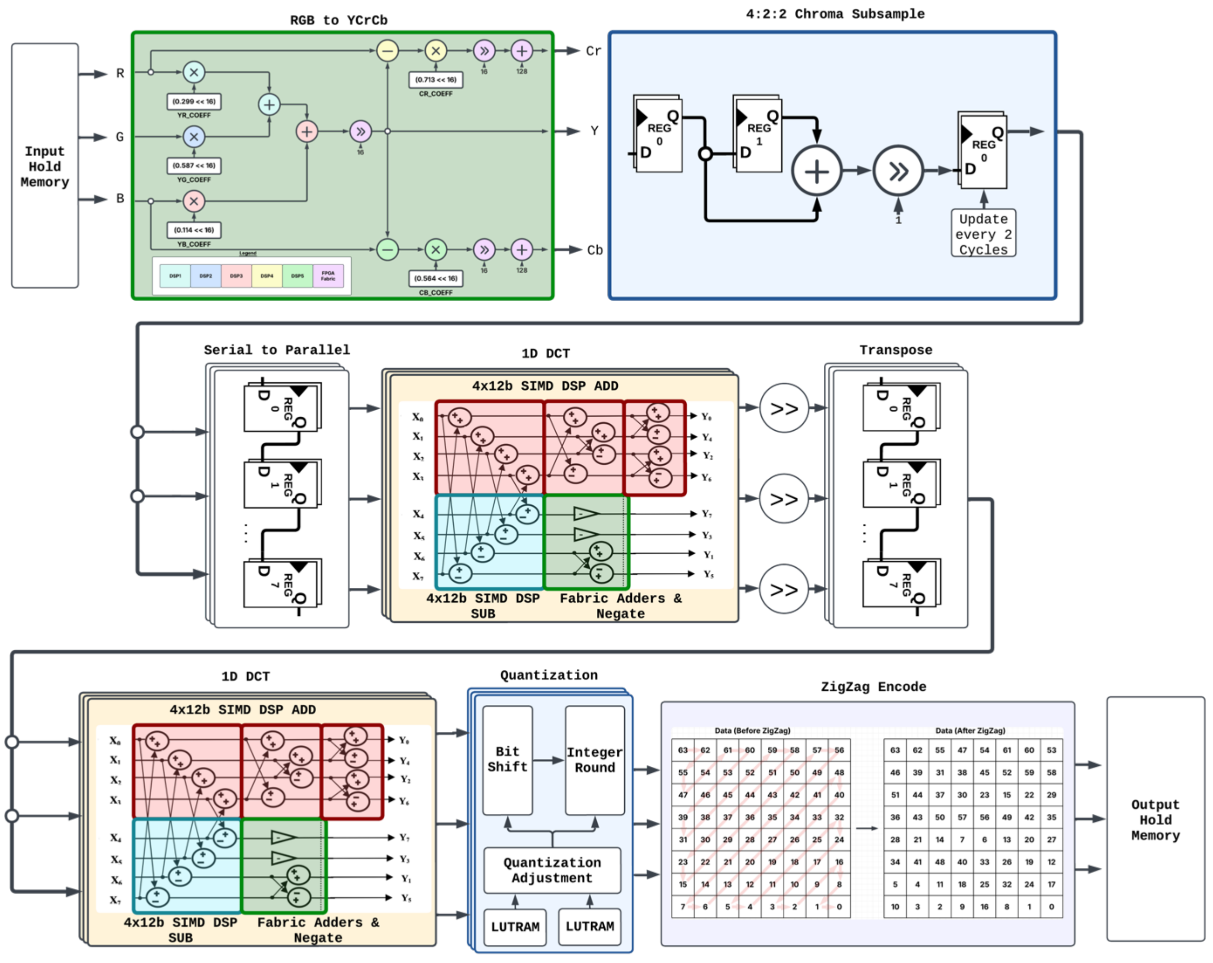
Integer based JPEG Encoding on FPGA’s In 7 pages

Written & Developed By John Hofmeyr
03/15/2025

Full JPEG Pipeline:

When implementing JPEG encoding, 6 main stages are required.
- RGB to YCrCb conversion & Chroma Subsampling
- 1D Discrete Cosine Transform (1D DCT)
- Block Transposition
- 1D DCT
- Quantization
- ZigZag Ordering

In this implementation, entropy encoding has been omitted due to time constraints. After ZigZag ordering, run length encoding is applied followed by Huffman encoding. The following diagram illustrates the simplified pipeline. Note, Zigzag ordering is implemented using the same method as the transpose buffer, with the only change being how the output bus is split.



