



**I
N
A
O
E**

FPGA-based compressed sensing reconstruction of sparse signals

by

Héctor Daniel Rico Aniles

Thesis

Submitted to the Program in Electronics Science,
Electronics Science Department
in partial fulfillment of the requirements for the degree of

Master in Electronic Science

at the

**Instituto Nacional de Astrofísica, Óptica y
Electrónica**

September 2014
Tonantzintla, Puebla

Advisor:

Dr. Juan Manuel Ramírez Cortés

Co-Advisor:

Dr. José de Jesús Rangel Magdaleno

©INAOE 2014

All rights reserved

The author hereby grants to INAOE permission to reproduce and
to distribute copies of this thesis document in whole or in part



Abstract

Compressed sensing is a recently proposed technique aiming to acquire a signal with sparse or compressible representation in some domain, using a number of samples under the limit established by the Nyquist theorem. The challenge is to recover the sensed signal solving an underdetermined linear system. Several techniques such as the l_1 minimization, Greedy and combinatorial algorithms can be used for that purpose. Greedy algorithms have been found to be more suitable in hardware solutions, however they rely on efficient matrix inversion techniques in order to solve the underdetermined linear systems involved. In this work, a FPGA-based Greedy algorithm architecture with a Chebyshev-type method to solve matrix inversion problem is presented. The architecture was developed for Xilinx Virtex 4 XC4VSX25, Xilinx Spartan 6 XC6SLX45, Altera Cyclone IV EP4CGX150DF31C7 and Altera Cyclone II EP2C35F672C6 FPGAs. The described architecture represents a low-cost and generic solution, robust to changes in word length and signal size. Besides, a MATLAB Graphical User Interface is developed for compressed sensing theory exploration focused on matrix and transform test. MATLAB GUI uses the Compressed Sampling Matching Pursuit algorithm to recover the sensed signal; reconstruction can easily be extended to other compressed sensing algorithms.

Acknowledgements

To the Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE) for all their support in my academic training. To CONACYT for the financial support for my graduate studies and this thesis.

To Ph.D. Juan Manuel Ramírez Cortés and Ph.D. José de Jesús Rangel Magdaleno for all the patience and spent time taken on the direction of my thesis. Also, for guiding me and teaching me to conduct research.

To M. Sc. Jorge Miguel Pedraza Chávez, Ph.D. Esteban Tlelo Cuautle and Ph.D. Jorge Martínez Carballido for their corrections and suggestions to improve this thesis.

To my parents Mr. Héctor Rico and Mrs. María Aniles and to all the rest of my family and loved ones who always believed in me.

Contents

Abstract	i
Acknowledgements	ii
Contents	iii
List of Figures	v
List of Tables	vii
1 Introduction	1
1.1 Compressed Sensing Applications	3
2 Theoretical Fundamentals	5
2.1 Compressed Sensing	5
2.1.1 Sensing Matrix	6
2.1.1.1 Null Space Condition	7
2.1.1.2 Restricted Isometry Property	7
2.1.2 Reconstruction Algorithms	8
2.1.2.1 l_1 minimization	8
2.1.2.2 Greedy algorithms	9
2.1.2.3 CoSaMP Algorithm	9
2.2 Matrix Inversion	10
2.2.1 Moore-Penrose Inverse	10
2.2.2 Analytical Inversion	11
2.2.2.1 Adjoint matrix.	12
2.2.2.2 Cofactor	12
2.2.3 Chebyshev-type Method	13
2.2.3.1 Mathematical Formalization	13
2.2.3.2 Preconditioning	13
2.2.3.3 Word length	14
2.2.4 QR Decomposition	15
2.3 Transforms	16
2.3.1 Discrete Fourier Transform	16
2.3.2 Wavelet Transform	18
2.3.3 Discrete Cosine Transform	20

3	FPGA implementation	21
3.1	Matrix Inversion Architecture	21
3.1.1	System Structure	22
3.1.2	Preconditioner	23
3.1.3	Core	25
3.1.4	Verifier	27
3.1.5	Multiplexer	28
3.1.6	Control and storage	28
3.2	CoSaMP Architecture	30
3.2.1	Transposer	30
3.2.2	Support	31
3.2.3	Matrix Creation	33
3.2.4	Vector copier	34
3.2.5	Vector reset	35
3.2.6	Matrix inversion	36
3.2.7	Multiplication	36
3.2.8	Subtraction and Verifier	36
3.2.9	Control	37
4	MATLAB Graphical User Interface	39
4.1	GUI Inputs	39
4.1.1	Data type	39
4.1.2	Sensing matrix	41
4.1.3	Transform	42
4.1.4	Other inputs	44
4.2	GUI outputs	45
5	Results	48
5.1	FPGA architectures	48
5.1.1	Matrix Multiplication Architecture	48
5.1.2	Matrix Transposer Architecture	49
5.1.3	Matrix Inversion Architecture	52
5.1.4	CoSaMP Architecture	54
5.2	Matlab Graphical User Interface	56
6	Conclusion and Future Work	62
6.1	Future Work	63
6.2	Publications Obtained from this Thesis	63
A	CoSaMP VHDL code	64
	Bibliography	88

List of Figures

1.1	a) Original image. b) 256x256 image, reconstructed from 1300 measurements [1]	3
2.1	Graphical representation of compressed sensing concept [2].	6
2.2	Sinusoidal signal.	17
2.3	DFT of sinusoidal signal from figure 2.2.	17
3.1	Matrix inversion system structure.	23
3.2	Preconditioner block diagram.	24
3.3	Max block structure.	25
3.4	Core block diagram.	26
3.5	MAC data flow.	27
3.6	Verifier block diagram.	28
3.7	RAM memory block.	29
3.8	FPGA CoSaMP Architecture blocks.	30
3.9	Transposer address generator.	32
3.10	Row address generator path.	33
3.11	Column address generator path.	33
3.12	Decomposed vector in elements set and index set.	34
3.13	Great-to-small architecture.	34
3.14	Reading address of matrix creation block.	35
3.15	Vector copier structure.	35
3.16	Subtraction-verifier block structure.	37
4.1	Graphical User Interface as a compressed sensing tool.	40
4.2	Data type selection.	40
4.3	Data file selection.	40
4.4	Sensing matrix selection panel.	41
4.5	Custom sensing matrix name error.	42
4.6	Matrix dimension and sensed vector length mismatch.	42
4.7	Wavelet setting menu.	42
4.8	Discrete cosine transform setting menu.	43
4.9	Custom transform panel.	43
4.10	Wrong transform selection error.	44
4.11	Wrong inverse transform selection error.	44
4.12	Input parameters.	44
4.13	Output GUI labels.	45
4.14	Image simulation axes display.	46
4.15	1-D simulation axes display.	47

5.1	Matrix multiplication latency.	50
5.2	Matrix multiplication working frequency.	50
5.3	Matrix transposer latency.	51
5.4	Matrix transposer maximum working frequency.	52
5.5	Matrix inversion latency at different word lengths.	53
5.6	Maximum frequency comparison.	55
5.7	CoSaMP architecture Latency comparison at different sparse levels.	56
5.8	CoSaMP blocks latency for a signal with s=9 and 36 bits word length. Divided in a) pre-pseudoinversion, b) pseudoinversion and c) post-pseudoinversion.	57
5.9	CoSaMP architecture simulation.	57
5.10	Lena testing image.	58
5.11	Pacman testing image.	58
5.12	Transform comparison.	59
5.13	Reconstruction error.	59
5.14	Haar transform for different threshold level.	60
5.15	64 pixels lena testing image at 1.5 scale.	60
5.16	Haar trasform for different images.	61

List of Tables

2.1	CoSaMP Algorithm	10
2.2	QRD-CGS	15
2.3	QRD-MGS	16
2.4	R inversion using back substitution	16
3.1	Matrix Inversion Algorithm	22
3.2	Optimized CoSaMP Algorithm	31
5.1	Matrix Multiplication Xilinx Virtex 4	49
5.2	Matrix Multiplication Xilinx Spartan 6	49
5.3	Matrix Transposer Xilinx Virtex 4	50
5.4	Matrix Transposer Xilinx Spartan 6	51
5.5	Matrix inversion Dsp blocks usage	52
5.6	Matrix inversion and CoSaMP in Xilinx Virtex 4	53
5.7	Matrix Inversion and CoSaMP in Xilinx Spartan 6	54
5.8	Matrix Inversion and CoSaMP in Altera Cyclone II	54
5.9	Matrix Inversion and CoSaMP in Altera Cyclone IV	54

Chapter 1

Introduction

In the middle of a digital revolution, a plethora of sensing systems has been developed, dealing with the compromise between data size, resolution, and quality. Traditionally, the sampling rate of acquisition systems follows the mathematical analysis established by Nyquist and Shannon [3, 4] in the so called sampling theorem. Derived from their work, it is widely known that for the reconstruction of a signal with a bandwidth F , the sampling frequency F_s must be at least two times the signal bandwidth, $F_s \geq 2F$. When a signal is acquired and digitized considering the Nyquist-Shannon theorem, dealing with large data sets becomes a challenge.

However, some devices used for signal acquisition are physically impossible to implement. In 1999 Walden [5] presented a survey on Analog to Digital Converters (ADC) in which he reports that the maximum nyquist sampling rate attained was 8 Giga samples per second. Nevertheless, when the sampling frequency is duplicated the ADC resolution decreases one bit. Despite there were high sampling frequencies, bit resolution was sacrificed, as at 8 GHz just 3 bits of resolution were achieved.

Due to the big amount of data, it is often necessary to compress a signal that has been acquired according to the specifications of the Nyquist-Shannon theorem. Transform Coding is a popular technique that aims to find a base or frame where the signal has a sparse or compressible representation [6]. A signal of length N is k -sparse if it can be represented with k elements, $k \ll N$, and is compressible if the signal can be well approximated with those elements, even when a small amount of information may be lost. Some signal formats that exploit transform coding concept are: JPEG, JPEG2000, MPEG on images, MP4 and AVI on video and MP3 on audio.

In the past few years a new sensing technology has appeared. Donoho in 2006 [7], coined the term compressed sensing to this technology. The field of compressed sensing

has grown from the work of Candés, Romberg and Tao and Donoho, where they showed that a finite dimension signal, having a sparse or compressible representation can be reconstructed from a small set of linear and non-adaptive measurements [7, 8].

Compressed Sensing aims to acquire a compressed signal representation by a series of direct measurements in some domain where the signal may have a sparse representation, without going through the usual stage of acquiring N samples. The measurements are acquired by computing $M < N$ inner products between a signal of interest x and a sensing vector collection $\{\phi\}_{j=1}^M$ as in $y_j = \langle x, \phi_j \rangle$. The measurement vectors ϕ_j^M are arranged in an $M \times N$ matrix Φ [2].

Having less equations than unknowns, the goal of compressed sensing is to solve an underdetermined linear system to find the sensed signal x . This can be expressed as in (2.1).

While a compressed signal has been acquired and the data sets to deal with are smaller, a special effort must be done on the signal reconstruction. Unlike the reconstruction of an acquired signal considering Nyquist-Shannon sampling theorem, compressed sensed signals cannot be reconstructed with a simple interpolation of the measurements. Hence, several compressed sensing recovery algorithms have been developed, such as l_1 minimization, combinatorial algorithms and *Greedy* algorithms [9].

l_1 minimization is a convex optimization problem aiming to find the minimization of the sparse signal representation l_1 norm. Despite these compressed sensing recovery algorithms are a powerful tool they are not suitable for hardware implementation [9].

Combinatorial algorithms are another option to reconstruct a compressed sensed signal. Most of these algorithms are predate the compressed sensing literature. In some applications, as computation on data streams, combinatorial algorithms were used to recover x from $y = \Phi x$, essentially the same as the sparse recovery problem in compressed sensing [9].

Greedy algorithms are the most commonly used for implementation on hardware. These algorithms are a set of methods that iteratively construct an approximation of the sensed signal. Starting from a zero vector, a set of nonzero elements is estimated adding new elements on every iteration. Such algorithms often produce a fast convergence and can be applied to large data sets [9, 10].

In the Greedy algorithms it is required to solve a matrix inversion. There are two implemented methods that are frequently used to find a matrix inversion in hardware, CORDIC algorithm [11, 12] and QR decomposition (QRD) [13–15]. Besides, there are

iterative matrix inversion methods such as the Newton or Schulz method [16, 17] and the recently proposed Chebyshev-type method [18].

Chebyshev-type matrix inversion method was proposed by Amat et al. in 2003 [19]. In this work this matrix inversion method has been used in the context of compressed sensing.

1.1 Compressed Sensing Applications

Compressed sensing is a promising technology capable to give a new approach for sensing systems design, improving their performance. CS has been incorporated to several application fields, such as the MRI (Magnetic Resonance Imaging), sub-nyquist sampling systems, sensor networks, face recognition, and texture detection [9].

Duarte et al. [1] introduced a single-pixel camera using compressed sensing theory. They replaced CCD or CMOS sensor for one photon detector, a digital micro-mirror device (DMD), two lenses and an analog-to-digital (A/D) converter. The new device is capable to reconstruct 256x256 pixels images from about 1300 measurements. Figure 1.1 shows the reconstructed image.

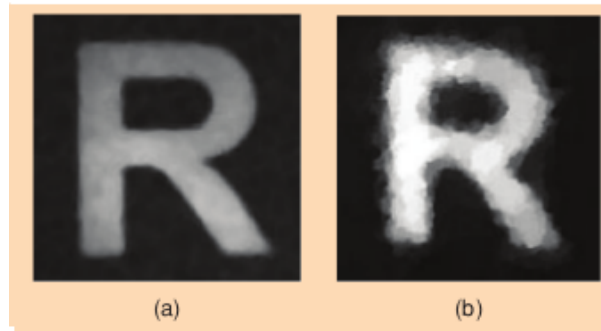


FIGURE 1.1: a) Original image. b) 256x256 image, reconstructed from 1300 measurements [1]

Compressed sensing has been used to improve the Magnetic Resonance Imaging (MRI) process. This is achieved increasing imaging speed [20]. Vasanawala in 2010 [21] made a comparison of two MRI techniques: parallel imaging and MRI using compressed sensing. Images acquired by compressed sensing technique were preferred more often; they had a significantly higher image quality rating and greater delineation of anatomic structures than did images obtained with the traditional parallel reconstruction method.

First sub-nyquist sampling hardware system was implemented in 2011 by Mishali et al. [22]. This system supports input signals up to 2 GHz with a sample rate of 280 MHz.

A CMOS architecture with built-in single-shot compressed sensing was design by Oike and Gamal in 2013 [23]. The image sensor can operate in compressed sensing mode with compression ratios $1/4$, $1/6$ or $1/8$ at 480, 960 or 1920 fps, respectively; or in normal capture mode with no compression at a frame rate of 120 fps.

In 2011 Balouchestani [24] proposed a novel approach for wireless sensor network nodes testing based on compressed sensing theory aiming to increase reliability. That year he presented a low power wireless network design also using compressed sensing [25]. In both cases improvements were due to reduction on the number of bits used for data transmission on the sensor network.

The low performance and high cost of the infrared (IR) photo detectors have prevented the widespread utilization of IR cameras on various fields. Carbon nanotube (CNT) has excellent optical properties that can be used for IR images sensors. However, it is difficult to fabricate a CNT sensor array. To improve IR sensors performance with CNT-based sensors and overcome the fabrication issue, Hongzhi et al. [26] presented a single nano-photodetector IR camera based on compressed sensing.

Literature review demonstrates the relevance of the compressed sensing nowadays. Thus, in this thesis a greedy algorithm named Compressed Sampling Matching Pursuit (CoSaMP) has been implemented on an FPGA architecture using the Chebyshev-typed method for solving the matrix inversion problem. Additionally a MATLAB Graphical User Interface (GUI) has been developed as an educational tool for Compressed Sensing theory exploration.

This thesis is organized as follows: Chapter 1 presents an introduction and motivation of this work. Chapter 2 summarizes the theory associated to compressed sensing in the context of the Nyquist theorem and Transform Coding. Chapter 3 limns two VHDL architectures: matrix inversion architecture and a Compressed Sampling Matching Pursuit algorithm architecture. In Chapter 4 a MATLAB Graphical User Interface for compressed sensing theory exploration is described. Chapter 5 presents obtained results and discussion. Conclusions of the work are presented in Chapter 6.

Chapter 2

Theoretical Fundamentals

2.1 Compressed Sensing

Compressed sensing aims to acquire a compressed signal representation by a series of direct measurements in some domain where the signal may have a sparse representation, without going through the usual stage of acquiring N samples. The measurements are acquired by computing $M < N$ inner products between a signal of interest x and a collection $\{\phi\}_{j=1}^M$ as in $y_j = \langle x, \phi_j \rangle$. The measurement vectors ϕ_j^M are arranged in an $M \times N$ matrix Φ [2].

Having less equations than unknowns, the goal of Compressed Sensing is to solve an underdetermined linear system to find the sensed signal x . This can be expressed as in (2.1).

$$y = \Phi x = \Phi \Psi s \tag{2.1}$$

Where:

y measurements and $y \in R^M$,

Φ sensing matrix and $\Phi \in R^{M \times N}$,

Ψ sparse base or frame $R^{N \times N}$,

x sensed signal and $x \in R^N$,

s sparse signal representation.

If, however, signal x has a sparse representation s in a basis Ψ , it can, under certain conditions, be recovered from measurements y . A graphical representation of compressed sensing can be seen in figure 2.1.

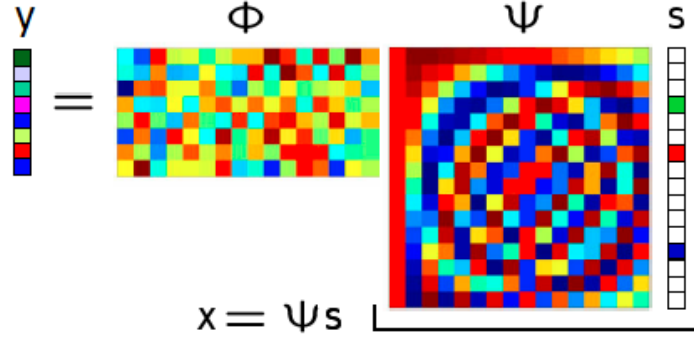


FIGURE 2.1: Graphical representation of compressed sensing concept [2].

To reconstruct the sensed signal from the taken measurements y is required to solve for s in equation (2.1); re-written as (2.2).

$$s = (\Phi\Psi)^{-1}y \quad (2.2)$$

Signal sparse representation s must be multiplied by the orthogonal frame in order to recover signal of interest, equation (2.3).

$$x = \Psi(\Phi\Psi)^{-1}y \quad (2.3)$$

One of the limitations of CS is that it is strictly necessary to find a sparse frame in order to apply compressed sensing theory for sensing a signal. Recovery problem requires $M = N$ to be well-posed so it can be solved from equation (2.2). Therefore, if $M < N$ the recovery problem becomes ill-posed. This missing link is found by adding a prior information of the sensed signal x , that is the signal sparsity or sparse level k [27].

Compressed sensing reconstruction algorithms take the known sparsity level and keep track of the k -nonzero elements or k -larger elements of the signal.

2.1.1 Sensing Matrix

Sensing matrix construction for compressed sensing is an important matter. There are two main theoretical questions in CS. First, how should we design the sensing matrix Φ

to ensure that it preserves the information in the signal x ? Second, how can we recover the original signal x from measurements y ? [9].

Sensing matrix Φ can be designed in such a way that it will be able to reconstruct signal s from measurements y whether signal has a sparse representation. Φ matrix structure must satisfy a set of desirable properties.

2.1.1.1 Null Space Condition

The null space of Φ is denoted by

$$\mathcal{N}(\Phi) = \{x : \Phi x = 0\} \quad (2.4)$$

For an underdetermined linear system there are an infinite number of solutions. It is clear that for any pair of distinct vectors x, x' we must have $\Phi x \neq \Phi x'$, since otherwise it would be impossible to distinguish x from x' with solely measurements y [9]. A common way for characterizing the Null Space property is known as the spark [28].

Spark of a given matrix Φ is defined as the smallest number of columns of Φ that are linearly independent. In order to fulfill null space condition, spark of matrix Φ must be greater than $2 \times k$, this is :

$$\text{spark}(\Phi) > 2k \quad (2.5)$$

Where, k is the signal sparsity level.

2.1.1.2 Restricted Isometry Property

Null space property guarantees that signal x can be reconstructed from measurements y , but this guarantees do not account the presence of noise. In [29] Candés and Tao introduced the *Restricted Isometry Property (RIP)* with which a signal in the presence of noise can be correctly reconstructed.

A matrix Φ satisfies the k -order restricted isometry property, if there exist a $\delta \in (0, 1)$ such that

$$(1 - \delta)\|x\|_2^2 \leq \|\Phi x\|_2^2 \leq (1 + \delta)\|x\|_2^2 \quad (2.6)$$

There are two types of sensing matrices, deterministic [30, 31] and random matrices. Deterministic matrices are more difficult to construct, but their RIP can easily be computed and they can be constructed on the fly. Random matrices have two main drawbacks. First, they may require a lot of storage. Second, there is no efficient algorithm for testing the RIP [32]. However, Gaussian random matrices fulfill, with high probability, the restricted isometry property condition and can be used as sensing matrices [2].

