UCLouvain

École polytechnique de Louvain

An Embedded JPEG Compression Scheme With Mixed-Signal Near-Sensor DCT Processing

Author: **Théo Vanden Driessche**
Supervisor: **David Bol**
Readers: **Christophe De Vleeschouwer, Martin Lefebvre**
Academic year 2022–2023
Master [120] in Electrical Engineering

Contents

Acknowledgement

I would like to thank my thesis advisor, Professor David Bol, for his availability and valuable feedback during our bi-weekly meetings. His guidance has greatly influenced the direction and quality of this research work.

I am grateful to Martin Lefebvre, the designer of the MANTIS System On Chip. His continuous support, even for tough debugging, technical knowledge, and willingness to assist throughout the year, have been invaluable. Without his guidance and assistance, conducting this research work to such an extent would not have been possible.

I would also like to express my appreciation to Professor Christophe De Vleeschouwer for agreeing to be the reader of my thesis.

I would like to extend my gratitude to Emiel Deprost, PhD student at UGent IDLab, and Guillaume van der Rest, master student at UCLouvain, for generously dedicating their time to read and provide valuable feedback on this thesis. Their insightful comments and suggestions have greatly contributed to the refinement and improvement of the text.

Finally, I would like to acknowledge the support of my family and friends. Their encouragement and understanding have been invaluable throughout this academic journey.

Abstract

This master thesis presents an innovative embedded JPEG compression scheme with mixed-signal near-sensor 2D DCT processing, implemented on the CMOS imager System On Chip (SoC) MANTIS developed at UCLouvain. The goal of this work is to enable efficient image compression while minimizing power consumption by working in the analog domain.

The thesis begins with an in-depth explanation of the JPEG compression algorithm and definition of image quality metrics. Subsequently, the focus shifts towards exploring approximations of the 2D DCT suitable for analog implementation. These approximations are meticulously detailed, taking into account the unique characteristics and constraints of the analog voltage mode MAC units with programmable weights embedded in MANTIS.

The proposed compression scheme is implemented on the SoC MANTIS developed at UCLouvain, leveraging its capabilities and highlighting his limitations for future improvements. Simulation results are presented to demonstrate the effectiveness of the scheme. The achieved image compression ratio is 3.39 bits per pixel (bpp) while maintaining an average Peak Signal-to-Noise Ratio (PSNR) of 27dB and consuming only 266 pJ/pixel, validating the scheme's energy-efficient design.

The findings of this thesis contribute to the field of embedded image processing, particularly in the context of mixed-signal near-sensor architectures. The successful implementation on the SoC MANTIS developed at UCLouvain demonstrates the feasibility and practicality of the proposed scheme. It offers a promising solution for achieving high-quality image compression with reduced power consumption, opening doors for various applications in areas such as wireless imaging systems, resource-constrained devices, and Internet of Things (IoT) applications.

Acronyms

1D one-dimensional. 16

2D two-dimensional. 16

AC alternating current. 23, 25, 26

ADC analog-to-digital converter. 61

AHB advanced high-performance bus. 51

APDCBT all phase discrete cosine biorthogonal transform. 15

APIDCBT all phase inverse discrete cosine biorthogonal transform. 15

APS active-pixel sensor. 50

bpp bits per pixel. 72, 73

CFA color filter array. 14

CORDIC coordinate rotation digital computer. 33, 34

DAC digital-to-analog converter. 45

DC direct current. 7, 23–26

DCMI digital camera memory interface. 51

DCT discrete cosine transform. 1, 7, 13–23, 32, 34–36, 38–40, 61

DMA direct memory access. 51

EOB end of block. 26

FIR finite impulse response. 39

IDCT inverse discrete cosine transform. 7, 15–17, 19, 20, 38

IP intelectual property. 14

IRQ interrupt request. 69

JPEG joint photographic experts group. 13, 14, 16, 22–24, 26, 28, 29

KCL Kirchhoff's current law. 34

MAC multiply and accumulate. 7, 32, 34, 35

MANTIS . 36

MSB most significant bit. 7, 38

MSE mean square error. 26

MSSIM mean SSIM. 27, 28

OTA operational transconductance amplifier. 35

PSNR peak signal-to-noise ratio. 8, 9, 26, 27, 36, 43, 44, 46, 47, 87

RGB red green blue. 13, 14, 27

SAR successive approximation register. 51

SC switched capacitor. 34

SCA switched-capacitor amplifier. 50, 51

SSIM structural similarity index measure. 8, 9, 26–28, 43, 44, 47, 87

List of Figures

List of Tables

Introduction

The increasing need for small mobile devices with low-power images or videos acquisition capabilities, such as wireless capsule endoscopy [27] or environmental monitoring [7], has led to the development of hardware providing near-sensor image compression. The Discrete Cosine Transform (DCT) is widely used as first step in image and video compression standards including JPEG [35], H.26x and MPEG.

In this work, we implement a complete compression scheme of the JPEG standard with the Mantis CMOS imager SoC featuring near-sensor processing capabilities developed at the ECS group of UCLouvain leveraging analog domain voltage mode switched-capacitors amplifiers (SCAs) MAC units, with programmable weights, used to perform the DCT. The signals in the analog domain are digitized with successive approximation registers (SARs) analog-to-digital converters (ADC) for further processing in the digital domain. We achieved a compression ratio of 3.39 bits per pixel (bpp), while maintaining an average Peak Signal-to-Noise Ratio (PSNR) of 27 dB with a power consumption estimated at 266 pJ/pixel. We also explored the possibility to embed the quantization and Huffman coding steps of the JPEG algorithm in software on the ARM Cortex-M4 processor on-chip, which showed poor results.

We first detail the JPEG algorithm. Then, to provide a comprehensive evaluation of our implementation, we compare our results with state-of-the-art analog and digital implementations available in the literature in chapter 2. Then we propose a way of approximating the DCT specifically tailored for constrain of voltage mode analog implementation. We perform an in-depth analysis and observe the impact on image quality of those approximations. We then present the hardware of MANTIS and quantify the limitations, for performing the DCT, of the embedded SCAs and ADC. This serves as a foundation for building minimal hardware requirements. We finally discuss possible hardware and software improvements.

The work is structured as follows:

- **Chapter 1** describes the complete JPEG compression method.

- **Chapter 2** describes the state of the art of near-sensor analog image compression based on the DCT. We give a brief description of common implementation with their advantages and disadvantages.
- **Chapter 3** highlights analog considerations to perform the DCT and impact of non idealities on the image quality then we describe the impact of the coefficients resolution and how we can approximate the DCT with low bit resolution coefficients.
- **Chapter 4** presents the mixed-signal hardware circuits from MANTIS SoC that we will use for the implementation and the actual implementation. We finally discuss the results, limitations and possible improvements of the compression scheme.
- **Chapter 5** discusses possible improvements.

Chapter 1

JPEG

JPEG [35] is a lossy compression method which typically achieves a compression ratio of 10 without introducing humanly perceptible losses and thus maintaining good image quality. The JPEG compression scheme for colored image, as described in the corresponding ISO/IEC 10918 standard [4], can be broken down as:

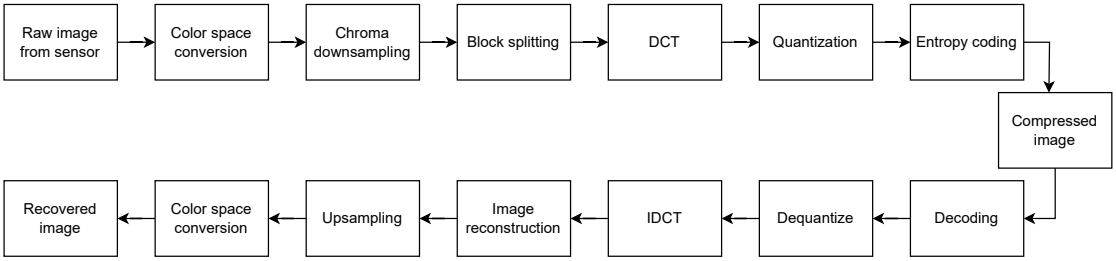


Figure 1.1: JPEG compression and decompression scheme

1. Color space conversion: RGB to YCbCr
2. Chrominance downsampling
3. Block splitting
4. discrete cosine transform (DCT)
5. Quantization
6. Entropy coding

We will now explain each step of the JPEG in more detail.

1.1 Color space conversion

The JPEG algorithm works with images in the YCbCr color space, where Y is the luminance, Cb the blue chrominance and Cr the red chrominance. This representation strongly decorrelates the color from the intensity channels, which allows further independent processing.

Typical color image sensors use a color filter array (CFA) on top of the monochrome image sensor for color separation. Instead of capturing three color samples, typically RGB, at each pixel location, a mosaic pattern samples only one color at each grid location. A common CFA is the Bayer filter mosaic [8] as shown in figure 1.2 for which green elements are twice more frequent, which is similar to the human eye color sensitivity.

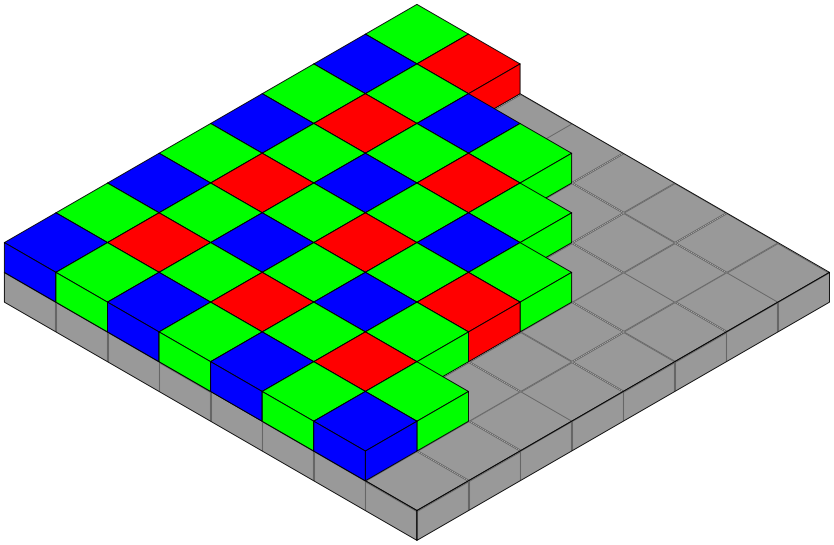


Figure 1.2: Bayer filter mosaic on top of monochromatic image sensor array[10]

To retrieve the 3 color components at each pixel location we need to perform interpolation with the surrounding pixels to retrieve the 2 missing color components, this process is called debayering or demosaicing. Multiple debayering algorithms exist [22] to either output an image in the RGB or YCbCr[11] color space. These algorithms are mostly implemented digitally [6, 45] some are given as digital IP such as the PG286 from Xilinx [45]. However, there are currently no analogic debayering algorithms in the literature. Other approaches exist, such as moving the debayering step after the decompression and not before. However, this approach should not use the DCT for compression since the Bayer filter induces high amplitude variations between adjacent pixels. This introduces high frequency components in the image which will lead to poor compression performances if *dct is

used. In [46, 44] a novel method replaces the DCT and IDCT with an all phase discrete cosine biorthogonal transform (APDCBT) and an all phase inverse discrete cosine biorthogonal transform (APIDCBT), this type of approach could be implemented in the analog domain similarly to the DCT implementation given in this work.

1.2 Color downsampling

Medical studies have shown that the human eye is less capable of discerning color/chroma differences than intensity/luma differences[1]. This particularity of the human vision allows the downsampling of the chroma component with no visual difference perceived by the viewer.

Typically, the chroma channels are downsampled by a factor 2 and is expressed as 4:2:2. This notation refers to a:

- 4: 4x4 luma reference block
- 2: 2 chroma samples for the first luma row
- 2: number of changes of chroma for the first and second row

with 4:4:4 representing the full chroma resolution. A visual representation of color downsampling can be seen in 1.3

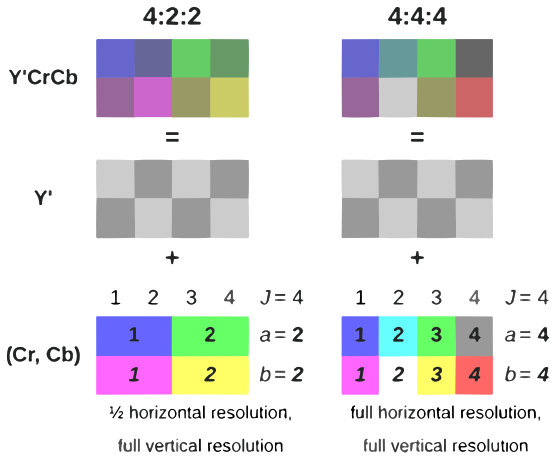


Figure 1.3: Color downsampling example 4:4:4 and 4:2:2 (Image from [37])

To represent the actual implementation and hardware in this work, further development will ignore the color components and only consider a grayscale image.

1.3 Block spliting

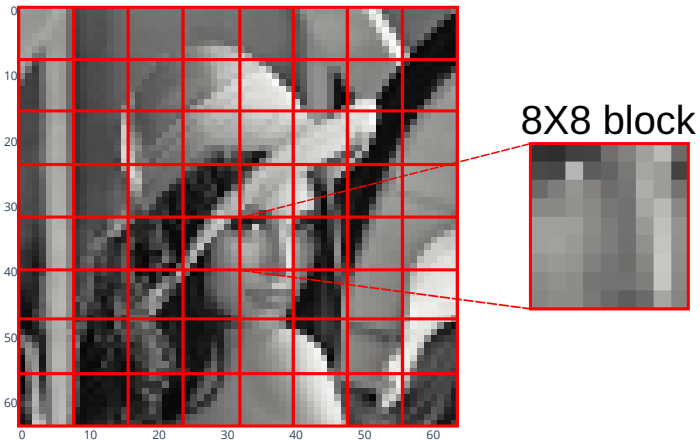


Figure 1.4: Block spliting of Lena 64x64 grayscale in 8x8 blocks

In this step, each component of the original image is split in blocks of 4X4, 8X8 or 16X16 pixels. This is represented in figure 1.4 where a 64X64 pixels grayscale image is split in 64 blocks of 8X8 pixels. Further processing is always done per block.

1.4 discrete cosine transform (DCT)

The DCT is used in the JPEG scheme for his high energy compaction capabilities on natural images. The DCT was first introduced in 1974 [5] as a tool for image compression and image processing. We will in the following paragraph describe the 1D and 2D DCT in different forms.

1D DCT: Matrix form

The 1D DCT is given by:

$$\text{DCT:} \quad Y_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad \text{for } k = 0,\dots N-1 \tag{1.1}$$

$$\text{IDCT:} \quad x_k = \frac{1}{2}Y_0 + \sum_{n=1}^{N-1} Y_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad \text{for } k = 0,\dots N-1 \tag{1.2}$$

with $x \in \mathbb{R}^{\mathbb{N}}$ the signal vector.

2D DCT

We can extend the one-dimensionnal definition to two dimensions by first computing a 1D DCT in one dimension, then computing the DCT in the other dimension. For N_1 and N_2 representing the first and second dimension respectively, the 2D DCT can be expressed as:

$$\text{2D DCT:} \quad Y_{k_1,k_2} = \sum_{n_2=0}^{N_1-1} \left(\sum_{n_1=0}^{N_2-1} x_{n_1,n_2} \cos\left[\frac{\pi}{N_2} \left(n_1 + \frac{1}{2} \right) k_2 \right] \right) \cos\left[\frac{\pi}{N_1} \left(n_2 + \frac{1}{2} \right) k_1 \right] \tag{1.3}$$

$$= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1,n_2} \cos\left[\frac{\pi}{N_1} \left(n_1 + \frac{1}{2} \right) k_1 \right] \cos\left[\frac{\pi}{N_2} \left(n_2 + \frac{1}{2} \right) k_2 \right]. \tag{1.4}$$

2D DCT: Matrix form We can also define the two-dimensional DCT in matrix form using the product of cosine relationship [1] and some algebra. We consider the same amount of samples in each dimension such that $N_1 = N_2 = N$ we can then define the coefficients of the NxN matrix C, as

$$C_{pq} = \begin{cases} \frac{1}{\sqrt{N}} & p = 0, \quad 0 \leq q \leq N-1 \\ \sqrt{\frac{2}{N}} \cos \frac{\pi(2q+1)p}{2N} & 1 \leq p \leq M-1, \quad 0 \leq q \leq N-1 \end{cases} \tag{1.5}$$

Some interesting properties of the coefficient matrix C:

- **Orthogonality** $C^{-1} = C^T$ if C is square
- **Periodicity** due to the periodicity of the cosine function, we can observe that the term: $|\cos\left(\frac{\pi(2q+1)p}{2N}\right)|$ can only take $N-2$ different values. Thus, C has only $N-1$ different absolute value coefficients. Or $2(N-1)$ different coefficients if we consider the sign
- **Range** $-\sqrt{\frac{2}{N}} < C_{pq} < \sqrt{\frac{2}{N}}$

We can then express the DCT and IDCT as

$$\text{2D DCT:} \quad Y = CXC^T \tag{1.6a}$$

$$\text{2D IDCT:} \quad X = C^T Y C \tag{1.6b}$$

[1] $\cos a \cos b = \frac{\cos(a+b)+\cos(a-b)}{2}$ [12]

For the case N=8 we can observe that the matrix is constructed with only 7 different coefficients[2] :

$$C = \begin{bmatrix} a & a & a & a & a & a & a & a \\ b & c & d & e & -e & -d & -c & -b \\ f & g & -g & -f & -f & -g & g & f \\ c & -e & -b & -d & d & b & e & -c \\ a & -a & -a & a & a & -a & -a & a \\ d & -b & e & c & -c & -e & b & -d \\ g & -f & f & -g & -g & f & -f & g \\ e & -d & c & -b & b & -c & d & -e \end{bmatrix} \tag{1.7}$$

2D DCT: Tensor form Starting with the matrix form of the 2D DCT given in equation **??** we can observe that each term of the DCT of X, $Y_{p,q}$ can be expressed as

$$Y_{p,q} = \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_{i,j} X_{i,j} \tag{1.8}$$

With each $\alpha_{i,j}$ being the product of two coefficients of the C matrix. We obtain these coefficients by inspection of the Y matrix.
The previous expression can be reformulated as the Frobenius inner product [39] of two (NxN) matrices. A containing the α_{ij} coefficients and X.

$$Y_{p,q} = \langle A, X \rangle_F \tag{1.9}$$

To ease further mathematical development, let us first quickly introduce the Einstein notation[38].

Einstein Notation The Einstein notation or the Einstein summation notation is a useful convention when manipulating high dimensional vectors or, more formally, tensors. Let us see a simple example: translating the dot product with the Einstein notation.

---

[2] With $a = \frac{1}{\sqrt{N}}$, $b = \sqrt{\frac{2}{N}} \cos\left(\frac{\pi}{2N}\right)$, $c = \sqrt{\frac{2}{N}} \cos\left(\frac{3\pi}{2N}\right)$, $d = \sqrt{\frac{2}{N}} \cos\left(\frac{5\pi}{2N}\right)$, $e = \sqrt{\frac{2}{N}} \cos\left(\frac{7\pi}{2N}\right)$, $f = \sqrt{\frac{2}{N}} \cos\left(\frac{8\pi}{2N}\right)$, $g = \sqrt{\frac{2}{N}} \cos\left(\frac{6\pi}{2N}\right)$

Let $a \in \mathbb{R}^3$ and $b \in \mathbb{R}^3$, the dot product of those two vectors can be expressed as:

$$a^T b = \sum_{i=1}^{3} a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{1.10}$$

With the Einstein notation this become

$$a^T b = a_i a^i \tag{1.11}$$

Which simply means: The repeating lower and upper indices means the summation of the product of coordinates along this axis. This can be extended to higher dimensions. Some useful properties of the Einstein notation:

- **indices contraction** $Y_\mu = A_{\mu\nu} X^\nu \in \mathbb{R}^N$ with $A_{\mu\nu} \in \mathbb{R}^N$ and $X^\nu \in \mathbb{R}^M$. Repeating lower and upper indices reduces the dimension of the output tensor.
- $A_{\mu\nu} X^\nu = A^{\mu\nu} X_\nu$[3]

Knowing this, we can now reformulate expression 1.4 as, following the same development we can construct the A'_{pqij} used to compute the IDCT

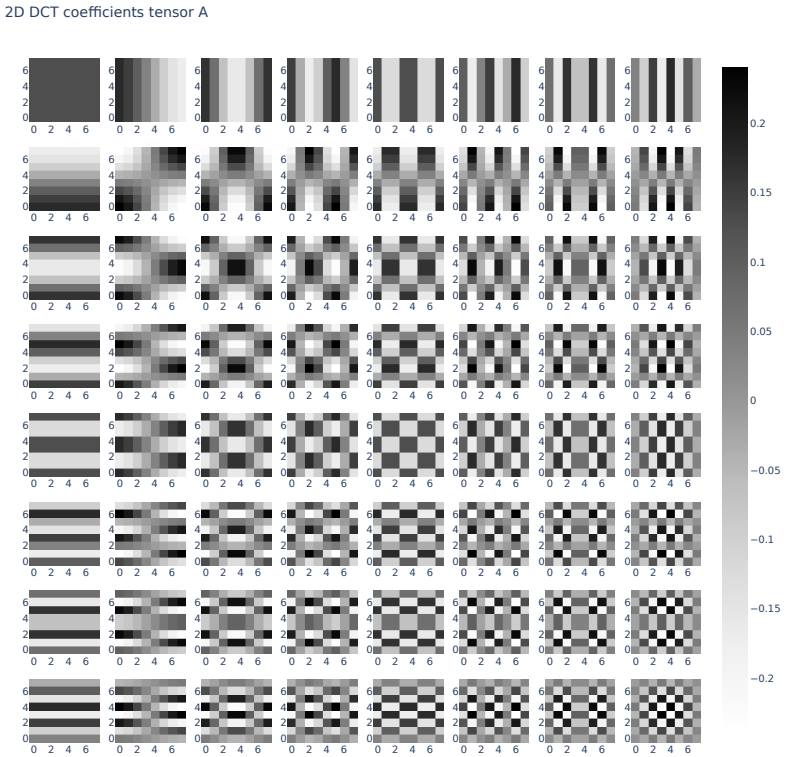
$$\text{2D DCT:} \quad Y_{pq} = A_{pqij} X^{ij} \tag{1.12a}$$
$$\text{2D IDCT:} \quad X_{pq} = A'_{pqij} Y^{ij} \tag{1.12b}$$

With:

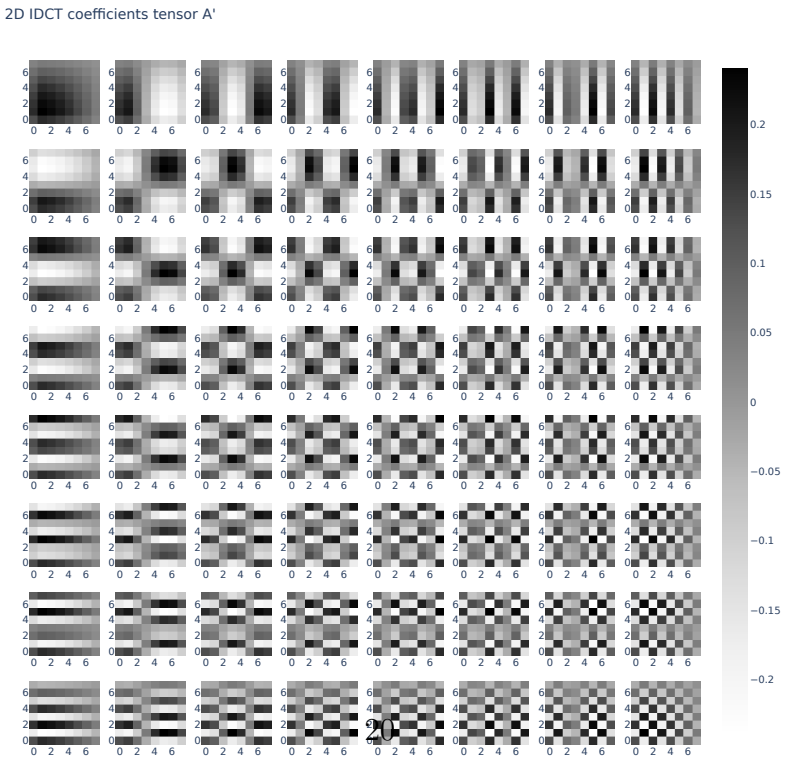
- $X_{ij} \in \mathbb{R}^{N \times N}$ the input two-dimensional signal or image block
- $Y_{pq} \in \mathbb{R}^{N \times N}$ the output of the DCT of X_{ij}
- $A_{pqij} \in \mathbb{R}^{N \times N \times N \times N}$ This tensor contains the N^2 (NxN) matrices A used to compute each coefficients of the DCT of X.
- And with the indices: $i, j, p, q = \{k | k \in \mathbb{Z} \text{ and } 0 \leq k \leq N-1\}$

From previous consideration of the C matrix we can conclude that:

- **Periodicity** Since the coefficients in tensor A are constructed as the product of two coefficients of the matrix C, A has only $\frac{N \cdot (N-1)}{2}$ different coefficients in absolute value.
- **Range** $-\frac{2}{N} < A_{pqij} < \frac{2}{N}$



(a) DCT coefficient tensor A for N=8



(b) IDCT coefficient tensor A' for N=8

Figure 1.5: DCT and IDCT coefficients tensors

For $N = 8$ we can visualize the inverse and direct coefficient tensors in figure 1.5.

Both ways of computing the 2D DCT gives the exact same output. We can observe the output of the 2D DCT of a whole picture in figure 1.6 with a close look at the DCT coefficients of a block.

