FPGA Implementation of JPEG Encoder

Krishna Swaroop Dhulipalla

Birla Institute of Technology and Sciences, Pilani.

f20170315@pilani.bits-pilani.ac.in

***Abstract*: - This is a summary report for a Study Oriented Project. The project is an attempt to better understand the intricacies of the JPEG compression algorithm. This has been done by implementing the algorithm on a FPGA using HDL Synthesis. A JPEG Encoder along with a testbench has been developed. The language used for describing the Encoder is Verilog. This report explains the details of various blocks in the Encoder while also drawing parallels to the standard JPEG compression algorithm.**

***Keywords:* - Verilog, Field Programmable Gate Arrays, Joint Photographic Experts Group Compression, HDL Synthesis.**

I. INTRODUCTION

In this day and age of rapid automation, there is a huge amount of data transfer. Transferring raw data is highly inefficient as there is redundancy in the data being transferred. To take advantage of this characteristic of digital media, the Joint Photographics Experts Group came up with a standard for lossy compression. This has come to be known as JPEG. It is a commonly used method of lossy compression for digital images. It is the most widely used format with billions of JPEG images being produced every day. A highlight of this technique is the adjustable compression ratio that can be tweaked as per requirement. Typically, JPEG achieves a 10:1 compression with little loss in image quality.

In this project, we explore the details of the .jpg format. This project covers Baseline Sequential JPGs and particularly the JFIF Implementation of it.
We will first discuss the various steps involved in this process and then move on to giving details about the FPGA Implementation. Section II will discuss the various Encoder steps. Section III contains details regarding the various HDL blocks that will implement the various steps mentioned in Section II. Section IV will then elaborate on the limitations of the implementation and the future work involved.

II. JPEG Compression Algorithm

Digital Images are predominantly captured on the RGB spectrum. So, the camera will output raw pixel values in a 3D matrix whose dimensions will depend on the sensor resolution. For example, a 1920 x 1080 sensor will output a 1920 x 1080 x 3 matrix. This will the input for our Encoder.

A. *Affine Transform*

The first step in the compression process is an affine transform. This step converts the RGB data to YCbCr spectrum. Here, Y is luminance and Cb, Cr are chrominance values which represent 2 coordinates in a system which measures the nuance and saturation of the colour. Let us name this step as “Affine Transform”. The exact methodology of the transform will be discussed in Section III.

B. *Sampling and Level Shifting*

The next step in the process will be sampling and level shifting. The JPEG standard takes into account the fact that the eye seems to be more sensitive at the luminance of a color than at the nuance of that colour. So, on most JPGS, luminance is taken in every pixel while chrominance value is taken as mean value of a 2 x 2 block of pixels.

This value is arbitrary and can be changed depending on the function. These sampled values are currently in the system as 8-bit unsigned values. They are converted to an 8-bit signed representation by “level shifting” them. This is done by subtracting 128 from their value.

C. *DCT Transform*

Moving on, the image is them broken into 8 x 8 blocks of pixels. This block then undergoes a Discrete Cosine Transform operation. If the X dimension of the original image is not divisible by 8, the encoder will make it divisible by completing remaining right columns with the right most column of the original image.

Similarly, if the Y dimension is not divisible, the remaining lines are completed with the bottom-most line of the original image. The 8 x 8 blocks are processed from left to right and from top to bottom. The purpose of the DCT transform is that instead of processing the original samples, the encoder works with the spatial frequencies present in the original image. These spatial frequencies are very related to the level of detail present in the input image. High spatial frequency corresponds to high levels of detail, while lower frequencies correspond to lower levels of detail.

The mathematical definition of DCT is as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad k = 0, \dots, N-1.$$

Using the above formula is computationally very hard to implement. So, a faster algorithm called AA&N [3] is usually used. Let us name this step as "DCT Transform".

