1998

Real-Time Implementation of JPEG Encoder/ Decoder

Thomas M. Czyszczon
Rochester Institute of Technology

Roy S. Czernikowski
Rochester Institute of Technology

Muhammad Shaaban
Rochester Institute of Technology

Kenneth Hsu
Rochester Institute of Technology

Recommended Citation
Thomas M. Czyszczon, Roy S. Czernikowski, Muhammad E. Shaaban, Kenneth W. Hsu, "Real-time implementation of JPEG encoder/ decoder", Proc. SPIE 3422, Input/Output and Imaging Technologies, (18 June 1998); doi: 10.1117/12.311097; https://doi.org/ 10.1117/12.311097

Real-Time Implementation of JPEG Encoder/Decoder

T. M. Czyszczon, R. S. Czernikowski, M. Shaaban, K. W. Hsu

Department of Computer Engineering,
Rochester Institute of Technology, Rochester, NY

ABSTRACT

With the immense size of images, compression has become a common way of minimizing the amount of storage necessary for images. This is also beneficial for transmission purposes. The Joint Photographic Experts Group (JPEG) standard is frequently used for still images. This standard is very flexible and many of the same algorithms can be used for video applications. Video applications require large amounts of data to be processed every second. Therefore, the following describes the hardware design of a chip allowing for high-speed compression. The design uses the JPEG algorithms and is targeted towards ASIC design. Further plans include use of field programmable gate arrays (FPGAs). The hardware design is based on grayscale images and only works with the raw image data.

Keywords: JPEG, compression, FPGA, video, still image, ASIC, real-time[5]

1. JPEG STANDARD

The JPEG standard is based off the Discrete Cosine Transform (DCT). It gives a lot of flexibility so as to attain a desired compression ratio (CR). The compression ratio is defined as the ratio of the number of bits in an image before compression to the number of bits after compression. There are two types of compression: lossless and lossy.

Lossless compression will give the best image quality; however, it will also give the smallest CR. Ideally, there will be no errors in the image when it is decompressed, but due to necessary rounding of values for compression and decompression, there will be occasional minor differences that occur. A majority of these errors will not be visible to the human visual system (HVS)[2].

Lossy compression will help increase the CR. In most cases, the CR is inverse proportional to the image quality. The larger the CR, the more distorted the image will be after it is decompressed. There is a limit to the CR with respect to the HVS. The HVS is unable to notice minor losses to the image quality, but after a certain CR the image quality will become noticeably distorted. The CR limit is different for each image.

The JPEG standard works well with images that do not have a lot of sharp contrast in them. This occurs with images where the neighboring pixel values are relatively close to each other. In the case of compressing an image with a lot of text in it, the algorithm would not be able to minimize the size much since there can be many large differences between adjacent pixels.

1.1 Compression

The compression algorithm consists of the three following components: DCT, quantization, and encoding. The quantization component is only necessary with lossy compression where some of the image quality is sacrificed for a better compression ratio. The component can be removed to allow for lossless compression. An image is initially broken up into 8x8 blocks. Each of these 8x8 blocks then goes through the following steps.

1.1.1 DCT

The DCT is based off the Fourier Transform.[1] This transform places an 8x8 block of data in the frequency domain, which represents the contrasts between the pixel values. The following equation is performed on each of the 8x8 blocks of the image.

$$F(u,v) = \frac{C(u)C(v)}{4} \sum_{j=0}^{7} \sum_{k=0}^{7} f(j,k) \cos\left[\frac{(2j+1)u\pi}{16}\right] \cos\left[\frac{(2k+1)v\pi}{16}\right] \quad (1)$$

$$C(u) = \frac{1}{\sqrt{2}} \text{ for u=0, and 1 for u=1,2,..,7}$$

$$C(u) = \frac{1}{\sqrt{2}} \text{ for u=0, and 1 for u=1,2,..,7}$$

The result of this equation is an 8x8 matrix representing the frequency domain of the pixel values in the original 8x8 block. Most of the image data will be retained in only a portion of the matrix

1.1.2 Quantization

Quantization is used to allow for a better CR. A typical table used for quantization is given below:

16	11	10	16	24	50	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

Figure 1: Quantization table

Each component in the transform matrix, F(u,v), is divided by the corresponding value in the quantization matrix, Q(u,v), based on the following formula:

$$F'(u,v) = \begin{cases} \left[\dfrac{F(u,v) + \left[\dfrac{Q(u,v)}{2}\right]}{Q(u,v)}\right], F(u,v) \geq 0 \\ \left[\dfrac{F(u,v) - \left[\dfrac{Q(u,v)}{2}\right]}{Q(u,v)}\right], F(u,v) < 0 \end{cases} \quad (2)$$

This formula takes into account rounding off to the nearest integer value. The values in the table can be adjusted for more specific applications. Quality factors (QF) can also be included such as dividing or multiplying the entire quantization matrix by 2 to improve image quality or to increase the CR, respectively. The result is an 8x8 quantized matrix, F'(u,v).

