

**FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO**



# **Algorithm and Hardware Design for Image Restoration**

**Bernardo Manuel Aguiar Silva Teixeira Cardoso**

FOR JURY EVALUATION

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Vítor Manuel Grade Tavares

Second Supervisor: Xin Li (Carnegie Mellon University)

June 29, 2015



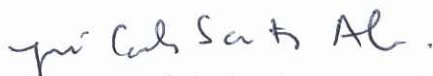


A Dissertação intitulada

“Algorithm and Hardware Design for Image Restoration”

foi aprovada em provas realizadas em 23-07-2015

o júri



Presidente Professor Doutor José Carlos dos Santos Alves  
Professor Associado do Departamento de Engenharia Eletrotécnica e de  
Computadores da Faculdade de Engenharia da Universidade do Porto



Professor Doutor Jorge Miguel N. S. Cabral  
Professor Auxiliar do Departamento de Electrónica Industrial da Escola de  
Engenharia da Universidade do Minho



Professor Doutor Vitor Manuel Grade Tavares  
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores  
da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.



Autor - Bernardo Manuel Aguiar Silva Teixeira Cardoso



# Resumo

O reconhecimento automático de cenários com base em informações presentes numa imagem é uma tarefa importante para dispositivos portáteis com muitas aplicações práticas (por exemplo, detecção facial num *smart phone*). Por outro lado, o consumo energético e a espessura de dispositivos portáteis são duas considerações importantes subjacentes ao projeto de telefones e *tablets*. O principal efeito colateral destas restrições é um módulo de câmara mais pequeno, que funciona com lentes com pequenas aberturas e sensores com pixéis pequenos - ambos limitando a quantidade de luz detetada pela câmara. Isto resulta numa redução da qualidade de fotografias em condições adversas de imagem, como por exemplo, pouca luz ou tempo de exposição pequeno para a deteção de objetos em movimento. A qualidade reduzida manifesta-se num aumento do ruído, aumento de *blur*, e falta de contraste. Por sua vez, estas condições não ideais limitam, em prática, a precisão do reconhecimento de cenários.

Neste trabalho, é apresentado um mecanismo para a eliminação de ruído em imagens (*image denoising*), que opera em tempo real e com pouco esforço computacional, melhorando a qualidade das fotografias tiradas com uma câmara e, consequentemente, a precisão do reconhecimento de cenários. Uma nova arquitetura de *hardware* é proposta para implementar o algoritmo de restauração de imagens acima mencionado com o suporte de uma FPGA.

O objetivo principal do trabalho é melhorar drasticamente a qualidade das imagens e, consequentemente, a qualidade de classificação em tempo real, tirando partido da implementação em FPGA proposta.



# Abstract

Automatic recognition of a scene based on the information present in an image is an important task for portable devices with many practical applications (e.g., face detection on a smart phone). On the other hand, power consumption and thinness of portable devices are two important considerations behind the design of cell phones and tablets. A side-effect of these constraints is a smaller camera module that works with lenses with small apertures and sensors with tiny pixels - both limiting the amount of light sensed by the camera. It results in reduced quality of photographs under adverse imaging conditions such as low light and small exposure time for imaging fast moving objects. The reduced quality manifests itself in increased noise, increased blur, and lack of contrast. These non-idealities, in turn, limit the accuracy of scene recognition in practice.

In this work, it is proposed to build a real time, computationally inexpensive image denoising engine that enhances the quality of photographs taken by a camera and, consequently, the accuracy for scene recognition. A novel hardware architecture is proposed to implement the aforementioned image restoration algorithm with FPGA.

The main goal is to largely enhance the quality of images and, hence, the quality of classification in real time by taking advantage of the proposed FPGA implementation.



# Agradecimentos

Este trabalho representa a conclusão de cinco anos de aprendizagem, acumulação de conhecimento, trabalho duro e vários momentos e experiências inesquecíveis. Todos estes divididos entre viver e viajar em várias cidades por toda a Europa e, mais recentemente, na América do Norte.

Todas essas experiências não poderiam ser possíveis sem o apoio constante e completo de toda a minha família. Quero por isso agradecer a toda a minha família, especialmente os meus pais e avós. Para os meus pais, muito obrigado por todo o apoio e por sempre providenciarem todas as condições para o meu desenvolvimento como pessoa, por todas as lições de vida, por toda a paciência e sacrifícios, por sempre incentivando-me a ser melhor e melhor, e por apoiarem todas as minhas decisões. Para os meus avós, obrigado por todas as valiosas lições de vida e histórias fantásticas, por sempre me apoiarem em toda a minha vida, especialmente nos últimos cinco anos, e por sempre estarem presentes, de uma maneira ou de outra.

De seguida, quero agradecer aos meus orientadores da dissertação, o Professor Vítor Grade Tavares e o Professor Xin Li, por todo o apoio prestado durante os últimos seis meses. Para o Professor Vítor, por cultivar o meu interesse pela eletrónica, por todas as discussões e ajuda fornecida durante os últimos dois anos e especialmente nos últimos 6 meses, e por me encorajar a prosseguir este projeto. Para o professor Xin, por todo o apoio durante a minha estadia de três meses na Carnegie Mellon University, por sempre manter um olhar atento sobre o meu trabalho, e por ter sempre as melhores ideias e ajudar com todos os problemas. Quero, também, estender um agradecimento ao Minho Won, por ajudar com tudo quando era necessário. Finalmente, um agradecimento especial a todos os professores que conheci e trabalhei nos últimos cinco anos, o que contribuiu extremamente para o meu futuro como Engenheiro.

Durante esta viagem de cinco anos tive a sorte de conhecer muitas pessoas maravilhosas, que eu me orgulho de chamar de amigos. Tenho imensa pena, mas por outro lado sorte, de dizer que não é possível incluir todos nesta pequena página. Por isso, quero fazer um agradecimento especial a alguns deles. Para o António, por me apoiar desde que me lembro, e por ser como um irmão. Para o Trio do Creamfields: Xico e Yak, por todos os bons momentos e aventuras que passamos durante estes cinco anos (#VaiTodo). Para a CMU team: Xico, JM e Cesário, pelas trocas de ideias, as brincadeiras ocasionais e as roadtrips. Para os companheiros de festivais: Fiskas, Jarro, Joaquina, Mafalda, por todos os festivais e verões incríveis, e por estarem sempre presentes, mesmo não falando tão frequentemente como gostaríamos. E por último, mas não menos importante, para o Bife, Carlos, Tico, Pete D, Pereira, Diogo, Vini, Conde, Duda, um obrigado por estes cinco anos de estudo e festas, e por todos juntos fazermos de Electro 2010 um dos melhores anos de sempre!

Bernardo Cardoso





# Acknowledgments

This work represents the conclusion of five years of learning, knowledge accumulation, hard work and various unforgettable moments and experiences. All of these span across living and traveling in various cities all across Europe and, more recently, North America.

All of these experiences couldn't be possible without the constant and full support of all my family. I want to thank everyone of them, specially my parents and grandparents. To my parents, thank you for all the support and for providing all the best for my development as a person, for all the life lessons, for all the patience and sacrifices, for always encouraging me to be better and better, and for supporting all my decisions. To my grandparents, thank you for all the valuable life lessons and amazing stories, for always supporting me throughout all my life, specially in the last five years, and for always being there for me.

I want to extend a thank you to my dissertation advisers, Professor Vítor Grade Tavares and Professor Xin Li, for all the assistance provided during the past six months. To Professor Vítor, for cultivating my interest in electronics, for all the discussions and help provided during the past couple of years and specially the last 6 months, and for encouraging me to pursue this project. To Professor Xin, for all the support during my three month stay at Carnegie Mellon University, for always keeping a close eye on my work, and for always having the best ideas and helping with all the problems. Also, I would like to extend a thank you to Minho Won, for helping with everything when it was necessary. Finally, a special thank you to all the Professors that I met and worked with in the past five years, which contributed greatly for my future as an Engineer.

During this five year journey I was fortunate enough to meet many amazing people, which I'm proud to refer to as friends. I am sorry, but also fortunate, to say that I can't possible include all of them in this small page. Hence, I want to extend a special thank you to some of them. To António, for supporting me since I can remember, and for being like a brother. To the Creamfields Trio: Xico and Yak, for all the good times and adventures we had through this five years (#VaiTodo). To the CMU team: Xico, JM and Cesarius, for the exchange of ideas, the occasional banter and the roadtrips. To the best festival crew: Físgas, Jarro, Joaninha, Mafalda, for all the amazing festivals and summers, and for always being there, even when we don't talk as often as we would like. And last, but not least, to Bife, Carlos, Tico, Pete D, Pereira, Diogo, Vini, Conde, Duda, for these five years of studying and partying together, and for making Electro's class of 2010 one for the books.

Bernardo Cardoso



*“Quiet people have the loudest minds.”*

Stephen Hawking



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Objectives . . . . .	2
1.3	Challenges . . . . .	2
1.4	Structure . . . . .	3
<b>2</b>	<b>Image Denoising and current State of the Art</b>	<b>5</b>
2.1	Image Denoising . . . . .	5
2.2	Transform domain denoising . . . . .	7
2.2.1	Wavelets . . . . .	8
2.2.2	DCT . . . . .	10
2.2.3	Hard thresholding . . . . .	11
2.2.4	Wiener filtering . . . . .	12
2.3	Compressed Sensing based denoising . . . . .	12
2.3.1	PCA . . . . .	13
2.3.2	K-SVD . . . . .	14
2.4	Related Work . . . . .	15
2.4.1	Denoising Algorithms . . . . .	15
2.4.2	Hardware Implementations . . . . .	17
<b>3</b>	<b>BM3D Denoising Algorithm</b>	<b>21</b>
3.1	BM3D Algorithm . . . . .	21
3.2	Implications for Hardware Design . . . . .	22
3.2.1	Modifications . . . . .	23
3.3	MATLAB Implementation . . . . .	24
3.3.1	Initialization . . . . .	24
3.3.2	Noise power estimation . . . . .	25
3.3.3	Execution . . . . .	25
3.3.4	Results . . . . .	26
<b>4</b>	<b>Hardware Implementation</b>	<b>29</b>
4.1	System Architecture . . . . .	29
4.2	The Matching Processor . . . . .	31
4.2.1	Memory Module . . . . .	31
4.2.2	$\ell_1$ norm Unit . . . . .	33
4.2.3	Sorter Unit . . . . .	34
4.2.4	Control Modules . . . . .	35
4.3	The Denoising Pipeline . . . . .	38

4.3.1	Multiplier-less DCT and IDCT . . . . .	39
4.3.2	Haar Transform Modules . . . . .	42
4.3.3	Hard Thresholding Module . . . . .	43
4.3.4	Position Memory and Decoder . . . . .	44
4.3.5	Memory Module . . . . .	45
4.3.6	Control Module . . . . .	45
4.4	Top Level Module . . . . .	47
4.4.1	Memory Modules and Control . . . . .	47
4.4.2	Master Control . . . . .	51
4.5	BM3D IP Core . . . . .	53
4.5.1	AXI Lite Slave . . . . .	54
4.5.2	The MMCM . . . . .	54
<b>5</b>	<b>Test Methodology and Results</b>	<b>55</b>
5.1	Experimental Setup . . . . .	55
5.1.1	ZYNQ Board Setup . . . . .	56
5.1.2	C Program . . . . .	57
5.2	Results . . . . .	59
5.2.1	FPGA Resource Usage . . . . .	59
5.2.2	Denoising Performance . . . . .	60
5.2.3	Execution Time . . . . .	63
5.2.4	Power Consumption . . . . .	65
<b>6</b>	<b>Conclusions and Future Work</b>	<b>69</b>
6.1	Conclusions . . . . .	69
6.2	Future Work . . . . .	70
<b>A</b>	<b>Result Images</b>	<b>71</b>
	<b>References</b>	<b>79</b>

# List of Figures

2.1	One step of a wavelet decomposition and reconstruction. Taken from [7] . . . . .	9
2.2	2D separable implementation of DWT and its inverse. Taken from [8] . . . . .	10
2.3	Basis functions for a 8x8 2D DCT . . . . .	11
2.4	a) Hard thresholding operator. b) Soft thresholding operator. Adapted from [7] . .	12
2.5	The K-SVD algorithm. Taken from [18] . . . . .	14
3.1	a) Original Cameraman image with size 256x256; b) Noisy image with $\sigma = 25$ (PSNR=20.18 dB); c) Basic estimate image obtained by modified BM3D (PSNR=27.16 dB); d) Basic estimate obtained by original BM3D (PSNR=29.14 dB) . . . . .	27
3.2	a) Original Lena image with size 512x512; b) Noisy image with $\sigma = 25$ (PSNR=20.19 dB); c) Basic estimate image obtained by modified BM3D (PSNR=30.32 dB); d) Basic estimate obtained by original BM3D (PSNR=31.37 dB) . . . . .	28
4.1	BM3D System Architecture Block Diagram . . . . .	30
4.2	Matching Processor Schematic . . . . .	32
4.3	RTL schematic of the $\ell_1$ norm unit . . . . .	34
4.4	RTL schematic of one sorter cell . . . . .	35
4.5	State diagram of the matching processor's memory control module . . . . .	36
4.6	State diagram of the matching processor's control module . . . . .	37
4.7	Denoising Pipeline Schematic . . . . .	38
4.8	IDCT Flow graph algorithm. Taken from [49] . . . . .	40
4.9	Transposer structure. Adapted from [51] . . . . .	41
4.10	RTL schematics of the haar and inverse haar transform modules . . . . .	43
4.11	RTL schematics of the hard thresholding module . . . . .	44
4.12	State diagram of the denoising pipeline control module . . . . .	46
4.13	State diagram of the master control module . . . . .	51
4.14	BM3D IP core block . . . . .	53
5.1	Board Design Schematic . . . . .	56
5.2	a) Basic estimate image obtained by modified BM3D (PSNR=27.16 dB) in MATLAB; b) Basic estimate obtained by modified BM3D (PSNR=27.17 dB) in hardware. Noisy image was corrupted with noise power of 25. . . . .	60
5.3	a) Basic estimate image obtained by modified BM3D (PSNR=30.32 dB) in MATLAB; b) Basic estimate obtained by modified BM3D (PSNR=30.35 dB) in hardware. Noisy image was corrupted with noise power of 25. . . . .	62
5.4	a) Average PSNR (dB) for 512x512 images with noise powers from 5 to 50. b) Average SSIM for 512x512 images with noise powers from 5 to 50. . . . .	66

5.5	a) FPGA vs CPU1 speedup for increasing image resolutions. b) FPGA vs CPU2 speedup for increasing image resolutions. Images are in the same order as in tables 5.6 and 5.7. . . . .	67
5.6	Power consumption distribution of the BM3D system for different operating frequencies. . . . .	68
A.1	a) Original image <i>City</i> with 1MP resolution (1280x960). b) Original image <i>Coat</i> with 2MP resolution (1920x1080). c) Original image <i>Bridge</i> with 3MP resolution (2048x1536). d) Original image <i>Palace</i> with 5MP resolution (2560x1920). e) Original image <i>NYC</i> with 8MP resolution (3264x2448). . . . .	72
A.2	a) Noisy image <i>City</i> with noise power of 40. b) Denoised image with PSNR=24.88 dB and SSIM=0.761. . . . .	73
A.3	a) Noisy image <i>Coat</i> with noise power of 20. b) Denoised image with PSNR=33.29 dB and SSIM=0.882. . . . .	74
A.4	a) Noisy image <i>Bridge</i> with noise power of 50. b) Denoised image with PSNR=23.98 dB and SSIM=0.740. . . . .	75
A.5	a) Noisy image <i>Palace</i> with noise power of 30. b) Denoised image with PSNR=28.21 dB and SSIM=0.811. . . . .	76
A.6	a) Noisy image <i>NYC</i> with noise power of 40. b) Denoised image with PSNR=29.38 dB and SSIM=0.848. . . . .	77



# List of Tables

3.1	PSNR (dB) results for 256x256 Cameraman Image . . . . .	26
3.2	PSNR (dB) results for 512x512 Lena Image . . . . .	26
4.1	Hard coded image widths . . . . .	48
5.1	FPGA Resource Usage . . . . .	59
5.2	PSNR (dB) results for 256x256 Cameraman Image . . . . .	60
5.3	PSNR (dB) results for 512x512 Lena Image . . . . .	61
5.4	PSNR (dB) results comparison for 1,2,3,5 and 8 MP images. . . . .	62
5.5	SSIM results comparison for 1,2,3,5 and 8 MP images. . . . .	63
5.6	FPGA execution time results (in seconds) for images of increasing resolutions and frequencies. . . . .	63
5.7	Comparison of CPU and FPGA @125MHz execution time results (in seconds) for images of increasing resolutions. . . . .	64
5.8	PL and PS power consumption (W) for different frequencies of operation. . . . .	65



# Abbreviations

ADC	Analog to Digital Converter
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
AXI	Advanced eXtensible Interface
BM3D	Block Matching and 3D Filtering
BRAM	Block RAM
CAN	Controller Area Network
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CS	Compressed Sensing
CSR	Centralized Sparse Representation
CWT	Continuous Wavelet Transform
DCT	Discrete Cosine Transform
DDR3	Double Data Rate Type 3
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
DSP	Digital Signal Processor
DWT	Discrete Wavelet Transform
FCT	Fast Cosine Transform
FF	Flip-Flop
FFT	Fast Fourier Transform
FGA	Flow Graph Algorithm
FPGA	Field Programmable Gate Array
fps	Frames per Second
FSM	Finite State Machine
GPU	Graphics Processing Unit
IDCT	Inverse Discrete Cosine Transform
I/O	Input/Output
IP	Intellectual Property

JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
LUT	Look-up Table
MAD	Median Absolute Deviation
MMCM	Mixed-Mode Clock Manager
MP	Mega Pixel
MRA	Multiresolution Analysis
MSB	Most Significant Bit
MSE	Mean Squared Error
NCSR	Nonlocally Centralized Sparse Representation
OS	Operating System
PCA	Principle Component Analysis
PL	Programmable Logic
PS	Processing System
PSNR	Peak Signal to Noise Ratio
RAM	Random Access Memory
RGB	Red Blue Green
RTL	Register Transfer Level
SAPCA	Shape Adaptive Principle Component Analysis
SCN	Sparse Coding Noise
SNR	Signal to Noise Ratio
SoC	System on Chip
SPI	Serial Peripheral Interface
SSH	Secure Shell
SSIM	Structural Similarity Index
SVD	Singular Value Decomposition
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

# Chapter 1

## Introduction

### 1.1 Motivation

In the rising market of portable devices, thinness and power consumption are two main considerations when producing a competitive and successful device, for example, a cell phone or a tablet. A negative side effect of these constraints is a smaller camera module, which in turn has lenses with small apertures and sensors with tiny pixels. Both these characteristics limit the amount of light that is sensed by the camera, resulting in a reduced quality of photographs under adverse conditions, such as, low light environments and small exposure times when dealing with fast moving objects. This reduced quality appears in the image as increased motion blur, lack of contrast and increased noise, all non-idealities that limit the accuracy of scene recognition.

Automatic scene recognition based on the information present in an image is an important task for portable devices with many practical applications, for example, face detection on a smart phone. This way, the corrupted image needs to be restored in order to achieve better results of recognition, which motivates the use of an image restoration algorithm to process the image before the recognition task. Hence, it is of major importance to have fast and effective restoration of an image in order to process the recognition task in real time.

The image restoration task is a heavily researched subject, with many algorithms being proposed in the past few years, originating from various areas of research such as probability theory, statistics, linear and nonlinear filtering, and spectral analysis. On the other hand, all these algorithms share a similar concept: they rely on implicit or explicit assumptions about the true image, in order to separate it properly from the noise. This separation is made possible by applying transforms to image patches or using dictionaries, in order to make the resulting image representation sparse. Then, a simple thresholding operation can be applied, discarding the smaller coefficients, which are assumed to be noise.

Nonetheless, most of these algorithms are highly computational demanding, due to the use of transforms or dictionaries (which in turn involve training). Hence, they take a large amount of time to produce a restored image, when implemented in software on a general purpose CPU. This

motivates the implementation of specialized hardware to deal with a corrupted image in real time, producing the restored image in the fastest time possible and with enhanced quality.

In this work, a computationally inexpensive, low power and real time image denoising approach will be presented, based on modifications of the BM3D image denoising algorithm, proposed in [1]. This algorithm enhances the quality of photographs taken by a camera and, consequently, the accuracy for scene recognition. A novel hardware architecture will be presented to implement the aforementioned image denoising algorithm using an FPGA.

## 1.2 Objectives

Image denoising algorithms can be implemented using general purpose CPUs, GPUs or specialized cores. The easiest solution is the implementation in a high level language such as C or C++ in a CPU, however it is also the slowest. A GPU is specialized to deal with graphics and therefore it is more adequate to use as an image processor. On the other hand, most GPUs consume a lot of power when faced with the kind of tasks posed by most denoising algorithms. Therefore, the best solution is the development of a specialized core, able to denoise an image in real time and with low power consumption, without compromising the quality of the restored image.

With the reduction of costs in the fabrication of CMOS circuits, the FPGA platforms are a very appealing solution for the fast prototyping of novel hardware implementations. Hence, the main objectives of this work are: the development, on an FPGA, of a fully specialized core for image denoising using the BM3D algorithm; achieve extensive improvements of the quality of images; and guarantee real time performance, with low power consumption.

## 1.3 Challenges

The hardware implementation of an image denoising algorithm poses many challenges. To the best of the author's knowledge, there is no work on the literature addressing the hardware implementation of the BM3D image denoising algorithm, which establishes the main challenge of this work. Furthermore, a second challenge is that the BM3D algorithm is highly complex, containing intensive arithmetic operations that need to be optimized. This is a challenge for designing fast hardware that can perform such operations without a loss of precision in the results obtained. The third challenge is the small amount of work found in the literature regarding the implementation of other denoising algorithms in hardware. Hence, the architecture must be developed almost from scratch. The final challenge regards the real time performance and power consumption trade-off. It is necessary to push the limits when designing hardware that needs simultaneously to be the fastest possible, while operating with minimal power consumption.

## 1.4 Structure

In addition to the Introduction, this document contains five more chapters. In chapter 2, the background on image denoising concepts is presented. This is done in a top down approach, starting from more general concepts and towards more specialized ones with direct applications in the algorithm implemented. Furthermore, in this chapter, the current state of the art in image denoising algorithms and their hardware implementations is presented. In chapter 3, the chosen image denoising algorithm is analyzed, as well as its software implementation in MATLAB. The results of this implementation are also presented and discussed. Follows chapter 4, in which the hardware implementation is analyzed in extensive detail, since this is the main focus of this work. In chapter 5, the performance of the implemented design is evaluated, and several results are presented. Finally, in chapter 6, the results and contributions of this work are summarized, and possibilities for future work are presented.





## Chapter 2

# Image Denoising and current State of the Art

In this chapter a simplified approach to all the concepts necessary for understanding the literature on image denoising will be presented. Furthermore, related work on image denoising algorithms and respective hardware implementations will be analyzed and discussed.

### 2.1 Image Denoising

In the process of capturing an image, for example, using a CMOS image sensor in a regular photographic camera, there are various constraints that influence the quality of the image produced. These constraints generate non-idealities in the image that from a visual standpoint manifest themselves as distortion, blur, degradation in an apparent random way (Gaussian noise), etc. The purpose of image restoration is to restore such an image to its original content and quality.

**Image Restoration** is the operation of taking a corrupted/noisy image and estimating the clean original image. Corruption may come in many forms such as motion blur, noise, and camera misfocus. [2]

**Image Denoising** is a method of image restoration that deals specifically with restoring an image that has been corrupted with noise. Image noise can be defined as a random variation of brightness or color in images produced by the components of a digital camera that intervene in the process of forming said images. Image noise can be of the following types [3]:

- **Gaussian noise:** statistical random noise, that affects each pixel independently of its position and signal intensity. It is caused primarily by Johnson-Nyquist noise (thermal noise) coming from the signal amplifier in CMOS image sensors. This is the cause of the constant noise level that can be seen in dark areas of an image, which is commonly known as white noise.
- **Salt and Pepper noise:** a wide variety of processes that result in the same basic image degradation are referred to as Salt and Pepper noise. This degradation occurs only for a

few pixels, but these pixels are very noisy, causing an effect similar to sprinkling white and black dots on the image (thus the name salt and pepper). This type of noise can be caused by ADC errors, bit errors in transmission and others.

- **Shot noise:** also called photon counting noise, it is caused by statistical quantum variations in the number of photons sensed at a given exposure level. Given its quantum nature, this type of noise is always present in any imaging device, and it follows a Poisson distribution, with an intensity proportional to the square root of the image intensity.
- **Quantization noise:** converting a continuous random variable to a discrete one results in quantization noise. In images, this occurs in the acquisition process, when the pixels of a sensed image are quantized to a number of discrete levels. This type of noise has an uniform distribution.
- **Anisotropic noise:** this type of noise is, as the name states, orientation dependent and it can cause periodic artifacts in images, visible as vertical or horizontal stripes for example.

From all the noise types the most frequent and thus the one with an increased impact in image quality is Gaussian noise. Considering this, the focus of this work will be applying image denoising in order to restore an image affected by Gaussian noise with different powers.

