Exploring the Implementation of JPEG Compression on FPGA

Ann Malsha De Silva, Donald G. Bailey, Amal Punchihewa
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
annmalsha@hotmail.com, D.G.Bailey@massey.ac.nz, G.A.Punchihewa@massey.ac.nz

Abstract—**This paper presents the implementation of the JPEG compression on a field programmable gate array as the data are streamed from the camera. The goal was to minimise the logic resources of the FPGA and the latency at each stage of compression. The modules of these architectures are fully pipelined to enable continuous operation on streamed data. The designed architectures are detailed in this paper and they were described in Handel-C. The compliance of each JPEG module was validated using MATLAB. The resulting JPEG compressor has a latency of 8 rows of image readout plus 154 clock cycles.**

Keywords-Image Compression, JPEG; FPGA; Handel-C; DCT; Zigzag; Quantization; Huffman Coding

I. Introduction

In image processing, image compression can improve the performance of the system by reducing the cost and time in image storage and transmission without a significant reduction of the image quality. A monochrome image can be defined over a matrix of picture elements (pixels), with each pixel represented by an 8-bit gray scale value. This representation of image data could demand large storage and bandwidth to transmit. The purpose of image compression is to reduce the size of the representation and, at the same time to keep most of the information contained in the original image [1]. Image compression can be lossy or lossless. Lossy compression gives a greater reduction in data volume compared to lossless compression; however only an approximation to the original image can be reconstructed.

There are several standards for image compression and decompression (CODEC) such as Joint Photographic Experts Group (JPEG) [2], JPEG2000 [3], Graphic Interchange Format (GIF) [4], Portable Network Graphics (PNG), Tagged Image File Format (TIFF). JPEG compression is the most widely used form of lossy image compression and is based on the discrete cosine transform (DCT). A compressed image in JPEG format can be approximately 10% of the original size depending on the information contained within the image and compression quality. This results in a 90% decrease in the needed bandwidth [5]. Image and video codecs are implemented mainly in software as signal processors can manage the operations although the computational overhead is quite large. These operations can also be efficiently implemented in hardware [6].

Field programmable gate arrays (FPGAs) are a relatively new technology, which combines the properties of the traditional hardware and software alternatives. It can provide speed, performance and flexibility since it implements a parallel and pipelined version of the algorithm [7]. The latest FPGAs have millions of reconfigurable gates, capable of running at clock speeds of hundreds of megahertz (MHz) and are therefore well-suited for graphics and image processing. FPGA based designs generally comprise a large number of simple processors which all work in parallel and may compete for memory access or other resources [8].

II. Motivation

Processing time and power restrictions imposed on dedicated embedded systems make software compression unviable in many applications. Power efficiency and fast compression are often performance critical factors. For most digital image codecs, increasing the compression has been achieved at the cost of increasing the complexity of the techniques and implementations. These restrictions usually mandate a dedicated hardware implementation of a JPEG compressor, especially in applications such as digital cameras, DVD players, traffic controllers, secure ticketing, and many more. As the JPEG compression process is complex, its design in hardware is demanding [9].

FPGAs are well suited for many embedded systems applications because they have several desirable attributes such as, small size, low power consumption, a large number of I/O ports, and a large number of computational logic blocks [10]. Images have a high degree of spatial parallelism, thus image processing applications are ideally suited to implementation on FPGAs which contain large arrays of parallel logic and registers and can support pipelined algorithms [8].

JPEG is an international standard for still-image compression and it has been widely used since 1987 [2]. This research is concerned about the implementation of real time JPEG compression for gray-scale images on to FPGA.

There are many research papers published in conference proceedings and journal papers about JPEG compression using FPGAs (for example [6, 9, 11, 12]). Unfortunately, as a result of page limits and space constraints, many of these papers give the results of the implementation of various sections of the JPEG compression, but present relatively few design details.

978-1-4673-2393-2/12/$31.00 ©2012 IEEE

Some researchers only focus on implementing either 2D-DCT or Huffman coding onto FPGAs [1, 13-16].

This research focuses on the real-time implementation of JPEG compression on an FPGA device as the data are streamed from a camera while minimising

- the logic resources of the FPGA, and
- the latency at each stage of the JPEG compression.

The basic structure of the JPEG compression algorithm is shown in Fig. 1; the development of design architectures for each module is described in later sections. The modules of these architectures are fully pipelined and targeted to FPGA implementation using Handel-C. Each module was evaluated using MATLAB to develop the test-bench.

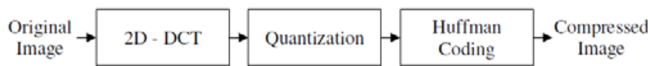

Figure 1. Basic architecture of JPEG compression

III. Methodology

First, a fixed point JPEG algorithm has been developed in MATLAB. This serves two purposes. First, it identifies the minimum number of bits required to represent each stage within the FPGA without introducing significant error. This is important, because an FPGA implementation is not restricted to work with the standard 8, 16, or 32 bit word lengths used by software. The speed can be increased, and resources required can be reduced by reducing the number of bits. Second, it also provides ground truth data for bench-testing the resulting FPGA algorithm.