1.1.3 Encoding

With encoding, the coefficients of the incoming matrix, either F(u,v) or F'(u,v) depending on whether quantization was used, are read in a zigzag order:

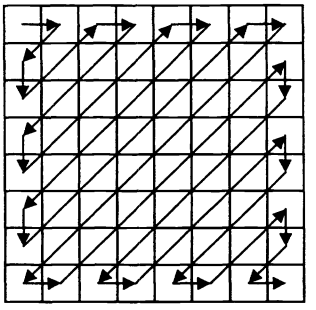


Figure 2: Zigzag ordering[4]

Initially, a new DC-coefficient is determined using differential pulse-code modulation (DPCM)[2]. The DC-coefficient is the first value in the matrix. This is determined by taking the difference between the current DC-coefficient and the DC-coefficient of the previous 8x8 block used. If there was no previous block, then the previous value is set to zero. A difference magnitude coding, SSSS, is then determined from the following table. The SSSS value is a 4-bit value representing the size of the value:

SSSS	Difference	Code Length	Code Word
0	0	2	00
1	-1,1	3	010
2	-3,-2,2,3	3	011
3	-7..-4,4..7	3	100
4	-15..-8,8..15	3	101
5	-31..-16,16..31	3	110
6	-63..-32,32..63	4	1110
7	-127..-64,64..127	5	11110
8	-255..-128,128..255	6	111110
9	-511..-256,256..511	7	1111110
10	-1023..-512,512..1023	8	11111110
11	-2047..-1024,1024..2047	9	111111110

Table 2: Difference magnitude coding and typical Huffman table[3]

The SSSS size value obtained above is then coded using a Huffman table. Huffman tables are based primarily on the probabilities of the value used[6]. The more frequently used values have the shortest codes assigned to them. The value itself is attached afterwards but without the most significant bit (MSB) since the SSSS value represents the MSB.

The remaining values in the matrix are called the AC-coefficients. These values are encoded slightly differently using an 8-bit value represented as RRRRSSSS[6]. The run-length, 4-bit RRRR value, is the number of zeros preceding a non-zero value using the zigzag format of reading the matrix. The non-zero value is then coded by size, 4-bit SSSS value, as was described for the difference magnitude. If the run of zeros is greater than 16, then it is split up into intervals of 16. Therefore, a run of 16 zeros is represented by the RRRRSSSS value of F0. The 00 (EOB) value is used when there are less than 16 values remaining in the block and they are all zeros.

The RRRRSSSS values are encoded with a Huffman table. The value, without the MSB, is attached to the end of the Huffman code as was done with the difference magnitude.

1.2 Decompression

Decompression consists of three main components: decoding, dequantization, and inverse Discrete Cosine Transform (IDCT). The dequantization component is not used if the original image was not quantized.

1.2.1 Decoding

The data coming into the decoder is decoded using the same Huffman tables described for compression. The first data of the block will represent the difference magnitude of the DC-coefficient. The code is looked up in the table to get the SSSS value. This tells the decoder how many additional bits to read to get the value of the difference. This value is then added to the previous DC-coefficient to get the current DC-coefficient value.

The same is done with the AC-coefficient values. The code is looked up in the Huffman table to determine the run length and size, RRRRSSSS. RRRR zeros are entered to the matrix in zigzag ordering and SSSS additional bits are read to get the non-zero value. The MSB is added to the to the beginning of the value obtained since it was coded in the SSSS value. This value is than placed in the matrix. This is done for all the 8x8 blocks coming in.

1.2.2 Dequantization

Dequantization is only used if the compressed image was quantized. The same table is used as was in the quantization step of compression. This time through, each value in the decoded 8x8 matrix is multiplied by the appropriate value in the 8x8 quantization matrix. The result will be an 8x8 matrix similar to the 8x8 matrix formed from the DCT in the compression algorithm. Some of the values will be off considerably due to the quantization. This truncates a lot of values so when they are multiplied by the table again, some of the precision is lost. Most of the lower frequencies will be more accurate since they are most significant to the image quality.

1.2.3 IDCT

To obtain the original pixel values, the IDCT must be operated on the matrix from the decoding stage or dequantization stage depending on whether lossless or lossy compression was used. The IDCT is described by the following equation:

$$f(j,k) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\left[\frac{(2j+1)u\pi}{16}\right]\cos\left[\frac{(2k+1)v\pi}{16}\right] \tag{3}$$

$$C(u) = \frac{1}{\sqrt{2}} \text{ for u=0, and 1 for u=1,2,..,7}$$

$$C(u) = \frac{1}{\sqrt{2}} \text{ for u=0, and 1 for u=1,2,..,7}$$

With lossless compression, the 8x8 block resulting from the IDCT will be nearly identical to the original[3]. Some undistinguishable errors will occur due to rounding. Lossy compression will result in some differences due to the quantization.