2.1.2 Reconstruction Algorithms

After signal x has been sensed and measurements are stored in vector y an endeavor to recover the signal has to be made. There are a variety of algorithms that have been used in applications such as sparse approximations, statistics and theoretical computer science that were developed to exploit sparsity in other contexts and can be brought to bear on the CS recovery problem [9, 33]. There are two main categories for signal recovery in compressed sensing: l_1 minimization and *greedy algorithms*.

2.1.2.1 l_1 minimization

Minimization approach is a convex optimization problems aiming to find the minimization of a variable subject to one or more conditions. For compressed sensing the l_0 norm was the first attempt to recover x by solving the optimization problem of the form

$$\hat{x} = \operatorname{argmin} \|x\|_0 \quad \text{subject to} \quad \Phi x = y \quad (2.7)$$

Where $\|x\|_0$ is the nonzero entries of x .

This is with the previous knowledge that measurements y are from a highly sparse signal. l_0 norm has the inconvenience that is a combinatorial problem with prohibitive complexity if solved numerically [34]. An alternative to overcome this problem is to replace l_0 norm by l_1 norm and solve a computationally tractable model given by (2.8).

$$\hat{x} = \operatorname{argmin} \|x\|_1 \quad \text{subject to} \quad \Phi x = y \quad (2.8)$$

The l_1 minimization has the advantage of giving an uniform reconstruction. However, is still a too complex problem for real time recovery and is not suitable for hardware implementation [13].

2.1.2.2 Greedy algorithms

Besides l_1 minimization, greedy algorithms are an alternative for compressed sensing signal reconstruction. Most common greedy algorithms are the Matching Pursuit MP[35] (known as pure greedy algorithm), Orthogonal Matching Pursuit OMP[36], Stage wise Orthogonal Matching Pursuit StOMP, Compressed Sampling Matching Pursuit CoSaMP [37], Gradient Pursuit GP [38] and Conjugate Gradient Pursuit [38].

In 2012 Lifeng Du et al. [10] presented a greedy algorithms analysis. Based on their results, they concluded that Orthogonal Matching Pursuit, Stage wise Matching Pursuit and Compressed Sampling Matching Pursuit have a better performance due to significantly smaller error in the case of a small sparsity or more measurements. Signal reconstruction error was the least for CoSaMP algorithm shown in Table 2.1.

2.1.2.3 CoSaMP Algorithm

The algorithm starts with a trivial initial guess $a = 0$. During each iteration, CoSaMP performs five major steps [37].

- ① *Identification*. The algorithm forms a proxy of the residual from current samples and locates the largest elements of the proxy.
- ② *Support Merger*. In first iteration support merger is not used. On following iterations, it merges the support of the current signal approximation and the newly identified components, this is $T = \Omega \cup \text{supp}(a^{i-1})$
- ③ *Estimation*. The algorithm solves a matrix inversion problem in order to approximate the target signal on the merged components on support set T . In this step is required to find the pseudoinverse (\dagger) of a full rank tall matrix. A submatrix Φ_T is constructed with columns of Φ which index number is in the support set T .
- ④ *Pruning*. The algorithm produces a new approximation by retaining only the largest entries in the least-squares signal approximation.
- ⑤ *Sample Update*. Finally the samples are updated, so that they reflect the residual, the part of the signal that has not been approximated.

The mentioned algorithm steps are repeated until halting condition is met. Compressed Sampling Matching Pursuit algorithm is described in Table 2.1.

TABLE 2.1: CoSaMP Algorithm

Input: Sampling matrix Φ ; Sample data y ; Sparsity level k
During the 1^{st} iteration
As $a^{(0)} = 0$ and $r = y$ $u = \Phi^T r$ ① $T = \text{supp}(u_{2k})$ $b _T = (\Phi _T)^\dagger y$ ③ $b _{T^c} = 0$ $a^{(1)} = b_k$ ④ $r = y - \Phi a^{(1)}$ ⑤
During the i^{th} iteration
$u = \Phi^T r$ ① $\Omega = \text{supp}(u_{2k})$ $T = \Omega \cup \text{supp}(a^{i-1})$ ② $b _T = (\Phi _T)^\dagger y$ ③ $b _{T^c} = 0$ $a^{(1)} = b_k$ ④ $r = y - \Phi a^{(1)}$ ⑤
Output
An s-sparse approximation a of the target signal.

2.2 Matrix Inversion

Matrix inversion is a core issue for reconstructing a compressed sensed signal. Greedy algorithms require to compute matrix inversion of a non-square matrix also called Moore-Penrose matrix inversion denoted by Φ^\dagger . Three ways to cope with the matrix inversion problem are: analytical, Chebyshev-typed method and QR decomposition.

2.2.1 Moore-Penrose Inverse

The Moore-Penrose inverse of a matrix $A \in R^{m \times n}$, denoted by A^\dagger , is a matrix X that satisfies the following four Penrose equations [39]

$$AXA = A \tag{2.9}$$

$$XAX = X \quad (2.10)$$

$$(AX)^* = AX \quad (2.11)$$

$$(XA)^* = XA \quad (2.12)$$

Where A^* is the conjugate transpose of A .

If above equations are fulfilled by matrix A and its pseudoinverted matrix $X = A^\dagger$, Moore-Penrose inversion will be given by equations (2.13) and (2.14).

$$A^\dagger = (A^T A)^{-1} A^T \quad (2.13)$$

$$A^\dagger = A^T (A A^T)^{-1} \quad (2.14)$$

Equations (2.13) and (2.14) are the left side and right side Moore-Penrose pseudoinversion. A rectangular matrix cannot have a two sided inverse as either that matrix or its transpose has a non-zero null space, this can be stated as follow.

- $A^T A$ is invertible when A has a full column rank.
- $A A^T$ is invertible when A has a full row rank.

The column rank of a matrix is the size of the collection of all linearly independent columns. The row rank is the size of the collection of all linearly independent rows. For any given matrix row rank equals column rank.

For a matrix $A \in R^{m \times n}$, if $m = n$ then $A^\dagger = A^{-1}$.

2.2.2 Analytical Inversion

A well-known method for finding matrix inversion is the analytical method that uses the adjoint matrix. Matrix inversion is given by equation (2.15)[40].

$$A^{-1} = \frac{1}{\det A} \text{adj} A \quad (2.15)$$

Where:

$adj A$ is the adjoint matrix,

$det A$ is the matrix determinant.

2.2.2.1 Adjoint matrix.

Given a matrix $A(a_{ij})$ spanned by its a_{ij} elements, whose cofactor matrix is $B(A_{i,j})$, adjoint matrix will be (2.18).

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \quad (2.16)$$

$$B_{m,n} = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{pmatrix} \quad (2.17)$$

$$adj A = B' = \begin{pmatrix} A_{1,1} & A_{2,1} & \cdots & A_{m,1} \\ A_{1,2} & A_{2,2} & \cdots & A_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,n} & A_{2,n} & \cdots & A_{m,n} \end{pmatrix} \quad (2.18)$$

2.2.2.2 Cofactor

The cofactor ij of a given matrix A , denoted as A_{ij} can be computed using equation (2.19).

$$A_{ij} = (-1)^{i+j} |M_{ij}| \quad (2.19)$$

Where, $|M_{ij}|$ is the determinant of the minor matrix of A , formed by taking out row i and column j from A .

2.2.3 Chebyshev-type Method

In 2003, Amat et. al. [19] introduced an iterative Chebyshev-type method of third order or cubic convergence to find the inverse of a given matrix.

Few years later in 2011, Hou-Biao Li et. al.[18] compared the method proposed by Amat with the iterative Newton method and demonstrated that Chebyshev-type method has less computational complexity and needs a small number of iterations to find the solution. In that work they suggested a preconditioning technique for the initial guess of the method to ensure that the method will converge.

2.2.3.1 Mathematical Formalization

The mathematical formalization of the Chebyshev-type method is given by (2.20).

$$N_{m+1} = N_m(3I - AN_m(3I - AN_m)) \quad (2.20)$$

Where:

N_{m+1} next inverse approximation,

N_m previous inverse approximation,

I identity matrix,

A matrix to be inverted.

2.2.3.2 Preconditioning

In order to find the solution to the inverse matrix problem through the Chebyshev-type method, it is important to choose a suitable initial guess, otherwise the method diverges.

With the equation (2.21) a suitable initial guess can be computed to ensure the method's convergence.

$$N_0 = \frac{A^T}{||A||_1 ||A||_\infty} \quad (2.21)$$

Where:

N_0 initial guess,

A matrix to be inverted,

A^T transpose of A ,

$\|A\|_1$ max value of the summation of the elements on each column (2.22),

$\|A\|_\infty$ max value of the summation of the elements on each row (2.23).

$$\|A\|_1 = \max_j \left\{ \sum_{i=1}^n |a_{ij}| \right\} \quad (2.22)$$

$$\|A\|_\infty = \max_i \left\{ \sum_{j=1}^n |a_{ij}| \right\} \quad (2.23)$$

2.2.3.3 Word length

Word length is an important issue on designing an FPGA architecture; for an iterative matrix inversion is defined in the preconditioning stage, equation (2.21).

Matrix A with size n and elements ranging in $[0 - c]$ interval will give, in the preconditioning stage, a maximum number to be represented and can be computed with equation (2.24).

$$\max_n = \|A\|_1 * \|A\|_\infty = (cn)^2 \quad (2.24)$$

The minimum number greater than zero to be represented is:

$$\min_n = \frac{\min(A)}{(cn)^2} \quad (2.25)$$

Where:

$\min(A)$ is the minimum number greater than zero contained in A .

If \min_n is greater than zero, then \min_n must be multiplied by a constant d in order to fulfill inequality in (2.26). The Word length will be defined by equation (2.27).

$$2 > \min_n * d \geq 1 \quad (2.26)$$

$$\text{word_length} = 2 * (\text{bits_d} + \text{bits_max_n}) \quad (2.27)$$

Where:

$bits_d$, number of bits to represent d ,

$bits_max_n$, number of bits to represent max_n .

2.2.4 QR Decomposition

QR Decomposition is a method where matrix A is decomposed onto two matrices Q and R (2.28) [41]. Matrix inversion using QR decomposition can be computed by equation (2.29).

$$A = QR \quad (2.28)$$

$$A^{-1} = R^{-1}Q^T \quad (2.29)$$

Two algorithms to perform QR decomposition are the Classical Gram-Schmidt (QRD-CGS) and the Modified Gram-Schmidt (QRD-MGS). The QRD-CGS has a round off error when using a fixed point calculation. QRD-MGS overcomes this issue and is numerically and accuracy superior to CGS [42]. QRD-CGS algorithm is shown in table 2.2 and QRD-MGS algorithm in table 2.3.

TABLE 2.2: QRD-CGS

<p>For $j = 1 : n$; $w_j = A_j$ for $i = 1 : (j - 1)$; $R_{i,j} = \langle A_j, Q_i \rangle$ $w_j = w_j \setminus R_{i,j}Q_i$ end $Q_j = \frac{w_j}{ w_j _2}$ $R_{j,j} = w_j _2$</p>
--

Where, one index as in A_j or Q_i means the j or i column, two indices like $R_{i,j}$ indicates an element of R and w_j is a temporary vector.

TABLE 2.3: QRD-MGS

For $j = 1 : n$; $w_j = A_j$ for $i = 1 : (j - 1)$; $R_{i,j} = \langle w_j, Q_i \rangle$ $w_j = w_j \setminus R_{i,j} Q_i$ end $Q_j = \frac{w_j}{\ w_j\ _2}$ $R_{j,j} = \ w_j\ _2$

In (2.29) R^{-1} is calculated using back substitution [42]. Computation algorithm of R^{-1} can be seen in table 2.4.

TABLE 2.4: R inversion using back substitution

For $j = 1 : n$; For $i = 1 : (j - 1)$ $iR_{i,j} = iR(i, 1 : (j - 1)) * R(1 : (j - 1), j)$; end $iR_{1:(j-1),j} = \frac{-iR_{1:(j-1),j}}{R_{j,j}}$ $iR_{j,j} = \frac{1}{R_{j,j}}$ end

Where, iR is the inverted matrix.

2.3 Transforms

For a signal to be suitable for compressed sensing it is necessary to find a sparse representation or domain. Therefore, transforms are used for this duty. Three common used transforms are: Discrete Fourier Transform (DFT), Wavelet Transform (WT) and discrete cosine transform (DCT).

2.3.1 Discrete Fourier Transform

Fourier analysis gives frequency information about a signal. Some signals present sparsity at the frequency domain, as example: sinusoids. Figures 2.2 and 2.3 show how periodic signal are highly sparse in the frequency domain applying DFT .

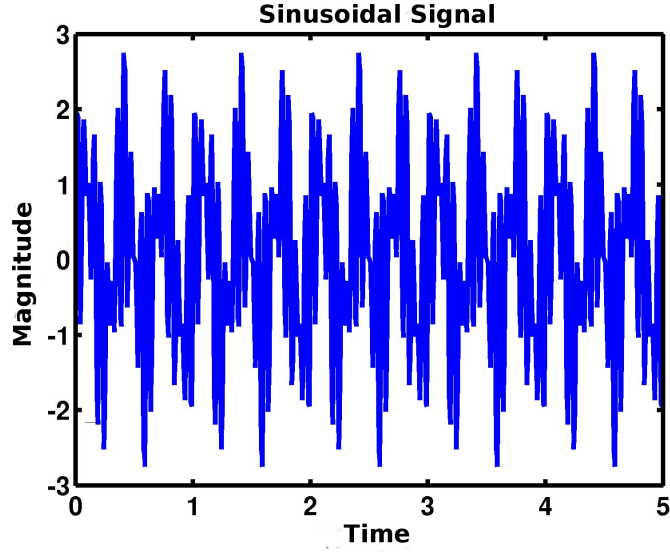


FIGURE 2.2: Sinusoidal signal.

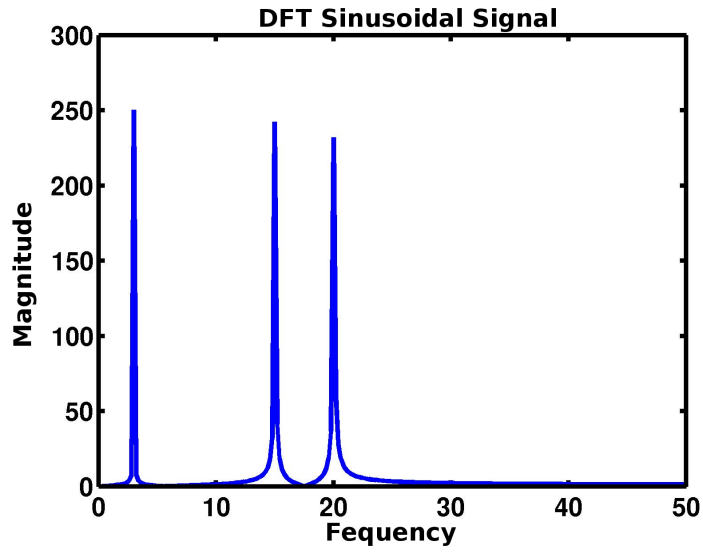


FIGURE 2.3: DFT of sinusoidal signal from figure 2.2.

DFT and its inverse IDFT are given by equations (2.30) and (2.31). The two equations give a numerical algorithm to obtain the frequency response of $x(n)$.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j(\frac{2\pi}{N})kn} = \sum_{n=0}^{N-1} x(n) W^{kn}, \quad (2.30)$$

$$0 \leq k \leq N - 1$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j(\frac{2\pi}{N})kn} = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W^{-kn}, \quad (2.31)$$

$$0 \leq n \leq N - 1$$

Direct computation of DFT and IDFT using (2.30) and (2.31) requires N^2 multiplications and $(N - 1)^2$ additions; so they become very large numbers when N is chosen very large, in order to increase the resolution of the frequency response $X(k)$ of a given signal.

Fast Fourier Transform is an algorithm to compute the Discrete Fourier Transform and its inverse. FFT improves computational efficiency, when the value for the radix N is chosen as 2^R , where R is an integer, the number of multiplication is of order $(N/2)\log_2(N)$ and the number of additions is reduced to $N\log_2 N$ [43].

2.3.2 Wavelet Transform

The wavelet transform can be used as yet another way to describe the properties of a waveform that changes over time, but in this case the waveform is divided not into sections of time, but segments of scale.

In wavelet analysis a variety of different probing functions may be used, but the family always consists of enlarged or compressed version of the basic function, as well as translations. This lead to the equation for the continuous wavelet transforms (CWT):

$$W(a, b) = \int_{-\infty}^{\infty} x(t) \frac{1}{\sqrt{|a|}} \Psi * \left(\frac{t - b}{a} \right) dt \quad (2.32)$$

Where b acts to translate the function across $x(t)$ and a varies the time scale of the probing function, Ψ . Wavelet coefficients describe the correlation between the waveform and the wavelet function various translations and scales. If the mother wavelet or wavelet function, $\Psi(t)$, is appropriately chosen, then it is possible to reconstruct the original waveform from wavelet coefficients [44].

Time range of a wavelet function, δt_{Ψ} , can be specified by the square root of the second moment of a given wavelet about its time center, equation (2.33). The center time t_0 is given by equation (2.34).

$$\delta t_{\Psi} = \sqrt{\frac{\int_{-\infty}^{\infty} (t - t_0)^2 |\Psi(t/a)|^2 dt}{\int_{-\infty}^{\infty} |\Psi(t/a)|^2 dt}} \quad (2.33)$$

$$T_0 = \frac{\int_{-\infty}^{\infty} t |\Psi(t/a)|^2 dt}{\int_{-\infty}^{\infty} |\Psi(t/a)|^2 dt} \quad (2.34)$$

Frequency range is given by equation (2.35) around its center frequency given by (2.36)

$$\delta\omega_{\Psi} = \sqrt{\frac{\int_{-\infty}^{\infty} (\omega - \omega_0)^2 |\Psi(\omega)|^2 d\omega}{\int_{-\infty}^{\infty} |\Psi(\omega)|^2 d\omega}} \quad (2.35)$$

$$\omega_0 = \frac{\int_{-\infty}^{\infty} \omega |\Psi(\omega)|^2 d\omega}{\int_{-\infty}^{\infty} |\Psi(\omega)|^2 d\omega} \quad (2.36)$$

CWT is highly redundant having many more coefficients than needed to represent a signal. This becomes a problem due to high computational cost for signal recovery. The *discrete wavelet transform* (DWT) overcomes this issue by restricting translation and scale variations, usually to powers of 2 [44].

DWT can be performed using equations (2.37) and (2.38). Where ϕ is the scaling function, $c(n)$ is a series of scalars that defines the specific scaling function and $d(n)$ is a series of scalars that are related to the waveform $x(t)$ and define the discrete wavelet in terms of the scaling function. Relationship of $d(n)$ and signal $x(t)$ can be seen in the inverse discrete wavelet transform equation (2.39) [44].

$$\phi(t) = \sum_{n=-\infty}^{\infty} \sqrt{2} c(n) \phi(2t - n) \quad (2.37)$$

$$\Psi(t) = \sum_{n=-\infty}^{\infty} \sqrt{2} d(n) \phi(2t - n) \quad (2.38)$$

Inverse DWT is defined by equation 2.39.

$$x(t) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} d(k, l) 2^{-k/2} \Psi(2^{-k}t - l) \quad (2.39)$$

Here k and l are related to parameter a and b of equation 2.32 as: $a = 2^k$, $b = 2^k l$. $d(k, l)$ is a sampling of the continuous wavelet coefficients, $W(a, b)$, at discrete points k and l

2.3.3 Discrete Cosine Transform

The discrete cosine transform (DCT) [45] has been an important processing tool for digital signals. Due to the sparsity levels it generates in signals, DCT is used in digital signals formats such as MP3 for audio and JPEG for images.

For a signal $x(m)$ $m = 0, 1, 2, \dots, (M - 1)$ DCT is defined as [45]:

$$G_x(0) = \frac{\sqrt{2}}{M} \sum_{m=0}^{M-1} x(m) \quad (2.40)$$

$$G_x(k) = \frac{\sqrt{2}}{M} \sum_{m=0}^{M-1} x(m) \cos \frac{(2m+1)k\pi}{2M}, \quad (2.41)$$

$$k = 1, 2, \dots, (M - 1)$$

Where $G_x(k)$ is the k th coefficient.

Two dimension DCT can be performed on an image with equation (2.42). Image is divided into blocks of 8x8 pixels and $x[m, n]$ represents the image pixel values in a block.

$$G[u, v] = \frac{C[u]C[v]}{4} \sum_{m=0}^7 \sum_{n=0}^7 x[m, n] \cos \frac{(2m+1)u\pi}{16} \cos \frac{(2n+1)v\pi}{16}, \quad (2.42)$$

$$0 \leq u, v \leq 7.$$

where

$$C[u] = \begin{cases} \frac{1}{\sqrt{2}} & u = 0 \\ 1 & 1 \leq u \leq 7 \end{cases}$$

The DCT, which belongs to the family of sinusoidal transforms, has received special attention because of its success in the compression of real-world images [46].