The classic image denoising problem can be described as follows: an ideal image  $\mathbf{y}$  is affected by additive zero-mean white and homogeneous Gaussian noise,  $\mathbf{n}$ , with standard deviation  $\sigma(n)$ . The measured image  $\mathbf{z}$  is given by

$$z(x) = y(x) + n(x), \quad x \in X \quad (2.1)$$

where  $x$  is a 2D spatial coordinate that belongs to the image domain  $X$ . In order to be able to compare different image denoising algorithms, a measure of their performance must be chosen. This can be done by computing some well known quantities, for example, the signal to noise ratio (SNR), peak signal to noise ratio (PSNR) or mean squared error (MSE). The SNR can be defined [4] as

$$SNR = \frac{\sigma(y)}{\sigma(n)}, \quad (2.2)$$

where  $\sigma(y)$  denotes the empirical standard deviation of  $y$ ,

$$\sigma(y) = \sqrt{\frac{1}{|X|} \sum_{x \in X} (y(x) - \bar{y})^2}, \quad (2.3)$$

and  $\bar{y}$  is the average gray-level value. However, in order to compute the SNR it is necessary to know beforehand the value of the standard deviation of the noise  $\sigma(n)$ , which can only be obtained by estimation or formally computed when the noise model is known. In order to eliminate this

dependency of the noise variance, one can use the MSE which is given by

$$MSE = \frac{1}{|X|} \sum_{x \in X} (y(x) - \hat{y}(x))^2, \quad (2.4)$$

where  $\hat{y}$  is the estimation of the original image produced by the algorithm. It is obvious that this measure relies on the knowledge of the original (noise free) image, which is used to evaluate and compare different algorithms in a controlled simulation where noise is added to a set of images and applied to the algorithm. A similar measure to the MSE is the PSNR, which, assuming that the images are normalized, is given in dB by

$$PSNR = 10 \log_{10} \left( \frac{1}{|X|^{-1} \sum_{x \in X} (y(x) - \hat{y}(x))^2} \right) = 10 \log_{10} \left( \frac{1}{MSE} \right) \quad (2.5)$$

Being specified in dB, the PSNR leads to an easier comparison of results and performance of different algorithms, which is why it is the most commonly used measurement of denoising performance in the literature.

Another quantity used for evaluating the quality of images is the Structural Similarity (SSIM) index, proposed by Wang *et al.* in 2004 [5], which computation relies on trying to approximate the way images are perceived by vision. This is achieved by comparing local patterns of pixels intensities. The SSIM measurement is obtained by computing the following expression:

$$SSIM(y, \hat{y}) = \frac{(2\mu_y \mu_{\hat{y}} + c_1) \cdot (2\sigma_{y\hat{y}} + c_2)}{(\mu_y^2 + \mu_{\hat{y}}^2 + c_1) \cdot (\sigma_y^2 + \sigma_{\hat{y}}^2 + c_2)}, \quad (2.6)$$

where  $c_1$  and  $c_2$  are two constants that have the function of stabilizing the division and are proportional to the dynamic range of the pixel representation of the image, i.e., the number of pixels. For grayscale images of 8 bits, the dynamic range is 0 to 255, yielding  $c_1 = 6.5025$  and  $c_2 = 58.5225$ .

In the past few years, plenty of image denoising methods were studied and created, originating from various areas of research such as probability theory, statistics, linear and nonlinear filtering, and spectral analysis. One common aspect shared by all these methods is that they rely on implicit or explicit assumptions about the true image, in order to separate it properly from the noise. Two methods that have become the state of the art in denoising, and the most researched in the past decade, are transform domain denoising and compressed sensing based denoising.

## 2.2 Transform domain denoising

In natural images it is frequently observed the repetition of familiar structures and textures, which means that the image signal is not random, and similarity between regions of an image (local similarity) and between different regions (nonlocal similarity) occurs.

This encouraged the development of transforms that can approximate an image by linear combination of few basis elements, leading to a sparse representation of the image in the transform domain. Therefore, when an image is affected by Gaussian noise, it is expected that in the transform

domain this noise appears as low magnitude coefficients. By discarding this small coefficients, after applying the inverse transform, an approximation of the original image is obtained, which means that this method can be used to effectively denoise an image.

The quality of the denoised image depends mainly on the sparsity of the representation of the image in the transform domain. This sparsity depends on the transform used and on the original signal properties. Obviously, the original signal cannot be controlled, so the efficiency of the denoising relies on the transform chosen. There are various types of transforms that can be applied to images in order to obtain a sparse representation, but in this work there is a special interest in evaluating two of them: the discrete wavelet transform (DWT) and the discrete cosine transform (DCT). Regarding the process of discarding the small coefficients, which is usually called *shrinkage*, two different methods will be presented: hard thresholding and wiener filtering.

### 2.2.1 Wavelets

As defined in the celebrated book of I. Daubechies, *Ten Lectures on Wavelets*, the wavelet transform is a tool that cuts up data or functions into different frequency components, and then studies each component with a resolution matched to its scale. [6]

Wavelets are a mathematical tool with applications in many areas, for example, signal analysis, numerical analysis and physics. In the scope of this work, the intended application is signal analysis, specifically image processing. This way, a brief analysis on wavelets will be presented, inspired in the books of M. Jansen [7] and I. Daubechies [6].

There are various types of wavelet transforms, for example, continuous wavelet transform (CWT), discrete wavelet transform (DWT), lifting scheme, etc. An image produced by a digital camera is a discrete signal in both spatial directions, meaning that the appropriate transform to analyze an image is the discrete wavelet transform.

Wavelets are defined by a wavelet function  $\psi(x)$  called the *mother* wavelet and a scaling function  $\phi(x)$  called the *father* function, both in the time domain. Translating and dilating these functions allows the definition of *child* functions, forming a subspace on which the signal being transformed is decomposed. These sets of functions are given by

$$\psi_{m,n}(x) = 2^{-m/2} \psi(2^{-m}x - n) \quad (2.7)$$

$$\phi_{m,n}(x) = 2^{-m/2} \phi(2^{-m}x - n) \quad (2.8)$$

Any signal  $f(x)$  can then be reconstructed using the following formula

$$f(x) = \sum_{m,n} \langle f, \psi_{m,n} \rangle \psi_{m,n}(x), \quad (2.9)$$

where  $\langle f, \psi_{m,n} \rangle$  represents the inner product between the signal and a given *child* function, which is called a *wavelet coefficient*. For this reconstruction to be valid it is necessary that the set of

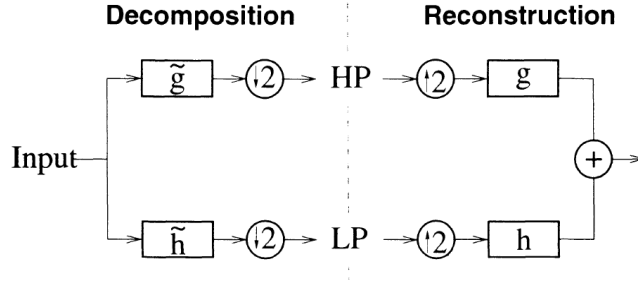


Figure 2.1: One step of a wavelet decomposition and reconstruction. Taken from [7]

functions  $\{\psi_{m,n} : m, n \in \mathbb{Z}\}$  form an orthonormal basis of  $L^2(\mathbb{R})$ . However, there is an exception to this rule, used for example, by the family of bi-orthogonal wavelets, which require two scaling and two wavelet base functions, resulting in the associated DWT to be invertible but not necessarily orthonormal.

In image processing, the set of scaling functions corresponds to the classical pixel representation of an image, while the wavelet basis "breaks" the image into a set of details at different locations and scales. This is said to be a more accurate way of representing an image, because it is closer to how we look at them: first we see general features and at a more careful inspection, we uncover the details. This is the main motivation for using the DWT to represent natural images.

In order to reveal details at different scales, the DWT takes advantage of something known as multiresolution analysis (MRA) which is defined as a nested set of function spaces. This is an algebraic concept that extends beyond the scope of this work, so it will not be explained. Interested readers should refer to [6] for a detailed mathematical treatment of MRA, or to [7] for a slightly lighter and concise approach.

From the study of MRA two main equations of wavelet theory arise, which are called the *dilation equation* and the *wavelet equation*. These equations are given, respectively, for the *father* and *mother* functions, as

$$\exists \mathbf{h} \in \ell_2(\mathbb{Z}) : \varphi(x) = \sqrt{2} \sum_{k \in \mathbb{Z}} h_k \varphi(2x - k) \quad (2.10)$$

$$\exists \mathbf{g} \in \ell_2(\mathbb{Z}) : \psi(x) = \sqrt{2} \sum_{k \in \mathbb{Z}} g_k \psi(2x - k) \quad (2.11)$$

In the case of a bi-orthogonal basis, there are duals  $\tilde{\mathbf{h}}$  and  $\tilde{\mathbf{g}}$ . Given these *filters*, each corresponding base function can be obtained by solving dilation and wavelet equations. An efficient realization of the DWT can then be implemented using filter banks with the filters  $\mathbf{h}$ ,  $\tilde{\mathbf{h}}$ ,  $\mathbf{g}$  and  $\tilde{\mathbf{g}}$ , which are used to decompose and reconstruct the signal, as can be seen in figure 2.1. Usually the wavelet filters are high pass (explaining the HP in the figure), meaning they enhance details, and the scaling filters are low pass, which means they have a smoothing effect. The filter bank DWT implementation has the advantage that MRA becomes simply a cascading of filter banks, using the low pass output as the new signal.

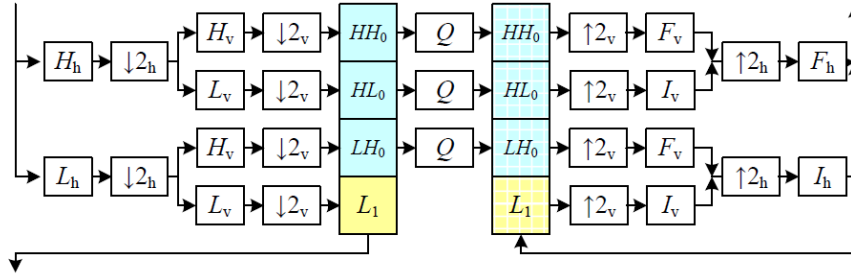


Figure 2.2: 2D separable implementation of DWT and its inverse. Taken from [8]

In order to analyze images, it is necessary to use a 2D DWT. This can be done by simply applying the DWT on all rows and then on all columns of the image, which results in four types of coefficients, or *sub-bands*. These sub-bands contain different image information according to the filtering applied, for example, the HH sub-band contains diagonal features of the image, because it corresponds to high pass filtering in both directions and the LH sub-band contains vertical structures, corresponding to low pass filtering the columns and high pass filtering the rows. The same logic applies to the remaining sub-bands HL and LL. A separable implementation of the 2D DWT using filter banks can be seen in figure 2.2.

Choosing the wavelet function and the corresponding scaling function (or equivalently the decomposition and reconstruction filters) can be a cumbersome task. To this end, the extensive studies on wavelets originated several families of wavelet functions that are used in all wavelet related applications. Examples of this families are the Haar-Wavelet, which is the first wavelet ever defined; the Daubechies wavelets,  $Db_p$  with  $p$  vanishing moments; and the bi-orthogonal wavelets  $BiorN_d.N_r$  with  $N_d$  vanishing moment in the decomposition and  $N_r$  in the reconstruction.

### 2.2.2 DCT

The discrete cosine transform (DCT) is a frequency domain transform, that decomposes a given signal into a sum of cosine functions with different frequencies [9]. By discarding small high frequency components, the DCT is used to do *lossy* compression of audio (MP3 standard) and images (JPEG standard). The DCT is very similar to the familiar DFT, with the obvious distinction being that it uses only cosine functions to decompose a signal, instead of both cosines and sines. There are eight types of DCTs, however, the most commonly used is the type II DCT, which is referred to as "the DCT" and is given by

$$X_k = \sum_{n=0}^{N-1} x_n \cos \left[ \frac{\pi}{N} \left( n + \frac{1}{2} \right) k \right] \quad k = 0, \dots, N-1 \quad (2.12)$$

This transform is equivalent to a DFT of  $4N$  real inputs where the even indexed elements are zero. The corresponding inverse transform is the type III DCT, which is referred to as IDCT, and

is given by

$$X_k = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos \left[ \frac{\pi}{N} \left( k + \frac{1}{2} \right) n \right] \quad k = 0, \dots, N-1 \quad (2.13)$$

The 2D DCT is simply a separable product of the DCT along each dimension of an image, i.e., the 1D DCT performed along the rows and then the columns of the image. Fast computation of the DCT can be done by a special fast cosine transform (FCT) which is simply an adaptation of the DFT counterpart, the FFT. In image processing, the 2D DCT is used to process blocks, commonly of size 8x8 (as in JPEG), which produces a matrix of 64 coefficients that represents how much of each basis functions the image contains. A visual representation of this basis functions for grayscale images can be seen in figure 2.3. The first coefficient represents the lowest frequency (DC), and traveling in a zig-zag pattern from the upper left corner to the lower right corner represents an increase in frequency of both dimensions.

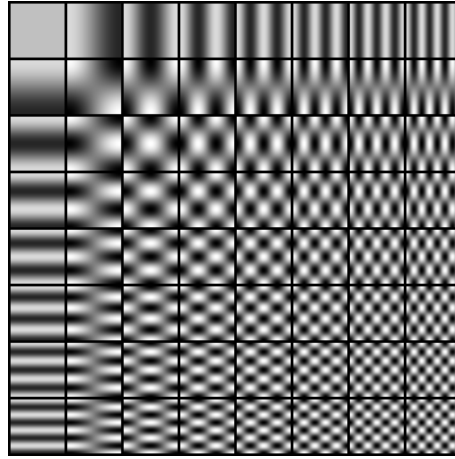


Figure 2.3: Basis functions for a 8x8 2D DCT

### 2.2.3 Hard thresholding

The hard thresholding operator is necessary for filtering the transform domain coefficients of an image, when a good estimation of the original image power spectrum is not available. Hard thresholding is the simplest shrinkage operator and it can be seen as a "keep or kill" procedure [7], because all values below a certain threshold  $\lambda$  are set to zero, while values above the threshold remain the same. This is given by the following expression

$$w_\lambda = \begin{cases} w & \text{if } |w| \geq \lambda \\ 0 & \text{if } |w| < \lambda \end{cases} \quad (2.14)$$

A similar shrinkage operator is soft thresholding, where coefficients above the threshold are shrunk in value by the amount of the threshold  $\lambda$ . This means that the soft thresholding function is continuous, which is an advantage to some algorithms where discontinuous operators cause problems. Plots of both operators can be seen in figure 2.4.

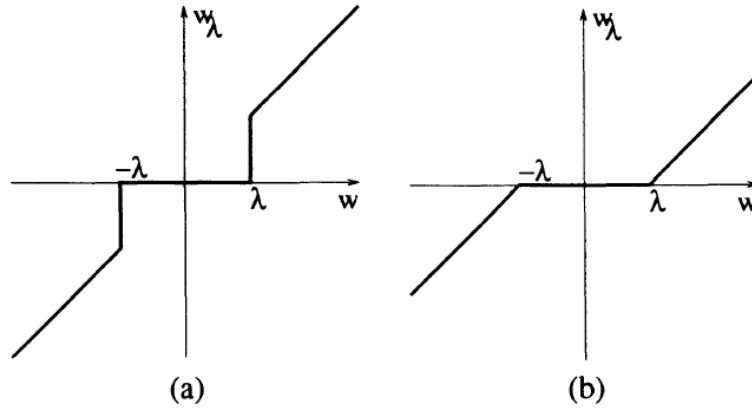


Figure 2.4: a) Hard thresholding operator. b) Soft thresholding operator. Adapted from [7]

### 2.2.4 Wiener filtering

Wiener filtering is a statistical based approach used to produce an estimate of a target random process, when a noisy measurement is available, as well as the signal and noise spectra. This way, the Wiener filter minimizes MSE between the estimated process and the desired one [10]. Deriving the expression of the Wiener filter requires a statistical approach to images. In this approach, an image is modeled as a random noise field whose expected magnitude at each frequency is given by [8]

$$E[S(w_x, w_y)^2] = P_s(w_x, w_y), \quad (2.15)$$

where  $P_s(w_x, w_y)$  is the power spectrum of the image, and  $E[\cdot]$  denotes the expected value. Using this expression and probabilistic arguments, for example, the Bayes' Rule, the 2D Fourier transform of the optimum Wiener filter needed to denoise an image, can be expressed as

$$W(w_x, w_y) = \frac{P_s(w_x, w_y)}{P_s(w_x, w_y) + \sigma_n^2}, \quad (2.16)$$

where  $\sigma_n^2$  is the power of the AWGN that corrupts the image. This way, the Wiener filter can be used for denoising an image, when a good estimation of its original power spectrum is available. For interested readers, the complete analysis to derive the Wiener filter expression can be found in [8].

## 2.3 Compressed Sensing based denoising

Sparse representation of signals has been a heavily researched subject in the past decade. The brief introduction of this subject presented here is based on M. Elad's book, *Sparse and Redundant Representations* [11]. The main concept consists in using an overcomplete dictionary matrix  $D \in \mathbb{R}^{n \times K}$ , containing  $K$  prototype signal atoms, to represent a signal  $y \in \mathbb{R}^n$  as a sparse linear combination of these atoms. Using this dictionary, an exact representation for the signal is given



as  $y = Dx$ . However, in many applications, an exact representation is hard to obtain, and thus, an approximate solution is  $y \approx Dx$ , subject to  $\|y - Dx\|_p \leq \varepsilon$ . Measuring the approximation error is usually done using the  $\ell^p$  norms with  $p=1, 2$ , and  $\infty$ .

When using overcomplete dictionaries, i.e.,  $n < K$ , an infinite number of solutions are available for the representation problem. This way, it is necessary to impose constraints on the solution, and since high sparsity is desired, the solution with fewest nonzero coefficients is the most appealing. This sparsest approximate representation is the solution of

$$(P_{0,\varepsilon}) \quad \min_x \|x\|_0 \quad \text{subject to} \quad \|y - Dx\|_2 \leq \varepsilon, \quad (2.17)$$

where  $\|\cdot\|_0$  is the  $\ell^0$  norm, counting the nonzero entries of a vector.

As was already discussed, images can be sparsely represented using for example a wavelet transform, leading to effective denoising algorithms using wavelets that exploit overcomplete representations. In the case of wavelets, the dictionary is completely defined when choosing the wavelet and scaling functions and their *childs*. However, in the case of compressed sensing (CS), the goal is to achieve the sparsest representation of an image in the spatial domain, i.e., the dictionary has to contain the building blocks (*atoms*) necessary to represent any image. This requires solving 2.17, which is proven to be an NP-hard problem [12]. Therefore, approximate solutions to the problem are considered instead, with many approximation algorithms being proposed in the last few years. These algorithms are in general greedy and examples are the matching pursuit and orthogonal matching pursuit. There is also the basis pursuit, which relaxes the  $\ell^0$  norm in 2.17 to an  $\ell^1$  norm, convexifying the problem. Details about these algorithms extend beyond the scope of this work, hence, detailed descriptions can be found, respectively in [13], [14] and [15].

In order to develop an efficient denoising algorithm using a CS prior, the choice of dictionary is of particular importance. For this choice there are two options: a set of pre-specified functions, as is the case when using wavelets, DCT or other transforms; design the dictionary by adapting it to fit a given set of signal examples, as is the case when using training based methods, such as the PCA and K-SVD.

### 2.3.1 PCA

Principle component analysis (PCA) is a statistical procedure that "extracts" the principal components of a set of observations with correlated variables using an orthogonal transformation. This principal components are linearly uncorrelated, and thus PCA can effectively decorrelate a signal. It was first formulated by Pearson in [16], and was further developed by Hotelling in [17].

As described in the original work of Pearson, the PCA can be seen as the line or plane that closest fits a system of points in an  $n$ -dimensional space. Each component of the PCA "explains" the variance in the data that the previous component is unable to fit to, i.e., the first component is a linear combination of original variables weighted so that it represents the maximum variance in the data, the second accounts for the variance not represented in the first, and so on.

The PCA can be used to develop a dictionary for the dataset it is applied to, so it can be used to obtain such a dictionary that leads to a sparse representation of an image, which can then be used to denoise said image. Usually the PCA is done by a singular value decomposition (SVD), which means it is a highly computationally intensive method.

### 2.3.2 K-SVD

The K-SVD is a dictionary training algorithm, that utilizes effective sparse coding and a Gauss-Seidel like accelerated dictionary update method. It is an extension of the popular k-means algorithm, which is used to train data sets in clustering problems. The full operation of this algorithm is complex and so, it will not be presented here. The algorithm itself is presented in figure 2.5 and was taken from the original paper of M. Aharon, M. Elad and A. Bruckstein, *K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation* [18]. For those interested in the algorithm, please refer to this article for a detailed explanation.

Task: Find the best dictionary to represent the data samples  $\{\mathbf{y}_i\}_{i=1}^N$  as sparse compositions, by solving

$$\min_{\mathbf{D}, \mathbf{X}} \{ \|\mathbf{Y} - \mathbf{DX}\|_F^2 \} \quad \text{subject to} \quad \forall i, \|\mathbf{x}_i\|_0 \leq T_0.$$

Initialization : Set the dictionary matrix  $\mathbf{D}^{(0)} \in \mathbb{R}^{n \times K}$  with  $\ell^2$  normalized columns. Set  $J = 1$ .

Repeat until convergence (stopping rule):

- *Sparse Coding Stage*: Use any pursuit algorithm to compute the representation vectors  $\mathbf{x}_i$  for each example  $\mathbf{y}_i$ , by approximating the solution of

$$i = 1, 2, \dots, N, \quad \min_{\mathbf{x}_i} \{ \|\mathbf{y}_i - \mathbf{D}\mathbf{x}_i\|_2^2 \} \quad \text{subject to} \quad \|\mathbf{x}_i\|_0 \leq T_0.$$

- *Codebook Update Stage*: For each column  $k = 1, 2, \dots, K$  in  $\mathbf{D}^{(J-1)}$ , update it by
  - Define the group of examples that use this atom,  $\omega_k = \{i \mid 1 \leq i \leq N, \mathbf{x}_T^k(i) \neq 0\}$ .
  - Compute the overall representation error matrix,  $\mathbf{E}_k$ , by

$$\mathbf{E}_k = \mathbf{Y} - \sum_{j \neq k} \mathbf{d}_j \mathbf{x}_T^j.$$

- Restrict  $\mathbf{E}_k$  by choosing only the columns corresponding to  $\omega_k$ , and obtain  $\mathbf{E}_k^R$ .
  - Apply SVD decomposition  $\mathbf{E}_k^R = \mathbf{U}\mathbf{\Delta}\mathbf{V}^T$ . Choose the updated dictionary column  $\tilde{\mathbf{d}}_k$  to be the first column of  $\mathbf{U}$ . Update the coefficient vector  $\mathbf{x}_{\tilde{R}}^k$  to be the first column of  $\mathbf{V}$  multiplied by  $\mathbf{\Delta}(1, 1)$ .
- Set  $J = J + 1$ .

Figure 2.5: The K-SVD algorithm. Taken from [18]

## 2.4 Related Work

### 2.4.1 Denoising Algorithms

Donoho and Johnstone were the first to explore the wavelet based denoising and the development of the *shrinkage* algorithm. In their works, they applied wavelet theory to signals in general, and to different concepts of signal processing and mathematics, such as minimax estimation [19], spatial adaptation [20], and smooth functions [21]. In Donoho's work [22] the shrinkage of wavelet coefficients by applying soft thresholding is used to denoise signals. Other works using wavelet shrinkage found in the literature are [7], [23] and [24]. The wavelet transform can be used to form redundant representations of blocks in an image, which results in a shift invariant property, as described in [25].

Regular wavelet transforms are not effective when representing certain image features, such as smooth textures. This results in blurring when reconstructing or denoising an image from its wavelet coefficients. Taking this into account, new methods to denoise an image were developed, using new multiscale and directional (anisotropic) transforms, such as the wedgelet [26], contourlet [27], curvelet [28], bandelet [29] and steerable wavelet [30].

Dabov *et al.* [1] proposed in 2007 a novel method for image denoising based on collaborative filtering in transform domain. This algorithm is called block matching and 3D filtering (BM3D) and comprises three major steps. First, a set of similar 2D image fragments (i.e. blocks) is grouped into 3D data arrays that are referred to as groups. This step is referred to as block matching. Second, a 3D transform is applied to the groups, resulting in a sparse representation, that is filtered in the transform domain, and after inversion of the transform, produces the noise-free predicted blocks. This step is referred to as collaborative filtering. Finally, the predicted noise-free blocks are returned to their original positions to form the recovered image. BM3D relies on the effectiveness of the block matching and collaborative filtering to produce good denoising results.