2. ARCHITECTURE

To be able to handle a large throughput, a pipelined architecture is used[7]. This allows the hardware to run at a pixel rate, which means that a pixel from the image is read in with each clock cycle[8]. Also, since the DCT is orthogonal, the same hardware is used for the DCT and IDCT to help keep the size of the hardware down to a minimum[1]. By having the DCT and IDCT share the same hardware, this also limits the design to performing only compression or decompression at a time.

2.1 DCT/IDCT architecture[9,10,11]

By multiplying a factor of one to the original DCT equation, the formula can be shown as follows:

$$F(u,v) = \frac{C(u)C(v)}{8} \sum_{j=0}^{7} \sum_{k=0}^{7} f(j,k)\left(\sqrt{2}\cos\left[\frac{(2j+1)u\pi}{16}\right]\right)\left(\sqrt{2}\cos\left[\frac{(2k+1)v\pi}{16}\right]\right) \tag{4}$$

Also, the DCT is a separable transform,

$$g(j,v) = \sum_{k=0}^{7} f(j,k)C(v)\left(\sqrt{2}\cos\left[\frac{(2k+1)v\pi}{16}\right]\right), \qquad F(u,v) = \frac{1}{8}\sum_{j=0}^{7} g(j,v)C(u)\left(\sqrt{2}\cos\left[\frac{(2j+1)u\pi}{16}\right]\right) \quad (5)$$

This equation can be viewed as a matrix multiplication,

$$[g] = [f][D] \quad , \qquad [F] = \frac{1}{8}[D]^T[g] \, , \qquad [D] = \begin{bmatrix} 1 & c & a & d & 1 & e & b & f \\ 1 & d & b & -f & -1 & -c & -a & -e \\ 1 & e & -b & -c & -1 & f & a & d \\ 1 & f & -a & -e & 1 & d & -b & -c \\ 1 & -f & -a & e & 1 & -d & -b & c \\ 1 & -e & -b & c & -1 & -f & a & -d \\ 1 & -d & b & f & -1 & c & -a & e \\ 1 & -c & a & -d & 1 & -e & b & -f \end{bmatrix} \quad (6)$$

where, $a = \left(\sqrt{2}\right)\cos\left(\frac{2\pi}{16}\right), b = \left(\sqrt{2}\right)\cos\left(\frac{6\pi}{16}\right), c = \left(\sqrt{2}\right)\cos\left(\frac{\pi}{16}\right), d = \left(\sqrt{2}\right)\cos\left(\frac{3\pi}{16}\right), e = \left(\sqrt{2}\right)\cos\left(\frac{5\pi}{16}\right), f = \left(\sqrt{2}\right)\cos\left(\frac{7\pi}{16}\right)$

The above equation shows there are a fixed number of DCT coefficient values available and these cosine values can be stored in look-up tables. This helps eliminate the need for a multiplier. Also, these same tables can be used for the two stages of the DCT. The largest look-up value needed would be a 12-bit value. To help minimize the size of the tables, each cosine product is broken up into two tables. The first table holds the index values from 0 to 63. The second table holds the index values for multiples of 64 up to 4096. The total product will then be the sum of the partial products. The magnitude of the look-up tables requires 12-bits for storage. The remaining 4-bits are used as the precision of the cosine values.

The first stage of the DCT generates the [g] matrix from equation 6 one row at a time. Having eight accumulators running in parallel as shown in Figure 3 does this. Initially, all the accumulators are set to zero. The first pixel of the row of the original data is read in and the eight coefficient products corresponding to the first row of the [D] matrix are obtained and added to the accumulators. The eight coefficient products corresponding to the second row of the [D] matrix are obtained and added to the accumulators but this time using the second pixel value as the index. The procedure is done for the 8 pixels in the row of the original image data. The result will be the corresponding row in the [g] matrix.