Then most of the effort has gone into mapping the algorithm into a form suitable for FPGA implementation. The aim was to make the implementation as efficient as possible. The resulting algorithm was implemented using the hardware description language, Handel-C [17]. Two separate software suites were used during the hardware implementation: Mentor Graphics' DK design suite, and Altera's Quartus II design suite. Each stage of the algorithm was validated through simulation by comparing the results with the MATLAB results. Finally the implementation was targeted to a Cyclone 4 FPGA on an Altera DE2-115 development board.

IV. Implementation of Discrete Cosine Transform

A. Discrete Cosine Transform

The DCT is the basis for the JPEG compression standard. This was first introduced in 1974 by Ahmed et al. [18]. The primary purpose of image transformation within an image coding context is to concentrate the energy into as few components as possible [10]. This enables efficient compression by quantizing smaller elements to zero.

The DCT transforms the image from a spatial domain representation to a frequency domain representation. This has the advantage that not every dimension has the same importance for the visual quality of the image. Visually less significant components can be quantized more heavily without compromising the visual quality.

For JPEG, the image is segmented into non-overlapping 8×8 blocks, with a 2D-DCT algorithm applied to each block. Since the 2D-DCT is separable, it can be split into a series of 1D-DCTs on the rows, and then on the columns.

$$D = MXM^{\mathrm{T}} \tag{1}$$

where X is the 8×8 data block and M is the matrix of DCT coefficients.

Exploiting separability, there are numerous ways to efficiently implement the 2D-DCT as a series of 1D-DCTs in both software or hardware [19].

B. Transformation from Streamed to Blocked Data

JPEG uses block processing to maintain locality of the data to give good compression (nearby pixel values are highly correlated). As the data are streamed from the camera whole rows at a time, it is necessary to buffer 8 rows of the image before an 8×8 block of data is available for processing.

Dual-port RAM blocks on the FPGA sufficient to hold 16 image rows are used. One port is used to write the values being streaming from the camera, while the second port is used to read the values in block order. Once the first 8 rows have been written to the row buffers, a signal indicates data availability for the DCT. The data in these 8 rows are read and processed in block order while data continues to stream into the next 8 rows. The two sets of 8 rows effectively form a ping-pong buffer with data being written to one half and read from the other.

C. Proposed DCT Architecture for 1D-DCT

There are several fast parallel algorithms for implementing the DCT (see for example [16, 20-22]). The most efficient algorithm is that of Loeffler et al. [16], which requires 11 multiplications and 22 additions. The number of multiplications can be reduced to 5 if a scaled DCT is acceptable [12] (the scale factors can be compensated by the quantization stage at no extra cost).

The major limitation of these methods is the resulting architecture. They sacrifice regularity to reduce the number of multipliers. That is generally not a good trade-off in FPGA design, where highly irregular architectures generally do not translate into efficient implementations, primarily because of increased routing cost. A balance between the number of multipliers and quality of architecture is essential for efficient implementation.

Modern FPGAs have plentiful multipliers. This enables a simpler and elegant pipelined design to be implemented. It is based on a first order factorization by Woods et al. [23]. This reduces the number of multiplications for each frequency sample from eight to four by calculating the even and odd frequencies separately:

$$\begin{bmatrix} F[0] \\ F[2] \\ F[4] \\ F[6] \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 \\ c_2 & c_6 & -c_6 & -c_2 \\ c_4 & -c_4 & -c_4 & c_4 \\ c_6 & -c_2 & c_2 & -c_6 \end{bmatrix} \begin{bmatrix} f[0]+f[7] \\ f[1]+f[6] \\ f[2]+f[5] \\ f[3]+f[4] \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} F[1] \\ F[3] \\ F[5] \\ F[7] \end{bmatrix} = \begin{bmatrix} c_1 & c_3 & c_5 & c_7 \\ c_3 & -c_7 & -c_1 & -c_5 \\ c_5 & -c_1 & c_7 & c_3 \\ c_7 & -c_5 & c_3 & -c_1 \end{bmatrix} \begin{bmatrix} f[0]-f[7] \\ f[1]-f[6] \\ f[2]-f[5] \\ f[3]-f[4] \end{bmatrix} \quad (3)$$

where $c_k = \frac{1}{2}\cos\frac{k\pi}{16}$.

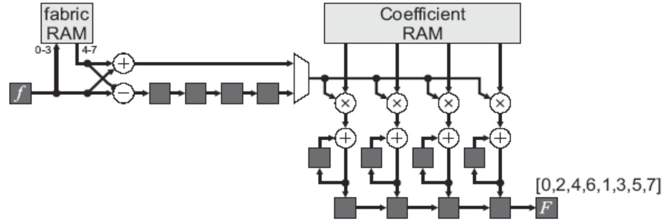

Figure 2. Block diagram of proposed DCT architecture [10]

Our implementation of DCT is shown in Fig. 2. The whole process is pipelined to operate on streamed input data at one pixel per clock cycle. The memory at the start holds the first 4 samples, and returns them in reverse order to calculate the sum and differences in (2) and (3). Each multiply and accumulate unit is reset every four clock cycles, and calculates a separate output frequency. First, four clock cycles calculate the even frequencies using (2), and then while the results are being streamed out, the odd frequencies are calculated.

