

FPGA Implementation of a JPEG Compression Pipeline

1st Sam Vinu-Srivatsan

Department of EECS

Massachusetts Institute of Technology
samhita@mit.edu

2nd Fadi Farag

Department of EECS

Massachusetts Institute of Technology
fadif@mit.edu

3rd Emilia Szczepaniak

Department of EECS

Massachusetts Institute of Technology
eszczep@mit.edu

Abstract—We present a series of hardware-accelerated modules ready to be inserted into a JPEG image encoder pipeline implemented on the Xilinx Real Digital Urbana FPGA. The system contains some Python parts for streamlined color-space conversion and input-matrix generation—where each matrix represents an 8×8 block in an image frame, called a minimum coded unit (MCU)—and output Huffman encoding. It also contains multiple modules in Verilog that perform blockwise 2D discrete cosine transform (DCT), quantization, zig-zag scanning, and run-length/run-delta compression. The eventual integration of these modules for JPEG compression use would support streaming pixel input from the OV5640 camera sensor into and out of the Double Data RATE 3 (DDR3) off-chip memory, buffering lines of camera pixels into 8×8 BRAM blocks for DCT, and the utilization of BRAM look-up tables to store Huffman and quantization coefficients. The encoder implements quantization tables, and entropy-coded segments, and mathematically complex cosine transforms directly in logic, achieving deterministic latency and high throughput. We plan to evaluate the performance, compression ratio, and image quality of these in-hardware modules across FPGA LUT/DSP usage and discuss future improvements for smooth integration.

Index Terms—JPEG, Huffman, entropy encoding, DCT, compression, FPGA, Hardware Acceleration

I. PROJECT OVERVIEW AND GOALS

Compressing many files into a JPEG format can be a time-intensive process. Our project aims to boost throughput and accelerate bottleneck stages of the compression process via hardware acceleration. We focus on accelerating the Discrete Cosine Transform (DCT), quantization, zig-zag ordering, run-length compression, and run-delta sequencing. The complexity of this mathematical pipeline makes the acceleration non-trivial. While significant parallelism can be easily achieved in a single dimension of the DCT, achieving optimal parallelism across all dimensions simultaneously is complex. Implementing computationally demanding mathematics efficiently in hardware using high-precision arithmetic increases circuit complexity, which can lower performance on FPGAs.

Our initial goals for this project were:

- Demonstrate output data via UART
- Demonstrate DCT on a 640px by 480px frame from the lab camera
- Demonstrate Huffman on a 640px by 480px frame from the lab camera

- Output a JPEG Entropy-Coded Segment or stream of image data (does not include headers due to unnecessary overhead)

II. SYSTEM ARCHITECTURE OVERVIEW

The JPEG encoder system on FPGA consists of the following major subsystems:

- **Input Interface:** Captures raw RGB or YCbCr pixel data from a sensor or memory. Converts RGB to YCbCr using fixed-point arithmetic. Input data is split into Y, Cr, and Cb channels, each of which are processed separately through the pipeline shown in Fig 1.
- **8×8 Block Line Buffer Module:** Using a modified line buffer, we can buffer pixels from our DRAM into more easily accessible BRAM 8×8 pixel blocks called minimum coded units (MCU).
- **2D-DCT:** Performs 2D DCT in 2 steps using fixed arrays, so the only operations are addition and multiplication. This operation sorts our column values into a matrix sorted by frequency with the top-left corner having a frequency of 0 and the bottom-right corner having the largest non-zero frequency. Since the first top-left coefficient is 0, it will be referred to as the DC coefficient for the MCU. All other terms will be referred to as AC coefficients since they have a non-0 frequency.
- **Quantization Unit:** A quantization matrix is then applied to this 2D DCT matrix. This matrix significantly amplifies lower frequencies over higher frequencies. This is due to the way we perceive light and detailed versus high-level information. The exact quantization table to be applied will depend on what channel is being processed. The luminance channel and chrominance channels have different tables, each of which will be stored in BRAM.
- **Zig-Zag Scanner:** The quantized matrix will be mostly filled with zeroes, which are unimportant information. One way to achieve efficient compression is by condensing these zeroes. Before we can do this, we must first flatten our 2D matrix into a 1D array. This is done by appending elements from the matrix in a zig-zag pattern, which will be explained in further detail below.
- **Run-Length Encoder:** Our zig-zagged, flattened 1D array (representing one 8×8 MCU) is filled with many consecutive zeroes. Instead of storing all of them, we

implement a Run-Length Encoder that condenses this list into a list of run-value pairs, where a "run" is the amount of zeroes between two nonzero values and a "value" is the nonzero value that comes after a series of zeroes.