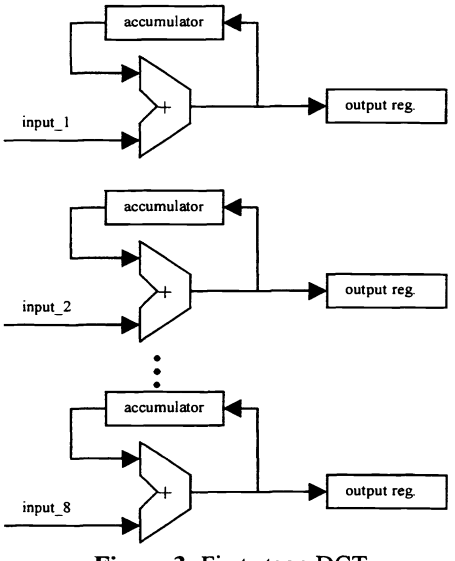


Figure 3: First stage DCT

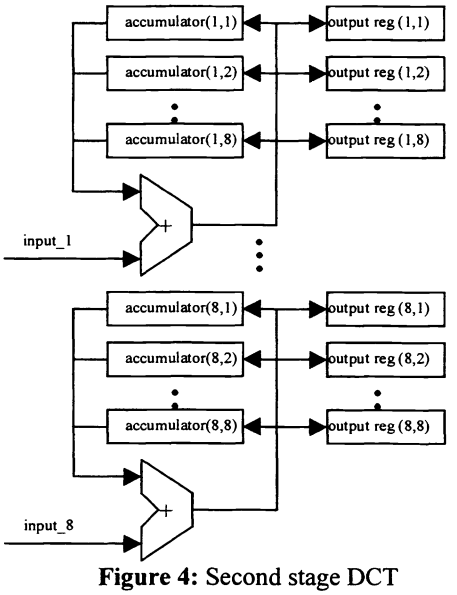


Figure 4: Second stage DCT

The second stage of the DCT receives data from the output of the first stage one value at a time. These values are from a row of the [g] matrix calculated in the first stage. The architecture is similar to the first stage except that there are eight accumulators for each adder. Initially, all 64 accumulators are set to zero. The coefficient products corresponding to the first row of the [D] matrix are obtained using the first value in the row of the [g] matrix as the index. These products are then added to all the first accumulators of the eight adders. Next, the first row [D] matrix coefficient products using the second value in the row as the index is added to all the second accumulators. This same thing is done for all the values in the [g] matrix row. For the second [g] matrix row coming in, the same procedure is used except the second row [D] matrix coefficient products are used. Once all 64 values of the [g] matrix are processed. The result will be the 8x8 DCT of the original 8x8 image data. The values are output one value at a time in the zigzag ordering described by Figure 2.

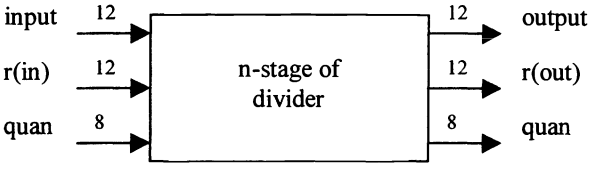
The same procedure is done for the IDCT except the transpose of the [D] matrix is used to obtain the appropriate coefficient products. The following shows the matrix representation for the IDCT.

$$[g] = [F][D]^T \qquad [f] = \frac{1}{8}[D][g] \tag{7}$$

Any value that ends up being greater than 255 is set to 255 and all negative values are set to 0. This is done since all the pixel values must be in the range of 0 to 255 coming from the IDCT and due to rounding and quantization, some values from the IDCT can be out of those ranges. These values are output row by row instead the zigzag ordering.

2.2 Quantization/Dequantization

The quantization stage consists of a 12-bit by 8-bit divider. The hardware is designed as a 12-stage pipeline where each stage determines the value of a bit in the result. The following figure shows what each stage does.



$$\text{if } input \geq quan * (2^n), \tag{8}$$
$$output = input - quan * (2^n),$$
$$r(out) = r(in) + (2^n)$$
$$\text{else}$$
$$output = input,\ r(out) = r(in)$$

Figure 5: Stage of divider for quantization

The stages are arranged in the order of most significant bit down to least significant bit of the result. Initially, the result is started at zero and the input value is the value coming from the DCT plus half of the quantization value. This takes account rounding off to the nearest integer. The input value is a signed value so that just the magnitude is operated on. The quantization value is determined from the Figure 1. The values from the quantization table are read in zigzag order since the values coming from the DCT output will be in zigzag ordering. The result at the last stage will be the result of the input divided by the quantization value.

This design includes a 3-bit field to specify the CR level. A level of 0 indicates lossless compression. Values of 1 through 7 indicate better CR respectively. This is done by taking the quantization table and multiplying all the values by $2^{(4-n)}$. Therefore, for n=1 the table would be divided by 8. For n=4, the table would remain as it is. The smaller the quantization values are, less information is lost in compression.

The dequantization step uses a similar method in multiplying a 12-bit number by an 8-bit number. The same blocks are used as shown in Figure 5 except now only 8 are necessary. Some of the same blocks are used so as to save hardware space on the use of registers. Each stage does the following calculation:

$$r(out) = r(in) + (input) * (2^n) * (quan_{(n)}), \quad \text{where } quan_{(n)} \text{ is the n}^{\text{th}} \text{ bit of quan} \tag{9}$$

The same quantization table is used as for quantization stage except the values read in normal row by row manner. The zigzag ordering will be taken care of in the decoding stage. This output will go to the IDCT stage.