Chapter 3

FPGA implementation

A CoSaMP FPGA-based architecture is proposed for compressed sensed signal reconstruction. The matrix inversion process, required by the CoSaMP algorithm, is based on an iterative Chebyshev-type method. This chapter is divided into two main sections: matrix inversion architecture and CoSaMP architecture.

3.1 Matrix Inversion Architecture

In this section, the implementation of the Chebyshev matrix inversion algorithm, based on the mathematical formulation proposed in [19][18] and described on section 2.2.3, is presented. The algorithm has been divided, as shown in Table 3.1, into three stages: preconditioning stage, based on equation (2.21), iterative stage, based on equation (2.20) and verification stage, based on the premise of (3.1).

$$AA^{-1} = I \quad (3.1)$$

In the preconditioning stage, matrix A is transposed and $\|A\|_1$ and $\|A\|_\infty$ are calculated, using equations (2.22) and (2.23). The output of this stage is the initial guess, N_0 , of the matrix inversion, which is saved into a RAM memory.

The iterative stage has been divided into its simplest operations such as matrix multiplication and subtraction. Every step at this stage is saved into an embedded RAM memory.

Verification stage takes the output of the matrix multiplication, $A * N_m$ in step ① of the iterative stage, after ③ has been done so the multiplication is made between the

matrix A and the previously found inverse matrix approximation N_{m+1} . If the result meets the condition established in (3.1) a finish signal is sent to the control block and the inverse matrix $A^{-1} = N_{m+1}$ has been found.

TABLE 3.1: Matrix Inversion Algorithm

Input: Matrix A , Dimension of A .
Preconditioning
$\ A\ _1, \ A\ _\infty$ Transpose of A , $N_0 = \frac{A^T}{\ A\ _1 \ A\ _\infty}$
Iterative
① $3I - AN_m$ ② $3I - AN_m(3I - AN_m)$ ③ $N_{m+1} = N_m(3I - AN_m(3I - AN_m))$
Verification
When iterative stage computes $A * N_{m+1}$ in step ① if $(A * N_{m+1}) \approx I$ finish algorithm else $N_m = N_{m+1}$ iterative stage goes to ② end
Output: N_{m+1} as the inverted matrix A^{-1} .

3.1.1 System Structure

The developed architecture has been divided into the following main blocks: preconditioner, core, verifier, multiplexer, control, and storage. These blocks are described in detail in the next subsections.

Figure 3.1 graphically depicts the system structure and its composition blocks.

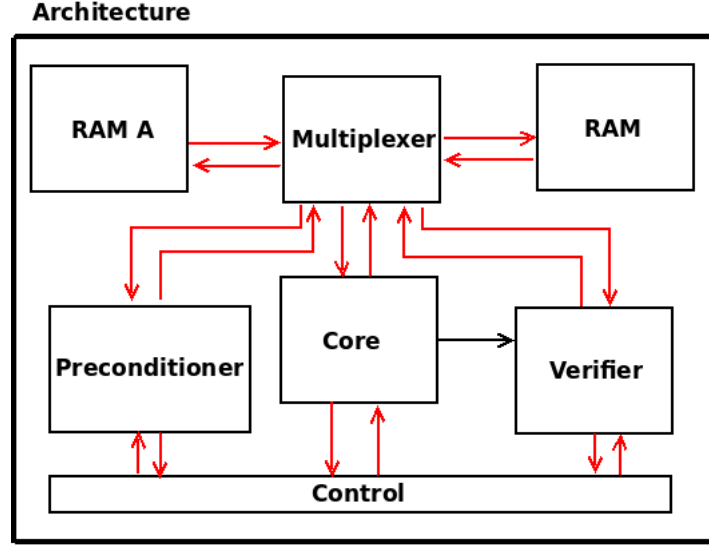


FIGURE 3.1: Matrix inversion system structure.

3.1.2 Preconditioner

Preconditioner blocks aim to give the first approximation to the inverse matrix A^{-1} , taking as input the elements of RAM A which contains matrix A . If initial guess is not appropriately chosen, inversion method will not converge to the solution. This block is constituted by three sub-blocks: max, transpose and division, as shown in figure 3.2.

Equations (2.22) and (2.23) are computed in the max sub-block with the max row and max column blocks. At the output of the max block the product $\|A\|_1 * \|A\|_\infty$ is obtained using a multiplier; this output is saved in a register and sent as input to the division sub-block.

Max block performs the addition of the elements in every column and row; the result is loaded in registers 1. When the addition of all elements in a column and the addition of all element in a row are finished, the comparator sub-block determines if current results are greater than a previous larger column and row elements addition stored in registers 2. If current results are greater they replace the data contained in registers 2, otherwise data is kept. A graphical description of the max block can be seen in figure 3.3.

The transpose sub-block has an input (columns) and output (rows) address generators, each one composed by two counters and one adder. Input address generator reads RAM A as columns and output address generator sends the data to be stored as rows; in that way the matrix A is transposed. Max and transpose sub-blocks are executed in parallel.

Once the transpose and Max sub-blocks are finished, the finite state machine sends a start signal to the division sub-block. Initially the division sub-block takes the output

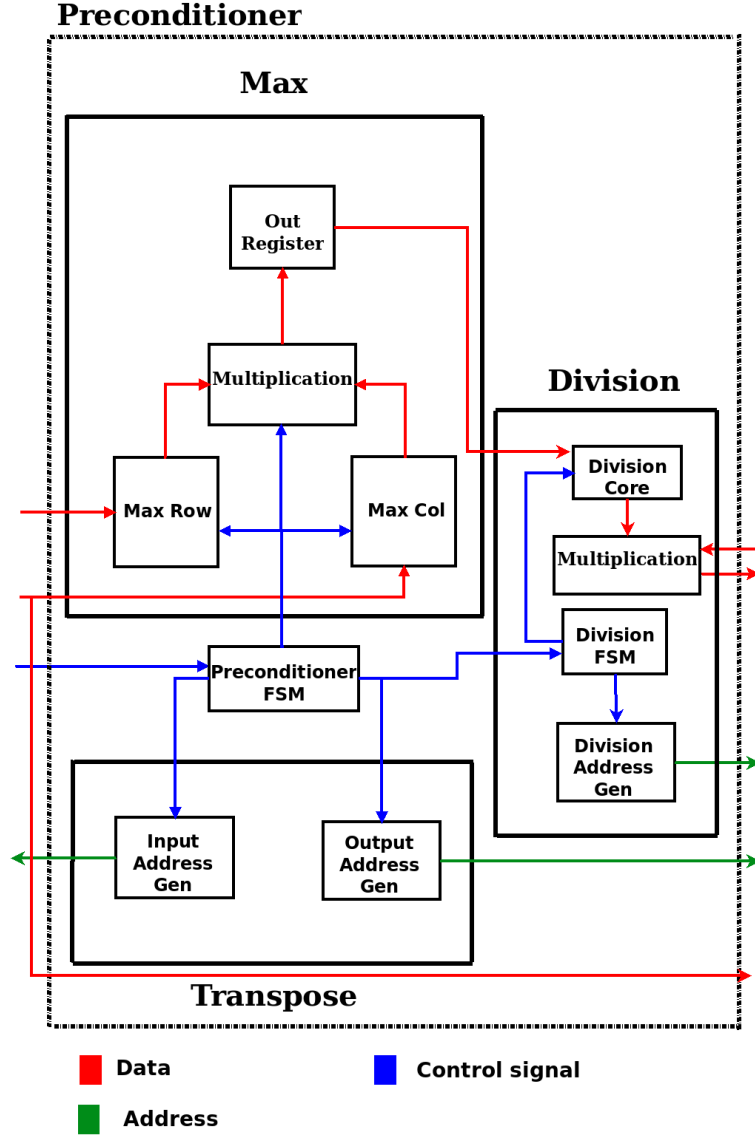


FIGURE 3.2: Preconditioner block diagram.

of the max sub-block, given by $\|A\|_1 \|A\|_\infty$, and performs the operation described in equation (3.2) to find *fact*. After *fact* has been found, the sub-block multiplies every element of A^T by *fact*.

$$fact = \frac{1}{\|A\|_1 \|A\|_\infty} \quad (3.2)$$

After this process is finished, the first approximation, N_0 , has been found and sent to memory. The block sends a signal to the control block indicating the core block to start the first iteration.

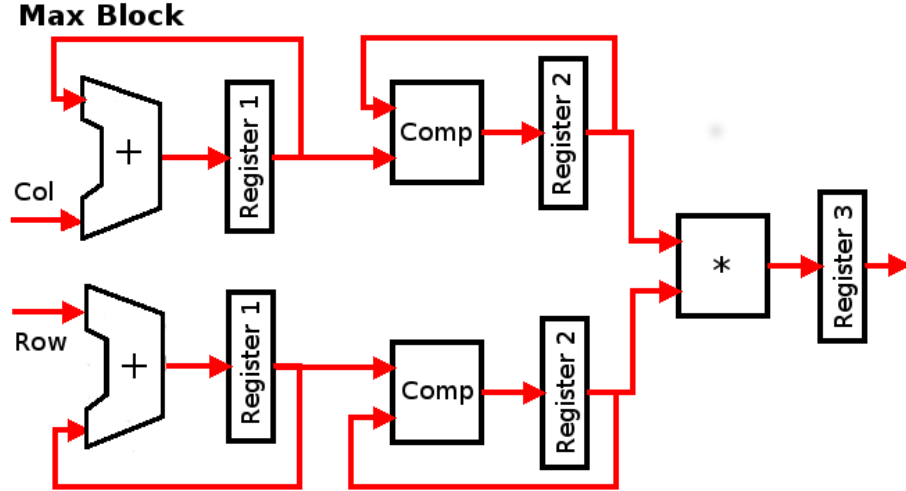


FIGURE 3.3: Max block structure.

3.1.3 Core

The core block is composed of two sub-blocks: matrix multiplication and subtraction. Based on those two sub-blocks it computes the next inverse matrix approximation, N_{m+1} from (2.20). The block distribution is shown in figure 3.4.

Multiplication sub-block is a generic Matrix-Matrix multiplicator. The architecture has a MAC (Multiply ACcumulate) structure and five address generators. The MAC is constituted by four parallel multipliers and one adder, so it computes the multiplication of eight elements, four of each matrix, at the same time. In [47] MAC is constituted just by one multiplier and one accumulator. Address generators are distributed as follow: two for reading row elements, two for reading column elements and one for generating the output address.

MAC data flow is shown in figure 3.5. First, registers 1 are loaded with the data to be multiplied; after one clock cycle, registers 2 load the other four elements to be multiplied. Multipliers have one cycle delay so the correct product of all four multipliers will be at accumulator inputs one cycle after registers 2 are loaded. Register 3 is storing accumulator output while size dim of multiplied matrix is reached; afterwards, register 4 loads result and sends it to the RAM memory.

An improvement in computational time with respect to a previous version reported in [47] was implemented, by designing a parallel structure to perform multiplication and subtraction operations in this block. This approach cuts down the number of steps in the inversion algorithm from six to three.

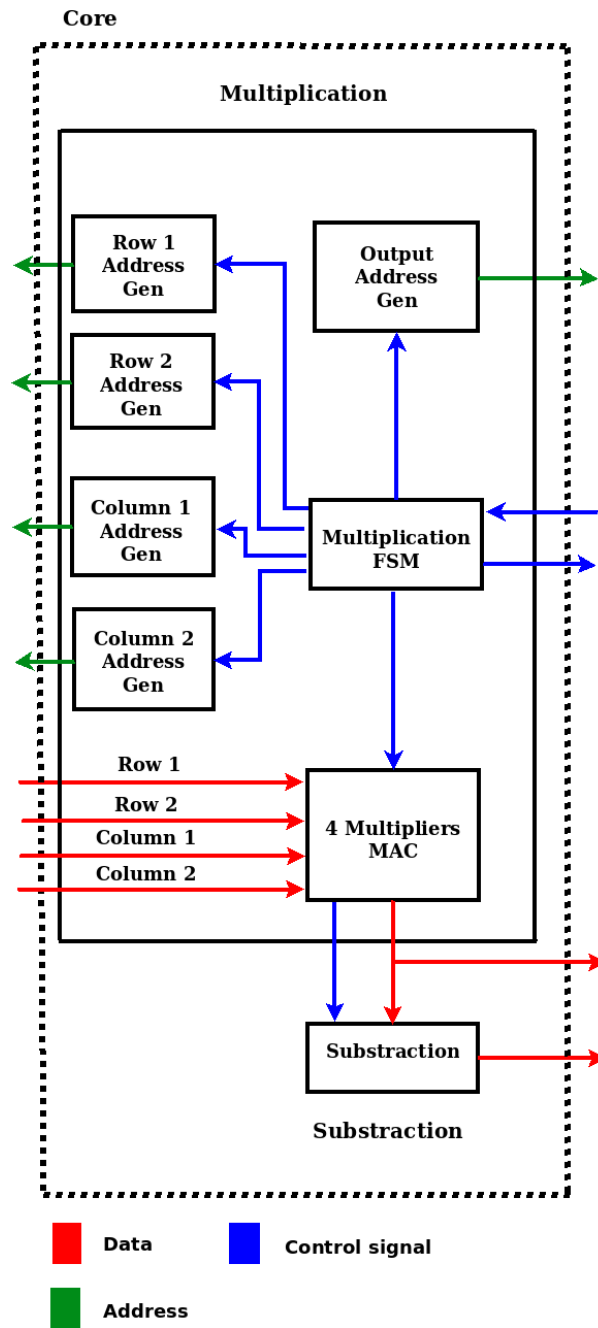


FIGURE 3.4: Core block diagram.

Subtraction is a combinatorial sub-block that takes the MAC output and does the subtraction given by $3I - MAC_dat_out$. The address where the result will be stored is the same as the MAC output address.

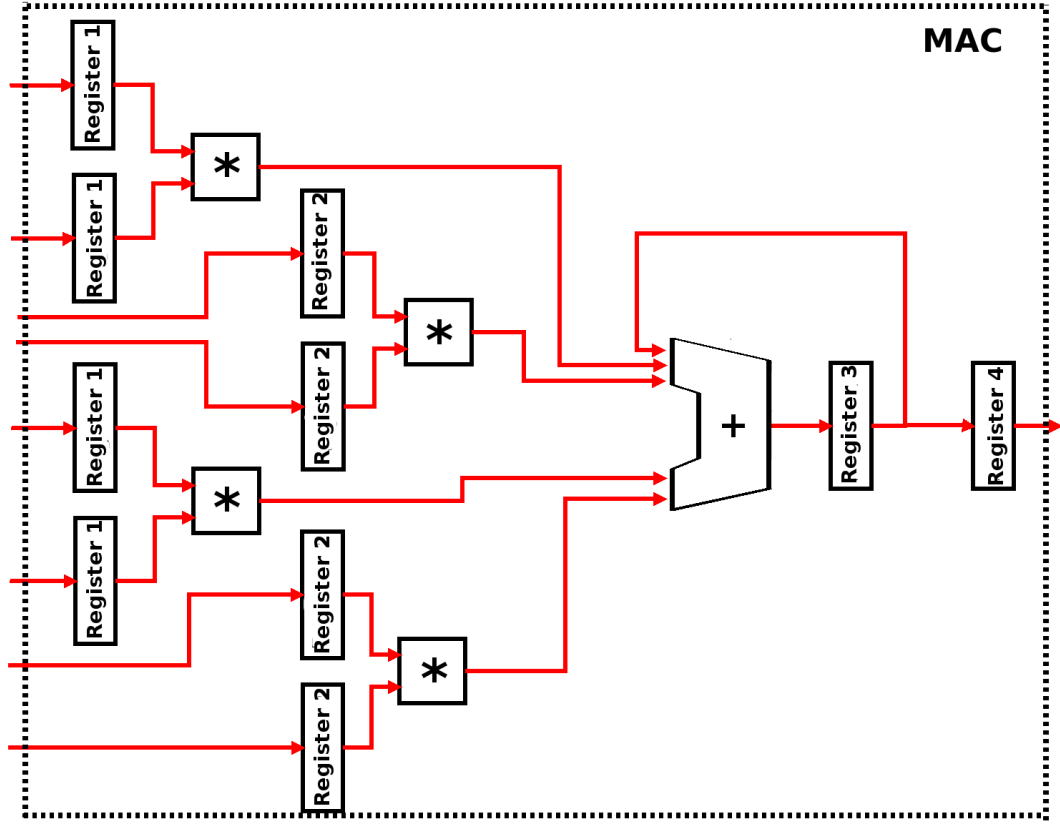


FIGURE 3.5: MAC data flow.

3.1.4 Verifier

Verifier block is constructed using a comparator, a counter and a finite state machine. While multiplication $A * M_n$ in the step ① of the iterative stage is being computed, verifier checks up if that multiplication gives as result an identity matrix.

For the verifier sub-block it is important to know if the multiplication result will be stored in or out the matrix diagonal; thus multiplication sub-block sends a signal called *Diag-sign* to the verifier. *Diag-sign* points out if the data under the verifying analysis is a matrix diagonal element. The elements in the matrix diagonal must be of value one.

The counter keeps track of wrong elements. Once the multiplication process finishes, the counter is checked in order to decide the next step; if the counter's output is zero a stop signal is sent and the inverted matrix given by $A^{-1} = N_{m+1}$ has been found. If the counter's output is greater than zero the iterative process continues through step ②.

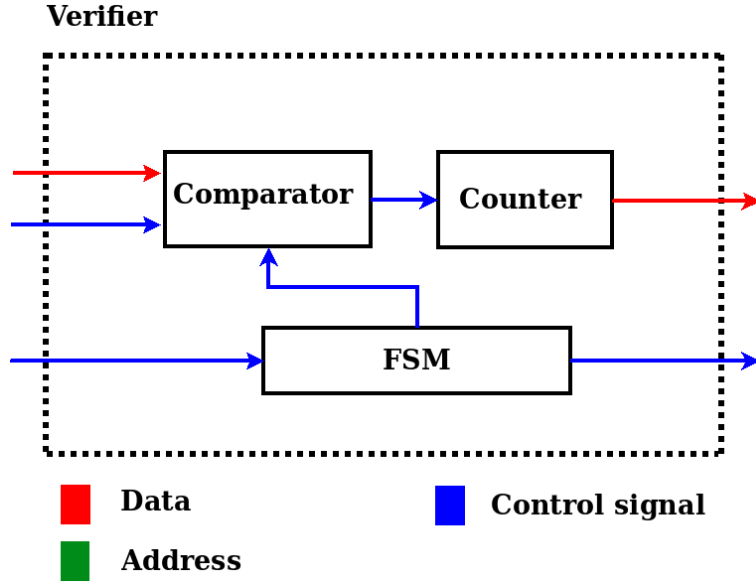


FIGURE 3.6: Verifier block diagram.

3.1.5 Multiplexer

The multiplexer block, consisting on several multiplexers and demultiplexers, addresses every block output data to the corresponding input of a block or memory. Multiplexers are important for reducing FPGA RAM resource utilization. RAM memory can be re-used by blocks that function at different times; this prevent to have repeated data. RAM contents can be deleted after they are not needed anymore.

3.1.6 Control and storage

For the control of the system a finite state machine (FSM) has been implemented. FSM carries out the blocks control to, recursively, find the inverse matrix as follows:

- ① Matrix A is transposed and Max values $\|A\|_1, \|A\|_\infty$ are computed.
- ② Equation (3.2) is performed.
- ③ Initial guess $N_{m=0}$ is calculated, equation (2.21) .
- ④ $3I - AN_m$ is computed and stored. Besides, verifier block checks if $A * N_m \approx I$.
- ⑤ If $A * N_m \approx I$, inversion process is finished.
- ⑥ If $A * N_m \neq I$, inversion process continues with ⑦.
- ⑦ Core block performs and stores the operation $3I - AN_m(3I - AN_m)$.

- ⑧ Core block computes $N_m(3I - AN_m(3I - AN_m))$.
- ⑨ $N_m = N_{m+1}$.
- ⑩ Process goes to step ④.

An embedded RAM memory is used to store the data. Each memory is a dual access RAM, so two elements can be withdrawn and one element can be stored, all simultaneously. The block has three access port address: two for reading and one for writing. Data to be stored in memory is sent to the memory input data port. Memory elements are withdrawn to access them at output data 1 and output data 2 ports. WE (Write Enable) port enables writing into memory; writing address and data to be written could be ready but data is not written until WE signal value is one.

