action code to encode the image. This is followed with a "get response" command, which will wait for a response from the DSP. The DSP response contains an integer specifying how many words must be read from SDRAM to obtain the assembled entropy encoded data stream. This integer will be used to run a loop and read SDRAM in blocks of 64 elements, until completed. After this data is written to the output file, the end of image marker is written and the file is closed.

3.3.3 TCL Interface

The entire PC level design was wrapped in a DLL which could be interpreted through a TCL command line input. This was chosen as a suitable scripting language as it is fairly easy to mock up a development interface. Buttons were easily configured to run low level PC functions, or even communicate directly

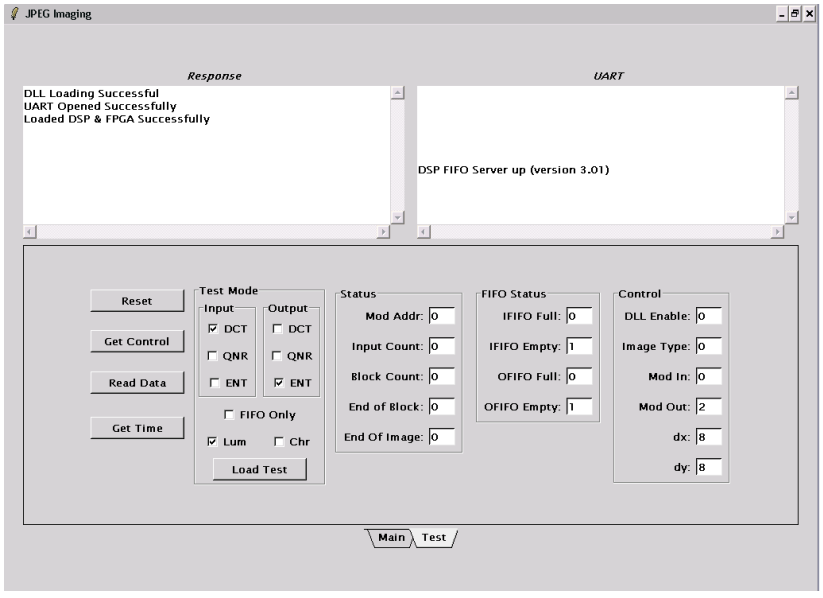

Figure 3.24: TCL Interface

to the DSP, or FPGA. The DSP has a UART feedback channel, which was interpreted in TCL and allowed for helpful low level feedback from the DSP regarding state, or data elements in small transactions. This interface provided a means to test via modular addressing, as well as feedback of the control and status registers.

In the end, a simple script was created, to call the encode image function, assuming that the input PPM image file was in the appropriate place. The output file is generated in a known location as well. This file could be decoded using any image viewer on a computer that can open JPEG files, such as an internet browser.

4 Discussion

The core FPGA design had impressive results as described throughout the system overview sections. In simulation on the FPGA, assuming we had streaming inputs and outputs, the encoder takes 162 cycles for the first block, and as long as the data stream is constant, it will take an additional 64 cycles per block after the first. This means that a 640x480 grayscale image would take 307,298 cycles, which at 25MHz is 12.29ms. This is roughly one fourth of the time that is necessary to make motion JPEG at 24 frames per second. As this core design is fully synchronous, it should give the same result in implementation.

However, that is not the complete story. Unfortunately, some limitations inherent to the PLogic PCI design do not provide an interface for fully streaming inputs and outputs. The DSP external memory interface, which facilitates data transfers between the FPGA and DSP, is operating on a different clock than the FPGA core. In order to ensure reliable data transfer the interrupt driven DMA controller was used for transferring data to and from the FPGA FIFO interface. At the same time, on this bus, the DSP must service the SDRAM refresh rates and such.

Blocks	FPGA Time	DSP Time
1	98.84 us	99.4 us
2	193.64 us	194.76 us
4	383.72 us	385.78 us
8	755.96 us	768 us
16	1535.24 us	1535.94us

Table 4.1: Block Encoding Results

Table 4.1 above shows some statistics taken for actual encoding time from input to output of the FPGA. The FPGA reports the number of clocks it took from the first output to an end of block or end of image signal depending on what is asked for. The DSP uses its high resolution timers to collect a sample of when the encode process started, and when it completed. These times imply that the pipelining in the design is not being taken advantage of. It also shows that when more reads are requested from the FPGA, through interrupts, the writes are likely staggered such that the data input is not streaming. This is another design flaw where the output FIFO is only used as a single level, or dual level FIFO at best.

Aside from the disappointments in speed at the lower level, there were also deficiencies designed in at the higher levels. At the PC level, parsing the input and creating the output files, although trivial, added to the completion time. Also, being only able to read 64 elements at a time, and write 64 elements at a time to SDRAM definitely added time. It seems as if there is greater time taken in setting up, and creating an output, than the actual encoding process.