2.3 Encoding

The Huffman encoding is done using a 5-stage pipeline. The Huffman lookup tables are included between the input and shift stages and the arbiter contains a buffer. The following figure describes the structure:

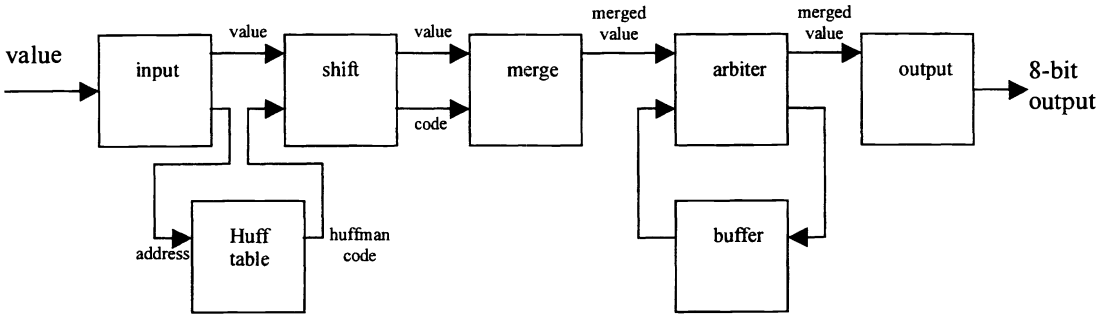


Figure 6: Huffman encoding architecture

The value from the DCT or the quantization stage enters the input stage of the Huffman encoder in the appropriate zigzag ordering. Here the previous DC-coefficient is stored so that the DPCM can be used to determine the difference magnitude. This stage also collects the number of zeros coming in and determines the size of the non-zero value so as to get the AC-coefficient code. An address is determined so as to get the appropriate Huffman code from the table and the input value minus the MSB is passed onto the next stage.

The Huffman lookup table is comprised of 256x21-bit entries. Sixteen bits are used allow for the maximum size code and five bits are necessary to store the size of the code since the codes are of variable length.

The shift stage receives the value from the input and the Huffman code from the table. This stage then shifts the values so that the MSB is in the leftmost position in the registers. These values are then sent on to the merge stage.

The merge stage then combines the Huffman code and value into a 27-bit maximum value, 16 bits maximum for the code and 11 bits maximum for the magnitude. These values are sent on to the arbiter.

The arbiter combines the merged value and the five-bit count for the size in a 32-bit value. This value is then stored into a buffer. The size of the buffer is 128x32 bits. The reason for this size will be explained in the decoding section.

The output stage receives data from the buffer through the arbiter. This stage outputs the coded version of the image 8-bits at a time. The buffer is used so as to increase the throughput of the encoder. Since the output stage can receive up to 27-bit codes at a time, it would need to stall the pipeline so as to give it time to send out the value in 8-bit increments. The buffer allows the rest of the pipeline to continue without stalling. The buffer clears up when there are many zeros coming into the input of the encoder since the input stage does not output anything until a non-zero value comes in. However, there still can be cases where the entire buffer fills up. This occurs when trying to compress things such as text images since JPEG does not work well with images that have many sharp contrasts. There are occasions where the compressed version of the image is actually larger than the original image as will be shown with some examples in the result section.

2.4 Decoding

The decoding stage consists of a 4-stage pipeline with two buffers. The following diagram illustrates the architecture of the Huffman decoder:

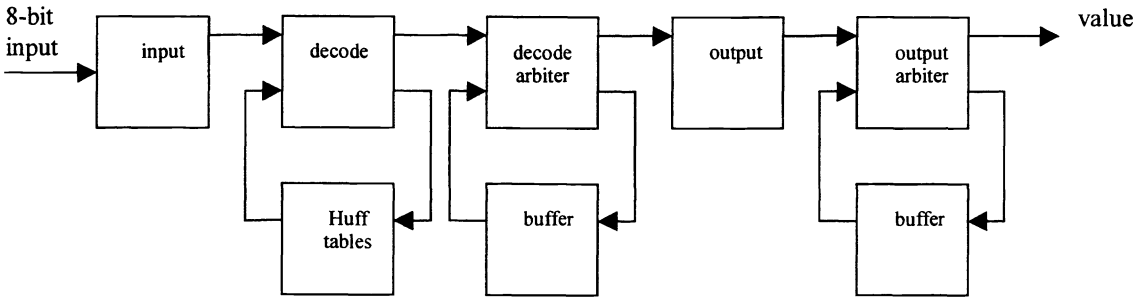


Figure 7: Huffman decoding architecture

The input stage acts as a buffer to the encoder. The decode stage converts the Huffman code to the appropriate SSSS value for DC-coefficients and RRRRSSSS values for the AC-coefficients. These values are determined through the lookup table. The tables are broken up into three components each with 256 entries: AC, DC, and jump table. The first eight bits of the value coming in are addressed into either the AC or DC table depending on the case. The table entry will either be a result or addresses to the jump table if the Huffman code is longer than 8 bits in length. The result will have the SSSS or RRRRSSSS value along with the code length since the codes are all variable length. There are two addresses in the jump entry along with the conditions to determine which address is taken based on the incoming values to the decode stage. This is done to help minimize the size of the tables. The jump table consists of just results as described for the AC and DC tables since the Huffman code cannot be larger than 16 bits.

