JPEG Decoding Accelerator

Matthew Button mbutton | Kyle Park kylepark | Velu Manohar velu | Muhammad Khan mamankh | Sahil Vemuri sahilvnv

1. Abstract

JPEG is a widely used image compression standard, but the decompression process is computationally intensive and due to its widespread usage, provides a reason for having a dedicated module in a System-on-Chip (SoC) platform. This project introduces a hardware accelerator designed to efficiently perform JPEG decompression on a mobile System-on-Chip. The design integrates key stages of the decoding pipeline—including entropy decoding, coefficient reconstruction, inverse transforms, chroma upsampling, and color space conversion—into a pipelined, and almost multiplier free architecture. The accelerator was evaluated on a variety of JPEG images and demonstrated significant performance gains compared to software-based decoding, making it well-suited for embedded and mobile applications.

2. Introduction

JPEG is one of the most widely adopted standards for lossy image compression due to its ability to significantly reduce file sizes while maintaining acceptable visual quality. It is used extensively across mobile devices, digital cameras, web platforms, and embedded systems, making it a critical part of the global image processing pipeline [1]. While the compression process is usually performed offline or on high-performance servers, decompression must often happen in real time, especially on power-constrained devices such as smartphones, tablets, and Internet-of-Things (IoT) platforms.

JPEG decompression involves multiple computation-heavy stages, including entropy decoding, dequantization, inverse discrete cosine transform (IDCT), chroma upsampling, and color space conversion. These steps place a considerable load on general-purpose processors, especially in embedded contexts where performance, energy efficiency, and thermal budgets are tightly constrained.

To address these challenges, this work presents a hardware accelerator for JPEG decompression designed specifically for integration into mobile SoCs. The design implements a pipelined JPEG decoding architecture while maintaining low area and power overhead. Key components include:

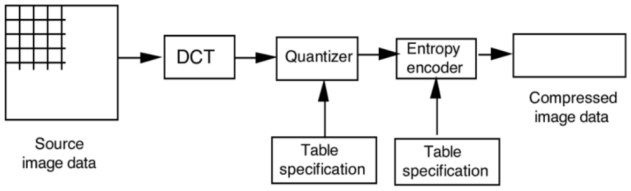

Figure 1. JPEG Encoding Process

- A high-throughput Huffman decoder that uses parallel bitmask matching and supports variable-length codewords,
- A multiplier-free 2D IDCT using Canonical Signed Digit (CSD) approximations and shift-add logic for efficient computation,
- A chroma supersampling module that outputs four upsampled blocks per cycle to match the resolution of luminance data, and
- A color conversion unit using CSD-based fixed-point arithmetic for real-time YCbCr to RGB transformation.

By implementing the full decoding pipeline in hardware, this accelerator enables faster image rendering and lower CPU usage, making it suitable for real-time video, camera preview, and image-intensive mobile applications. Experimental results demonstrate that the design achieves considerable speedup compared to software-based decoding such as MATLAB's imread() function. Figure 1 shows a high level diagram of the JPEG encoding process, which is what the decoder will perform the inverse of.

3. Survey of Previous Related Work

3.1. GPU-Based JPEG Decoding Using CUDA

Tade and Ansari [1] present a CUDA-accelerated JPEG decoder aimed at improving the performance of the decompression pipeline on general-purpose GPUs. Their approach focuses primarily on offloading the inverse discrete

cosine transform (IDCT), one of the most computationally demanding stages in JPEG decoding. By leveraging CUDA's thread-level parallelism, they implement an 8×8 IDCT kernel that processes DCT blocks concurrently across hundreds of GPU threads. Their implementation uses floating-point arithmetic and applies a standard separable 2D IDCT method, executing row-wise and column-wise transforms sequentially. The authors report substantial speedups when decoding large images, especially when compared to software-based decoding on CPUs.

While their results demonstrate the effectiveness of using GPUs for accelerating JPEG decoding, their approach targets desktop-class computing environments with relatively abundant power and thermal budgets. This makes the solution less suitable for resource-constrained embedded or mobile platforms, where power efficiency and predictable latency are critical. Moreover, the use of floating-point operations and reliance on GPU memory hierarchies introduces complexity and energy overhead.