Fixed point arithmetic is used to simplify the multiplier logic. Scaling the coefficients by a power of 2 makes all of the operations integer.

$$c'_k = \text{round}\left(c_k 2^B\right) \quad (4)$$

The number of bits output from the multiply and accumulate is reduced by truncating unneeded bits. Initialising the accumulator with an appropriate value converts the truncation to rounding.

In this research, pipelining is used to begin a new DCT before the previous DCT is completed. The multiply and accumulate units are then utilized with every clock cycle. The outputs are not in natural order; for JPEG compression, this does not matter because the data will be reordered later using a zigzag buffer.

D. DCT Word Length Optimisation

During the design stage, the data width of each term is chosen individually to minimize the logic usage while keeping the truncation error within a pixel value of 1 in the reconstructed image, and keeping sufficient bits to avoid overflow. It is necessary to determine how many bits are required to represent the DCT coefficients, and since a multiplication increases the bitwidth of the numbers, the output of the multiply and accumulate is an obvious truncation point. Table I describes the bit allocation for each step in the DCT process.

TABLE I. PROPAGATION OF BIT WIDTH AND BINARY POINT POSITION

	Sign	Total bits	Binary places
Pixels	Signed*	8	0
DCT Coefficients	Signed	*N*	*N*
Sum and Difference	Signed	9	0
After multiplication	Signed	*N*+9	*N*
After row DCT	Signed	*N*+11	*N*
Want *R* bits after row DCT. Therefore drop (*N*+11-*R*) bits			
After truncation of row DCT	Signed	*R*	*R*-11
Sum and Difference	Signed	*R*+1	*R*-11
Multiplication	Signed	*N*+*R*+1	*N*+*R*-11
After column DCT (2D-DCT)	Signed	*N*+*R*+2	*N*+*R*-11
Want *C* bits after 2D-DCT. Therefore drop (*N*+*R*+2-*C*) bits			
After truncation of 2D-DCT	Signed	*C*	*C*-13

* Unsigned pixels are offset by 128 before DCT
N = Number of bits to represent DCT coefficients
R = Number of bits required after row DCT (1D-DCT)
C = Number of bits required after column DCT (2D-DCT)

First, a large number of bits are assigned to *R* and *C* to ensure that these will not significantly limit the accuracy of the result (20 bits are sufficient). Then the number of bits required to represents DCT coefficients, *N*, is varied, and the 2D-DCT is performed. An inverse DCT is calculated on the result using double precision floating point with MATLAB's inbuilt IDCT function. The difference between the original image and the reconstruction is taken to determine the errors introduced through the reduced precision arithmetic.

The results for the Barbara image, shown in Fig. 3, indicate that 9 bits are sufficient for representing the DCT coefficients. Having more than 9 bits will not give any significant improvement to the image.

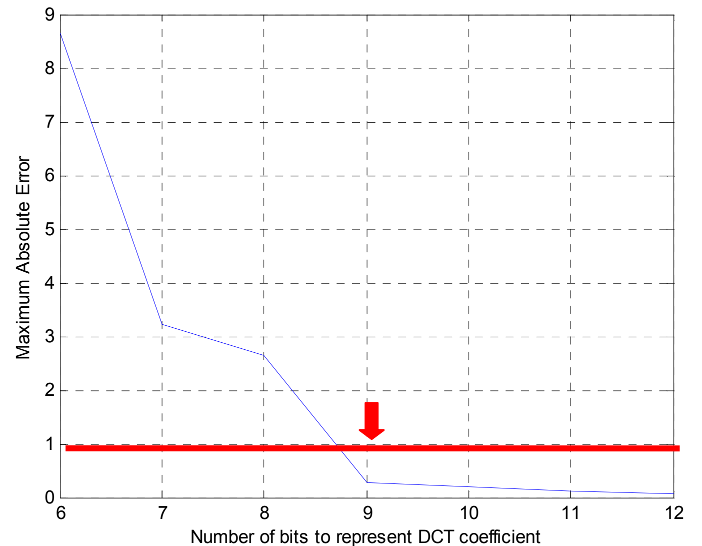

Figure 3. Effect of different DCT coefficient quantisations with R = 20, and C = 20. Arrow shows quantization which gives less than 1 MAE.

A similar process was conducted to determine the optimum values for *R* and *C*. The results of these experiments are shown in Fig. 4 and Fig. 5. From Fig. 4, 12 bits are

required to represent the output of the row DCT, and from Fig. 5, 14 bits are required to represent the output of the 2-D DCT.