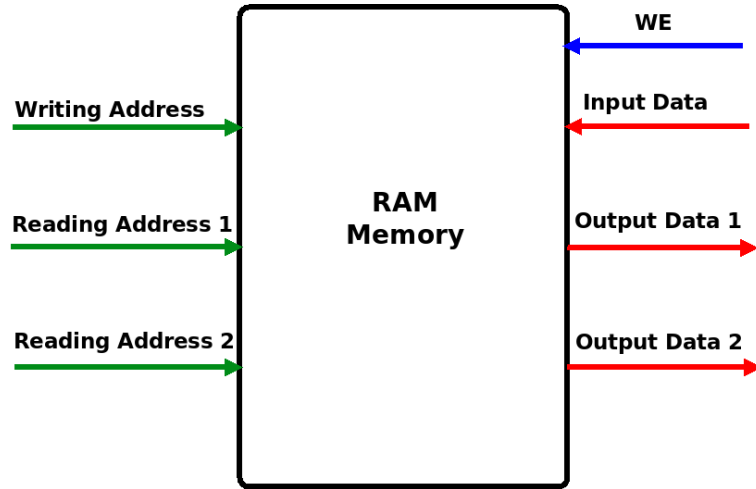


FIGURE 3.7: RAM memory block.

3.2 CoSaMP Architecture

In 2010, Jicheng Lu et al. [11] presented an optimization of the CoSaMP algorithm introduced in [37], see table 2.1. In this work, the optimized CoSaMP algorithm shown in table 3.2 was implemented. Several FPGA blocks were developed to, altogether, carry out the compressed sensed signal reconstruction. Figure 3.8 depicts FPGA blocks that computes CoSaMP algorithm of table 3.2.

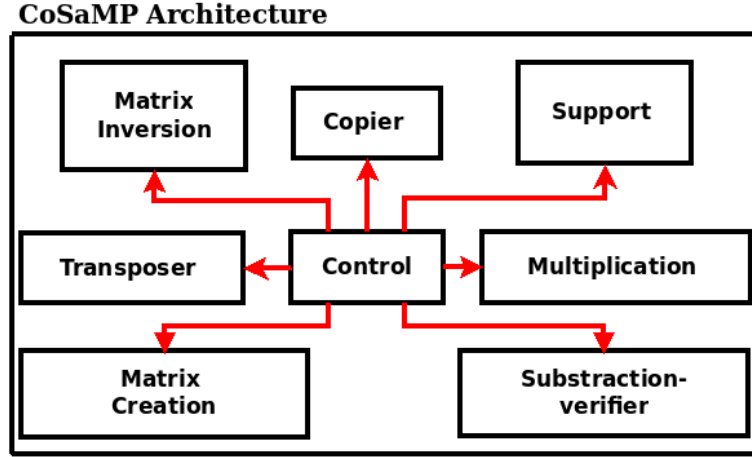


FIGURE 3.8: FPGA CoSaMP Architecture blocks.

3.2.1 Transposer

The transposer block aims to give a transpose version of any matrix and is needed when the Moore-Penrose pseudoinversion, given by (2.13) and re-written in (3.3), has to be computed.

$$A^\dagger = (A^T A)^{-1} A^T \quad (3.3)$$

Operation is performed on a Φ sub matrix called Φ_T and equation (3.3) becomes (3.4). Φ_T matrix is created by the matrix creation block described in the matrix creation subsection .

$$(\Phi_T)^\dagger = ((\Phi_T)^T (\Phi_T))^{-1} (\Phi_T)^T \quad (3.4)$$

Transposer block has two address generators whose architecture is as shown in figure 3.9. Counter 0 to dim counts from zero to a specified matrix dimension; for both, row

TABLE 3.2: Optimized CoSaMP Algorithm

Input: Sampling matrix Φ ; Sample data y ; Sparsity level k
During the 1st iteration As $a^{(0)} = 0$ and $r = y$ $u = \Phi^T r$ $T = \text{supp}(u_{3k})$ $b _T = (\Phi _T)^\dagger y$ $b _{T^c} = 0$ $a^{(1)} = b_k$ $r = y - \Phi a^{(1)}$
During the ith iteration $u = \Phi^T r$ $P = \text{supp}(a^{(i-1)})$ $\hat{u} = u _{P^c}$ $\Omega = \text{supp}(\hat{u}_{2k})$ $T = \Omega \cup P$ $b _T = (\Phi _T)^\dagger y$ $b _{T^c} = 0$ $a^{(i)} = b_k$ $r = y - \Phi a^{(i)}$
Output An k -sparse approximation a of the target signal.

and column address generators, this parameter will be matrix j-dimension. Counter dim counts on dim increments. Here, dim for row address generator is matrix j-dimension and for column address generator is matrix i-dimension.

The first address purpose is to read the matrix to be transposed as row elements. Second address is to store the read row element as a column element. Figures 3.10 and 3.11 show the path that transposer address generators follow while they are reading matrix A and writing matrix A^{-1} .

3.2.2 Support

The support (supp) of a vector is the index set of the non-zero vector elements. Figure 3.12 shows a vector decomposed in two sets: elements and index. The support of this

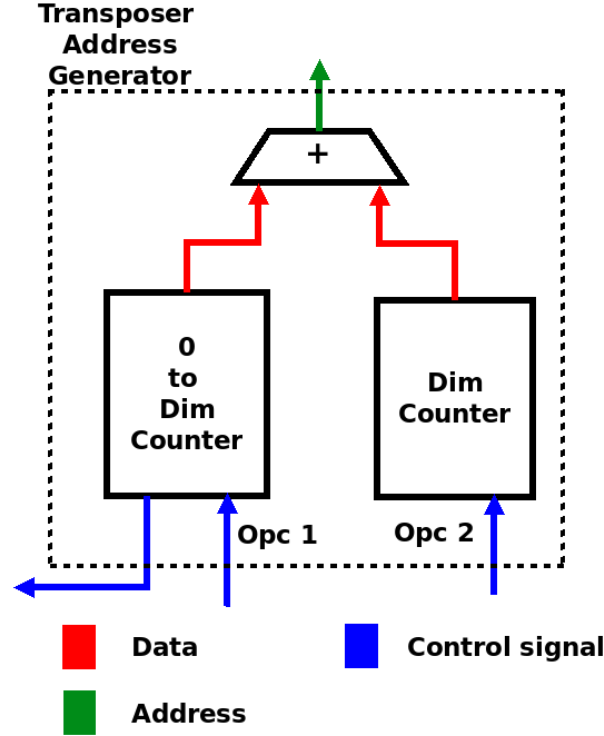


FIGURE 3.9: Transposer address generator.

vector will be the index subset $T = [1 \ 3 \ 4]$.

Greedy algorithm usually require fewer elements than the whole support set; this is dependent on the sparsity level k of the compressed sensed signal. When k -support is needed index of k greater elements of the vector are meant to be taken. For example figure 3.12, vector k -support with $k = 2$ will be $T = [1 \ 3]$ as the two greater elements are 5 and 9 and their index are the needed k -support.

In order to find k -support of a given vector, for any k , a great-to-small arranging block is implemented; in that manner index of k larger elements are at the first k memory locations.

The Great-to-small block architecture is limned on figure 3.13. This blocks access two RAM memories, one is where vector data is stored and the other has the index of the data. Vector data is loaded in register 1 and 2 and at the same time index of that data are loaded in register 5 and 6. The two loaded elements are compared; register 3 loads the larger one and register 4 loads the smaller one. Multiplexers 1 and 2 send to the block outputs the data and its index to be written into memory, addressing first the larger element.

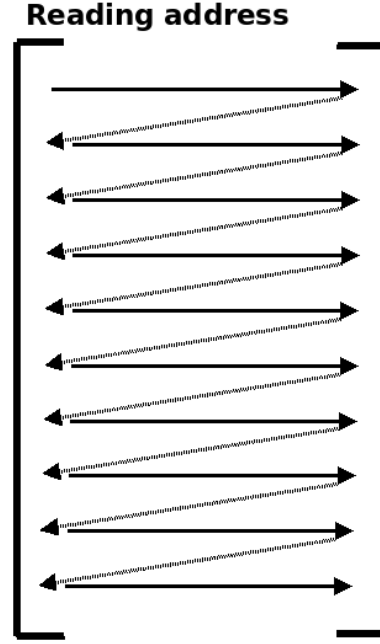


FIGURE 3.10: Row address generator path.

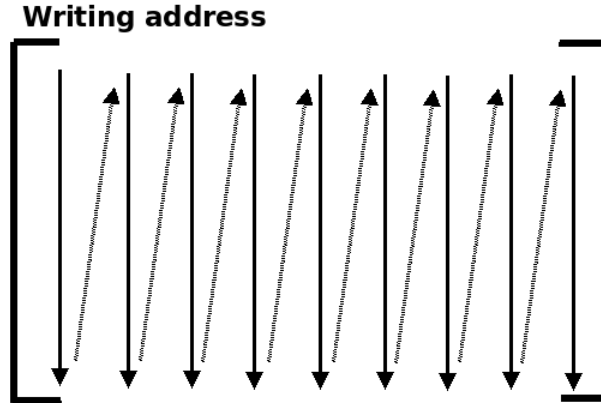


FIGURE 3.11: Column address generator path.

In this block, the vector elements are arranged in descendent order comparing pairs of elements and re-arranging them if needed. At the end the index of k , $2k$ or $3k$ larger elements can be taken to construct the required support.

3.2.3 Matrix Creation

Matrix creation block creates a tall matrix, Φ_T , of size m by $3k$, where m is the number of samples and k the sparsity level. This matrix is a sub-matrix of the sensing matrix Φ and is created taking $3k$ columns of Φ .

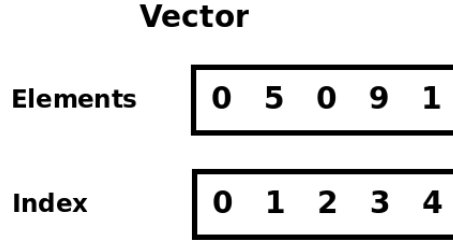


FIGURE 3.12: Decomposed vector in elements set and index set.

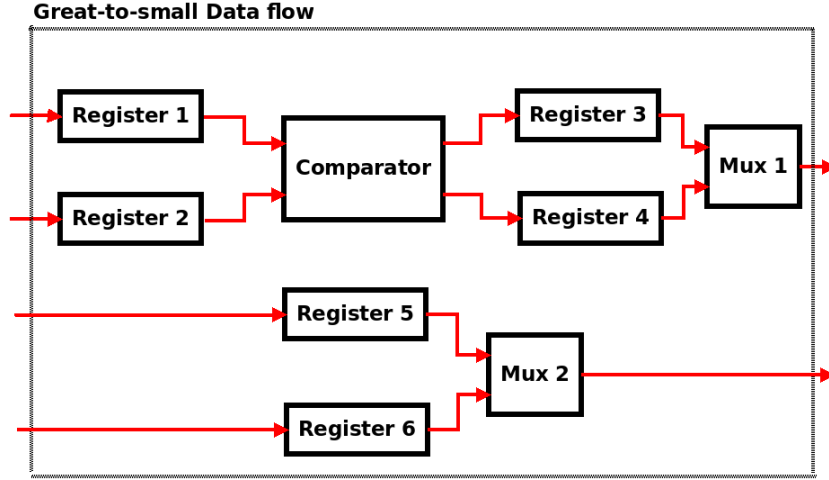


FIGURE 3.13: Great-to-small architecture.

The chosen Φ columns that span the matrix Φ_T are those whose index are in the support vector; i.e. if support vector is $T = [5 \ 2 \ 4]$ then columns 5, 2 and 4 form Φ_T . To achieve the matrix creation the architecture has two counters: one that goes from 0 to $3k$ and the second one that counts in increments of $3k$. The output of the 0 to $3k$ counter is used as the support memory reading address. The output of the support is added to the $3k$ counter output to form the Φ memory reading address, see figure 3.14. Writing address is generated in a similar way as the transposer reading address, see figure 3.9.

3.2.4 Vector copier

Copier block is used to perform a vector replica in operations $a^{(1)} = b_k$ and $\hat{y} = y|_{P_C}$. Copier is composed by one counter and a finite state machine. The counter generates the reading and writing address, both being the same.

The finite state machine controls the counter and generates the memory write enable (WE) signal. Figure 3.15 depicts the vector copier structure.

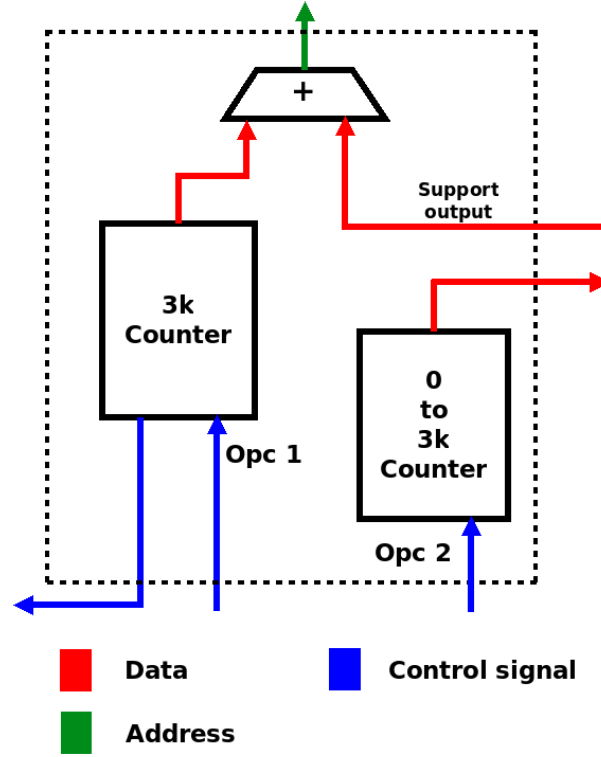


FIGURE 3.14: Reading address of matrix creation block.

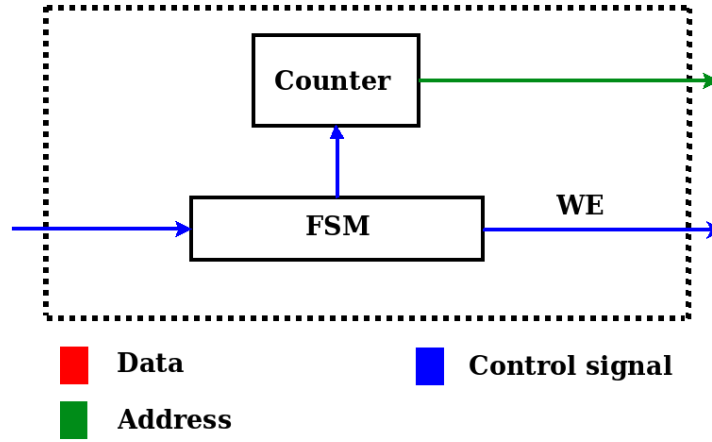


FIGURE 3.15: Vector copier structure.

The output of the memory where the vector to be copied is stored is directly connected to the data input of the memory that will contain the vector replica.

3.2.5 Vector reset

In order to implement CoSaMP algorithm vectors b , \hat{y} , Ω , T and P need to be reset on each iterations. b and \hat{y} must start with all their elements values set to zero. Support

vectors Ω , T and P are reset by setting their values in ascendant order from 0 to $n - 1$, where n is the sensed signal length.

Two architectures are required to carry out this task: zero reset and ascending order reset. These architectures are identically composed by a counter to generate the memory address where the data will be written and a finite state machine to generate the memory write enable (WE) signal.

The difference between these two architecture is the output data to be written; for the zero reset the output data always will be zero and for the ascending order reset the output data will be the same generated address.

3.2.6 Matrix inversion

Matrix inversion is the architecture depicted in figure 3.1 and described in the previous section. For the CoSaMP algorithm the matrix size to be inverted depends on the sparsity of the compressed sensed signal and always is $3k$ by $3k$, where k is the sparsity level.

3.2.7 Multiplication

Multiplication is the generic matrix/matrix multiplication sub-block whose nucleus is limned in figure 3.5. As a generic architectures, it can multiply matrix and vectors of any size and it is also flexible to word length variations.

3.2.8 Subtraction and Verifier

CoSaMP algorithm at its final step computes a residual to reflect the part of the signal that has not been approximated. This operation is defined in equation 3.5. When residual, r , is smaller than an established threshold, CoSaMP iterations stop.

$$r = y - \Phi a^i \quad (3.5)$$

Where:

r Residual,

y samples vector,

Φ sensing matrix,

a^i current signal approximation.

In this stage multiplication Φa^i , subtraction $y - \Phi a^i$ and verification of r value are performed in parallel. Multiplication is computed by the multiplication block and a Subtraction-verifier block is implemented to execute the remaining two operations. Subtraction-verifier block is depicted in figure 3.16.

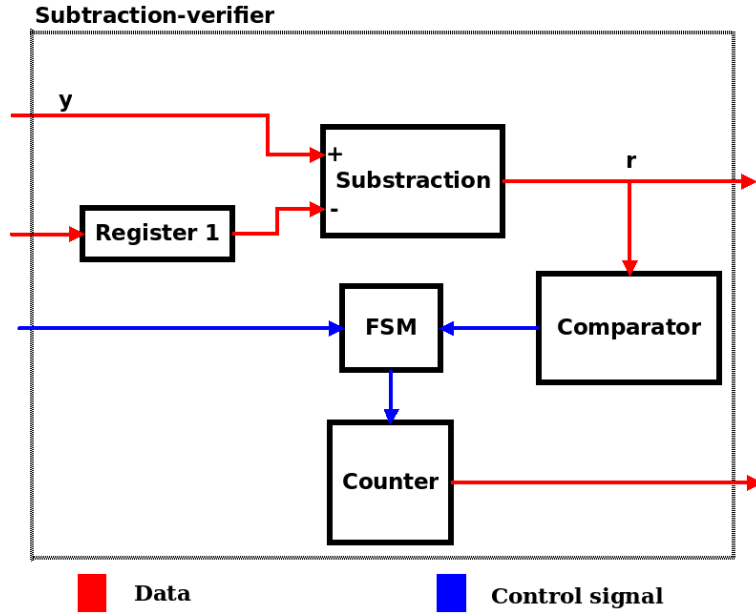


FIGURE 3.16: Subtraction-verifier block structure.

While the residual vector, r , is computed, comparator checks if every element of r meets the specified threshold condition. If this condition is met, comparator sends a signal to the finite state machine in order to increment the counter. Counter keeps track of the non desirable values of r . At the end of the r calculation, if counter output is zero CoSaMP algorithm is finished, otherwise another iteration is started.

3.2.9 Control

Control block has a ROM Memory and a finite state machine. The ROM memory has all control signals stored and the finite state machine controls ROM access address. The computing flow of the implemented CoSaMP architecture is as follow.

- ① Operation $u = \Phi^T r$ is computed.
- ② Support $T = \text{supp}(u_{3k})$ is obtained.
- ③ Sub-matrix Φ_T is created.

- ④ Pseudoinverse $(\Phi_T)^\dagger$ is performed.
- ⑤ Multiplication $b = (\Phi_T)^\dagger y$ is computed.
- ⑥ Signal approximation is created by copying the k larger elements of b , $a = b_k$.
- ⑦ Residual is calculated, $r = y - \Phi a$.
- ⑧ If residual r meets threshold condition, CoSaMP algorithm is finished.
- ⑨ If residual value is above specified threshold, CoSaMP continues with another iteration.
- ⑩ Operation $u = \Phi^T r$ is computed.
- ⑪ Counter of subtraction-verifier block is reset.
- ⑫ Support of previous signal approximation, $P = \text{supp}(a)$, is obtained.
- ⑬ A copy of vector $\hat{u} = u$ is made taking out all elements whose index are in the support vector P .
- ⑭ Support $\Omega = \text{supp}(\hat{u})$ is obtained.
- ⑮ Support T is obtained by merging P and Ω , $T = P \cup \Omega$.
- ⑯ Algorithm goes to step ③

Storage of all vector and matrices is achieved by implementing RAM memories as the one shown in figure 3.7.

Chapter 4

MATLAB Graphical User Interface

MATLAB graphical user interfaces (GUI) are friendly and mostly point and click software applications, useful to organize data and display results in a fast and easy way. For this work a MATLAB GUI has been developed as a tool for compressed sensing theory exploration. Its main purpose is to rapidly test sensing matrix and transforms to 1-D and 2-D signals. CoSaMP algorithm is used to reconstruct the sensed signal, but as a flexible interface it can be easily modified to use other compressed sensing algorithms. Figure 4.1 shows the user interface.

GUI features, leaving MATLAB code as implicit, can be divided into two categories: inputs and outputs.

4.1 GUI Inputs

The user can define a set of conditions to perform an experiment. These are defined by the data type, sensing matrix, transform, maximum compressed sensing algorithm iterations, signal sparsity, sparsity threshold and name of the file where results will be stored; these inputs are chosen to fulfill the test needs.

4.1.1 Data type

The interface can work with two different data types: images and 1-D signals; it has two mutually exclusive radiobuttons in order to select the data type, see figure 4.2.