RGB to YCrCb Conversion:

RGB to YCrCb conversion is achieved via the implementation of the following formulas, sourced from the OpenCV RGB to YCrCb documentation:

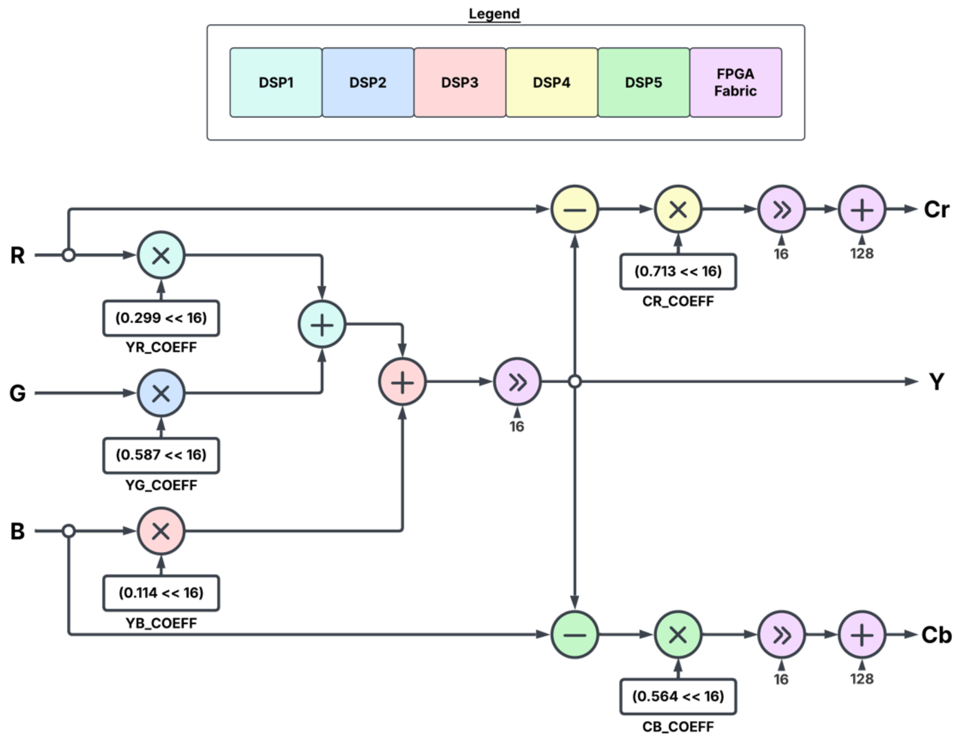
$$\begin{aligned} Y &= 0.299 * R + 0.587 * G + 0.144 * B \\ Cr &= [(R - Y) * 0.713] + 128 \\ Cb &= [(B - Y) * 0.564] + 128 \end{aligned}$$

Documentation Link: https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html

To avoid floating/fixed point operations with fractional coefficients, the coefficients are multiplied by a given power of 2. This allows for integer based operations and removes the need for any division. 2^{16} was chosen as it provides a good balance between accuracy and resource usage. The power of 2 multiplication is achieved via a left bit shift of 16. This multiplication is later undone via right shift of 16. The modified equations are shown below:

$$\begin{aligned} Y &= \{[(0.299 \ll 16) * R] + [(0.587 \ll 16) * G] + [(0.144 \ll 16) * B]\} \gg 16 \\ Cr &= \{[(R - Y) * (0.713 \ll 16)]\} \gg 16 + 128 \\ Cb &= \{[(B - Y) * (0.564 \ll 16)]\} \gg 16 + 128 \end{aligned}$$