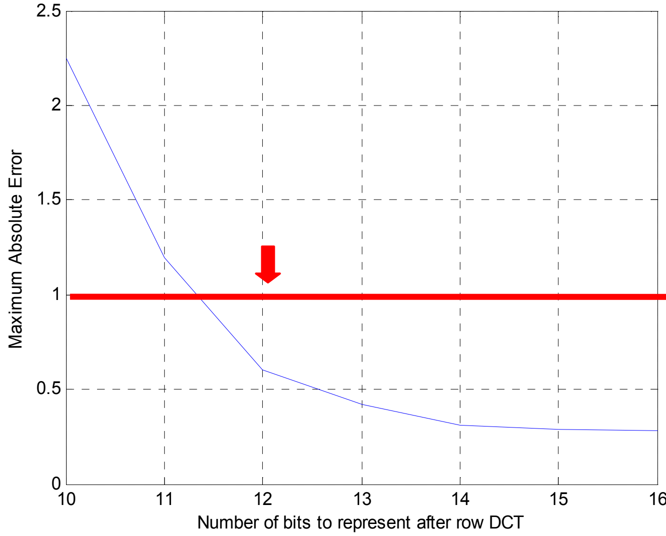

Figure 4. Effect of different row DCT quantisations with N = 9, and C = 20 Arrow shows quantization which gives less than 1 MAE.

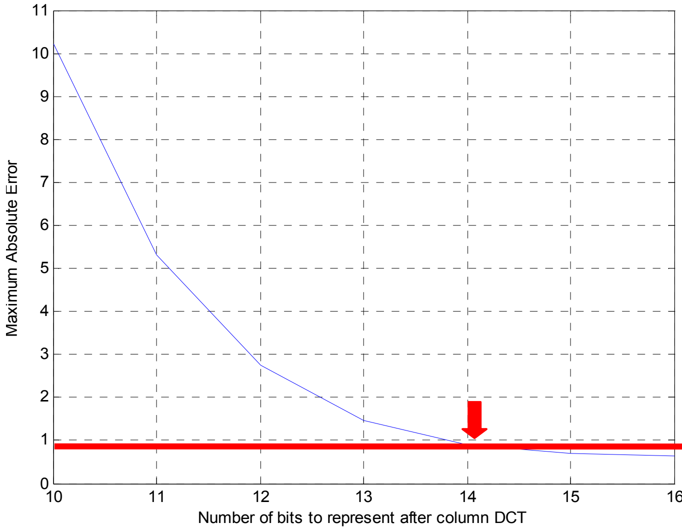

Figure 5. Effect of different column DCT quantisations with N = 9 and R = 12 Arrow shows quantization which gives less than 1 MAE.

The reconstructed Barbara image had a RMSE of 0.0435 and a maximum absolute error of 0.8982 pixel value on the greyscale pixel range (0 to 255). Lena image had a RMSE of 0.04389 and a maximum absolute error of 0.8131 pixel value. These error values shows that there has no impact on visual or perceived quality of the reconstructed image.

E. 2-D DCT implementation

With separability, (1) can be transformed to

$$D^{\mathrm{T}} = M(MX)^{\mathrm{T}} \tag{5}$$

This implies the hardware implementation shown in Fig. 6. The 1D-DCT algorithm is applied first on the data for the rows and the results are stored in a transpose buffer. Once eight row DCTs have completed, the data can be streamed out from the transpose buffer in column order for the column DCT. With careful design of the transpose buffer, the data for the next 8×8 block can be loaded during the operation of the column DCT, creating a pipelined architecture. Since the 2-D DCT samples will be reordered later with the zig-zag reordering, the fact that the output stream is ordered by column can be corrected later.

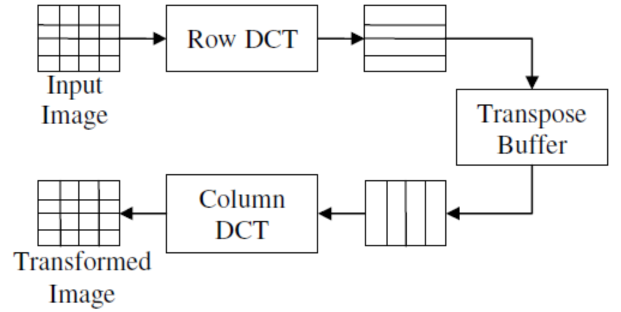

Figure 6. Generic 2-D DCT architecture

The only difference between the row and column DCTs is the number of bits used at each stage of the calculation.

F. Transpose Buffer

A transpose buffer is used to connect the two 1-D DCT architectures. To enable simultaneous read and write access, a dual-port RAM block on the FPGA is used. One port is used to write the results of the row transform, while the second port is used to read the values in column order. Although the memory can be reused, requiring only a 64×12-bit memory, the design is simplified by using a 128×12-bit memory. This is divided into two blocks of 64 buffers, operated in ping-pong mode. The row data is written into the first buffer during the first 64 cycles. The next block is written into the second block, while the column data is read out of the first block. The transpose buffer therefore has 64 clock cycles latency.