In contrast, our work implements a hardware JPEG decoding accelerator in Verilog, optimized for integration into mobile SoC architectures. Rather than relying on floating-point units or massive thread parallelism, our design uses fixed-point arithmetic and shift-add-based logic to approximate multiplications via CSD representations. Specifically, our 2D IDCT pipeline is based on a modified version of Loeffler's algorithm, which eliminates multipliers entirely in favor of hardware-friendly additions and shifts, reducing both area and power. Unlike the CUDA approach that treats each DCT block independently on a massively parallel GPU, our design is deeply pipelined—capable of accepting a new block every cycle after initial latency, making it more suitable for real-time processing in streaming multimedia systems.

3.2. Accelerating JPEG Decompression on GPUs

Weißenberger and Schmidt [2] presented a GPU-based JPEG decompression architecture that exploits fine-grained parallelism inherent in block-based image processing. Their work demonstrates the feasibility of high-throughput decompression by leveraging the massively parallel processing capabilities of modern GPUs. The resulting implementation significantly outperforms baseline CPU decoders and even specialized libraries like NVIDIA's `nvJPEG`, especially for high-resolution images.

While GPU acceleration provides impressive throughput, it is not always ideal in embedded or resource-constrained systems due to power and thermal limitations. As such, hardware-based acceleration using FPGAs or ASICs remains a compelling alternative if the need for multimedia processing is high, offering predictable latency and lower power consumption. This project aims to explore such an alternative by designing a JPEG decoding accelerator in Verilog, focusing on low-level parallelization of the IDCT and Huffman decoding stages.

3.3. An FPGA-based JPEG Preprocessing Accelerator for Image Classification

In contrast, FPGA-based accelerators offer a promising alternative for efficient JPEG decoding in resource-constrained environments. Li et al. [3] proposed an FPGA-based JPEG preprocessing accelerator aimed at improving the throughput and energy efficiency of image classification tasks. Their design focuses on accelerating non-convolutional operations, including JPEG decoding, image block splicing, and scaling, which are often bottlenecks in end-to-end image classification pipelines. By implementing these preprocessing steps on an FPGA, they achieved a throughput of 875.67 frames per second and an energy efficiency of 0.014 J/frame on a Xilinx XCZU7EV FPGA. When integrated with an Inception V3 accelerator, the end-to-end system demonstrated a 28.27× speedup over CPU-based implementations and a 2.32× improvement in energy efficiency compared to GPU-based systems.

These studies highlight the potential of hardware accelerators in enhancing JPEG decoding performance. As a result, our project aims to develop a Verilog-based JPEG decoding accelerator suited for mobile SoC platforms. By focusing on hardware-level optimizations, we aim to achieve real-time JPEG decoding with minimal power and area overhead, making it suitable for embedded and mobile applications.

3.4. Improved Loeffler-Based 2D DCT/IDCT Hardware Acceleration

Zhou and Pan [4] present a hardware accelerator for 2D 8×8 DCT/IDCT operations, utilizing an enhanced Loeffler architecture. Their design features an 8-stage pipeline that optimizes the data stream of the Loeffler 8-point 1D DCT/IDCT, tailored for image and video processing applications. By employing fixed-point arithmetic and Canonical Signed Digit (CSD) encoding, the architecture achieves a multiplication-free approximation of DCT coefficients using only adders and shifters. A notable innovation is their fast parallel transposed matrix architecture, which efficiently handles row-column coefficient conversions with reduced circuit complexity. Implemented on a Virtex-7 XC7VX330T FPGA, the accelerator operates at 288 MHz, achieving a throughput of 558 million pixels per second and processing Full HD frames at up to 269 frames per second. The design completes 2D DCT/IDCT operations on 8×8 blocks in just 33 clock cycles.

In our project, we adapt this multiplier-free approach for the 2D IDCT, leveraging CSD-based approximations and shift-add logic to eliminate the need for multipliers. However, our design diverges in several key aspects. While

Zhou and Pan focus on a high-throughput solution suitable for high-resolution video processing, our implementation targets integration for low power consumption and minimal area overhead. Additionally, our architecture integrates the entire JPEG decoding pipeline—including entropy decoding, dequantization, chroma upsampling, and color space conversion—into a cohesive, low-latency system, while they only create an accelerator for the IDCT.