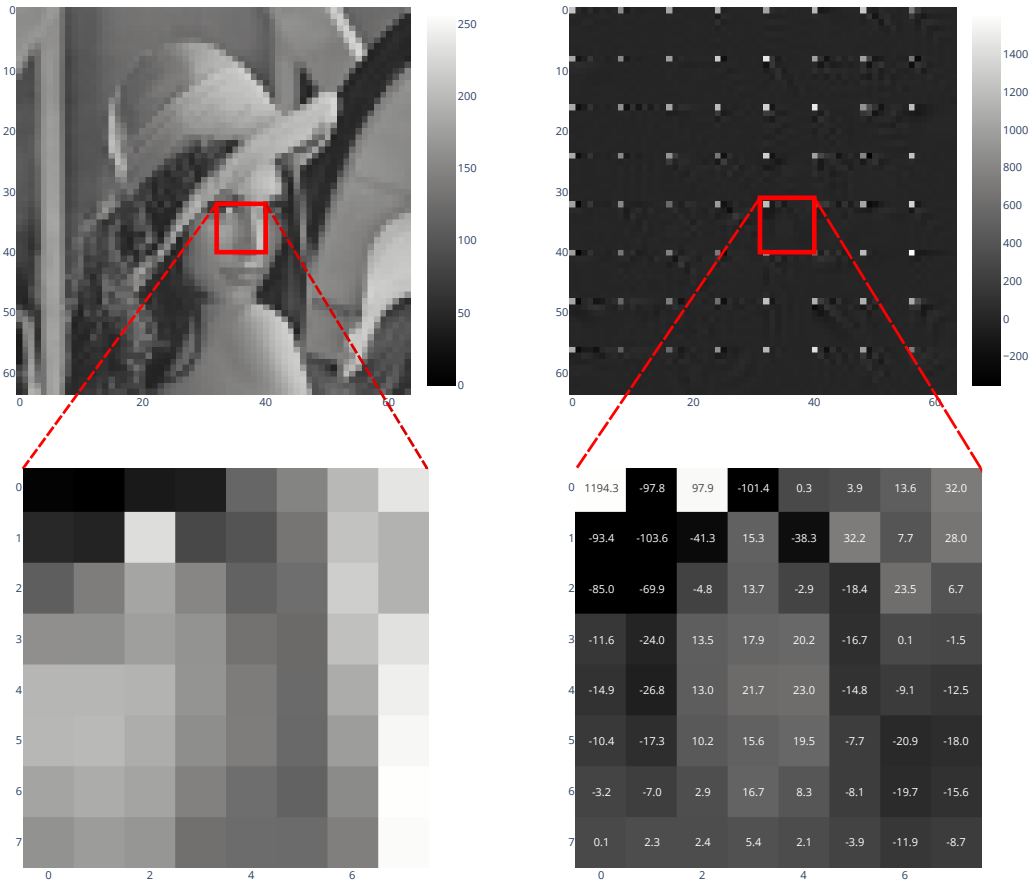


Figure 1.6: DCT of 64x64 Lena

[3]In this context we use an euclidean metric, this identity will be incorrect for a different metric. See [40] for further information.

1.5 Quantization

The quantization step is the main source of information loss in the JPEG algorithm. While previous steps only changed the representation of the original image without loss of information, the quantization step looses information by reducing the number of bits used to represent the DCT coefficients. We can formulate the quantization step as:

$$q_{ij} = \left\lfloor \frac{Y_{ij}}{Q_{ij}} \right\rceil \tag{1.13}$$

with

- $\lfloor \rceil$: the "round to nearest integer" operator;
- Y_{ij} the (i, j) term of the DCT;
- Q_{ij} the (i, j) entry of the quantization matrix Q;
- q_{ij} the quantized value.

Similarly we can define the dequantization as:

$$Y'_{ij} = q_{ij} \cdot Q_{ij} \tag{1.14}$$

The quantization matrix is defined to give more importance to low frequency values. The human eye is less capable of distinguishing rapid variation of brightness, this allows us to reduce the amount of information contained in the highest frequency components of the DCT. A typical luma quantization matrix is shown in figure 1.7.

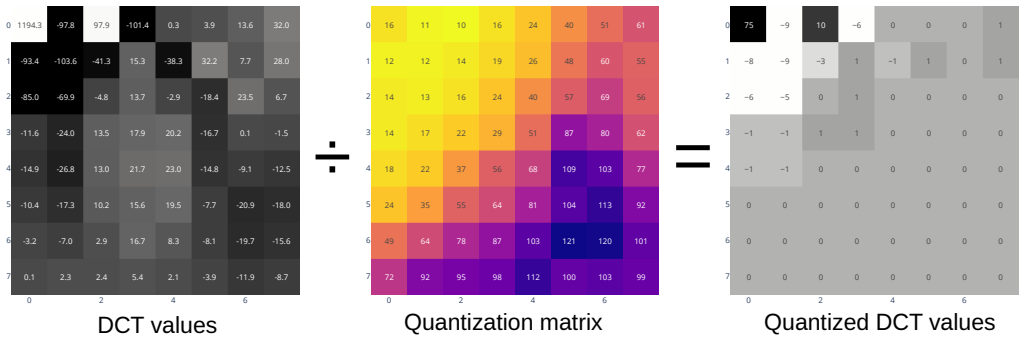


Figure 1.7: Quantization of one DCT block

Different quantization matrices are applied to the chroma components. Finding good quantization coefficients is context dependent and requires fine-tuning[13,

26]. The quantization matrix will influence the compression ratio of the image, a quantization matrix containing only ones will produce a near lossless image after decompression, while a quantization matrix with large coefficients will produce a highly compressed but low quality image.

1.6 Entropy coding

The last step of the JPEG compression is entropy coding. The goal of entropy coding is to reduce the amount of bits necessary to encode the same information, there is no information loss during this step. First, the DCT values are reordered in a zigzag sequence, as depicted in figure 1.8. Then, as described in the JPEG ISO standard [35], differential entropy coding is performed for the DC components and amplitude coding for the AC components of the DCT.

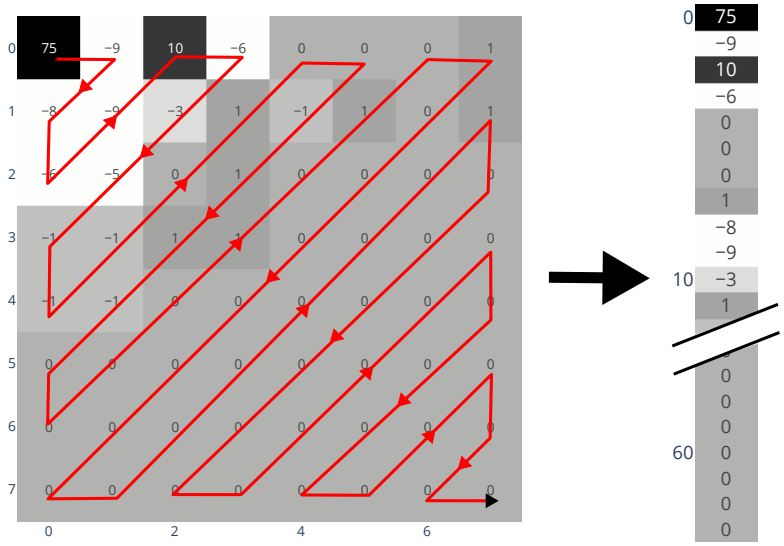


Figure 1.8: Zigzag reordering of $Y \in \mathbf{R}^{8\times8}$ to $\mathrm{ZZ} \in \mathbf{R}^{64}$

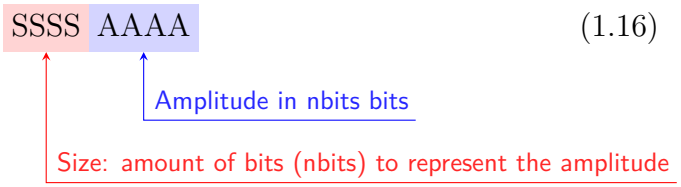
1.6.1 Differential coding of the DC terms

The DC's components are encoded differentially as the difference between the actual DC component and a predictor (PRED). The predictor is the DC component of the previous block of the same channel.

$$\mathrm{DIFF} = \mathrm{ZZ} - \mathrm{PRED} \tag{1.15}$$

The first predictor value is initialized at PRED = 0.

The computed difference is then encoded as a bitstring, following this structure:



(1.16)

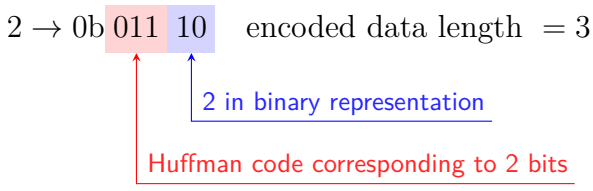
The size part of the code (SSSS) is an Huffman prefix code and represents the amount of bits necessary to store the amplitude part of the code (AAAA). The amplitude can either be positive or negative, negative values are encoded as DIFF − 1. Thus, the most significant bit of AAAA is 1 for positive values and 0 for negative.

The prefix codes or Huffman codes are build with the following constraints:

- **Bi-unique**: Each code represents uniquely each symbols
- **Minimum length**: Each prefix code is selected such that the most frequent symbols are encoded with the shortest prefix codes.

We can build a binary tree following these constraints, the whole process of building the tree will not be discussed in details since we will use the computed Huffman codes given in the JPEG ISO standard[35].

For the DC term, each possible size or symbol, from 0 to 11, is represented by a Huffman code[17] and is given in the column "Code word" of table 1.6.1, a part of the table is represented as a tree in figure 1.6.1. For example, the amplitude 2 will be encoded as the bitstring:



(1.17)

SSSS	DIFF values	Code word
0	0	00
1	-1,1	010
2	-3,-2,2,3	011
3	-7..,-4,4,..7	100
4	-15..,-8,8,..15	101
5	-31..,-16,16..31	110
6	-63..,-32,32,..63	110
7	-127..,-64,64,..127	1110
8	-255..,-128,128,..255	11110
9	511..256,256..511	111110
10	1023..512,512..1023	11111110
11	2047..1024,1024..2047	111111110

Table 1.1: DC Huffman codes

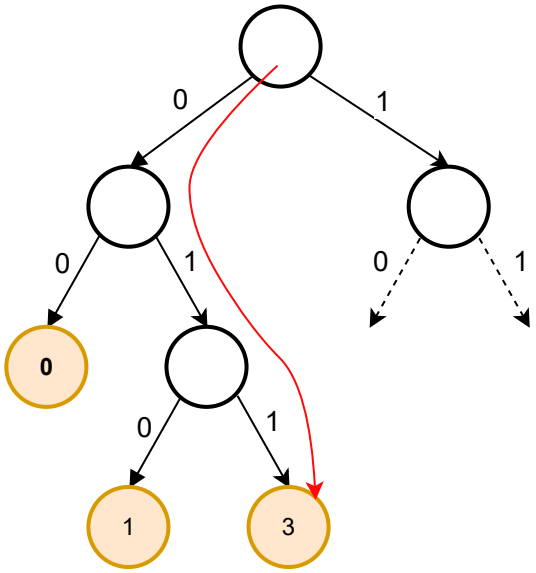


Figure 1.9: DC Huffman code tree, with path for 2 bits length amplitude in red

For this example, if the raw data was encoded as an 8 bit type, we observe that the encoded data uses less bits than the raw data.

1.6.2 Amplitude coding of the AC terms

All the AC coefficient are run-lenght encoded in zig zag order, and are encoded as a bitstring following this structure:

$$(\text{RRRR}\ \text{SSSS})\ \text{AAAA} \tag{1.18}$$

Amplitude in nbits bits

Size: amount of bits (nbits) to represent the amplitude

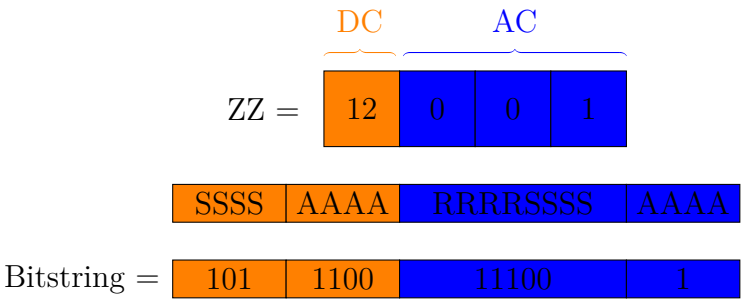
Zero run lenght

The composed prefix code word RRRRSSSS contains the run length and the number of bits used for the amplitude encoding:

- SSSS, similar to the DC terms encoding, gives the amount of bits used to represent the amplitude of the non-zero AC coefficient. From 0 to 15 bits.
- RRRR give the amount of preceding zero coefficient from 0 to 15.

In the JPEG ISO/IEC 10918-1 256 code words are given representing each combination of run length and size [35].

For example, we can encode a simple zigzag sequence with this encoding scheme:



The final bitstring uses only 13 bits.

Special symbols

Once the block is completely encoded or if there are no non zero coefficient left a special symbol, end of block (EOB), is added to the bitstring. In the JPEG ISO standard the correspond to the code 0b1010.

Decoding

Since we have access to the binary tree defining the Huffman codes for the DC and AC coefficients, we simply read the bitstring as a path of the DC or AC tree until reaching a leaf. The leaf will give information on how to read the coefficient.

1.7 Image quality metrics

To assess the quality of a compression scheme, we need to use a metric that can quantify the quality of the reconstructed image after compression. Two metrics are commonly used:

- peak signal-to-noise ratio (PSNR)[42]
- structural similarity index measure (SSIM)[43, 36]

PSNR The *psnr is a logarithmic quality metric which is based on pixelwise mean square error (MSE) between the two images. Given a grayscale image I and

a reconstructed image K of size $m \times n$, the PSNR is defined as:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I_{ij} - K_{ij}]^2 \tag{1.19}$$

$$PSNR_{\text{grayscale}} = 10 \cdot \log_{10} \left(\frac{\text{MAX}(I)^2}{MSE} \right) \tag{1.20}$$

With $\text{MAX}(I)$ the dynamic range corresponding to $2^{\text{bit\_depth}} - 1$. If the image has a bit depth of 8 bits, the dynamic range is 255.
A PSNR between $30dB$ and $40dB$ is considered good quality for lossy compression. Above $40dB$ it is considered near lossless, while below $30dB$ corresponds to a visible degradation of the original image.
Similarly, we can define the PSNR of an RGB colored image as:

$$MSE_{\text{RGB}} = \frac{1}{3mn} \sum_{c \in \{r,g,b\}} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I_{c,ij} - K_{c,ij}]^2 \tag{1.21}$$

$$PSNR_{\text{RGB}} = 10 \cdot \log_{10} \left(\frac{\text{MAX}(I)^2}{MSE} \right) \tag{1.22}$$

SSIM the SSIM gives a measure of the similarity between two images. The SSIM is evaluated on M, $(N \times N)$ windows of the original and reconstructed image, respectively I_j and K_j, it is defined as:

$$\text{SSIM}(I_j, K_j) = \frac{\left(2\mu_{I_j}\mu_{K_j} + c_1\right)\left(2\sigma_{I_j K_j} + c_2\right)}{\left(\mu_{I_j}^2 + \mu_{K_j}^2 + c_1\right)\left(\sigma_{I_j}^2 + \sigma_{K_j}^2 + c_2\right)} \tag{1.23}$$

With the terms:

- μ_{K_j},μ_{I_j}: the mean pixel value of the current window j
- σ_{K_j},σ_{I_j}: the standard deviation
- $\sigma_{K_j I_j}$ the covariance of K_j and I_j
- $c_1 = (k_1 \cdot \text{MAX(I)})^2$, $c_2 = (k_2 \cdot \text{MAX(I)})^2$ with $k_1 = 0.01$ and $k_2 = 0.03$

Finally we compute the SSIM of the whole image as the mean SSIM (MSSIM) by taking the mean of the SSIM of each block. Considering that the image is split in M blocks the MSSIM is given as:

$$\text{MSSIM}(\text{I}, \text{K}) = \frac{1}{M} \sum_{\text{j}=1}^{M} \text{SSIM}(\text{I}_\text{j}, \text{K}_\text{j}) \tag{1.24}$$

The MSSIM ranges from 0 to 1. With 1 meaning a perfect match between the two images, generally a SSIM above 0.97 is considered good quality.

Now that we have discussed the JPEG compression scheme, we will discuss the existing hardware, either analogic or digital, implementations in the literature with a focus on power consumption.

Chapter 2

Analog and digital 2D-DCT in the literature

2.1 Energy consumption

From the previous description of the JPEG stages, we can estimate the power consumption breakdown of each stage based on the operations performed by each of them. We can model each stage by the model given in [30, 25], these models give us some insight on which operation is the most power hungry/needs the most processing, but they are a very weak approximation of the effective power consumption since it is very dependent of the type of either hardware or software implementation. We can estimate the total energy consumption E_p as:

$$E_p = E_{dct} + E_q + E_{zz} + E_{Huff} \tag{2.1}$$

With $E_{dct}, E_q, E_{zz}, E_{huf}$ the energy consumed for the DCT, quantization, zig-zag and Huffman coding that we will estimate later on.

DCT We recall from previous section, the 2D DCT is given as

$$Y = CXC^T \quad \text{with} \quad Y, C, X \in \mathbb{R}^{N \times N} \tag{2.2}$$

There are N^2 coefficients in Y, each requires N multiplications and (N-1) addition two times or N MAC operations. We can then estimate the energy consumption of the DCT of an image of resolution $k \times k$ which has $\left(\frac{k}{N}\right)^2$ blocks of size $N \times N$ as:

$$E_{dct} = 2\left(\frac{k}{N}\right)^2 N^2 \left(N \cdot e_{MAC}\right) \tag{2.3}$$

with e_{MAC} the energy required to perform a multiplication then an addition.

Quantization Two operations are performed on each coefficient of the DCT Y: division and rounding such that the total energy consumption of this step is:

$$E_q = \left(\frac{k}{N}\right)^2 N^2(e_{div} + e_{rounding}) \tag{2.4}$$

with:

- e_{div} the energy required to perform a division;
- $e_{rounding}$ the energy required to perform a rounding.

Zig-zag This operation is is a simple rearrangement of the $(N^2 - 1)$ AC coefficients from low to high frequencies. Each AC coefficient is shifted to its position, thus we consider a shift operation. The total energy consumption of this step is then:

$$E_{zz} = \left(\frac{k}{N}\right)^2 (N^2 - 1)e_{sh} \tag{2.5}$$

with:

- e_{sh} the energy required to perform a shift operation

Huffman encoding We have the two different energy models for the AC and DC coefficients:

$$E_{Huff,DC} = (2e_{fetch} + e_{write} + e_d) \tag{2.6}$$
$$E_{Huff,AC} = m(2e_{fetch} + e_{write}) \tag{2.7}$$

with:

- e_{fetch} the energy required to perform a read in memory;
- e_d the energy required to perform a subtraction;
- e_{write} the energy required to perform a write in memory;
- m amount of couples (RUN LENGTH, AMPLITUDE) of the current block.

We can see that we have two fetch and one write operation. From the previous description of the Huffman encoding, we have to look into two tables in order to correctly encode a value. The first fetch is needed to find the category of the coefficient, corresponding to the number of bits required to store the coefficient. The second fetch is required to read the effective Huffman code corresponding to (RUN

LENGTH, AMPLITUDE) for AC coefficients and to the difference of amplitude between near block for the DC coefficients.
For the whole image we have the total energy consumption of the Huffman encoding step:

$$E_{Huff} = \sum_{j=1}^{(k/N)^2} E_{Huff,DC,j} + E_{Huff,AC,j} \tag{2.8}$$

Total Energy consumption Considering the DCT is computed on a digital processor, we can define the energy of each operation as the product of the amount of cycles required for this operation by the energy consumed per cycle[1].

$$e_{operation} = N_{cycle} e_{cycle} \tag{2.9}$$

From the ARM documentation on cycle per operation for cortex-M4 processors [3] we have the cycle count per operation given in table 2.1. With the information in table 2.1 and the model to estimate the power consumption of the JPEG algorithm, considering that a load and store in memory is performed for each MAC operation of the DCT and taking the worst-case Huffman encoding without any run length, thus $m = 63$, we obtain an estimate of the cycle count for a single block of each stage in table 2.2.

Operation	Assembler	Cycle Count
Multiply then accumulate float	VMLA.F32	3
Divide float	VDIV.F32	14
Load memory float	VLDM.32	1
Store memory float	VSTM.32	1
Substract float	VSUB.F32	1
Move float	VMOV	1

Table 2.1: Cortex-M4 operations cycle counts

Step	Cycles estimation	Fraction estimation[%]	Energy consumption from [30] [pJ]	Proportion from [30] [%]
DCT	2560	68.95	1575936	62.13
Quantization	896	24.13	164554	31.02
Zig-Zag	64	1.72	786739	0.37
Huffman	193	5.20	9324	6.49

Table 2.2: Cycles count estimation and proportions comparison from literature

Using the power consumption model[30, 25] given above and comparing with the power consumption of a digital implementation[21, 30] [2] found in the literature, we summarize the results in table 2.2. With these results we obtain an approximate power consumption repartition per stage in Fig. 2.1. From this plot

[1] In practice, a memory access will consume more than a mathematical operation per cycle.
[2] Measurements come from the average of multiple ARM processor

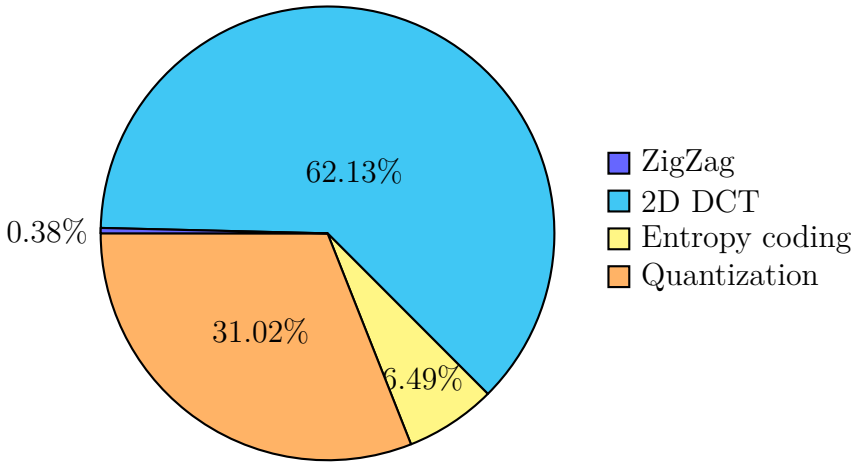


Figure 2.1: Power repartition estimation

we can clearly see that the DCT and the quantization step dominate since the first requires a lot of multiply and accumulate (MAC) operations and the second requires division operations. With these results in mind, we will further focus on the 2D-DCT step and we will discuss how it can be implemented in the analog domain to reduce power consumption and compare it with different digital implementation. A few example, and their performance, of 2D DCT processor found in the literature are given in table 2.3. Comparing the performance of each implementation is quite difficult since they all approach the problem differently and with different constrains. Furthermore, they often do not give sufficient details to correctly evaluate the power consumption of their implementation.

2.2 Implementation comparisons

We will now discuss in more details implementations of the 2D-DCT in the digital and analog domains. Table 2.3 regroups some interesting example, and their performances, of 2D DCT core found in the literature. We defined the two figures of merits FOM_1, the processing energy required per pixel, and FOM_2, FOM_1 scaled to the smallest technology of $65nm$, as:

$$\text{FOM}_1 = \frac{\text{Energy per frame}}{\text{Frame resolution}} \cdot 1000 \tag{2.10}$$

$$\text{FOM}_2 = \frac{1}{\alpha^3}\,\text{FOM}_1 \tag{2.11}$$

With $\alpha_{0.18\mu m \to 65nm} = \dfrac{180nm}{65nm} = 2.77$ and $\alpha_{0.35\mu m \to 65nm} = 5.384$.

	[21]	[31]	[33]	[18]	[14]	[29]	[28]	[19]
Year of publication	2006	2017	2019	2013	2021	2006	2010	1997
Area [mm^2]	0.389	2.462	1.21	0.0667	N/A	0.03	N/A	1.1
Number of transistors per DCT unit	N/A	N/A	N/A	N/A	1216	3066	2752	N/A
Transform duration [ns]	N/A	128	N/A	12207	894	2500	70	2700
Clock Frequency [MHz]	12.58	125	6	10	N/A	N/A	N/A	0.76
Average power [mW]	1.53	12.76	0.2395	0.010	0.184	2.5	3.28	5.4
Energy per frame [nJ]	N/A	52.26[a]	16349[b]	0.12	2.6	6.5	0.2296	14.5
PSNR [dB]	32.51	35.9	34.52	22.4	39.6	30	25.6	31.4
FOM_1 [pJ/pixel]	N/A	25.52	159.66	1.875	40.63	101.56	3.59	226.57
FOM_2 [pJ/pixel]	N/A	N/A	7.52	0.26	40.63	4.78	0.17	1.45
Technology	0.18μm	Virtex5 FPGA	0.18μm	0.18μm	65 nm	0.18μm	0.18μm	0.35μm
Frame resolution	512x512	256x256	320x320	8x8	8x8	8x8	8x8	128x128
Implementation type	Digital FPGA	Digital FPGA	Digital ASIC	Analog charge sharing Charge mode	Analog current mirrors Current mode	Analog current mirrors Current mode	Analog current mirrors Current mode	Analog switched capacitor Voltage mode
DCT type	Integer	Integer	CORDIC	2 stage matrix	1 stage matrix	2 stage matrix	2 stage matrix	2 stage matrix
Measurement type	Simulation	Measurements	Measurements	Simulation	Simulation	Measurements	Simulation	Simulation

Table 2.3: State of the art of digital and analog implementation of 2D-DCT

[a] $= 12.76mw \cdot 16ns \cdot 256\text{pixel} = 204e-12 \cdot 256$

[b] 4 cycles per pixels

Digital

The three digital implementations approach the computation of the DCT differently, either by using an algebraic integer implementation which simplifies the DCT to simple integer addition and bit shifts [24] without multiplications for [21, 31]. Or by approximating the DCT with an coordinate rotation digital computer (CORDIC) algorithm [34] in [33]. In either case, their power consumption is significantly higher than any of the 5 analog implementation. We will not further discuss digital implementations but explore in more depth the analog ones.