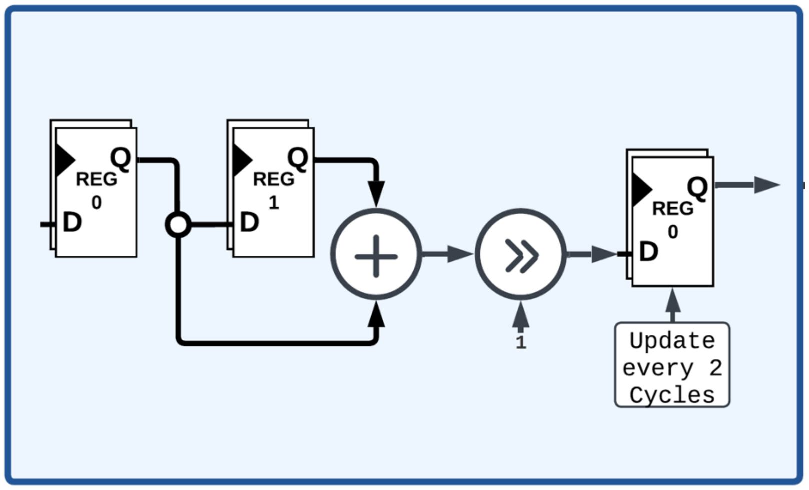
Input RGB values are resized to 24-bit Signed prior to performing all operations. Once all operations are complete and the Y, Cr and Cb values have been right shifted by 16, they are resized to unsigned 8-bit as they will always be positive and in the range of 0-255. The following diagram illustrates the hardware implementation of the equations above:



4:2:2 Chroma Subsampling:

4:2:2 Chroma Subsampling is applied to the Cr and Cb channels. This is an optional step. Subsampling is realized by taking the average of two input Cr/Cb values. One Cr/Cb value is input to the Subsampling module every clock cycle while one averaged value is output every 2 cycles. The diagram below illustrates the hardware implementation used. FPGA fabric adders are utilized for the addition in this stage. A register chain delay module is used to ensure the Y channel data arrives to its processing core at the same time as the subsampled Cr/Cb data.

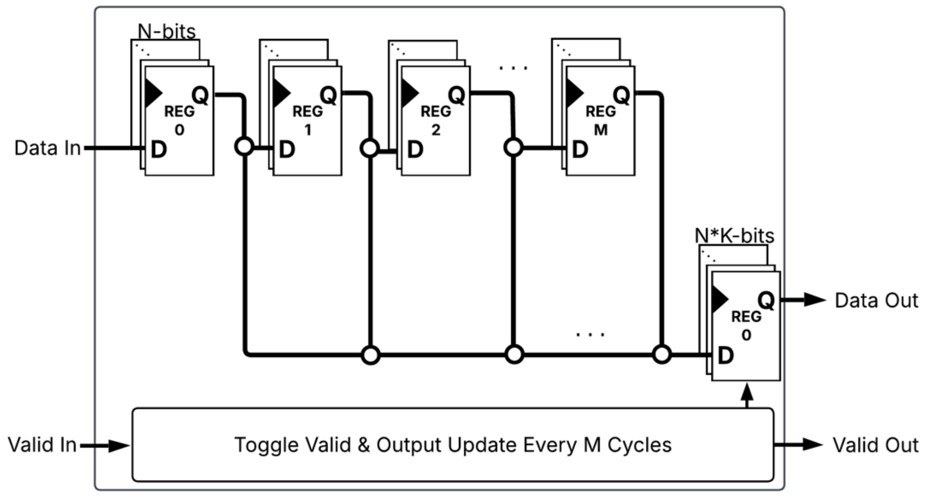
4:2:2 Chroma Subsample



Serial-In, Parallel-Out Buffer:

