Implementing JPEG Encoding on an FPGA

Elizabeth Rabenold, Jordan Wilke
Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology
Email: {rabenold, wilke18}@mit.edu

***Abstract*—This work presents a JPEG image encoder written in System Verilog for deployment on an FPGA. An image is fed in over a daughter board camera module, downsampled, then sent through the JPEG encoding pipeline. Notable features implemented in this system include the Discrete Cosine Transform, Run-Length Encoding, and Huffman Coding. An FPGA is selected for implementation due to innate performance advantages — namely the parallelization, customization, and optimization that can be achieved on the hardware. The encoded image data is sent off-board via UART and saved using the JFIF standard in Python to verify operation.**

I. Introduction

In the world of digital media, memory is a key constraint. Storage space is limited, and uncompressed images can take up too much space. We can use compression algorithms to reduce the size of the image file, allowing us to store the image more efficiently and process or transmit it faster. One widely-used and nearly ubiquitous compression technique is JPEG, which can achieve file compression rates of up to 20x. Though some of the original image data may be lost in the compression process, what is lost is most often imperceptible to the human eye, and the memory and performance benefits are worthwhile. The JPEG compression algorithm has some computationally heavy components, including the discrete cosine transform and run-length coding. An FPGA is a convincing platform for a JPEG compression system due to its modularity, ability to parallelize, fast processing speeds, and flexibility. The following paper describes a JPEG compression system implemented in System Verilog written for the fall offering of 6.S965 at the Massachusetts Institute of Technology.

II. Methodology

A. Input and Frame Buffer

An input image of dimensions 1280x720 pixels encoded as 16-bit 565 RGB is fed into the system on a clock frequency of 200MHz, then downsampled such that every fourth pixel is stored into a two-port BRAM serving as a frame buffer. The frame buffer has dimensions 320x180 with a color depth of 16, for an overall size of 115.2KB. The frame buffer is then read out at a clock speed of 100MHz for downstream processing.

B. Matrix Generation

Since JPEG uses the DCT as one of its main forms of compressing information, we separate the pixels of the image into 8x8 matrices that the DCT algorithim expects such that locally similar pixel frequencies can be encoded. Since the image does not fit perfectly into 8x8 blocks, the pixels at the bottom are repeated for the purposes of the algorithm such that the entire image is broken into 920 (40x23) 8x8 blocks.

C. Color Conversion

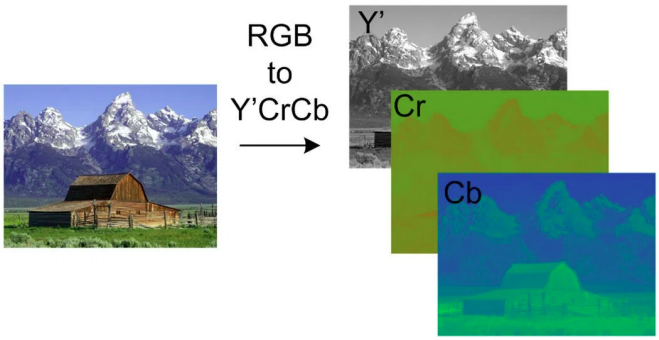

Fig. 1: An RGB image separated into luminance and chrominance channels for JPEG compression.

Each 8x8 matrix is fed into a module to convert each 565 RGB pixel encoding (bit depth 16) into separate luminance (Y) and chrominance (Cb, Cr) channels. YCbCr color conversion provides improved compression efficiency over RGB encoding by separating an image's brightness and color information. The Y channel retains most of the original image's detail, grayscaled, while the Cb and Cr channels carry only color difference information, as seen in Fig. 1. Given that humans are more sensitive to brightness than to color details, most of the data is in the luminance channel, meaning that the chrominance channels are quantized more aggressively. The sparser chrominance coefficient matrices mean that the Cb and Cr channels can be more heavily compressed relative to the Y channel without a significant drop in perceived image quality. This process is often refered to as downsampling (or chroma subsampling), with a typical JPEG downsampling of 4:2:0 which refers to reducing the chrominance in the horizontal and vertical directions by a factor of 2. The resulting values for each component are typically stored as 8-bit numbers, and to help with the encoding, they are centered around zero. In the first iteration of this system, the chrominance channels are discarded, and only the luminance values are propagated downstream. Future extensions of this work will work to preserve all three channels by implementing improved parallelization strategies and better memory management.

D. Discrete Cosine Transform

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where

- u is the horizontal spatial frequency, for the integers $0 \leq u < 8$.
- v is the vertical spatial frequency, for the integers $0 \leq v < 8$.
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases}$ is a normalizing scale factor to make the transformation orthonormal
- $g_{x,y}$ is the pixel value at coordinates (x, y)
- $G_{u,v}$ is the DCT coefficient at coordinates (u, v).