Analog

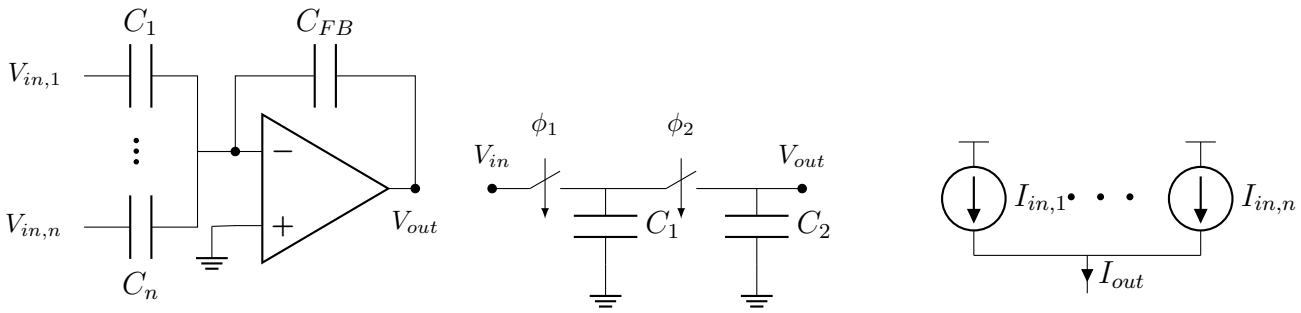
[16, 20] As seen in the previous sections, the DCT, in each of its form, consists only of MAC operations, this operation is achieved in the analog domain by one of the following way:

- Current mode: current mirrors for multiplication and Kirchhoff's current law (KCL) for addition [14, 29, 28]
- Charge mode: addition and multiplication with switched capacitor (SC) circuits [19, 18]
- Voltage mode: multiply and add realised with opamp amplifiers[32]
- Other approaches like translinear[23] or time-based[16] are not discussed in this work.

Each mode is represented in Fig. **??** with its transfer function. Fig. 2.2a demonstrates a voltage mode MAC unit in a single cell. In Fig. 2.2b only the multiplication is performed but addition could be handled in a similar way by loading capacitors indepently then connecting them together to perform charge sharing between them which would results as the addition of the charges. Fig. 2.2c demonstrates a current based approach where each current source can be seen as a current mirror, multiplication is done by scaling of the transistors by the same coefficients needed for the multiplication as shown in Fig. 2.3 while addition is simply performed with KCL.

Each way has its own advantages and inconvenient that we gathered from the literature in table 2.4

Also, the algorithm used to compute the 2D-DCT has an impact on the analog implementation. Mainly the most common 2D-DCT algorithms found in the literature are



$V_{out} = \sum_{i=1}^{n} \frac{C_i}{C_{FB}} V_{in,i}$

(a) Voltage mode

$V_{out} = \frac{C_1}{C_1 + C_2} V_{in}$

(b) Charge mode

$I_{out} = \sum_{i=1}^{n} I_{in,i}$

(c) Curent mode

Figure 2.2: Simple analog MAC units

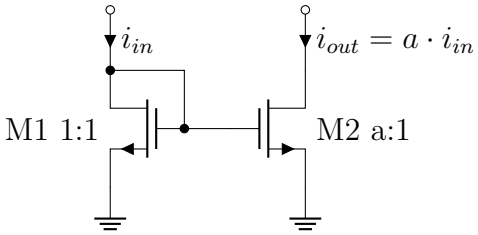


Figure 2.3: Curent mirror multiplication

	Current mode	Charge mode	Voltage mode
Advantages	• Precise with class AB current mirrors	• simple • Higher bandwidth/speed	• ratio of capacitors
Disadvantages	• Mismatch in current mirrors • Higher power consumption due to current sources	• Switching error • Small capacitor sensitive to missmatch	• Need an OTA (Noise, bandwidth,...)

Table 2.4: Comparison of analog MAC units implementations

- Row/column separation algorithm: implemented in 2 stages 1D-DCT core with a transposing analog memory in between as depicted in Fig. 2.4 [29, 28, 19, 18]
- Direct 2D-DCT: directly compute the 2D-DCT in a single stage [14]

In table 2.5 we compare the two algorithms and their analog implementations from the literature. In short, the two stages implementations require less area, are more prone to cumulative errors while single stage are more robust to errors but require more area.

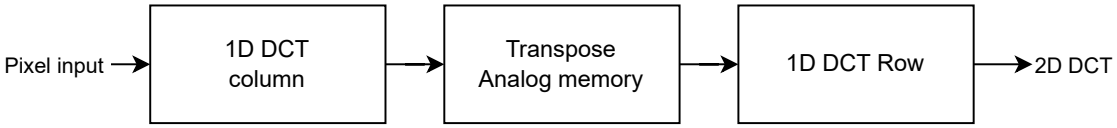


Figure 2.4: 2 Stage analog 2D DCT

	2 stage Row/Col separation	1 stage directe 2D-DCT
Advantages	• smaller area	• No cumulative errors Single stage
Disadvantages	• Sequential • Prone to cummulative errors	• Higher area More operations
MAC operations [#]	N^3	N^4

Table 2.5: Comparison of 2D-DCT algorithm and implementation (block size $N \times N$)

From the results gathered in table 2.5, 2.3 and 2.4 we can see that multiple attempts to perform the 2D-DCT in the analog domain have been tried, with either good PSNR or low power consumption. By comparing the FOM_2 and PSNR in table 2.3 of each implementation, we can see that the voltage mode approach offers a good trade-off between image quality and power consumption but no recent research has been conducted in this direction. In the following sections we will describe an implementation of the 2D-DCT in voltage mode with MANTIS by first exploring some approximations of the 2D-DCT and their impact on image quality. The approximation that we will discuss will use the fact that we can diminish the amount of unique coefficients to compute the DCT such that it will reduce the area required to store these coefficients. This tradeoff will be later studied.

Chapter 3

2D-DCT approximation

3.1 Data set description

In this work, we used the six well-know grayscales standard images in Fig. 3.1 with a resolution of 128×128 pixels and a bit depth of 8 bits.

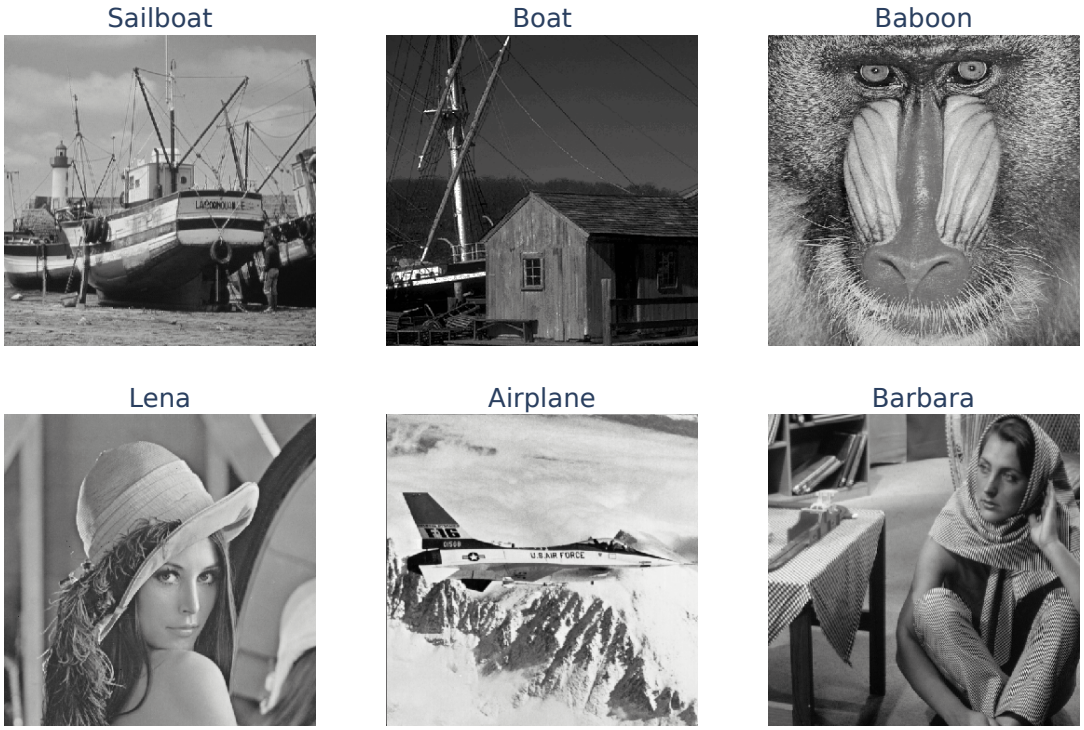


Figure 3.1: Picture data set of 128×128 grayscale pictures

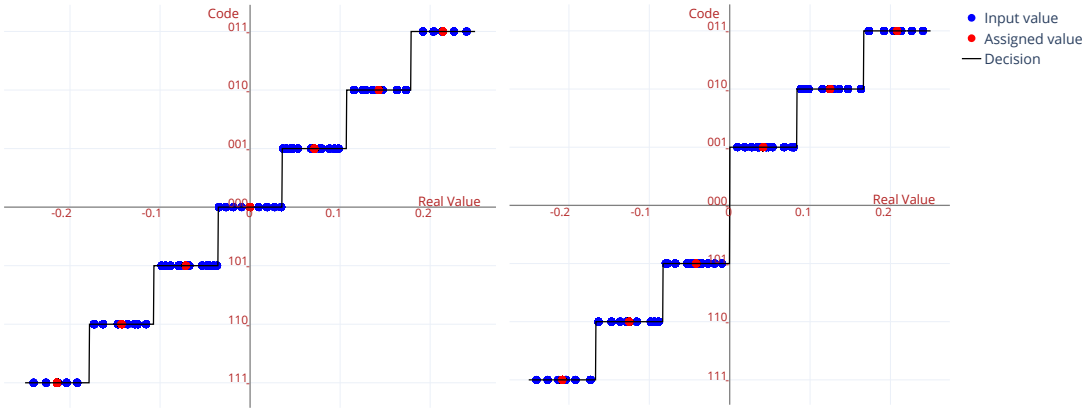
3.2 Weights quantization

In Section 1.4 we have shown two ways to compute the 2D-DCT. Let us call them the matrix and tensor form. Equations are recalled just below.

	Matrix	Tensor
2D-DCT	$Y = CXC^T$	$Y_{pq} = A_{pqij}X^{ij}$
2D-IDCT	$X = C^TYC$	$X_{pq} = A'_{pqij}Y^{ij}$

The coefficients tensors $C \in R^{N\times N}$ and $A \in R^{N\times N\times N\times N}$ which are real numbers respectively comprised in $C_{pq} \in [-0.5, 0.5]$ and $A_{pqij} \in [-0.25, 0.25]$. Since they cannot be represented with an infinite precision, they must be quantized in some way. Let us observe the impact of quantizing the coefficients by reducing the amount of bits to represent them.

We considered a mid-tread and mid-rise uniform quantization with 1 to 10 bits, plus 1 bit for the sign. The major difference between these two types of quantization is that the mid-rise one does not include zero as a possible step. For $N = 8$ we have respectively 8 and $\frac{8 \cdot 7}{2} = 28$ different values. We represented the two types of quantization in Fig. 3.7 for the tensor approach.



(a) (b)

Figure 3.2: Mid-tread (3.2a) and mid-rise (3.2b) uniform quantization for the 56 unique coefficients of tensor A quantized with 2+1 bits. In blue the real coefficients and their assigned code. In red the retrieved real value after quantization. The code is given as a 3-bit string with 1 as MSB for negative value and 0 for positive ones

With these modified coefficient values by the process of quantization, we have

to evaluate the performance of the new transformation with the approximated coefficients tensors or $\hat{A}$. First, we briefly describe how the quality of the approximation can be measured.

Methodology to estimate the approximation quality

We know that the DCT has a high energy compaction capability, and this is a quality we would like to preserve even with approximation. We can follow the methodology suggested in [15], in which the spectral structure is assessed. We consider each row of the coefficient matrix C as the coefficients of a FIR filter. Similarly for the tensor representation we can change the tensor A to a two-dimensional tensor $T \in R^{N^2 \times N^2}$ by flattening each $N \times N$ filter to a row of the reconstructed matrix as shown in Fig. 3.4. The resulting matrix is given in Fig. 3.3.

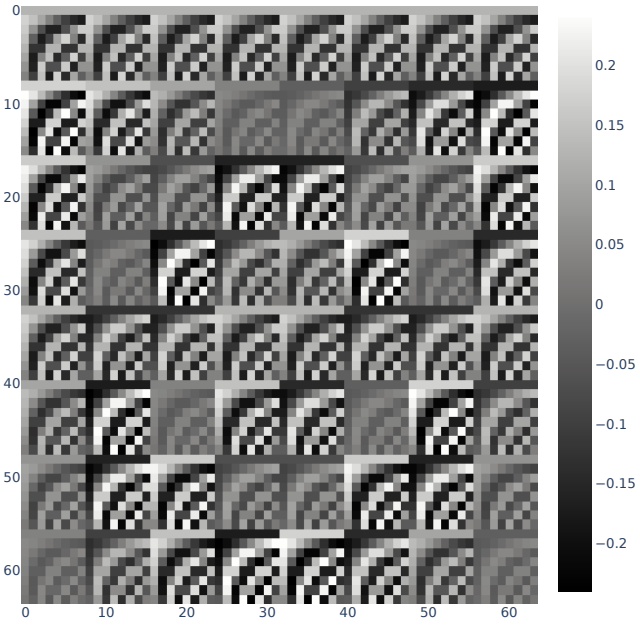


Figure 3.3: $T \in \mathbb{R}^{64 \times 64}$, the flattened tensor A for $N = 8$, each row corresponding to a filter.

Then we can obtain the transfer function related to each row of T

$$H_k(\omega, T) = \sum_{n=1}^{64} T_{kn} \exp\left(-jn\omega\right) \quad k = 1, .., 64 \tag{3.1}$$

with $\omega \in [0, \pi]$.
We define an energy-related figure of merit as the square of the difference norm of

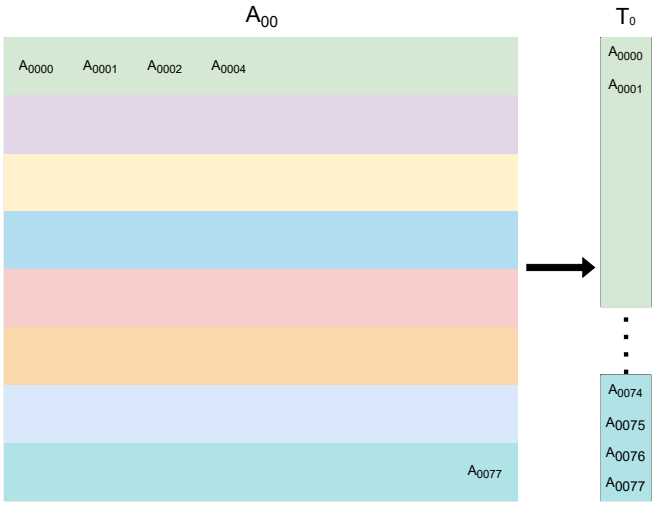


Figure 3.4: Method to flatten each $N \times N$ in A to a row of matrix T

the original DCT coefficients and their approximation.

$$D_k(\omega, T_1, T_2) = |H_k(\omega, T_1) - H_k(\omega, T_2)|^2 \tag{3.2}$$

Where T_1 is the original tensor and T_2 the approximation. This gives us an idea of how well the approximation retains the spectral property of the original transform. In Fig. 3.5 we compute D_k for an approximation with 1, 2 and 6 bits for some values of k, we can observe that the 6 bits is nearly exact as the original while the 1 and 2 bits present some differences.

We can finally define the total spectral error quantity as

$$\delta(T_1, T_2) = \sum_{k=1}^{64} \int_0^{\pi} D_k(T_1, T_2) d\omega \tag{3.3}$$

Simulation

With this metric we can evaluate each quantization method with respect to their resolution in 3.6 where we can see that the mid-tread better preserves the spectral structure at low resolution compared to the mid-rise one. Also, we can observe that the total error δ decreases with the number of bits used for the approximation. The metric δ does not mean anything by itself, and should be used to **compare** different approximations. For the purpose of setting a reference, we compute the total spectral error of the original 2D DCT matrix T with the following transform matrices; a constant matrix equal to 0.25, a random matrix uniformly distributed between 0 and 0.25 and a zero matrix. Those transform have large total spectral errors of respectively: 992.73, 465.83 and 201.06. This helps support δ as a good metric to evaluate the quality of an approximated transformation.

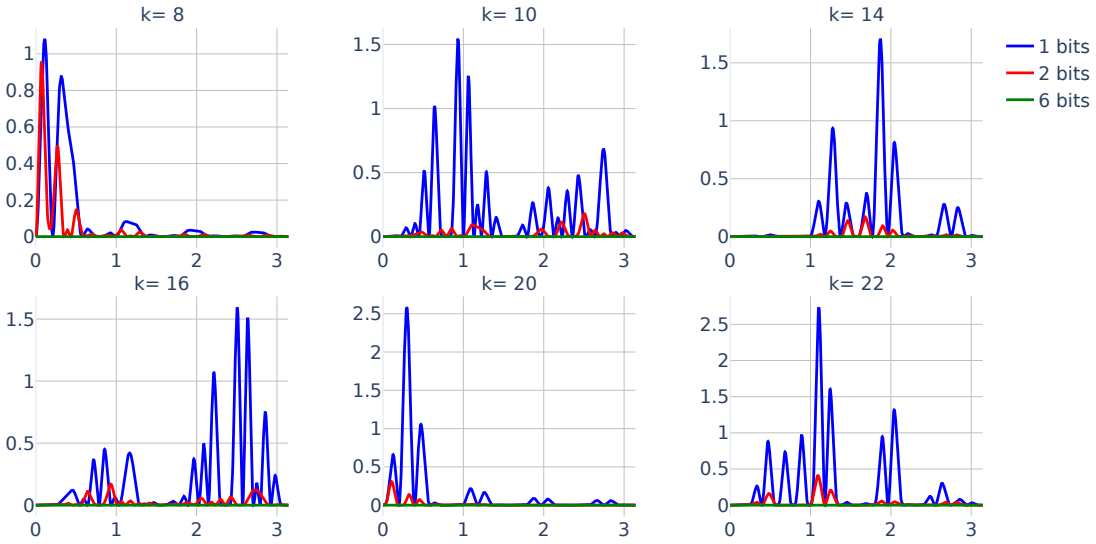


Figure 3.5: Spectral structure comparison: D_k in function of ω for 1-,2- and 6-bits 2D DCT approximation.

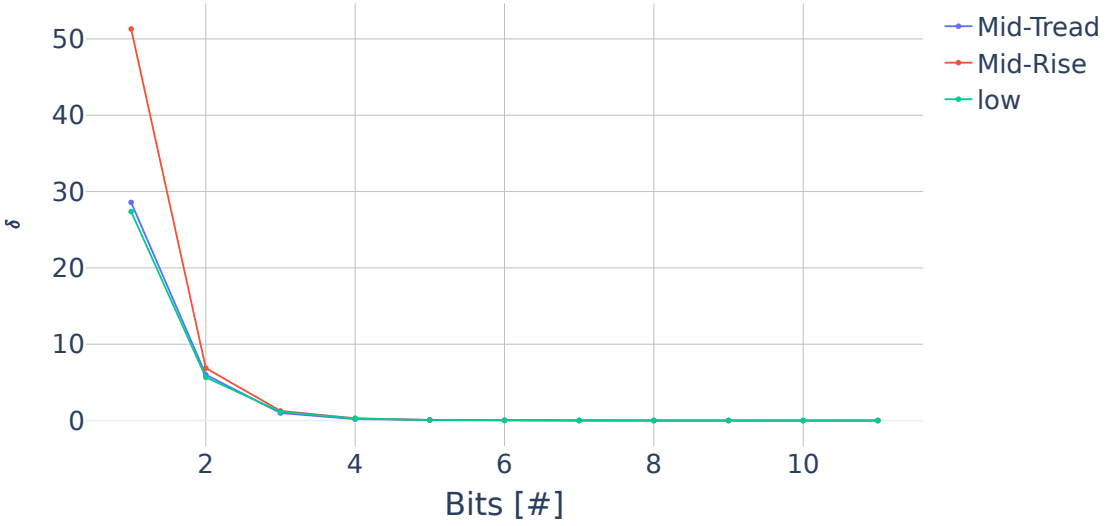
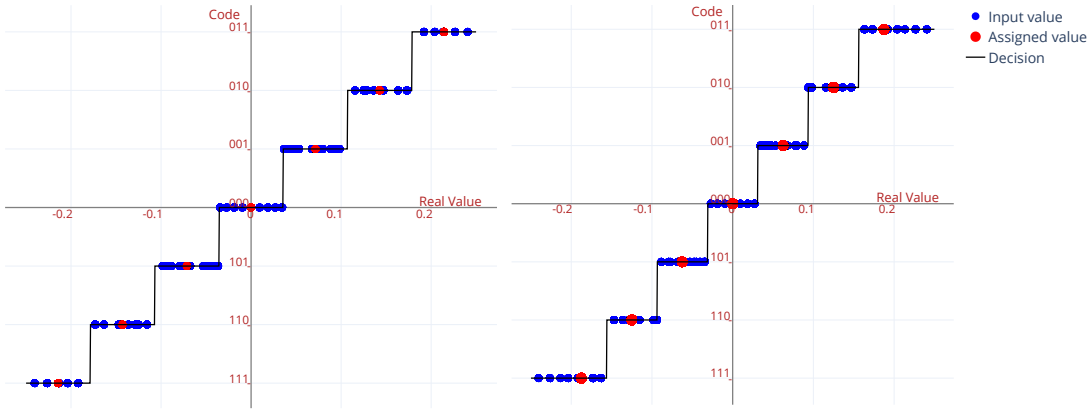


Figure 3.6: Total spectral error δ for mid-tread, mid-rise and low quantization scheme with respect to 2D DCT coefficients quantization resolution in bits.

Furthermore, after some experimentation with the placement of the decisions steps in the quantization we were able to obtain slightly better performances on average with the quantization scheme given in 3.7b. This proposed quantization scheme has smaller quantization steps closer to zero, which avoids having too many

values assigned to zero, at the compromise of a larger error at extreme values.



(a) (b)

Figure 3.7: Mid-tread (3.2a) and low (3.2b) uniform quantization. The quantization step is smaller for the low method resulting in less values assigned to zero and more to the last step.

Conclusion

Increasing the amount of bits used to represent the coefficients past a certain point does not have a large impact on the spectral structure of the 2D DCT, and even a low resolution yields acceptable performance. We can safely guess that the energy compaction of the DCT will not be impacted either when using low resolution coefficients. We will strengthen this assumption with the following developments. From now on, we will use the quantization scheme given in Fig. 3.7b.

3.3 Hard thresholding

We will now see the impact of hard thresholding high frequencies components of the 2D-DCT to zero. The idea behind hard thresholding is that high frequency components contain little information, which can be discarded. In practice, we keep the n first coefficients of the 2D-DCT in zig-zag order and set the remaining coefficients to zero, this corresponds to coefficient tensor $A_{\mathrm{HT},n}$ similar to the one given in 3.8.

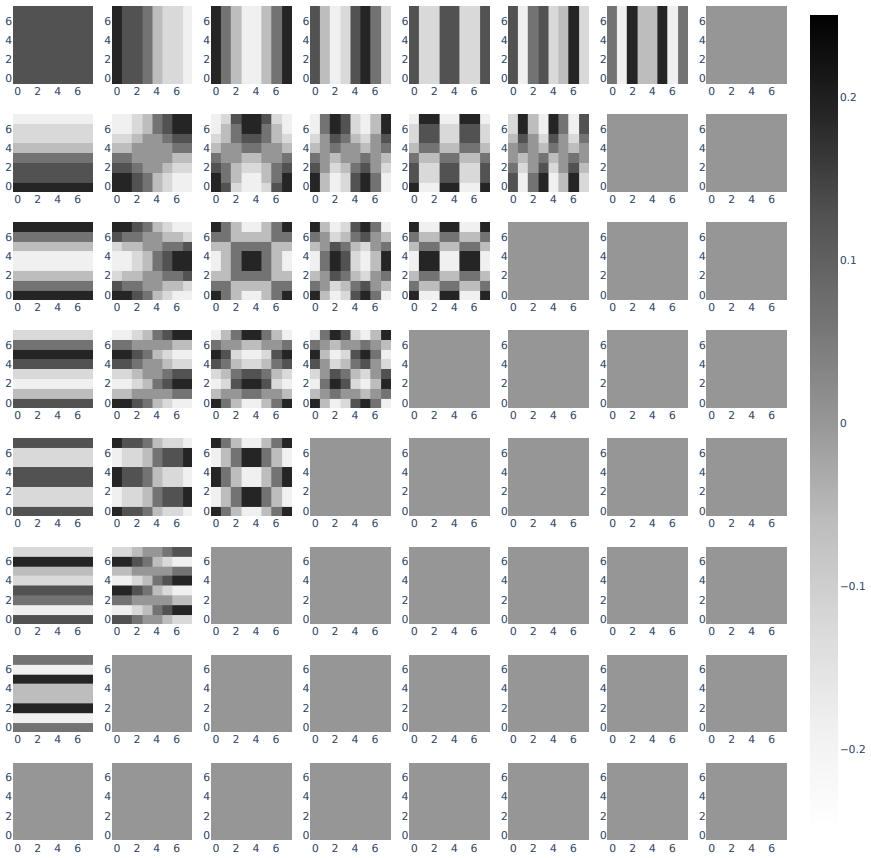


Figure 3.8: Coefficient tensor $A_{\mathrm{HT},28}$ keeping 28 coefficients matrices corresponding to the 28 lowest frequencies terms of the 2D-DCT and setting the remaining coefficients to 0.

3.3.1 Simulation

To measure the impact of hard thresholding on image quality, we compute the forward 2D DCT with the hard threshold tensor $A_{\mathrm{HT},n}$ on the test images, then we compute the inverse 2D DCT with the full inverse tensor A'. We can then observe the evolution of PSNR and SSIM in Fig. B.1, where we can see that with half the coefficients of the DCT discarded we keep a good image quality with a PSNR of 35.5 dB and 95.5% SSIM. We can also observe, in Tab. 3.1, that the degradation in PSNR between the 2+1 bits and 10+1 bits coefficients resolution is negligible.