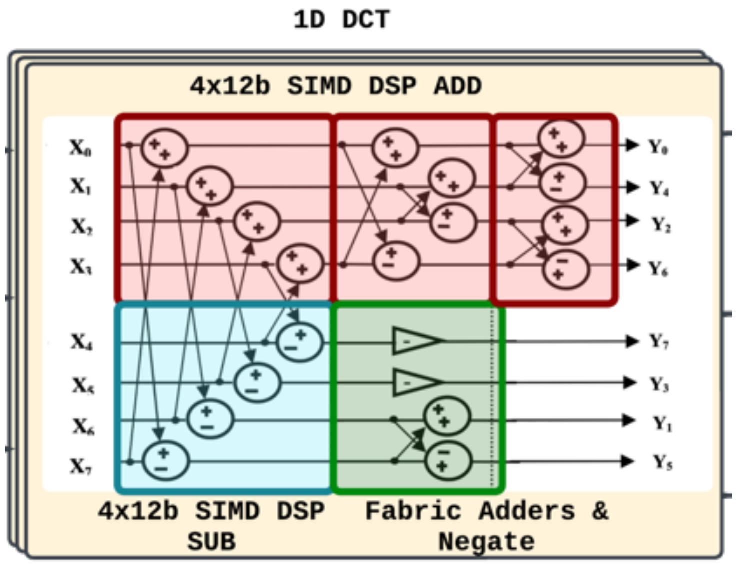
SIPO buffers are used throughout the processing cores to convert a serial pixel stream into a parallel stream. The implementation of the SIPO buffer is a **N-bit** wide by **M-element** deep shift register, with **K-taps** (Number of taps can be less than or equal to M). The diagram below illustrates the hardware implementation for the SIPO register:

Serial to Parallel



1D DCT Butterfly Stage:

The 1D DCT butterfly stage is implemented as follows. This DCT butterfly method was obtained from the following research paper: https://ieeexplore.ieee.org/abstract/document/5561411



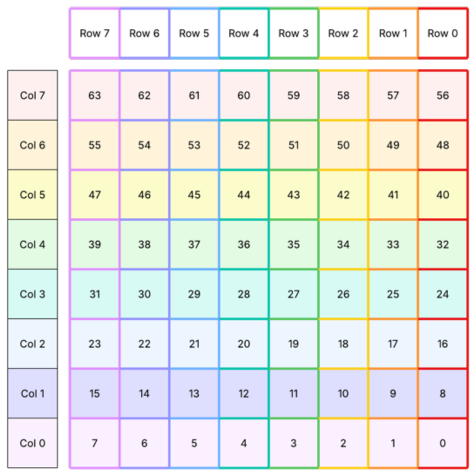
The implementation of this DCT Stage is realized using 4 DSP cores along with FPGA Fabric Adders and negation. The red and blue blocks and indicate the use of DSP48E2's in SIMD mode. The DSP's were configured to perform 4x12-bit signed Addition or subtraction. In the case of mixed addition/subtraction, a single operation was chosen, and operand negation was performed in fabric. The output of the DCT stage is reordered from [Y0, Y4, Y2, Y6, Y7, Y3, Y1, Y5] to [Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7].

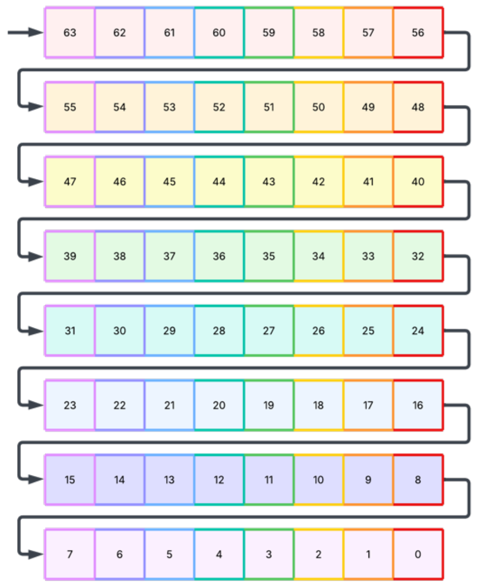
The input to the first DCT stage is 8-bits wide per element. These inputs are resized to signed 12-bit before being processed. After processing the output elements are divided by 8 to reduce the number of bits needed from 12 to 9. This step is done to reduce resource usage in the transpose and second DCT stage.