V. Implementation of Quantization and Zigzag Transform

A. Quantization

Quantization is an extremely important step in the JPEG compression algorithm, as it helps to reduce a considerable amount of data, thus reducing the entropy in the data stream. The quantization is an integer division of all the 2D-DCT coefficients by constants. For image compression, different transformed coefficients have different visual significance, so different quantization step sizes are used for each coefficient. JPEG has a suggested table specifying the quantization step sizes for each coefficient [2] which is shown in Table II.

The quantization step will perform the operation:

$$Cq_{ij} = \mathrm{round}\left(\frac{C_{ij}}{Q_{ij}}\right) \tag{6}$$

where Cq_{ij} is the quantized coefficient; C_{ij} is the 2-D DCT coefficient; Q_{ij} is the quantization constant.

TABLE II. STANDARD LUMINANCE QUANTIZATION TABLE

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

Quantization is the operation that introduces data losses in the JPEG compression process. This intended to remove the less important components to the visual reproduction of the image. The aim of quantization is to compress the image as much as possible without visible artefacts. Each step size ideally should be chosen as the perceptual threshold for the visual contribution of its corresponding cosine basis function. Larger quantization values will result in visual artefacts.

Typically a single quality control parameter is used to control image quality and compression in an image codec. This compression factor used to scale the values in the quantization table, effectively adjusting the number of steps in the resulting quantized value [19]. Larger quantization steps will lead to greater distortion and a smaller resulting data set.

B. Hardware Implementation of Quantization

Quantization is an integer division of each DCT coefficient by the corresponding constant, and rounding the result to the nearest whole number. The division can be implemented very efficiently in hardware as a shift operation if the quantization factors are restricted to powers of 2 [11]. However, this is overly restrictive.

We have implemented the division using a signed by unsigned non-restoring divider. The algorithm used to implement the division is based on the formula derived in [24].

$$R_i = 2R_{i-1} + \begin{cases} D & \text{if } R_{i-1} < 0 \\ \overline{D} + 1 & \text{if } R_{i-1} \geq 0 \end{cases} \tag{7}$$

$$q_i = \begin{cases} -1 & \text{if } R_{i-1} < 0 \\ 1 & \text{if } R_{i-1} \geq 0 \end{cases} \tag{8}$$

where R_i is the partial remainder, D is the divisor, $\overline{D}$ is its one's complement, and q_i is the ith bit of the quotient.

The advantage of using non-restoring division over the standard restoring division is that a test subtraction is not required. The sign bit determines whether an addition or subtraction is used. The disadvantage is that an extra bit must be maintained in the partial remainder to keep track of the sign.

The quantization module is shown in Fig. 7. It consists of RAM to store the quantization table. Quantization was implemented assuming a quality factor of 50 (which uses the quantisation values in Table II without scaling). The input to the quantization module has 14 bits and the output has 8 bits. The latency of the quantization module is 1 clock cycle. Note that the quantization table is reordered from that shown in Table II to account for the transpose and even/odd grouping of the data from the DCT.

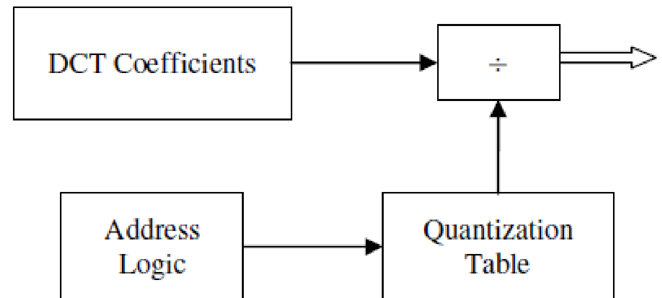

Figure 7. Quantization Module

C. ZigZag Transform

After quantization many of the high frequency components are set to zero. Zigzag coding will transform the 8×8 block into a sequential list of 64 values. The zigzag process organises the sequence to have the lower frequency components, which are less likely to be zero, in the first part of the data stream. This attempts to organise the data to have long runs of zeros, especially at the end, making run length coding very efficient.

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure 8: Sequence obtained by zig-zag reordering

D. Hardware Implementation of Zigzag buffer

The architecture for the zigzag buffer is very similar to the transposition buffer used in the 2-D DCT. A dual-port 128×8-bit memory block is split into two 64 entry buffers, used in ping-pong mode. Quantized data is written to one buffer with the zig-zag ordered data is read from the other buffer. After each 8×8 block, the role of the buffers is swapped.

The architecture for reading the zigzag buffer is shown in Fig. 9. The address logic consists of a counter. The zigzag lookup table (LUT) converts the sequential address to the

address within the coefficient block containing the zig-zag ordered coefficient. The LUT is implemented using a memory block on the FPGA. The LUT entries also take into account the non-sequential ordering of the data from the DCTs, and that the data is transposed.

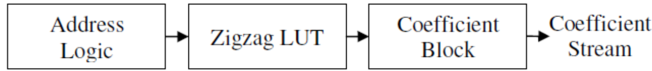

Figure 9. Implementation of zigzag reordering

VI. Implementation of Entropy Coding

Each 8×8 block has one DC coefficient, and 63 AC coefficients. Sequences of successive zero AC coefficients are run length encoded to reduce the number of symbols which need to be output for each block.