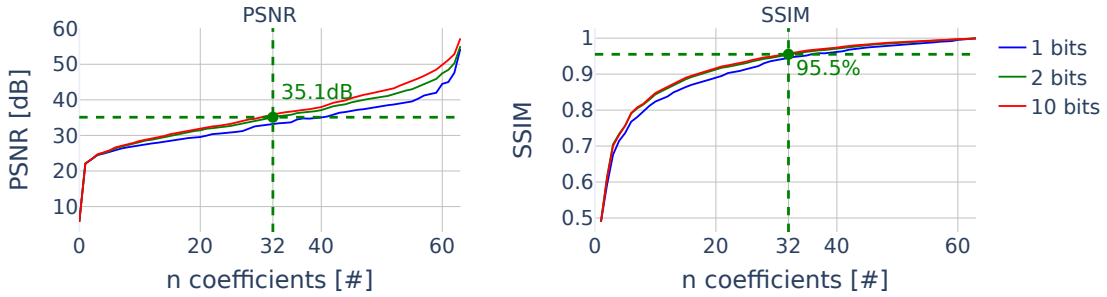


Figure 3.9: Evolution of average PSNR and SSIM for different hard thresholding level and bit resolution approximations

	PSNR [dB]	SSIM [%]
1 bit	33.24	94.6
2 bits	35.11	95.5
3 bits	35.74	95.8
10 bits	35.91	95.8

Table 3.1: PSNR and SSIM for $n = 32$ coefficients with 1-,2-,3- and 10 bits coefficients resolution.

3.3.2 Conclusion

From the previous results we can see that we can keep a PSNR above 30 dB with only the 15 first zig-zag coefficients approximated with only 2+1 bits. From now on, unless told otherwise, we will be working with $n = 32$ coefficients and tensor $A = A_{\mathrm{HT},32}$ with a resolution of 2+1 bits.

3.4 Noise

The previous sections have covered approximations and considerations which are useful for an analog implementation. However, in the analog domain we have to deal with additional non-idealities such as noise, mismatch and process variations that lead to small variations around the expected values. This section is dedicated to an analysis of their impact on the coefficients of tensor A and, in turn, the consequences on the quality of the reconstructed image in order to budget the maximal tolerated error.

We can consider two approach for modeling the perturbation of the coefficients:

1. Per coefficient error
2. Fully random error

In a first case, we consider that the same coefficients in A are subject to random but systematic errors. We add an error term to each of the 56 **unique** coefficients. This could represent the error caused by process variation in the DAC used in a DCT unit where identical coefficients are also impacted in an identical way, i.e. with the same error.

In a second case, we add an error tensor $E \in \mathbb{R}^{N\times N\times N\times N}$ to the coefficients tensor A. In that case, identical coefficients can have different noise values. This is a more general consideration of errors as we do not make the previous assumption that the same coefficients will have the same error term. This representation is more realistic in the case where multiple DCT unit work in parallel and the error is not systematic.

We consider an additive gaussian error with zero mean $e \sim \mathcal{N}(0, \sigma^2)$ of variance σ^2 thus we construct the noisy tensor in the first case with the noisy coefficient:

$$a_{n,i} = a_i + e_i \quad \text{with } a_i = \text{UNIQUE}(A) \quad \forall i \tag{3.4}$$

With the UNIQUE operator returning the unique values contained in A.

For the second case we add a noise tensor to the coefficient tensor. Taking the 2D DCT transform in a single matrix T as described in Fig. 3.3.

$$T_n = T + E \quad \text{with } E \sim \mathcal{N}(0, \sigma^2) \in \mathbb{R}^{N^2\times N^2} \tag{3.5}$$

Thus the impact on the error term on the reconstructed image will be[1]

$$X' = T^T Y = T^T(T + E)X = X + T^T EX \tag{3.6}$$

Error term

We can see that an error term is added to the reconstructed image.

3.4.1 Simulation

To evaluate the impact of an error term we perform an analysis with the Monte Carlo[41] method with 50 random samples[2]. Let us first define the tolerance t such that each coefficient lies with 95% (2σ) confidence within that tolerance. For example if $a_n \pm 1\%$ it means

$$a_n = a + e \sim \mathcal{N}\left(a, \left(\frac{0.01a}{2}\right)^2\right) \tag{3.7}$$

[1] It is worth recalling that T is orthonormal such that $T^{-1} = T^T$.

[2] The random sampling is performed for each block.

Remark: sum of IID random variable

Beforehand we can prove that the fully random approach will have less impact on the PSNR than the per coefficient approach, we will briefly discuss this.

First we know that each term of the 2D-DCT can be obtained as the sum of product of a coefficient a_{kij} with a pixel x_{ij}:

$$y_k = \sum_{i=1}^{8}\sum_{j=1}^{8} a_{kij}x_{ij} \tag{3.8}$$

From the fully random approach and knowing that the mean and variance of a random variable X multiplied by a constant K is given as $\mathrm{E}[KX] = K \cdot \mathrm{E}[X]$ and $\mathrm{V}[KX] = K^2 \cdot \mathrm{V}[X]$ we have:

$$a_{n,kij} \cdot x_{ij} \sim \mathcal{N}\left(a_{n,kij}x_{ij}, (x_{ij} \cdot \sigma)^2\right) \tag{3.9}$$

Since the random variables $a_{n,kij}$ are independent we have that the mean and variance of the sum of independent Gaussian variable is the sum of individual mean and variances such that:

$$y_{n,k} = \sum_{i=1}^{8}\sum_{j=1}^{8} a_{n,kij}x_{ij} \sim \mathcal{N}\left(\sum a_{n,kij}x_{ij}, \sigma^2 \sum x_{ij}^2\right) \tag{3.10}$$

For the per coefficient approach, for ease of development let consider a single coefficient $a_n = a + e \sim \mathcal{N}(a, \sigma^2)$ such that:

$$y_{n,k} = \sum_{i=1}^{8}\sum_{j=1}^{8} a_{n,kij}x_{ij} = a_{n,k}\sum_{i=1}^{8}\sum_{j=1}^{8} x_{ij} \tag{3.11}$$

We can take a_n out of the sum since we have the same error term added to the same coefficient, thus the distribution become:

$$a_{n,k}\sum_{i=1}^{8}\sum_{j=1}^{8} x_{ij} \sim \mathcal{N}\left(a_{n,k}\sum x_{ij}, \sigma^2(\sum x_{ij})^2\right) \tag{3.12}$$

By comparing the distributions given in 3.12 and 3.10, they have the same mean as expected bu the variance is a bit different. From the Cauchy-Schwarz inequality we have that

$$\sum x^2 \leq \left(\sum x\right)^2 \tag{3.13}$$

This explain why we have better performances with the fully random approach.

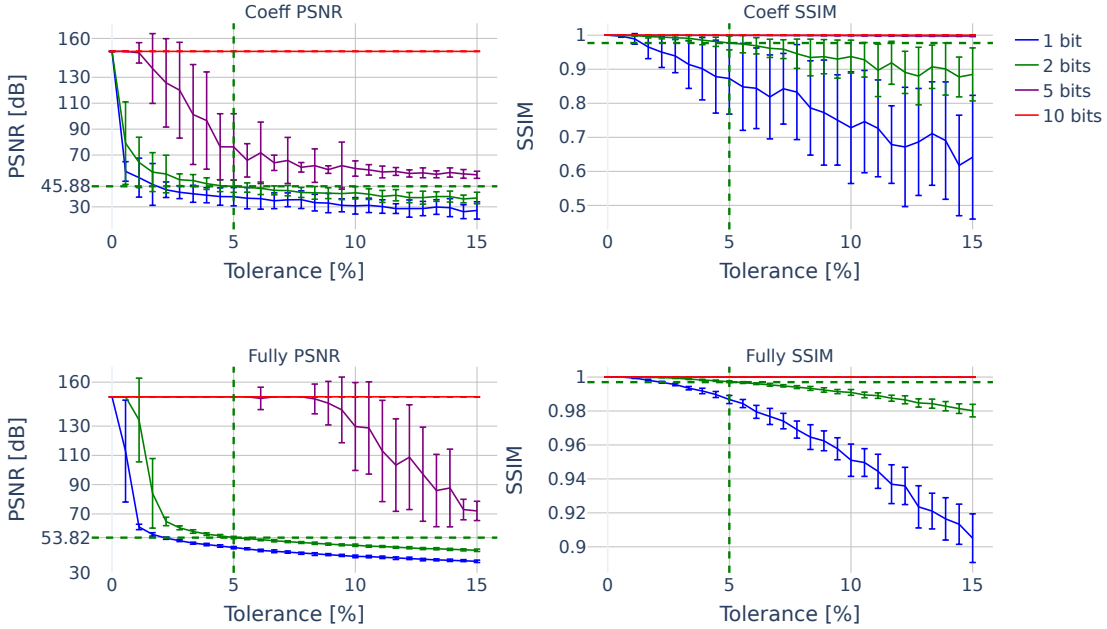


Figure 3.10: Monte Carlo Simulation with 50 draws of PSNR and SSIM for different tolerance values. We do not hard threshold the coefficient and infinite PSNR was clipped to 150dB.

Results

The results of the Monte Carlo simulation are gathered in Fig. 3.10. As expected, we can see that the PSNR and SSIM for the fully random approach is a bit higher than the per coefficient approach for the same tolerance level. Furthermore, the bit resolution has a big impact on the sensitivity of an error term. High resolution approximations are nearly insensitive to error terms while low resolution approximations are more sensitive. For 5% tolerance with 2 bits resolution we obtain PSNR of 53.82dB, which we can consider as good performance.

Also, we have only considered stochastic error since systematic errors could be compensated by a calibration phase for example by adapting the coefficients of the inverse DCT.

3.5 Results summary

We can combine the approximation and hard thresholding for a chosen tolerance level to observe the cumulative effects on PSNR and SSIM. The results are shown on Fig. 3.11. We can see that the degration of PSNR caused by considering a tolerance level of 5% on the coefficients value with hard thresholding with $n = 32$ is negligeable, it decreases the PSNR and SSIM by less than 1%. We could increase considerably the tolerance on the coefficients without degrading the image quality as shown in Fig. 3.12 for a 2 bits resolution and $n = 32$.

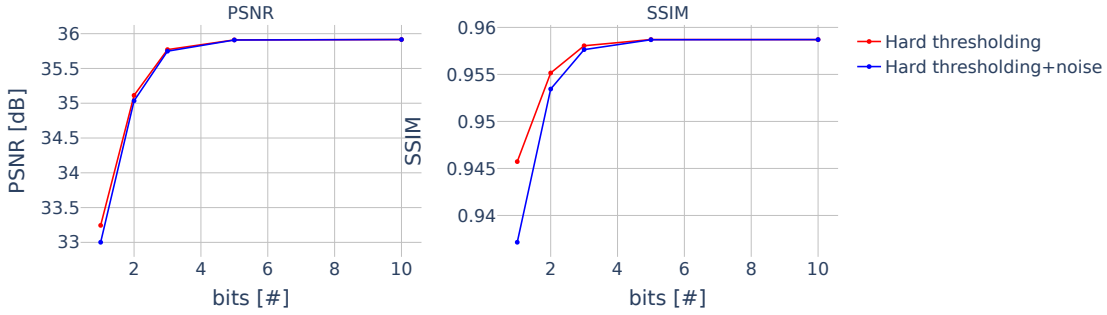


Figure 3.11: PSNR and SSIM of combined hard thresholding with $n = 32$ and $t = 5\%$ tolerance.

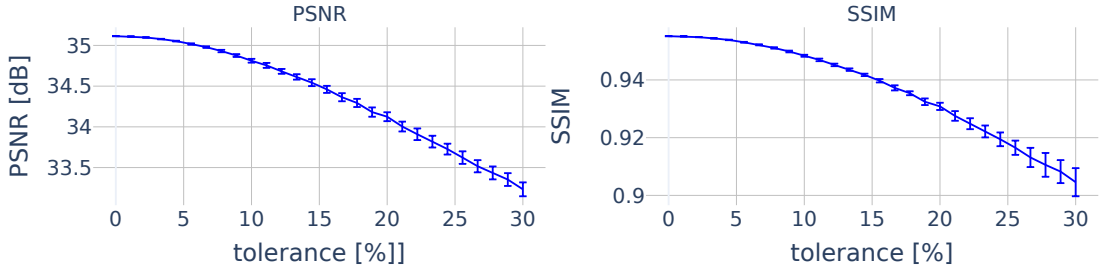


Figure 3.12: PSNR and SSIM of combined hard thresholding with $n = 32$ and $t = 5\%$ tolerance.

Finally we can observe qualitatively the impact of hard thresholding and noise in Fig. 3.13. With a tolerance of $t = 5\%$ for the middle image we can observe little degradation of the image, edges are less sharp and there are some graininess with a relatively strong PSNR of 39dB. For the third image, with a PSNR of 35.58dB, with $t = 30\%$ we can clearly see graininess on the zoomed image. The degradation of image quality is more noticeable.

Further analysis will be focused on the actual hardware implementations, and

in the next chapter we will introduce MANTIS, an SoC with SCAs MAC units.

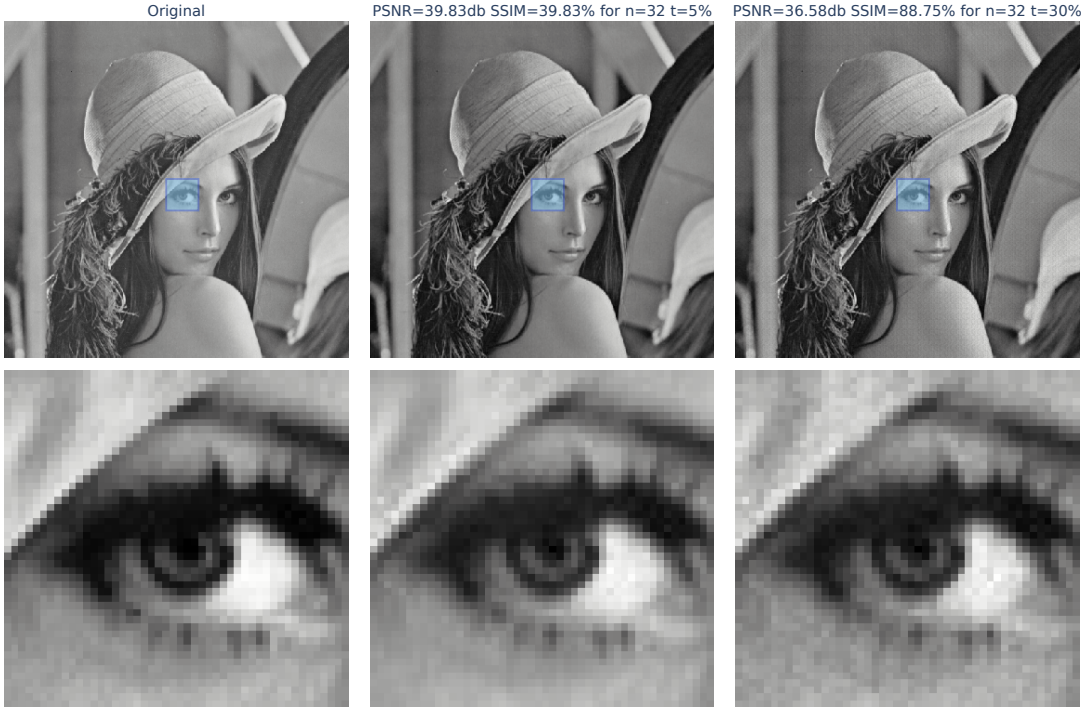
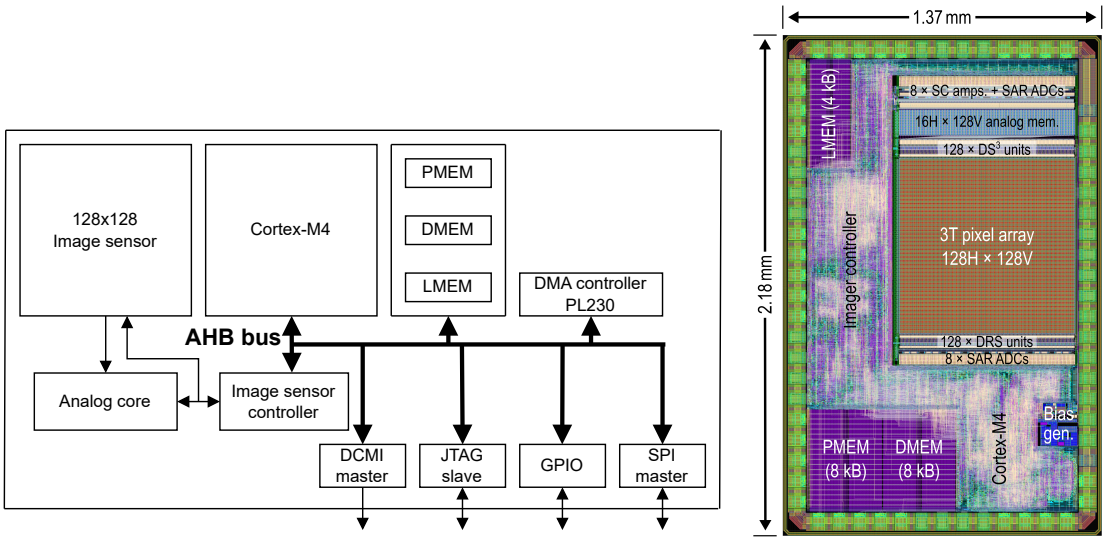


Figure 3.13: Original image and 2 compressed decompressed images with 2 bits approximation and $n = 32$. The middle image has $t = 5\%$, the third has $t = 5\%$. The second row is a zoom on the blue box of the full size images.

Chapter 4

Hardware implementation with MANTIS

MANTIS is an SoC built at ECS group at UCLouvain, a high-level diagram of the chip is given in Fig. 4.1b.



(a) MANTIS SoC high level diagram

(b) MANTIS SoC layout

It features:

- An 128×128 image sensor composed of an array of 3T APS pixels.
- An analog core capable of performing MAC operations with eight switched-capacitor amplifier (SCA) working in parallel on 16×16 blocks of pixels of the image sensor with configurable 3+1 bits multiplicative coefficients. It is

driven by a digital controller. It also embeds 8 SAR ADC's. The LMEM allows to store 32 16×16 coefficients matrix used in the MAC units.

- A program and data memory (PMEM and DMEM) of 2000 words of 32 bits (8kB)
- A local memory (LMEM) used to store the coefficients of the analog core for the MAC operations. It has a size of 512 words of 64 bits (2kB).
- A Cortex-M4 processor clocked at 4 MHz.
- A DMA controller which allows memory transfers between peripherals without the need of the Cortex-M4 processor.
- A master digital camera memory interface (DCMI) to transfer the pixels outside the SoC. Either raw from the image sensor or processed by the analog or Cortex-M4 processor.
- The processor, the image sensor controller, the memories and all the peripherals are connected with an advanced high-performance bus (AHB)[2] which is an on-chip bus protocol with a 32-bits bus width allowing fast transfers.

We will now describe in details the analog core, how we will use it to perform the 2D-DCT and its limitations.

4.1 Analog core

The analog core contains 8 independent MAC units each one made of a single-ended SCA (Fig. 4.2) and 16 programmable capacitors (C_i) for the multiplication (Fig. 4.8a) and a variable feedback capacitor (C_{FB}) to tune the gain (Fig. 4.8b). We can control the operation of the SCA handle positive and negative weights. Since the OTA is single ended, the zero is the common mode voltage $V_{CM} = V_{DD}/2$ and we can consider values below V_{CM} as negative and above as positive.

4.1.1 Switched-capacitor amplifier

In Fig. 4.2, we can see an SCA composed of an OTA, 16 input signals ($V_{in,i}$) with their respective programmable capacitors C_i and phase control signals $\phi_{1,i}$, $\phi_{2,i}$ controlling addition or subtraction and finally the variable feedback capacitor C_{FB}. The programmable capacitors are controlled digitally with by a 4-bits weight W[3:0] which activates or deactivates transmission gates to binary-scaled unitary capacitors as shown in Fig. 4.8. The MSB W[3] gives the sign of the weight,

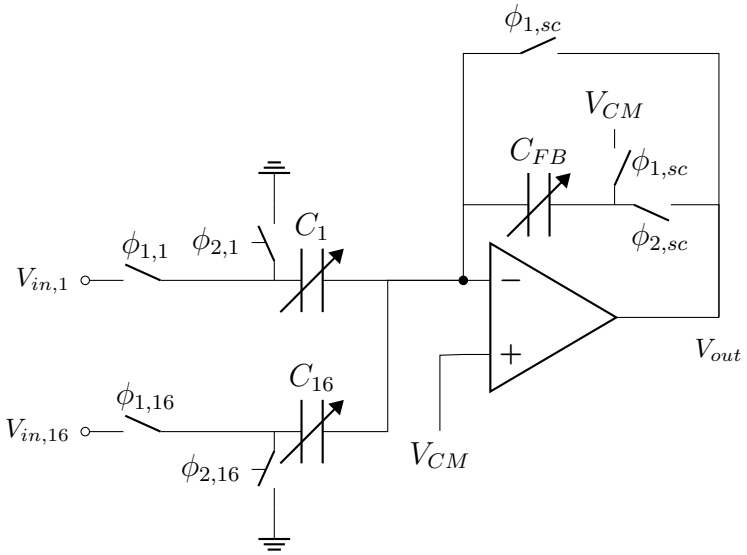


Figure 4.2: MANTIS switched capacitor amplifier

positive for W[3]=0 or negative for W[3]=1, and it will influence the control signal $\phi_{1,i}$ and $\phi_{2,i}$ of the corresponding input. The three LSBs control the value of C_i between 0 to $7C_u$.

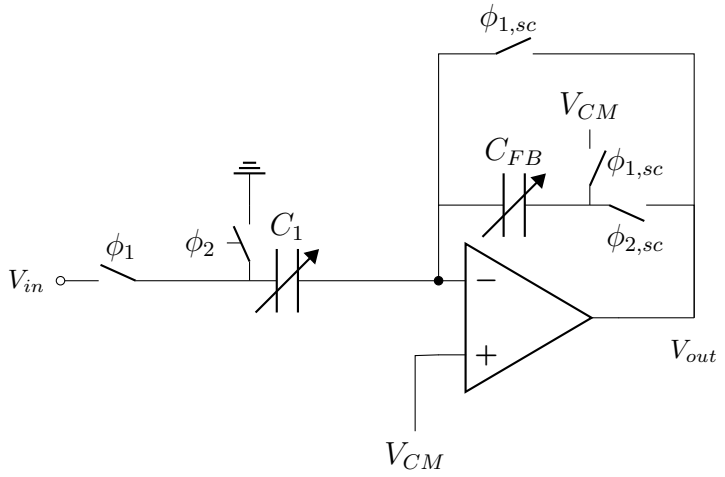
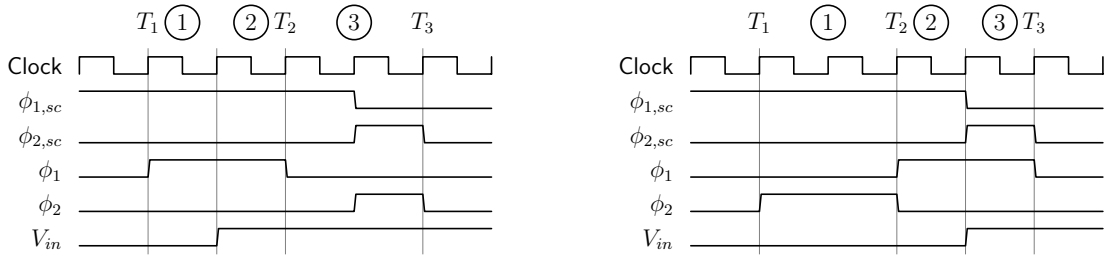


Figure 4.3: MANTIS switched-capacitor amplifier with a single input

To explain the working principle of the SCA either for addition or subtraction we will use a simplified SCA with a single input as depicted in Fig. 4.3 with the timing diagram of the signal $\phi_1, \phi_2, \phi_{1,sc}$ and $\phi_{1,sc}$ given in Fig. 4.4 either for addition or subtraction.



(a) addition (b) subtraction

Figure 4.4: Timing diagrams for addition and subtraction with SCA. With a clock of 4 MHz.

Starting from the circuit in Fig. 4.3 we will observe the evolution of charges between the phases knowing that charges are conserved between phases[1] to determine the output voltage. We recall that the amount of charges that a capacitor can hold is given by:[2]

$$Q = C\Delta V_C \tag{4.1}$$

We also make the following assumptions to simplify the development:

- We do not consider charge injection cause by the switches.
- The OTA is considered ideal: No offset voltage and infinite gain.

These assumptions are detailed in appendix A.