The input to the second DCT stage is 9-bits wide per element. These inputs are once again resized to signed 12-bit before being processed. After applying the second DCT stage, the output is not divided by 8 and the 12-bit signed coefficients are passed directly to the quantization stage.

<u>Transposition:</u>

In this design, 8-pixel wide columns are processed by the first DCT stage. The second DCT stage must then process 8-pixel wide rows. In order to do this, a transposition buffer is used to store an 8x8 or 64-element array of processed pixels. This is realized via a modified version of the SIPO buffer previously discussed. In this design, two 9*8-bit (or 72-bit) wide by 8 element deep SIPO buffers are used. A read-write FSM is then used to alternate reading and writing between the two buffers. When writing to buffer 0, data is read from buffer 1 and when writing to buffer 1, data is read from buffer 0. A mux is used along with careful bus slicing to read out one row each clock cycle. Data is shifted in from left to right, so the most significant 72-bits represent column 7 while the least significant 72 bits represent column 0. The following diagram represents how the columns are inserted. The color outline of each cell represents a row and the color fill of each cell represents each column. The rows and columns in this diagram are swapped as rata is read in column wise. Each number within the cell represents an element.


Row 7 Row 6 Row 5 Row 4 Row 3 Row 2 Row 1 Row 0
Col 7 63 62 61 60 59 58 57 56
Col 6 55 54 53 52 51 50 49 48
Col 5 47 46 45 44 43 42 41 40
Col 4 39 38 37 36 35 34 33 32
Col 3 31 30 29 28 27 26 25 24
Col 2 23 22 21 20 19 18 17 16
Col 1 15 14 13 12 11 10 9 8
Col 0 7 6 5 4 3 2 1 0



In total there are 8x72-bit taps, one on column (8x9-bit register). When combined this results in an output bus width of 8*72 bits or 576-bits. In order to correctly read out each row, A mux is used to select between the correct bits based on a counter value from 0 to 7. For example, when reading row 0, the following bits would be concatenated to form the single 72-bit output:

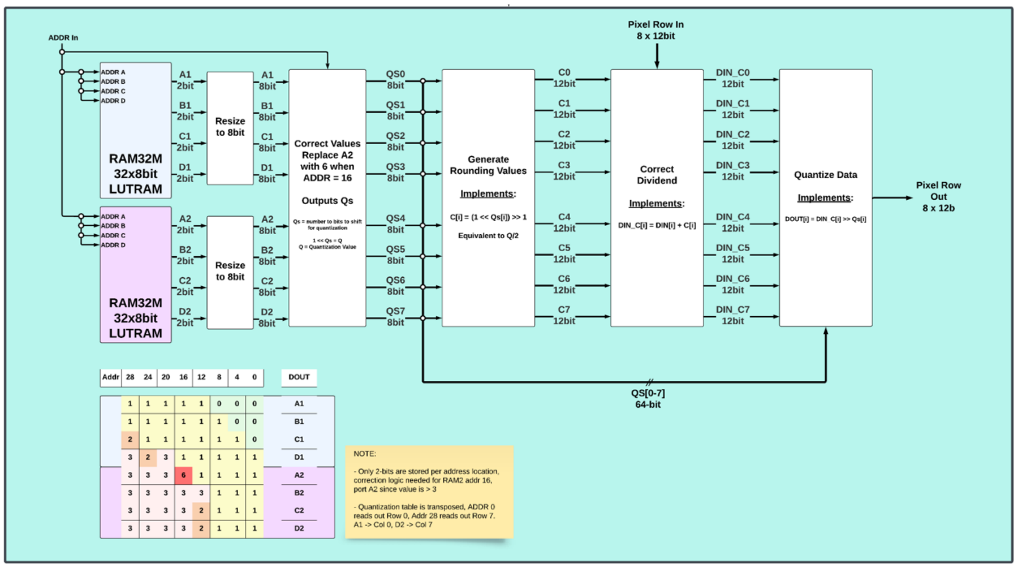
***Output* = (512 - 504) & (440 - 432) & (368 - 360) & (296 - 288) & (224 - 216) & (152 - 144) & (80 - 72) & (8 - 0)**