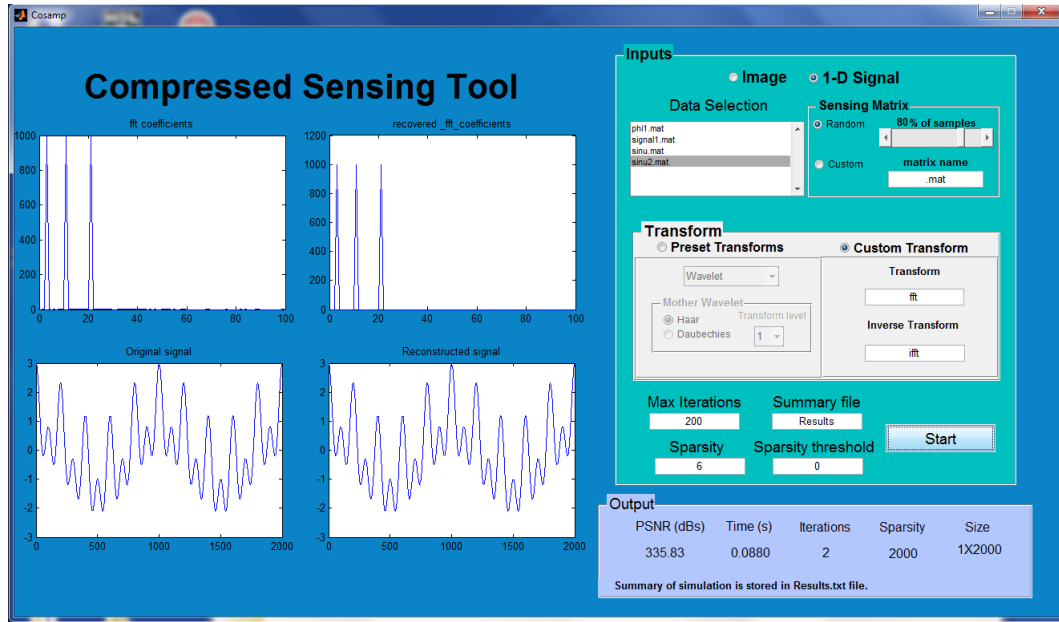


FIGURE 4.1: Graphical User Interface as a compressed sensing tool.

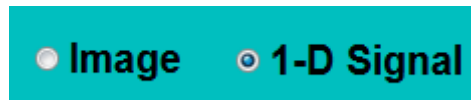


FIGURE 4.2: Data type selection.

Two folders must be created in the code root directory to store the signals that will be processed, these folders are *images*, for images and *signals* for 1-D signals. When a data type is selected, for example 1-D signals, GUI accesses the signals folder and populates a listbox with the name of all files in that folder whose extension is *.mat*, this is depicted in figure 4.3.

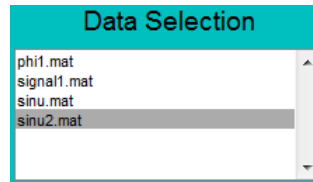


FIGURE 4.3: Data file selection.

Images must be in *.jpeg* and *rgb* formats. Images are read using MATLAB function `imread('images/images_name.jpeg')`. Once an image is loaded, it is converted to gray scale using `rgb2gray(images)` function.

1-D signals are stored in a *.mat* file. The *.mat* file needs to contain the signal in a variable named *x*. Signal is loaded using MATLAB `load('signals/signal_name.mat')`

function.

Loaded data is simulated to be compressed sensed computing operation Φx . Subsequently, it is reconstructed with the CoSaMP algorithm.

4.1.2 Sensing matrix

Sensing matrix, as described in previous section, is an important element for compressed sensing that can lead to success or failure on the signal reconstruction. MATLAB GUI let the user to choose the proper sensing matrix for an specific problem, see figure 4.4.

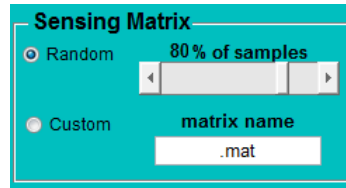


FIGURE 4.4: Sensing matrix selection panel.

Random sensing matrix, as it has been said, fulfill the rip condition so they are, with high probability, good to be a sensing matrix; therefore, random matrices are the default sensing matrix option. The m by n sensing matrices are generated using `phi = rand(n,m)` MATLAB instruction, where n is the sensed signal length and m is the number of samples, a percent of n .

The percentage of measurements, m , is controlled by a slider GUI object. Slider values vary from 0 to 1 representing a 100% scale. The number of measurements must be greater than three times signal sparsity, this is:

$$m \geq 3k \quad (4.1)$$

Custom matrix radiobutton gives the option to use any other matrix that were previously created and saved as a .mat file. The slider is disabled and the textbox is enabled when the custom matrix option is selected. Sensing matrix is imported using MATLAB function `load(matrix_name.mat)`. By default textbox text is `.mat`; if this text is leaved as it is, the code interprets that no sensing matrix has been selected and shows a message error, see figure 4.5.

The j -dimension of loaded custom matrix must be the same as the sensed signal length (n). Otherwise, a message error window, shown in figure 4.6, is displayed.

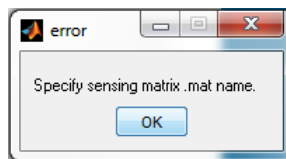


FIGURE 4.5: Custom sensing matrix name error.

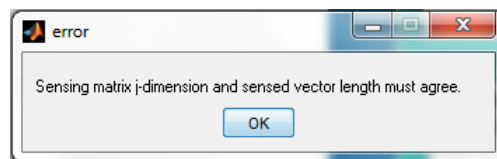


FIGURE 4.6: Matrix dimension and sensed vector length mismatch.

4.1.3 Transform

Transform selection is crucial for compressed sensing, for different transforms signal sparsity might vary. This GUI let the user to select which transform will be used to simulate the compressed sensing acquisition. Preset or a user defined transforms can be selected.

There are two preset transforms defined for for 1-D and 2-D: wavelet and discrete cosine transform. From a pop-up menu one of the preset transforms can be chosen; as default wavelet transform is selected. Wavelet and DCT require different parameter to be set. A transform parameters setting menu is displayed for the selected preset transform.

Wavelet transform can be computed using *haar* or *daubechies* mother wavelet, with six decomposition levels. Figure 4.7 shows wavelet setting menu.

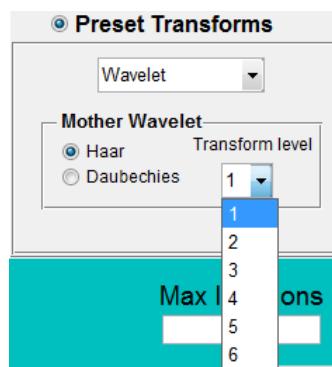


FIGURE 4.7: Wavelet setting menu.

Wavelet setting menu becomes invisible when DCT transform is selected; in its place slider and label GUI objects show up, see figure 4.8. These objects are used to set a threshold for the DCT, all signal elements under that threshold are set to zero. Slider

values variate from 0 to 1 with increments of 0.001, giving threshold values ranging between 1 to 1000.

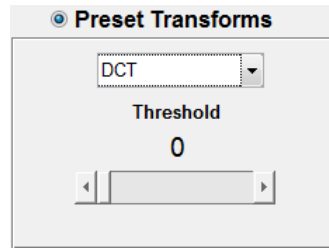


FIGURE 4.8: Discrete cosine transform setting menu.

Besides the preset transforms, a custom user defined transform can be used in this GUI. Preset transform panel is disabled when custom transform option is selected. The custom transform must be coded as a function in MATLAB language and stored in a file with .m extension; the file have to be placed in the root GUI folder. The input of the transform function is an image or an 1-D signal vector, the output should be a coefficient matrix of the same size as the image or a vector of the same length of the 1-D signal. Custom transform panel is shown in figure 4.9.

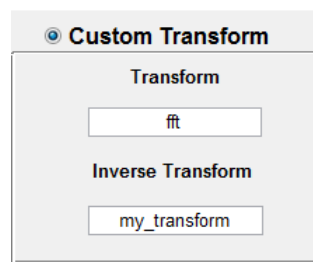


FIGURE 4.9: Custom transform panel.

In addition to the transform .m file an inverse transform file is needed to reconstruct the 1-D signal or image from its recovered transform coefficients. Structure of the inverse transform file is similar to the transform structure.

The two text box in the custom transform panel are use to write down the transform and inverse transform names. Their initial text is *my_transform*. If transform textbox text is the same as its default, it is infer that no transform has been selected and the message box shown in figure 4.10 is displayed. The same for inverse transform textbox, inverse transform error message is shown in figure 4.11.

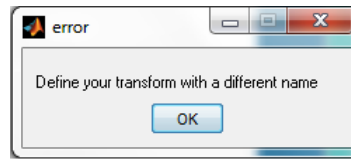


FIGURE 4.10: Wrong transform selection error.

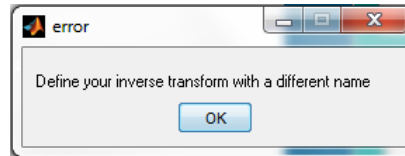


FIGURE 4.11: Wrong inverse transform selection error.

4.1.4 Other inputs

In other inputs are enclosed the parameters shown in figure 4.12: max iteration, sparsity, sparsity threshold and summary file name.

 A GUI panel with a teal background. It contains four input fields arranged in a 2x2 grid. The top-left field is labeled 'Max Iterations' and contains the value '200'. The top-right field is labeled 'Summary file' and contains the text 'Results'. The bottom-left field is labeled 'Sparsity' and contains the value '1'. The bottom-right field is labeled 'Sparsity threshold' and contains the value '0'. To the right of these fields is a large grey button labeled 'Start'.

FIGURE 4.12: Input parameters.

Max iteration refers the maximum number of iteration for the CoSaMP algorithm.

Sparsity is a parameter that defines how many elements of a signal or image are the most representative. Sparsity parameter can be larger or smaller than the real signal sparsity. For example almost all FFT coefficients of a sinusoidal signal are close to zero, but are not zero and just one of them is the most representative so the user can define sparsity parameter as one.

Sparsity threshold is used to set to zero all transform coefficients under that threshold. GUI computes the real sparsity of the signal so the user can find out the threshold value above which the most significant transform coefficients are located.

GUI generates a .txt summary file which is described in next subsection. In the textbox text the summary file name is written.

4.2 GUI outputs

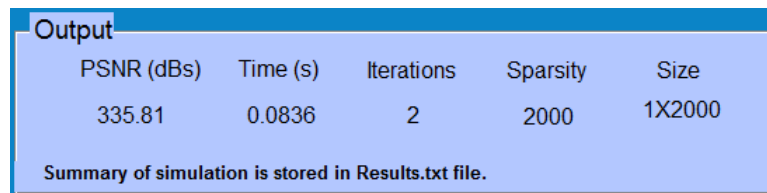
After the algorithm is finished the result information is given mainly by three different ways: label text objects, graphs and plots and a summary file.

In the GUI design are five labels that give a first glimpse to the summary of the experiment. Those labels can be seen in figure 4.13.

PSNR value is the peak signal to noise ratio between the original and recovered image; mean squared error (MSE) is computed between original and recovered 1-D signal. PSNR and MSE are used to measure the reconstruction performance.

Time expresses in seconds how long it took for the CoSaMP algorithm computation. Iteration label shows the number of iteration CoSaMP algorithm took to recover the transform coefficients.

Sparsity label give the real sparsity of the signal; for figure 4.13 was the sparsity of the FFT coefficients for a three added sinusoidal signal with two thousand samples length; it can be seen that no one of its coefficients was zero, although user defined sparsity for this experiment was three. Last label shows the under examination signal or image size.



PSNR (dBs)	Time (s)	Iterations	Sparsity	Size
335.81	0.0836	2	2000	1X2000

Summary of simulation is stored in Results.txt file.

FIGURE 4.13: Output GUI labels.

Four axes are other kind of GUI output objects. Upper left axes display the transform coefficients and upper right axes display the recovered transform coefficients. Lower left axes display the original signal or images and lower right axes the recovered signal or image.

Figure 4.14 depicts an experiment for an image of 16 by 16 pixel size using DCT tranform with threshold = 50.

The experiment of a three sinusoidal added signal using Fast Fourier Transform, whose output label values are in figure 4.13, is depicted in figure 4.15.

The GUI creates a folder named results in the root directory. Every time the code is executed, a folder with the date and time of the simulation is created in it. A .mat file, containing simulation variables, is saved into the simulation folder. This is in order to reproduce the simulation under the same conditions.

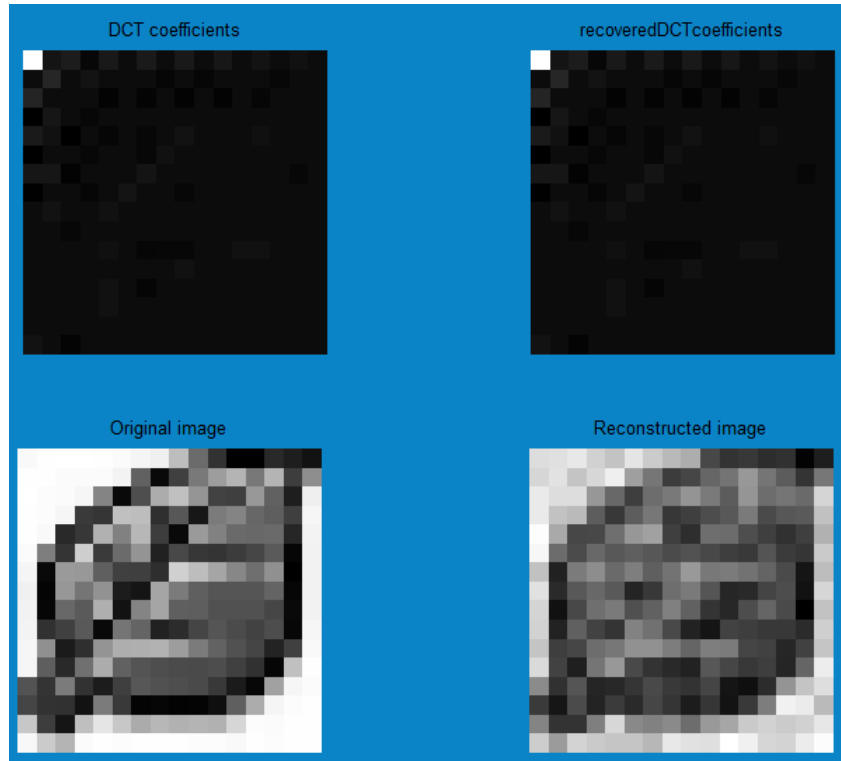


FIGURE 4.14: Image simulation axes display.

Additionally a .txt file is created to store simulations summary. Summary information is organized as follows.

- *folder*, is the folder, name with the simulation date, where data is stored.
- *file*, is the time and file name of the simulation.
- *testing signal*, is the name of the image or 1-D .mat used for that simulation.
- *Size*, is the size of the signal.
- *Transform*, is the used transform.
- *Samples percent*, is the value of m , that is the percent of n . It is expressed in values ranging between 0 and 1.
- *dec_lev*, is the Wavelet decomposition level, if used transform is not wavelet a value of NA will appear.
- *DCT_Thresh*, is the discrete cosine transform threshold. If a different transform is used NA will appear.
- *k*, is the real signal sparsity.

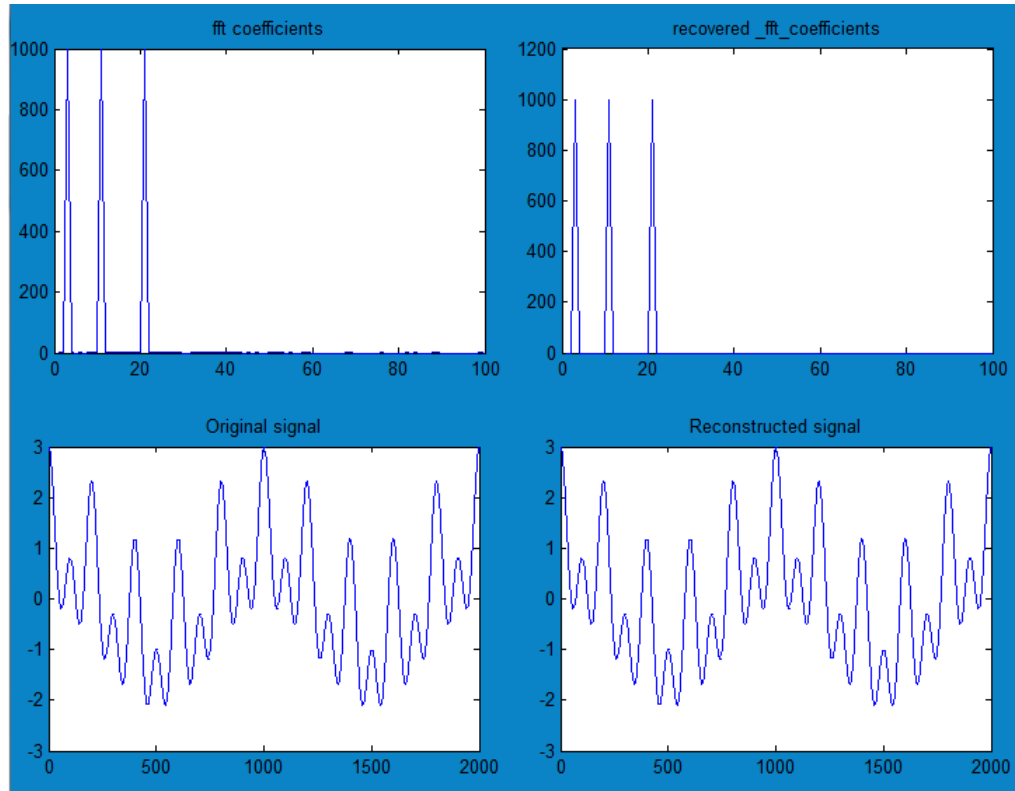


FIGURE 4.15: 1-D simulation axes display.

- $k \text{ thresh}$, is the sparsity threshold, used to decrease the sparsity of a signal.
- $\text{fix } k$, is the user defined signal sparsity.
- fom , is the figure of merit value that measures the signal reconstruction. Peak Signal to Noise Ratio for images and mean square error for 1-D signals.
- $Time$, is the CoSaMP computation time (seconds).
- $Iterations$, are the CoSaMP iterations taken to recover a signal.
- max iterations , are the maximum number of iterations defined by user to automatically break CoSaMP algorithm.

An important conclusion can be made through summary file: if iterations are equal to max iterations, signal was not correctly recovered.

Chapter 5

Results

In this work an FPGA architecture for compressed sensing signal recovery has been developed. This architecture includes matrix multiplication, matrix transposition, matrix inversion and a Compressed Sampling Matching Pursuit. Besides, a matlab graphical user interface has been developed as a compressed sensing theory exploration tool.

5.1 FPGA architectures

FPGA architectures are validated by their device utilization, latency and maximum working frequency. This implementation is focused on resource optimization. Usage of DSP blocks, the most limited resources, is mainly affected by the word length.

5.1.1 Matrix Multiplication Architecture

The matrix multiplication architecture is a generic FPGA structure capable to multiply matrices and vectors of any size. Tables [5.1](#) for Virtex 4 and [5.2](#) for Spartan6 summarize the device resource utilization for the matrix multiplication structure for different word lengths: 20, 26 and 36 bits. It can be seen that this architectures leaves enough space to implement other architectures.

TABLE 5.1: Matrix Multiplication Xilinx Virtex 4

Device Utilization Summary				
Logic Utilization	36 Bits	26 Bits	20 Bits	Available
Slices	771	457	529	10240
Slice Flip Flops	797	519	440	20480
4 input LUTs	2168	676	900	20480
DSP48s	32	16	4	128

TABLE 5.2: Matrix Multiplication Xilinx Spartan 6

Device Utilization Summary				
Logic Utilization	36 Bits	26 Bits	20 Bits	Available
Slice Registers	785	528	497	54575
Slice LUTs	1138	644	1246	27288
LUT-FF pairs	389	198	279	4780
DSP48A1s	32	16	4	58

Latency for the matrix multiplication architecture is depicted by the graph in figure 5.1. Latency increases on an exponential way as a function of matrix size n .

Figure 5.2 shows the working frequency behavior due to word length variations and different FPGA platforms. The maximum achieved working frequency for this structure is 164 Mhz in a virtex 4 using a word length of 20 bits.

5.1.2 Matrix Transposer Architecture

Matrix transposition is another important operation when working with matrices. The transposer architecture performs this operation. Being, as the matrix multiplication structure, a generic architecture it can transpose any matrix of any size. Matrix transposer device utilization for a virtex 4 and spartan 6 is summarize in tables 5.3 and 5.4. Dsp blocks are not needed for this structure and as it can be seen in the summary tables, transposer architecture uses small amount of the board resources.

The architecture is sensible to address length variations. The address length is the number of used bits for the memory address. It was synthesized for two address lengths 20 and 8 bits.