4. Description of Design

Each of the modules described in Figure 2 were implemented in Verilog. Below are the descriptions of the core modules:

4.1. Header Extraction

The encoded information of a JPEG is really folded into sections that constitute its header. Two byte markers indicate the start of a specific segment of data. These segments contain key information such as the image size, the sub sampling method, the quantization coefficient tables, and the Huffman symbols and lengths. From the onset we designed our accelerator to be passed a pure bit-stream over an AXI (Advanced eXtensible Interface) bus. We selected AXI in particular for its ubiquity particularly for FPGAs [5]. Hardware platforms with configurable FPGA modules could befit from on-the-fly JPEG acceleration. Much of the header decoding is a serial operation, but the structure of the header itself does not lend itself easily to hardware processing. As a prepossessing step we utilize a Python script that converts a JPEG image into a System Verilog (.svh) array of 32 bit lines. Our system simulates the passing of the JPEG bitstream in 32-bit (AXI compatible) lines by walking this preprocessed array. True implementations would perform this with DMA transactions coordinated by the CPU.

Reading the segments presents some difficulty because there is only a guarantee of byte alignment in the JPEG protocol, and there is a weak ordering of segments prior to the Start-of-Scan demarcation. Two byte markers can appear in four possible slots of the input lines or even cross the divide between two lines creating offsets in the data processing that propagate as we read in these tables and parameters. These marked segments are also variable length. For example, after witnessing a `0xFFC4` marker, there could be one, two, or as many as four Huffman tables that follow. Two distinct images that contain four Huffman tables might use a single or up to four separate markers requiring flexibility in our hardware implementation. We also attempt to be maximize efficiency and push a full 32-bits of our eventual scan stream into the accelerator. However, we are slightly inhibited by scattered instances of 'bit stuffing' markers that require delaying until we can pass a full line into the subsequent FIFO block.

Because we selected a very specific baseline JPEG protocol: ITU-T T.81 (1992) / ISO/IEC 10918-1 [6], we were able to simplify the state machine significantly. Guarantees of note include:

- *8-bit color precision*
- *Sequential (one-pass) encoding*
- *Huffman only codes (no arithmetic)*
- *A max of 2 AC and 2 DC tables*
- *Single SOS without restart markers*
- *4:2:0 Chroma sub-sampling*

Our header decoder allows for multiple images to be passed in continuously through the decoder. Tables are updated before a subsequent Start-of-Scan stream is pushed through the remaining modules. This presents an advantage for near contiguous JPEG workloads for example in applications for streaming or computer vision.

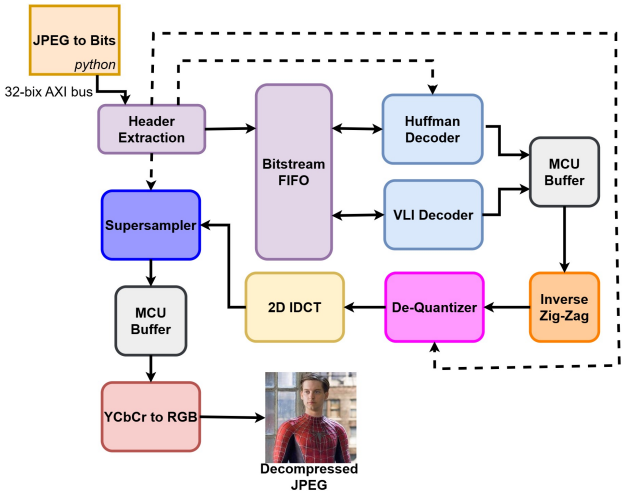

Figure 2. JPEG Decoding System Block Diagram

4.2. Huffman Decoding

After the header is read the symbols and lengths are passed through a Huffman modules to generate codes. This operation involves bit shifts and adds and is very quick as codes are constrained to under 16 bits and there are 256 or fewer symbols. As the Start-of-Scan stream comes in from the FIFO we examine 16 bits at a time using parallel look-ups against all Huffman codes loaded from the header. Each Huffman code has a corresponding length, and the decoder uses bit-masks to search for matches of different lengths against the current bit-stream prefix. Once a matching code is found, the decoder outputs the corresponding symbol from the Huffman table.