D. Zig-Zagging

After performing the DCT transform, the next step is the zig-zagging on the coefficients in 8 x 8 matrix block. The zig-zagging pattern is as follows:

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

After traversing the zig-zag, we now have a vector with 64 coefficients (0, 63). The zig-zagging is done so that we get the ascending order of spatial frequencies. So now, we have a vector sorted by the criteria of spatial frequency. The first value in the vector corresponds to the lowest spatial frequency present in the image. This is called the DC term. This step can be named as "Zig-Zagging".

E. Quantization

Now that we have a sorted vector, we can move on to the next step. This step involves quantization. Each value is divided by a dividend specified in the "Quantization table" and then rounded off to the nearest integer.

The Quantization table has been mentioned in Appendix-A. This table is based on "psychovisual thresholding". Most existing encoders use simple multiples of this example. This process has a key role in JPEG compression. It removes high frequencies present in the original image. This is acceptable because our eye is much more sensitive to lower spatial frequencies than to higher frequencies. Most images have a lot of information stored in low spatial frequencies, so the drop in detail is unnoticeable. As a consequence of the Quantization step, the spatial frequency vector now has a lot of consecutive zeroes.

F. *Run Length Encoding*

The next step in the encoding process in Zero Run Length Coding (RLC). The quantized spatial frequency vector that has a lot of zeroes is encoded using run length coding. The encoding of the DC term will be discussed later. The rest of the vector is encoded using RLC. This step also includes a EOB marker at the end of the vector. The last block also has an End of File (EOF) marker. This implementation uses FF as an EOF marker.

G. *Huffman Encoding*

The final step in the Encoding process is the Huffman coding. Instead of storing the actual value, the JPEG standard specifies that we store the minimum size in bits in which we can keep that value (called the category of the value) and then a bit-coded representation of that value. The table depicting the relation between values and their categories is given in Appendix-A. After this representation is found, it is then Huffman encoded. The Huffman Encoded bit stream is then stored in the JPG file as a stream of bits.

Now, let's discuss the encoding of the DC term. The DC term is the coefficient in the quantized vector corresponding to the lowest frequency in the image. The authors of the JPEG standard noticed that there's a close connection between DC coefficients of consecutive blocks. So, the standard specifies that the DC encoded as the difference between consecutive terms. The difference values are then Huffman encoded.

The Huffman Encoded DC term differences are then added to the beginning of the bitstream. The resulting bitstream is stored in a .jpg file. A JPEG decoder has to be employed to open such a file. This concludes the discussion on the various steps

III. FPGA Implementation

An end-to-end JPEG Encoder and the accompanying testbench has been developed. The top-level model has two blocks- fifo_out and ff_checker.

The fifo_out module contains the circuitry that will produce the required JPEG bitstream while the ff_checker keeps track of control signals and checks for FF which is the End of File (EOF) marker. Given below is an RTL synthesis figure of the Top-Level Model:

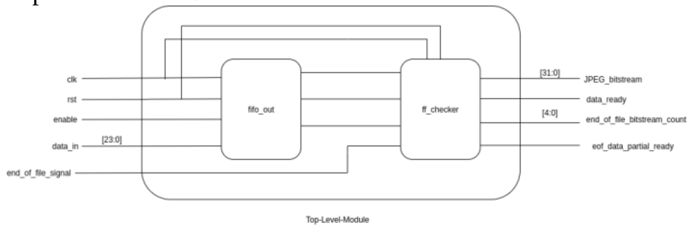


Fig. 1 Overview and Connections of Top-level Module

So essentially, the fifo_out module will perform all the steps discussed in Section II.

This module performs all the manipulations on the input image and gives a JPEG bitstream along with necessary information for controlling the end First-In-First-Out (FIFO) module. This module has 5 inputs- clock signal, reset, enable, data input and EOF signal.
The JPEG bitstream is output on the signal JPEG_bitstream, a 32-bit bus. The first 8 bits will be in positions [31:24], the next 8 bits are in [23:16], and so on. The data in JPEG_bitstream is valid when the signal is data_ready is high. data_ready will only be high for one clock cycle to indicate valid data. On the final block of data, if the last bits do not fill the 32-bit bus, the signal eof_data_partial_ready will be high for one clock cycle when the extra bits are in the signal JPEG_bitstream. The number of extra bits is indicated by the 5-bit signal end_of_file_bitstream_count.
The data input signal is 24 bits long. This bit sequence has information regarding the 3 colour schemes of each pixel. Each scheme of RGB has 8 bits of data embedded in the 24-bit sequence. This blackbox has two main components- FIFO and pre-FIFO.