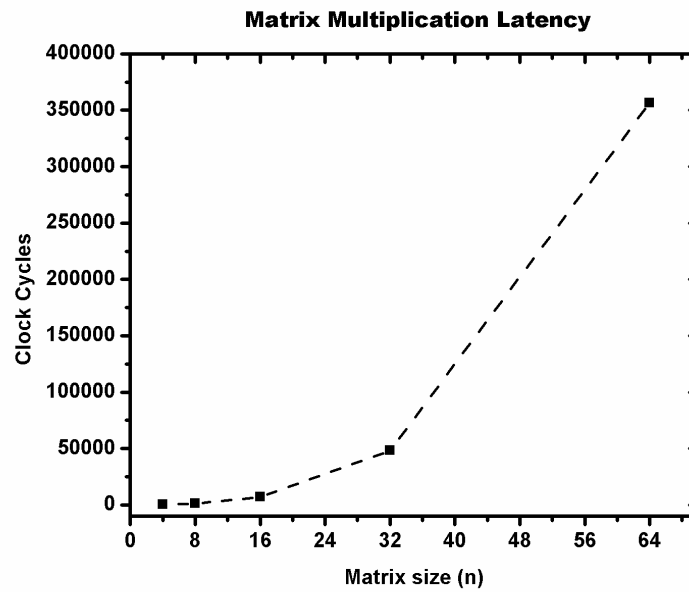


FIGURE 5.1: Matrix multiplication latency.

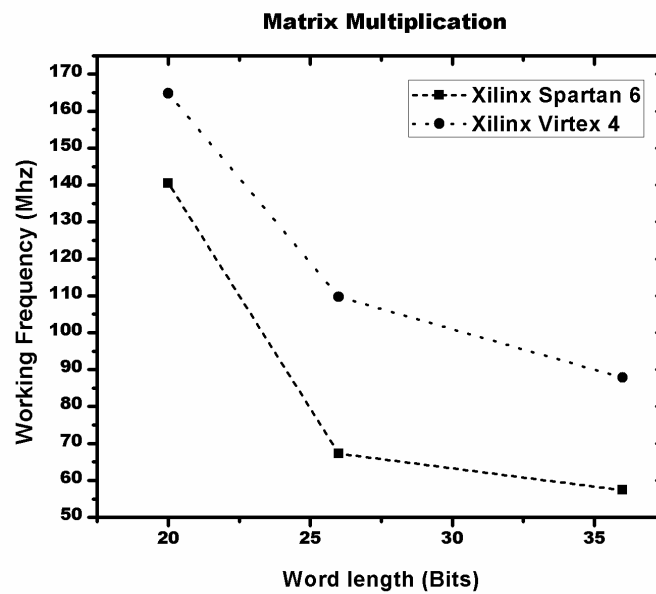


FIGURE 5.2: Matrix multiplication working frequency.

TABLE 5.3: Matrix Transposer Xilinx Virtex 4

Device Utilization Summary			
Logic Utilization	20 Bits	8 Bits	Available
Slices	102	57	10240
Slice Flip Flops	92	44	20480
4 input LUTs	195	108	20480
DSP48s	0	0	128

TABLE 5.4: Matrix Transposer Xilinx Spartan 6

Device Utilization Summary			
Logic Utilization	20 Bits	8 Bits	Available
Slice Registers	101	53	54575
Slice LUTs	155	81	27288
LUT-FF pairs	93	47	4780
DSP48A1s	0	0	58

Matrix transposition operation, based on the proposed FPGA architecture, has a latency shown in figure 5.3. This results are for square matrices of size n . However, the structure is proficient to transpose a matrix of any dimension.

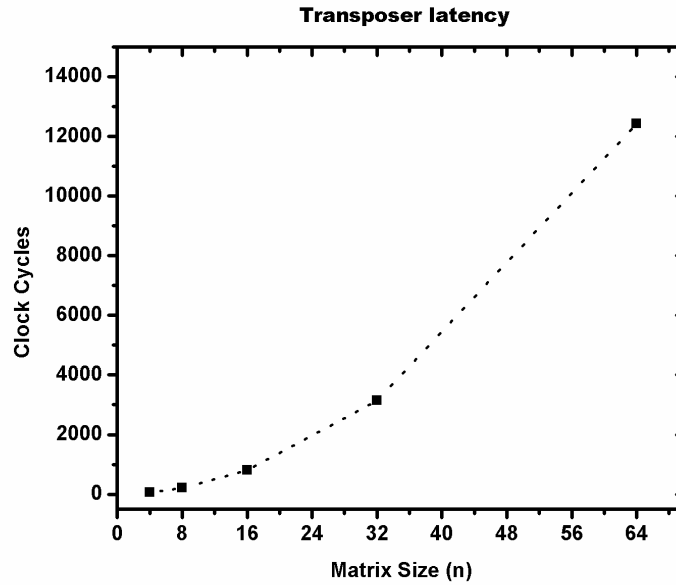


FIGURE 5.3: Matrix transposer latency.

Maximum working frequency is limned by the graph in figure 5.4. This architecture works with generating address so maximum working frequency is not affected by the word length; on the other hand address length directly affects the maximum working frequency. In spite of that the maximum working frequency difference between the architecture implemented on a virtex 4 and a spartan 6 is considerable, the maximum working frequency for the spartan 6 is still an excellent working frequency.

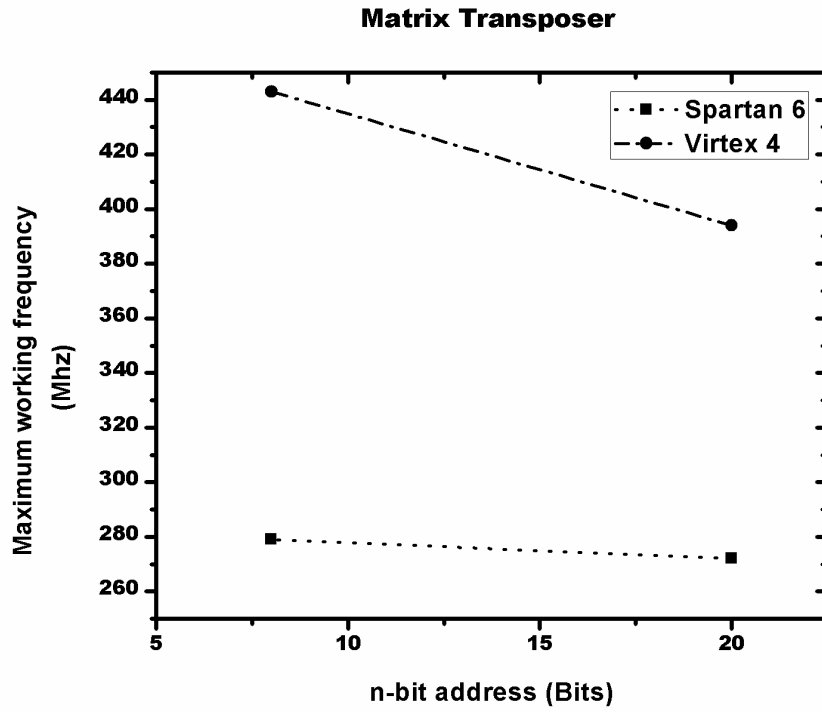


FIGURE 5.4: Matrix transposer maximum working frequency.

5.1.3 Matrix Inversion Architecture

Latency for the matrix inversion architecture, based on the iterative chebyshev-type method, it is mainly affected by the matrix size. Figure 5.5 shows a graph of the structure latency for different matrix sizes and word lengths, y-axis is in a logarithmic scale.

Table 5.5 shows that if word length does not change, DSP blocks usage remains unchanged. The architecture was implemented in various FPGA platforms: Virtex 4 XC4VSX25, Spartan 6 XC6SLX45, Cyclone IV EP4CGX150DF31C7 and Cyclone II EP2C35F672C6. Device utilization for the different FPGAs families for 36 bits word length and 8x8 matrix size is summarized in tables 5.6, 5.7, 5.8 and 5.9. Due to the resource utilization, cheap FPGA families such as the Spartan 6 can be used.

TABLE 5.5: Matrix inversion Dsp blocks usage

Matrix size (n)	This work			[41, 48]
	20 bits	26 bits	36 bits	19 bits
4 x 4	7	28	56	12
6 x 6	7	28	56	18
8 x 8	7	28	56	24

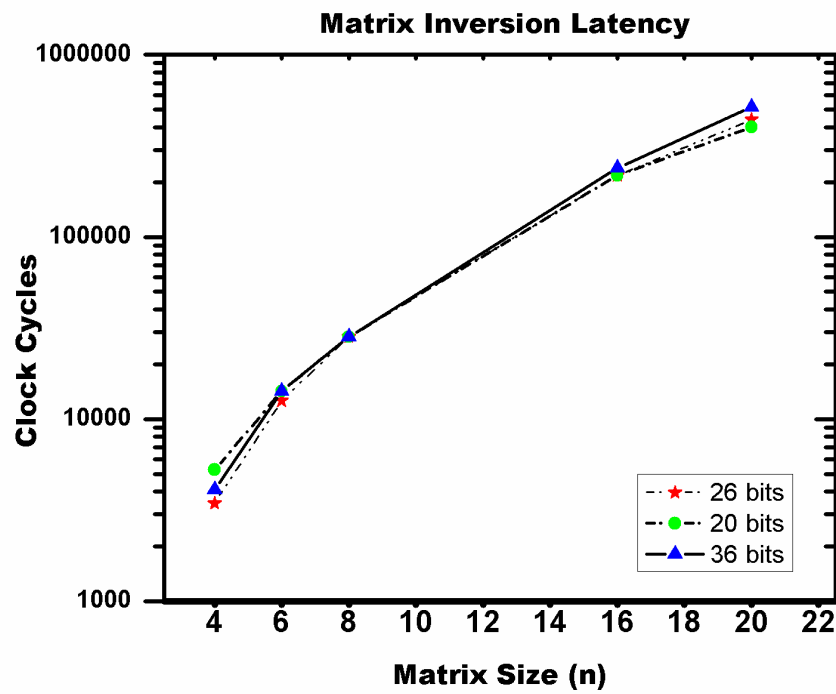


FIGURE 5.5: Matrix inversion latency at different word lengths.

TABLE 5.6: Matrix inversion and CoSaMP in Xilinx Virtex 4

Device Utilization Summary			
Logic Utilization	Inversion	CoSaMP	Available
Slices	1382	2835	10240
Slice Flip Flops	1190	1875	20480
4 input LUTs	2133	5084	20480
FIFO16/RAMB16s	12	53	128
DSP48s	56	56	128

TABLE 5.7: Matrix Inversion and CoSaMP in Xilinx Spartan 6

Device Utilization Summary			
Logic Utilization	Inversion	CoSaMP	Available
Slice Registers	1245	1917	54575
Slice LUTs	2013	3942	27288
LUT-FF pairs	560	1079	4780
Block RAM/FIFO	6	44	116
DSP48A1s	56	56	58

TABLE 5.8: Matrix Inversion and CoSaMP in Altera Cyclone II

Device Utilization Summary			
Logic Utilization	Inversion	CoSaMP	Available
Logic elements	2333	13830	33216
Registers	934	6030	33216
Memory bits	27648	383040	483840
Multipliers 9-bit	56	56	70

TABLE 5.9: Matrix Inversion and CoSaMP in Altera Cyclone IV

Device Utilization Summary			
Logic Utilization	Inversion	CoSaMP	Available
Logic elements	2352	4321	149760
Registers	970	1478	149760
Memory bits	27648	391488	6635520
Multipliers 9-bit	56	56	720

Figure 5.6 shows the maximum working frequency for the matrix inversion structure implemented in the different FPGA boards varying the word length.

5.1.4 CoSaMP Architecture

The Compressed Sampling Matching Pursuit (CoSaMP) algorithm was implemented including the iterative Chebyshev-type matrix inversion method. Latency for CoSaMP

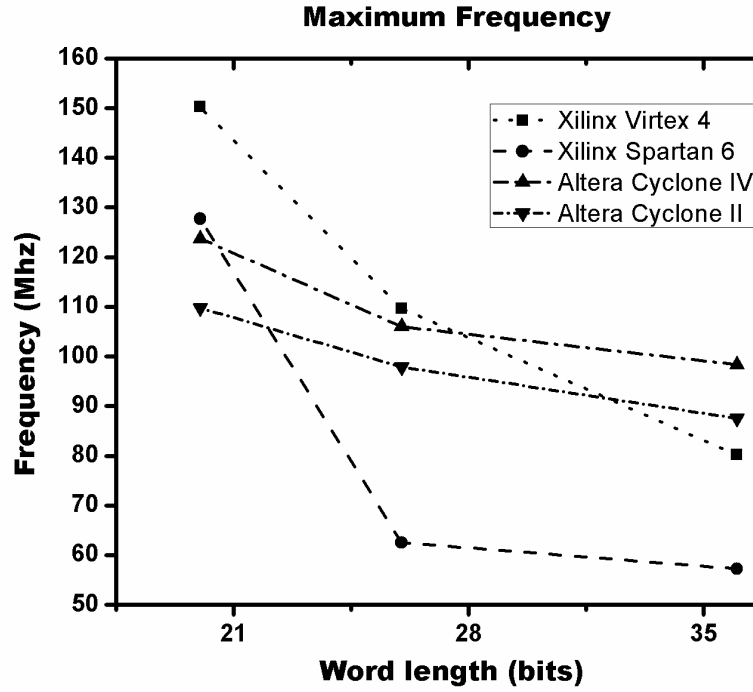


FIGURE 5.6: Maximum frequency comparison.

architecture is shown in figure 5.7. Latency was computed using test vectors of different sparseness, ranging from one to nine. Figure 5.8 depicts latency of an iteration of the CoSaMP algorithm partitioned into its main blocks. This picture is divided in three parts: pre-pseudoinversion, pseudoinversion and post-pseudoinversion.

Maximum working frequency for the CoSaMP algorithm varies as data word length change and is the same as the matrix inversion maximum working frequency. Obtained results are depicted in figure 5.6.

Aldec Active-HDL software was used to simulate the VHDL architecture. A simulation of the CoSaMP algorithm can be seen in figure 5.9; a signal with nine sparse level was recovered in five iterations. In descendent order, simulated signals are: reset, clock, CoSaMP finish signal, CoSaMP ready signal, pseudoinversion ready signal, inversion ready signal, matrix multiplication ready signal, support ready signal, matrix creation ready signal, vector copier ready signal, vector reset ready signal and transposer ready signal. For the ready signals a zero value is when the block is working and one when the block is ready to be used.

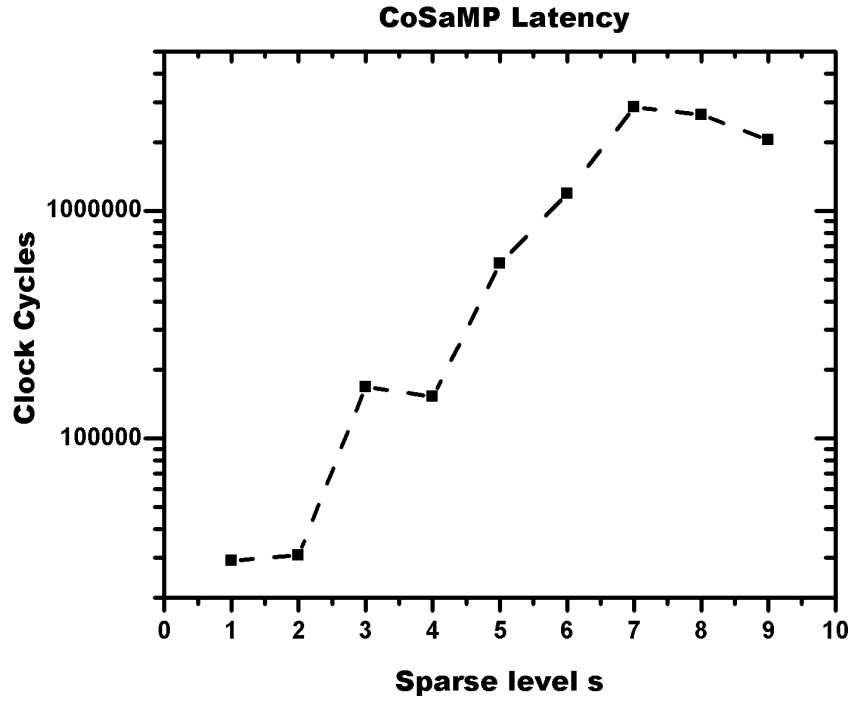


FIGURE 5.7: CoSaMP architecture Latency comparison at different sparse levels.

5.2 Matlab Graphical User Interface

Various test were performed using the developed MATLAB graphical user interface, in order to extract the signal features useful for compressed sensing. The 64x64 pixels images of figures 5.10 and 5.11 were used as testing images and as 1-D testing signals ten sinusoidal waveforms, described by the following equations and stored in a 2000 length vector, were used.

- ① $\cos(2\pi t)$
- ② $\cos(2\pi t) + 0.4\sin(2\pi 7t)$
- ③ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t)$
- ④ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t)$
- ⑤ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t) + 0.04\cos(2\pi 20t)$
- ⑥ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t) + 0.04\cos(2\pi 20t) + 0.06\sin(2\pi 25t)$
- ⑦ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t) + 0.04\cos(2\pi 20t) + 0.06\sin(2\pi 25t) + 0.058\cos(2\pi 29t)$

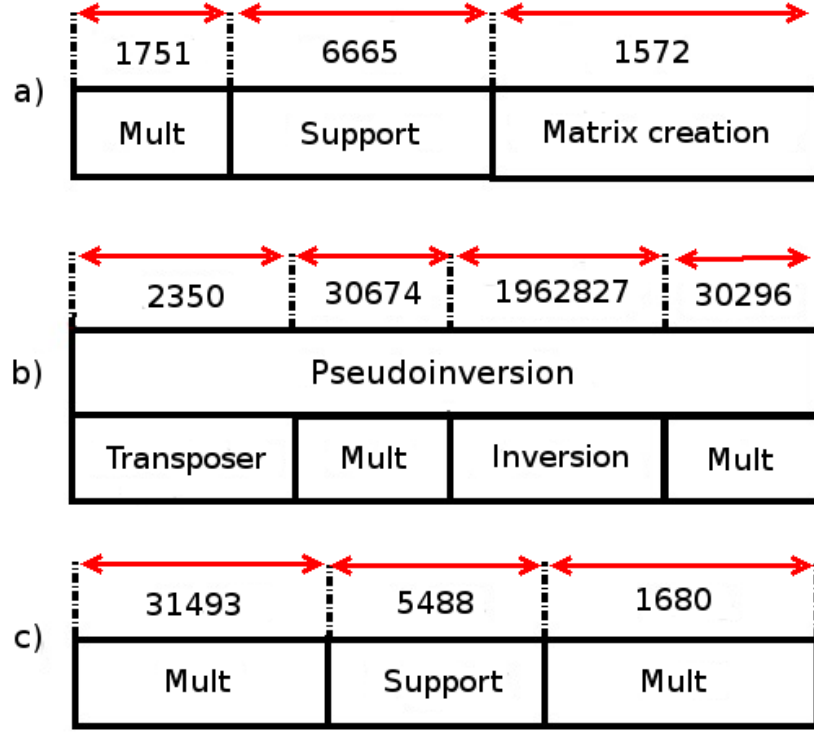


FIGURE 5.8: CoSaMP blocks latency for a signal with $s=9$ and 36 bits word length. Divided in a) pre-pseudoinversion, b) pseudoinversion and c) post-pseudoinversion.

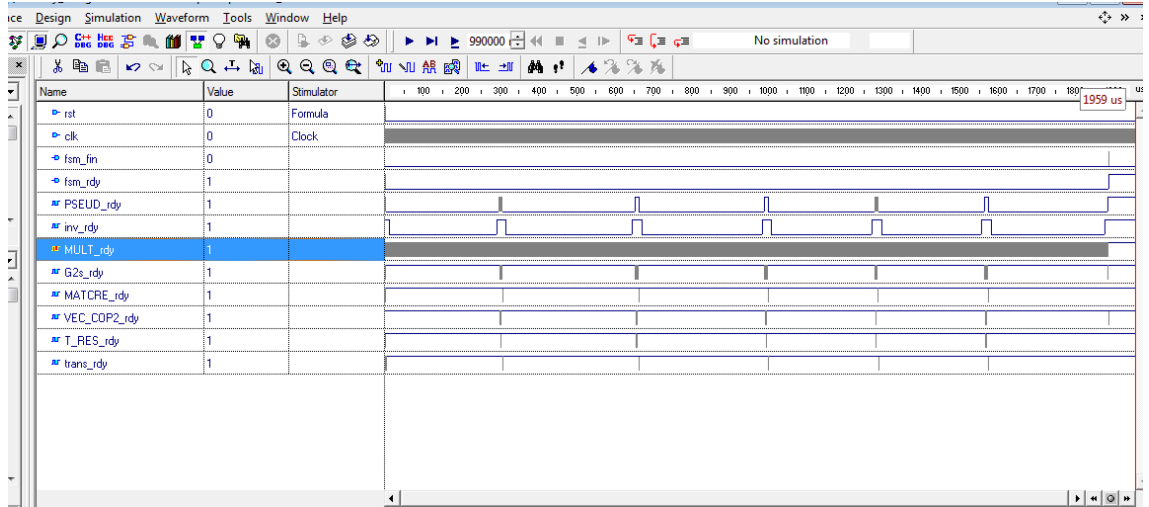


FIGURE 5.9: CoSaMP architecture simulation.