The final stage in JPEG compression is entropy coding. This assigns a variable length code to each symbol in the output stream based on the frequency of occurrence [10]. The objective of entropy coding is to use fewer bits to represent a symbol which appears more frequently and more bits to represent a symbol which appears less frequently. With JPEG compression, the symbols actually encoded are the run-length of consecutive zero coefficients, and size of the quantized coefficient. The coefficient itself is simply written to the output bitstream (also using a variable number of bits). One of the most common forms of entropy coding is Huffman coding. This uses the optimum integer number of bits for each symbol.

The Huffman coding module consists of two interrelated modules which are the run length encoder and the Huffman encoder. Huffman coding requires a Huffman code table to be specified by the application. Since the DC and AC coefficients have quite different statistics, a separate Huffman table is used for the DC and AC symbols. Each block consists of one DC codeword, and one or more AC codewords. The same Huffman tables used to compress an image will be used to decompress it.

A. DC Differential Coding

Differential coding is used to reduce the entropy of the DC coefficient. The DC value of the first block is passed directly to the Huffman coding module. Afterward, the value coded is the difference between the DC value of the current block and the DC value of the previous block.

$$DC_{code} = DC_i - DC_{i-1} \tag{9}$$

The symbol actually coded is the size of DC_{code} according to the range in Table III [2]. This is then followed by the actual coefficient using the required number of bits.

B. Run length Coding

The first step in entropy coding of the AC coefficients is run length coding. Run length encoding counts the number of zero coefficients before each non-zero coefficient in the zigzag transformed data. The size of the non-zero coefficient is also determined (from Table III [2]). The combined run length and coefficient size jointly make up an AC symbol, which is Huffman encoded, and followed by the non-zero coefficient [11].

C. Huffman Coding

This is the last step in the encoding process. It packs the data by assigning unique variable length codewords for each symbol that can be recovered without loss during decompression. The default Huffman tables defined in [2] were used for encoding the DC and AC symbols. These have been developed from the average statistics of a large set of images with 8-bit precision [2].

TABLE III. The Relationship Between Size And Amplitiude

Size	Amplitude
0	0
1	-1, 1
2	-3, -2, 2, 3
3	-7 ~ -4, 4 ~ 7
4	-15 ~ -8, 8 ~ 15
5	-31 ~ -16, 16 ~ 31
6	-63 ~ -32, 32 ~ 63
7	-127 ~ -64, 64 ~ 127
8	-255 ~ -128, 128 ~ 255
9	-511 ~ -256, 256 ~ 511
10	-1023 ~ -512, 512 ~ 1023
11	-2047 ~ -1024, 1024 ~ 2047

Two special symbols are introduced. One is EOB which represents that the remainder of the block is zeros. The second is ZRL which is used when a run-length is greater than 16 is encountered. Since a maximum run of 15 zeros is allowed, the ZRL code represents a block of 16 zeros without a corresponding non-zero coefficient.

D. Hardware Design for Huffman Coding

The architecture of the entropy encoder is shown in Fig. 10. The zero-run counts the number of successive zeros in the streamed output from the zigzag process. If the input coefficient is zero, the zero-run counter increments by one. If the input coefficient is nonzero, the coefficient is sent out to the size detector and the zero-run counter is reset to zero.

The size detector determines the number of bits required to represent the coefficient. First, 1 is subtracted from negative coefficients, and then the most significant bits which are identical are eliminated. The remaining bits form the coefficient value, and the number of bits is the size according to Table III. The bit counting is performed efficiently in a single clock cycle using a multiplexer based successive approximation counter. For the 8-bit input coefficient, the counter works as follows:

- the 5 MSBs of the remainder are checked. If all 5 bits are same, the 4 least significant bits (LSBs) are selected and a 0 is output, otherwise the 4 MSBs are selected and a 1 is output.
- the process is repeated, checking the 3 MSBs of the result, and selecting either the 2 LSBs or 2 MSBs.
- Finally the process is repeated checking the 2 remaining bits.
- The three bits output give the size of the coefficient.

These results (run length, size and coefficient amplitude) are then stored in FIFO buffer which will then be used to determine the Huffman code. The FIFO buffer allows the ZRL and EOB codes to be encoded without having to pause the incoming coefficient stream.

A lookup table based approach has been used to assign the Huffman code for each coefficient. The size and run length are looked up in the Huffman table to determine the corresponding Huffman code (and length of the Huffman codeword, which is used for packing the bits into the output). The Huffman code is concatenated with the coefficient amplitude to form the codeword for each coefficient. Obviously, codewords obtained are variable in length. A barrel shift is used to align the codeword with the remaining bits. The lengths are added to the number of remaining bits with the most significant bits used to determine the number of completed bytes.