Chen and Wu [31] proposed a modification to the BM3D algorithm that achieves better PSNR and visual results for images contaminated with high levels of noise. The method is called bounded BM3D and it differs from the original BM3D in the block matching and collaborative filtering of the second stage of the algorithm, i.e., the basic estimate is computed in the same way to generate a pilot signal for the second stage. The difference starts in the beginning of the second step, where the basic estimate is partitioned into regions (image segmentation) and the boundaries between those regions are detected. This allows for a bounded search in the block matching, i.e., only blocks in the same image region of the reference block are candidates for grouping. However, a block can be contained in two or more regions (coherent segments), and in this case, partial block matching is applied, where blocks that belong to several regions are partitioned using binary masks to separate the segments. This poses a great advantage when compared to BM3D: the partitioning avoids dealing with edges, hence avoiding problems when representing them in 3D transform domain. Nevertheless, in order to do the wiener filtering of these non square segments, shape adaptive DCT has to be applied as the 2D transform, which is more computationally intensive. As proposed, this method achieves better PSNR, with gains of 0.23 - 1.33 dB compared

to BM3D, and better visual results, specially in edges and textures. It is worth to remark that a similar approach was also taken by Dabov *et al.* in [32] with the development of shape adaptive BM3D. Continuing this research, in [33], Dabov *et al.* presented an improvement to their previous SA-BM3D algorithm, called BM3D-SAPCA, where SAPCA stands for shape adaptive principle component analysis. The PCA is applied instead of wavelets or DCT for the 2D transform, and it achieves better denoising results than the previous BM3D methods.

Elad and Aharon [34] devised an image denoising method based on sparse representations over learned dictionaries. In the transform domain methods, there is also an implicit dictionary, given by the atoms defined by the 2D DCT or a wavelet transform. However, this dictionaries, in spite of being overcomplete, cannot represent efficiently all the information in an image. This way, Elad and Aharon researched the possibility of using K-SVD to train a global dictionary, using a database of noise free images. They found that for the task of image denoising, the choice of images to train on is crucial, and while a good general dictionary that fits all images well can be found, in order to achieve high denoising performance (comparable to BM3D and other methods), a more complex model is necessary using several dictionaries switched by content. Therefore, in their work, they adopted a different direction, by training the dictionary directly on the noisy image. At a first sight, this seems to have no impact in the overall quality of the algorithm. However, Elad and Aharon found a way to combine the denoising and training steps into the same framework. The algorithm starts with the DCT dictionary and then performs  $J$  iterations of sparse coding followed by dictionary training, minimizing the representation error in each iteration. The results using this adaptive dictionary were far more promising than those using a general dictionary, which makes this a very attractive and robust method for image denoising.

Dong *et al.* [35] presented a novel sparse representation model for image restoration tasks, called centralized sparse representation (CSR). They introduce the concept of sparse coding noise (SCN), which is simply defined as the difference between the coding vector of the noisy image and the coding vector of the original noise free image. However, in most cases, the original image is not available, and so, its coding vector is not known. Nevertheless, a good estimator for this vector is its mean value, which in turn can be approximated by the mean value of the coding vector of the noisy image, by assuming that the SCN has nearly zero mean (which is confirmed empirically in their work). This model is named centralized sparse representation because it enforces the coding vector to approach its distribution center, i.e., the mean value. In order to compute the mean value of the coding vector, groups of similar patches are created, so that their sparse codes can be averaged, for each patch in the image. This way, the CSR model can be written as a minimization problem, unifying the local sparsity of each patch and the nonlocal similarity induced sparsity (from similar patches) into a variational formulation (two variable parameters control the weight of each type of sparsity). This model can be iterated until convergence: by setting the initial estimate for the mean value of the coding vector to zero, an initial estimate is obtained, from which the groups are formed and the new mean value is computed and used in the following iteration. Being a sparse representation based model, CSR relies on a dictionary, that in this work is obtained by PCA. The CSR model converges to the desired sparse code when the joint sparse coding and non-

local clustering falls into a local minimum. Results of this method for image denoising are very similar to those achieved by BM3D. In 2013, Dong *et al.* [36] proposed a modification of their previous work on CSR, by adding the idea of nonlocality to the sparse model, allowing to remove the local sparsity term. This way, the nonlocally centralized sparse representation (NCSR) model achieves better denoising performance with the same computational effort.

One of the most recent works on image denoising found in the literature is by Zhong *et al.* [37]. In their work a combination between the BM3D algorithm and the nonlocal centralization prior exploited in [35] allows for very competitive results, particularly for images corrupted with high levels of noise. The main idea is to replace the 1D transform with a shrinkage model based on the nonlocal centralization prior. This allows the combination of the efficiency and effectiveness of wavelet or DCT transforms (when compared with the iterated approach of the CSR model) and the nonlocal and local sparsity unification provided by the CSR model. Moreover, the CSR model is expanded by allowing the norms used in the shrinkage function to vary. This way, three different shrinkage functions are proposed: one taking advantage of the  $\ell_1$  norm for the local sparsity and the  $\ell_2$  norm for the nonlocal, other with a double  $\ell_1$  norm, and a final one using a nonlocal prior to remove the local sparsity term (as proposed by [36]). In concluding remarks, essentially, this work proposes an efficient combination of the transform domain approach for sparse representation of images and group matching (from BM3D), with more options for advanced shrinkage functions (based on CSR and NCSR) to replace the 1D transform and the hard thresholding or wiener filtering used in BM3D.

From the methods presented in this section, the algorithm chosen for this work was the original BM3D, because of its high efficiency and very good denoising performance, with average computational burden (when compared with dictionary based methods for example). This way, in chapter 3 the BM3D algorithm will be analyzed in full extent.

### 2.4.2 Hardware Implementations

Memik *et al.* [38],[39] were among the first to propose an FPGA implementation of an image restoration algorithm. The algorithm used is a simple neighborhood iterative restoration algorithm, that for each pixel in the image applies a convolution with a kernel that uses the eight neighbor pixels to restore the actual pixel value. The algorithm is run for the number of iterations necessary so that the residual is under a specified threshold, and in each iteration, all the image pixels are processed. However, this is the software approach of the algorithm, that is very slow, which motivates the hardware implementation in order to achieve a faster solution. The setup of the hardware consists on a memory to store the image, a pixel processor that executes the algorithm, and the channel of communication between both. Since the communication in this channel is the bottleneck of the system, there is no gain in executing the algorithm as it is done in software, by sending nine pixel values at a time and writing back to memory. Instead, parallelism of the algorithm is exploited, and an array of pixel processors allows for a parallel computation of several pixel values. Nevertheless, space in an FPGA is usually limited, and the number of pixel processors usually is lower than the number of pixels in an image (for usual image resolutions of

256x256 or 512x512). This way, the image is segmented into regions, and each region is loaded into the FPGA, processed, and stored back into memory. To avoid border effects and block artifacts, the segments of the image are allowed to overlap, and the overlapping restored portions are discarded. The hardware implementation described achieves up to a ten times speedup in the runtime of the image restoration algorithm.

Saldaña and Arias-Estrada [40] developed a reconfigurable systolic-based architecture for low level image processing tasks on an FPGA. The architecture is tuned to allow the efficient convolution of a filter kernel with an image, in a windowing based approach. The main module is a 2D customizable systolic array, with the size of the window to be used, of processing elements (PEs). The image pixels are read from an external memory and are placed on internal memories implemented as Block RAM's, using a Router to manage the data transfers. The 2D systolic array is built by interconnecting several PEs, which are activated every clock cycle, following a pipeline scheme. The PEs are specially designed in order to support the operations involved in most window based operators, and their architecture consists of an ALU, a shift register and an accumulator. Each processing element executes three operations in every clock cycle: computation of the pixel value to be passed to the next cycle, accumulation of the output register calculated at the previous cycle with the new value at the output of the ALU and loading the new mask coefficient and transmission of the previous to the next PE. With the pipelined systolic architecture, a throughput of one window output per clock cycle is achieved. Finally, the architecture was synthesized in a Xilinx VirtexE FPGA, with a 7x7 systolic array (and corresponding sized window operation), resulting in 49 PEs and an overall area occupancy of 37%. The clock frequency achieved was 66MHz, resulting in 200 fps for 640x480 resolution gray level images.

Joshi *et al.* [41] presented an FPGA implementation of a wavelet based image denoising algorithm. This implementation consists of four chained main modules: the lifting scheme based wavelet module, the windowing module, the denoising module and the inverse wavelet module. The first module computes the 2D DWT on the rows and columns of the image, by applying a lifting scheme implementation of the Daubechies 9/7 biorthogonal wavelet. Each wavelet module computes 4 rows and 4 columns at a time thanks to 4 parallel row and column modules. The second module is the windowing module, which contains various shift registers in order to implement neighborhood observation and to analyze the wavelet coefficients for the different image sub-bands. Then, the denoising module computes the denoised coefficient, using different arithmetic operations implemented as a 10 input squaring module, followed by a subtraction module, a summation unit and a comparator. Finally, the inverse wavelet module applies the inverse DWT to the denoised coefficients and stores the denoised image in the image memory. The results are presented in terms of frames per second (fps), i.e., how many images can be denoised per second, and the values are 83 fps for a 256x256 size image and 28 fps for 512x512.

Brylski and Strzelecki [42] proposed the implementation of a parallel image processor. The main idea of the implementation consists of a matrix of active nodes, which correspond to the image pixels, connected with each other by weights that depend on the neighboring pixels. The implementation is intended to perform segmentation operations in binary images. The design con-



sists of a microcontroller connected with the PC by Ethernet and with an FPGA by SPI. The FPGA contains a control unit and the active matrix of  $N \times N$  nodes. The central unit block implements the SPI connection and the clock manager that controls the matrix of nodes. The node block contains several input/output signals in order to communicate with neighboring nodes, and its structure is fairly complex. The main block is the C driver, that performs the node algorithm and controls the work of other node unit. The complete module was synthesized in a Xilinx FPGA with  $17 \times 17$  matrix elements and uses approximately 85% of the slices in the FPGA.

In the work of Di Carlo *et al.* [43], an adaptive image denoising IP core (AIDI) is presented, intended for real time applications. The algorithm implemented is based on an adaptive gaussian filter, which adapts its variance pixel by pixel according to estimates of the gaussian noise corrupting the image and the local variance of the expected noise free image. This way, the AIDI core contains three main modules: the noise variance estimator (NVE), the local variance estimator (LVE) and the adaptive gaussian filter. First, the image pixels are sent in parallel to the NVE and an external memory through a 32 bit interface, and the NVE computes the estimation of the Gaussian noise affecting the image. Then, when this step is complete, the image is loaded to the LVE, that computes the local variance associated with each pixel and outputs one of this values per clock cycle, thanks to its pipelined architecture. Finally, the outputs of the LVE and NVE are fed into the adaptive gaussian filter, which computes the optimal filter variance and then filters the image applying Gaussian smoothing. The AIDI core was synthesized in a Xilinx Virtex 6 FPGA, occupying close to 20% of the LUTs and 1.7% of the block RAMs, and achieving 68 fps for images with  $1024 \times 1024$  pixels.

In Gabiger-Rose *et al.* [44], image denoising is done in real time by a bilateral filter implemented in a fully synchronized architecture on an FPGA. The implementation presented has three main advantages, enabling real time processing and effective utilization of resources: data is sorted into equal groups assigned to separate pipelines, the clock frequency is raised accordingly to the data flow and no external image buffer is necessary. Each functional unit of the bilateral filter consists of a register matrix, a photometric filter and a geometric filter. The register matrix is composed by several cascaded registers and multiplexers, allowing the parallel calculation of 24 weights used by the following filter stages, and its output consists of six groups which are fed to the photometric filter stage at four times the pixel clock. The photometric filter consists of six identical pipelines, each processing a group of pixels at every clock cycle, and the output consists of the weighted pixels sorted into six groups, the current center pixel being computed and the photometric coefficients for each group. The final stage is the geometric filter, that is implemented as a separable 1D filter for the vertical and horizontal directions, and its output consists of the filtered kernel result and a normalization factor. The normalization of the results is done by a simple division stage at the end of the data path. The algorithm was synthesized in a Xilinx Virtex 5 FPGA, achieving 52 fps for a  $1024 \times 1024$  resolution image, and occupying 14% of slices, 23% of block RAMs and 60% of DSP slices. In terms of the denoising results, an approximate 0.2 dB loss was verified when comparing with the MATLAB implementation of the algorithm.

In all the works presented, some similarity in the hardware implementation of image denoising

algorithms can be recognized. These similarities consist of the main ideas that are used when designing such systems. These main ideas are: the use of multiple parallel instances of a pixel processor; usage of large external memories to hold the complete image, and smaller internal memories to hold image sections currently being processed; transferring the image data in bursts of many pixels (usually full sections), in order to reduce the bottlenecks usually found in data transfer protocols; the use of pipelined architectures whenever possible, in order to minimize the execution time. All these main ideas are employed in this work, in order to develop the hardware architecture for the BM3D denoising algorithm in the most efficient way. The developed architecture is fully described in chapter 4, where it will be possible to recognize these ideas in different parts of the implementation. Furthermore, due to the complexity of the BM3D algorithm, the usage of parallel processors and efficient memory modules are the main advantages of the hardware implementation designed in this work.



## Chapter 3

# BM3D Denoising Algorithm

In this chapter, a more detailed explanation of the BM3D denoising algorithm will be provided. Then, some modifications to the algorithm will be presented as well as a MATLAB implementation and the respective results.

### 3.1 BM3D Algorithm

As previously referred in section 2.4.1, the BM3D algorithm is a novel image denoising algorithm proposed in 2007 by Dabov *et al.* [1]. Since this is the algorithm used in this work and only a brief description was given, a more detailed explanation follows. Recalling the previous explanation, it is already known that the BM3D algorithm consists of three steps: the block matching, the collaborative filtering and the final aggregation.

In the block matching step, blocks that are similar are grouped together in a 3D array, which enhances the sparsity in the transform domain. This is done by matching: the process of finding a block similar to a given reference one. The similarity of two blocks is inversely proportional to their distance, i.e., the smaller the distance between two blocks, the more similar they are. In order to form a group, a bound (threshold) on this distance is set, and if an  $\ell_2$  norm is used, this threshold is the radius of the circle containing the blocks of the group and the reference block is the center of this circle. The block matching is performed for every reference block in an image, using a sliding window approach, producing a group for every block, meaning that these groups are not necessarily disjoint, which provides an overcompleteness property. Usually, similar blocks are only found in the same regions of an image, which motivates the restriction of searching candidate blocks in a fixed neighborhood around the currently processed block. This restriction makes the block matching step faster, which significantly affects the total running time of the algorithm. Another restriction that provides a speedup of the algorithm is limiting the maximum number of blocks in a group, i.e., only a specified number of blocks with the lower distances to the reference block are kept in the group.

The collaborative filtering comprises three steps. First a 3D (or separable 2D and 1D) transform is applied to a group. Then, the transform coefficients are shrunk, by using hard thresholding

or wiener filtering in order to attenuate the noise. Finally, inverting the transform produces the estimates for all grouped fragments. The transform takes advantage of correlation in each grouped fragment (a peculiarity of natural images) and correlation between fragments of the same group to produce a sparse representation of the blocks of the group, making the shrinkage very effective in attenuating the noise. In order to reduce the complexity of applying the transforms, which results in faster execution, 2D and 1D separable transforms are desired. This way, the transforms used can be wavelet decompositions such as Daubechies or biorthogonal wavelets, the Haar wavelet or the DCT. The authors compared several transforms and higher PSNR values were obtained for the DCT transform for the 2D transform, and the Haar wavelet for the 1D transform.

After the estimates for each block are available, they are returned to their original positions, and because of the overcompleteness of the block matching, there can be more than one block containing the same image pixel, i.e., overlapping. This way, in the process of aggregation, the blocks are summed by a weighted average, using a kaiser window for reducing border effects because of the block processing.

The BM3D algorithm is comprised of two "runs" of the aforementioned described steps. First, the noisy image is processed using the block matching, collaborative filtering and aggregation, using hard thresholding in the shrinkage of the transform coefficients. This produces a basic estimate for the original noise free image. Then, using this basic estimate as input, block matching is applied, being more accurate because the noise is already significantly attenuated. The same groups formed in this basic estimate are formed in the original image. Then, the collaborative filtering and aggregation is applied, but wiener filtering is used instead of hard thresholding for the shrinkage. The wiener filter uses the basic estimate energy spectrum as the true energy spectrum of the image, and allows for a more efficient filtering than hard thresholding, improving the final image quality.

### 3.2 Implications for Hardware Design

With a full explanation of the BM3D algorithm provided, one can see that it is quite complex and computationally demanding. This means that some bottlenecks need to be identified and possibly modified for hardware feasibility.

Looking closely at the BM3D algorithm, the first possible bottleneck is the calculation of the distance between two blocks in the block matching step. This calculation consists of the  $\ell_2$  norm of the difference of the two blocks being matched, meaning that, for example for an 8x8 block, 64 multiplications are necessary, which is foreseen to be a problem in terms of hardware resources. The second implication for the hardware implementation is set by the maximum number of blocks in a group,  $N_{max}$ , which requires a sorting operation, that chooses the  $N_{max}$  blocks with the smaller distances to the reference block.

The collaborative filtering step of the BM3D algorithm is where most computational effort is required, when applying the 2D and 1D transforms to the blocks. The sequence of computations of this step, i.e., 2D transform, followed by 1D transform in the 3rd dimension, followed by hard

thresholding or wiener filtering and inversion of the transforms, and the fact that this sequence is applied to every group, evidences a possibility of a fully pipelined architecture for the collaborative filtering step.

The final step of the BM3D algorithm is the aggregation of the denoised blocks to form the final image. This task does not imply a large computational effort, as only additions and one last division per image pixel is necessary. However, due to the high overcompleteness provided by the algorithm, many denoised blocks overlap, which means that the best approach to build the final image is to create an image buffer with the same pixel size as the original image (and another one for the weights of the groups). This means that for large image sizes (3, 5 and 8 megapixels for example) the memory necessary for these buffers is very large, meaning that an hardware implementation is very hard to accomplish.

### 3.2.1 Modifications

In order to tackle the bottlenecks and implications derived by the BM3D algorithm, the following modifications and decisions were made:

- In the calculation of the distance between two blocks, the  $\ell_2$  norm is replaced by the simpler, multiplier-less  $\ell_1$  norm. With this change, the absolute value of the distance will be different, while the relative value is expected to remain similar, leading to very similar matching, hence effectively producing the same grouping for each reference block.
- With the distance calculation planned to be pipelined, a new distance is produced at each clock cycle, and can be sorted in a process similar to the well known insertion sort, meaning that the threshold for the distances can be removed, because in terms of clock cycles there is no difference if the distance is bigger or smaller than the threshold.
- In the collaborative filtering step, when implementing the algorithm in software, the blocks of a group are available all at the same time, which allows for a full decomposition in the 3rd dimension (same pixel position of each block in the group) using  $\log_2(N_{max})$  levels of the haar wavelet. However, as a pipelined implementation of this step is desired, parallelization is required in order to have at least two blocks being processed at each clock, so that one level of the haar wavelet decomposition can be applied. For each additional level of decomposition, the number of parallelization doubles, which in turn halves the execution time of this step and enhances denoising because signal sparsity increases with each level of wavelet decomposition. Nevertheless, this approach has the disadvantage of doubling the hardware resources needed for this pipeline between each level of parallelization.

With these modifications to the BM3D algorithm, denoising performance is expected to drop, mainly due to the use of lower levels of decomposition of the haar wavelet, leading to lower signal sparsity, which in turn leads to an ineffective denoising by hard thresholding. On the other hand, by taking advantage of a specialized hardware implementation for the modified algorithm, the execution time and power consumption are expected to decrease.

### 3.3 MATLAB Implementation

In order to set a benchmark for direct comparison with the results of the hardware implementation, the BM3D algorithm was implemented in MATLAB, including the described modifications.

#### 3.3.1 Initialization

The first step of the implementation is the definition of the parameters that control the BM3D algorithm, which influence the quality of denoising and the run-time of the algorithm. These parameters are:

- $N$ : the block size of image patches to be processed. The value is set to  $N = 8$ .
- $N_{step}$ : the step used in the processing of reference blocks. This value is chosen to be bigger than one in order to speedup the algorithm by a factor of approximately  $N_{step}^2$ . The value is set to  $N_{step} = 4$ .
- $N_{max}$ : this is the maximum number of blocks in each group, also decreasing the runtime of the algorithm, as already mentioned. This value is set to  $N_{max} = 16$ .
- $N_S$ : the size of the neighborhood centered in the reference block from where candidate matching blocks are searched. This value is set to  $N_S = 39$ .
- $\lambda_{3D}$ : the hard thresholding constant for the first stage, which is set to  $\lambda_{3D} = 2.7$ .
- $\beta$ : the parameter for the Kaiser Window used to reduce border effects. This value is set to  $\beta = 2.0$ .

With all the parameters set, a noisy test image is created. For this, the same dataset of images is used as in the original work. The image is read into MATLAB and a random Gaussian noise with power (standard deviation)  $\sigma$  is added. Next, the transform matrix for the 2D DCT is hard coded, so it can be used to apply the transform to each block by a simple matrix multiplication, and the kaiser window used in the aggregation is also computed and stored.

With the noisy image created and the 2D DCT transform matrix computed, the next step is the initialization of the algorithm. The first step is the pre computation of the 2D transform on every possible image block, with each block of coefficients being stored on a cell array called *tBlocks*. Follows the initialization of two buffers used in the aggregation step at the end of the algorithm: the image buffer, which will store the filtered blocks estimates, and the weights buffer, which stores the weights for the aggregation process.

### 3.3.2 Noise power estimation

The next step is the estimation of the noise standard deviation, which is done by applying the Median Absolute Deviation (MAD) estimator. The MAD estimator for a certain parameter of interest of a statistical distribution is defined as [45] :

$$MAD = \hat{\sigma} = K \text{median}_i |x_i - \text{median}_j x_j|, \quad (3.1)$$

where K is a constant scale factor, which depends on the distribution and the parameter of interest. In the case of a Gaussian distribution and for estimation of standard deviation, K has the value of 1.4826. The MAD estimator is applied on the set composed by the high frequency coefficients of each distinct 8x8 block on the image, which are almost pure noise. However, an upward bias can affect this estimation due to the presence of true signal in the high frequency coefficients [20]. Noise power estimation is necessary in the collaborative filtering step, to define the threshold in hard thresholding, which is given by:

$$\lambda = \hat{\sigma} \lambda_{3D}, \quad (3.2)$$

and for the group weight calculation, given by:

$$w = \frac{1}{N_{har} \hat{\sigma}^2}, \quad (3.3)$$

where  $N_{har}$  is the number of retained (nonzero) coefficients after hard thresholding.

### 3.3.3 Execution

Next, the execution of the algorithm begins. Starting at the top left pixel of the image and moving  $N_{step}$  pixels along the horizontal direction, the BM3D algorithm is applied to each block defined by the current pixel position. For each block, group matching is applied, searching all blocks in the neighborhood and storing the positions and distance to the reference block of those that match. Then, the positions are ordered by ascending distance, and the first  $N_{max}$  blocks are extracted from the  $tBlocks$  array, creating a group. The next step is the collaborative filtering, starting with applying the 1D transform, i.e., one level of the haar wavelet decomposition, producing the 3D transform domain representation of the group. Follows the actual filtering, which is performed by hard thresholding using the previous computed threshold  $\lambda$  (equation 3.2). Along with the filtering, the weight for the group in the final aggregation step is computed by equation 3.3. Then, the 1D and 2D transforms are inverted, resulting in the estimated noise free blocks of the group. Finally, the denoised blocks are added in the image buffer in the correct position, after multiplication by the Kaiser Window and the weight. At the same time, the weight is added to the corresponding position in the weights buffer. When all the image is processed, the basic estimate is obtained by simple element wise division of the image buffer by the weights buffer.

### 3.3.4 Results

With the MATLAB implementation complete, its performance is tested on two different images (Cameraman and Lena) and compared against the original BM3D.

Table 3.1: PSNR (dB) results for 256x256 Cameraman Image

$\sigma$	Noisy Image	Original BM3D	Modified BM3D
5	34.16	38.20	34.79
10	28.13	33.94	31.25
15	24.62	31.67	29.42
20	22.12	30.24	28.18
25	20.18	29.14	27.16
30	18.60	28.25	26.28
35	17.30	27.41	25.53
40	16.10	26.33	24.83
45	15.08	26.07	24.21
50	14.16	25.55	23.65

Table 3.1 shows the PSNR results in dB for the image Cameraman for both original and modified BM3D. As expected, it can be seen that the modified BM3D achieves worse denoising results, mainly due to the loss of sparsity by only using one level of the haar wavelet decomposition. This results amount for an average absolute PSNR loss of 2.13 dB (relative loss of 7.11%). The diminished performance of the modified BM3D can be confirmed by the lower visual quality of the result image, which can be seen in figure 3.1.

Table 3.2: PSNR (dB) results for 512x512 Lena Image

$\sigma$	Noisy Image	Original BM3D	Modified BM3D
5	34.15	38.63	36.84
10	28.14	35.65	34.30
15	24.62	33.83	32.58
20	22.12	32.47	31.32
25	20.19	31.37	30.32
30	18.60	30.46	29.52
35	17.26	29.65	28.84
40	16.10	28.72	28.24
45	15.08	28.45	27.72
50	14.16	27.92	27.24

Table 3.2 shows the PSNR results in dB for the image Lena for both original and modified BM3D. Once again, the modified BM3D achieves worse denoising results. However, this results amount for a smaller average absolute PSNR loss of only 1 dB (relative loss of 3.08%). This smaller difference can be explained by the fact that the Lena image is bigger (512x512 versus

256x256 for Cameraman), meaning that there are more similar blocks in the image, which enhances the sparsity of the transform domain coefficients of each group, consequently improving filtering. The result images can be seen in figure 3.2.