Every 8×8 pixel block (canonically deemed a minimum coded unit (MCU) in JPEG) starts with a DC term (intuited as the brightness of that MCU). This first term uses a delta encoding from preceding terms and is handled with simple subtraction. AC terms (for subsequent block entries) use variable length encoding (intuited as the spatial details of the JPEG). These AC terms each contain a run length (how many zeros precede the next non-zero value in the zig-zag scan), and a Variable Length Integer (VLI) size, which tells how many bits should be read next to form the actual value (amplitude) of the non-zero coefficient. The decoder uses this VLI to fetch the correct number of bits from the input FIFO for the VLI decoder, which reconstructs the original quantized DCT coefficient. These coefficients are then stored into a 64-element buffer, representing an 8x8 MCU.

4.3. 8x8 Block Buffer

The 8x8 block buffer functions as a circular FIFO that reconstructs a complete 64-coefficient block from run-length encoded JPEG data. First, it receives input from the Huffman decoder and VLI decoder, which provide a run-length and the corresponding coefficient value. Using a tail pointer, the buffer skips ahead by the run-length, effectively inserting that number of zeros into the output block. It then writes the decoded coefficient at the updated position. This process continues until either the buffer fills all 64 positions or an End of Block (EOB) symbol is received, which indicates that the remaining positions should be padded with zeros. Once either condition is met, the buffer outputs the full 8x8 coefficient block for dequantization.

4.4. Inverse Zig Zag

In JPEG encoding, the 64 DCT coefficients of an 8×8 block are arranged in a zig-zag order before compression. This ordering groups the low-frequency coefficients first (which carry most of the image's visual information) and places the high-frequency coefficients later, which are often zero after quantization. This pattern increases the effectiveness of run-length encoding (RLE) by clustering long runs of zeros together toward the end of the sequence. Consequently, during decoding, the 8x8 block needs to be "inverse zig zagged" to reverse the ordering, restoring the coefficients to their original 8×8 spatial positions. A hardware module implements this using a lookup table where each address corresponds to a position in the 1D zig-zag input and outputs the correct 2D (row, column) index in the 8×8 block.

4.5. De-quantization

The dequantization stage restores the scale of the DCT coefficients that were previously compressed during JPEG encoding. Each coefficient in the reordered 8×8 block is multiplied by a corresponding quantization factor retrieved from the quantization table. These quantization values vary by frequency component, with lower-frequency coefficients typically receiving smaller weights to preserve more detail.

To maintain hardware efficiency, the dequantization module is implemented using fixed-point arithmetic, with all operations designed to avoid multipliers where possible. This is achieved by encoding quantization table values using Canonical Signed Digit (CSD) representations, lowering power consumption and circuit complexity.

The module processes all 64 coefficients in parallel over multiple cycles, feeding the scaled output into the subsequent IDCT stage. Special care is taken to ensure that the bit width of the dequantized values accommodates potential overflow while maintaining sufficient dynamic range to preserve image fidelity.

4.6. 2D Inverse Discrete Cosine Transform (IDCT)

To perform the 2D IDCT, an improved version of Loeffler's algorithm was used [2]. Loeffler's algorithm uses 29 additions and 11 multiplications. The improved version increases the number of pipelined stages from 4 to 8. Figure 3 shows the pipeline for the improved Loeffler's algorithm. In addition, the multipliers are replaced by using Canonical Signed Digit Representation approximations of constant terms like cos(pi/8) and cos(pi/8) allowing for these computations to be done combinationally, only using adds and shifts. From the output of the 8x8 block in the dequantization, each row of 8 elements is fed into a 1D IDCT module using the improved loeffler's algorithm which requires 8 cycles to compute the output of the row. The output of each row is then gathered in another 8x8 arranged such that the output of each of the 8 rows are transposed and then each row is then fed into another 1D IDCT, which is used to compute the IDCT of each column. In total, an 8x8 input requires 33 clock cycles to compute. See Figure 4 for the 2D IDCT module pipeline.