Addition

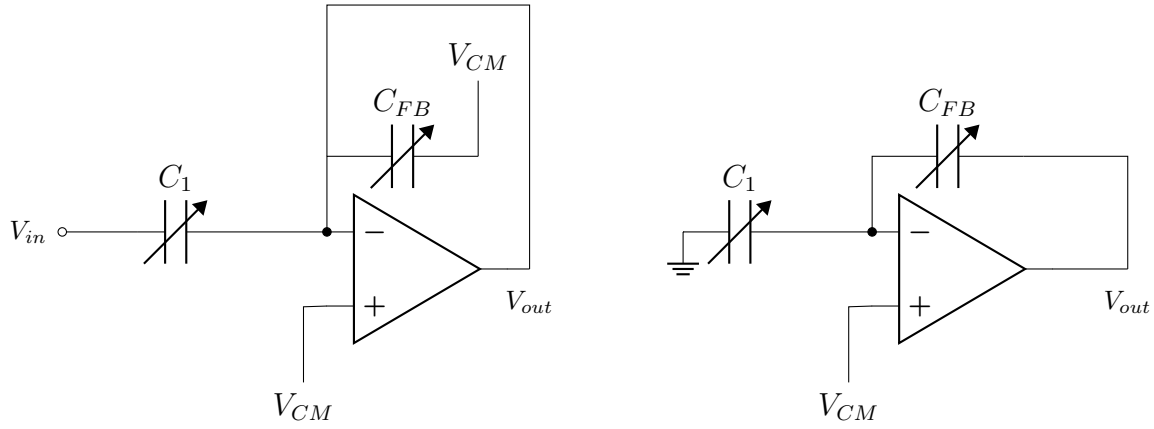
Before the first phase we enable the OTA. During the first phase, the reset phase, we charge the node $V_{AMP,IN-}$ to V_{CM} and we connect C_1 to V_{in}.
Then during the second phase, the sampling phase, V_{in} corresponds to the analog pixel, such that the charges at node $V_{AMP,IN-}$ in C_1 and C_{FB} become:

$$Q_1(t = T_2^-) = -C_1 \cdot (V_{in} - V_{CM}) \tag{4.2}$$
$$Q_{FB}(t = T_2^-) = 0 \tag{4.3}$$

The third phase consists in connecting the sampling capacitor C_1 to ground and transferring its charges to C_{FB} such that the charges contained in each capacitors become:

$$Q_1(t = T_3^-) = C_1 \cdot V_{CM} \tag{4.4}$$
$$Q_{FB}(t = T_3^-) = -C_{FB} \cdot (V_{out} - V_{CM}) \tag{4.5}$$

[1]Without considering the charges injected by the switches.
[2]We expect the phases to last sufficiently to charge completely the capacitors.



(a) Phase 1 and 2 (b) Phase 3

Figure 4.5: Three phases of operation for addition with SCA.

By conservation of charges at node $V_{AMP,IN-}$ between the two stages, since during the transition period all switches are open and no charges leak outside this node, we have to fulfil:

$$Q_{FB}(t = T_2^-) + Q_1(t = T_2^-) = Q_{FB}(t = T_3^-) + Q_1(t = T_3^-) \tag{4.6}$$
$$0 - C_1 \cdot (V_{in} - V_{CM}) = -C_{FB} \cdot (V_{out} - V_{CM}) + C_1 \cdot V_{CM} \tag{4.7}$$

Which leads to the final expression of V_{out}

$$\boxed{V_{out} = V_{CM} + \frac{C_1}{C_{FB}} V_{in}} \tag{4.8}$$

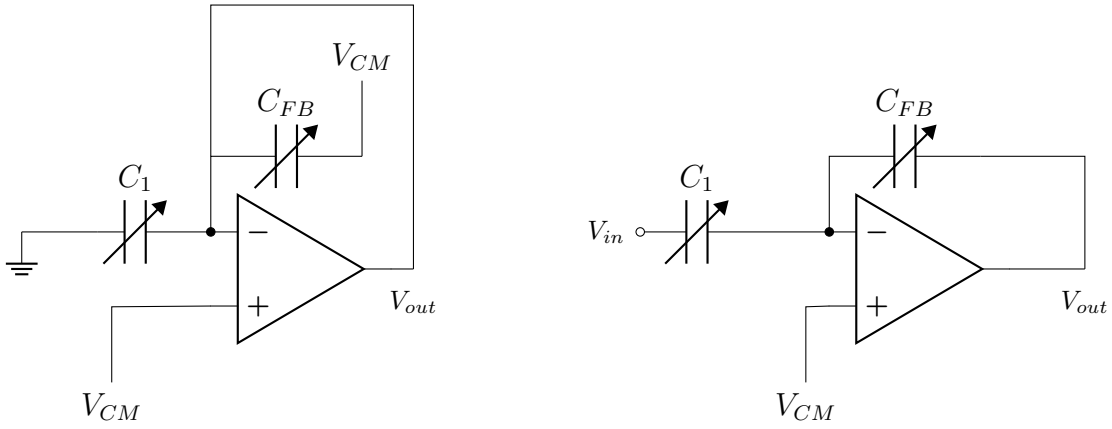
Subtraction

Subtraction is obtained by first setting C_1 to ground and C_{FB} to V_{CM}. This lead to the following charges being stored in the capacitors

$$Q_1(t = T_2) = C_1 \cdot V_{CM} \tag{4.9}$$
$$Q_{FB}(t = T_2) = C_{FB} \cdot \Delta V_{C_{FB}} = 0 \tag{4.10}$$

During the last phase, we load C_1 with V_{in} and connect C_{FB} to V_{out}. Doing so, we know that the charges stored in each capacitors at $t = T_3$ are

$$Q_1(t = T_3^-) = -C_1 \cdot (V_{in} - V_{CM}) \tag{4.11}$$
$$Q_{FB}(t = T_3^-) = -C_{FB} \cdot (V_{out} - V_{CM}) \tag{4.12}$$



(a) Phase 1 and 2 (b) Phase 3

Figure 4.6: Three phases of operation for subtraction with SCA.

As for the adder, the charges at node $V_{AMP,IN-}$ are conserved between phases 2 and 3 such that

$$0 + C_1 \cdot V_{CM} = -C_1 \cdot (V_{in} - V_{CM}) - C_{FB} \cdot (V_{out} - V_{CM}) \tag{4.13}$$

Which leads to the final expression of V_{out}

$$\boxed{V_{out} = V_{CM} - \frac{C_1}{C_{FB}} V_{in}} \tag{4.14}$$

Multiple inputs

We can connect multiple inputs with their capacitor to the negative input node of the OTA as in Fig. 4.2. For each input, we either perform an addition or subtraction by changing their respective control signals ϕ_1 and ϕ_2 with the multiplexers shown in Fig. 4.8a. The resulting output voltage is thus given by

$$\boxed{V_{out,j} = V_{CM} + \sum_{i=1}^{16} \sigma(W_{kji}[3]) \cdot \frac{C_{kji}}{C_{FB}} V_{in,ji} \quad \text{with} \quad \sigma(x) \begin{cases} 1 & \text{if } x \text{ is } 0 \\ -1 & \text{if } x \text{ is } 1 \end{cases}} \tag{4.15}$$

Equation 4.15 performs the MAC operation on a single row of the coefficient matrix k with a 16×16 pixels block, which correspond to a single element $y_{00,j}$ as presented in Fig. 4.10. In the next section we will see how we can accumulate all the rows by storing each partial result on a sampling capacitor.

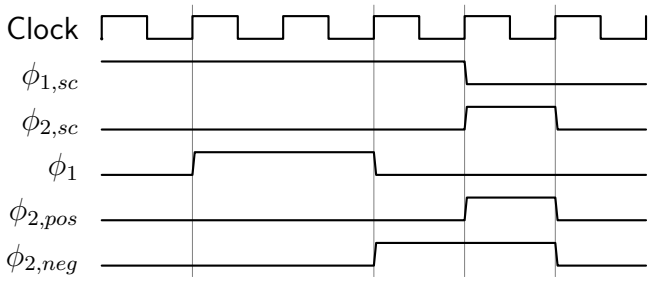
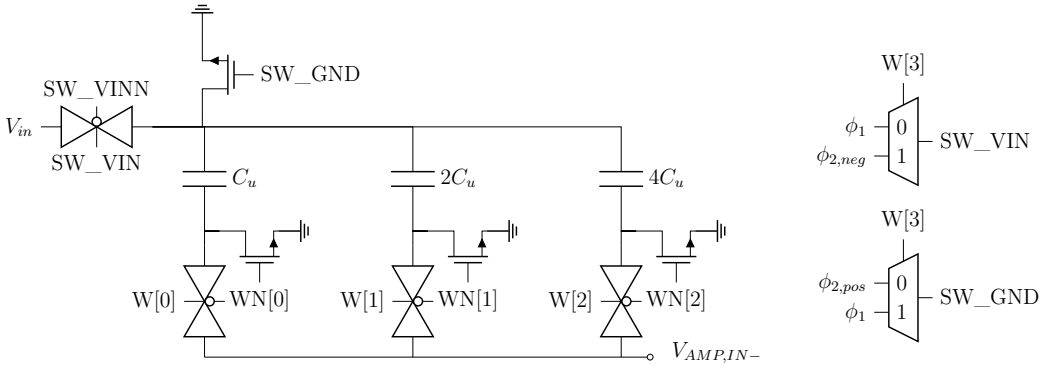
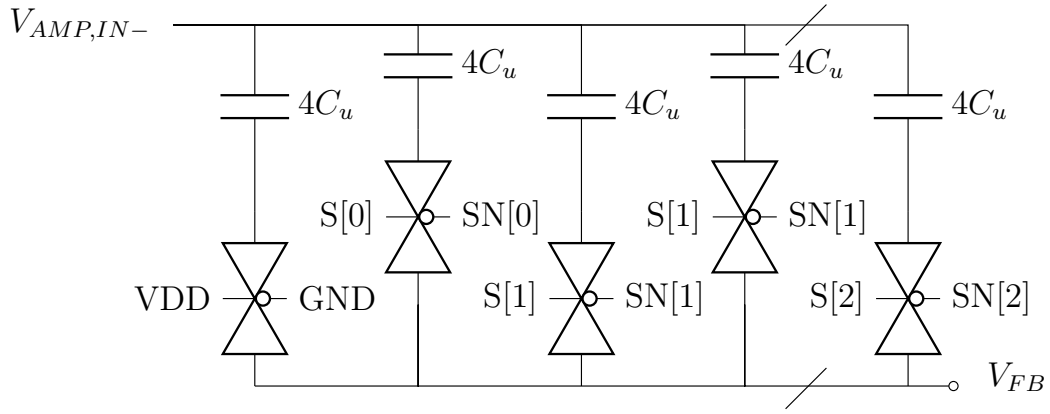


Figure 4.7: Timing diagram of signal used to control the SCA for addition and subtraction



(a) Coefficients capacitor banks with control signal multiplexers



(b) Feedback capacitor bank

Figure 4.8: Coefficient and feedback capacitors

4.1.2 Sample and hold

Since now we have only seen the computation of a single row, to compute the 16 rows we store intermediate row results on capacitors as show in Fig. 4.9. The image sensor controller controls the switches sequentially until the 16 rows are

stored on each capacitor C_u. Then the controller closes all the switches such that the voltage V_{out} become by charge conservation[3]

$$V_{out} = \frac{1}{16}\sum_{j=1}^{16} V_{out,j} = V_{CM} + \frac{1}{16}\sum_{j=1}^{16}\sum_{i=1}^{16} \sigma(W_{kji}[3]) \cdot \frac{C_{kji}}{C_{FB}} V_{in,ji} \tag{4.16}$$

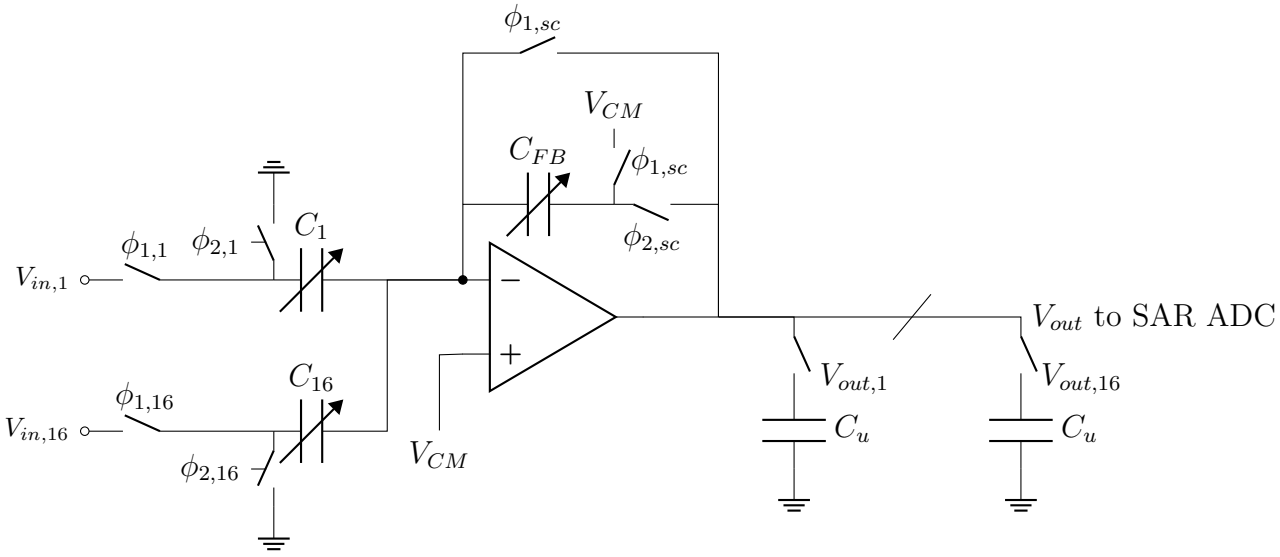


Figure 4.9: Switched capacitor amplifier with charge sharing

4.1.3 SAR ADC

The output voltage V_{out}, after charge sharing, is digitized with an 8-bit SAR ADC in the image sensor controller. The data is then further processed in the Cortex-M4 processor or transfered with DMA to the DCMI.

We do not consider any noise in the SAR ADC, such as comparator noise, thermal noise and internal DAC noise.

4.1.4 Analog core controller

The controller works in a row-wise fashion as shown in Fig. 4.10. Each coefficients matrix are stored in sixteen 64-bit word in the LMEM, since each weights are 4-bit

[3]Example for 2 capacitors:

1. Before: $Q_1 = C_u V_1$ and $Q_2 = C_u V_2$
2. After: $Q_f = (Q_1 + Q_2) = 2C_u V_{out}$ such that $V_{out} = \dfrac{V_1 + V_2}{2}$

long, the 64-bit word corresponds to the 16 coefficients of one row of the filter. We define the block column and row $\in [0, .., 7]$ as the index of a 16×16 blocks in the image sensor as shown in Fig. 4.11.
We can describe the controller operation with the pseudo code given in Algo. 1.

Algorithm 1 MAC operations controller

$y^{part} \leftarrow 0 \in \mathbb{R}^{8\times16}$
$y \leftarrow 0 \in \mathbb{R}^{8}$
for $i = 0$ to 7 **do** ▷ Each vertical blocks
  for $k = 0$ to n_{filts} **do** ▷ Each coefficient matrix
    for $l = 0$ to 15 **do** ▷ Each row of block
      $y_l^{part} += W_{k,l} \times V_{in,il}$ ▷ Compute each partial DCT
    end for
    $y = \sum_{l=0}^{15} y_l^{part}$ ▷ Charge sharing
    $TO\_REG(\mathrm{ADC}(y))$ ▷ Writes the eight 8 bits results to register
  end for
end for

1. The 8 MAC units work in parallel to perform the MAC operations row by row and stores each partial results on the sampling capacitors.

2. When the 16 rows are computed, the results are digitized in parallel by eight 8 bits SAR ADCs which write the digital values in a 64 bits register, each 8 bit corresponding to the results of a block of the image.

3. For the same block row, we compute each filters with their coefficients stored in the LMEM.

4. Once all filters are computed for the first block row we perform the same operation for the next block row and so on for the remaining 6 rows.

Remarks The feedback capacitor is also programmable but we can only program it at the beginning of the acquisition. Such that C_{FB} is constant for every coefficient matrix.

4.2 Analog core limitations and non idealities

Now that we have seen the general operation of the analog core, we will discuss some non idealities and hardware limitations specific to the MANTIS analog core architecture.

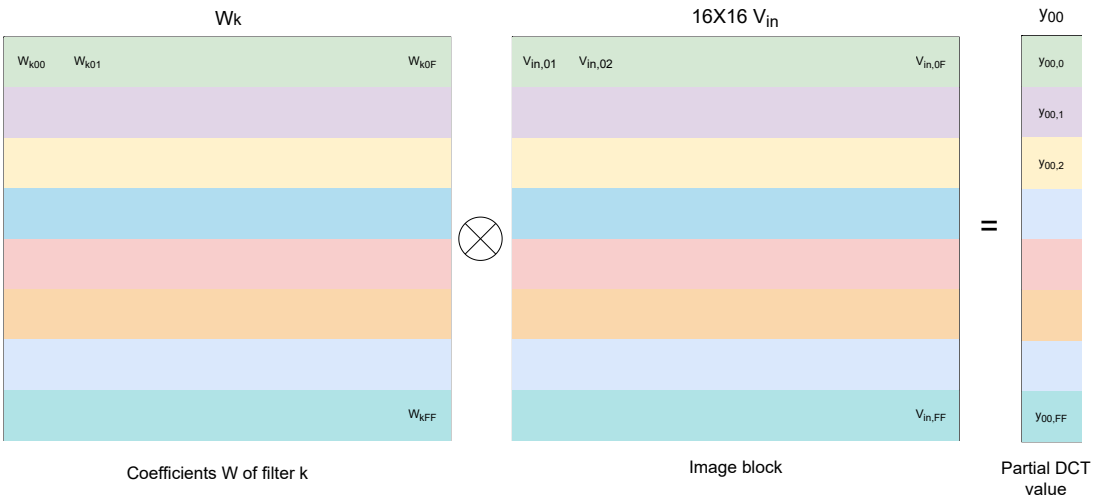


Figure 4.10: MAC operation of coefficients W_k with a 16 block. With the $\otimes$ operator corresponding to a row-wise MAC. Indices are given in hexadecimal representation.

4.2.1 Weights and saturation

We recall the output voltage equation of a single row MAC operation

$$V_{out} = V_{CM} + \sum_{i=1}^{16} \sigma(W_i[3]) \cdot \frac{C_i}{C_{FB}} V_{in,i} \tag{4.17}$$

With the implemented programmable capacitors shown in Fig. 4.8 we have the possibility to select the following values:

- $C_i \in [0, ..., 7] \cdot C_u$
- $C_{FB} \in [1, ..., 8] \cdot 4 \cdot C_u$

Thus the ratio $\frac{C_i}{C_{FB}}$ ranges in:

$$\frac{C_i}{C_{FB}} \in [0, 1/32, ..., 7/32, ..., 7] \tag{4.18}$$

We know that the dynamic range of V_{out} at the output of the SCA is limited by the power rails V_{EE} and V_{DD} of the OTA such that $V_{out} \in [0V, 1.2V]$. Thus, knowing $V_{CM} = V_{DD}/2 = 0.6V$, the SCA will not saturate if the following conditions are

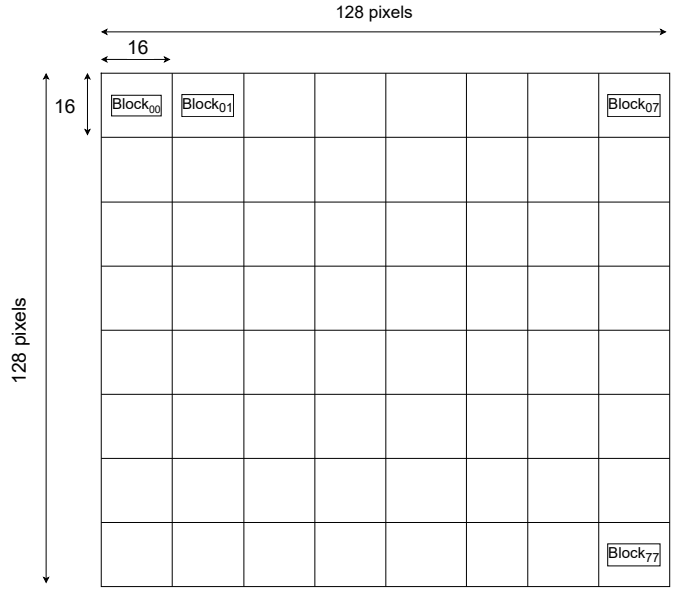


Figure 4.11: Block decomposition of the 128x128 image sensor

respected:

$$0 \leq V_{CM} + \sum_{i=1}^{16} \sigma(W_i[3]) \cdot \frac{C_i}{C_{FB}} V_{in,i} \leq 1.2 \tag{4.19}$$

$$-0.6 \leq \sum_{i=1}^{16} \sigma(W_i[3]) \cdot \frac{C_i}{C_{FB}} V_{in,i} \leq 0.6 \tag{4.20}$$

Knowing that $V_{in,i}$ are voltages comprised between 0 and 1.2V corresponding to pixels intensities and considering normal operation without over or under exposure we can expect that most acquired pixels will be at 0.6V, we can confirm this assumption by observing the cumulative histogram of the images data set shown in section 3.1, where we scaled the 8-bit amplitudes comprised between 0 and 255 to a linear voltage comprised between $0V$ and $1.2V$. The mean and median pixel intensity are respectively 0.60V and 0.63V.

Considering a constant filter by setting all the weights to the third smallest possible positive value of 3/32 we see that we will hit the upper boundary of the previous condition.

$$16 \cdot \frac{3}{32} \cdot 0.6 = 0.9 \nleq 0.6 \tag{4.21}$$

We will thus saturate the output and we will have an undesired behaviour of the MAC unit. This will be a limiting factor in order to obtain good results to compute the DCT.

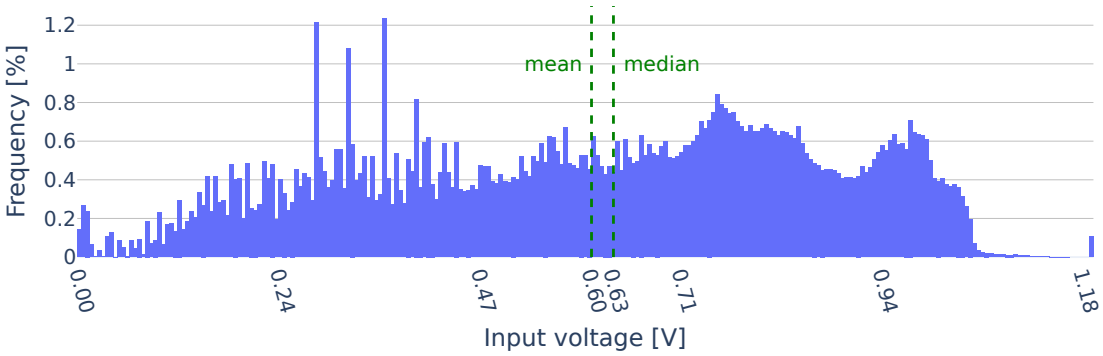


Figure 4.12: Cumulative histogram of the 6 images presented in section 3.1 with the 8-bit depths amplitudes scaled between $0V$ and $1.2V$.

4.2.2 SAR ADC consideration

Since now we were working with real values since we considered all the previous steps to be performed in the analog domain. But once the DCT is computed we need to go to the digital domain for further processing, this is done with an analog-to-digital converter (ADC).

The ADC has a finite resolution such that there will exist a difference between the analog signal and the quantized signal. This is called the quantization error as shown in Fig. 4.13. For an ideal ADC the maximum quantization error depends only on the resolution of the ADC (N) and the full scale range (FS) and is given as:

$$Q = \frac{\text{FS}}{2^{N+1}} \tag{4.22}$$

4.3 2D DCT with mantis

Let us first see how we compute the DCT with Mantis, we have seen that the output voltage at the output of the MAC unit is given by

$$V_{out} = V_{CM} + \frac{1}{16}\sum_{j=1}^{16}\sum_{i=1}^{16}\sigma(W_{kji}[3]) \cdot \frac{C_{kji}}{C_{FB}}V_{in,ji} \tag{4.23}$$

We can rearrange the previous equation to look like:

$$V_{out} - V_{CM} = \sum_{j=1}^{16}\sum_{i=1}^{16}\alpha_{kji}V_{in,ji} \tag{4.24}$$

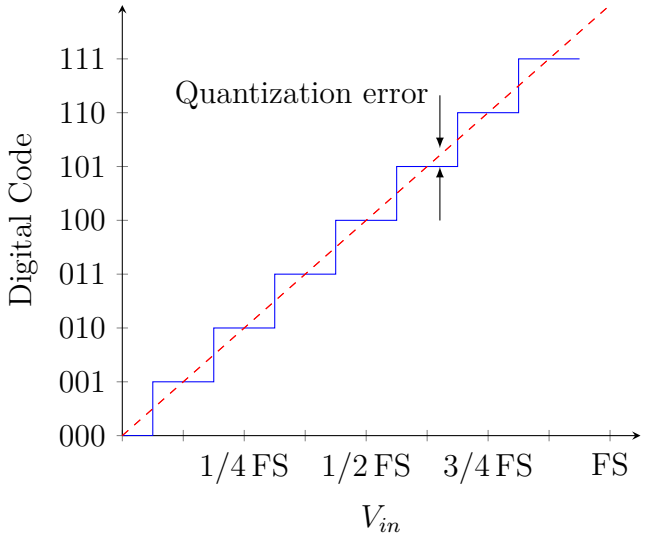


Figure 4.13: 3 bits ADC quantization error highlight

with $\alpha_{kji} = \sigma(W_{kji}[3]) \cdot \dfrac{C_{kji}}{16 \cdot C_{FB}}$

From the tensor form of the 2D DCT, $Y_{pq} = A_{pqji} X^{ji}$, we can rewrite tensor A as N^2 coefficient matrices $A^k \in \mathbb{R}^{\mathbb{N}\times\mathbb{N}}$ with k corresponding to a unique pair of indices (p, q) of tensor $A_{pq} = A^k$ with $p, q \in [0, .., N]$ and $k \in [0, ..N^2]$.

$$Y^k = \sum_{j=1}^{N} \sum_{i=1}^{N} A^k_{ji} \cdot x_{ji} \quad \text{with } A^k \in \mathbb{R}^{N\times N}, \tag{4.25}$$

We want to match the colored terms in equations 4.24 and 4.25, therefore we define $\hat{A}$ the 3-bit approximation of tensor A with quantization shown in Fig. 4.14

$$A^k_{ji} \approx \hat{A}^k_{ji} = \frac{1}{4} \cdot \overline{2 - 1} \tag{4.26}$$

and knowing that the programmable capacitors C_{kji} and C_{FB} written in term of the 3 LSBs of W and S

$$C_{kji} = \sum_{l=0}^{2} 2^l \cdot W_{kij}[l] \cdot C_u \tag{4.27}$$

$$C_{FB} = \left(\sum_{m=0}^{2} (2^m \cdot S[m]) + 1 \right) \cdot 4 \cdot C_u \tag{4.28}$$

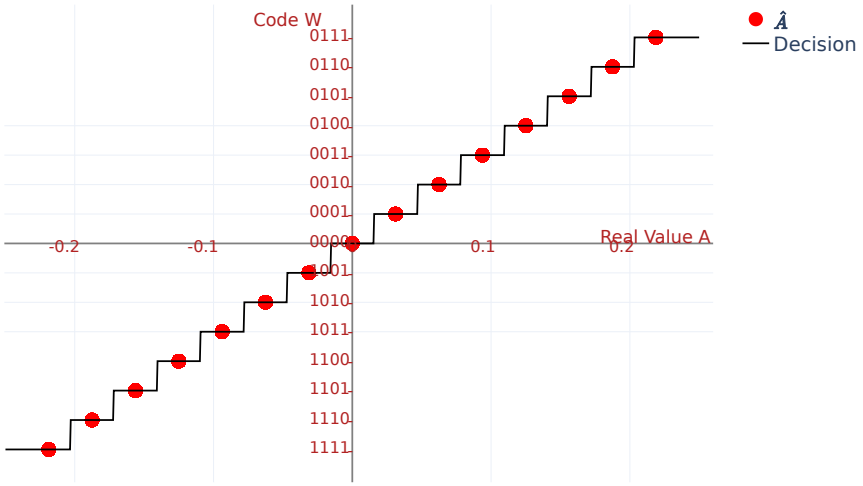


Figure 4.14: 3-bit approximation of the coefficients in A with the corresponding digital code

such that

$$\alpha_{kji} = \frac{1}{16} \cdot \sigma(W_{kji}[3]) \cdot \underbrace{\frac{\sum_{l=0}^{2} 2^l W_{kij}[l]}{4 \cdot \left(\sum_{m=0}^{2}(2^m \cdot S[m]) + 1\right)}}_{= \hat{A}^k_{ji}} \tag{4.29}$$

The orange term corresponds to the approximation $\hat{A}^k_{ji}$ if the weight of the feedback capacitor is set to its maximum value of 7, resulting to $C_{FB} = 32 \cdot C_u$. The division by 4 is required since the quantization scheme considers values ranging between 0 and $1/4 = 0.25$.
Selecting $S = 0b111$ and W_{kij} as the digital code of the approximated coefficient $\hat{A}^k_{ji}$ we will have

$$Y^k = 16 \cdot (V_{out} - V_{CM}) = \sum_{j=1}^{16} \sum_{i=1}^{16} \hat{A}^k_{ji} V_{in,ji} \tag{4.30}$$

In practice we construct the tensor $\hat{A}^k \in \mathbb{R}^{16\times16}$ as the up-sampling by two in each direction of the 2D DCT coefficient tensor for 8×8 blocks. This is equivalent to down sampling the 128×128 image by average pooling of size 2×2 to an 64×64 image. So it matches the developments done in the previous chapters.