Figure 3.1: a) Original Cameraman image with size 256x256; b) Noisy image with  $\sigma = 25$  (PSNR=20.18 dB); c) Basic estimate image obtained by modified BM3D (PSNR=27.16 dB); d) Basic estimate obtained by original BM3D (PSNR=29.14 dB)



Figure 3.2: a) Original Lena image with size 512x512; b) Noisy image with  $\sigma = 25$  (PSNR=20.19 dB); c) Basic estimate image obtained by modified BM3D (PSNR=30.32 dB); d) Basic estimate obtained by original BM3D (PSNR=31.37 dB)



## Chapter 4

# Hardware Implementation

In this chapter, the hardware implementation of the modified BM3D algorithm will be described in detail. This will be done in a top to bottom approach, starting from the system architecture and high level description, followed by a complete characterization of each individual block and its operation.

### 4.1 System Architecture

This section presents a simple analysis of each block contained in the system architecture, and its operation. An overview of the system architecture is shown in figure 4.1. The architecture can be divided in four main elements: the array of matching processors; the denoising pipeline; the various memory modules, with different functions; and the multitude of control modules, which are not shown in the figure for simplification purposes.

The array of matching processors is responsible for the block matching step of the BM3D algorithm. Each of the 16 matching processors performs block matching for a different reference block in parallel. These reference blocks have overlapping neighborhoods (two consecutive reference blocks have a shift of 4 pixels), meaning that the array of processors operates on a "big" neighborhood, which is a concatenation of all neighborhoods for each individual reference block, and overlapping data is shared by successive processors. When the array of processors completes its operation, each processor outputs the positions of the 16 blocks that have matched the reference block (the first position is the reference block itself), and, since there are 16 processors, these positions are concatenated in a 16x16 bus.

With the matching step done, and the group positions available, follows the collaborative filtering step. This step is performed in the denoising pipeline, which operates as follows: the positions outputted by the matching processors are stored in a fifo; then, the blocks corresponding to two positions are extracted from an internal memory containing the "big" neighborhood, at a rate of 1 row per clock cycle; then, the collaborative filtering is performed on these blocks by an efficient pipelined architecture that implements the 2D DCT and Haar transforms, hard thresholding and

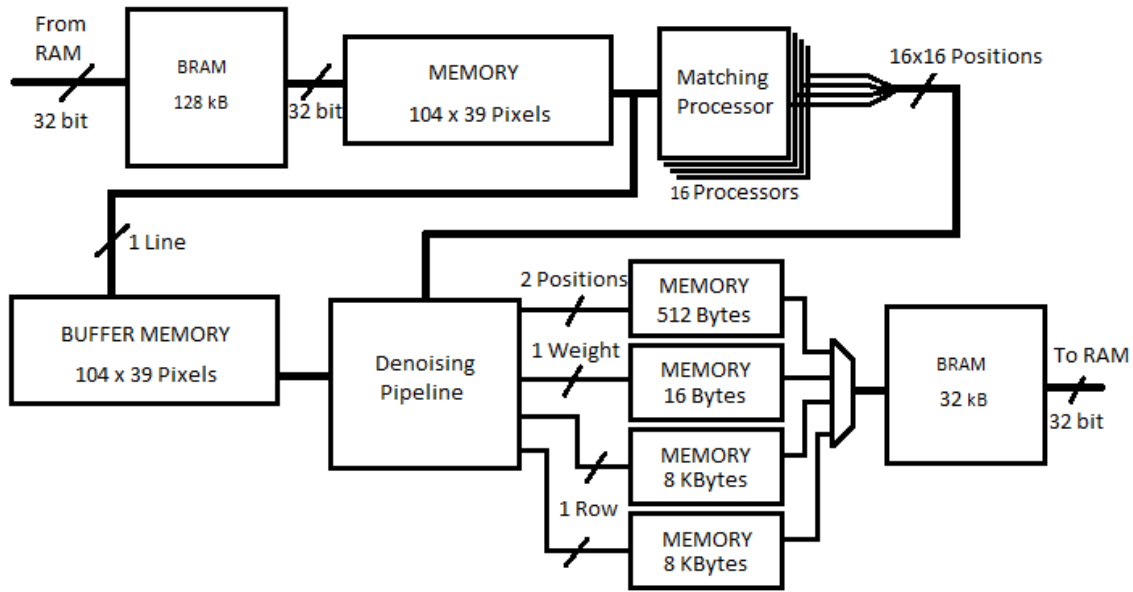


Figure 4.1: BM3D System Architecture Block Diagram

the inversion of said transforms; finally, the denoised blocks, their positions and the group weight are outputted and stored in memory blocks.

As can be seen in figure 4.1, there are 8 memory modules in the system architecture:

- Input Block RAM (BRAM), size 128 KBytes, 32 bit wide: holds the image data in line order, i.e. consecutive memory positions contain 4 consecutive pixels of a line of the image.
- Next neighborhood memory, size 104x39 pixels (4056 bytes), input 32 bit wide, output 104 pixels wide (832 bit): contains the next neighborhood to be processed by the matching processors.
- Buffer neighborhood memory, size 104x39 pixels (4056 bytes), 104 pixels wide (832 bit): holds the neighborhood currently being processed and feeds it to the denoising pipeline when the matching step finishes.
- Positions output memory, size 512 Bytes, 32 bit wide: each memory position contains two relative positions of its corresponding image block.
- Weights output memory, size 16 Bytes, input 8 bit wide, output 32 bit wide: each memory position holds the weight of a group.
- Two data output memories, size 8 KBytes each, input 64 bit wide, output 32 bit wide: each memory position contains a row of a denoised image block.
- Output BRAM, size 32 KBytes, 32 bit wide: holds the output data of the BM3D algorithm: the denoised blocks, their positions in the original image and the group weights.

All the blocks described above need very strict control in order to operate properly. This is achieved by the following 7 control modules:

- Next memory control module: responsible for controlling data transfers between the input BRAM and the next neighborhood memory.
- Load denoise memory control module: responsible for loading the contents of the buffer memory to the denoising pipeline.
- Positions memory control module: responsible for storing positions data in the positions memory.
- Weights memory control module: responsible for storing weight data in the weights memory.
- Data memory control module: responsible for storing denoised blocks data in the parallel data output memories.
- Copy memory control module: responsible for controlling data transfers between the 4 output memories and the output BRAM.
- Master control module: as indicated by the name, this module is responsible for the control of the whole system, i.e. it controls all the modules described above, plus the internal control modules present in the array of processors and the denoising pipeline.

With the system architecture explained, the following sections contain a meticulous description of the operation of each module in the system.

## 4.2 The Matching Processor

This section elaborates on the operation and design of the matching processor. The schematic for the processor is presented in figure 4.2. As can be seen, the processor consists of three modules: a memory; an arithmetic unit that calculates the  $\ell_1$  distance between two blocks; and a sorter unit that orders and stores the 16 smaller distances and respective block positions.

The matching processor needs two control modules in order to operate correctly: a memory control module and a processor control module. However, as already mentioned, the BM3D system uses an array of 16 matching processors operating in parallel, meaning that only one of each control modules is necessary in order to control all processors at the same time.

### 4.2.1 Memory Module

The memory module is responsible for feeding the 8x8 blocks to the  $\ell_1$  norm unit. The memory is structured in a specific way such that it is possible to provide each consecutive 8x8 block. This is achieved by using five, 64 bit wide memory LUTs that have 39 positions and are byte addressable.

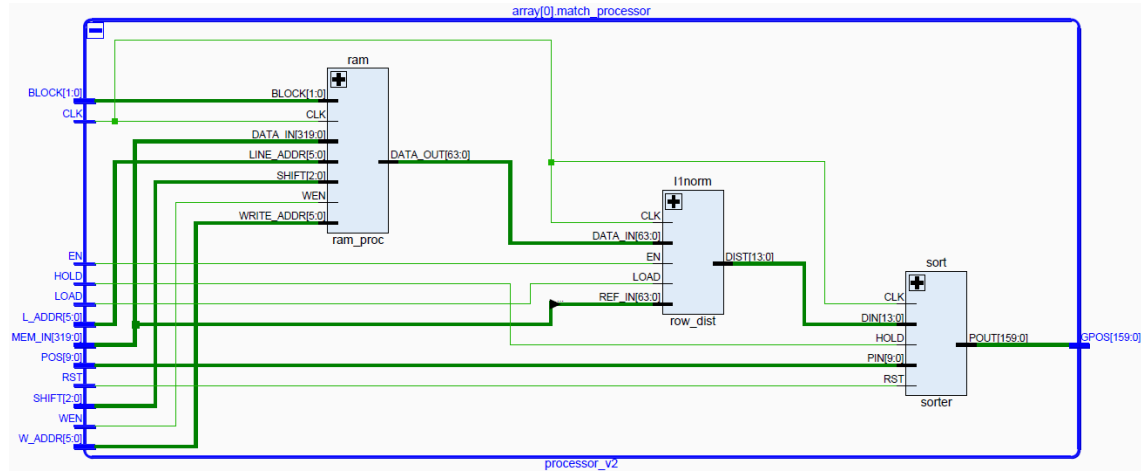


Figure 4.2: Matching Processor Schematic

The width is set to 64 bits because this is exactly the size of one block row (8 pixels =  $8 \times 8$  bits). Hence, five memories in parallel contain 40 pixels, which means that 8 bits (1 pixel) are never used, since the neighborhood size is  $39 \times 39$ . However, this structure is necessary and very effective for a sliding window approach as will be explained next.

Firstly, the number of bits for each coordinate needs to be defined. Within a  $39 \times 39$  neighborhood,  $32 \times 32$  distinct  $8 \times 8$  blocks can be found. Therefore, 5 bits are enough to code each coordinate of the block, i.e., each position has 10 bits: the  $y$  coordinate in the 5 most significant bits (MSBs) and the  $x$  coordinate in the remaining 5 bits. Now, let us consider for example that the current block being processed is at  $x = 13$  and  $y = 25$ . The  $8 \times 8$  block defined by this coordinates spans from  $x = 13$  to 20 and  $y = 25$  to 32. The  $y$  coordinate simply chooses the memory position, 25 in this case, of the first row to be extracted, meaning that 8 clock cycles are necessary to provide a full  $8 \times 8$  block. This, as it will be seen later, is not a problem, since the  $\ell_1$  norm unit accepts block rows as input. The  $x$  coordinate however, has a different treatment: the 2 MSBs define a signal called *block* and the remainder 3 bits define a signal called *shift*. The value 13 in binary is given, in 5 bits, by 01101, meaning that *block* = 1 and *shift* = 5. The *block* signal chooses the pair of memories to be read, i.e., in this case, the value 1 indicates that the desired block is read from memories 2 and 3 (memories are numbered 1 to 5). Then, the *shift* signal indicates how many bytes (pixels) of each memory should be selected, i.e., in this case, the value 5 indicates the 3 least significant bytes from memory 2 and the 5 most significant bytes from memory 3.

Another advantage of this memory structure is that a full neighborhood row can be loaded into memory at each clock cycle, because the five internal memories can be loaded in parallel. This saves some clock cycles when initializing the memory, and processing can even start after the first row that contains the reference block is loaded. This will be further explained in the control modules subsection (4.2.4).

### 4.2.2 $\ell_1$ norm Unit

In the block matching step, the distance that needs to be calculated is given by:

$$d = \sum_{i=1}^8 \sum_{j=1}^8 |X_{REF}^{i,j} - X^{i,j}|, \quad (4.1)$$

where  $X_{REF}^{i,j}$  is line number  $i$  and column number  $j$  of the reference block, and, accordingly,  $X^{i,j}$  is the same line and column from the block being matched.

Let us consider two consecutive 8x8 blocks in the vertical direction of the image,  $X_1$  and  $X_2$ . The distances for these blocks are given, respectively, by:

$$d_1 = \sum_{i=1}^8 \sum_{j=1}^8 |X_{REF}^{i,j} - X_1^{i,j}| \quad \text{and} \quad d_2 = \sum_{i=1}^8 \sum_{j=1}^8 |X_{REF}^{i,j} - X_2^{i,j}| \quad (4.2)$$

However, as the blocks are consecutive, i.e., they only differ in one row from one another, we can write the following relation:

$$X_2 = \bigcup_{i=1}^8 X_2^i = \bigcup_{i=1}^7 X_1^{i+1} \cup X_2^8 \quad (4.3)$$

This way, combining equations 4.2 and 4.3, we get the following expression for  $d_2$ :

$$d_2 = \sum_{i=1}^7 \sum_{j=1}^8 |X_{REF}^{i,j} - X_1^{i+1,j}| + \sum_{j=1}^8 |X_{REF}^{8,j} - X_2^{8,j}| \quad (4.4)$$

With this result, it is possible to see, that to calculate the distance for each new 8x8 block, 7 rows from the previous block can be reused. This is of major importance for the design of the  $\ell_1$  norm unit, because it means that a pipelined architecture can be developed, taking advantage of seven shift registers in order to store the old rows of previous blocks. With this architecture, an output of a new block distance per clock can be achieved, after a lag of eight clocks for the first 8x8 block for each neighborhood column. As grayscale images are used, the maximum value possible for the distance is  $64 \times 255 < 2^6 \times 2^8 = 2^{14}$ , meaning that the  $\ell_1$  norm unit is designed with 14 bit operations, in order to avoid overflows.

Taking into account the design considerations described, the  $\ell_1$  norm unit operates as follows:

1. The reference block is loaded line by line into 8 registers connected in series (shift register), meaning that in the end, the first line is at the last register.
2. The same is done for the first block of the first neighborhood column, with the difference that there are only 7 shift registers to store the 7 previous lines, while the current line is directly at the input of the module.
3. Follows an array of subtractions and a modulus operation that process each pair of row registers, producing the  $\ell_1$  norm of each pixel of each row.

4. Then, an adder tree with intermediate pipeline stages sums each pixel distances to produce the final distance for the block being processed. This tree starts with 32 adders (since there are 64 pixels) and 32 registers and then at each stage the number of blocks is halved.

The register transfer level (RTL) schematic of the  $\ell_1$  norm unit can be seen in figure 4.3. With this pipelined architecture, the  $\ell_1$  norm unit introduces a delay of 5 clock cycles, plus the initial 8 clock cycles for loading the first block of each column. Hence, there is a 13 clock cycle delay between the start of processing and the cycle when the first correct distance is at the input of the sorter unit.

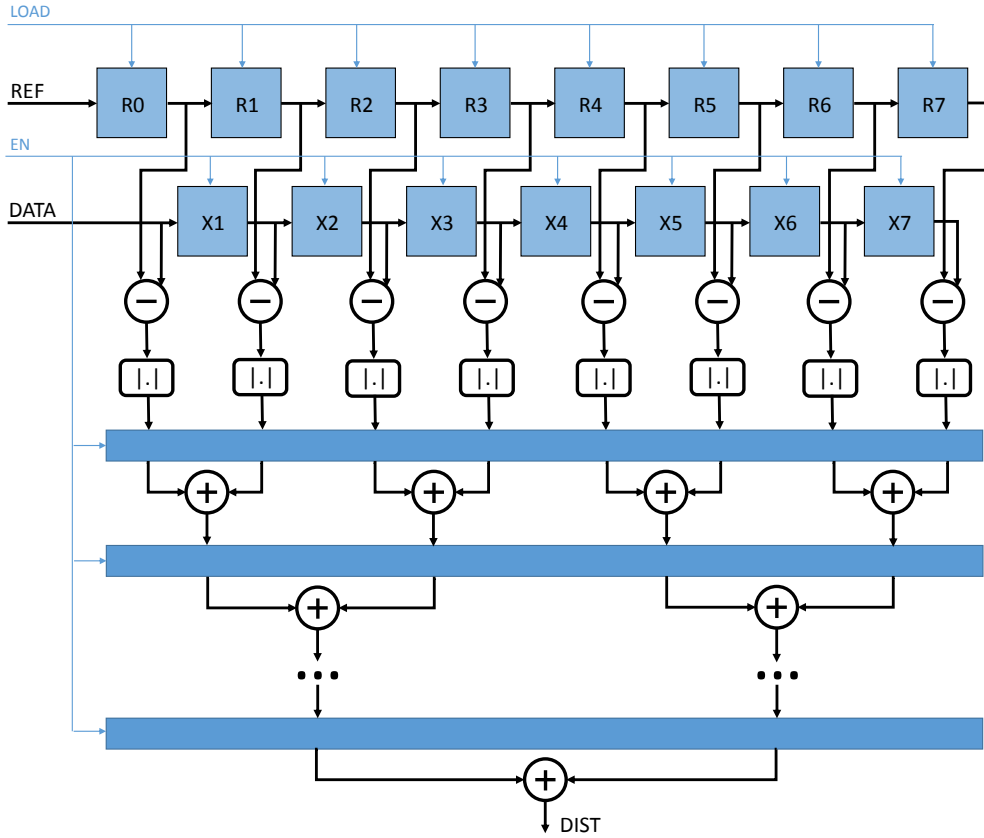


Figure 4.3: RTL schematic of the  $\ell_1$  norm unit

### 4.2.3 Sorter Unit

The sorter unit is responsible for choosing the 16 smaller distances, from the 1024 ( $39 \times 39$  neighborhood has  $32 \times 32$  distinct  $8 \times 8$  blocks) that are produced by the  $\ell_1$  norm unit. The design of this unit was adapted from the work of Hussain *et. al* [46]. It consists of 16 sorter cells in series and each cell contains two registers, one to store the distance and another to store the block position associated to that distance, and a comparator, that checks if the input distance of the cell is smaller than the currently stored one. If it is indeed smaller, the input distance is stored in the cell, and the previously stored one is passed onto the next sorter cell, otherwise, if the input distance is larger, it

is directly passed onto the next sorter cell. This way, at each clock cycle all the sorter cells operate in this manner and a new distance is inputted to the first cell of the chain, effectively sorting the 16 smaller distances. Hence, after all distances have been computed, there is a delay of 16 clock cycles, at the worst case. This happens when the last distance available is the 16th smaller and needs to be passed by all the sorter cells until it reaches the last and its finally stored. The RTL schematic of one single sorter cell can be seen in figure 4.4.

At the end of this 16 clock cycles, the first cell in the chain contains the smaller distance, the second contains the second smaller distance, and so forth. Each cell has a parallel output connected to the positions register, and at the end of the sorting, the output of each cell is gathered into a bus, that now contains the 16 positions for all the blocks that matched the reference one, effectively forming a group. In order to allow a correct sorting, each cell's register is initialized with the maximum possible value (14 bits at 1), so that at initial stages any distance compared is always smaller. Due to the  $\ell_1$  norm unit's architecture, the values at the input of the sorter are not always correct. Hence, an holding mechanism was implemented, consisting of a simple multiplexer controlled by an input denominated *hold*, that when is set to one, inputs the maximum possible value to the sorter, so that no register is changed.

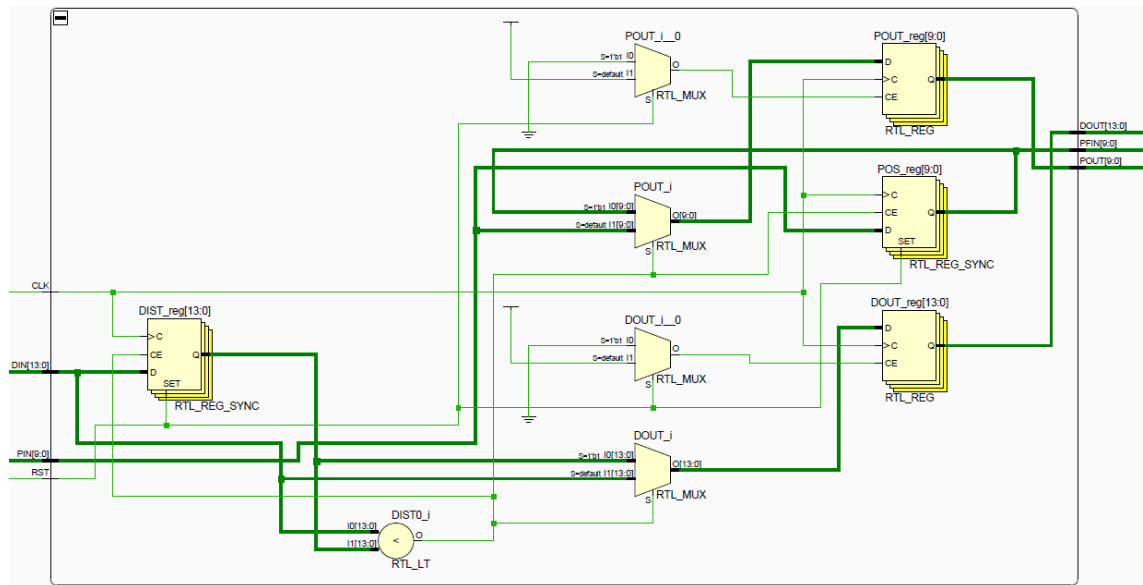


Figure 4.4: RTL schematic of one sorter cell

#### 4.2.4 Control Modules

As previously mentioned, the array of matching processors is controlled by two control modules: the memory control module and the processor control module.

The memory control module is modeled as a finite state machine (FSM) with four states and its state diagram can be seen on figure 4.5. It operates as follows:

1. Waiting in IDLE state for the signal *start*, sent by the top level master control, that signals the beginning of the block matching step.
2. When the signal *start* is detected with the value 1, the state machine moves to state LOAD\_MEM, that loads each processor's memory, by incrementing the write address from 0 to 38.
3. When the address reaches 14, the state changes to LOAD\_REF, where the memory keeps being loaded and the *load* control signal is activated, enabling the reference block's registers in the  $\ell_1$  norm unit, which are then loaded with the correct block row (this is done by a simple bit selection from the data bus).
4. When the address reaches 22 the state changes to SEND\_INIT, a single clock cycle state that sends the *init* signal to the processor control module. After one clock cycle the state goes back to LOAD\_MEM.
5. When the address reaches 38, the state goes back to IDLE and each processor's memory is fully loaded with the respective neighborhood.

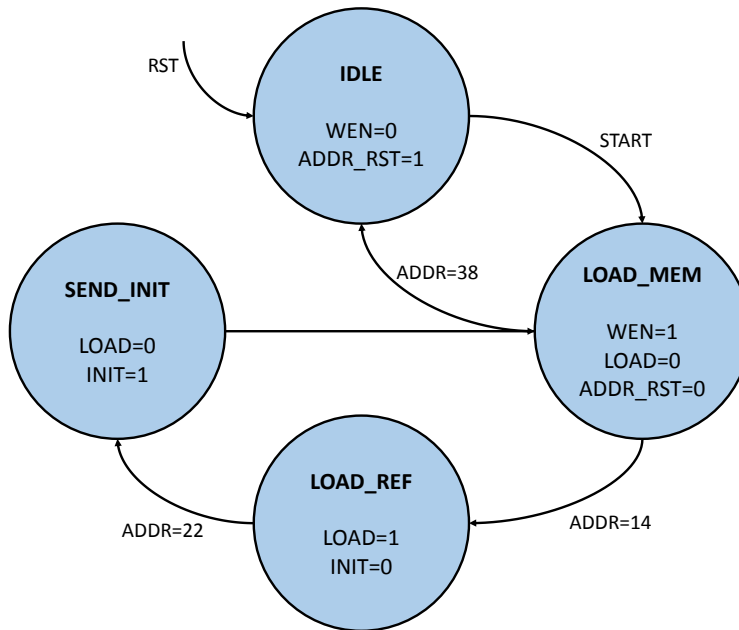


Figure 4.5: State diagram of the matching processor's memory control module

The processor control module is also modeled as an FSM with six states and its state diagram can be seen on figure 4.6. It operates as follows:

1. Waiting in IDLE state for the signal *init*, sent by the memory control module, which signals that the reference block is loaded into the  $\ell_1$  norm unit and thus the processor can start.
2. When the signal *init* is detected with the value 1, the state machine moves to state START\_PROC, that enables the shift registers in the  $\ell_1$  norm unit, so that block rows can be loaded, but keeps



the sorter on hold by setting  $hold = 1$ . Simultaneously, the read address is incremented from 0 to 38 at each clock cycle.

3. When the read address reaches 13, the first correct distance is available, thus the state changes to **PROCESS**,  $hold$  is set to 0 and the  $y$  coordinate starts being incremented.
4. When the read address reaches 38, the state is changed to **NEXT\_COL**, where the variable controlling the column being processed ( $col$ ) is incremented and the read address is reset. On the next clock cycle the state changes to **START\_NEXT**;
5. While the read address is smaller than 5 the state is kept on **START\_NEXT**. This is due to the fact that while a new column is already being inputted to the  $\ell_1$  norm unit, due to its pipelined architecture with 5 stages, there are still 5 distances of the previous column that need to be sorted. When the read address reaches 5, the state changes depending on the value of  $col$ : if it is bigger than 0 the state changes to **NEXT\_POS**, else it changes to **IDLE**, i.e. the processing is over.
6. In the **NEXT\_POS** state, the  $x$  coordinate is incremented and, one clock cycle after, the state returns to **START\_PROC**.