- **Run-Delta Encoder** For DC coefficients, we execute a run-delta encoding, which outputs the difference between our original DC coefficient and the DC coefficient of the first MCU.
- **Huffman Encoder:** Huffman coding compresses our data further by allocating smaller bit sizes to the most frequent inputs. The exact bit size and representation for each of our encoder values is given by one of four tables specified by the JPEG ISO [1]. These tables are also stored in BRAM.
- **Output Interface:** Streams our Huffman-encoded JPEG bytes to an external laptop for formatting and decoding via UART.

The total latency of this process is calculated to be approximately 80 cycles per MCU as follows:

- 2-D DCT: 10 cycles (upwards estimate of 7 cycles)
- Quantization: 2 pipelined combinational stages, 2 cycle
- Zig-Zag: combinational, 1 cycle
- Run-Delta and Run-Length: 63 cycles in parallel

III. INPUT PIPELINE AND COLOR PROCESSING

Fig 1 does not address the data acquisition process because we utilize previous coursework to acquire data from an OV5640 camera. We will use our work from Lab 5, reutilizing all modules from the left of the diagram shown in this block diagram by Steinmeyer et. al. up until and excluding `channel_select`. We will then replace the downsampling, BRAM frame buffer, and scaling logic modules on the left with a DDR3 memory and stacker interface from Lab 7 as shown in Steinmeyer et. al. diagram. After retrieving the Y, Cr, and Cb values from the `rgb_to_ycrCb` module, we will use a modified version of our `line_buffer` module from Lab 6 to buffer out eight columns of size eight, and feed them into the DCT pipeline. This 8×8 pixel "tile" is called a minimum coded unit (MCU) in JPEG compression.

IV. DCT AND QUANTIZATION PIPELINE

After the input image block is resized and converted to YCbCr, the JPEG encoder must transform each MCU from the spatial domain to the frequency domain. We do this via a DCT and quantization. Our current working Python prototype and our FPGA implementation use different algorithmic approaches, so some care must be taken during verification.

A. Discrete Cosine Transform

On hardware, we use a direct matrix-multiplication approach, explained below. The 2D DCT is used in JPEG to transform an 8×8 block of image pixels into frequency coefficients. It can be computed efficiently using a matrix multiplication approach.

1) *Step-by-Step Computation:* To compute the 2D DCT using the direct matrix method:

- 1) Retrieve a column from the line buffer output.
- 2) Compute an intermediate result by multiplying the column vector of the 8×8 block by the DCT matrix
- 3) Multiply each row of the intermediate result by the DCT matrix

2) *2-D DCT Matrix Representation:* Given the definition of a DCT matrix, the 2D DCT of an 8×8 block s_{xy} is:

$$S_{uv} = \frac{1}{4} C_u C_v \sum_{y=0}^7 \sum_{x=0}^7 s_{xy} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

3) *Explanation of Terms:*

- s_{xy} : input pixel value at row y and column x .
- S_{uv} : DCT coefficient at frequency (u, v) .
- C_u, C_v : normalization factors, $\frac{1}{\sqrt{2}}$ if index is 0, otherwise 1.
- $\cos \frac{(2x+1)u\pi}{16}$: cosine basis function for row x and horizontal frequency u .
- $\cos \frac{(2y+1)v\pi}{16}$: cosine basis function for column y and vertical frequency v .

[2]

4) *Range Considerations:* Each summand in the DCT formula has magnitude roughly s_{xy} multiplied by two factors whose absolute value ≤ 1 . Since there are 64 terms:

$$\max \text{sum} \approx 64 \cdot 2^8 = 2^{14} = 16384$$

After scaling by $\frac{1}{4}$, the maximum DCT coefficient requires about 12 bits. In practice, due to the $C_u C_v$ factor, the required dynamic range does not exceed 11 bits. For inputs in $[0, 255]$, coefficients remain below 2048, confirming 11-bit sufficiency.

Unlike FFT-based DCTs, the direct matrix approach uses the same scaling factors and coefficient normalization that the JPEG standard assumes, and aligns with the way hardware JPEG cores internally decompose the 2D DCT into two 1D stages.