4.3.1 ADC and saturation impact on image quality

Following the development made in chapter 3 we can simulate the impact on image quality of performing the 2D DCT with MANTIS by considering saturation of the SCA and quantization error.
Let first consider a more general case with two degrees of freedom:

- The ADC resolution;
- A programmable feedback capacitor.

The first allows to control the resolution of the digitized signal V_{out} and the programmable feedback capacitor allows to control row wise saturation as was previously discussed in section 4.2.1. Increasing the feedback capacitor will reduce the risk of saturation, such that in the analog domain we have

$$V_{out} = V_{CM} + \frac{1}{16}\sum_{j=1}^{16}\sum_{i=1}^{16}\sigma(W_{kji}[3]) \cdot \frac{\left(\sum_{l=0}^{2} 2^l W_{kij}[l]\right) C_u}{n_{FB} \cdot C_u} V_{in,ji} \tag{4.31}$$

with the feedback capacitor given by $C_{FB} = n_{FB} \cdot C_u$ with n_{FB} a positive natural number which is, for MANTIS, a multiple of 4 between 4 and 32. The saturation condition of a single row become

$$-0.6 \leq \sum_{i=1}^{16}\sigma(W_i[3]) \cdot \frac{\sum_{l=0}^{2} 2^l W_{kij}[l]}{n_{FB}} V_{in,i} \leq 0.6 \tag{4.32}$$

$$-0.6 n_{FB} \leq \sum_{i=1}^{16}\sigma(W_i[3]) \cdot \left(\sum_{l=0}^{2} 2^l W_{kij}[l]\right) \cdot V_{in,i} \leq 0.6 \cdot n_{FB} \tag{4.33}$$

Ideally we should not saturate for the worst case of pixels and coefficients, $V_{in,i} = 1.2V$ and $\sum_{l=0}^{2} 2^l W_{kij}[l] = 7 \quad \forall i$ and finally $W_a[3] = W_b[3] \quad \forall a \neq b$. Therefore we can find the ideal n_{FB} as

$$-0.6 n_{FB} \leq 134.4 \leq 0.6 \cdot n_{FB} \implies n_{FB} \geq 224 \tag{4.34}$$

Which corresponds to a feedback capacitor at least 7 times bigger than the one already included in the SoC. Let us simulate the impact of saturation on image quality, for this simulation we consider the 64 3-bit approximation of 2D DCT filters without additive noise. We can observe the results of the simulation in figure 4.15. We observe two distinctive trends, for $n_{FB} \leq 40$ the SAR resolution has no influence on image quality. Thus the saturation of the SCA dominates the degradation of image quality. For $n_{FB} > 40$ we see that for each SAR resolution

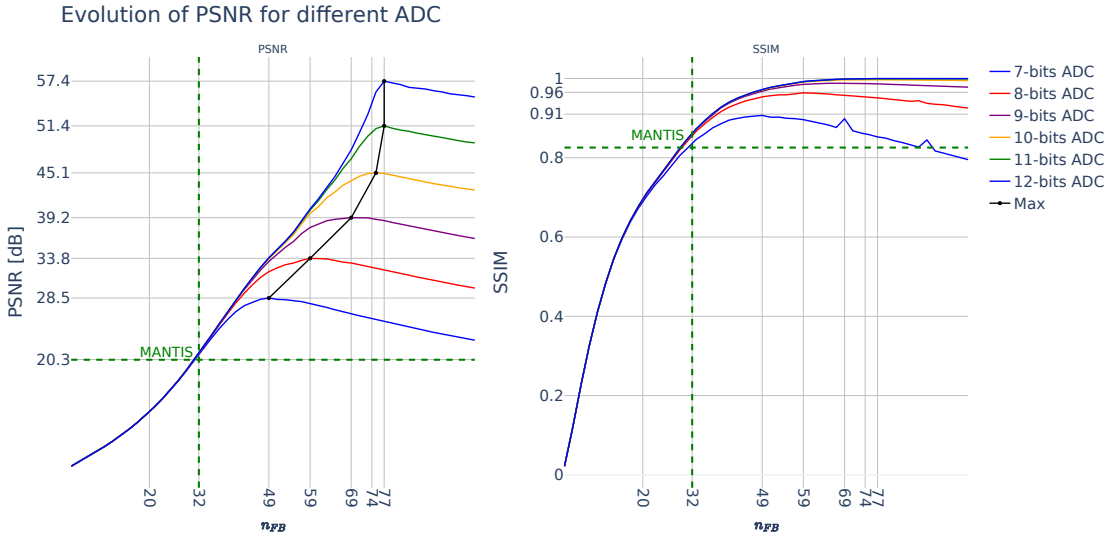


Figure 4.15: PSNR and SSIM evolution with respect of feedback capacitor and SAR ADC resolution. For 3+1 bit coefficients approximation and 64 filters.

we reach a peak then the image quality decreases. This means that the saturation of the SCA does not dominate the degradation of image quality, but the limiting factor is the SAR resolution. After the peak, the image quality decreases, this is explained by the fact that if n_{FB} is too large, the term $\frac{\sum_{l=0}^{2} 2^l W_{kij}[l]}{n_{FB}} V_{in,i}$ will not cover the full scale comprise between -0.6 and $0.6V$ such that increasing n_{FB} for a constant SAR ADC resolution is equivalent to loosing resolution of the SAR ADC since the input of the ADC does not cover the full scale range.

Unfortunately we see that with 3-bit coefficients approximation on MANTIS with an 8-bit SAR ADC and a maximum of $n_{FB} = 32$, even with 64 filters which is greater than the actual maximum amount of 32 filters that MANTIS can handle, the performances are quite low at around 20.3dB and we can conclude that the limitation is caused by the saturation of the SCA and not the SAR ADC resolution. But as we have seen in chapter 3, we can decrease the resolution to 2 bits, which in this context is similar to multiplying n_{FB} by two. We obtain decent results with 64 filters as shown in Fig. 4.16. With an 8-bit ADC and $n_{FB} = 32$ we have a PSNR of 33.06dB and 96% SSIM. We can finally observe in Fig. 4.17 the impact of hard thresholding with an 8-bit SAR ADC and $n_{FB} = 32$. With MANTIS, we can at most use 32 filter, which correspond to hard thresholding of the 32 highest frequencies filters, we have a PSNR of 27.74dB and 91% SSIM.

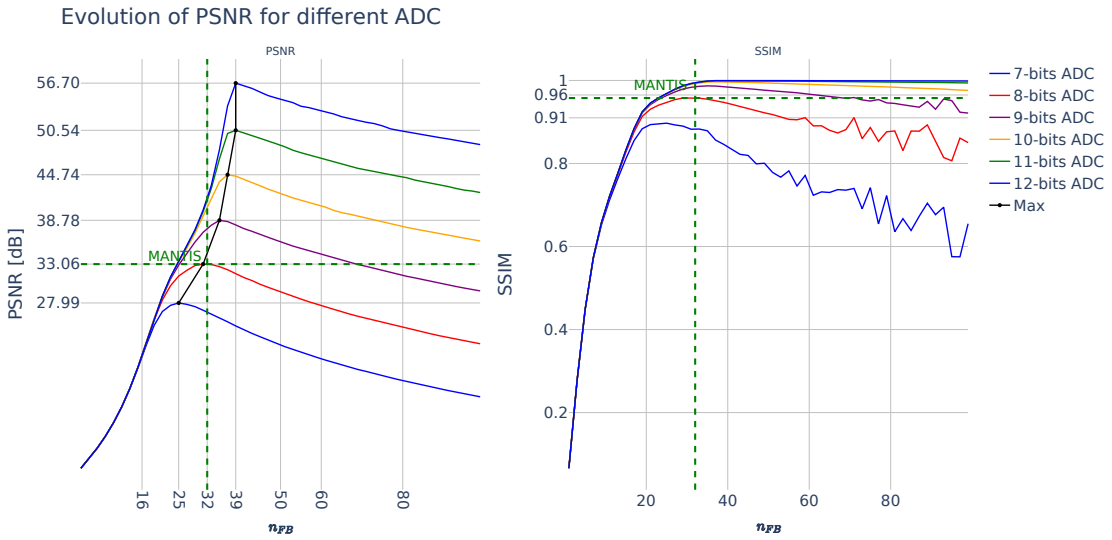


Figure 4.16: PSNR and SSIM evolution with respect of feedback capacitor and SAR ADC resolution. For 2+1 bit coefficients approximation and 64 filters.

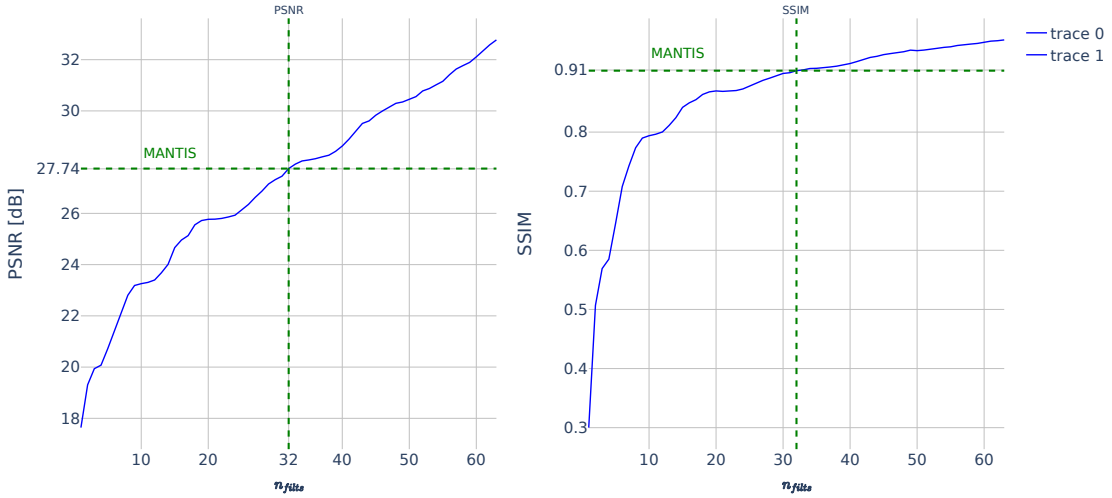


Figure 4.17: PSNR and SSIM evolution with respect of the amount of filters for 2+1 bit coefficients approximation and 8-bit SAR ADC.

4.3.2 Results summary

To summarize we have three sources of degradation on the reconstructed image,

- Hard thresholding [4]
- SCAs saturation
- 8-bit SAR ADC

As stated previously, even though MANTIS handle 16×16 pixels block, we perform the 2D DCT on 8×8 block by up sampling the coefficients which is similar to downsampling the original 128×128 pixels image to a 64×64 pixels image as shown in Fig. 4.18.

Figure 4.18: Downsampled image

In Fig. 4.19 we can observe qualitatively the cumulative degradation caused by each step. The hard thresholding already sets the PSNR to 30.67dB. Then there is little degradation caused by the SCAs saturation, the PSNR decreases only by 0.33dB. Finally the 8-bit SAR ADC decreases the PSNR by 2.59dB to 27.75dB. From these results we can conclude that we need at least 32 filters, we can tolerate some saturation to an extent but the resolution of the ADC becomes critical if we want to maintain sufficient precision while preventing saturation.

4.4 Digital implementation

MANTIS integrates a Cortex-M4 processor which allows further processing of the results of the analog MAC units. We utilized this processor to perform the two

[4] In appendix B we performed the simulations and impact of hard thresholding for 64×64 image pixels. Results are a bit lower than for 128×128 images discussed in chapter 3.

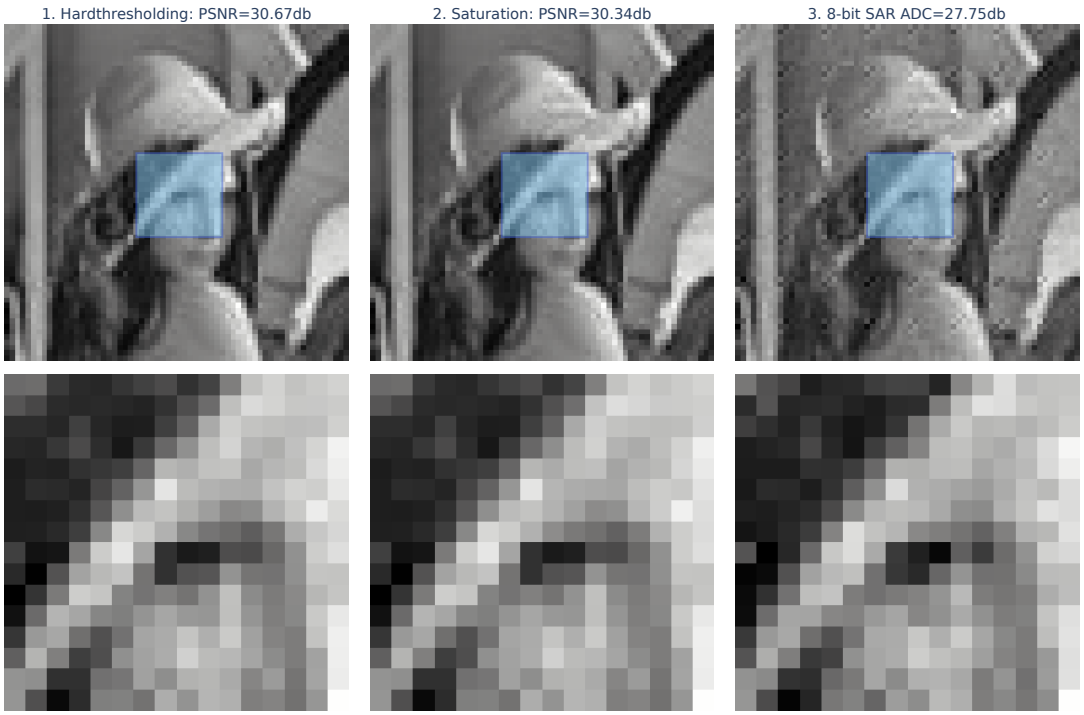


Figure 4.19: Impact of non-idealities

last steps of the JPEG algorithm; quantization and Huffmann encoding.

Why quantizing before encoding? The goal of quantization is to reduce the amount of bits necessary to reprensent a value. For example, if we quantize the 8-bit integer *255* by 16. Originally we would need

$$255 = 0\text{b}11111111 \tag{4.35}$$

$$\text{Quantized} \quad \left\lfloor \frac{255}{16} \right\rceil = 16 = 0\text{b}00011111 \tag{4.36}$$

While 255 needed 8 bits, the quantized value,16, could be represented only by its 5 MSBs. Without quantization, we will have a lot of integer covering the whole 8-bit range, such that entropy coding those values will not have the expected behaviour. We will end up with encoded data larger than the raw data! For example, if we had not quantized the value *255* which is at least represented with 8 bits.

The way the data is entropy encoded in the JPEG algorithm would result in a bitstring looking like:

$$255 \rightarrow 0\text{b}\ 11110\ 1111\ 1111 \quad \text{encoded data length } = 4 + 8 = 12\text{bits} \tag{4.37}$$

255 in binary representation

Huffman code corresponding to 8 bits length

Which is significantly greater than the original 8 bits. Without quantizing, such cases are more frequent.

From the previous remark we see that quantization is sometimes necessary if we want to compress the raw data depending on the condition of operation. In our case, we can observe the histogram of the 2D-DCT performed with mantis in Fig. 4.20, where we can see that most values are centered around 0, thus we can expect that huffman encoding on the raw data will be able to compress data.

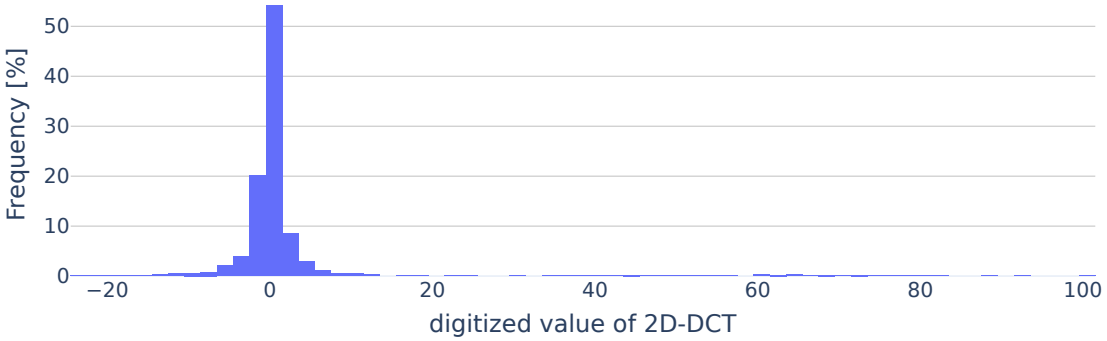


Figure 4.20: Histogram of raw centered 8-bit 2D DCT values.

4.4.1 Description of the digital implementation

In order to encode the raw data coming from the analog core we have to store the raw data in the DMEM which is available to the Cortex-M4 processor. Therefore we configure the DMA to transfer one of the two 64-bit register containing the eight 8-bit raw DCT values corresponding to one filter applied to the eight blocks of a row to a buffer, `img_fragment`, allocated in the DMEM . The DMA works in a Ping-Pong fashion, filling `img_fragment` 64-bit by 64-bit with the content of one 64-bit register then the other. It is triggered by an interrupt request (IRQ) generated by the image sensor controller when one of the 64-bit register is filled. For

timing constrain reasons, we first completly stores the full 2D-DCT in the buffer before quantization and encoding, such that the size of `img_fragment` corresponds to $n_{\text{filts}} \cdot 8 \cdot 8 = 1984$ bytes for $n_{\text{filts}} = 31$[5].

Once the buffer is filled, we perform quantization and entropy coding as described in the JPEG ISO. The encoded data is then sent trough the DCMI. The pseudo code is given in Algo. 2, where we can observe that the code is split into three main functions wich respectively:

- `read_block`: reads the `img_fragment` and reconstruct the actual block.
- `quantize`: quantize the values of the DCT. [6]
- `encode_block`: Entropy coded the block and sends it to the DCMI.

Algorithm 2 Embedded pseudo code

```
img_frag ← malloc(n_filts · 8 · 8)                          ▷ Image buffer

function ENCODE( )
    for row = 0 to 8 do                                      ▷ Each row
        for col = 0 to 8 do                                  ▷ Each column
            block ← READ_BLOCK(img_frag, row, col)           ▷ Read
            block ← QUANTIZE(block)                          ▷ Quantize
            ENCODE_BLOCK(block)                              ▷ Huffman
        end for
    end for
end function

function DMA_IRQ( )                                          ▷ Fill buffer by DMA
    img_frag[i] ← IMAGE_SENSOR_DOUT                          ▷ Each pixels
    if i ← i + 1 == n_filts · 8 · 8 then                     ▷ Filled
        ENCODE( )
    end if
end function
```

The complete C code is given for reference in appendix C included with a custom Python Huffman decoder.

[5] Due to an hardware bug we can not have 32 filters.

[6] in practice this is performed during `read_block` for optimization

Memory constrains The embedded program and data memory are only 8kB, we had to be careful while implementing the firmware to respect these memory constrains. Some basic rules were followed to reduce the usage of PMEM:

- Constants, such as Huffman codes, were encoded with the smallest data type possible.
- All unnecessary functions, such as `printf`, were carefully commented.
- We used the optimization flag `-Os` in which tries to generate the smallest compiled code.

With these optimizations, we could obtain a compiled code of 5252 bytes which fits the 8kB PMEM. Similarly we had to be careful while allocating dynamic memory in order to not overflow the DMEM.

4.4.2 MANTIS power consumption estimation

We can distinguish three modes of operation:

- RAW: the raw data coming from the analog core is sent through the DCMI without processing.
- Huffman without quantization: we encode the raw data without quantization.
- Huffman with quantization: We encode the quantized data.

The whole digital part of the SoC is designed with the Cadence software's stack, since we were not able to have access on time to the physical chip, we have performed power consumption analysis with the Cadence simulation stack. Since the analog core is not simulated physically, only the behaviour is simulated, we have to make some assumptions from results obtained with analog design tools such as Eldo.

MAC unit

A single MAC unit was simulated with Eldo for a single block of an 16×16 random image with voltages uniformly distributed between $0V$ and $1.2V$ and the coefficients of the 2D DCT 2 bit approximation. Running at 4MHz with typical process corners and a temperature of 25 °C we obtained an average power consumption of $p_{\text{avg,mac}} = 9.43\mu W$. The MAC unit needs 6 clock cycle to operate correctly, we can then estimate the energy consumption of a single MAC operation as

$$E_{\text{MAC}} = \frac{N_{\text{cycl,MAC}}}{f_{\text{CLK}}} \cdot p_{\text{avg,mac}} = 14.15pJ \tag{4.38}$$

wich corresponds to multiplication and addition of 16 pixels by 16 coefficients. A single pixel will thus on average require $E_{\text{PIX}} = 1/16 \cdot E_{\text{MAC}} = 0.88pJ/pixel$.

Analog core

In practice, we estimate the analog core and the required bias and reference current sources generation with the values given in table 4.1 [7]:

Circuit	average power $[\mu W]$
Image sensor and analog core	197.77
MAC units	2.33
Reference current sources	0.05
Bias voltage generation	19.7

Table 4.1: Average power consumption of analog core circuits.

Digital parts

To estimate the power consumption of the digital parts of the SoC, we used activity annotation on post route physical simulations. Then to extract the power reports we used Innovus `report_power` tool, which gives the internal and switching power consumption.

RAW For the raw mode, there is no processing on the pixels done by the Cortex-M4, the data is simply sent to the DCMI. We obtained, from the simulation, the results given in table 4.2. It requires $266.64pJ$/pixel and 6.2ms to compute the first 31 filters of the DCT and transmit the $31 \cdot 8 \cdot 8 = 1984$ bytes to the DCMI. Even though there is no entropy coding, we can consider this mode as a lossy compression scheme such that we achieved to store an image of $64 \times 64 = 4096$ pixels on 1984 bytes. Resulting in a bits per pixel (bpp) measure of 3.3875bpp and a PSNR of 27.75 dB.

Huffman encoding without quantization For this mode of operation, we have to distinguish two phases of operation: the first phase, the acquisition, fills the buffer `img_buffer` while the second phase, the encoding, reads the buffer and encodes it in order to send it to the DCMI. For the acquisition stage, the image sensor and the analog core are powered on, this phase lasts 6.2 ms. During the encoding phase, only the Cortex-M4 needs to stay on, the image sensor and controller are powered off, such that we can consider that they do not draw

[7] Values obtained thanks to the designer of Mantis, Martin Lefebvre.

Power	Power [μW]	Energy [nJ]	Energy per pixel [pJ]
Internal	366.12	2257.13	137.76
Switching	122.74	756.69	46.18
Image sensor and analog core	200	1233.00	75.26
Image sensor	197.77	1235.58	74.80
MAC units	2.33	14.42	0.88
Reference current sources	0.05	0.31	0.02
Bias voltage generation	19.7	121.45	7.41
Total	708.61	4368.58	266.64

Table 4.2: Average power consumption for raw pixels sending with 31 filters with a duration of 6.2ms .

current during this phase. The encoding lasts in average 54.77ms. The power simulation results are gathered in table 4.3. Such that the total energy consumption is 1497.06pJ/pixel for a processing time per frame of 61.57ms. It achieved a compression rate of 1.705 bpp while maintaining the image quality to a PSNR of 27.75dB.