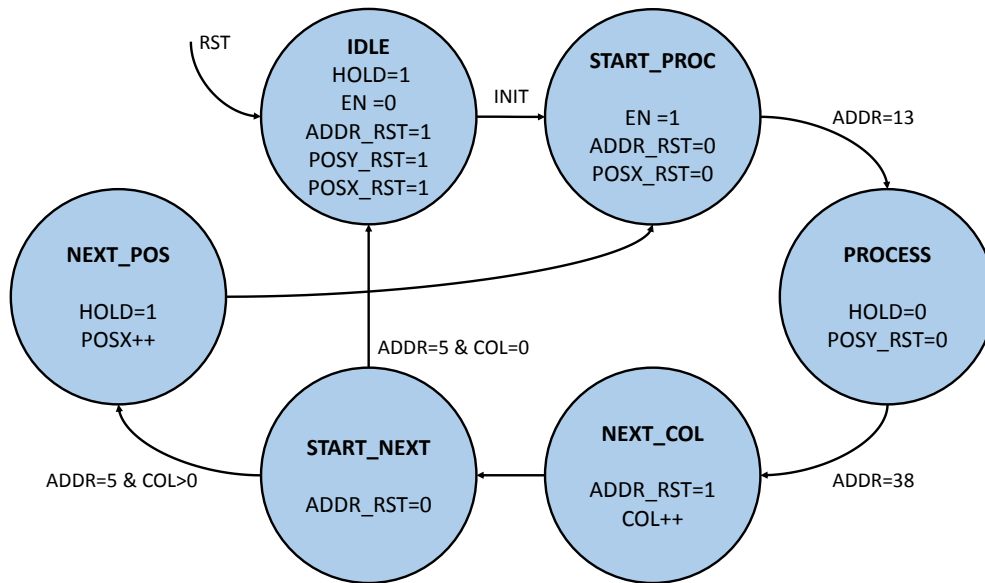


Figure 4.6: State diagram of the matching processor's control module

The processor control module is also responsible for the generation of two signals that are inputs of other control modules in the architecture. These signals are: *load\_den\_mem*, that indicates the buffer memory control module to load the neighborhood to the denoising pipeline; and *start\_den*, that indicates to the denoising pipeline that the groups are formed and it can start its operation.

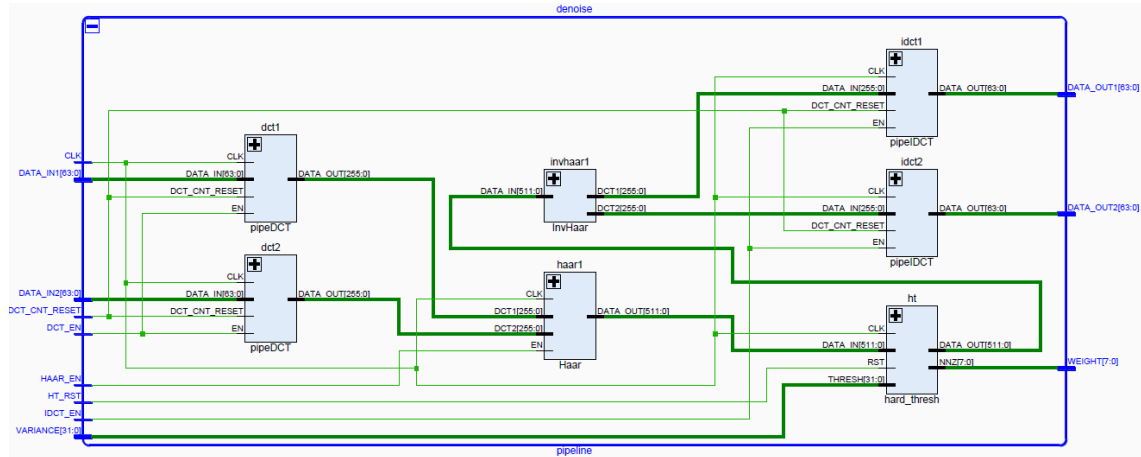


Figure 4.7: Denoising Pipeline Schematic

### 4.3 The Denoising Pipeline

This section elaborates on the operation and design of the denoising pipeline. The schematic for the pipeline is presented in figure 4.7. As can be seen, the pipeline consists of seven modules: two 2D DCT transforms, one haar wavelet transform, one hard thresholding module, one inverse haar module and two 2D IDCT transforms. These modules are all connected in series, forming a continuous pipeline where two 8x8 image blocks are parallelly processed. Furthermore, to support the operation of the pipeline, there are two memory modules: one for the positions of all groups and another for the neighborhood data; a position decoder and a control module.

The pipeline data path is 32 bits wide, using a fixed point implementation with 16 bits for the integer part and another 16 for the decimal part. In order to avoid overflow, at the beginning of the pipeline, input data is placed at the decimal point and then shifted 7 bits to the right (division by  $2^7$ ). The data flows in the pipeline as follows:

1. The positions register outputs two consecutive group positions for eight clock cycles.
2. The positions are converted from each group's local neighborhood to the "big" neighborhood in the position decoder and are inputted into the data memory module.
3. The memory module outputs a row of each of the desired 8x8 blocks at each clock cycle.
4. Each row is inputted to the respective 2D DCT module, that performs the transform and outputs a block column at each clock cycle.
5. The haar transform is performed on the two block columns and the coefficients are inputted into the hard thresholding module.
6. This module discards the coefficients below the threshold and then inputs them into the inverse haar transform.

7. This module inverts the haar transform and then the two 2D IDCT modules invert the 2D transform.
8. The end of the pipeline is reached with two block rows being outputted at each clock cycle.

### 4.3.1 Multiplier-less DCT and IDCT

The 2D DCT and IDCT transform modules have the same basic design: one 1D DCT module transforms the first dimension (rows in the case of the DCT and columns in the case of the IDCT), then a skewed array of registers transposes the data (rows to columns or columns to rows) and then another 1D DCT performs the transform on the second dimension. Furthermore, the 2D IDCT module also contains a rounding module at the output.

First, let's address the design of the 1D DCT and IDCT blocks. These blocks are crucial to the performance of the pipeline, because they account for the largest arithmetic effort of the implementation. In order to optimize the DCT and IDCT transforms, various algorithms have been proposed. One of the most efficient algorithms was proposed by Loeffler *et al.* [47] in 1989, using only 11 multiplications and 29 additions. This number of multiplications was proven to be the lowest achievable by a theoretical bound. However, in order to obtain high clock frequencies and lower hardware resource usage, multiplier-less DCT and IDCT transforms are desired. To achieve this, the multiplications in the flow graph algorithm (FGA) are replaced with additions and shifts. This is exactly what Chen *et al.* [48] and Aakif *et al.* [49] did in their works. In the latter, there is a direct comparison with the work of Chen *et al.* with better results being achieved. This way, to achieve better performance, the design of the DCT and IDCT modules is based on the work of Aakif *et al.*.

All the multiplications in the DCT and IDCT have a constant operand, i.e. the signal from the flow graph is multiplied by a constant value. These values are dual in the DCT and IDCT, i.e., the same constants are used, but in symmetric positions in the FGA, meaning that the same shifts and additions modules can be implemented for replacing the multiplications in both the DCT and IDCT designs. This way, only the design of the IDCT will be analyzed, since all the design choices are mimicked in the DCT.

The FGA of the IDCT implementation proposed in [49] can be seen in figure 4.8. The 11 multiplication blocks have 10 different coefficients ( $\sqrt{2}$  is used two times), meaning that 10 blocks need to be implemented using shifts and additions to replace the multiplication. In order to achieve this, a fixed point representation of the coefficients is needed. Aakif *et al.* chose an unsigned 12 bit precision representation, as recommended by the IEEE 1180-1990 standard [50]. Using this fixed point representation it is possible to implement any coefficient needed for the IDCT with a maximum delay of 3 additions/subtractions. Let us analyze for example the  $a$  coefficient, with the value  $\sqrt{2}$ , which is given by 1.011010100001 in the 12 bit precision representation. Let  $x$  be the input of the multiplier and  $z$  the output, i.e.,  $z = \sqrt{2} x$ . Looking at the binary value of  $a$ , one can

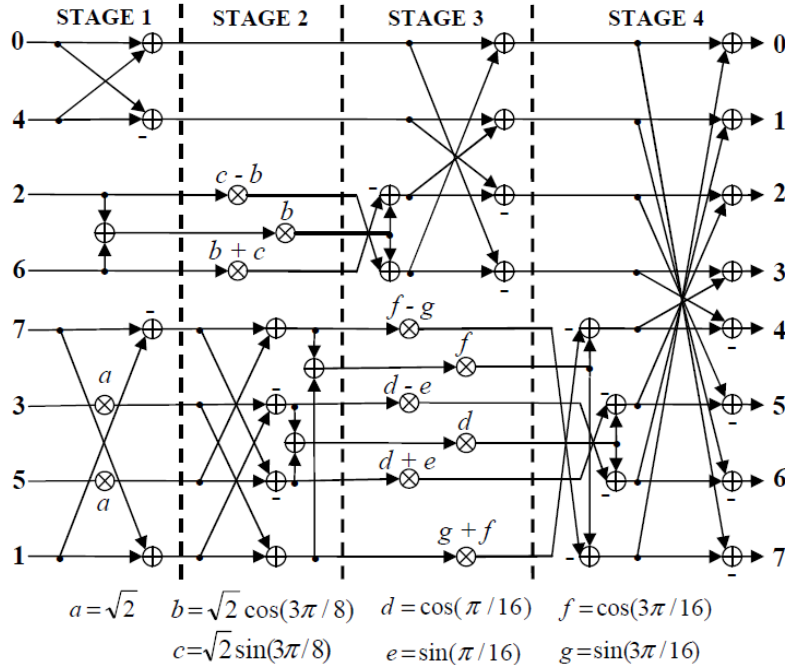


Figure 4.8: IDCT Flow graph algorithm. Taken from [49]

see that  $z$  can be given by:

$$z = x + x \gg 2 + x \gg 3 + x \gg 5 + x \gg 7 + x \gg 12 \quad (4.5)$$

However, in their work, Aakif *et al.* tried to minimize the number of additions and shifts used. Hence, they tried to find common expressions (similar to a factorization operation) that could simplify the multiplication, and use parallel additions. Therefore, they defined the following implementation for the  $a$  coefficient:

$$\begin{aligned}
 y_1 &= x \gg 12 + x \gg 7; \\
 y_2 &= x + x \gg 2; \\
 y_3 &= y_1 + y_2; \\
 z &= y_3 + y_2 \gg 3;
 \end{aligned}$$

This implementation uses 4 shifts and 4 additions, and only a delay of 3 additions, because  $y_1$  and  $y_2$  are computed in parallel, followed by  $y_3$  and finally  $z$ .

This analysis is repeated for all the coefficients necessary for the IDCT implementation, but this will not be covered in this work. With the multiplier-less modules designed, the FGA of the IDCT can be developed. This is done by following the design shown in figure 4.8, applying the operations to each input and adding registers at the end of each stage to achieve a pipelined design. This means that the 1D IDCT module is responsible for a delay of 3 clock cycles in the denoising

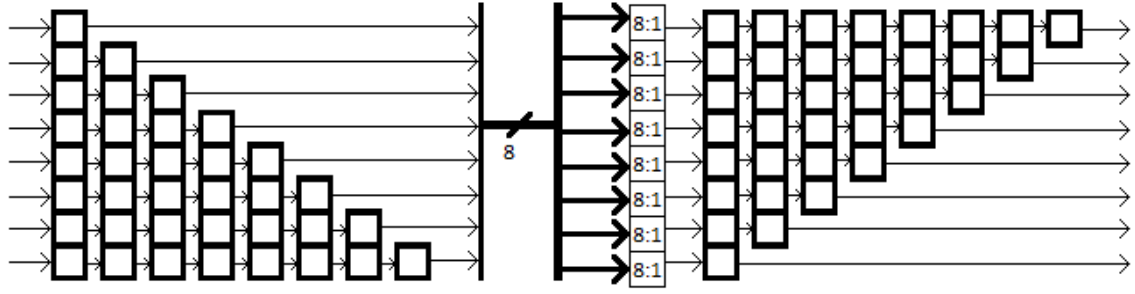


Figure 4.9: Transposer structure. Adapted from [51]

pipeline.

With the 1D DCT and IDCT modules analyzed, let's now look to the skewed array of registers. This module is placed between the first and second 1D DCT and IDCT modules to transpose the data without breaking the pipeline. The design of the transposer was proposed by Aggoun and Jalloh in 2003 [51]. For a block size of  $8 \times 8$ , it consists of two arrays of skewed registers, with 36 registers each, and eight 8to1 multiplexers. The structure of the transposer can be seen in figure 4.9.

Let  $R_1^i$  and  $R_2^i$  be the row with  $i$  registers of, respectively, the first and second arrays of registers and  $M_j$ , the multiplexer of row  $j$ . It can be seen that for each row  $i$ , the multiplexer  $M_i$  is connected to the row of registers  $R_2^{9-i}$ . All the multiplexers are controlled by the same 3 bit signal, meaning that for the data transposition to be correct, the order of the inputs in each multiplexer has to be different. For the first multiplexer, the order of inputs is the same as the first array of registers, i.e., input  $i$  is connected to  $R_1^i$ . However, for the second multiplexer, each input is shifted in a circular manner, i.e., input  $i$ , with  $i$  from 2 to 8, is connected to  $R_1^{i-1}$  and input 1 is connected to  $R_1^8$ . This shift is repeated for each following multiplexer. This way, a full cycle of operation of the transposer is given by:

1. The first 8 coefficients of the 1D DCT/IDCT are loaded into the first position of the  $R_1^i$  arrays.
2. The second 8 coefficients from the first transform are loaded into the first position of the  $R_1^i$  arrays, leading to the shift of the first coefficients and the multiplexer select signal has the value 1. Hence, the data from  $R_1^1$  is shifted into  $R_2^8$ , while all the other data from the first coefficients remains in  $R_1$ .
3. A similar shift occurs as in the previous step and the select signal is incremented. Hence, the data from  $R_1^1$  is shifted into  $R_2^7$  and the data from  $R_1^2$  is shifted into  $R_2^8$ .
4. The last step is repeated for 6 more clock cycles, and at the last one, the value from  $R_1^1$  is shifted to  $R_2^2$ ,  $R_1^2$  is shifted to  $R_2^2$  and so on.

At the end of the last cycle, the transposed data can be found at the first register of each row of  $R_2$ , and it can be loaded to the second 1D DCT/IDCT transform. After this initial 9 clock cycles,

new data is always flowing from the 1D transform into the transposer, which starts its operation from the beginning (the select signal of the multiplexer overflows and starts from 0), effectively pipelining this process. This means that the transposer contributes with a delay of 9 clock cycles for the denoising pipeline.

Using 8 bits for the grayscale representation of the input noisy image means that all the values are between 0 and 255. However, when filtering is performed, the transform domain coefficients of each patch are modified (some are discarded), resulting in the possibility of producing negative values and values larger than 255 when inverting the haar and DCT transforms. Hence, a rounding module is necessary in order to avoid overflow/underflow when truncating the 32 bit representation used in the denoising pipeline to the 8 bit one used in grayscale images. The rounding module analyses the 32 bit value, and if it is bigger than 255, it is truncated to 255. On the other hand, if it is smaller than 0, it is truncated to 0. Values between 0 and 255 are rounded to the nearest integer, and ties (decimal part equal to 0.5) are rounded towards the nearest even number.

Finally, the clock delay of the 2D DCT and IDCT modules can be computed by adding the individual delays of each component of the module. Each module has an input pipeline stage, followed by the 1D transform, the transposer, and another 1D transform. This yields a delay contribution of  $1+3+9+3=16$  clock cycles for the denoising pipeline.

#### 4.3.2 Haar Transform Modules

As previously mentioned, there are two haar related modules in the pipeline: the haar transform and the inverse haar transform modules. The design of both modules is very simple, since the haar transform itself is also very simple.

As it is applied on the 3rd dimension of the group, and only one level of decomposition is possible, the haar transform takes the same pixel position from each column of the two blocks and computes its sum and difference. This originates 8 sums and 8 differences, which are divided by 2 and correspond, respectively, to the low pass and high pass coefficients. This can be expressed, for each column pixel, by the following equations:

$$C_{LP} = \frac{X_1 + X_2}{2} \quad \text{and} \quad C_{HP} = \frac{X_1 - X_2}{2} \quad (4.6)$$

This way, the haar transform module contains: two input registers, one for each block's column; then, eight adders and eight subtractors that compute the coefficients; and an output register. This means that the haar transform contributes with a delay of 2 clock cycles for the denoising pipeline.

The inverse haar module is a purely combinational module, that reconstructs the column data from the low pass and high pass coefficients. This reconstruction simply requires additions and subtractions of the correct high pass and low pass coefficients. Considering again each column pixel, we have the following expressions:

$$X_1 = C_{LP} + C_{HP} \quad \text{and} \quad X_2 = C_{LP} - C_{HP} \quad (4.7)$$

This way, the inverse haar module contains eight adders and eight subtractors that operate on each pair of high pass and low pass coefficients, outputting the two block's columns. The RTL schematics of both the haar transform module and inverse haar transform module can be seen on figure 4.10.

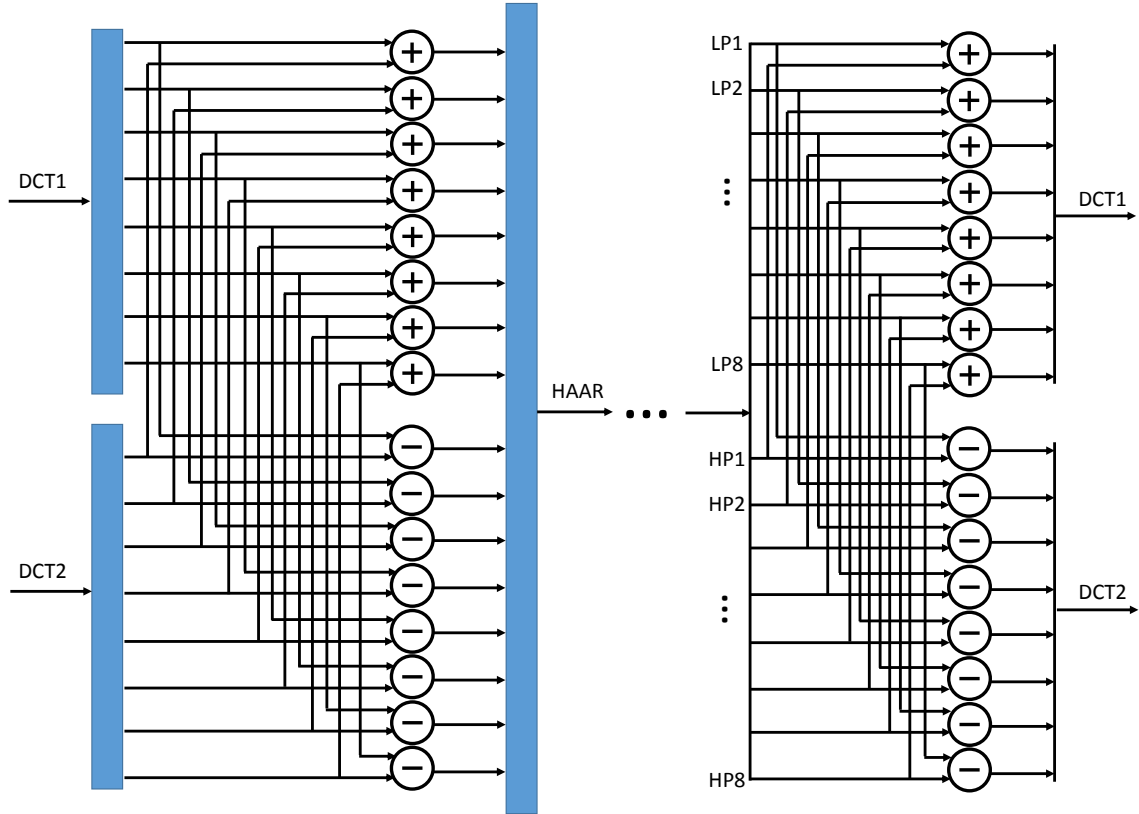


Figure 4.10: RTL schematics of the haar and inverse haar transform modules

### 4.3.3 Hard Thresholding Module

The hard thresholding module is responsible for filtering the transform domain coefficients that are considered to be noise. As previously explained, this is done by setting to zero the coefficients that are lower, in absolute value, than a certain threshold. Furthermore, the hard thresholding module is also responsible for counting the number of non-zero coefficients in each group. The module operates as follows:

- The modulus of each of the 16 coefficients is computed and after the result is compared to the threshold, using a less than operator.
- The output of each of the comparators drives the reset signal of a register, so that if the value is 1, the register is set to zero, otherwise it holds the original value of the coefficient.

- In parallel, the comparators outputs are negated and summed by a tree of adders, in order to count the number of non-zero coefficients. This value is added to an accumulator register, that at the end of 64 clock cycles (16 blocks per group, 8 rows per block, 2 blocks in parallel, yields  $16 \times 8 / 2 = 64$ ) holds the value of the group weight.
- This register is then reset at the end of processing of each group.

The hard thresholding module only contains 1 register for each input, meaning that it contributes with a single clock cycle delay for the denoising pipeline. The RTL schematic of the hard thresholding module can be seen in figure 4.11.

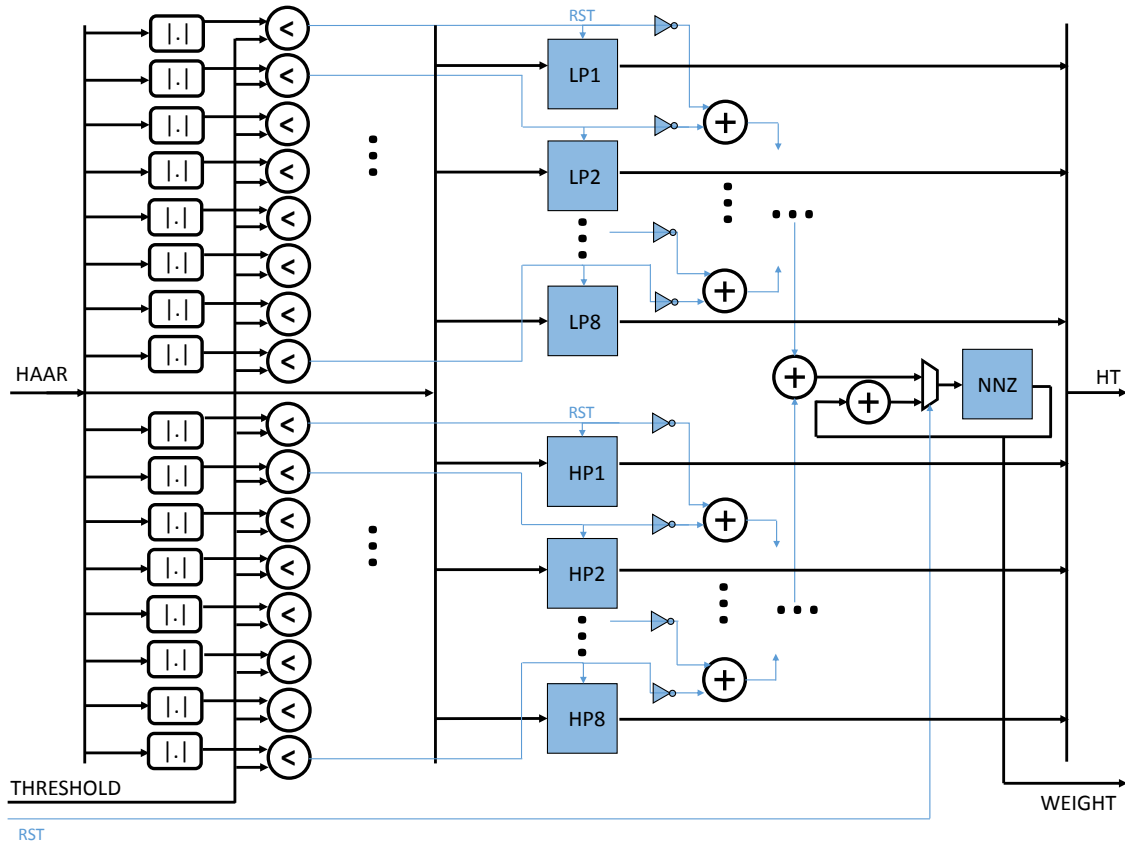


Figure 4.11: RTL schematics of the hard thresholding module

#### 4.3.4 Position Memory and Decoder

The positions memory consists of a wide register, that holds all the positions for the 16 groups formed in the block matching step, and outputs two positions in parallel. Since each pair of coordinates has 10 bits, and each group has 16 blocks, the register has  $10 \times 16 \times 16 = 2560$  bits. In order to output the correct positions, the position memory also has two counters: a 3 bit clock counter and a 7 bit position counter. The clock counter is necessary so that the positions are outputted for 8 clock cycles, since the memory module outputs one block row per clock cycle.



Then, when the clock counter reaches 7, the position counter is incremented. Finally, there is a multiplexer that selects the positions from the register according to the position counter, i.e., when the position counter is zero, the 1st and 2nd positions of the register are outputted, and for each increment in the counter, the following two positions are outputted. This means that to process all the groups the number of clock cycles necessary is  $8 \cdot 16 \cdot 16 / 2 = 1024$ . The position memory is controlled by two signals: *load* that enables the input of the register, and *en*, which enables the clock counter.