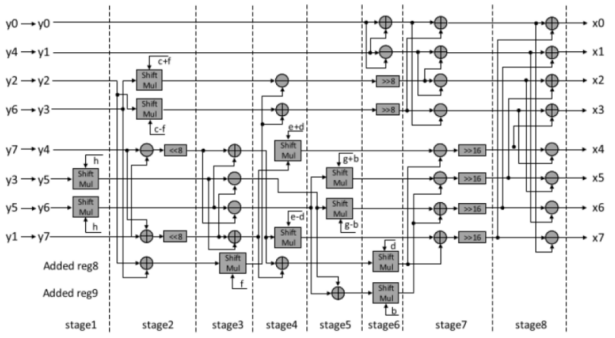

Figure 3. 1D IDCT Pipeline using improved Loeffler's Algorithm

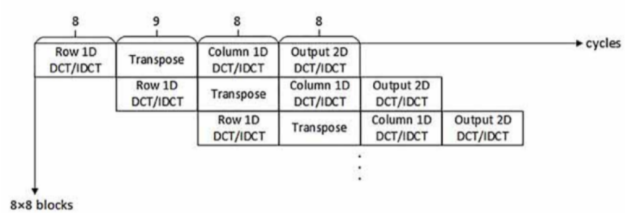

Figure 4. 2D IDCT Pipeline

4.7. Chroma Supersampling

During the JPEG encoding process, the chroma components (Cb and Cr) are stored at half the resolution of the luminance component in both horizontal and vertical dimensions (See Figure 5). While decoding, the Cb and Cr need to be brought back to full resolution so they can be aligned pixel-by-pixel with the Y data for proper color reconstruction.

The module upsamples each 8×8 chroma block into four 8×8 blocks. The supersampled chroma data is output as four channels per cycle—one for each of the upsampled blocks. These outputs are collected in a buffer along with the corresponding Y blocks to form full-resolution YCbCr data for downstream color conversion.

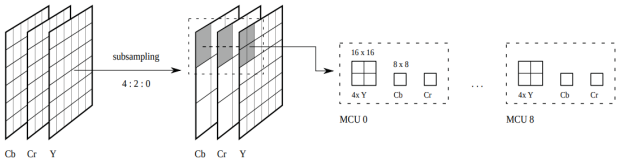

Figure 5. 4:2:0 Chroma Subsampling Example

To improve the visual quality of the upsampled chroma components, we implemented a bilinear interpolation module that performs full-resolution interpolation across the entire 8×8 output grid. Unlike the nearest-neighbor approach, which simply replicates chroma values, this module calculates each output pixel by blending the four surrounding input pixels using bilinear weights derived from their relative positions. The implementation avoids costly multipliers by leveraging simple shift-and-add operations, ensuring it remains hardware-efficient while producing smoother, more natural color transitions in the final image.

4.8. Color Space Conversion

Once full-resolution YCbCr blocks are available, they are converted to the RGB color space using integer approximation formula and CSD for final image reconstruction. Multiplications are implemented using shift-and-add operations, reducing the need for complex arithmetic units and maintaining hardware efficiency. This conversion enables the final RGB bitmap to be assembled and displayed.

5. Experimentation and Methodology

We tested our design using multiple JPEG images of different resolutions and compared the time to run MATLAB's imread() function on the image to simulation time of the accelerator with the chosen clock period after synthesis. Figure 6 shows a comparison of the decoded image using the accelerator and MATLAB.

Image	Dim.	Cycles	Hardware Time (s)	MATLAB Time (s)	Speed up	PSNR (dB)
spider-man	256x256	19243	0.000173	0.01762	101.74	28.21
tiger	900x599	366535	0.00330	0.016998	5.15	26.59
cat	1200x734	249763	0.00225	0.026119	11.62	24.56
nebraska	1280x800	339360	0.00305	0.022645	7.41	28.73

Table 1. Runtime and PSNR comparison between hardware decoder and MATLAB baseline. Hardware time calculated using a 9 ns clock period.