Fig. 2: The DCT equation for an 8x8 matrix, also known as a type-II DCT.

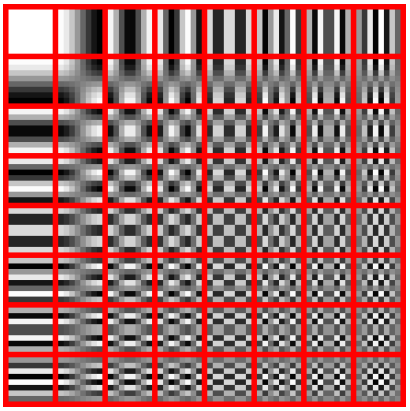

Fig. 3: The DCT transforms an 8×8 block of input values to a linear combination of these 64 patterns, known as the basis functions.

After converting each image into its constituent components, the 8x8 matrics for each component, here just the luminance component, are converted using the Discrete Cosine Transform (DCT), a type of fourier transform to convert the luminance values into the frequency domain. The exact equation for the DCT is given in Fig. 2, showing that each resulting entry in the 8x8 matrix after the transform is a linear combination of the information of all of the other entries in the matrix. Notably Fig. 3 shows how information becomes broken down as a linear combination of the information in each of these patterns. We make efficiency gains by using the symmetry of the cosine function and the fact that in this equation, the cosines will only range between 32 values. We precompute these as well as the results of the alpha functions. We use 10-bit fixed-point approximations noting that this will scale our final result by 30 bits which we can downshift by after all of the computations have been completed.

E. Quantization

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

Fig. 4: A prototypical quantization matrix representing a quality of 50% as specified by the JPEG standard.

The quantization matrix is used to compress each 8x8 matrix taking advantage of the fact that people are not able to notice differences in high frequency components as well as they can for low frequency components. By doing this, we are able to round many of the high-frequency components to 0. This quantization process, specifically the rounding, are the lossy parts of the JPEG compression. A quantization matrix of all 1s could be used for lossless encoding. To help with the performance of our implementation, we convert each of the quantization entries into their reciprocals, again using 10-bit fixed-point values, allowing us to multiply rather than divide the values which can be done in roughly 1 clock cycle for small enough outputs of the DCT. Fig. 4 shows an example quantization matrix typically used for compressing the luminance component. Since people are less susceptible to the compression of the chrominance components those quantization matrices usually include larger numbers even for the earlier entries. The JPEG standard allows multiple quantization matrices to be used for different components.

F. Run-Length Encoding

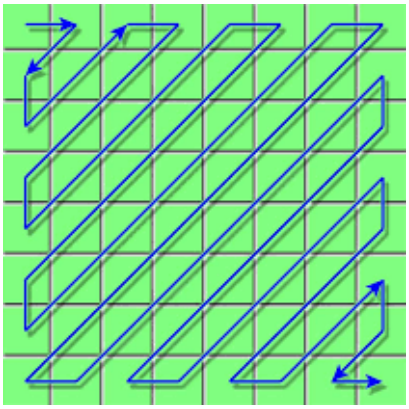

Fig. 5: Zig-Zag matrix traversal performed on the 8x8 matrix prior to Run-Length Encoding.

Run-length encoding (RLE) is used to losslessly compress the coefficients of the quantized DCT matrix. High frequency values from the discrete cosine transform will tend to have

a near-zero or zero coefficient. Furthermore, the DCT pushes entries with lower frequency into the top-left of the output matrix, while the high frequency entries fall to the low-right. First, zig-zag encoding (Fig. 5) is used to flatten the 2-D array in such a manner that captures the most significant low-frequency coefficients first. This is done in one combinational step, so latency is low. Run-length encoding then traverses the zig-zag array, tracking the 'run-length' of adjacent entries that share the same value. Instead of maintaining separate entries per each instance of a value, the encoding algorithm pairs each value with its run-length. This is particularly beneficial for further compressing the quantized DCT matrix, as most entries will be zero. Figure 6 demonstrates the output of the RLE compression module. For a 1-D array of length 64 of quantized DCT coefficients, the length of the encoding is reduced to 19 entries, demonstrating the efficacy of RLE encoding.

```
====
Original array is length 64:
[5, 1, 0, 3, 3, 3, 4, 3, 5, 5, 0, 3, 0, 0, 5, 2, 0
, 3, 1, 4, 4, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
====
RLE output is length 19:
[(5, 1), (1, 1), (0, 1), (3, 3), (4, 1), (3, 1), (
5, 2), (0, 1), (3, 1), (0, 2), (5, 1), (2, 1), (0,
 1), (3, 1), (1, 1), (4, 2), (2, 1), (1, 1), (0, 4
1)]
====
```

Fig. 6: The RLE module applied to a length 64 array.