In between the output of the positions memory and the data memory, there is a position decoder module. This module is purely combinational and converts the positions to the correct addresses and shifts in the memory module. This is done by summing the clock counter value to the *y* coordinate (in order to select the correct row). Since each matching processor operates on a local neighborhood, its positions are also local, meaning that the *x* coordinate needs to be translated to the "big" neighborhood that contains all the 16 matching processors. The step between each reference block is 4 pixels, which means that this translation is simply a sum of the local *x* coordinate with 4 times the matching processor ID (from 0 to 15). This way, it is assured that the correct blocks are extracted from the data memory module.

### 4.3.5 Memory Module

The data memory module used in the denoising pipeline follows the same design as the memory in each matching processor. However, there are two main differences. The first is the size: in the matching processor each memory holds a  $39 \times 39$  neighborhood, meaning that five, 64 bit wide, memories with 39 positions are used. However, for the denoising pipeline, the memory needs to hold the full neighborhood of all the 16 matching processors, which is of size  $(39 + 4 \cdot 15) \times 39 = 99 \times 39$ . In order to store 99 pixels without modifying the size of each individual memory (8 pixels, i.e. a block row), 13 memories are necessary, which amounts to 104 pixels. This means that the last 5 pixels of the last memory are not relevant.

The second main difference is the fact that this memory module can output two rows in parallel. This is needed due to the parallelism necessary for the haar transform to be applied in the 3rd dimension. To achieve this, the memory contains two ports, each accepting a line address and *shift* and *block* signals, as in the matching processor memory. Furthermore, as happened in the matching processor's memory, a full neighborhood row can be loaded in a single clock cycle, because all 13 internal memories operate in parallel.

### 4.3.6 Control Module

The control module is extremely important for the correct operation of the denoising pipeline, because it needs to enable each module at the correct clock cycle, in order not to break the pipeline. The clock cycle delay that each module contributes to the pipeline was addressed in each module's

subsection. Hence, the total delay of the pipeline is given by:

$$N_{CLK} = 16 + 2 + 1 + 16 = 35 \quad (4.8)$$

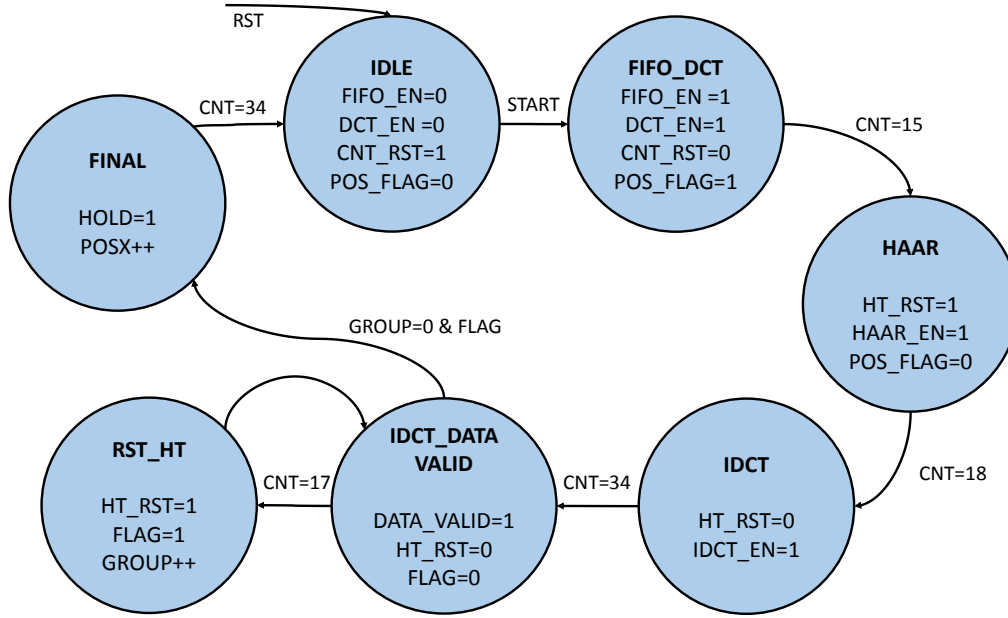


Figure 4.12: State diagram of the denoising pipeline control module

The denoising pipeline control module is modeled as an FSM with seven states and its state diagram can be seen on figure 4.12. It operates as follows:

1. Waiting in IDLE for the *start* signal, sent by the processor control module in the array of matching processors, that signals the end of the block matching step.
2. When the *start* signal is detected with the value 1, the state machine changes to state FIFO\_DCT, that enables the positions memory and the two 2D DCT modules, as well as a position flag. Furthermore, an internal counter is activated.
3. When the counter reaches 15, the state machine changes to state HAAR, that enables the haar transform module. Although the 2D DCT introduces a delay of 16 clock cycles, the transition value for the counter is 15, because the enable signal will only be seen with the correct value at the haar transform module one clock later of being changed in the control module.
4. When the counter reaches 18, the state machine changes to state IDCT, that enables the IDCT module, and disables the reset signal for the accumulator that stores the group weight in the hard thresholding module.

5. When the counter reaches 34, the state machine changes to state IDCT\_DATA\_VALID, that activates the *data\_valid* signal, indicating that the data output of the pipeline is already valid.
6. When the counter reaches 17, after intended overflowing, the state machine changes to state RST\_HT, that is a single clock state responsible for resetting the group weight accumulator. In this state, there is also an active flag, that enables a register, which stores the current group weight, before being reset. After one clock cycle, the state returns to IDCT\_DATA\_VALID.
7. This process is repeated for every group, and when a group counter overflows (reaches 0 for the second time), the state machine changes to state FINAL.
8. In this last state, there is still a need to wait 16 clock cycles, so that the last data rows can reach the end of the pipeline (from the hard thresholding module to the end there is a delay of 16, introduced by the 2D IDCT module). When this happens, the state machine changes to state IDLE, and the collaborative filtering step is done.

## 4.4 Top Level Module

This section elaborates on the operation and design of the top level module. This module is responsible for connecting the array of matching processors with the denoising pipeline, plus the memory and control modules necessary for the correct operation of the system. There are six memory modules and seven control modules, which will all be explained in the following sections.

### 4.4.1 Memory Modules and Control

The first memory module is the next neighborhood memory, which is responsible for holding the next data to be processed, while the matching processors are working on the current neighborhood. This memory contains 26 parallel, 32 bit wide memories with 39 positions, so that it holds a full 104x39 neighborhood. Each individual memory is loaded sequentially, selected by the *shift* input. However, the output port of the memory gathers the output of the same position of all individual memories. This way, the output of the memory can provide a full neighborhood row (104 pixels) at each clock cycle.

This memory module is controlled by the next control module. This module is an FSM with four states and operates as follows:

1. Waiting in state IDLE for the *start* signal sent by the master control module, which signals that a new neighborhood should be loaded from the input BRAM.
2. When the *start* signal is detected with the value 1, the state machine changes to state LOAD\_ROW, that activates the write enable for the memory, and starts the counter for the *shift* signal.

3. When the *shift* signal reaches the value 24, the state machine changes to state ADDR\_INC, that increments the write address and resets the *shift* value.
4. If the write address is smaller than 38, meaning that there are still rows to be loaded, the state goes back to LOAD\_ROW, and the previous step is repeated. Otherwise, the state goes to FINAL, which is just a single clock cycle buffer state that resets the state machine variables and sends the *end* pulse to the master control module. Finally, the state machine moves back to state IDLE.

This control module is also responsible for determining the correct read address from the input BRAM, in order to fetch the correct data. The read address is calculated from 5 other variables: the write address, the *shift* signal, the two coordinates for the current neighborhood being processed and a parameter called *line\_width*. The read address is given by:

$$read\_addr = x\_addr + (K + 8) * y\_addr, \quad (4.9)$$

where *x\_addr* is the address part due to the *x* coordinate, *y\_addr* is due to the *y* coordinate and *K* is the image width divided by 4 (32 bit positions can hold 4 pixels). This multiplication simply "skips" the number of rows, so that the correct row is selected. In order to avoid this multiplication, the allowed values for image widths are hard coded, and the *line\_width* parameter controls a multiplexer that adds different shifts off the *y\_addr* to achieve the correct result. The allowed image widths and corresponding *line\_width* and *K* values can be found in table 4.1. These values are the usually found in test images (256x256 and 512x512) and real images with 1, 2, 3, 5 and 8 megapixels.

Table 4.1: Hard coded image widths

<i>line_width</i>	<b>Width</b>	<i>K</i>
0	256	64
1	512	128
2	1280	320
3	1920	480
4	2048	512
5	2560	640
6	3264	816

The constant value 8 in equation 4.9 is due to zero padding the image. The image is padded with a frame of zeros of size 15 in the top and left sides, and 17 in the right and bottom sides. This means an additional 32 (hence, 8 memory positions) pixels in each image row. The multiplication by  $(K + 8)$  is then implemented as additions and shifts, by finding the correct sum of powers of 2 that sum to *K*. For the image widths considered, the shifts from 3 to 9 are hard coded, and then *line\_width* controls a multiplexer that sums the correct shifts. For example, if the image width is 1280, *K* is 320, which means that the shifts 6 and 8 need to be summed, plus the zero pad (shift of 3), which is independent of the image width.

The values of  $x\_addr$  and  $y\_addr$  are given by the following equations:

$$\begin{aligned} x\_addr &= x\_curr \ll 4 + shift, \\ y\_addr &= y\_curr \ll 2 + w\_addr, \end{aligned}$$

where  $x\_curr$  and  $y\_curr$  are the coordinates of the current neighborhood (these are signals sent by the master control module). The shift of 2 in the  $y$  coordinate is due to step of 4 pixels between reference blocks. However, in the  $x$  coordinate, the shift is 4, since 16 processors in parallel mean that the first reference block from consecutive neighborhoods are  $16*4=64$  pixels apart.

The second memory module in the top level is the buffer memory. This memory is the simplest in the architecture, because it only serves as a buffer between the matching processor and the denoising pipeline. When the neighborhood to be processed is loaded from the next memory to the matching processors, it is parallelly loaded into the buffer memory. Then, when the processor control activates the *load\_den\_mem*, the buffer control loads the contents of the buffer memory into the internal memory of the denoising pipeline.

The buffer control module is a very simple FSM with only two states: IDLE and LOAD\_MEM. The state is IDLE until the *load\_den\_mem* signal has the value of 1. Then the state machine changes to state LOAD\_MEM and a counter is activated. When this counter reaches 38, the state goes back to IDLE and the denoising pipeline memory has the contents of the buffer memory.

The third memory module is the positions memory. This memory stores the positions of the blocks that are outputted by the denoising pipeline. There are  $16*16=256$  positions to be stored, and each position occupies 16 bits, which means that the positions memory has a depth of 128 and is 32 bit wide. The two most significant bytes of a memory position hold the position of the block outputted by the second 2D IDCT module, and the remainder bytes hold the other position.

The position memory is controlled by the position control module. This module is an FSM with four states and operates as follows:

1. Waiting in state IDLE for the *pos\_flag* signal, sent by the denoising pipeline control when the first positions are being outputted.
2. When the *pos\_flag* signal is detected with the value 1, the state machine changes to state WRITE, that enables the write enable of the memory and starts a counter, writing the first positions value in the memory. On the next clock cycle, the state changes to WAIT and the write enable is set to zero.
3. When the counter reaches the value 7, meaning that the positions are going to change, the state goes back to WRITE and the new positions are written to memory.
4. This steps are repeated until the address overflows. When this happens, the state machine changes to state FINAL, which is a buffer state, and then back to IDLE.

The fourth memory module is the weights memory. This memory stores the group weights outputted by the denoising pipeline. Since there are 16 groups, and each weight occupies 8 bits,

the weights memory is composed by 4 smaller memories, 8 bit wide and with 4 positions. The memory input is only 8 bits and there is a byte enable to choose in which memory to store data. However, as the output will go to the output BRAM, data is read from all four memories in parallel in order to have 32 bits width.

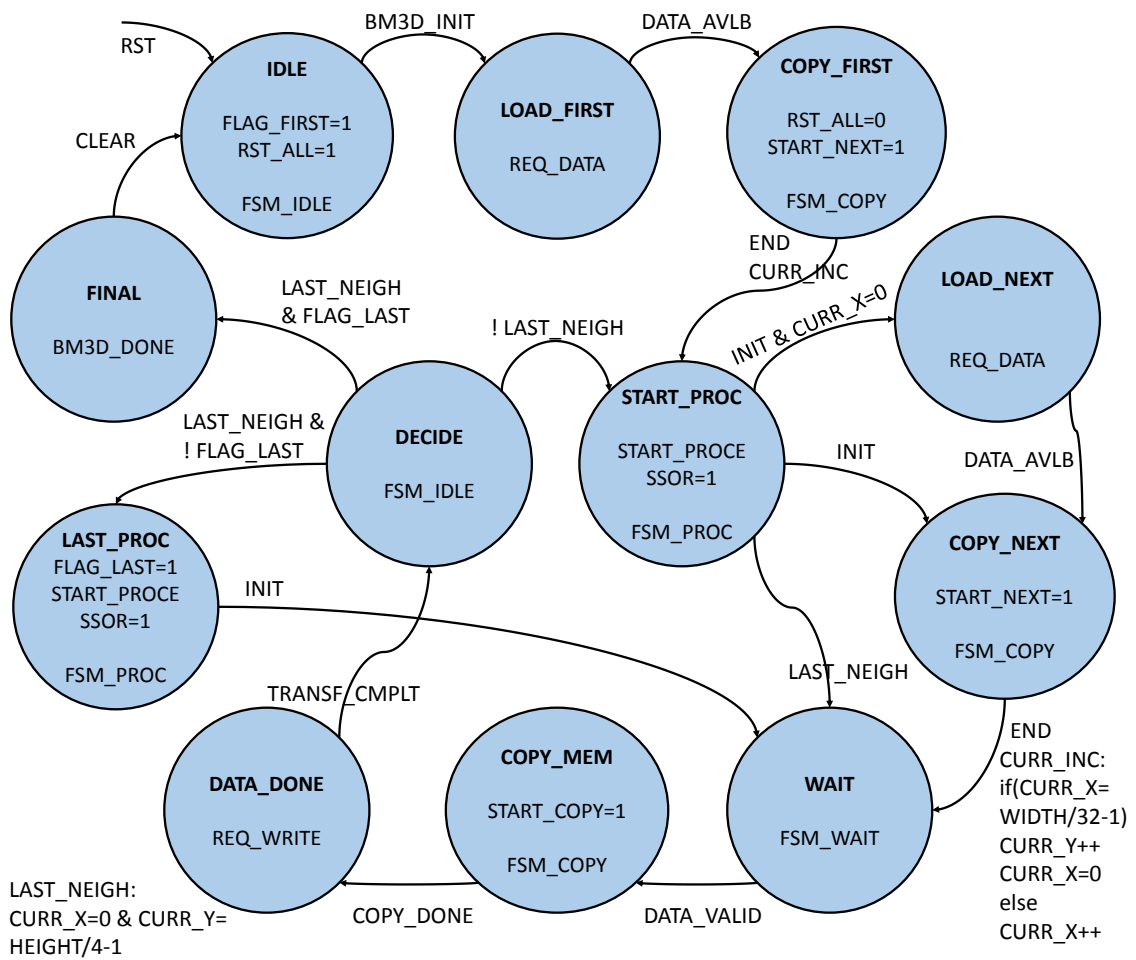
This memory is controlled by the weights control module. This module is an FSM with three states: IDLE, WRITE and INC\_ADDR. When in state IDLE, the state machine waits for the weight flag, that signals a new weight is available, and then moves to the WRITE state. In this state the write enable of the memory is activated and the weight is stored. Furthermore, the byte enable is incremented, so that the next weight is stored in the next smaller memory inside the weight memory. If the byte enable has the value of 3, the state changes to INC\_ADDR, and the write address is incremented, and then the state changes back to IDLE. This process is repeated each time the weight flag as the value 1. Otherwise, if the byte enable is not 3, the state goes directly to IDLE, because an increment of the write address is still not necessary.

The fifth and sixth memory modules are the same: data memory. This memory stores the actual denoised blocks of the image. There is a need of two modules in parallel because the denoising pipeline outputs two block rows per clock cycle. The data memory consists of two 32 bit wide memories in parallel, with 1024 positions. This value is due to the size of the data in 8x8 block rows, which is  $16 \times 16 \times 8 = 2048$ , and each row occupies 64 bits (8 pixels). This way, each of the memories is loaded with half of a row at the same address. When reading the data out of the memory, the LSB of the read address acts as the memory selector, in order to consecutively output the correct half rows. These memories are controlled by the data control module. This module is a simple counter with the same bits as the address of the memories (10 bits). This counter is enabled by the *data\_valid* signal, meaning that while the data outputted from the denoising pipeline is correct, the address is incremented and the data is correctly stored in the data memories.

Finally, there is an additional module regarding the memory interfaces and its control. This is the copy control module, which is responsible for copying the data from the positions, weights and data memories into the output BRAM. It is modeled as an FSM with eight states, that controls the read address for each memory and the write address of the output BRAM. There is also a multiplexer that is changed according to which memory is being copied. The four memories are copied sequentially, starting by the first data memory, then the second data memory, the positions memory, and finally the weights memory. Between each copy there is a state that resets the read address and increments the multiplexer select signal. The write address however, is always incremented so that data is not overwritten in the output BRAM. The final composition of the output BRAM is then: all the data from even numbered blocks of the groups (8 KB), followed by all the data from odd numbered blocks (8 KB), followed by all the positions (512 Bytes), and finally all the group weights (16 Bytes).

### 4.4.2 Master Control

The last module of the top level design is the master control module. As the name states, this module is responsible for controlling all the remaining control modules and the BM3D system in general. This module has three registers to communicate with the outside: a status register, a control register and an image size register. The control register is used to issue commands to the BM3D module, while the status register contains the current status of operation of the system, and the image size register is used to input the image size, so that the master control can compute how many neighborhoods the image is divided into. The master control module is modeled as an FSM with 12 states and its state diagram can be seen on figure 4.13. It operates as follows:



1. Waiting in state IDLE for the BM3D\_INIT command in the control register. When this command is detected the state changes to LOAD\_FIRST, that sets the status register to the value REQ\_DATA. This tells the user of the BM3D system that the input BRAM can be filled with data.

2. Waiting in state `LOAD_FIRST` for the `DATA_AVLB` command in the control register. When this command is detected, it means that the input BRAM contains the first 39 lines of the image (first neighborhoods). Then the state changes to `COPY_FIRST`, that sends the *start\_next* signal to the next memory control module, in order to load the first neighborhood.
3. When the *end* signal from the next memory control module has the value 1, the state machine changes to state `START_PROC` and the *curr\_x* and *curr\_y* counters are properly incremented. In this state, the *start\_processor* signal is sent to the processor control, in order to start the block matching step.
4. Then, there are three possibilities:
  - (a) If the *curr\_x* and *curr\_y* counters indicate the last neighborhood, the state machine changes to state `WAIT`.
  - (b) If the *curr\_x* counter is zero and the processor is starting, signaled by *init*, the state machine changes to state `LOAD_NEXT`. This state has the same behavior as state `LOAD_FIRST` and the following state is `COPY_NEXT` instead of `COPY_FIRST`.
  - (c) Else, when the processor is starting, the state machine changes to state `COPY_NEXT`. This state has the same behavior as state `COPY_FIRST` and the following state is `WAIT`.
5. When in the `WAIT` state, if the *data\_valid* signal is detected with value 1, the state changes to `COPY_MEM`. This state sends the *start\_copy* signal to the copy control module in order to copy the data as soon as possible.
6. When the *copy\_done* signal is detected with the value 1, the state machine moves to state `DATA_DONE`. In this state, the value `REQ_WRITE` is written in the status register, to signal the user that the output BRAM has new data that can be read.
7. When the user finishes reading the data, it writes the `TRANSF_CMPLT` command in the control register, and the state changes to `DECIDE`. In this state a decision is made: if the last neighborhood has been reached but the *last* flag is zero, the state changes to `LAST_PROC`, where the last processing is done; else the state changes to `START_PROC` and processing continues.
8. When in the `LAST_PROC` state, the *last* flag is set and steps 5 to 7 are repeated, with the difference that in the `DECIDE` state, as the *last* flag is 1, the next state is `FINAL`.
9. In this last state, the status register is set to `BM3D_DONE`, signaling that the BM3D system has finished processing the image.



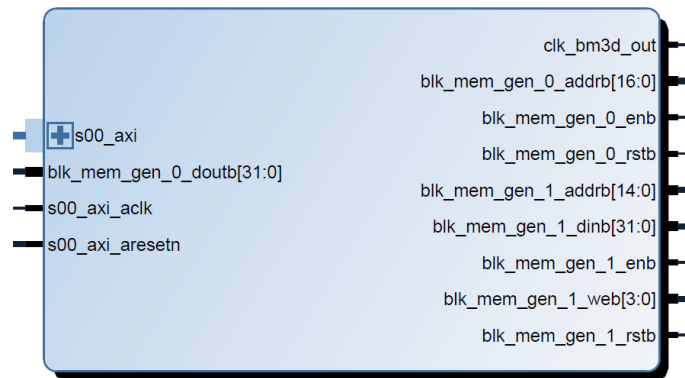


Figure 4.14: BM3D IP core block

## 4.5 BM3D IP Core

This section elaborates on the design of the BM3D IP core, which is responsible for connecting the BM3D top level module with an AXI Lite Slave interface and a Mixed-Mode Clock Manager (MMCM) module in order to package the whole system as an IP core that can be integrated in any embedded system that supports the Advanced eXtensible Interface (AXI) standard for inter-chip communication. The BM3D IP core functional block can be seen in figure 4.14. It contains the following inputs and outputs:

- **s00\_axi**: AXI Lite interface port.
- **s00\_axi\_aclk**: Input clock for the AXI interface.
- **s00\_axi\_aresetn**: Input reset for the AXI interface.
- **blk\_mem\_gen\_0\_doutb**: 32 bit data input from the input BRAM.
- **clk\_bm3d\_out**: Output clock from the BM3D IP, that is the clock input for port B of the BRAM's.
- **blk\_mem\_gen\_0\_addrb**: 17 bit address output for port B of the input BRAM.
- **blk\_mem\_gen\_0\_enb**: Enable output for port B of the input BRAM.
- **blk\_mem\_gen\_0\_rst**: Reset output for port B of the input BRAM.
- **blk\_mem\_gen\_1\_addrb**: 15 bit address output for port B of the output BRAM.
- **blk\_mem\_gen\_1\_dinb**: 32 bit data output for port B of the output BRAM.
- **blk\_mem\_gen\_1\_enb**: Enable output for port B of the output BRAM.
- **blk\_mem\_gen\_1\_web**: Write enable output for port B of the output BRAM.
- **blk\_mem\_gen\_1\_rst**: Reset output for port B of the output BRAM.

### 4.5.1 AXI Lite Slave

The AXI Lite Slave block is responsible for handling the AXI communications coming from the outside of the IP block. The AXI protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) open standard from ARM, which defines specifications for on chip interconnects and communications, that facilitates the development of system on chip (SoC) designs. AXI is a master-slave based protocol, and the BM3D IP is a slave, meaning that it cannot initiate data transfers. The Lite interface is used for control purposes, i.e., it does not allow memory mapped or stream access (which are also specified in the AXI protocol). This way, the AXI Slave block contains four registers: the three registers described in the master control subsection 4.4.2 and an extra debug register. In the case of the control and image size registers, these can be written and read through the AXI interface. On the other hand, the status and debug registers, are written by the master control module, so only reading from the AXI interface is allowed. This block was not developed, because it is included in the intellectual property (IP) of the Vivado Design Suite, and can be directly instantiated in the design.

### 4.5.2 The MMCM

The MMCM module is used to generate multiple clocks with defined phase and frequency relationships to a given input clock. Since the AXI Lite block can only operate at a maximum frequency of 50 MHz, the MMCM was included in the design in order to allow the BM3D module to operate at higher frequencies. This higher speed clock is outputted by the BM3D IP core and connected to the external BRAMs, in order that the whole system is synchronized. This way, the usage of the MMCM module creates two clock domains in the design. This can be problematic, as problems of meta-stability can occur in the domain crossing. Hence, a synchronizer module was included in each signal that crosses the two clock domains. This module consists of two stages of registers, that are clocked by the destination clock in the domain crossing, so that the input signal can be detected and stored. The module is included in the BM3D IP core, to synchronize the connections between the AXI Lite module and the master control module. This way, four 32 bit synchronizer modules were used to sync the four registers, plus one 1 bit synchronizer module for the AXI reset signal, which is also used to reset the master control module, and thus, the whole system.

## Chapter 5

# Test Methodology and Results

In this chapter, the tools and methodology used to test the BM3D hardware implementation will be presented. Furthermore, the results obtained will be exposed and discussed. These results include: denoising performance evaluation; execution time comparison against a CPU implementation of the algorithm; and power consumption versus speed trade-off assessment.