The resulting coefficients exhibit small rounding errors from fixed-point arithmetic at around 0.5 LSB on average per coefficient. This is within JPEG compliance and comparable to existing hardware implementations. The 8-bit approach also uses fewer LUTs than higher-precision configurations. The pipeline latency is approximately eight cycles per 8×8 block, after which the core outputs eight coefficients per cycle.

B. Quantization

Following the DCT, the quantization stage scales each coefficient using tables stored in BRAM: two 64-entry, 8-bit tables for the luminance and chrominance components. Errors from division are avoided in hardware by doing an elementwise multiplication and bitshift.

C. Zig-Zag Scan

Finally, the 64 quantized coefficients are reordered using a hard-coded zig-zag scan table, implemented in BRAM, and serialized into the Huffman coding stage.

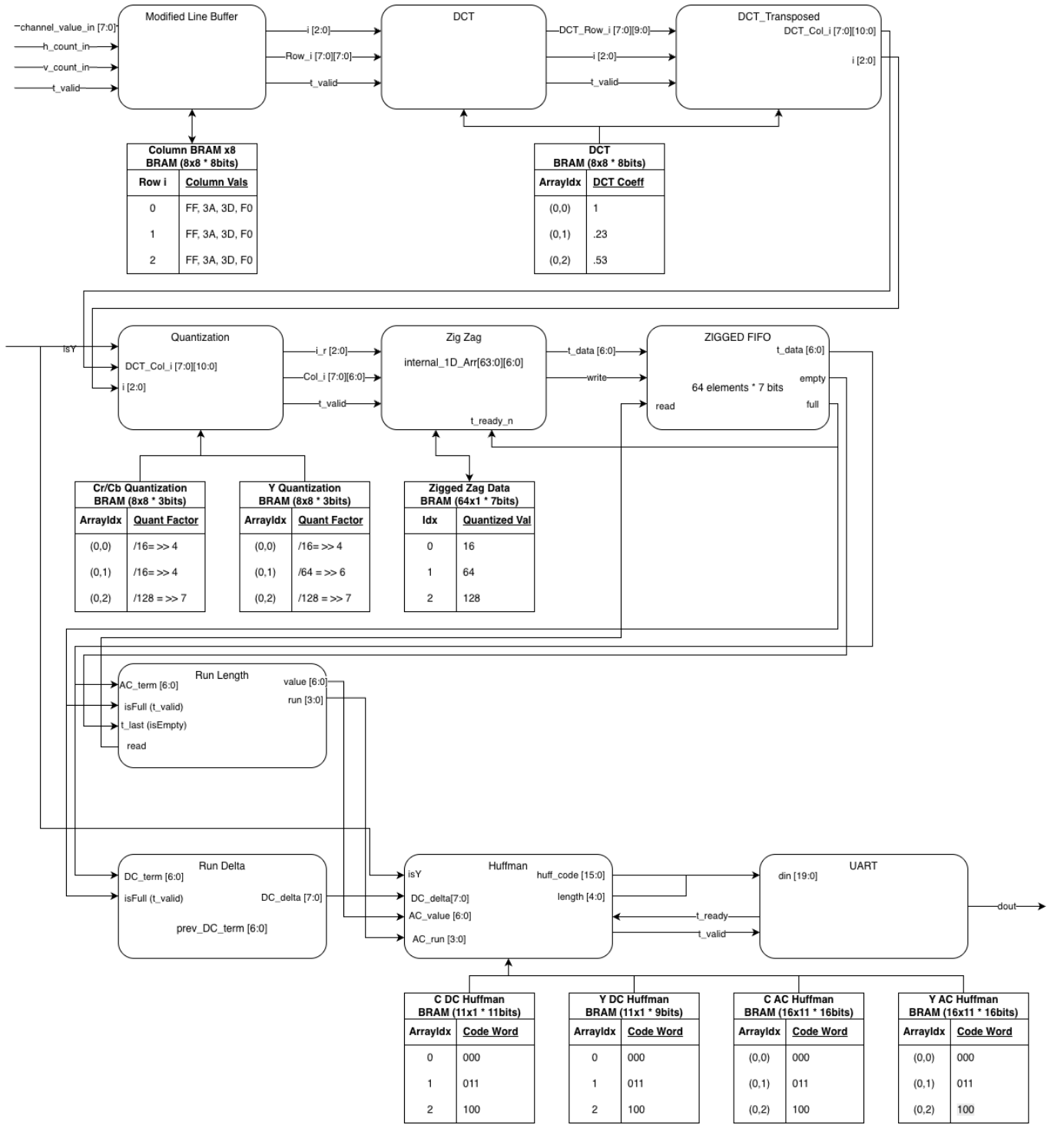


Fig. 1. Block Diagram per channel (Y, Cb, Cr)

V. ENTROPY ENCODER

The entropy encoding pipeline consists of three types of encoding called run-delta (RDE), run-length (RLE), and Huffman encoding. This pipeline takes in the output of the zig-zagged array (1D array with 64 values).