The results of this design in terms of compression were quite impressive. Three different test images were used to get a good range of image types from those that would expect less compression and those that would expect greater compression based upon the image details. It can be seen that around high frequency components that artifacts will occur.

4.1 Future Work

In the process of developing this project, there were several ideas that were never integrated into the design. As this is the JPEG baseline algorithm, there are

Image	Aspect Ratio	Input(Bytes)	Output (Bytes)	Compression
baboon	200x200	120,015	9,411	12.75
lena	512x512	786,448	31,262	25.16
peppers	512x512	786,448	28,033	28.05

Table 4.2: Compression Results

many extensions. The building blocks used in this design can be utilized in the JPEG extensions, as well as other compression algorithms. Also, there are many optimizations which can be performed on this design to improve throughput.

Interleaving Image Components Currently, the way this implementation runs is to encode the image components sequentially. This can potentially lower the time required to encode a color image.

Sub-sampling Chrominance Components The chrominance components are currently encoded at a 1:1:1 ratio. This can be modified to fit a different compression scheme such as 4:2:2. This would further improve compression, with potentially minimal loss in quality.

Custom Tables By studying a known data set, quantization tables can be optimized to further compress the data in the image. These tables could be loaded through the DSP rather than being static as they are now. This could improve compression, and would only require some design changes on the FPGA, and the supporting software interface on the DSP and PC. Similarly, the Huffman tables could be optimized for a given data set, based on analysis of the source.

Quality Factor The quantization tables can be scaled according to a quality factor selected by the user. This has a direct impact on compression ratios. This will allow a user to set the desired level of compression for a given application.

JPEG Extensions and Other Algorithms These components used for the baseline JPEG algorithm can be applied to other types of compression sequences. The JPEG extensions such as hierarchical mode, and progressive mode, can use some blocks from this encoder to fit those specifications. Similarly other compression algorithms such as JPEG2000 use similar techniques, but in the case of JPEG2000 the DCT is replaced by a discrete wavelet transform.

FPGA Interface Optimization Work needs to be done to optimize the FIFO interface to the encoder. This interface must be improved to help achieve higher throughput, as right now, the current interrupt driven DMA transfers are sub-optimal, as seen from the experimental results in table 4.1.

Motion JPEG Motion JPEG is arguably the most interesting application to this project. Simulation results show that the throughput of the FPGA encoder core is fast enough to achieve motion JPEG required speeds. This project would likely require a daughter circuit board to mate to the I/O of the FPGA pinned out on the PLogic PCI card. This daughter board would likely have a camera, and memory, which could allow for multiple data paths in sending data to the encoder, and retrieving data to store into memory.

Figure 4.1: Image: Baboon Source

Figure 4.2: Image: Baboon Result

Figure 4.3: Image: Lena Source

Figure 4.4: Image: Lena Result

Figure 4.5: Image: Peppers Source

Figure 4.6: Image: Peppers Result

Bibliography

[1] Iain Richardson. *Video Codec Design - Developing Image and Video Compression Systems*. Copyright by John Wiley & Sons Ltd. 2002 (ISBN 0471485535)

[2] Peter Symes. *Video Compression Demystefied*. Copyright by The McGraw-Hill Companies, Inc. 2001 (ISBN 0071363246)

[3] Gonzalez Woods. *Digital Image Processing*. Copyright by Prentice Hall, Inc. 2002 (ISBN 0201508036)

[4] John Watkinson. *The Art of Digital Video*. Copyright by John Watkinson, 2000 (ISBN 0240512871)

[5] Effelsberg Steinmetz. *Video Compression Techniques*. Copyright by dpunkt—Verlag fur digitale Technologie GmbH, 1998 (ISBN 3920993136)

[6] Weidong Kou. *Digital Image Compression - Algorithms and Standards*. Copyright by Kluwer Academic Publishers, 1995 (ISBN 079239626X)

[7] Netraveali Haskell. *Digital Pictures - Representation, Compression and Standards*. Copyright by ATT Bell Laboratories, 1995 (ISBN 030644917X)

[8] Anil K. Jain. *Fundamental of Digital Image Processing*. Copyright by Prentice-Hall, 1989 (ISBN 0133361659)

[9] Pennebaker and Mitchell. *JPEG Still Image Data Compression Standard*. Copyright by Van Nostrand Reinhold, 1993 (ISBN 0442012721)

[10] International Telecommunication Union. *Information Technology - Digital Compression and Coding of Continuous-Tone Still Image - Requirements and Guidelines*. Copyright by ITU, 1993 (iso10918-1)