This output mapping corresponds to the following row/column output:

***Output* = (R0, C7) & (R0, C6) & (R0, C5) & (R0, C4) & (R0, C3) & (R0, C2) & (R0, C1) & (R0, C0)**

Quantization:

Quantization is achieved via bit shifting. Careful selection of quantization values is required, ensuring they are powers of 2 for this implementation. In order to store the required shit amounts, two 32x8 bit LUTRAM primitives are utilized. Each ram has 4 2-bit output ports. Due to this, special handling is required for values greater than 2-bits (ie 3 and above). In order to maintain single cycle reading, address based correction was implemented. Since the address and output port of all values >3 is known, an "if" statement may be used to selectively modify the output data based on which address is being read. This is utilized to modify the output of port A2 at address 16 so that it is correctly read as "6". The quantization table stored in memory is the base line minimum quantization. When applying correction, an optional scale factor can be added to increase the quantization amount which will reduce image quality but increase compression amount. The following diagram illustrates how quantization is implemented in hardware:



Addr	28	24	20	16	12	8	4	0	DOUT
	1	1	1	1	1	0	0	0	A1
	1	1	1	1	1	1	0	0	B1
	2	1	1	1	1	1	1	0	C1
	3	2	3	1	1	1	1	1	D1
	3	3	3	6	1	1	1	1	A2
	3	3	3	3	3	1	1	1	B2
	3	3	3	3	2	1	1	1	C2
	3	3	3	3	2	1	1	1	D2

NOTE:

- Only 2-bits are stored per address location, correction logic needed for RAM2 addr 16, port A2 since value is > 3

- Quantization table is transposed, ADDR 0 reads out Row 0, Addr 28 reads out Row 7. A1 -> Col 0, D2 -> Col 7

When applying quantization via bit shifting, rounding errors may occur. This is due to the fact that bit shifting integer values will floor the result value. For example, if the operation **(55/4)** were to be performed via bit shifting, one would perform **(55 >> 2)**. When **(55/4)** is performed, we obtain a result of **13.75**. When applying rounding, the expected value is **14**. But in the case of bit shifting with integer values, performing **55>>2** returns **13**, not **14**. In order to correct for this behavior, the following equation is used to implement ***Round(A/B)***:

$$\boldsymbol{Result} = \frac{\boldsymbol{A} + \left(\frac{\boldsymbol{B}}{\mathbf{2}}\right)}{\boldsymbol{B}}$$

When applied to Bit shifting, this equation is modified to the following equation:

$$\boldsymbol{Result} = (\boldsymbol{A} + [\mathbf{1} \ll (\boldsymbol{X} - \mathbf{1})]) \gg \boldsymbol{X}$$

Note: X = # of bits. This assumes that B is a power of 2 in the 1st equation and $X = \log_2(B)$

<u>Improvements to Consider:</u>

The following list summarizes potential improvements that can be made to this design

- Replacing integer-based calculations with fixed point to improve accuracy
- Modifying the Quantizer to apply division in the same manner as the YCrCb module, quantization values can be stored as (1/N) << 16, where N is the amount we want to divide by (ie if we want to divide by 70, we can store (1/70) << 16 = 936. This value is then multiplied by our DCT coefficient and right shifted by 16 to obtain the quantized value. The full equation would be:

  $$Quantized\ Value = (DCT\_Coeff * ([1/Quantization\_Value] << 16) >> 16$$

  Further improvement can be made by implementing the integer rounding logic discussed in the Quantization section
- Reducing resource usage for Transposition & Zigzag ordering. The current implementation uses very large register chains which use a significant number of LUT's and FF's
- Implementing 4:2:0 subsampling
- Implementing RLE and Huffman encoding