A. Run-Length Encoder

First, this array is passed into the run-length encoder. The run-length encoder takes an input of the lower 63 values, or the AC terms. Given this AC coefficient sequence (zeros included), we want to store it more efficiently. Instead of keeping every zero, we convert the sequence into (run, value) pairs,

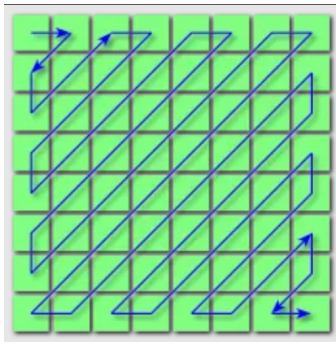


Fig. 2. Zig-Zag Matrix Scan, source: Wikimedia Commons

where `run` is the number of zeros before the next non-zero value, and `value` is that non-zero number.

1) *Small Example:* Consider the shortened AC sequence: $[0, 0, 5, 0, -3, 2]$ We process the sequence from left to right, yielding the following run-length encoded sequence: $[(2, 5), (1, -3), (0, 2)]$

2) *Bit Lengths*: The run can have a maximum value of 15, so, after calculation, they are stored as 4 bits each in a run FIFO. The values range from $[-127, 128]$, and are stored in 8 bits each in a value FIFO.

B. Run-Delta Encoder

The run-delta encoder (RDE) takes the DC term, which is the first value in the 1×64 array. The RDE calculates the difference between DC terms across different 8×8 MCUs. It does so by storing a previous DC term and computing and storing the delta between that and the current DC term.

After RDE, each 8×8 block is represented as a DC difference and a sequence of AC run-value symbols. In hardware, this stage formats these symbols into the precise categories required by the JPEG Huffman tables.

Each run-value pair is directly mapped into a single 8-bit symbol, where the upper nibble encodes the zero-run and the lower nibble encodes the magnitude category. Special symbols such as zero run-length (ZRL) (0xF0) and end-of-block (EOB) (0x00) follow the same structure. The hardware then passes these category codes to the Huffman block, which emits variable-length prefix codes for the bitstream.

C. Huffman Encoder

Huffman encoding uses BRAM-stored baseline JPEG tables for Y and C components. Each DC category or AC run-value symbol indexes its corresponding codeword, which the FPGA outputs into a variable-length bit-packer. The bit-packer serializes codes MSB-first, applies JPEG byte-stuffing (to distinguish between data and JPEG format markers), and aligns the final bitstream with marker boundaries.

VI. JPEG FILE FORMAT CONSTRUCTION

The final JPEG file is assembled in Python by joining marker segments and data. A marker segment is a specific, tagged block of data within a JPEG file that helps structure the

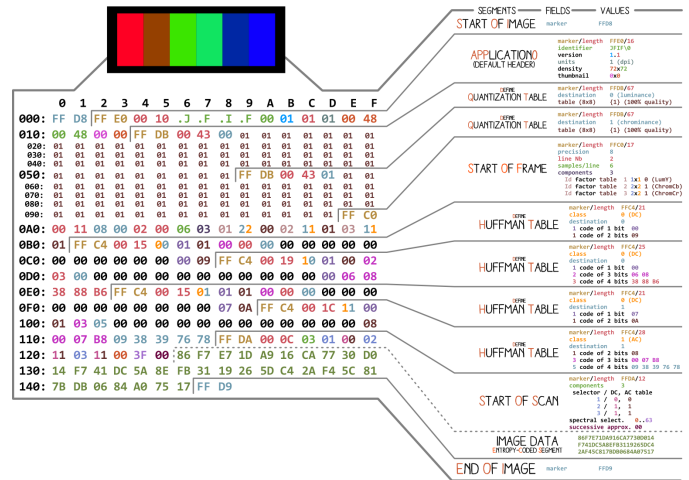


Fig. 3. JPEG File Format Markers and Data [3]

image information and associated metadata. These segments are two-byte codes (starting with `0xFF` followed by a unique identifier byte) that signal the type of data that follows in a JPEG file, allowing the decoding software to parse the file correctly or skip segments it does not understand.