	Acquisition phase			Encoding phase		
Power	Power [μW]	Energy [nJ]	Energy per pixel [pJ]	Power [μW]	Energy [nJ]	Energy per pixel [pJ]
Internal	269.84	1673.01	102.11	273.47	16681.67	1018.17
Switching	67.14	416.27	25.41	72.04	4394.38	268.21
Image sensor	197.77	1235.58	74.80	0.00	0.00	0.00
MAC units	2.33	14.42	0.88	0.00	0.00	0.00
Reference current sources	0.05	0.31	0.02	0.00	0.00	0.00
Bias voltage generation	19.7	122.14	7.45	0.00	0.00	0.00
Total	556.73	3451.73	**210.68**	345.51	21076.05	**1286.38**

Table 4.3: Average power consumption and energy consumption for 2D DCT with Huffman encoding.

During the encoding phase, we can distinguish three main steps, as was previously discussed in algo. 2. For this mode of operation, we only have the two steps `read_block` and `encode_block` that are called for each block which for each block lasts 87μs and, on average, 768.80μs[8]. Such that the processing time of the encoding phase is respectively 10.16% and 89.83% for these two functions. Knowing that we can evaluate the power consumption repartition of each step, in the pie chart in Fig. 4.21 and table 4.4. To estimate the power consumption of the 2D DCT, we consider the total energy consumed during the acquisition phase and we subtract the energy consumed by the image sensor. To estimate the duration of the 2D DCT we consider that it takes $6 + 8 = 14$ cycles to compute the 2D DCT with the SCA and digitize its result with the 8-bit ADC.

[8]These values were obtained by measuring the average duration of the function calls.

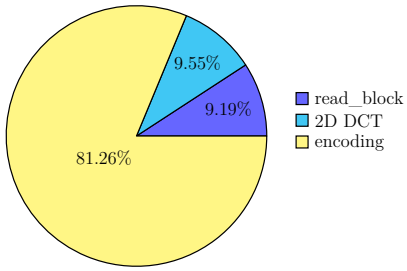


Figure 4.21: Power repartition, no quantization.

Step	Energy per pixel [pJ]	Duration per pixel [ns]
2D DCT	135.88[2]	218.75[1]
Read block	130.7	339.64
Encoding	1165.68	3003.26

Table 4.4: Summary of energy per pixel and duration for each step.

Huffman encoding with quantization First we can observe the impact of quantization on PSNR, SSIM and bpp in Fig. 4.22. Therefore we define a quality factor (QF) ranging from 1 to 100 which apply a scaling factor on the quantization matrix given in the JPEG ISO. Such that a QF of 1 correspond to a matrix full of 255 and an QF of 100 correspond to a matrix full of 1, the first quantizing all elements of the 2D DCT to 0, the second keep the elements as is. We see that

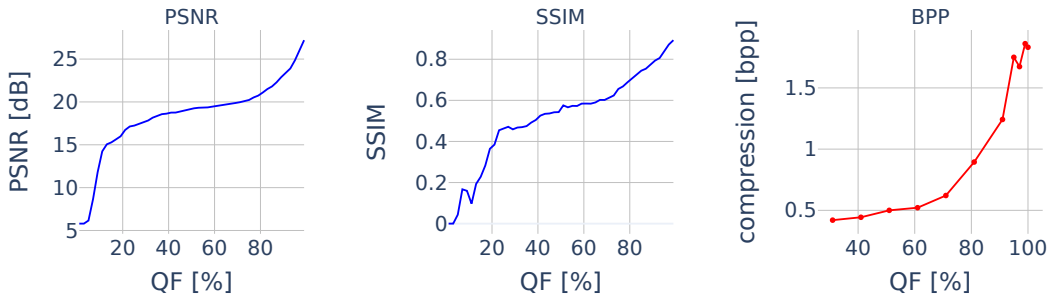


Figure 4.22: Impact of quantization on PSNR, SSIM and bpp

even tough we gain in compression we lose quickly in image quality, the choice of the necessity of a quantization and the value of a quality factor will depend on the required tolerance on image quality. In our case, introducing a quantization will decreases the too much the quality of the image. We will still study the impact of quantization done in the Cortex-m4.

Starting with the results of previous paragraph, the quantization function lasts $536.58\mu s$ per block resulting in an added processing time of $2097.1ns$/pixel But,

[1] 6 cycles SCA and 8 cycles ADC for 16 pixels at 4MHz

[2] We consider the power consumption of the acquisition phase without taking the image sensor

since there are less bits to encode per block, the encoding function is quicker. For example with a QF of 80%, the average encoding step per pixel is only $699.52ns$/pixel.

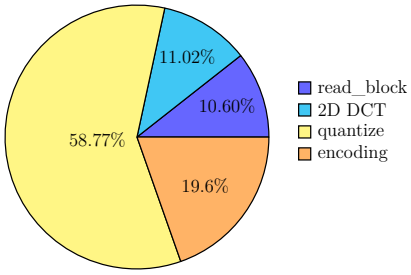


Figure 4.23: Power repartition, with quantization.

Step	Energy per pixel [pJ]	Duration per pixel [ns]
2D DCT	135.88	218.75
Read block	130.7	339.64
Quantization	724.57	2097.1
Encoding	241.69	699.52

Table 4.5: Summary of energy per pixel and duration for each step QF=80%

We can see in Fig. 4.23, that the quantization steps dominates. But since we reduce the amount of bits transferred, the overall energy consumption decreases compared to the encode mode without quantization. With also the consequence of loosing image quality.

Results summary We summarized the results in table 4.6.

	bpp	PSNR	Duration [ms]	Energy per pixel [pJ]	compressed image /s
RAW	3.39	27.75	6.2	266.64	161.29
No quantization	1.71	27.75	61.57	1497.06	16.24
With quantization	0.9	22.5	54.67	1232.84	18.29

Table 4.6: Results summary for the 3 different modes of operations.

We can see that we can operate MANTIS with different embedded firmware in order to tune the tradeoffs between energy consumption, compression ratio, PSNR and image per second. Current state of the art [9] ultra-low power transmitters achieve as low as 7 pJ/bit for -33 dBm of radiated power. With the encoding scheme implemented in the Cortex-M4, comparing the RAW mode with the mode with the mode with quantization we can reduce the amount of bits by a factor 3.384 while consuming 966.2 pJ/pixel more. For the case of a 64×64 image, we save 10 199 bits for the additional cost of $3.957\mu J$, compared with the $71.4nJ$ that would have been necessary to send the 10 199 bits with an ultra-low power transmitter. But with the raw mode, achieving a 3.39 bpp requiring $1.092\mu J$ to process a 64×64 image then $0.097\mu J$ to send the compressed image, compressing the image instead of sending a full 8 bit depth 64×64 image that would have required $2.294\mu J$ thus we divide by two the energy required to send the uncompressed image.

Chapter 5

Possible improvements

In the previous chapters we have seen a few limitation with the MANTIS SoC that have contributed to degraded performance in the context of 2D DCT. MANTIS was not originally built to compute the 2D DCT, thus specific requirements in order to do so were not met by design. The main limitations are coming from

- The amount of coefficient matrices limited to only 32 filters, and thus 32 2D DCT coefficients.
- Saturation in the SCAs.
- 8-bit SAR ADC.

5.1 Hardware improvements

As we have seen in chapter 3 and 4, using a 9-bit ADC with a feedback capacitor 25% bigger than the current one in the SCA and doubling the LMEM from 4kB to 8kB to be able to store the 64 coefficients matrix of the 2D DCT would, according to our simulation, increase the PSNR to at least 38.78 dB. Doing so, would also increase the effective area of the chip and also increase the power consumption, first because doubling the amount of filters, doubles the times the SCAs need to operate, increasing the resolution of the ADC also increases its power consumption since it requires a higher resolution DAC thus more capacitors to charge.

We could also consider a more appropriate hardware architecture, in MANTIS, the SCAs have 16 input and we have seen that saturation occurs, leading to degradation of the resulting image after decompression. In fact, the saturation depends on the coefficients matrix if we have a lot of big coefficient in the same row, the SCA will saturate. Increasing the amount of input of the SCAs would benefit

from the symmetry of most 2D DCT filters, a matrix with a big coefficient row has often an opposite row that will prevent the SCAs to saturate. Increasing the amount of input will furthermore increase the parallelization of MAC operations, but will have a huge cost in area since each input would need its programmable capacitor and the logic to control them. For example, increasing to 64 inputs each SCA would require 336 more unit capacitors and 576 NMOS and PMOS. But it will also decrease the amount of sampling capacitor required by 4, gaining 12 unit capacitors. But from the development we have done, we have seen that we could store the coefficients of the 2D DCT on only 2 bits, doing so would remove 4 C_u, 2 NMOS and 1 PMOS per programmable capacitor. For our hypothetical 64 input SCA with 2-bit programmable capacitors the balance would require 160 NMOS, 224 PMOS and 68 C_u more per SCA. Increasing the amount of input of the SCA does seem a costly solution in area without guaranteeing the all filters will not cause saturation. Another solution to prevent saturation would be to increase the feedback capacitor, using 2-bit programmable capacitors we save $64C_u$, 32 NMOS and 16 PMOS. Such that the saved space would allow to add 2 bits of control, 2 transmission gates and 8 C_u , allowing the feedback capacitor to reach $40C_u$ which is sufficient to prevent saturation on almost all filter, as we have see in chapter 4. The remaining space can be used to increase the resolution of the ADC, resulting in better performance. Another solution to prevent saturation would be to divide problematic filters in sub filters. For example, a filter with constant row values could be split in two, resulting in a filter with the same constants values for the first eight columns then we set the remaining 8 columns to zero. Then, we use the complementary filter with the eight first column set to zero and the last ones to their original value. This could prevent saturation at the cost of using more filters and thus reducing the amount of different filters available or increasing the processing time. It also adds some overhead at the post processing step.

5.2 Software improvements

Apart hardware considerations, some software improvements could be done in order to achieve better performance. In chapter 4, we have seen that the quantization step performed in the Cortex-M4 dominates the power consumption but allowed better compression. We could investigate ways to perform the quantization step in the analog domain by dividing the coefficient matrices by the value of the quantization matrix corresponding to this frequency. This would require better control of the feedback capacitor between the computation of each filters, currently MANTIS can not change the feedback capacitor during the acquisition. Furthermore, more investigation could be conducted in order to optimize the firmware in the Cortex-M4. Currently, before performing the entropy coding, we rearrange the image

fragments in blocks similarly to the actual JPEG description given in the corresponding ISO standard. We could investigate a more optimized way of entropy coding in order to utilize the current arrangement of pixels instead of changing it. Since MANTIS computes the values of each filter per row for the eight columns, a differential entropy coding, instead of amplitude, seems the most appropriate since we can benefit from the fact that adjacent block will have similar 2D DCT values for the same frequencies.

Chapter 6

Conclusion

During this work we have evaluated the possibility of performing the first step of the JPEG compression scheme, the 2D DCT, in the analog domain. Novel approximations adapted to the constrains of analog processing were given with estimation of the impact on image quality. A working implementation was proposed on the MANTIS CMOS imager SoC achieving a compression ratio of 3.39 bpp while maintaining an average PSNR of 27dB and consuming only 266 pJ/pixel which correspond to $FOM_2 = 12.51$ pJ/pixel as defined in chapter 2. Compared with results found in the literature, gathered in chapter 2, it may look like we performed worse than previous implementation but we have to recall that most results do not take in consideration the control logic required to perform the DCT. In Chapter 4, we dedicated our discussion to the power consumption reduction achieved by implementing near-sensor compression, as opposed to the transmission of raw data with an ultra-low power transmitter. We concluded that near-sensor compression resulted in a energy reduction of up to two times when compared to the energy consumption associated with raw data transmission.

The current interest in mixed-signal near-sensor processing for machine learning applications is undeniable. The development of the MANTIS SoC initially focused on its application as a convolution core in this context. However, leveraging the capabilities of MANTIS for image compression introduces a new functionality on the same hardware platform. This expansion of MANTIS's capabilities offers the potential for discovering novel applications beyond its original purpose. For example, in the fields of biomedical research or environmental monitoring, where power constraints are often a critical consideration, an SoC similar to MANTIS could, in a first phase, be used for event detection then, using the same hardware, enter a recording phase with compression with no added cost.

Furthermore we have discussed throughout chapter 3,4 and 5 possible hardware

and software improvements that could greatly increase the performance of the system but will need further research to evaluate them properly.

Bibliography

[1]

[2] Amba ahb protocol specification.

[3] Documentation x2013; Arm Developer — developer.arm.com. https://developer.arm.com/documentation/ddi0439/b/BEHJADED. [Accessed 26-May-2023].

[4] iso/iec 10918-1 jpg. https://www.iso.org/standard/18902.html, Feb 1994.

[5] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[6] S. Allwin Devaraj, Stalin Jacob, Jenifer Darling Rosita, and M. M. Vijay. Vlsi implementation of color demosaicing algorithm for real-time image applications. In Anandakumar Haldorai, Arulmurugan Ramu, Sudha Mohanram, and Mu-Yen Chen, editors, *2nd EAI International Conference on Big Data Innovation for Sustainable Cognitive Computing*, pages 317–329, Cham, 2021. Springer International Publishing.

[7] Alpha Yaya Balde, Emmanuel Bergeret, Denis Cajal, and Jean-Pierre Toumazet. Low power environmental image sensors for remote photogrammetry. *Sensors*, 22(19), 2022.

[8] Bryce E. Bayer. Color imaging array. https://patents.google.com/patent/US3971065A/en. US3971065A.

[9] Hansraj Bhamra, Yu-Wen Huang, Quan Yuan, and Pedro Irazoqui. An ultra-low power 2.4 ghz transmitter for energy harvested wireless sensor nodes and biomedical devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(1):206–210, 2021.

[10] CBurnet. A bayer pattern on a sensor in isometric perspective/projection. `https://commons.wikimedia.org/wiki/File:Bayer_pattern_on_sensor.svg`, year=2006.

[11] Colin Doutre, Panos Nasiopoulos, and Konstantinos N. Plataniotis. A fast demosaicking method directly producing ycbcr 4:2:0 output. *IEEE Transactions on Consumer Electronics*, 53(2):499–505, 2007.

[12] J. L. E. Dreyer. On Tycho Brahe's manual of trigonometry. *The Observatory*, 39:127–131, March 1916.

[13] Ferda Ernawan and Siti Nugraini. The optimal quantization matrices for jpeg image compression from psychovisual threshold. *Journal of Theoretical and Applied Information Technology*, 70:566–572, 12 2014.

[14] Anirban Ganguly and Ayan Banerjee. Precise realization of one-staged 2-d dct using analog current mode architecture in compressed sensing front-end. *Microelectronics Journal*, 115:105184, 2021.

[15] Tarek I. Haweel. A new square wave transform based on the dct. *Signal Processing*, 81(11):2309–2319, 2001.

[16] Sungjin Hong, Heechai Kang, Jusung Kim, and Kunhee Cho. Low voltage time-based matrix multiplier-and-accumulator for neural computing system. *Electronics*, 9(12), 2020.

[17] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[18] Bhuvanan Kaliannan and Vijaya Sankara Rao Pasupureddi. Implementation of a charge redistribution based 2-d dct architecture for wireless capsule endoscopy. pages 153–157, 2013.

[19] S. Kawahito, M. Yoshida, M. Sasaki, K. Umehara, D. Miyazaki, Y. Tadokoro, K. Murata, S. Doushou, and A. Matsuzawa. A cmos image sensor with analog two-dimensional dct-based compression circuits for one-chip cameras. *IEEE Journal of Solid-State Circuits*, 32(12):2030–2041, 1997.

[20] Edward H. Lee and S. Simon Wong. Analysis and design of a passive switched-capacitor matrix multiplier for approximate computing. *IEEE Journal of Solid-State Circuits*, 52(1):261–271, 2017.

[21] Meng-Chun Lin, Lan-Rong Dung, and Ping-Kuo Weng. An ultra-low-power image compressor for capsule endoscope. *Biomedical engineering online*, 5:14, 02 2006.

[22] Olivier Losson, Ludovic Macaire, and Yanqin Yang. Comparison of color demosaicing methods. *Advances in Imaging and Electron Physics*, 162:173–265, July 2010.

[23] Junjie Lu, Steven Young, Itamar Arel, and Jeremy Holleman. A 1 tops/w analog deep machine-learning engine with floating-gate storage in 0.13 ţm cmos. *IEEE Journal of Solid-State Circuits*, 50(1):270–281, 2015.

[24] H.S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-complexity transform and quantization in h.264/avc. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):598–603, 2003.

[25] Abdelhamid Mammeri, Ahmed Khoumsi, Djemel Ziou, and Brahim Hadjou. Modeling and adapting jpeg to the energy requirements of vsn, 2008.

[26] Seyed Jalaleddin Mousavirad and Luís A. Alexandre. Energy-aware jpeg image compression: A multi-objective approach. *Applied Soft Computing*, 141:110278, 2023.

[27] Maged Naser, Mohamed M. Naser, and Lamia H. Shehata. Wireless capsule endoscopy (wce): Review. *International Journal of Progressive Sciences and Technologies*, 36(1):150–167, 2022.

[28] Surya Prakash Noolu, Maryam Shojaei Baghini, and Rajbabu. Velmurugan. Efficient analog architectures for dct processing. pages 346–353, 2010.

[29] M. Pankaala, Kati Virtanen, and Ari Paasio. An analog 2-d dct processor. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(10):1209–1216, 2006.

[30] Farah Saab, Imad Elhajj, Ali Chehab, and Ayman Kayssi. Energy efficient jpeg using stochastic processing. pages 1–6, 12 2012.

[31] Ahmad Shabani and Somayeh Timarchi. Low-power dct-based compressor for wireless capsule endoscopy. *Signal Processing: Image Communication*, 59:83–95, 2017.

[32] M Thiruveni and M Deivakani. Design of analog vlsi architecture for dct. *International Journal of Engineering and Technology*, 2(8):1475–1481, 2012.

[33] Paweł Turcza and Mariusz Duplaga. Energy-efficient image compression algorithm for high-frame rate multi-view wireless capsule endoscopy. *Journal of Real-Time Image Processing*, 16(5):1425–1437, Oct 2019.

[34] J. Volder. The cordic computing technique. 1:257, mar 1959.

[35] G.K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[36] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[37] Wikipedia. Chroma subsampling — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Chroma%20subsampling&oldid=1151196884, 2023. [Online; accessed 04-May-2023].

[38] Wikipedia. Einstein notation — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Einstein%20notation&oldid=1147462442, 2023. [Online; accessed 05-May-2023].

[39] Wikipedia. Frobenius inner product — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Frobenius%20inner%20product&oldid=1123707604, 2023. [Online; accessed 05-May-2023].

[40] Wikipedia. Metric tensor — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Metric%20tensor&oldid=1153039464, 2023. [Online; accessed 05-May-2023].

[41] Wikipedia. Monte Carlo method — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Monte%20Carlo%20method&oldid=1156410026, 2023. [Online; accessed 25-May-2023].

[42] Wikipedia. Peak signal-to-noise ratio — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Peak%20signal-to-noise%20ratio&oldid=1145027401, 2023. [Online; accessed 15-May-2023].

[43] Wikipedia. Structural similarity — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Structural%20similarity&oldid=1142355078, 2023. [Online; accessed 15-May-2023].

[44] Songzhao Xie, Chengyou Wang, and Zhiqiang Yang. Bayer patterned image compression based on apidcbt-jpeg and all phase idct interpolation. 03 2013.

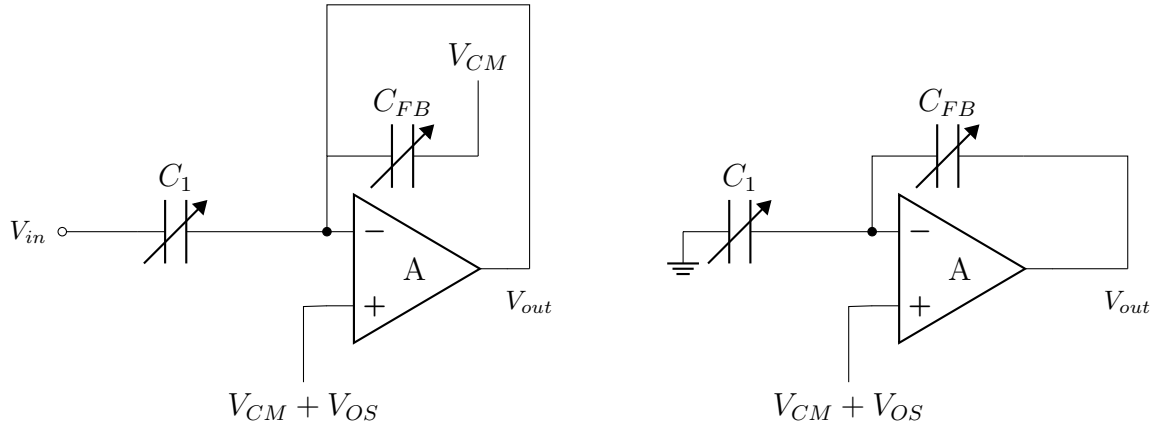
[45] Xilinx. Pg286 demoisaic xilinx ip.

[46] Xiao Zhou, Chengyou Wang, Zhi Zhang, and Qiming Fu. Interpolation filter design based on all-phase dst and its application to image demosaicking. *Information*, 9:206, 08 2018.

Appendix A

MANTIS SCA OTA

Addition



(a) Phase 1 (b) Phase 3

Figure A.1: Two phases of operation for addition with SCA with non ideal OTA.

We will see that the transfer function of the OTA is insensitive to OTA offset and gain. As shown in Fig. A.1, the OTA has a finite gain A and an offset voltage V_{OS} such that during the first phase V_{in} corresponds to the analog pixel and $V_{AMP,IN-}$ is given by

$$V_{AMP,IN-} = \frac{A}{A+1}(V_{CM} + V_{OS}) \tag{A.1}$$

such that the charges at node $V_{AMP,IN-}$ in C_1 and C_{FB} become:

$$Q_1(t = T_2^-) = -C_1 \cdot (V_{in} - V_{AMP,IN-}) \tag{A.2}$$
$$Q_{FB}(t = T_2^-) = -C_{FB} \cdot (V_{CM} - V_{AMP,IN-}) \tag{A.3}$$

After phase 1,the charges contained in each capacitors become:

$$Q_1(t = T_3^-) = C_1 \cdot V_{AMP,IN-} \tag{A.4}$$

$$Q_{FB}(t = T_3^-) = -C_{FB} \cdot (V_{out} - V_{AMP,IN-}) \tag{A.5}$$

By conservation of charges at node $V_{AMP,IN-}$ between the two stages, since during the transition period all switches are open and no charges leak outside this node, we have to fulfil:

$$Q_{FB}(t = T_2^-) + Q_1(t = T_2^-) = Q_{FB}(t = T_3^-) + Q_1(t = T_3^-) \tag{A.6}$$

$$-C_{FB} \cdot (V_{CM} - V_{AMP,IN-}) - C_1 \cdot (V_{in} - V_{AMP,IN-}) = -C_{FB} \cdot (V_{out} - V_{AMP,IN-}) + C_1 \cdot V_{AMP,IN-} \tag{A.7}$$

Which leads to the final expression of V_{out}

$$\boxed{V_{out} = V_{CM} + \frac{C_1}{C_{FB}} V_{in}} \tag{A.8}$$

Similar development could be done for subtraction.

Appendix B

64×64 Images

Partial analysis of 64×64 pixels images. All the development are similar to the one explained in the main text.

B.1 Hard thresholding

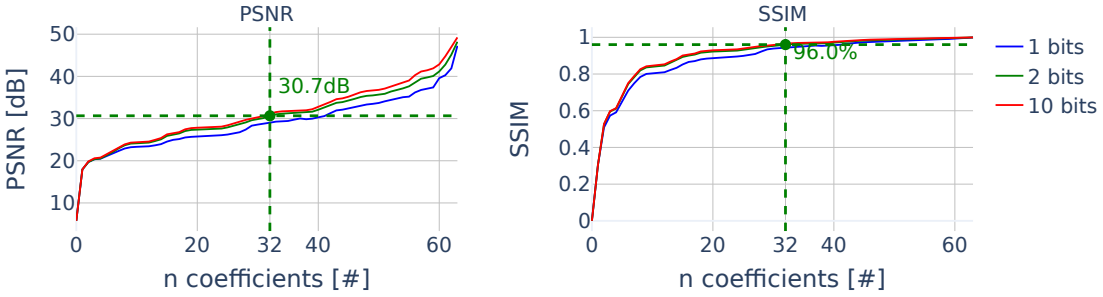


Figure B.1: Evolution of average PSNR and SSIM for different hard thresholding level and bit resolution approximations with an 64×64 image

Appendix C

Code

C-code used to quantize and Huffmann encode values.