Figure 6. Comparison between decoded image using accelerator (left) vs MATLAB imread() (right)

Metric	**Value**
Area	11,834.7 μm^2
Total Power	623.9 µW
Clock Frequency	111.11 MHz

Table 2. Post-synthesis area, power, and clock frequency

6. Analysis of Results

Based on Table 1, the accelerator demonstrates significant speedup across all tested images. For smaller images, the speedup reaches nearly 100×, while for larger images, the speedup remains substantial at approximately 7.5×. In terms of output quality, the accelerator delivers acceptable results, with PSNR values consistently at or above 28dB—an important threshold noted in [3] as sufficient for deep learning applications. This slight degradation in PSNR is expected, as our design minimizes the use of multipliers, relying instead on addition and shift operations throughout most of the pipeline, except during the dequantization stage.

Unlike prior JPEG accelerators such as [1], [2], and [3], which report performance in frames per second (FPS), we

were unable to conduct such measurements due to timing constraints. However, our synthesis results in Table 2 provide insight into the accelerator's efficiency. Notably, when compared to [1], our design achieves faster decoding on a larger image. While they report a decode time of 11.72ms for a 600×522 image, our accelerator processes a 900×599 image (Tiger) in just 3.3ms.

7. Conclusion

We have presented a complete hardware JPEG decoding accelerator targeted at mobile System-on-Chip (SoC) platforms, where power, area, and latency constraints are particularly critical. The design integrates all major stages of the JPEG decompression pipeline—including entropy decoding, dequantization, 2D IDCT, chroma upsampling, and color space conversion—into a streamlined, pipelined architecture that avoids the use of multipliers where possible.

The post-synthesis evaluation (Table 2) demonstrates substantial performance gains over software-based decoding, with the accelerator achieving up to 100× speedup (Table 1) for small images and consistent improvements across a range of resolutions. Despite the use of approximate arithmetic for power and area efficiency, the design maintains image quality within acceptable bounds, with PSNR values suitable for visual applications and machine learning pipelines.

With a modest silicon footprint and low power consumption, our implementation is well-suited for real-time image processing in embedded and mobile systems. Future work will focus on extending the architecture to support streaming video, optimizing memory bandwidth, and validating the system on FPGA and ASIC platforms.

8. Contributions

See Table 3 for each teach members contributions.

Name	**Work Done**	**%**
Matthew Button	Huffman decoding, Table Extraction, VLI Decoding	20%
Kyle Park	IDCT pipeline support, Chroma Upsampling, RGB Conversion	20%
Velu Manohar	2D IDCT design, Testbench, 1D IDCT	20%
Muhammad Khan	2D IDCT design, Testbench, PSNR analysis	20%
Sahil Vemuri	Inverse Zig-Zag, De-quantization, MATLAB Decoder	20%

Table 3. Team Member Contributions and Percentage Split

References

[1] R. Tade and S. Ansari, "Acceleration of jpeg decoding process using cuda," *International Journal of Computer Applications*, vol. 120, no. 9, pp. 1–5, 2015.

[2] A. Weißenberger and B. Schmidt, "Accelerating jpeg decompression on gpus," pp. 121–130, 2021.

[3] T.-Y. Li, F. Zhang, W. Guo, J.-L. Shen, and M.-Q. Sun, "An fpga-based jpeg preprocessing accelerator for image classification," *The Journal of Engineering*, vol. 2022, no. 9, pp. 919–927, 2022. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/tje2.12174

[4] Z. Zhou and Z. Pan, "Effective hardware accelerator for 2d dct/idct using improved loeffler architecture," *IEEE Access*, vol. 10, pp. 101 101–101 111, 2022.

[5] R. Bhaktavatchalu, B. S. Rekha, G. A. Divya, and V. U. S. Jyothi, "Design of axi bus interface modules on fpga," in *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, 2016, pp. 141–146.

[6] International Telecommunication Union, "Digital compression and coding of continuous-tone still images: Requirements and guidelines," International Telecommunication Union, Tech. Rep. T.81, September 1992, also published as ISO/IEC 10918-1:1994. [Online]. Available: https://www.w3.org/Graphics/JPEG/itu-t81.pdf