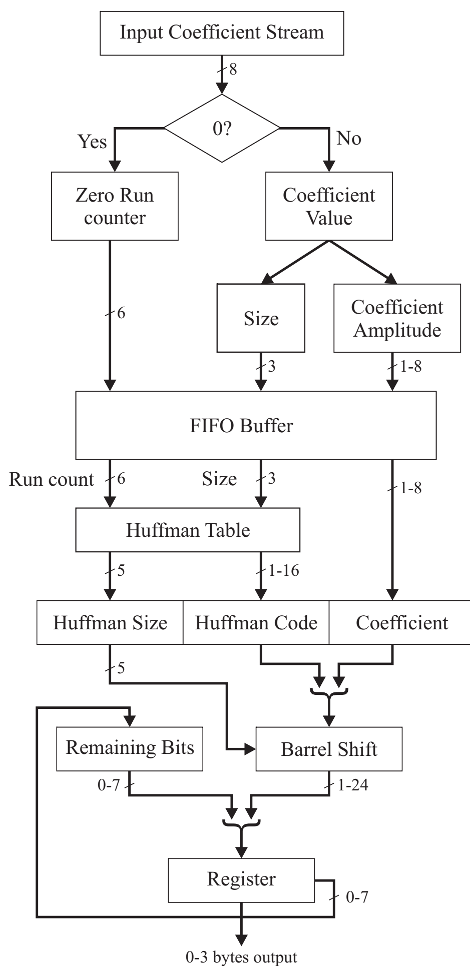

Figure 10. Huffman Coding architecture

The Huffman encoder outputs up to 24 bits for each symbol. These are saved into a FIFO buffer to enable them to be streamed out 8 bits at a time. Any bytes containing all 1s (used in JPEG to indicate an escape code) are followed with a byte containing all 0s so that the image will decode correctly.

VII. JPEG HEADERS

A JPEG file consists of more than just the encoded coefficients. A series of headers [2] is used to

- identify the fact that the file is JPEG encoded
- provide the size of the coded image
- provide additional metadata for the image
- specify the quantization and Huffman tables

In our application, a single quality factor was used, so the quantization tables and Huffman tables were fixed, enabling the pre-initialized headers to be stored in a memory block. The output byte stream was started by streaming the JPEG headers while the pipeline was being primed at the start of the image.

At the end of the image, the remaining bits were flushed by padding with 1s, and an end of image marker code was output.

VIII. RESULTS AND DISCUSSION

The latency of each step in the process is summarized in Table IV. As can be seen, the JPEG header only takes 328 clocks cycles to stream out. This gives enough time to process the JPEG compression for the first 8×8 block of the image (which takes 8 rows + 154 clock cycles).

TABLE IV. LATENCY OF EACH STAGE

	Latency (cycles)
Header	328
Block Processing	8 rows
1-D DCT	10
Transpose Buffer	64
2-D DCT	10
Quantization	1
ZigZag Transform	64
Entropy Coding	5
JPEG Compressor	154

Functional testing and timing analysis were carried out for each JPEG module on its own and then checked in an integrated test for the functionality to validate results in Handel-C simulations. Standard test images were used as benchmarks to test functionality of the design and the results validated using MATLAB. IrfanView and MATLAB were also used to check that the resulting compressed file could be successfully decoded.

Fig. 11 and Fig. 12 are two 256×256 gray scale images. The images are first compressed using quality factor 50 in Handel-C simulation and then decompressed using MATLAB. These are compared with saving the same images from MATLAB with the same quality factor. In Fig. 11, the compression ratio was 9.23:1 compared with MATLAB's 9.20:1. The small difference results from slight differences in the calculations when using fixed point arithmetic compared to double precision floating point in MATLAB. While the

differences calculated in section IV D are small (much less than one pixel value), they can move a pixel from one side of a quantisation boundary to another, resulting in significant differences in the compressed images.

The mean square error (MSE) between the original and compressed images from Handel-C and MATLAB are 14.00 and 13.84 respectively (PSNR 36.67 dB and 36.72 dB). These are sufficiently close to be satisfied that the hardware compression is not significantly different from the software compression in terms of quality. Indeed, the MSE between the two compressed images was only 0.8065.

For the image in Fig. 12, the compression ratios are 9.05:1 for Handel-C and 9.02:1 for MATLAB. The MSE of the reconstructed image is 16.03 from Handel-C and 15.91 from MATLAB (PSNR of 36.08 dB and 36.11 dB respectively). The MSE between the compressed images was 0.8335. These results are similar to the previous example.

Figure 11. Lena reconstructed image (PSNR 36.67 dB)

Figure 12. Peppers reconstructed image (PSNR 36.08 dB)

IX. CONCLUSION

This paper presents the design of an FPGA implementation of a JPEG compressor for 8 bit gray scale images. The JPEG compressor architecture for each module is presented, that minimises the logic resource of the FPGA and latency at each stage of the JPEG compression. The JPEG compressor was designed in a fully pipelined fashion. Improvements were made by minimising the latency, and increasing the performance. The proposed JPEG compressor architecture has a latency of 154 clock cycles plus 8 image rows.

The software and hardware based algorithms did have small differences in the compressed images as a result of simplifying the arithmetic in hardware. However, these differences were small, with no discernable difference in image quality between hardware and software compressed images.