- ⑧ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t) + 0.04\cos(2\pi 20t) + 0.06\sin(2\pi 25t) + 0.058\cos(2\pi 29t) + 0.064\sin(2\pi 32t)$
- ⑨ $\cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t) + 0.04\cos(2\pi 20t) + 0.06\sin(2\pi 25t) + 0.058\cos(2\pi 29t) + 0.064\sin(2\pi 32t) + 0.104\cos(2\pi 52t)$

$$\textcircled{10} \cos(2\pi t) + 0.4\sin(2\pi 7t) + 0.2\cos(2\pi 10t) + 0.024\sin(2\pi 12t) + 0.04\cos(2\pi 20t) + 0.06\sin(2\pi 25t) + 0.058\cos(2\pi 29t) + 0.064\sin(2\pi 32t) + 0.104\cos(2\pi 52t) + 0.096\sin(2\pi 100t)$$



FIGURE 5.10: Lena testing image.



FIGURE 5.11: Pacman testing image.

First test was performed to obtain the sparseness of a 1-D signal due to different transform condition. In this test two transform, Discrete Cosine Transform and Fast Fourier Transform, were computed for the ten sinusoidal modeled by equations in the previous list. Figure 5.12 shows the sparsity of each sinusoidal signal due to the transform, setting to zero all transform coefficients whose absolute value is under one. It can be seen that for sinusoidal signals FFT gives a lower sparsity level than DCT.

Although FFT offers more sparsity on sinusoidal signals, CoSaMP showed a better reconstruction performance for sinusoidal signals sensed using DCT. The mean squared error is computed between original and reconstructed signal. Reconstruction was obtained taking 1000 measurements to recover the 2000 length signal. Results can be seen in Figure 5.13.

Figure 5.14 depicts results of an experiment performed to extract Lena image sparsity using wavelet transform. In this experiment a Haar mother wavelet is used for different wavelet decomposition levels and threshold values. It can be seen that with a zero threshold image has a poor sparseness; a considerable difference is seen as sparsity threshold increases. The reconstructed lena image can be seen in figure 5.15(b), it was recovered in three CoSaMP iterations using 70% of the samples.

Image sparsity due to specific transform conditions is also affected by the image nature. This is shown in figure 5.16 where sparsity of pacman (figure 5.11) and lena (figure 5.10) images is compared. In this test a haar wavelet transform is computed with a sparsity threshold value of 60 and four different decomposition levels.

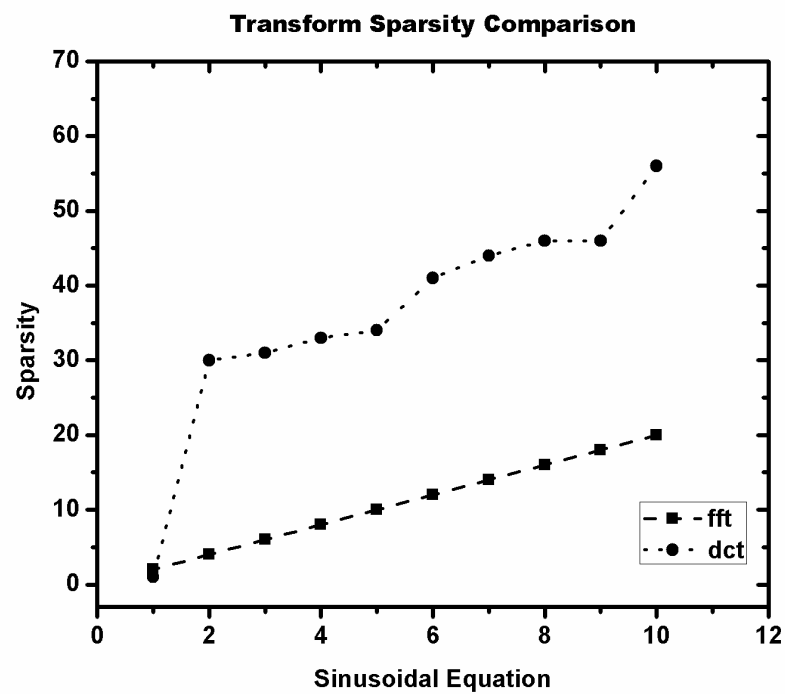


FIGURE 5.12: Transform comparison.

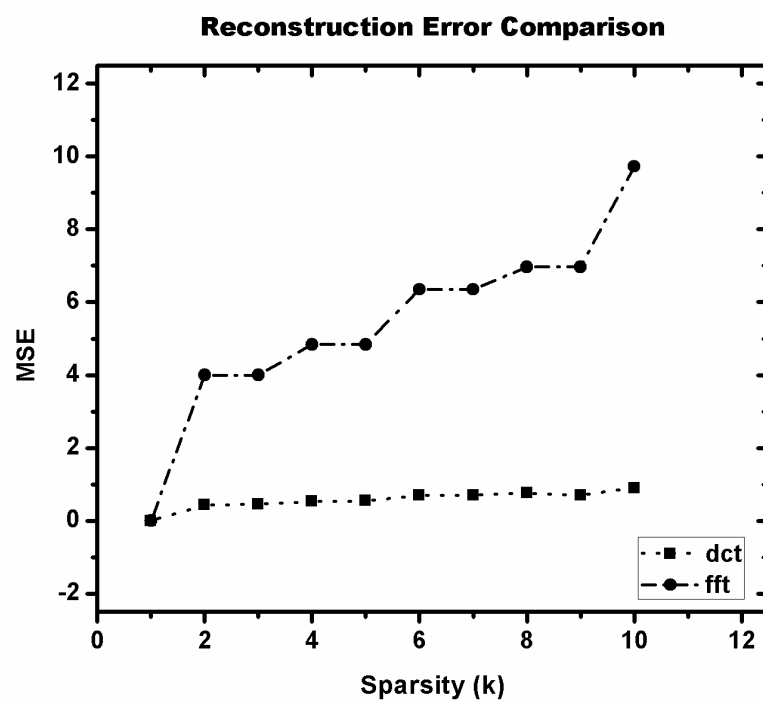


FIGURE 5.13: Reconstruction error.

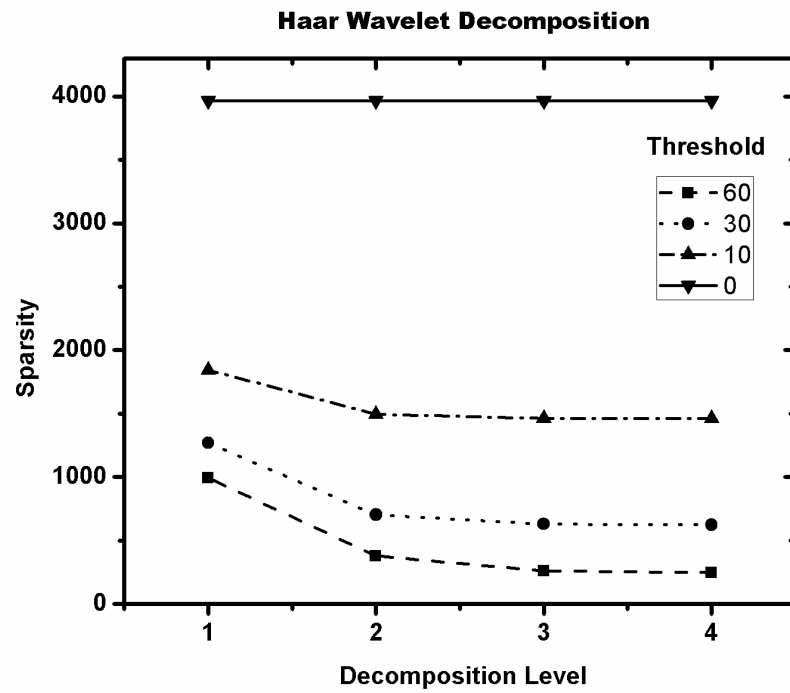


FIGURE 5.14: Haar transform for different threshold level.

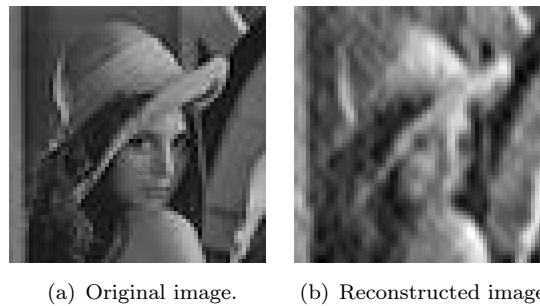


FIGURE 5.15: 64 pixels lena testing image at 1.5 scale.

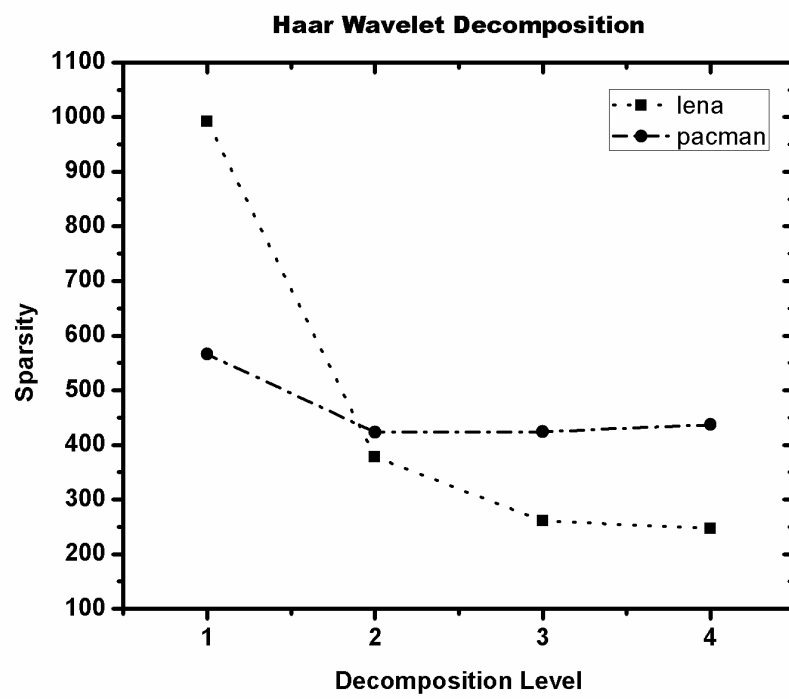


FIGURE 5.16: Haar transform for different images.

Chapter 6

Conclusion and Future Work

The aims of this work were to implement an FPGA-based architecture of a compressed sensing recovering algorithm and develop of a MATLAB Graphical User Interface as an educational compressed sensing theory exploration tool. As a result of this work generic FPGA blocks for matrix transposition and multiplication were developed; as generic structures they can be utilized in other field of applications, where matrix operations are needed. Also, an FPGA-based reconstruction architecture for compressed sensed sparse signal was implement with FPGA resource optimization.

The compressed sensing greedy algorithms are suitable for hardware implementation, thus a greedy algorithm named Compressed Sampling Matching Pursuit was used for this work. Synthesis results of the FPGA architecture show that the CoSaMP structure can be implemented on small low cost FPGA boards as the Spartan6. If the architecture is needed for a bigger application FPGA boards as the Virtex4 can be utilized.

CoSaMP algorithm requires to find the inverse of a matrix. The iterative Chebyshev-type method was proposed to carry out this task. It was found that the Chebyshev method iterations depend on the matrix data, affecting the FPGA inversion block latency. The inversion block make use of the generic matrix multiplication that is also used by the CoSaMP architecture. Thus, when CoSaMP block is implemented FPGA multipliers usage is increased just by two. The shortcomings of the Chebyshev-type matrix inversion are the memory usage, for large matrices external RAM is needed.

The FPGA implementation, with appropriate storage modifications, is robust to changes on word length and sensed signal size. Under the same word length, FPGA DSP blocks usages remains unchanged.

The MATLAB graphical user interface is useful for theoretical exploration, mainly for testing sensing matrix, transforms and signals. In the MATLAB GUI the test of sensing

matrix design and transforms under different conditions could give useful information about a specific signal that can be used for a compressed sensing application. MATLAB GUI uses the CoSaMP algorithm to reconstruct the signal but it can easily be extended for other compressed sensing algorithm.

6.1 Future Work

- Implement, based on developed blocks, a different matrix inversion method.
- Develop FPGA external RAM controller.
- Incorporate in MATLAB GUI other compressed sensing reconstruction algorithms.
- Modify FPGA architecture and MATLAB GUI in order to perform a CPU-FPGA hybrid simulations.

6.2 Publications Obtained from this Thesis

- H. D. Rico-Aniles, J. M. Ramirez-Cortes, and J. de J. Rangel-Magdaleno. **FPGA-based inversion matrix using an iterative chebyshev-type method in the context of compressed sensing**. International Instrumentation and Measurement Technology Conference (I2MTC), Montevideo, Uruguay, May 2014.

Appendix A

CoSaMP VHDL code

```
1 library IEEE;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use std.textio.all;
5
6
7 entity cosamp is
8   generic(
9     n : integer := 36;
10    m : integer := 21;
11    d : integer := 10);
12   port(
13     rst : in std_logic;
14     clk : in std_logic;
15     cosamp_str : in std_logic;
16     S1 : in std_logic_vector(m-1 downto 0);
17     S3 : in std_logic_vector(m-1 downto 0);
18     dim_i : in std_logic_vector(d-1 downto 0);
19     dim_j : in std_logic_vector(d-1 downto 0);
20     u_Di : in std_logic_vector(n-1 downto 0);
21     u_we : in std_logic;
22     u_Wa : in std_logic_vector(m-1 downto 0);
23     phi_Wa : in std_logic_vector(m-1 downto 0);
24     phit_Di : in std_logic_vector(n-1 downto 0);
25     phi_Di : in std_logic_vector(n-1 downto 0);
26     phi_we : in std_logic;
27     fsm_fin : out std_logic;
28     fsm_rdy : out std_logic
29   );
30 end cosamp;
```



```

31
32 architecture cosamp of cosamp is
33 ----- COMPONENTS
34 ----- MUXES -----
35 COMponent mux2_1_log_cosamp is
36   port (
37     IO,I1: in std_logic;
38     S : in std_logic;
39     y : out std_logic
40   );
41 end component;
42
43 component mux2_1_cosamp is
44   generic(n: integer :=18);
45   port (
46     IO,I1: in std_logic_vector(n-1 downto 0);
47     S : in std_logic;
48     y : out std_logic_vector(n-1 downto 0)
49   );
50 end component;
51
52 component mux3_1_log_cosamp is
53   port (
54     IO,I1,I2 : in std_logic;
55     S : in std_logic_Vector(1 downto 0);
56     y : out std_logic
57   );
58 end component;
59
60 component mux3_1_cosamp is
61   generic(n: integer :=18);
62   port (
63     IO,I1,I2 : in std_logic_vector(n-1 downto 0);
64     S : in std_logic_vector(1 downto 0);
65     y : out std_logic_vector(n-1 downto 0)
66   );
67 end component;
68
69 component mux4_1_cosamp is
70   generic(n: integer :=18);
71   port (
72     IO,I1,I2,I3 : in std_logic_vector(n-1 downto 0);
73     S : in std_logic_vector(1 downto 0);
74     y : out std_logic_vector(n-1 downto 0)

```

```

75 );
76 end component;
77
78 component mux4_1_log_cosamp is
79   generic(n: integer :=18);
80   port (
81     I0,I1,I2,I3 : in std_logic;
82     S : in std_logic_vector(1 downto 0);
83     y : out std_logic
84   );
85 end component;
86
87 component mux5_1_log_cosamp is
88   generic(n: integer :=18);
89   port (
90     I0,I1,I2,I3,I4 : in std_logic;
91     S : in std_logic_vector(2 downto 0);
92     y : out std_logic
93   );
94 end component;
95 ---- MULTIPLICATION-----
96 component Multiplication is
97   generic(n: integer := 36;
98     m: integer := 21;
99     d: integer := 10);
100  port(
101    rst: in std_logic;
102    clk: in std_logic;
103    str : in std_logic;
104    dim_i1 : in std_logic_vector(d-1 downto 0);
105    dim_j1 : in std_logic_vector(d-1 downto 0);
106    dim_i2 : in std_logic_vector(d-1 downto 0);
107    dim_j2 : in std_logic_vector(d-1 downto 0);
108    dat_row1 : in std_logic_vector(n-1 downto 0);
109    dat_row2 : in std_logic_vector(n-1 downto 0);
110    dat_col1 : in std_logic_vector(n-1 downto 0);
111    dat_col2 : in std_logic_vector(n-1 downto 0);
112    add_row1 : out std_logic_vector(m-1 downto 0);
113    add_row2 : out std_logic_vector(m-1 downto 0);
114    add_col1 : out std_logic_vector(m-1 downto 0);
115    add_col2 : out std_logic_vector(m-1 downto 0);
116    dat_out  : out std_logic_vector(n-1 downto 0);
117    add_out  : out std_logic_vector(m-1 downto 0);
118    diag_sign: out std_logic;

```

```

119     we      : out std_logic;
120     fin      : out std_logic;
121     rdy      : out std_logic
122   );
123 end component;
124 ----- great2small -----
125 component g2s is
126   generic(n: integer := 36;
127     m: integer := 21;
128     d: integer :=10);
129   port(
130     rst : in std_logic;
131     clk : in std_logic;
132     str : in std_logic;
133     dimj: in std_logic_vector(d-1 downto 0);
134     dat1: in std_logic_vector(n-1 downto 0);
135     dat2: in std_logic_vector(n-1 downto 0);
136     ind1: in std_logic_vector(m-1 downto 0);
137     ind2: in std_logic_vector(m-1 downto 0);
138     dat_o: out std_logic_vector(n-1 downto 0);
139     ind_o: out std_logic_vector(m-1 downto 0);
140     RA1 : out std_logic_vector(m-1 downto 0);
141     RA2 : out std_logic_vector(m-1 downto 0);
142     WA  : out std_logic_vector(m-1 downto 0);
143     we  : out std_logic;
144     fin : out std_logic;
145     rdy : out std_logic
146   );
147 end component;
148
149 component matcre is
150   generic(n: integer :=36;
151     m: integer :=21;
152     d: integer :=10);
153   port(
154     rst: in std_logic;
155     clk: in std_logic;
156     str: in std_logic;
157     dim_i : in std_logic_vector(d-1 downto 0);
158     dim_j : in std_logic_vector(d-1 downto 0);
159     S3    : in std_logic_vector(m-1 downto 0);
160     index : in std_logic_vector(m-1 downto 0);
161     dat_in      : in std_logic_vector(n-1 downto 0);
162     dat_out     : out std_logic_vector(n-1 downto 0);

```

```
163 read_address : out std_logic_vector(m-1 downto 0);
164 write_address : out std_logic_vector(m-1 downto 0);
165 sup_address   : out std_logic_vector(m-1 downto 0);
166 we           : out std_logic;
167 fin : out std_logic;
168 rdy : out std_logic
169 );
170 end component;
171
172 component ver_cop is
173   generic(n: integer := 36;
174     m: integer := 21;
175     d: integer := 10);
176   port(
177     rst: in std_logic;
178     clk: in std_logic;
179     str: in std_logic;
180     RAdd_str : in std_logic_vector(m-1 downto 0);
181     RAdd_end : in std_logic_vector(d-1 downto 0);
182     WrAdd_str: in std_logic_vector(m-1 downto 0);
183     dat_i    : in std_logic_vector(n-1 downto 0);
184     dat_o    : out std_logic_vector(n-1 downto 0);
185     read_add : out std_logic_vector(m-1 downto 0);
186     write_add: out std_logic_vector(m-1 downto 0);
187     we       : out std_logic;
188     fin      : out std_logic;
189     rdy      : out std_logic);
190 end component;
191
192 component ind_res is
193   generic(m: integer := 21;
194     d: integer := 10);
195   port(
196     rst : in std_logic;
197     clk : in std_logic;
198     str : in std_logic;
199     dim  : in std_logic_vector(d-1 downto 0);
200     address : out std_logic_vector(m-1 downto 0);
201     we     : out std_logic;
202     fin    : out std_logic;
203     rdy    : out std_logic
204   );
205 end component;
206
```

```

207 component bc is
208     generic(n: integer := 36;
209         m: integer := 21;
210         d: integer := 10);
211     port(
212         rst      : in std_logic;
213         clk      : in std_logic;
214         str      : in std_logic;
215         S3       : in std_logic_vector(m-1 downto 0);
216         dim      : in std_logic_vector(d-1 downto 0);
217         address:out std_logic_vector(m-1 downto 0);
218         dat_o    : out std_logic_vector(n-1 downto 0);
219         we       : out std_logic;
220         fin      : out std_logic;
221         rdy      : out std_logic
222     );
223 end component;
224
225 component subs_Ver is
226     generic(n: integer := 36;
227         m: integer := 21;
228         d: integer := 10);
229     port(
230         rst : in std_logic;
231         clk : in std_logic;
232         mult_we : in std_logic;
233         mult_o      : in std_logic_vector(n-1 downto 0);
234         u           : in std_logic_vector(n-1 downto 0);
235         mult_Add_o  : in std_logic_vector(m-1 downto 0);
236         rst_aux     : in std_logic;
237         subs_ver_o   : out std_logic_vector(n-1 downto 0);
238         subs_ver_add : out std_logic_vector(m-1 downto 0);
239         verifier_o  : out std_logic_vector(m-1 downto 0);
240         subs_ver_we  : out std_logic);
241 end component;
242
243 component Transposer is
244     generic(n: integer := 36;
245         m: integer := 21;
246         d: integer := 10);
247     port(
248         rst      : in std_logic;
249         clk      : in std_logic;
250         str      : in std_logic;