Each data block after a marker is generated as a big-endian hexadecimal string. The Python code computes all multi-byte length fields and inserts component information based on the image width, height, and color structure. Because JPEG uses a simple byte-oriented, error-checking-free container, these segments are directly concatenated. After the headers are assembled, the entropy-coded bitstream produced by the field-programmable gate array (FPGA) is inserted after the Start of Scan marker to form a complete JPEG image.

Python also formats the Define Huffman Table segments by converting the luminance and chrominance Huffman dictionaries into the JPEG-required “bits” and “values” lists. These are packed into a single marker block for each table, containing the table class, destination identifier, code length counts, and all symbols. This ensures compatibility with the FPGA’s Huffman encoder output.

VII. RESULTS

A. Design Summary

- We wrote the following stages of this pipeline in Verilog, to accelerate on the FPGA:
 - 2D DCT
 - Quantization
 - Zigzag
 - Run Length
 - Run Delta
 - UART Tx
 - UART Rx
- We wrote the following stages of this pipeline in Python:
 - Huffman
 - UART Tx
 - UART Rx

Module	LUTs	DSP
Top-level	1242	8
DCT-y	1110	8
UART Transmitter	38	0
UART Receiver	68	0
Event Counter	18	0
Total		

TABLE I
RESOURCES USED FOR MODULES BUILT IN TOP LEVEL DESIGN

- Quantization (in case issues with merging arose)
- Zigzag (in case issues with merging arose)
- Run Length (in case issues with merging arose)
- Run Delta (in case issues with merging arose)

B. Evaluation of Design

We chose to qualify the latency and throughput of our modules by doing xyz. Some overheads that we incurred are xyz.

According to the `post_place_util.rpt` file generated by Vivado, 0 BRAMs, only 8 DSPs, and 315 slices were used. On a larger scale, if we were able to integrate the other modules in our top level, we predict our BRAM usage should truly be at least $10 * 16 * 4096$ bits. We believe the reason Vivado was able to bring it down to 0 was due to the fact that very few modules were used, making it easier for Vivado to optimize these away. This agrees with what the logs reported as well.

Our timing requirements were to meet i, and we met those by a margin of xyz.

VIII. DISCUSSION

The primary usecase for the FPGA-accelerated JPEG is live photography compression We designed the initial deliverables for our project to make it useful for most people who now watch and stream data in 720p. With a few additional changes such as x or y, we could expand to additional use cases, like z.

Minimum Goals:

Ideal Goals:

Stretch Goals:

IX. CONCLUSION

X. CONTRIBUTIONS

Link to repository: <https://github.mit.edu/6205F25/fa25-6205-team45>

The contributions of each author are as follows:

- Fadi: Code DCT-x, DCT-y. Code and debug DCT-uv module. Block diagram. Debug UART top level.
- Sam: Debug DCT-x and DCT-y. Code and debug Quantize. Research math for DCT. Code and debug UART HDL modifications and Python serializer. Block diagram. Write preliminary and final reports. Design performance evaluation. Designing/coding Huffman encoder. (which didn't make it to the end!)
- Emilia: Code, debug, test Zigzag module. Debug and test Run length module. Research and resolve any memory

or timing issues. Source matrices and data for ROM and different modules. Write Code Provenance report.

Our sincerest thanks to all 6.205 course staff members, but especially Hasan Yildiz, Yeabsira Hawaz, Joe Steinmeyer, and Elise Wingard, who have been instrumental in helping us iterate through project ideas and debug. We also credit Yasoob Khalid, whose website <https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python> explained JPEG encoding projects with Python-implemented examples.

REFERENCES

- [1] “Information technology – digital compression and coding of continuous tone still images – requirements and guidelines,” no. ISO/IEC 10918-1:1993(E), 1993.
- [2] D. S. Community, *What is the maximum value that can result from a 2d dct?* <https://dsp.stackexchange.com/questions/43588/what-is-the-maximum-value-that-can-result-from-a-2d-dct>, Accessed: Nov. 26, 2025, 2017.
- [3] Y. Khalid, *Understanding and decoding a jpeg image using python*, <https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/>, Accessed: Nov. 5, 2025, Jul. 2020.