G. Huffman Coding

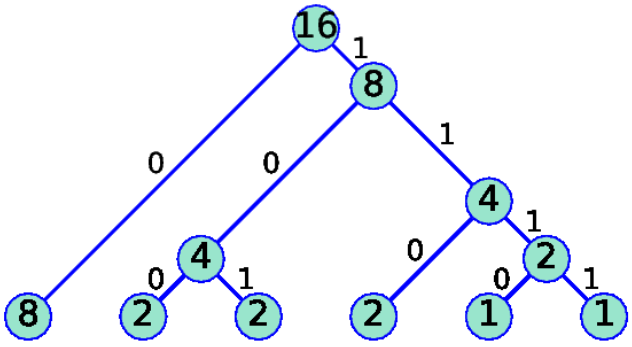

Fig. 7: Part of the Huffman Table for AC luminance coefficients. [6]

Once a matrix has been RLE encoded, each value run-length pair is converted into its Huffman code representation. The Huffman algorithm encodes each RLE symbol as a binary string. The designated string for each symbol can be selected in two ways. First, the matrix can be flattened into one dimension, sorted, then transformed into a Huffman Tree that tracks node neighbors, value, and run-length. Following the tree's construction, labels can be applied to each branch, and by traversing outward from the root, the binary strings for each symbol can be constructed. The authors began this approach and implemented a brick sorting algorithm and tree generation module, but ultimately elected to pursue a different method.

Table K.5 – Table for luminance AC coefficients (sheet 1 of 4)

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
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110

Fig. 8: Part of the Huffman Table for AC luminance coefficients. [1]

The alternative to generating the Huffman Tree locally is to use existing Huffman encoding tables, which have been developed from averaged statistics of a large set of images [1]. This is implemented in the system as a look-up table, and each run/size length is passed in to combinationally find its code word, as in Fig. [8]. As previously stated, in the first implementation of this system, the authors elected to only use the luminance channel because it carries the significant portion of the image data. As such, a chrominance look-up table was not implemented, but one exists. After all of the binary codes have been collected, they are flattened together into a 1-dimensional array that has a maximum size of 1024 bits, but likely much less considering the compression achieved by RLE.

H. Verification

The Huffman encoded image is sent off-board via a UART Transmitter module. This module delivers each encoded byte to the computer which then uses a Python script to save the encoded data into a JPEG file by appending the requisite header information and quantization and huffman tables so that the data can be decoded by any system.

Cocotb[8] and Cocotb[9] were used for testing and verification of the System Verilog modules via Python. Fast Signal Trace files were examined with a waveform viewer.

III. Results

The final project alas did not come to fruition as we would've hoped. Trying to interpret and understand the 186-page specification for the JPEG standard and the file format was non-trivial and it wasn't until the end that we really felt we understood the full process. As mentioned in the Huffman Encoding section, we spent a lot of time writing and testing the

sorting and tree building, and then discovered a much easier way to implement the encodings.

What we did end up with was multiple verified submodules, including the DCT conversion, quantization, run-length encoding, and Huffman Encoding, now just needing to integrate them properly. Additionally we have a fully working JPEG decoder in Python and a JPEG encoder for saving the final data to allow for testing that we meet the specifications of the standard.

Opportunities for extension obviously include full-scale integration of each of the verified modules. Additionally, we had hoped to take in videos and do MJPEG compression once we got JPEG working. Future directions would also include using the chrominance channels and full downsampling.

IV. Conclusion

Overall this project was very tough for us during a tough part of our year, but we were able to learn a lot from it. The JPEG standard, while being widely used, is a non-trivial implementation in hardware. It combines many simple ideas of different compressions and ways to save space without losing any perceived quality in an image. We also both learned to read through dense technical specifications and got practice with verifying our hardware solutions, prior to even writing them, to try and guarantee they work.

V. Codebase

The code for this project is available on Github at: https://github.com/rabenold/6sfinal_proj

References

[1] https://www.w3.org/Graphics/JPEG/itu-t81.pdf

[2] https://www.thewebmaster.com/jpeg-definitive-guide/

[3] https://www.geeksforgeeks.org/process-of-jpeg-data-compression/

[4] https://users.ece.utexas.edu/~ryerraballi/MSB/pdfs/M4L1.pdf

[5] https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/lossless.html

[6] https://en.wikipedia.org/wiki/JPEG

[7] https://asecuritysite.com/comms/dct

[8] https://cocotb-bus.readthedocs.io/en/latest/

[9] https://www.cocotb.org/