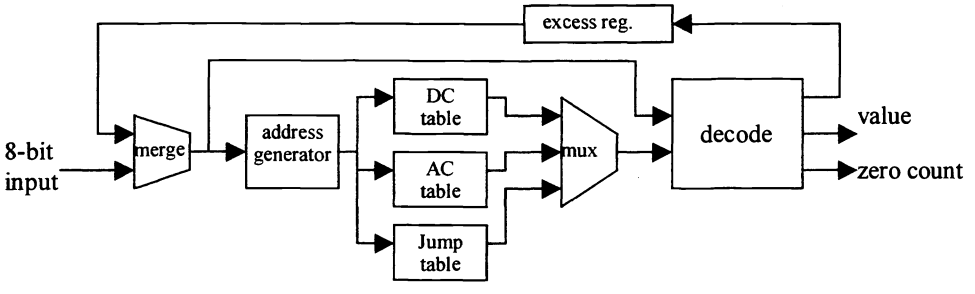


Figure 8: Decoding architecture

The decoding stage consists of a state machine with decode_1, decode_2, and value_read being the valid states. Initially, the state machine begins in the decode_1 state and the excess register is cleared. The state machine does not change any state unless the merged data between the excess register and the input is at least 8 bits or there are less than 8 bits and it is the end of the image. In the decode_1 state, the address generator determines whether the incoming code will be an AC or DC-coefficient based on internal counters. The sequence is always one DC-coefficient followed by 63 AC-coefficients repeated. The decode component then determines the output values and the next state. If the table lookup value is a jump address, the addresses and conditions are stored in the address generator, 8 bits are discarded from the merged data and the remaining bits are put in the excess register, and the next state will be decode_2. If the lookup value is a result, then the decode component determines whether the merged data contains the entire value also. If the value is contained in the merged data, the code and value are extracted from the merged data and the remaining bits are put in the excess register. Also, the value and zero count are then output to the next stage and the next state remains at decode_1. If the whole value is not contained in the merged data, then the code is extracted and the remaining bits are put in the excess register and the next state becomes value_read.

In the decode_2 state, an address is generated to the jump table based on the stored addresses, conditions, and merged data coming in. The decode stage then determines whether the code and whole value are contained in the merged data or not. If the value is contained in the merged data, the code and value are extracted from the merged data and the remaining bits are put in the excess register. Also, the value and zero count are then output to the next stage and the next state returns to decode_1. If the whole value is not contained in the merged data, then the code is extracted and the remaining bits are put in the excess register and the next state becomes value_read.

In the value_read stage, data is read in until there are enough bits available for the value. At this point, the value is extracted from the merged data and the value is output along with the zero count. The remaining bits are placed in the excess register and the state is returned to decode_1.

The decode arbiter is used to store the value and zero count in the buffer for the output stage. This buffer is 128x16 bits in size. Since the chip can only be in encode or decode mode at on time, the buffers are shared between the encoder and decoder. This allows some of the hardware to be minimized.

The output stage reads in data from the buffer through the decode arbiter. Here the zero values are extracted and sent on to the arbiter. The output arbiter then takes the values and returns them to normal row by row combinations since they are being read in the zigzag format. This requires a 64x12 bit buffer so as to accumulate the whole 8x8 block before sent on to the dequantization stage and IDCT. To improve the throughput, two 64x12 bit buffers are used to generate one 128x12 bit buffer. This is so that while one 64 block is being written to by the arbiter, the other 64 block can be sent on to the next stage concurrently. This is the reason for using a total of four 128x8 bit buffers for the design.

3. IMPLEMENTATION

The entire design was implemented in VHDL and tested using QuickVHDL. The design consists of 29 I/Os.

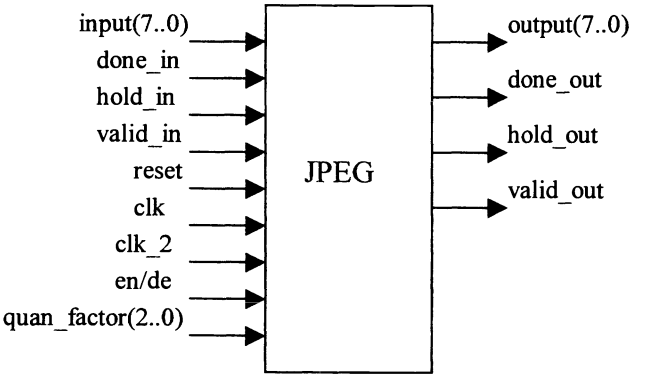


Figure 9: JPEG chip I/Os

The design consists of two clocks where clk_2 is twice the speed of clk. This is because the DCT tables are shared between the two stages so two lookups must be done for every clock cycle of clk. The en/de bit changes between encode and decode mode and the quan_factor allows for the different levels of compression as explained in the quantization stage. The input and output data lines are 8 bits in size. The done_in and done_out lines are used to indicate the end of an image. Therefore, any size image can be used as long as done_in is set on the last byte entered. The hold_out is necessary for when the buffers end up filling up internally to the chip and it cannot accept anymore data at the time. The hold_in line gives the user the option of pausing the chip.