### 5.1 Experimental Setup

The BM3D system was tested in the ZYNQ-7000 ZC706 SoC evaluation board from Xilinx. This board contains the XC7Z045 system on chip from Xilinx, which is divided into two sections: the Processing System (PS) and the Programmable Logic (PL). The PS section contains:

- Dual core ARM Cortex A9 processor with maximum frequency of 800 MHz, 32 KB of L1 cache and 512 KB of L2 cache.
- 256 KB on-chip memory.
- External memory interfaces with DDR3 support and an 8 channel DMA controller.
- Several I/O peripherals and interfaces, such as, Ethernet, USB, CAN, SPI, UART, among others.

The PL section is based on a Xilinx Kintex-7 FPGA, which contains:

- 350K logic cells (approximately equivalent to 5.2M ASIC gates).
- 218,600 lookup tables (LUTs).
- 437,200 flip-flops (FFs).
- 545 BRAM's, which amount to 2,180 KB of memory.
- 900 DSP slices.

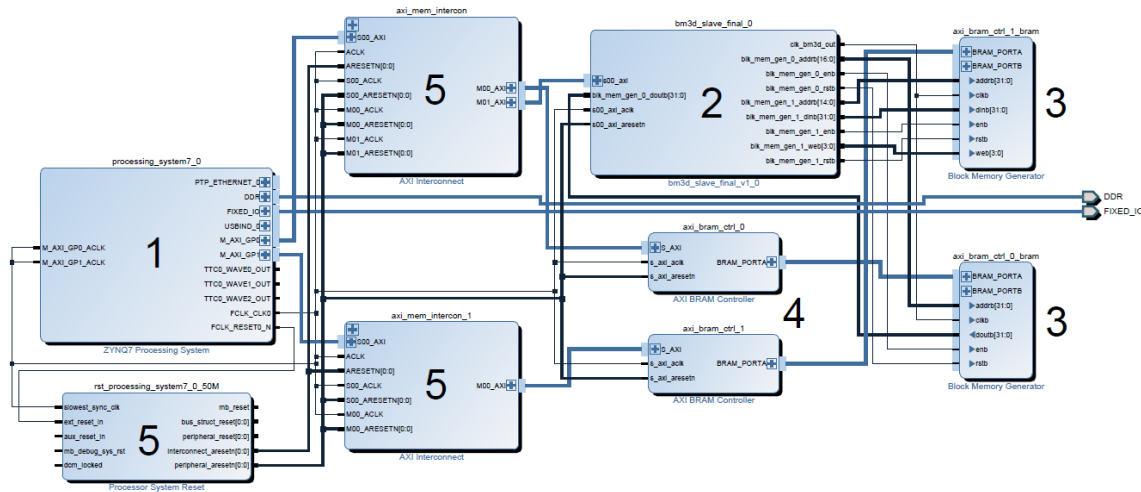


Figure 5.1: Board Design Schematic

The connection between PS and PL is assured by a high bandwidth interconnect based on the AMBA AXI specification. The ZYNQ board also contains 1GB of DDR3 RAM that can be used to store data by the PS, or the PL if the necessary hardware is implemented.

### 5.1.1 ZYNQ Board Setup

Designing an application for the ZYNQ board can be done in two different ways. The first option is a bare-metal application, i.e., the application runs directly on the SoC, with the ARM processor and programmable logic being programmed via JTAG. This option makes it difficult to load data into the system, in this case, the image to be denoised. This would be done using one of the communication ports of the SoC, for example UART, which would mean designing a program on the computer side in order to transfer the image to the board.

The second option is installing a linux based operating system (OS) in the SD card that is provided with the board. This OS runs on the ARM processor and the board can be accessed via UART or Ethernet, for example, using a secure shell (SSH) connection. This way, the image and the bitstream of the design can easily be loaded into the board and a C program can be developed to run on the OS. This option was the selected for testing the BM3D system, because it is simpler and less time consuming when compared with the first option.

In order to program the PL of the ZYNQ system and be able to access it from the PS, a board design must be created in the Vivado Design Suite. This design specifies how the PL and PS are connected and various IP modules can be added to the design. This modules are for example the BRAMs necessary for the input and output data of the BM3D system, and the respective AXI control modules. The schematic of the board design can be seen in figure 5.1. It contains five different elements, numbered from 1 to 5:

1. ZYNQ Processing System: This block represents the ZYNQ PS and it can be customized in order to configure AXI Master and Slave ports, alter the clock frequency, among other

configurations. For the BM3D system, 2 AXI General Purpose Master ports (GP0 and GP1) are enabled: one for communicating with the AXI Lite Slave and the input BRAM controller, and another dedicated to the output BRAM controller.

2. The BM3D IP block: This is the BM3D system, which is connected to port B of each BRAM and to the GP0 port of the PS.
3. BRAMs: This two blocks are, respectively from top to bottom, the output and input BRAMs.
4. AXI BRAM Controllers: This blocks are the BRAM controllers, which also have an AXI Slave port, so that they can be controlled by the PS.
5. Interconnects and Reset system: These blocks represent the interconnects that are actually present in the ZYNQ SoC and are used to connect AXI Masters to their respective Slaves.

In order that the PS can access the BRAM's and the AXI Lite registers, each AXI slave is mapped to a specific memory address. This way, data transfers are simply performed by writing/reading to/from each address. The address for the first AXI Lite register is 0x43C00000. The remaining registers are accessed by adding 4 to this address, since the registers are 32 bit wide. The order of the registers is: control register, image size register, status register, debug register. The base address for the input BRAM is 0x40000000 and for the output BRAM is 0x80000000.

With the board design done, a bitstream can be generated. This bitstream configures the SoC with the desired design, by programming the PL and the interconnects between the PS and the PL. The bitstream is then sent to the board via a terminal with an SSH connection to the ZYNQ board. Then, to program the board, there is a device driver called `xdevcfg` that is supplied with the linux OS from Xilinx, that programs the SoC via the ARM processor's JTAG port.

### 5.1.2 C Program

In order to control the BM3D system with the ZYNQ PS from the linux OS, a C program must be developed. This program is responsible for sending the correct commands to the BM3D system and answering to its requests, as well as placing the image data in the input BRAM and reading the processed data from the output BRAM. Furthermore, the ARM CPU running the program also performs the aggregation process, i.e, the last step of the BM3D algorithm. This way, the BM3D IP can be seen as a co-processor, that speeds up the block matching and collaborative filtering steps of the algorithm. The C program operates as follows:

1. The image to be denoised is contained in a binary file. At the beginning of the program this file is opened and the image is zero padded and copied to a known RAM address. Since this is a user space program, and the linux kernel does not allow access to physical addresses, a special function called `getvaddr` is used in order to map the desired physical address into virtual memory pages.

2. The pointers for writing/reading to/from the BRAMs and the AXI Lite registers are created, by using the *getvaddr* function. Then, an unsigned char array called *data*, where the data from the output BRAM is going to be stored, is allocated in memory. Furthermore, two 2D float arrays with the size of the image are allocated in memory, in order to perform the aggregation process, as described in section 3.1.
3. The image size register is loaded with the image width and height and the system is reset by writing 0xFF to the control register. Then, the BM3D\_INIT value is written to the control register, as well as the estimated noise power of the image.
4. The BM3D system starts, and the program waits for the ACK value on the status register. Then, it waits for the REQ\_DATA value, which signals the first data request from the system. Waiting is implemented as busy polling, i.e., a while control loop checking the value of the status register.
5. The first 39 rows of the image are loaded into the input BRAM, by directly copying the data from RAM to the BRAM, inside a for loop. Once the transfer is complete, the DATA\_AVLB command is written to the control register, followed by waiting for the ACK.
6. The program enters a while loop, with exit condition being that the status register has the value BM3D\_DONE, which signals that processing is done. Inside the loop there are two if blocks: one for data requests and another for write requests.
7. If a data request is detected (control register with value REQ\_DATA), the program loads the next 4 rows of the image into the input BRAM. Then, the same procedure as in step 5 is repeated.
8. If a write request is detected (control register with value REQ\_WRITE), the program stores the data from the output BRAM sequentially in the *data* array. This is done in a for loop, by directly copying from the output BRAM address pointer to each position *i* of the *data* array. Then, the aggregation step is performed for the data available.
9. The aggregation step consists of a chain of for loops. The first loop runs through each of the 16 matching processors (16 groups). Inside this loop, another one runs for each pair of blocks in the group (1 to 8). Finally, there are two more loops for each pixel of each block. In the most inner loop, each pixel value is multiplied by the kaiser window and the group weight, and then added to the correct position of the image buffer. At the same position of the weight buffer, the multiplication of the kaiser window with the group weight is also added.
10. Once the aggregation is complete, the TRANSF\_CMPLT command is written to the control register, followed by waiting for the ACK.
11. When the BM3D system finishes processing, the BM3D\_DONE value is placed in the status register, and the while loop finishes. Follows a write in the control register to reset the

system, and the final cleanup of the C program (closing files and clearing memory). Then, the division of the image and weight buffers is computed, in order to generate the result image. This is done by a loop, doing an element-wise division and writing the resulting pixel value into an output file. At the end of the program's execution, this binary file contains the pixel values of the whole image.

During the execution of the C program, the *clock* function is used to keep track of the time spent in processing tasks by the CPU (aggregation) and in data transfers, as well as the total time. This way, the time spent by the algorithm on the BM3D co-processor is computed as the difference between the total time and the processing and data transfers times.

## 5.2 Results

### 5.2.1 FPGA Resource Usage

As referred in section 5.1, the ZYNQ board's programmable logic has different resources available in different quantities for the implementation of reconfigurable designs. Table 5.1 shows the resource usage of the BM3D system in terms of absolute values and also percentage of utilization of the ZYNQ board.

Table 5.1: FPGA Resource Usage

Resource	Utilization	Available	Utilization (%)
FF	87251	437200	19.96
LUT	89270	218600	40.84
Memory LUT	15081	70400	21.42
BRAM	40.5	545	7.43
MMCM	1	8	12.50

As can be seen, the BM3D system needs a high number of hardware resources, due to the complexity of the operations performed. However, the highest utilization percentage is for LUTs at around 41%, meaning that the ZYNQ PL is under 50% utilization in all resources. Nevertheless, the ZYNQ PL contains a large number of resources, which can be misleading when analyzing the occupation of the BM3D system. The high number of LUTs is due to the arithmetic operations performed in both the matching processor's  $\ell_1$  norm units and the denoising pipeline's 2D DCT and IDCT modules. The flip-flop utilization is mainly due to the pipeline stages that are used in the whole architecture, while the memory LUT usage can be explained by all the special memories included in the design, that cannot be synthesized in BRAMs. The input and output BRAMs occupy 40.5 of the BRAMs in the PL, while only 1 out of 8 MMCMs is used. The values presented are for a frequency of 50 MHz in the whole system. With other tested frequencies for the BM3D system (75, 100 and 125 MHz) the occupation is virtually the same, as only the number of LUTs changes, with a maximum variation of 3 LUTs, which is negligible.

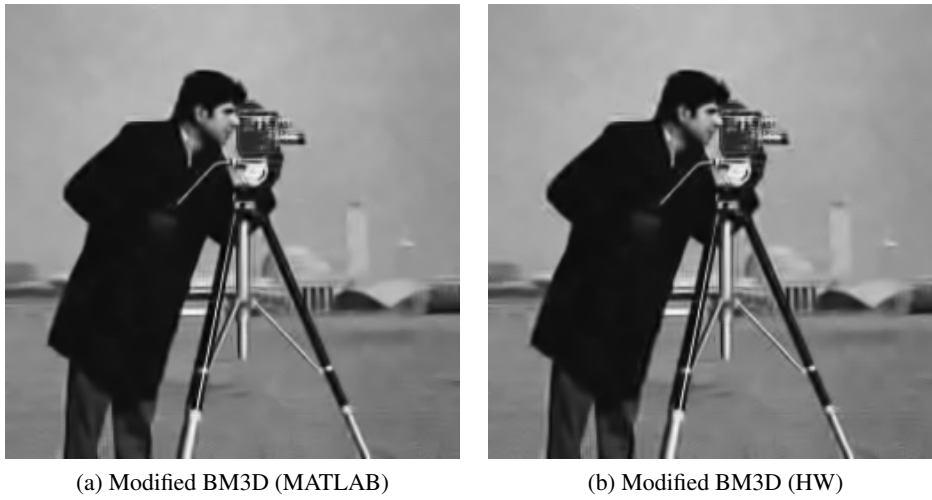


Figure 5.2: a) Basic estimate image obtained by modified BM3D (PSNR=27.16 dB) in MATLAB; b) Basic estimate obtained by modified BM3D (PSNR=27.17 dB) in hardware. Noisy image was corrupted with noise power of 25.

### 5.2.2 Denoising Performance

In order to test the denoising performance of the BM3D system, the output raw image file is downloaded from the ZYNQ board to the PC and read into an array by a MATLAB script. This script then computes the PSNR of the result image, by comparing it with the original image.

Table 5.2: PSNR (dB) results for 256x256 Cameraman Image

$\sigma$	Noisy Image	Original BM3D	Modified BM3D (MATLAB)	Modified BM3D (HW)
5	34.16	38.20	34.79	34.82
10	28.13	33.94	31.25	31.31
15	24.62	31.67	29.42	29.46
20	22.12	30.24	28.18	28.19
25	20.18	29.14	27.16	27.17
30	18.60	28.25	26.28	26.29
35	17.30	27.41	25.53	25.52
40	16.10	26.33	24.83	24.83
45	15.08	26.07	24.21	24.23
50	14.16	25.55	23.65	23.65

Table 5.2 shows the PSNR results in dB for the image Cameraman for the original BM3D algorithm, as well as both the modified versions: implementation in MATLAB and hardware. As mentioned in subsection 3.3.4, the modified BM3D achieves worse denoising performance. In the table, it can be seen that the hardware implementation results are as expected by the values obtained in MATLAB, with the majority of times even being slightly better. However, the PSNR difference is minimal (in the range of 0 to 0.03 dB), meaning that in terms of visual quality the



images are virtually the same. This can be confirmed by comparing both images side by side, as shown in figure 5.2. Nevertheless, this small PSNR difference can be explained by the fact that different rounding is used in the MATLAB and hardware implementations. In MATLAB, each pixel is rounded to an 8 bit grayscale representation after the aggregation step, i.e., there is a possibility that some pixels are larger than 255 or smaller than 0, because each pixel of each filtered block is not individually rounded. However, in the hardware implementation, this is exactly what happens, with each pixel being rounded at the output of the denoising pipeline. This way, it is guaranteed that during the aggregation step, since a weighted average is used, the pixel values never go outside the 0 to 255 range.

Table 5.3: PSNR (dB) results for 512x512 Lena Image

$\sigma$	Noisy Image	Original BM3D	Modified BM3D (MATLAB)	Modified BM3D (HW)
5	34.15	38.63	36.84	36.86
10	28.14	35.65	34.30	34.33
15	24.62	33.83	32.58	32.61
20	22.12	32.47	31.32	31.36
25	20.19	31.37	30.32	30.35
30	18.60	30.46	29.52	29.55
35	17.26	29.65	28.84	28.87
40	16.10	28.72	28.24	28.25
45	15.08	28.45	27.72	27.73
50	14.16	27.92	27.24	27.24

Table 5.3 shows the PSNR results in dB for the image Lena for the original BM3D algorithm, as well as both the modified versions: implementation in MATLAB and hardware. Once again, the modified BM3D achieves worse denoising results, and the implementations in MATLAB and hardware have the same results. Figure 5.3 shows the result images for both implementations of the modified BM3D algorithm.

With the BM3D system validated against the MATLAB implementation, a more extensive experience was performed. This experiment consisted of testing all 512x512 images in the image dataset used by the original BM3D paper with the same noise powers from the experiences before: 5 to 50 in steps of 5. Then, all the results obtained for each noise power are averaged for all the images in the dataset. This dataset consists of 6 images with 512x512 resolution: Lena, Barbara, Boats, Man, Couple and Hill. These are all standard images used for testing in image processing. The PSNR and SSIM of each image was measured, and a comparison of the results for the noisy image, the original BM3D and the modified BM3D on hardware can be seen in the graphs shown in figure 5.4.

The results show that both the PSNR and SSIM measurements for the hardware implementation have a relatively large decrease when compared to the original BM3D algorithm for low noise powers, with a slight increase when compared with the noisy image measurements. However, as the noise power increases, the results of the hardware implementation start to get on par with



Figure 5.3: a) Basic estimate image obtained by modified BM3D (PSNR=30.32 dB) in MATLAB; b) Basic estimate obtained by modified BM3D (PSNR=30.35 dB) in hardware. Noisy image was corrupted with noise power of 25.

those of the original BM3D, and both of them are extremely larger when compared to the noisy image. The PSNR difference between both implementations ranges from an absolute maximum of 2.48 dB for noise power of 5 to a minimum of 0.81 for noise power of 50, which corresponds, respectively to 6.58% and 2.97% in relative difference. Regarding the SSIM measurements, the difference ranges from a maximum of 0.046 for noise power of 15 to a minimum of 0.011 for noise power of 50, which corresponds, respectively to 5.46% and 1.61% in relative difference.

Table 5.4: PSNR (dB) results comparison for 1,2,3,5 and 8 MP images.

$\sigma$	Noisy	City		Coat		Bridge		Palace		NYC	
		SW	HW	SW	HW	SW	HW	SW	HW	SW	HW
10	28.14	34.27	31.95	37.61	36.59	35.21	33.02	35.83	34.29	38.08	37.29
20	22.11	30.31	28.28	34.32	33.29	31.45	29.24	32.23	30.44	34.29	33.59
30	18.59	28.15	26.28	32.28	30.85	29.18	26.97	30.14	28.21	32.02	31.25
40	16.09	26.01	24.88	30.70	28.84	27.02	25.30	28.47	26.56	30.24	29.38
50	14.14	25.30	23.83	29.83	27.06	26.11	23.98	27.43	25.20	29.31	27.76

Now that the good performance of the BM3D system for images of low resolutions is established, more tests are necessary for images of higher resolutions. Since there are no standard test images with high resolutions available, an 8MP camera was used to take some pictures and then these pictures were downsampled to the following resolutions: 1MP - 1280x960, 2MP - 1920x1080, 3MP - 2048x1536, 5MP - 2560x1920 and the original resolution of 8MP - 3264x2448. In order to maintain the quality of the pictures, the downsampling was performed in a free image processing software. This experiment consisted of testing these images with noise powers of 10 to

50 in steps of 10, in both the original BM3D algorithm and hardware implementation, and comparing the results by measuring PSNR and SSIM. The results of this experiment can be found in tables 5.4 and 5.5.

Table 5.5: SSIM results comparison for 1,2,3,5 and 8 MP images.

$\sigma$	Noisy	City		Coat		Bridge		Palace		NYC	
		SW	HW	SW	HW	SW	HW	SW	HW	SW	HW
10	0.582	0.944	0.920	0.927	0.920	0.941	0.921	0.940	0.924	0.951	0.949
20	0.331	0.882	0.856	0.874	0.882	0.889	0.871	0.878	0.858	0.896	0.910
30	0.222	0.819	0.804	0.817	0.844	0.831	0.824	0.818	0.811	0.834	0.877
40	0.162	0.748	0.761	0.759	0.809	0.766	0.780	0.759	0.776	0.770	0.848
50	0.124	0.715	0.725	0.739	0.775	0.737	0.740	0.727	0.747	0.746	0.821

The results show that for larger image resolutions the BM3D system continues to underperform the original implementation's PSNR results at about 1 to 2 dB. However, for high noise powers, it can be seen that the SSIM values achieved by the BM3D system surpass the ones obtained by the software implementation. Some result images obtained in this experiment can be found in appendix A, as well as the original images.

### 5.2.3 Execution Time

Regarding the execution time of the algorithm, testing was done on two different CPUs and on the ZYNQ board. In the CPUs, the original C software provided by the authors of the original BM3D paper was used <sup>1</sup>.

Table 5.6: FPGA execution time results (in seconds) for images of increasing resolutions and frequencies.

Image (Resolution)	FPGA @50MHz	@75MHz	@100MHz	@125MHz
Cameraman (256x256)	0.02883	0.01962	0.01492	0.01216
Lena (512x512)	0.11640	0.07911	0.06008	0.04864
City (1MP - 1280x960)	0.54903	0.37282	0.28280	0.22884
Coat (2MP - 1920x1080)	0.92683	0.62986	0.47768	0.38681
Bridge (3MP - 2048x1536)	1.40597	0.95502	0.72380	0.58497
Palace (5MP - 2560x1920)	2.20020	1.49499	1.13421	0.91690
NYC (8MP - 3264x2448)	3.58071	2.43341	1.84608	1.49307

Table 5.6 contains the execution times for images of increasing resolutions with constant noise power of 20, running on the ZYNQ board, with the BM3D system operating at 50,75,100 and 125 MHz. Table 5.7 contains the results for the execution times of the same images on two CPUs: CPU1, which is an Intel i5-3317U processor, running at 1.7 GHz, and CPU2, an Intel i5-4570 processor, running at 3.2 GHz. Better results are expected from CPU2, since it is a desktop

<sup>1</sup>Software available at <http://www.cs.tut.fi/foi/GCF-BM3D/>

processor, with more processing power and running at a higher frequency, while CPU1 is a low power laptop processor.

Table 5.7: Comparison of CPU and FPGA @125MHz execution time results (in seconds) for images of increasing resolutions.

Image (Resolution)	CPU1 @1.7GHz	CPU2 @3.2GHz	FPGA @125MHz
Cameraman (256x256)	0.59360	0.35580	0.01216
Lena (512x512)	2.38350	1.49210	0.04864
City (1MP - 1280x960)	11.2099	7.26179	0.22884
Coat (2MP - 1920x1080)	20.3389	12.9978	0.38681
Bridge (3MP - 2048x1536)	30.0177	19.3971	0.58497
Palace (5MP - 2560x1920)	47.5931	30.8079	0.91690
NYC (8MP - 3264x2448)	82.4164	51.3583	1.49307

As can be seen by these results, the BM3D system is significantly faster than both CPUs tested. To better evaluate these results, the speedup value is computed, which is simply the division of the execution time on the CPUs and on the ZYNQ board. This value was computed for all frequencies of operation and comparing with both CPUs and the results are presented in the graphs shown in figure 5.5.

As can be seen in the graphs, the profile of the speedup results is quite similar, with an almost monotonic increase with the resolution of the image. For the BM3D system running at 50 MHz (blue) and comparing with CPU1, the speedup values range from 20.6 to 23.0, averaging 21.4. For increasing frequencies, the values are even higher: average of 31.4 for 75 MHz (grey), 41.4 for 100 MHz (orange) and 51.1 for 125 MHz (yellow). When comparing with CPU2, the values are smaller, due to the higher performance of this processor. Even so, the speedup ranges from 12.3 to 14.3 in the slower frequency of 50 MHz, up to 29.3 to 34.4 for the highest frequency (125 MHz). These results are very satisfactory, since the ZYNQ board runs at a much lower frequency than the CPUs. It is also worth noting that as the image resolution increases, the speedup also increases, meaning that the BM3D system can handle bigger images with more ease than the CPUs.

Not included in the execution time results are the data transfer times and aggregation times. The data transfers using the experimental setup available are quite slow. This is due to a bottleneck in using the AXI general purpose master ports of the PS, which can only operate at 50 MHz, and the interconnect to the PL is not an high bandwidth one (only 32 bit wide). A solution to this would be to use one of the high performance (HP) AXI ports available in the connection between the PS and the PL, which allow 64 bit wide direct access to the system's RAM. However, these are AXI slave ports, meaning that the BM3D IP would have to implement an AXI full Master in order to control all the data transfers itself. While this is an advantage, because the BM3D IP operates without supervision of the CPU, directly fetching data from the RAM through the HP slave ports, the AXI Master logic is harder and more time consuming to implement and it is not available in the IP catalog of the Xilinx Vivado tool. Since this are data transfers, and their time is not counted as time actually performing the BM3D algorithm, the easier and faster approach was

taken, and the data transfer times disregarded. The same applies to the aggregation process, which is done simultaneously by the ARM CPU while the BM3D system is operating. The ARM CPU operates at a frequency of 666 MHz, and has much less processing power than the CPUs used to run the BM3D algorithm on a PC. This way, a direct comparison of the two times wouldn't be possible. Furthermore, the aggregation only contributes for about 2% of the time consumption in the BM3D algorithm, since the block matching and collaborative filtering steps are much more computationally intensive.