A. FIFO Out

The implementation has a serial input which means that a data accumulation is required to perform operations on a block of data. This module essentially accumulates data coming from the pre-FIFO module and transmits to output of fifo_out in chunks of data that correspond to bit sequence of an 8 x 8 block. The functioning of this module is based on the computer network concept of First-In-First-Out. The data transmission is controlled by an enable signal. The first data chunk to come into this module is the first one to go out. In essence, this acts as a data buffer/data check dam. It takes values from the Huffman blocks (discussed later) and stores them in 3 synchronous 32-bit FIFOs. These streams are used to generate the final JPEG bitstream

B. Pre-FIFO

This module is the heart of this project. It performs all the required operations on the incoming pixel data and generates the JPEG bitstream.

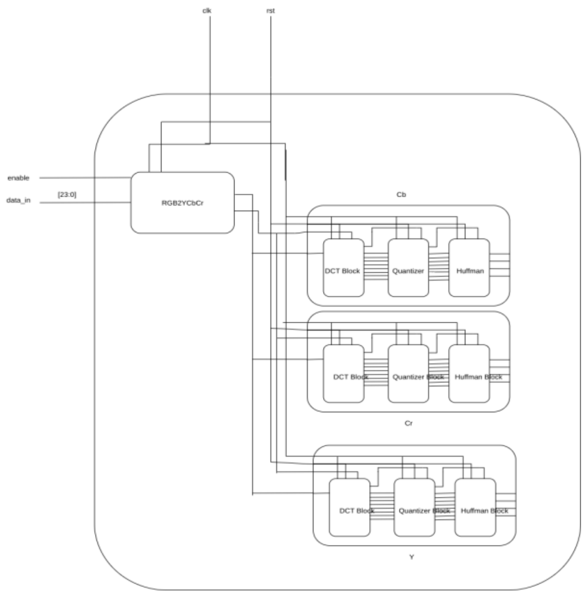


Fig. 2 Overview of Pre-FIFO Module

This module is made up of two blocks-

1. *RGB2YCbCr*

This module takes in input RGB data from the top-level module and converts it to YCbCr data in the manner specified in Section II.

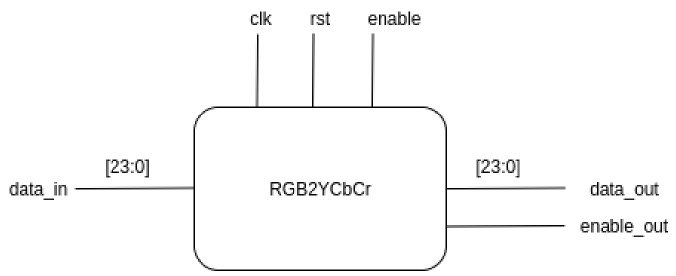


Fig. 3 Overview of RGB2YCbCr Module

The operation is based on the following formulas:

$$Y = .299 * Red + .587 * Green + .114 * Blue$$
$$Cb = -.1687 * Red + -.3313 * Green + .5 * Blue + 128$$
$$Cr = .5 * Red + -.4187 * Green + -.0813 * Blue + 128$$

2. Math Module

The Math module performs the remainder of the steps mentioned in Section II. This includes sampling and level shifting, DCT transform, Quantization, Zig-Zagging, RLC and Huffman Encoding including the DC term encoding. This module takes in YCbCr data stream of length 24-bits. The data is arranged as {Cr, Cb, Y}. So, [7:0] of the input data is directed to the Y Math Module, [15:8] is directed to Cb Math Module while [23:16] will be directed to Cr module.