C.1 dct.c

```
/*
 *-------------------------------------------------------------------
 * The confidential and proprietary information contained in this file may
 * only be used by a person authorised under and to the extent permitted
 * by a subsisting licensing agreement from ARM Limited.
 *
 *            (C) COPYRIGHT 2010-2013 ARM Limited.
 *                ALL RIGHTS RESERVED
 *
 * This entire notice must be reproduced on all copies of this file
 * and copies of this file may only be made by a person if such person is
 * permitted to do so under the terms of a subsisting license agreement
 * from ARM Limited.
 *
 *      SVN Information
 *
 *      Checked In          : $Date: 2013-03-27 23:58:01 +0000 (Wed, 27 Mar 2013) $
 *
 *      Revision            : $Revision: 242484 $
 *
 *      Release Information : Cortex-M System Design Kit-r1p0-00rel0
 *-------------------------------------------------------------------
 */
```

```
24
25 #ifdef CORTEX_M3
26 #include "CMSDK_CM3.h"
27 #else
28 #include "CMSDK_CM4.h"
29 #endif
30
31 #include <stdio.h>
32 #include <inttypes.h>
33
34 #include "uart_stdout.h"
35 #include "CMSDK_driver.h"
36 #include "dma.h"
37 #include "huffman.h"
38 #include "conf.h"
39
40 // ---------- Parameters to be modified ----------
41 #define DS                  1
42 #define S                   16
43 #define NFILTS              10
44 // -----------------------------------------------
45
46
47 #define NROWS               128
48 #define DATA_SIZE_BYTES     (((NROWS/DS)-16)/S + 1) // In the DCT case,
       8 bytes
49 #define DATA_SIZE_WORDS     ((DATA_SIZE_BYTES>>2) + ((DATA_SIZE_BYTES %
       4) != 0)) // In the DCT case, 2 words of 4 bytes (= 8 bytes)
50
51 #define N_ROW_BLOCKS NROWS/S
52 #define N_COL_BLOCKS NROWS/S
53
54 #define IMAGER_DMA_CH       0
55 #define DMA_CH_NSAMPLES     DATA_SIZE_WORDS
56 #define DMA_CH_R            5 // Not used here, should be more than
     DMA_CH_NSAMPLES
57
58 #define N_WORDS NFILTS*NROWS/S*2
59
60 extern  pl230_dma_data_structure* dma_struct;
61 uint8_t ping = 1;
62
63 uint32_t *image_fragment;
64 uint32_t *image_ptr;
65
66 uint32_t transfer_count = 0;
67 uint32_t ntransfers = 0;
68
69 int main(void)
```

```
70 {
71   ECS_initMCU();
72   ntransfers = DATA_SIZE_BYTES * NFILTS;
73
74   // Initialize UART
75   UartStdOutInit();
76   //printf("Image sensor test - DCT mode\n");
77   //printf("%d words, %d bytes\n", DATA_SIZE_WORDS, DATA_SIZE_BYTES);
78
79   // Write image sensor config. registers
80   CMSDK_IMAGE_SENSOR->MODE = 1;
81   CMSDK_IMAGE_SENSOR->EXPOSURE = 1024;
82   CMSDK_IMAGE_SENSOR->TREAD_PIX = 2;
83   CMSDK_IMAGE_SENSOR->NBITS_ADC = 8;
84   CMSDK_IMAGE_SENSOR->NFILTERS_MAC = NFILTS;
85   CMSDK_IMAGE_SENSOR->TWRITE_MEM = 3;
86   CMSDK_IMAGE_SENSOR->DOWNSAMPLE = DS;
87   CMSDK_IMAGE_SENSOR->STRIDE = S;
88   CMSDK_IMAGE_SENSOR->GAIN_MAC = 7;
89   CMSDK_IMAGE_SENSOR->TENABLE_DRS_OTA = 12;
90   CMSDK_IMAGE_SENSOR->OFFSET_MAC_0 = 0xF1349A86;
91
92   // Initialize DCMI
93   CMSDK_DCMI_Initialize(8, 2, 1, (uint32_t) (DATA_SIZE_WORDS-1), 0);
       //DATA_SIZE_BYTES
94   // 1st argument is the number of bytes to transmit (here, 128
       pixels of 1 byte)
95   // 2nd argument is PIXCLK_CTRL (0 for always on, 1 for manual, 2
       for auto)
96   // 3rd argument is ROW_READY_AUTO (0 for manual row_ready
       generation, 1 for automated)
97   // 4th argument is ROW_READY_ADDR (defines the address at which the
        row_ready signal is generated)
98   // 5th argument is DMA_IRQ_ENABLE (0 to disable DMA IRQ, 1 to
       enable DMA IRQ)
99   //printf("DCMI configured\n");
100
101  // Allocate memory (DMEM) for the feature maps
102  image_fragment = (uint32_t*) malloc(N_WORDS*sizeof(uint32_t));
103  //printf("image_fragment: %x\n", image_fragment);
104
105  // Initialize DMA
106  dma_data_struct_init();
107  dma_pl230_init((uint8_t) IMAGER_DMA_CH);
108  image_ptr = image_fragment;
109  dma_channel_cfg_init((unsigned int) CMSDK_IMAGE_SENSOR_DOUT, (
       unsigned int) image_ptr, &dma_struct->Primary[IMAGER_DMA_CH],
       DMA_CH_R, 2, DMA_CH_NSAMPLES);
110  image_ptr = image_ptr + 2;
```

```
111   dma_channel_cfg_init((unsigned int) (CMSDK_IMAGE_SENSOR_DOUT +
       DATA_SIZE_WORDS), (unsigned int) image_ptr, &dma_struct->Alternate
       [IMAGER_DMA_CH], DMA_CH_R, 2, DMA_CH_NSAMPLES);
112   //printf("dma_struct: %x\n", dma_struct);
113   //printf("DMA configured\n");
114
115   // Enable IRQs from the image sensor (DMA + image sensor)
116   NVIC_ClearPendingIRQ(DMA_IRQn);
117   NVIC_EnableIRQ(DMA_IRQn);
118   //printf("DMA IRQ enabled\n");
119   //NVIC_ClearPendingIRQ(IMAGE_SENSOR_IRQn);
120   //NVIC_EnableIRQ(IMAGE_SENSOR_IRQn);
121   //printf("Imager IRQ enabled\n");
122
123   // Activate DCMI
124   CMSDK_DCMI_Activate();
125
126   // Launch a capture
127   CMSDK_IMAGE_SENSOR->CAPTURE = 1;
128   CMSDK_IMAGE_SENSOR->CAPTURE = 0;
129
130   // Idle
131   while(1){
132     __WFI();
133   }
134
135   return 0;
136 }
137
138 void read_block(uint8_t * img, jpec_block_t * b){
139     /* Reads blocks coming from the camera module in the format:
140         - N_ROW_BLOCKS 1 byte value for each NFILTS
141         - Repeated N_COL_BLOCKS time
142         Ex: (4 col blocks, 2 row blocks, 2 filters (f1,f2), fi
       corresponding to the output
143         of the filter fi for the block at this position)
144        | Fist row       | Second row
145        |1st filt|2ndfilt|
146         f1f1f1f1f2f2f2f2f1f1f1f1f2f2f2f2
147         Output will be the block
148     If the quality factor is to high, it is more efficient to just
       send the raw data without huffman coding.
149     */
150
151     for( int i = 0 ; i<NFILTS; i ++){
152       b->zz[i] = (int8_t) ( (img[i*N_ROW_BLOCKS]-128)/jpec_qzr[i
       ]+0.5);
153     }
154 }
```

```
155
156 void compress(void){
157   int i = 0;
158   int temp = 0;
159
160   // Init of the block
161   jpec_block_t *b = malloc(sizeof(*b));
162   b->zz = malloc(sizeof(uint8_t)*NFILTS);
163   b->len = NFILTS;
164
165   // init of the state
166   jpec_huff_state_t *state = malloc(sizeof(*state));
167   state->nbits = 0;
168   state->dc = 0;
169   state->buffer= 0;
170   state->output_buffer = (uint8_t *) malloc(8*sizeof(uint8_t));
171   set_buffer_to_zero(8,state);
172
173   uint8_t * img = (uint8_t *) image_fragment;
174     for( uint8_t row=0; row <N_ROW_BLOCKS; row ++) {
175         for(uint8_t col=0; col <N_COL_BLOCKS; col ++){
176             read_block(img+col+NFILTS*N_COL_BLOCKS*row, b);
177             huff_encode_block_impl(b,state);
178         }
179     }
180
181   /* Send the remaining content of the buffer to the DCMI*/
182   if(state->output_buffer_size > 0){
183     uint32_t * first_word = state->output_buffer;
184     uint32_t * second_word = state->output_buffer+4;
185     CMSDK_DCMI_WriteDataIn(0, *first_word);
186     CMSDK_DCMI_WriteDataIn(1, *second_word);
187   }
188
189
190 }
191
192 void DMA_Handler(void)
193 {
194   /* Write DMA configuration registers in DMEM */
195   image_ptr = image_ptr + 2;
196   if (ping) {
197     minimal_dma_channel_cfg((unsigned int) CMSDK_IMAGE_SENSOR_DOUT, (
      unsigned int) image_ptr, &dma_struct->Primary[IMAGER_DMA_CH], 2,
      DMA_CH_NSAMPLES);
198   } else {
199     minimal_dma_channel_cfg((unsigned int) (CMSDK_IMAGE_SENSOR_DOUT +
      DATA_SIZE_WORDS), (unsigned int) image_ptr, &dma_struct->
     Alternate[IMAGER_DMA_CH], 2, DMA_CH_NSAMPLES);
```

```
200   }
201   ping = !(ping);
202   transfer_count+=2;
203 
204   if (transfer_count == N_WORDS) {
205     /* DCMI data transfer */
206     compress();
207     //printf("** TEST COMPLETED **\n");
208 
209     /*  Deactivate DCMI */
210     CMSDK_DCMI_Deactivate();
211 
212     /* End simulation */
213     UartEndSimulation();
214   }
215 }
216 
217 void IMAGE_SENSOR_IRQ_Handler(void)
218 {
219   //printf("** TEST COMPLETED **\n");
220 
221   /*  Deactivate DCMI */
222   CMSDK_DCMI_Deactivate();
223 
224   /* End simulation */
225   UartEndSimulation();
226 }
```

C.2 huffman.c

```
1 
2 #include "huffman.h"
3 #include "conf.h"
4 #define JPEC_INT_WIDTH (int)(sizeof(int) * CHAR_BIT)
5 
6 #define JPEC_HUFF_NBITS(JPEC_nbits, JPEC_val) \
7   JPEC_nbits = (!JPEC_val) ? 0 : JPEC_INT_WIDTH - __builtin_clz(
    JPEC_val)
8 
9 
10 
11 void huff_encode_block_impl(jpec_block_t *block,  jpec_huff_state_t *
    s){
12   int i = 1;
13   int val, bits, nbits;
14   int nz = 0;
15   int j  =  0;
16 
```

```
  /* DC coefficient differential encoding */
  val = block->zz[0] - s->dc;
  s->dc = block->zz[0];
  if (val < 0) {
      val = -val;
      bits = ~val;
  }else {
      bits = val;
  }


  JPEC_HUFF_NBITS(nbits, val);

  jpec_huff_write_bits(s, jpec_dc_code[nbits], jpec_dc_len[nbits]);

  if (nbits) jpec_huff_write_bits(s, (unsigned int) bits, nbits);
    /* AC coefficients encoding (w/ RLE of zeros) */

  for (i = 1; i < block->len; i++) {
    if ((val = block->zz[i]) == 0) nz++;
    else {
      while (nz >= 16) {
        jpec_huff_write_bits(s, jpec_ac_code[0xF0], jpec_ac_len[0xF0
    ]); /* ZRL code */
        nz -= 16;
      }
      bits = val;
      if (val < 0) {
        val = -val;
        bits = ~val;
      }
      JPEC_HUFF_NBITS(nbits, val);
      j = (nz << 4) + nbits;

      jpec_huff_write_bits(s, jpec_ac_code[j], jpec_ac_len[j]);
      if (nbits) jpec_huff_write_bits(s, (unsigned int) bits, nbits);
      nz = 0;
    }
  }

  if (block->len < 65) {
    jpec_huff_write_bits(s, jpec_ac_code[0x00], jpec_ac_len[0x00]);
   /* EOB marker */
  }
}
```

```
/* Write n bits into the JPEG buffer, with 0 < n <= 16.
 *
 * == Details
 * - 16 bits are large enough to hold any zig-zag coeff or the longest AC code
 * - bits are chunked into bytes before being written into the JPEG buffer
 * - any remaining bits are kept in memory by the Huffman state
 * - at most 7 bits can be kept in memory
 * - a 32-bit integer buffer is used internally
 * - only the right 24 bits part of this buffer are used
 * - the input bits and remaining bits (if any) are left-justified on this part
 * - a mask is used to mask off any extra bits: useful when the input value has been
 *   first transformed by bitwise complement(|initial value|)
 * - if an 0xFF byte is detected a 0x00 stuff byte is automatically written right after
 */
void jpec_huff_write_bits(jpec_huff_state_t *s, unsigned int bits, int n) {

  int32_t mask   = 0;
  int32_t buffer = 0;
  int32_t nbits = 0;
  int32_t chunk =  0;

  mask = (((int32_t) 1) << n) - 1;
  buffer = (int32_t) bits;
  nbits = s->nbits + n;
  buffer &= mask;
  buffer <<= 24 - nbits;
  buffer = buffer | s->buffer;

  while (nbits >= 8) {
      chunk = (int) ((buffer >> 16) & 0xFF);
      jpec_write_byte(chunk,s);
      if (chunk == 0xFF) jpec_write_byte(0x00,s);
      buffer <<= 8;
      nbits -= 8;
  }
  s->buffer = buffer;
  s->nbits = nbits;
}


void jpec_write_byte(uint8_t val,jpec_huff_state_t* state){
  state->output_buffer[state->output_buffer_size++] = val; /* Write value to buffer position */
```

```
106
107   if (state->output_buffer_size == 8){
108     uint32_t * first_word = state->output_buffer;
109     uint32_t * second_word = state->output_buffer+4;
110     CMSDK_DCMI_WriteDataIn(0, *first_word);
111     CMSDK_DCMI_WriteDataIn(1, *second_word);
112     set_buffer_to_zero(8,state);
113   }
114 }
115
116 void set_buffer_to_zero(uint8_t size, jpec_huff_state_t* state){
117   for(uint8_t i = 0 ; i< size; i++){
118     state->output_buffer[i] = 0;
119   }
120   state->output_buffer_size = 0;
121 }
```

C.3 conf.c

```
1  #include "conf.h"
2
3
4  const uint8_t jpec_dc_len[12] = { 2,3,3,3,3,3,4,5,6,7,8,9 };
5  const uint16_t jpec_dc_code[12] = {
6    0x000,0x002,0x003,0x004,0x005,0x006,
7    0x00e,0x01e,0x03e,0x07e,0x0fe,0x1fe
8  };
9
10 const uint8_t jpec_qzr[64] = {
11   6, 4, 4, 6, 10, 16, 20, 24,
12   5, 5, 6, 8, 10, 23, 24, 22,
13   6, 5, 6, 10, 16, 23, 28, 22,
14   6, 7, 9, 12, 20, 35, 32, 25,
15   7, 9, 15, 22, 27, 44, 41, 31,
16   10, 14, 22, 26, 32, 42, 45, 37,
17   20, 26, 31, 35, 41, 48, 48, 40,
18   29, 37, 38, 39, 45, 40, 41, 40,
19 };
20
21 const uint8_t jpec_ac_len[256] = {
22    4, 2, 2, 3, 4, 5, 7, 8,
23   10,16,16, 0, 0, 0, 0, 0,
24    0, 4, 5, 7, 9,11,16,16,
25   16,16,16, 0, 0, 0, 0, 0,
26    0, 5, 8,10,12,16,16,16,
27   16,16,16, 0, 0, 0, 0, 0,
28    0, 6, 9,12,16,16,16,16,
29   16,16,16, 0, 0, 0, 0, 0,
```

```
  0, 6,10,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0, 7,11,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0, 7,12,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0, 8,12,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0, 9,15,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0, 9,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0, 9,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0,10,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0,10,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0,11,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
  0,16,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0,
 11,16,16,16,16,16,16,16,
 16,16,16, 0, 0, 0, 0, 0
};

const uint16_t jpec_ac_code[256] = {
 0x000a,0x0000,0x0001,0x0004,0x000b,0x001a,0x0078,0x00f8,
 0x03f6,0xff82,0xff83,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x000c,0x001b,0x0079,0x01f6,0x07f6,0xff84,0xff85,
 0xff86,0xff87,0xff88,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x001c,0x00f9,0x03f7,0x0ff4,0xff89,0xff8a,0xff8b,
 0xff8c,0xff8d,0xff8e,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x003a,0x01f7,0x0ff5,0xff8f,0xff90,0xff91,0xff92,
 0xff93,0xff94,0xff95,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x003b,0x03f8,0xff96,0xff97,0xff98,0xff99,0xff9a,
 0xff9b,0xff9c,0xff9d,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x007a,0x07f7,0xff9e,0xff9f,0xffa0,0xffa1,0xffa2,
 0xffa3,0xffa4,0xffa5,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x007b,0x0ff6,0xffa6,0xffa7,0xffa8,0xffa9,0xffaa,
 0xffab,0xffac,0xffad,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x00fa,0x0ff7,0xffae,0xffaf,0xffb0,0xffb1,0xffb2,
 0xffb3,0xffb4,0xffb5,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x01f8,0x7fc0,0xffb6,0xffb7,0xffb8,0xffb9,0xffba,
 0xffbb,0xffbc,0xffbd,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x01f9,0xffbe,0xffbf,0xffc0,0xffc1,0xffc2,0xffc3,
 0xffc4,0xffc5,0xffc6,0x0000,0x0000,0x0000,0x0000,0x0000,
 0x0000,0x01fa,0xffc7,0xffc8,0xffc9,0xffca,0xffcb,0xffcc,
 0xffcd,0xffce,0xffcf,0x0000,0x0000,0x0000,0x0000,0x0000,
```

```
  0x0000,0x03f9,0xffd0,0xffd1,0xffd2,0xffd3,0xffd4,0xffd5,
  0xffd6,0xffd7,0xffd8,0x0000,0x0000,0x0000,0x0000,0x0000,
  0x0000,0x03fa,0xffd9,0xffda,0xffdb,0xffdc,0xffdd,0xffde,
  0xffdf,0xffe0,0xffe1,0x0000,0x0000,0x0000,0x0000,0x0000,
  0x0000,0x07f8,0xffe2,0xffe3,0xffe4,0xffe5,0xffe6,0xffe7,
  0xffe8,0xffe9,0xffea,0x0000,0x0000,0x0000,0x0000,0x0000,
  0x0000,0xffeb,0xffec,0xffed,0xffee,0xffef,0xfff0,0xfff1,
  0xfff2,0xfff3,0xfff4,0x0000,0x0000,0x0000,0x0000,0x0000,
  0x07f9,0xfff5,0xfff6,0xfff7,0xfff8,0xfff9,0xfffa,0xfffb,
  0xfffc,0xfffd,0xfffe,0x0000,0x0000,0x0000,0x0000,0x0000
};
```

C.4 huffman_decoder.py

```
import numpy as np
from bitstream import BitStream

jpec_dc_len = [ 2,3,3,3,3,3,4,5,6,7,8,9 ]

jpec_dc_code = [
    0x000,0x002,0x003,0x004,0x005,0x006,
    0x00e,0x01e,0x03e,0x07e,0x0fe,0x1fe
]

jpec_ac_len = [
     4, 2, 2, 3, 4, 5, 7, 8,
    10,16,16, 0, 0, 0, 0, 0,
     0, 4, 5, 7, 9,11,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 5, 8,10,12,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 6, 9,12,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 6,10,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 7,11,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 7,12,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 8,12,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 9,15,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 9,16,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0, 9,16,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0,10,16,16,16,16,16,16,
```

```
    16,16,16, 0, 0, 0, 0, 0,
     0,10,16,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0,11,16,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
     0,16,16,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0,
    11,16,16,16,16,16,16,16,
    16,16,16, 0, 0, 0, 0, 0
]

jpec_ac_code = [
    0x000a,0x0000,0x0001,0x0004,0x000b,0x001a,0x0078,0x00f8,
    0x03f6,0xff82,0xff83,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x000c,0x001b,0x0079,0x01f6,0x07f6,0xff84,0xff85,
    0xff86,0xff87,0xff88,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x001c,0x00f9,0x03f7,0x0ff4,0xff89,0xff8a,0xff8b,
    0xff8c,0xff8d,0xff8e,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x003a,0x01f7,0x0ff5,0xff8f,0xff90,0xff91,0xff92,
    0xff93,0xff94,0xff95,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x003b,0x03f8,0xff96,0xff97,0xff98,0xff99,0xff9a,
    0xff9b,0xff9c,0xff9d,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x007a,0x07f7,0xff9e,0xff9f,0xffa0,0xffa1,0xffa2,
    0xffa3,0xffa4,0xffa5,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x007b,0x0ff6,0xffa6,0xffa7,0xffa8,0xffa9,0xffaa,
    0xffab,0xffac,0xffad,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x00fa,0x0ff7,0xffae,0xffaf,0xffb0,0xffb1,0xffb2,
    0xffb3,0xffb4,0xffb5,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x01f8,0x7fc0,0xffb6,0xffb7,0xffb8,0xffb9,0xffba,
    0xffbb,0xffbc,0xffbd,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x01f9,0xffbe,0xffbf,0xffc0,0xffc1,0xffc2,0xffc3,
    0xffc4,0xffc5,0xffc6,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x01fa,0xffc7,0xffc8,0xffc9,0xffca,0xffcb,0xffcc,
    0xffcd,0xffce,0xffcf,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x03f9,0xffd0,0xffd1,0xffd2,0xffd3,0xffd4,0xffd5,
    0xffd6,0xffd7,0xffd8,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x03fa,0xffd9,0xffda,0xffdb,0xffdc,0xffdd,0xffde,
    0xffdf,0xffe0,0xffe1,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0x07f8,0xffe2,0xffe3,0xffe4,0xffe5,0xffe6,0xffe7,
    0xffe8,0xffe9,0xffea,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x0000,0xffeb,0xffec,0xffed,0xffee,0xffef,0xfff0,0xfff1,
    0xfff2,0xfff3,0xfff4,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x07f9,0xfff5,0xfff6,0xfff7,0xfff8,0xfff9,0xfffa,0xfffb,
    0xfffc,0xfffd,0xfffe,0x0000,0x0000,0x0000,0x0000,0x0000
]
to_bin = np.vectorize(lambda i,j: '{0:0{1}b}'.format(i,j))

class Node(object):
    def __init__(self,val = None):
```

```
84          self.val = val
85
86  class NodeTree(object):
87      def __init__(self, left=None, right=None,val =None):
88          self.left = left
89          self.right = right
90          self.val = val
91
92      def children(self):
93          return self.left, self.right
94
95      def __str__(self):
96          if val:
97              return str(val)
98          return str( (self.left, self.right))
99
100 def get_val(path,root):
101     if len(path) == 0:
102         return root.val
103     else:
104         if path[0] == "0":
105             return get_val(path[1:],root.left)
106         if path[0] == "1":
107             return get_val(path[1:],root.right)
108
109 def makeTree(root,path,val):
110     if len(path)>0:
111         if path[0] == '0':
112             if root.left == None:
113                temp = NodeTree()
114             else:
115                 temp = root.left
116             root.left = makeTree(temp,path[1:], val)
117
118         elif path[0] == '1':
119             if root.right == None:
120                 temp = NodeTree()
121             else:
122                 temp = root.right
123             root.right = makeTree(temp, path[1:], val)
124
125     if len(path) == 0:
126         return NodeTree(val=val)
127     return root
128
129 def RemoveFF00(data):
130     """
131     Removes 0x00 after 0xff in the image scan section of JPEG
132     """
```

```
133     datapro = []
134     i = 0
135     while True and i < len(data)-1:
136         b =  data[i]
137         bnext = data[i+1]
138         if b == 0xFF:
139             if bnext != 0:
140                 break
141             datapro.append(data[i])
142             i += 2
143         else:
144             datapro.append(data[i])
145             i += 1
146     datapro.append(data[i])
147     return datapro
148 
149 
150 
151 
152 def findDCSymbol(stream,root):
153     if root.val != None:
154         return root.val
155     else:
156         sel = stream.read(bool)
157         if sel == 0:
158             return findDCSymbol(stream,root.left)
159         if sel == 1:
160             return findDCSymbol(stream,root.right)
161 # for dc_code in jpec_dc_code:
162 
163 
164 def get_int(stream, len):
165     #sign = stream.read(bool)
166     if len == 0:
167         return 0
168     val = stream.read(bool,len)
169     sign = val[0]
170     bin_rep = ''
171     if sign:
172         for bit in val:
173             bin_rep += str(int(bit))
174         return int(bin_rep,2)
175 
176     else:
177         for bit in val:
178             bin_rep += str(int( not bool(bit)))
179         return -int(bin_rep,2)
180 
181 def read_block(stream,prev_dc=0):
```

```
182     block = np.zeros(64)
183 
184     dc_len = findDCSymbol(stream,DCtree)
185     print(dc_len)
186     block[0] = get_int(stream, dc_len-1)+prev_dc
187     coeff_count = 1
188 
189     while( coeff_count < 64):
190         code = findDCSymbol(stream,ACtree)
191         if code == (0,0): #EOB
192             break
193 
194         coeff_count += code[0] # Skip zeros
195         val = get_int(stream, code[1])
196 
197         block[coeff_count] = val+128
198         coeff_count +=1
199     return block
200 
201 ## Build tree for decoding
202 
203 DCtree = NodeTree()
204 ACtree = NodeTree()
205 
206 ac_code_val = []
207 ac_code_val.append((0,0)) # EOB
208 for i in range(16):
209     for j in range(1,11):
210         if i== 15 and j == 1:
211             ac_code_val.append((i,0)) # ZRL
212         ac_code_val.append((i,j))
213     if i == 14 or i == 15:
214         for k in range(5):
215             ac_code_val.append((0,0))
216     else:
217         for k in range(6):
218             ac_code_val.append((0,0))
219 
220 # Building DC tree
221 for val,path in enumerate(to_bin(jpec_dc_code,jpec_dc_len)):
222     indices = list(path)
223     makeTree(DCtree,indices,val+1)
224 
225 # Building AC tree
226 for val,path in enumerate(to_bin(jpec_ac_code,jpec_ac_len)):
227     indices = list(path)
228     if path != "0":
229         makeTree(ACtree,indices,ac_code_val[val])
230 
```

```
231 # Read Hufmman encode file as 8byte uint per line
232 path = "/home/thetheos/Documents/Universite/MA2/memoire/
        custom_huff_coding_mantis/huffman_encoded"
233 f = open(path, "rb")
234 arr = np.loadtxt(path,dtype=np.uint8)
235
236 to_bin_encoded = np.vectorize(lambda i: '{0:08b}'.format(i))
237 byte_list = list(to_bin_encoded(RemoveFF00(arr)))
238 stream = BitStream()
239 for byte in byte_list:
240     for bit in byte:
241         stream.write(int(bit),bool)
242 # print(stream)
243 prev_dc = 128
244 for i in range(64):
245     block = read_block(stream,prev_dc)
246     prev_dc = block[0]
247     print(" Block ", i,block)
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**