Simulation and testing was done by setting up a testbench over the entire design and sending actual image data through it. The images were compressed and stored to separate files. Next, the compressed files went through the model again in decode mode and the original image was restored and examined. C programs were written so as to examine the images and make comparisons between the original and the compressed versions.

4. RESULTS

A number of sample images were used in the testing including images with text in them. The following shows the results of lossless compression in terms of the compression ratio and the amount of error that occurred after compression. For lossless compression, error never exceeded one in magnitude from the original. The percent error represents the number of pixels that were off from the original by one. It is seen that the text image actually compressed to a larger size than the original. As was mentioned before, images with many sharp contrasting components does not compress well using the JPEG standard,

	Images							
	boat	car	chapel	coke	roy	sunset	surfer	text
size (bytes)	20,446	28,595	29,826	20,460	21,661	20,995	26,590	73,720
CR	3.21	2.29	2.20	3.20	3.03	3.12	2.46	0.89
% error (+/- 1 pixel)	5.5	6.7	6.4	4.7	5.6	6.4	5.7	5.5

Table 1: Lossless compression results for sample images

The same images were also compressed at the different levels made available. Again it can be seen that the text image did not compress as well as the others did. The different compression levels are illustrated in Figure 10 with the car image. The image quality is still relatively good up to level 4. Afterwards, degradation in the image is much more noticeable.

	Images							
	boat	car	chapel	coke	roy	sunset	surfer	text
Level 1 CR	4.09	3.11	2.89	4.12	3.89	3.99	3.33	1.46
Level 2 CR	5.01	3.83	3.62	4.97	4.75	4.82	4.15	1.84
Level 3 CR	6.22	4.87	4.64	6.03	5.91	5.95	5.32	2.34
Level 4 CR	6.80	5.62	5.24	6.64	6.53	6.47	5.92	3.05
Level 5 CR	8.85	7.38	7.14	8.44	8.49	8.49	8.35	4.15
Level 6 CR	10.51	8.96	8.80	9.84	10.15	10.20	10.10	5.53
Level 7 CR	11.89	10.44	10.34	10.88	11.52	11.33	11.31	6.74

Table 2: Lossy compression results for sample images

The results also showed that the images were able to compress and decompress at a pixel rate. This was obviously not the case for the text image since the compressed version was larger than the original. The latency for compression was found to be about 94 clock cycles for lossless compression and about 108 for lossy compression. These values do deviate for each image due to the variable code lengths in the Huffman coding. The latency for decompression is about 162 clock cycles for lossless and 171 for lossy decompression.

4.1 VHDL synthesis

The VHDL was synthesized using the Synopsys general libraries. With the gate level delays, the maximum speed was found to be about 20 MHz. This speed would allow for video applications, which generally need to handle about 10 MHz rates.

5. CONCLUSION

In this paper, a high speed architecture for JPEG compression is proposed so as to be able to run video applications. The architecture is meant for ASIC design but is also being looked at for FPGAs. The architecture itself creates relatively small errors for lossless compression and also allows for different levels of lossy compression depending on the application. The design is very flexible and could be also applied to color images. Additional features such as loadable quantization tables and Huffman tables is also being looked at.

6. REFERENCES

[1] K. R. Rao & P. Yip, "Discrete Cosine Transform." Academic Press, Inc., Boston, 1990.

[2] Ronald G. Matteson, "Introduction to Document Image Processing Techniques." Artech House, Inc., Boston, 1995.

[3] William B. Pennebaker, "JPEG Still Image Data Compression Standard." Van Nostrand Reinhold, New York, 1993.

[4] Rafael C. Gonzalez & Richard E. Woods, "Digital Image Processing." Addison-Wesley Publishing Company, Inc., New York, 1992.

[5] Edward R. Dougherty & Phillip A. Laplante, "Real-Time Imaging." IEEE Press, New York, 1995.

[6] Weidong Kou, "Digital Image Compression: Algorithms and Standards." Kluwer Academic Publishers, Boston, 1995.

[7] Phillip A. Laplante & Alexander D. Stoyenko, "Real-Time Imaging: Theory, Techniques, and Applications." IEEE Press, New York, 1996.