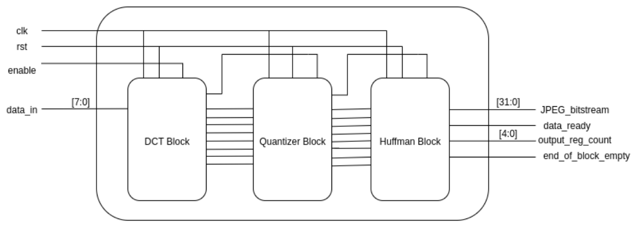


Fig. 4 Overview of the Math Module

Each block contains 3 separate modules-

- *DCT Block*

This module performs sampling and level shifting and DCT transform using the AA&N algorithm [3]

The output line bus implies an 8 x 8 matrix of data. Each line in Fig. 5 represents a bus 8 lines. Each line denotes a row in the 8 x 8 matrix. This data is then fed into the Quantizer block

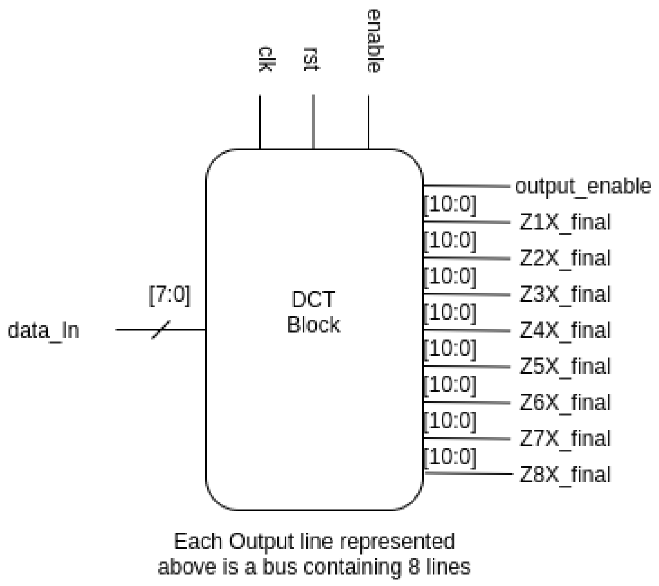


Fig. 5 Overview of the DCT Block

- Quantizer Block

This block takes in the 8 x 8 matrix data along with an enable signal to perform the Quantization operation on the input data. The output is also an 8 x 8 matrix which then moves to the Huffman Block to undergo further steps

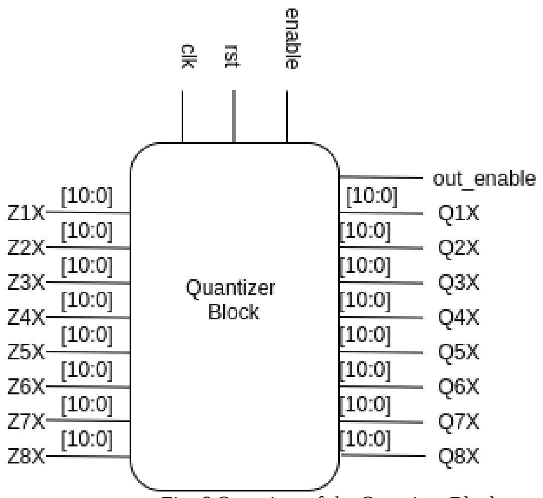
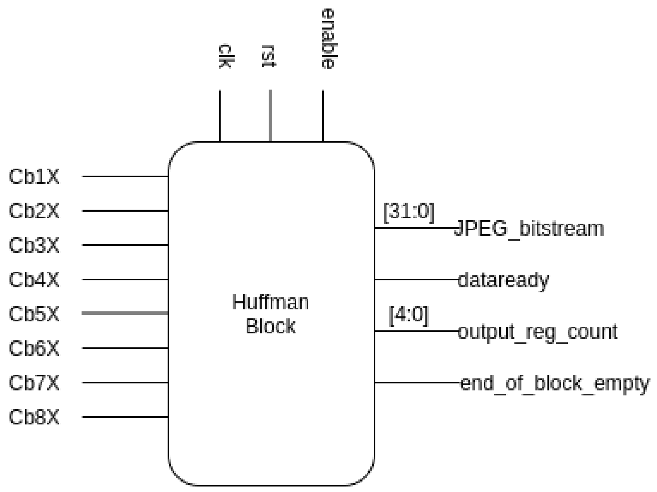