ACKNOWLEDGEMENTS

Ann Malsha De Silva acknowledges receipt of a Master of Engineering Scholarship from Massey University and Graduate Women Manawatu Postgraduate Scholarship from Graduate Women Manawatu Charitable Trust Inc.

REFERENCES

[1] V. A. M. Prakash and K. S. Gurumurthy, "A Novel VLSI Architecture for Digital Image Compression using Discrete Cosine Transform and Quantisation," *International Journal of Computer Science and Network Security,* vol. 10, no. 9, pp. 175-182, September 2010 2010.

[2] International Telecommunication Union, "Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines." vol. T.81, 1992.

[3] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 still image coding system: an overview," *IEEE Transactions on Consumer Electronics,* vol. 46, no. 4, pp. 1103-1127, 2000.

[4] GIF89a, "Graphics Interchange Format Specification," Columbus,: OH: CompuServe, Inc.

[5] S. Gordoni, "Investigation of Hardware JPEG Encoder Implementation and Verification Methodologies," Department of Electrical and Computer Engineering, University of California Santa Barbara 2006.

[6] S. A. K. Jilani and S. A. Sattar, "JPEG Image Compression using FPGA with Artificial Neural Networks," *International Journal of Engineering and Technology,* vol. 2, no. 3, pp. 252-257, 2010.

[7] J. Ahmad, K. Raza, M. Ebrahim, and U. Talha, "FPGA based implementation of baseline JPEG decoder," in *Proceedings of the 7th International Conference on Frontiers of Information Technology* Abbottabad, Pakistan: ACM, 2009, pp. 1-6.

[8] C. Johnston, D. Bailey, and P. Lyons, "A Visual Environment for Real-Time Image Processing in Hardware (VERTIPH)," *EURASIP Journal on Embedded Systems,* vol. 2006, no. 1, p. 072962, 2006.

[9] L. V. Agostini, I. S. Silva, and S. Bampi, "Multiplierless and fully pipelined JPEG compression soft IP targeting FPGAs," *Microprocess. Microsyst.,* vol. 31, no. 8, pp. 487-497, 2007.

[10] D. G. Bailey, in *Design for Embedded Image Processing on FPGAs*: John Wiley & Sons (Asia) Pte Ltd, 2011.

[11] S. H. Sun and S. J. Lee, "A JPEG Chip for Image Compression and Decompression," *The Journal of VLSI Signal Processing,* vol. 35, no. 1, pp. 43-60, 2003.

[12] M. Kovac and N. Ranganathan, "JAGUAR: a fully pipelined VLSI architecture for JPEG image compression standard," *Proceedings of the IEEE,* vol. 83, no. 2, pp. 247-258, 1995.

[13] R. Uma, "FPGA Implementation of 2-D DCT for JPEG Image Compression," *International Journal of Advanced Engineering Sciences and Technologies (IJAEST),* vol. 7, no. 1, pp. 001-009, 2011.

[14] H. Anas, S. Belkouch, M. El Aakif, and N. Chabini, "FPGA implementation of a pipelined 2D-DCT and simplified quantization for real-time applications," in *Multimedia Computing and Systems (ICMCS), 2011 International Conference on*, 2011, pp. 1-6.

[15] L. V. Agostini, I. S. Silva, and S. Bampi, "Pipelined fast 2D DCT architecture for JPEG image compression," in *Integrated Circuits and Systems Design, 2001, 14th Symposium on.*, 2001, pp. 226-231.

[16] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *International Conference on Acoustics, Speech, and Signal Processing, ICASSP-89.*, 1989, pp. 988-991 vol.2.

[17] Mentor, "Handel-C Language Reference Manual," Mentor Graphics Corporation, 2010.

[18] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," *IEEE Transactions on Computers,* vol. C-23, no. 1, pp. 90-93, 1974.

[19] I. E. G. Richardson, *Video Codec Design: Developing Image and Video Compression Systems*: Wiley, 2002.

[20] T. C. Chen, M. T. Sun, and A. M. Gottlieb, "VLSI implementation of a 16x16 DCT," in *International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, 1988, pp. 1973-1976 vol.4.

[21] H. Hou, "A fast recursive algorithm for computing the discrete cosine transform," *IEEE Transactions on Acoustics, Speech and Signal Processing,* vol. 35, no. 10, pp. 1455-1461, 1987.

[22] Z. Cvetkovic and M. V. Popovic, "New fast recursive algorithms for the computation of discrete cosine and sine transforms," *IEEE Transactions on Signal Processing,* vol. 40, no. 8, pp. 2083-2086, 1992.

[23] R. Woods, D. Trainor, and J. P. Heron, "Applying an XC6200 to real-time image processing," *IEEE Transactions on Design & Test of Computers,* vol. 15, no. 1, pp. 30-38, 1998.

[24] D. G. Bailey, "Space Efficient Division on FPGAs," in *Electronics New Zealand Conference (EnzCon'06)* Christchurch, NZ, 2006, pp. 206-211.