[8] Ioannis Pitas, "Parallel Algorithms: for Digital Image Processing, Computer Vision and Neural Networks." John Wiley & Sons, New York, 1993.

[9] Jen-Shiun Chiang & Hsiang-Chou Huang, "New Architecture for High Throughput-Rate Real-Time 2-D DCT and the VLSI Design." Ninth Annual IEEE International ASIC Conference Proceedings, Rochester, NY , 1996

[10] L. J. D'Luna, et.al., "An 8x8 Discrete Cosine Transform Chip with Pixel Rate Clocks." Third Annual IEEE International ASIC Conference Proceedings, Rochester, NY, 1990

[11] Wei-chun Chen, Yimu Fan, & Kenneth W. Hsu, "An 8x8 Discrete Cosine Transform Model Using VHDL." Fifth Annual IEEE International ASIC Conference Proceedings, Rochester, NY, 1992

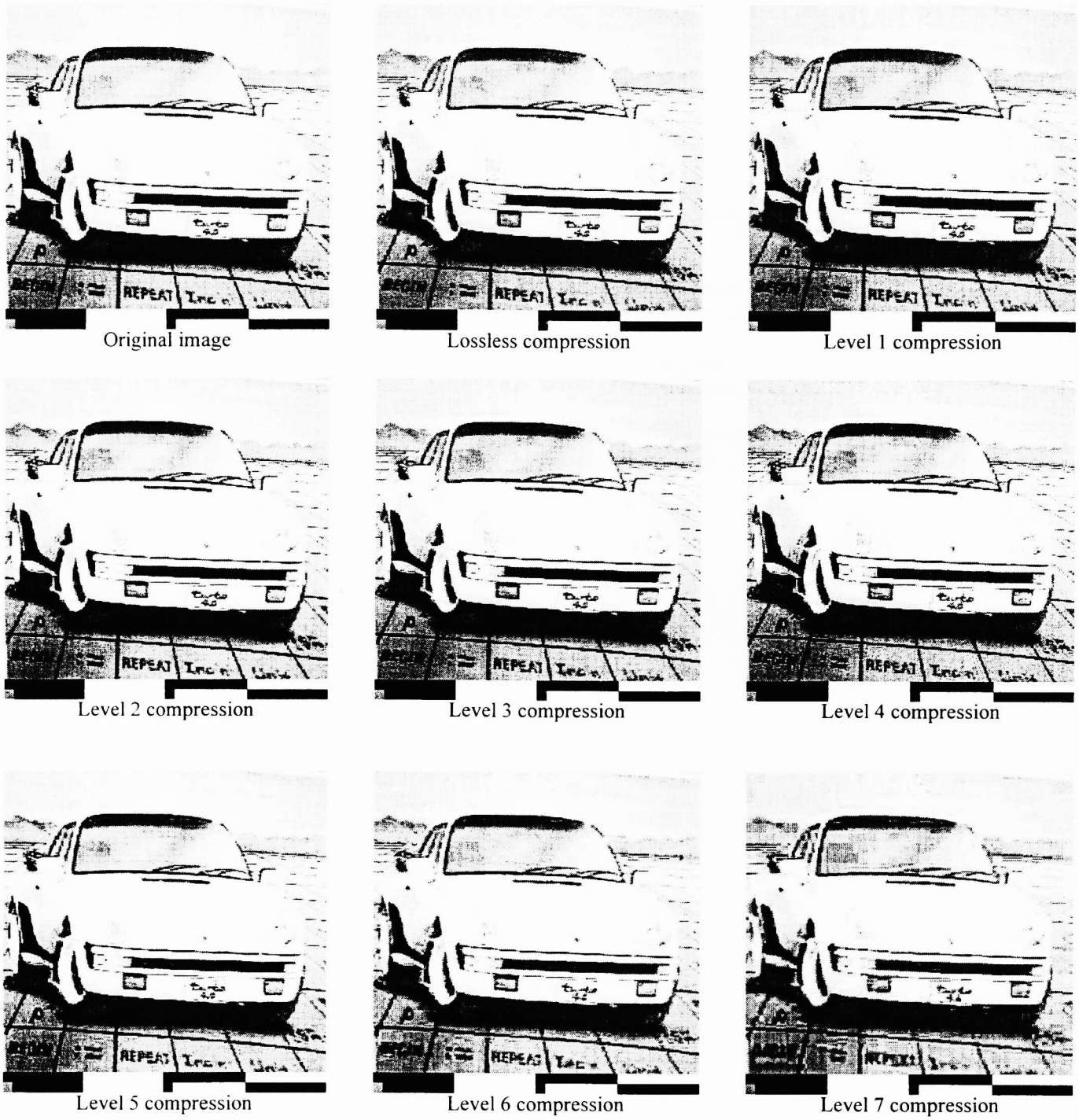


Figure 10: Compression results for car image