Fig. 6 Overview of the Quantizer Block

- Huffman Block

This module performs a multitude of operations on the 8 x 8 input matrix. First, a zig-zagging operation is performed. This will convert the matrix into a sorted vector of spatial frequencies in increasing order. Then RLC and Huffman Encoding is performed on the vector to generate the Huffman block outputs.



Each input line is a bus containing 8 lines.

Fig. 7 Overview of the Huffman Block

C. Testbench

A testbench for the model was also developed. This program feeds in RGB pixel data one after the other in blocks of 8 X 8. A picture whose rows and columns which are divisible by 8 has been chosen. I had to borrow a MATLAB script to write the functions necessary for inputting the data as it was too tedious to write input instructions manually. The testbench code is around 19 thousand lines.

All programs and HDL codes described till now are uploaded on my GitHub profile: https://github.com/krishna-swaroop/Image-Compression-Algorithms.git in the JPEGEncoder folder as well as mailed to the evaluator. The repository also contains programs to view Hyperspectral Data as was discussed in the MidSemester report.

This concludes the description of the various modules required to perform the steps necessary for JPEG compression.

IV. Limitations and Future Work

A. Limitations

The implemented algorithm is lossy due to finite accuracy in the quantization step. Also, various approximations and rounding off had to be done to implement the entire Encoder on a single FPGA board. More precise values will require greater computational power.

Due to Covid-19, the campus had to be shut down on March-15th. I managed to complete the HDL code after returning to my hometown. Since I don't have access to an FPGA here, various experiments to find out values of power consumption, compression ratio etc. couldn't be done.

B. Future Work

Future work will include running the HDL code on an actual FPGA and conduct the various experiments mentioned above. Other work may include attaching a software/hardware run Decoder to establish a complete data transmission system. But I would like to limit the scope of this SOP to just development of the Encoder

V. Conclusion

An FPGA based JPEG encoder and accompanying testbench have been developed over the course of a semester from January 2020 to April 2020. All underlying concepts and complexities of the implementation have been explored in this report. The HDL code written in Verilog for this project can be run on any FPGA with enough gates to realize the code. This has been developed with no specific IP core in mind.

The testbench gives 'ja.tif' (file specified in the GitHub repo) as an input to the top-level module of the JPEG Encoder core and checks if it receives the correct JPEG bitstream.

REFERENCES

[1] W.B. Pennebaker, J.L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.
[2] G.K. Wallace, "The JPEG still picture compression standard", Communications of the ACM, Vol. 34, No. 4, pp. 30-44, April 1991.

[3]   Miodrag Popovic and Tomislav Stojic, *The Fast Computation of DCT In JPEG Algorithm*
[4] S. Sanjeevannanavar and A. N. Nagamani, "Efficient design and FPGA implementation of JPEG encoder using verilog HDL," *International Conference on Nanoscience, Engineering and Technology (ICONSET 2011)*, Chennai, 2011, pp. 584-588, doi: 10.1109/ICONSET.2011.6168038.

[5] Palnitkar, Samir. *Verilog HDL: a guide to digital design and synthesis*. Vol. 1. Prentice Hall Professional, 2003.

[6] Schalkoff, Robert J. *Digital image processing and computer vision*. Vol. 286. New York: Wiley, 1989.

[7] Kusuma, Enas Dhuhri, and Thomas Sri Widodo. "FPGA implementation of pipelined 2D-DCT and quantization architecture for JPEG image compression." *2010 International symposium on information technology*. Vol. 1. IEEE, 2010.

[8] Agostini, Luciano Volcan, Ivan Saraiva Silva, and Sergio Bampi. "Pipelined fast 2D DCT architecture for JPEG image compression." *Symposium on Integrated Circuits and Systems Design*. IEEE, 2001.