#### 5.2.4 Power Consumption

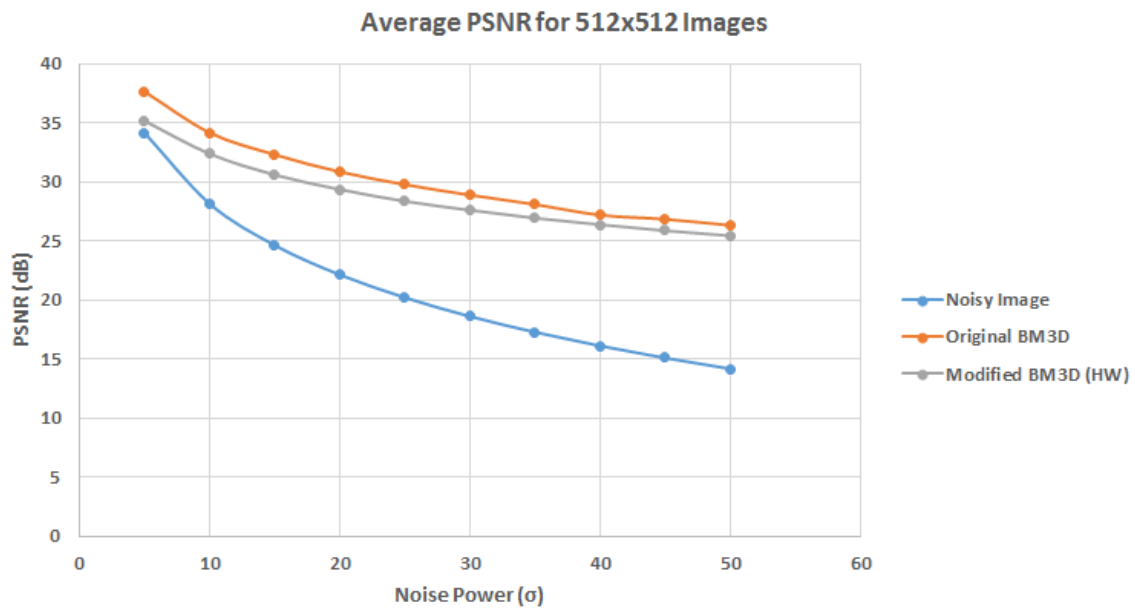
The power consumption distribution of the BM3D system for the different synthesized frequencies is shown in figure 5.6. These values were obtained using the analysis provided by the Vivado Design Suite after the design is implemented.

For all frequencies, power consumption is between 2 and 3 Watts, which is low considering the amount of FPGA resources used and the complexity of the BM3D algorithm. Furthermore, it is possible to see that a good distribution of dynamic versus static power is achieved, with 89 to 91% of dynamic power and 11 to 9% static power. The static power increase with frequency is completely negligible, with only an increase of 4 mW between 50 and 125 MHz. On the other hand, the dynamic power increases fairly with the frequency, which amounts for the total increase of the power consumption. This increase is perceptible in the Clocks, Signals and Logic categories of the dynamic power, whilst the BRAM and MMCM modules power usage stays constant. This is due to the fact that the increased frequency is applied to the BM3D IP core, and not to the remainder of the design (AXI Lite, BRAM, PS), meaning that all the signals and logic will have increased switching, therefore consuming more power.

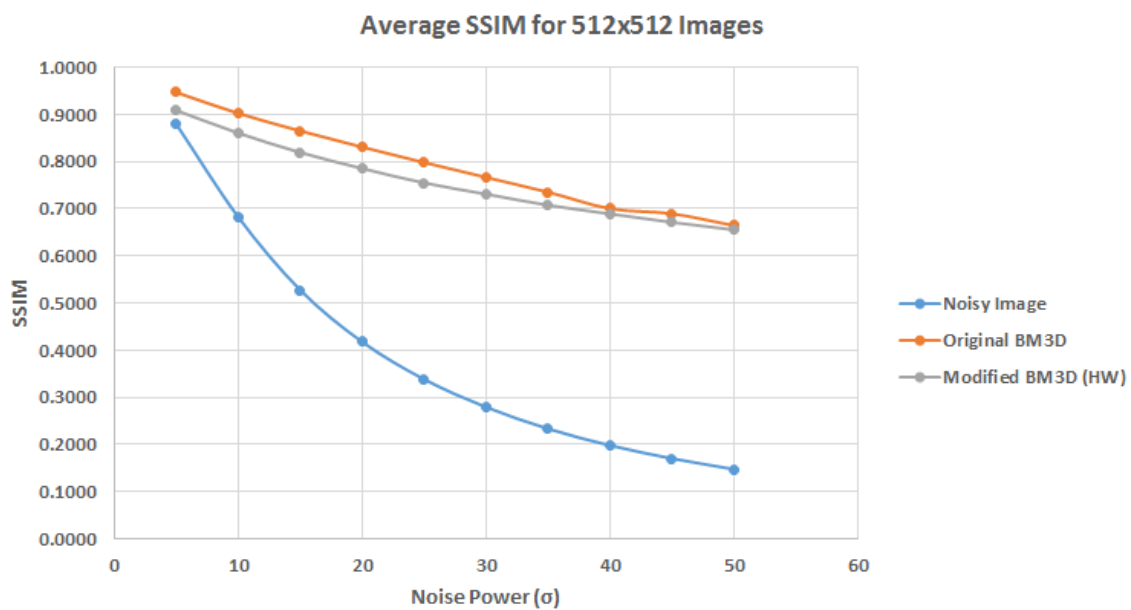
From the power distribution charts, it can be seen that the values of each category do not add up to the dynamic power shown. When this discrepancy was detected, the values of each category were summed and then the sum was subtracted from the value shown. For all frequencies this calculation always yielded the same value, 1.568 W, which is the dynamic power consumption estimation of the PS of the ZYNQ board, because this value never changes. Furthermore, when synthesizing only the BM3D IP core, the static power consumption stayed about the same value of 0.25 W. This is believed to be the static power consumption of the PL part of the ZYNQ SoC. This way, the power consumption can be recalculated to split the power usage of the PS and the PL. The results of this recalculation can be found in table 5.8.

Table 5.8: PL and PS power consumption (W) for different frequencies of operation.

Frequency	@50MHz	@75MHz	@100MHz	@125MHz
<b>Power Usage</b>	2.350	2.538	2.748	2.928
<b>PS Power</b>	1.568	1.568	1.568	1.568
<b>PL Power</b>	0.782	0.970	1.180	1.360

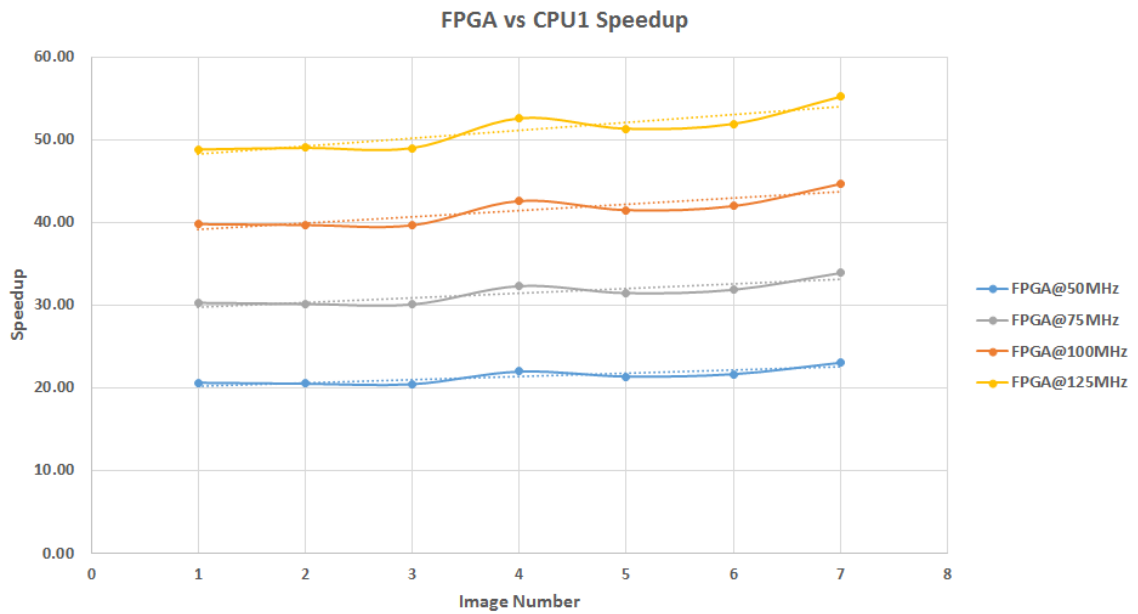


(a) Average PSNR

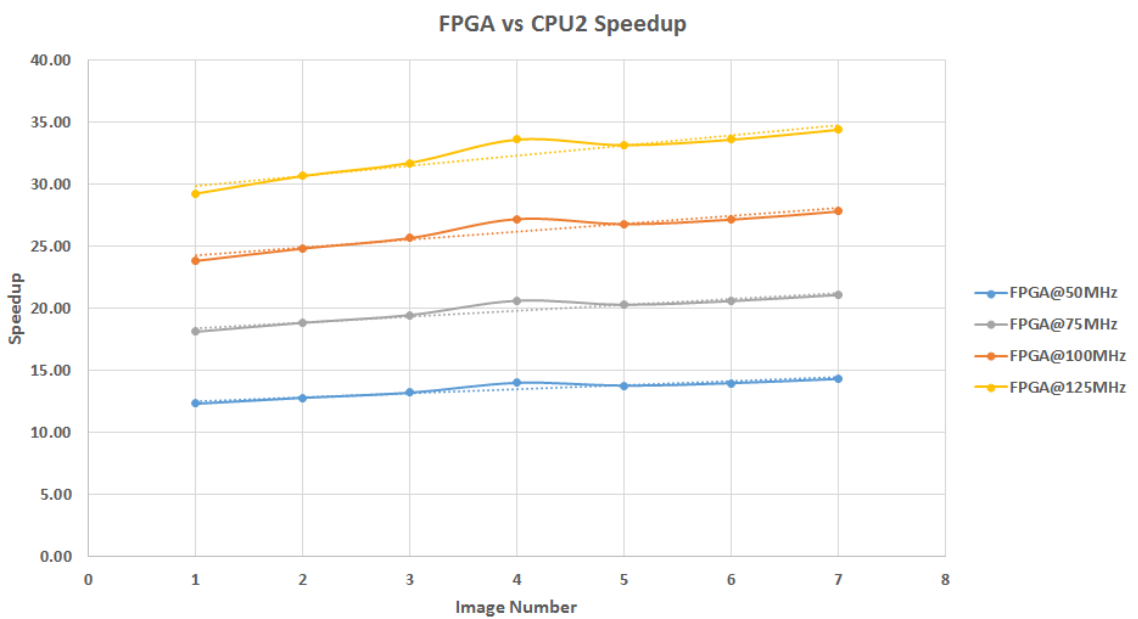


(b) Average SSIM

Figure 5.4: a) Average PSNR (dB) for 512x512 images with noise powers from 5 to 50. b) Average SSIM for 512x512 images with noise powers from 5 to 50.



(a) FPGA vs CPU1



(b) FPGA vs CPU2

Figure 5.5: a) FPGA vs CPU1 speedup for increasing image resolutions. b) FPGA vs CPU2 speedup for increasing image resolutions. Images are in the same order as in tables 5.6 and 5.7.

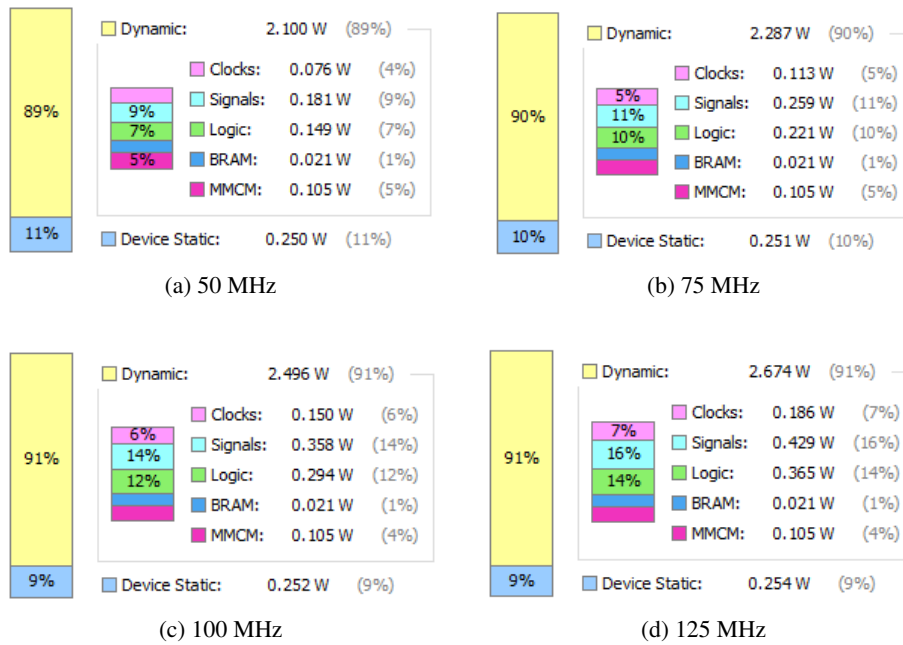


Figure 5.6: Power consumption distribution of the BM3D system for different operating frequencies.



## Chapter 6

# Conclusions and Future Work

### 6.1 Conclusions

In this work a novel hardware implementation for the BM3D image denoising algorithm was presented. Taking advantage of an FPGA, the BM3D IP core accelerates the process of image denoising, allowing for excellent denoising performance, whilst having a low power consumption. Considering its complexity, this work was developed in multiple phases.

The first phase was an extensive study of the background and state of the art in image denoising in order to understand and compare the BM3D denoising algorithm with others. The second phase was developing the BM3D algorithm in MATLAB, in order to gain some experience with image denoising and to further analyze in full extent the bottlenecks of the algorithm. Then, the third phase consisted of adjusting or modifying these bottlenecks in order to optimize these steps towards an efficient hardware implementation. Followed the most extensive phase of the work, which is the actual development of the hardware. This consisted in a sequential development of various hardware blocks, followed by their simulation, validation and testing. Then, the connection of all blocks to create the BM3D IP was performed, which eventually lead to more simulations and some design tweaks in order to achieve a fully working system. Finally, the last phase was the final testing of the system with noisy images, and the evaluation of several performance parameters: denoising performance by calculating the PSNR and SSIM measurements; run time comparison between hardware and software implementations of the algorithm; and power consumption of the hardware implementation.

Experimental results show that the BM3D IP can effectively denoise images of various resolutions and with a wide range of noise powers. Furthermore, although denoising performance is negatively affected when compared to the software implementation of the BM3D algorithm, the gains in execution time and power consumption are significant enough to consider the BM3D IP as an alternative for image denoising in software.

## 6.2 Future Work

Despite the fact that the results achieved by the BM3D system are quite satisfactory, there are some improvements worth mentioning as possibilities for future work.

The first improvement is the hardware implementation of the wiener filter in order to further improve the denoising results, specially in terms of SSIM. The design of this new implementation was already analyzed and involves the development of the wiener filter block, which includes implementing a division operation. This block would then be included in parallel to the hard thresholding block in the denoising pipeline currently implemented in the BM3D system and data would be multiplexed in order to choose which denoising technique to use. The second improvement would be increasing the parallelization of the data in the denoising pipeline, so that more levels of the haar wavelet decomposition could be applied. This improvement would lead to increased denoising performance, while also decreasing execution time. On the negative side, implementation area would increase dramatically.

The third improvement regards an extension of the system to color images. As proposed in the original paper, the BM3D algorithm can also be applied to color images, by converting an RGB colorspace to an YCbCr one, for example. In this colorspace, the Y channel is the luminance and Cb and Cr are the chrominance channels. With this separation, the BM3D is applied by performing block matching on the Y channel, and reusing the groups formed on all three channels in order to perform collaborative filtering separately on each channel. Regarding an hardware implementation, the same structure as the current BM3D system could be used, with an additional modification that would allow to skip the block matching step for the Cb and Cr channels. This way, on a first run, the Y channel would be processed exactly as a grayscale image, and all the group positions would be kept. Then, the Cb and Cr channels would be consecutively processed, using the groups previously formed and skipping the block matching step. However, this processing would eventually triple the execution time, since the size of the image data is also the triple of a grayscale image.

## **Appendix A**

### **Result Images**

In this appendix some denoised images for resolutions of 1,2,3,5 and 8 megapixels are presented. The noisy images counterparts are also shown. All the original images can be seen in figure [A.1](#).

(a) *City*(b) *Coat*(c) *Bridge*(d) *Palace*(e) *NYC*

Figure A.1: a) Original image *City* with 1MP resolution (1280x960). b) Original image *Coat* with 2MP resolution (1920x1080). c) Original image *Bridge* with 3MP resolution (2048x1536). d) Original image *Palace* with 5MP resolution (2560x1920). e) Original image *NYC* with 8MP resolution (3264x2448).



(a) Noisy Image with  $\sigma=40$



(b) Denoised Image

Figure A.2: a) Noisy image *City* with noise power of 40. b) Denoised image with PSNR=24.88 dB and SSIM=0.761.



(a) Noisy Image with  $\sigma=20$



(b) Denoised Image

Figure A.3: a) Noisy image *Coat* with noise power of 20. b) Denoised image with PSNR=33.29 dB and SSIM=0.882.



(a) Noisy Image with  $\sigma=50$



(b) Denoised Image

Figure A.4: a) Noisy image *Bridge* with noise power of 50. b) Denoised image with PSNR=23.98 dB and SSIM=0.740.



(a) Noisy Image with  $\sigma=30$



(b) Denoised Image

Figure A.5: a) Noisy image *Palace* with noise power of 30. b) Denoised image with PSNR=28.21 dB and SSIM=0.811.





(a) Noisy Image with  $\sigma=40$



(b) Denoised Image

Figure A.6: a) Noisy image *NYC* with noise power of 40. b) Denoised image with PSNR=29.38 dB and SSIM=0.848.



# References

- [1] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian. Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering. *IEEE Transactions on Image Processing*, 16(8):2080–2095, August 2007.
- [2] S Asha, S Bhuvana, and R Radhakrishnan. A Survey on Content Based Image Retrieval Based on Feature Extraction. *Int. J. Novel. Res. Eng & Pharm. Sci*, 1(06):29–34, 2014.
- [3] Alan C. Bovik. *Handbook of Image and Video Processing*. Academic Press, 2010.
- [4] A Buades, B Coll, and J Morel. A Review of Image Denoising Algorithms, with a New One. *Multiscale Modeling & Simulation*, 4(2):490–530, 2005.
- [5] Z. 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, April 2004.
- [6] I Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.
- [7] Maarten Jansen. *Noise Reduction by Wavelet Thresholding*, volume 161 of *Lecture Notes in Statistics*. Springer New York, New York, NY, 2001.
- [8] Richard Szeliski. *Computer vision: algorithms and applications*. 2010.
- [9] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete Cosine Transform. *IEEE Transactions on Computers*, C-23(1):90–93, January 1974.
- [10] Robert Grover Brown and Patrick Y. C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, 2012.
- [11] Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. 2010.
- [12] G. Davis, S. Mallat, and M. Avellaneda. Adaptive greedy approximations. *Constructive Approximation*, 13(1):57–98, March 1997.
- [13] S.G. Mallat. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.
- [14] Y.C. Pati, R. Rezaifar, and P.S. Krishnaprasad. Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. In *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers*, pages 40–44. IEEE Comput. Soc. Press, 1993.

- [15] Scott Shaobing Chen, David L. Donoho, and Michael A. Saunders. Atomic Decomposition by Basis Pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, January 1998.
- [16] K Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559 – 572, 1901.
- [17] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, 1933.
- [18] M. Aharon, M. Elad, and A. Bruckstein. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, November 2006.
- [19] David L. Donoho and Iain M. Johnstone. Minimax estimation via wavelet shrinkage. *The Annals of Statistics*, 26(3):879–921, June 1998.
- [20] David L. Donoho and Iain M. Johnstone. Ideal spatial adaptation by wavelet shrinkage. *Biometrika*, 81(3):425–455, September 1994.
- [21] David L. Donoho and Iain M. Johnstone. Adapting to Unknown Smoothness via Wavelet Shrinkage. *Journal of the American Statistical Association*, 90(432):1200–1224, February 1995.
- [22] D.L. Donoho. De-noising by soft-thresholding. *IEEE Transactions on Information Theory*, 41(3):613–627, May 1995.
- [23] A Charnballe, R A DeVore, N Y Lee, and B J Lucier. Nonlinear wavelet image processing: variational problems, compression, and noise removal through wavelet shrinkage. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 7(3):319–35, January 1998.
- [24] P. Moulin. Analysis of multiresolution image denoising schemes using generalized-Gaussian priors. In *Proceedings of the IEEE-SP International Symposium on Time-Frequency and Time-Scale Analysis (Cat. No.98TH8380)*, pages 633–636. IEEE, 1998.
- [25] D. L. Donoho and R. R. Coifman. Translation invariant denoising. In *Wavelets and Statistics*, pages 125–150. Springer-Verlag New York, New York, NY, 1995.
- [26] David L. Donoho. Wedgelets: nearly minimax estimation of edges. *The Annals of Statistics*, 27(3):859–897, June 1999.
- [27] Minh Do and Martin Vetterli. Contourlets. In *Beyond Wavelets*, pages 1–27. Academic Press, New York, NY, 2001.
- [28] Emmanuel J Candès and David L Donoho. New tight frames of curvelets and optimal representations of objects with piecewise C2 singularities. *Communications on Pure and Applied Mathematics*, 57(2):219–266, 2004.
- [29] E. Le Pennec and S. Mallat. Sparse geometric image representations with bandelets. *IEEE Transactions on Image Processing*, 14(4):423–438, April 2005.
- [30] W.T. Freeman and E.H. Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(9):891–906, 1991.

- [31] Qian Chen and Dapeng Wu. Image denoising by bounded block matching and 3D filtering. *Signal Processing*, 90(9):2778–2783, September 2010.
- [32] K Dabov, A Foi, V Katkovnik, and K Egiazarian. A nonlocal and shape-adaptive transform-domain collaborative filtering. ... *Int. Workshop on Local and Non ...*, 2008.
- [33] K Dabov, A Foi, V Katkovnik, and K Egiazarian. BM3D image denoising with shape-adaptive principal component analysis. *SPARS'09-Signal Processing ...*, 2009.
- [34] Michael Elad and Michal Aharon. Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries. *IEEE Transactions on Image Processing*, 15(12):3736–3745, December 2006.
- [35] Weisheng Dong, Lei Zhang, and Guangming Shi. Centralized sparse representation for image restoration. In *2011 International Conference on Computer Vision*, pages 1259–1266. IEEE, November 2011.
- [36] Weisheng Dong, Lei Zhang, Guangming Shi, and Xin Li. Nonlocally centralized sparse representation for image restoration. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 22(4):1620–30, April 2013.
- [37] Hua Zhong, Ke Ma, and Yang Zhou. Modified BM3D algorithm for image denoising using nonlocal centralization prior. *Signal Processing*, 106:342–347, January 2015.
- [38] S.O. Memik, K. Bazargan, and M. Sarrafzadeh. Image analysis and partitioning for FPGA implementation of image restoration. In *2000 IEEE Workshop on SiGNAL PROCESSING SYSTEMS. SiPS 2000. Design and Implementation (Cat. No.00TH8528)*, pages 346–355. IEEE, 2000.
- [39] S.O. Memik, A.K. Katsaggelos, and M. Sarrafzadeh. Analysis and FPGA implementation of image restoration under resource constraints. *IEEE Transactions on Computers*, 52(3):390–399, March 2003.
- [40] G. Saldana and M. Arias-Estrada. FPGA-Based Customizable Systolic Architecture for Image Processing Applications. In *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*, pages 3–3. IEEE, 2005.
- [41] Jonathan Joshi, Nisseem Nabar, and Parul Batra. Reconfigurable Implementation of Wavelet based Image Denoising. In *2006 49th IEEE International Midwest Symposium on Circuits and Systems*, volume 1, pages 475–478. IEEE, August 2006.
- [42] P. Brylski and M. Strzelecki. FPGA implementation of parallel digital image processor. In *Signal Processing Algorithms, Architectures, Arrangements, and Applications Conference Proceedings (SPA), 2010*, pages 25–28, 2010.
- [43] Stefano Di Carlo, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. AIDI: An adaptive image denoising FPGA-based IP-core for real-time applications. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*, pages 99–106. IEEE, June 2013.
- [44] Anna Gabiger-Rose, Matthias Kube, Robert Weigel, and Richard Rose. An FPGA-Based Fully Synchronized Design of a Bilateral Filter for Real-Time Image Denoising. *IEEE Transactions on Industrial Electronics*, 61(8):4093–4104, August 2014.

- [45] Peter J. Rousseeuw and Christophe Croux. Alternatives to the Median Absolute Deviation. *Journal of the American Statistical Association*, 88(424), February 1993.
- [46] Hanaa Hussain, Khaled Benkrid, Chuan Hong, and Huseyin Seker. An adaptive FPGA implementation of multi-core K-nearest neighbour ensemble classifier using dynamic partial reconfiguration. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 627–630. IEEE, August 2012.
- [47] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 988–991. IEEE, 1989.
- [48] Zhang-jin Chen and Zhi-gao Zhang. A High-Speed 2-D IDCT Processor for Image/Video Decoding. In *2009 2nd International Congress on Image and Signal Processing*, pages 1–4. IEEE, October 2009.
- [49] M. El Aakif, S. Belkouch, and M. M. Hassani. An efficient pipelined fast and multiplier-less 2-D IDCT for image/video decoding. In *2011 International Conference on Multimedia Computing and Systems*, pages 1–5. IEEE, April 2011.
- [50] IEEE Standard Specifications for the Implementations of 8X8 Inverse Discrete Cosine Transform, 1991.
- [51] A. Aggoun and I. Jalloh. Two-dimensional DCT/IDCT architecture. *IEE Proceedings - Computers and Digital Techniques*, 150(1):2, 2003.