```

```

251   dim_i   : in std_logic_vector(d-1 downto 0);
252   dim_j   : in std_logic_vector(d-1 downto 0);
253   add_in  : out std_logic_vector(m-1 downto 0);
254   add_out : out std_logic_vector(m-1 downto 0);
255   we      : out std_logic;
256   fin     : out std_logic;
257   rdy     : out std_logic
258 );
259 end component;
260
261 component inverter is
262   generic(n: integer := 36;
263     m: integer := 21;
264     d: integer := 10
265   );
266   port(
267     rst: in std_logic;
268     clk: in std_logic;
269     str: in std_logic;
270     dim: in std_logic_vector(d-1 downto 0);
271     Do_11,Do_12 : in std_logic_vector(n-1 downto 0);
272     Do_21,Do_22 : in std_logic_vector(n-1 downto 0);
273     Do_31,Do_32 : in std_logic_vector(n-1 downto 0);
274     Do_41,Do_42 : in std_logic_vector(n-1 downto 0);
275     Do_51,Do_52 : in std_logic_vector(n-1 downto 0);
276     Do_61,Do_62 : in std_logic_vector(n-1 downto 0);
277     mult_o      : in std_logic_vector(n-1 downto 0);
278     mult_add_row1,mult_add_row2 : std_logic_vector(m-1 downto 0);
279     mult_add_col1,mult_add_col2 : std_logic_vector(m-1 downto 0);
280     mult_add_o   : std_logic_vector(m-1 downto 0);
281     diag_sign    : in std_logic;
282     mult_we      : in std_logic;
283     mult_fin     : in std_logic;
284
285     Wa1,RA11,RA12 : out std_logic_vector(m-1 downto 0);
286     Di1           : out std_logic_vector(n-1 downto 0);
287     we1           : out std_logic;
288     Wa2,RA21,RA22 : out std_logic_vector(m-1 downto 0);
289     Di2           : out std_logic_vector(n-1 downto 0);
290     we2           : out std_logic;
291     Wa3,RA31,RA32 : out std_logic_vector(m-1 downto 0);
292     Di3           : out std_logic_vector(n-1 downto 0);
293     we3           : out std_logic;
294     Wa4,RA41,RA42 : out std_logic_vector(m-1 downto 0);

```

```

295   Di4          : out std_logic_vector(n-1 downto 0);
296   we4          : out std_logic;
297   Wa5,RA51,RA52 : out std_logic_vector(m-1 downto 0);
298   Di5          : out std_logic_vector(n-1 downto 0);
299   we5          : out std_logic;
300   Wa6,RA61,RA62 : out std_logic_vector(m-1 downto 0);
301   Di6          : out std_logic_vector(n-1 downto 0);
302   we6          : out std_logic;
303   mult_dat_row1,mult_dat_row2 : out std_logic_vector(n-1 downto 0);
304   mult_dat_col1,mult_dat_col2 : out std_logic_vector(n-1 downto 0);
305   mult_str      : out std_logic;
306   fin: out std_logic;
307   rdy: out std_logic
308 );
309 end component;
310
311 component pseudoinverter is
312   generic(n: integer := 36;
313     m: integer := 21;
314     d: integer := 10);
315   port(
316     rst  : in std_logic;
317     clk  : in std_logic;
318     str  : in std_logic;
319     dim_i : in std_logic_vector(d-1 downto 0);
320     dim_j : in std_logic_vector(d-1 downto 0);
321
322     -- data for/from multiplication
323     mult_str          : out std_logic;
324     mult_dim_i1,mult_dim_j1,mult_dim_i2,mult_dim_j2 : out std_logic_vector(d-1
325       downto 0);
326     mult_dat_row1, mult_dat_row2,mult_dat_col1,dd : out std_logic_vector(n-1
327       downto 0);
328     mult_add_row1,mult_add_row2, mult_add_col1, mult_add_col2 : in
329       std_logic_vector(m-1 downto 0);
330     mult_o          : in std_logic_vector(n-1 downto 0);
331     mult_add_o       : in std_logic_vector(m-1 downto 0);
332     mult_diag_sign, mult_we,mult_fin,mult_rdy : in std_logic;
333     -- data for/from transposer
334     trans_str          : out std_logic;
335     trans_dim_i,trans_dim_j : out std_logic_vector(d-1
336       downto 0);

```

```

334   trans_add_in,trans_add_out           : in std_logic_vector(m-1 downto
      0);
335   trans_we,trans_fin,trans_rdy         : in std_logic;
336
337   --- data for/from inverter
338   inv_str: out std_logic;
339   inv_dim: out std_logic_vector(d-1 downto 0);
340   inv_Do_11,inv_Do_12,inv_Do_21,inv_Do_22,inv_Do_31,inv_Do_32,inv_Do_41,
      inv_Do_42,inv_Do_51,inv_Do_52 ,inv_Do_61,inv_Do_62, inv_mult_o : out
      std_logic_vector(n-1 downto 0);
341   inv_mult_add_row1,inv_mult_add_row2,inv_mult_add_col1,inv_mult_add_col2,
      inv_mult_add_o : out std_logic_vector(m-1 downto 0);
342   inv_diag_sign   : out std_logic;
343   inv_mult_we      : out std_logic;
344   inv_mult_fin     : out std_logic;
345   inv_Wa1,inv_RA11,inv_RA12   : in std_logic_vector(m-1 downto 0);
346   inv_Di1          : in std_logic_vector(n-1 downto 0);
347   inv_we1          : in std_logic;
348   inv_Wa2,inv_RA21,inv_RA22   : in std_logic_vector(m-1 downto 0);
349   inv_Di2          : in std_logic_vector(n-1 downto 0);
350   inv_we2          : in std_logic;
351   inv_Wa3,inv_RA31,inv_RA32   : in std_logic_vector(m-1 downto 0);
352   inv_Di3          : in std_logic_vector(n-1 downto 0);
353   inv_we3          : in std_logic;
354   inv_Wa4,inv_RA41,inv_RA42   : in std_logic_vector(m-1 downto 0);
355   inv_Di4          : in std_logic_vector(n-1 downto 0);
356   inv_we4          : in std_logic;
357   inv_Wa5,inv_RA51,inv_RA52   : in std_logic_vector(m-1 downto 0);
358   inv_Di5          : in std_logic_vector(n-1 downto 0);
359   inv_we5          : in std_logic;
360   inv_Wa6,inv_RA61,inv_RA62   : in std_logic_vector(m-1 downto 0);
361   inv_Di6          : in std_logic_vector(n-1 downto 0);
362   inv_we6          : in std_logic;
363   inv_mult_dat_row1,inv_mult_dat_row2 : in std_logic_vector(n-1 downto 0);
364   inv_mult_dat_col1,inv_mult_dat_col2 : in std_logic_vector(n-1 downto 0);
365   inv_mult_str      : in std_logic;
366   inv_fin           : in std_logic;
367   inv_rdy           : in std_logic;
368
369   -- data for/from memories
370   Wa1,RA11,RA12   : out std_logic_vector(m-1 downto 0);
371   Di1             : out std_logic_vector(n-1 downto 0);
372   we1             : out std_logic;
373   Wa2,RA21,RA22   : out std_logic_vector(m-1 downto 0);

```



```

374   Di2           : out std_logic_vector(n-1 downto 0);
375   we2           : out std_logic;
376   Wa3,RA31,RA32 : out std_logic_vector(m-1 downto 0);
377   Di3           : out std_logic_vector(n-1 downto 0);
378   we3           : out std_logic;
379   Wa4,RA41,RA42 : out std_logic_vector(m-1 downto 0);
380   Di4           : out std_logic_vector(n-1 downto 0);
381   we4           : out std_logic;
382   Wa5,RA51,RA52 : out std_logic_vector(m-1 downto 0);
383   Di5           : out std_logic_vector(n-1 downto 0);
384   we5           : out std_logic;
385   Wa6,RA61,RA62 : out std_logic_vector(m-1 downto 0);
386   Di6           : out std_logic_vector(n-1 downto 0);
387   we6           : out std_logic;
388   Wa7,RA71,RA72 : out std_logic_vector(m-1 downto 0);
389   Di7           : out std_logic_vector(n-1 downto 0);
390   we7           : out std_logic;
391   Wa8,RA81,RA82 : out std_logic_vector(m-1 downto 0);
392   Di8           : out std_logic_vector(n-1 downto 0);
393   we8           : out std_logic;
394   Do_11,Do_12 : in std_logic_vector(n-1 downto 0);
395   Do_21,Do_22 : in std_logic_vector(n-1 downto 0);
396   Do_31,Do_32 : in std_logic_vector(n-1 downto 0);
397   Do_41,Do_42 : in std_logic_vector(n-1 downto 0);
398   Do_51,Do_52 : in std_logic_vector(n-1 downto 0);
399   Do_61,Do_62 : in std_logic_vector(n-1 downto 0);
400   Do_71,Do_72 : in std_logic_vector(n-1 downto 0);
401   Do_81,Do_82 : in std_logic_vector(n-1 downto 0);
402
403
404   fin : out std_logic;
405   rdy : out std_logic
406 );
407 end component;
408
409 component RAM_S2P is
410   generic(
411     m : integer := 8; -- Numero de bits
412     n : integer := 2; -- Lineas de direccion
413     k : integer := 4 -- Numero de localidades
414   );
415   port(
416     CLK : in  std_logic;      -- Reloj maestro
417     AE  : in  std_logic_vector(n-1 downto 0); -- Direccion de escritura

```

```

418 A1 : in std_logic_vector(n-1 downto 0); -- Direccion de lectura 1
419 A2 : in std_logic_vector(n-1 downto 0); -- Direccion de lectura 2
420 WE : in std_logic;          -- Escritura
421 DE : in std_logic_vector(m-1 downto 0); -- Dato de entrada
422 D1 : out std_logic_vector(m-1 downto 0); -- Dato de salida 1
423 D2 : out std_logic_vector(m-1 downto 0) -- Dato de salida 2
424 );
425 end component;
426
427 component FSM_ROM is
428 port(
429 address : std_logic_vector(4 downto 0);
430 dat_o : out std_logic_vector(105 downto 0));
431 end component;
432
433 component cosamp_fsm is
434 generic(m: integer :=21);
435 port(
436 rst : in std_logic;
437 clk : in std_logic;
438 str : in std_logic;
439 mult_fin: in std_logic;
440 g2s_Fin : in std_logic;
441 matcre_fin: in std_logic;
442 pseud_fin : in std_logic;
443 vec_cop2_fin : in std_logic;
444 vec_cop1_fin : in std_logic;
445 ver_o : in std_logic_vector(m-1 downto 0);
446 salida : out std_logic_vector(4 downto 0);
447 fin : out std_logic;
448 rdy : out std_logic
449 );
450 end component;
451
452
453 -----
454 ----- SIGNALS -----
455 -----
456 ----- FSM_ROM SIGNALS -----
457 signal FSM_ROM_add : std_logic_vector(4 downto 0);
458 signal FSM_ROM_dat : std_logic_vector(105 downto 0);
459 ----- MULTIPLICATION SIGNALS-----
460 SIGnal MULT_str, cosamp_mult_str : std_logic;

```

```

461 signal  MULT_dim_i1,MULT_dim_j1,MULT_dim_i2,MULT_dim_j2      :
      std_logic_vector(d-1 downto 0);
462 signal  MULT_dat_row1,MULT_dat_row2,MULT_dat_col1,MULT_dat_col2 :
      std_logic_vector(n-1 downto 0);
463 signal  MULT_add_row1,MULT_add_row2,MULT_add_col1,MULT_add_col2 :
      std_logic_vector(m-1 downto 0);
464 signal  MULT_o                                              :
      std_logic_vector(n-1 downto 0);
465 signal  MULT_add_o                                          :
      std_logic_vector(m-1 downto 0);
466 signal  MULT_diag_sign,MULT_we,MULT_fin,MULT_rdy            : std_logic;
467 signal      opm1,opm5 : std_logic;
468 signal      opm2,opm3,opm4,opm6,opm7,opm8,opm9      : std_logic_vector(1 downto
      0);
469
470 ----- GREAT2SMALL SIGNALS -----
471 signal  G2s_str                      : std_logic;
472 signal  G2s_dat1,G2s_dat2            : std_logic_vector(n-1 downto 0);
473 signal  G2s_ind1,G2s_ind2            : std_logic_vector(m-1 downto 0);
474 signal  G2s_dat_o                    : std_logic_vector(n-1 downto 0);
475 signal  G2s_ind_o,G2s_RA1,G2s_RA2,G2s_WA : std_logic_vector(m-1 downto 0);
476 signal  G2s_we,G2s_fin,G2s_rdy      : std_logic;
477 signal  opm10,opm11,opm12,opm13 : std_logic_vector(1 downto 0);
478
479 ----- MATRIX CREATION SIGNALS -----
480 signal  MATCRE_str          : std_logic;
481 signal  MATCRE_dat_o        : std_logic_vector(n-1 downto 0);
482 signal  MATCRE_RA,MATCRE_WA,MATCRE_sup_add : std_logic_vector(m-1 downto 0);
483 signal  MATCRE_we ,MATCRE_fin,MATCRE_rdy   : std_logic;
484
485 ----- Vector copier 1 SIGNALS -----
486 signal  VEC_COP1_str          : std_logic;
487 signal  VEC_COP1_RA_str       : std_logic_vector(m-1
      downto 0);
488 signal  VEC_COP1_RA_end      : std_logic_vector(d-1
      downto 0);
489 signal  VEC_COP1_WA_str,VEC_COP1_dat_i,VEC_COP1_dat_o,VEC_COP1_RA,VEC_COP1_WA
      : std_logic_vector(m-1 downto 0);
490 signal  VEC_COP1_we,VEC_COP1_fin,VEC_COP1_rdy      : std_logic;
491 signal  opm14,opm15,opm16,opm17 :std_logic;
492
493 ----- Vector copier 2 SIGNALS -----
494 signal  VEC_COP2_str          : std_logic;

```

```

495 signal VEC_COP2_RA_str          : std_logic_vector(m-1
      downto 0);
496 signal VEC_COP2_RA_end          : std_logic_vector(d-1
      downto 0);
497 signal VEC_COP2_WA_str,VEC_COP2_RA,VEC_COP2_WA : std_logic_vector(m-1 downto
      0);
498 signal VEC_COP2_dat_i,VEC_COP2_dat_o          : std_logic_vector(n-1
      downto 0);
499 signal VEC_COP2_we,VEC_COP2_fin,VEC_COP2_rdy   : std_logic;
500 signal opm18,opm19,opm20,opm61                :std_logic;
501
502 ----- index reset signals -----
503 SIGnal T_RES_str : std_logic;
504 signal T_RES_WA   : std_logic_vector(m-1 downto 0);
505 signal T_RES_we,T_RES_fin,T_RES_rdy : std_logic;
506
507 SIGnal P_RES_str : std_logic;
508 signal P_RES_WA   : std_logic_vector(m-1 downto 0);
509 signal P_RES_we,P_RES_fin,P_RES_rdy : std_logic;
510
511 SIGnal OHM_RES_str : std_logic;
512 signal OHM_RES_WA   : std_logic_vector(m-1 downto 0);
513 signal OHM_RES_we,OHM_RES_fin,OHM_RES_rdy : std_logic;
514
515 ----- bc reset signals -----
516 SIGNAL B_RES_str          : std_logic;
517 signal B_RES_add          : std_logic_vector(m-1 downto 0);
518 signal B_RES_dat_o        : std_logic_vector(n-1 downto 0);
519 signal B_RES_we,B_RES_fin,B_RES_rdy : std_logic;
520
521 SIGNAL y2_RES_str          : std_logic;
522 signal y2_RES_add          : std_logic_vector(m-1 downto 0);
523 signal y2_RES_dat_o        : std_logic_vector(n-1 downto 0);
524 signal y2_RES_we,y2_RES_fin,y2_RES_rdy : std_logic;
525
526 ----- SUBS_VER SIGNALS -----
527 signal subs_Ver_rst          : std_logic;
528 signal subs_ver_o            : std_logic_vector(n-1 downto 0);
529 signal subs_ver_add,verifier_o : std_logic_vector(m-1 downto 0);
530 signal subs_ver_we           : std_logic;
531 ----- Transposer SIGNALS -----
532 signal trans_str              : std_logic;
533 signal trans_dim_i,trans_dim_j : std_logic_vector(d-1
      downto 0);

```

```

534 signal trans_add_in,trans_add_out          : std_logic_vector(m-1
      downto 0);
535 signal trans_we,trans_fin,trans_rdy        : std_logic;
536
537 ----- INVERTER SIGNALS -----
538 signal inv_str: std_logic;
539 signal inv_dim: std_logic_vector(d-1 downto 0);
540 signal inv_Do_11,inv_Do_12,inv_Do_21,inv_Do_22,inv_Do_31,inv_Do_32,inv_Do_41,
      inv_Do_42,inv_Do_51,inv_Do_52 ,inv_Do_61,inv_Do_62, inv_mult_o :
      std_logic_vector(n-1 downto 0);
541 signal inv_mult_add_row1,inv_mult_add_row2,inv_mult_add_col1,inv_mult_add_col2
      ,inv_mult_add_o : std_logic_vector(m-1 downto 0);
542 signal inv_diag_sign : std_logic;
543 signal inv_mult_we : std_logic;
544 signal inv_mult_fin : std_logic;
545 signal inv_Wa1,inv_RA11,inv_RA12 : std_logic_vector(m-1 downto 0);
546 signal inv_Di1 : std_logic_vector(n-1 downto 0);
547 signal inv_we1 : std_logic;
548 signal inv_Wa2,inv_RA21,inv_RA22 : std_logic_vector(m-1 downto 0);
549 signal inv_Di2 : std_logic_vector(n-1 downto 0);
550 signal inv_we2 : std_logic;
551 signal inv_Wa3,inv_RA31,inv_RA32 : std_logic_vector(m-1 downto 0);
552 signal inv_Di3 : std_logic_vector(n-1 downto 0);
553 signal inv_we3 : std_logic;
554 signal inv_Wa4,inv_RA41,inv_RA42 : std_logic_vector(m-1 downto 0);
555 signal inv_Di4 : std_logic_vector(n-1 downto 0);
556 signal inv_we4 : std_logic;
557 signal inv_Wa5,inv_RA51,inv_RA52 : std_logic_vector(m-1 downto 0);
558 signal inv_Di5 : std_logic_vector(n-1 downto 0);
559 signal inv_we5 : std_logic;
560 signal inv_Wa6,inv_RA61,inv_RA62 : std_logic_vector(m-1 downto 0);
561 signal inv_Di6 : std_logic_vector(n-1 downto 0);
562 signal inv_we6 : std_logic;
563 signal inv_mult_dat_row1,inv_mult_dat_row2 : std_logic_vector(n-1 downto 0);
564 signal inv_mult_dat_col1,inv_mult_dat_col2 : std_logic_vector(n-1 downto 0);
565 signal inv_mult_str : std_logic;
566 signal inv_fin : std_logic;
567 signal inv_rdy : std_logic;
568 ----- PSEUDOINVERTER SIGNALS -----
569 signal PSEUD_mult_str : std_logic;
570 signal PSEUD_mult_dim_i1,PSEUD_mult_dim_j1,PSEUD_mult_dim_i2,
      PSEUD_mult_dim_j2 : std_logic_vector(d-1 downto 0);
571 signal PSEUD_mult_row1,PSEUD_mult_row2,PSEUD_mult_col1,PSEUD_mult_Col2 :
      std_logic_vector(n-1 downto 0);

```

```

572
573
574
575 signal PSEUD_str : std_logic;
576
577 -- data for/from memories
578 signal PSEUD_Wa1,PSEUD_RA11,PSEUD_RA12 : std_logic_vector(m-1 downto 0);
579 signal PSEUD_Di1 : std_logic_vector(n-1 downto 0);
580 signal PSEUD_we1 : std_logic;
581 signal PSEUD_Wa2,PSEUD_RA21,PSEUD_RA22 : std_logic_vector(m-1 downto 0);
582 signal PSEUD_Di2 : std_logic_vector(n-1 downto 0);
583 signal PSEUD_we2 : std_logic;
584 signal PSEUD_Wa3,PSEUD_RA31,PSEUD_RA32 : std_logic_vector(m-1 downto 0);
585 signal PSEUD_Di3 : std_logic_vector(n-1 downto 0);
586 signal PSEUD_we3 : std_logic;
587 signal PSEUD_Wa4,PSEUD_RA41,PSEUD_RA42 : std_logic_vector(m-1 downto 0);
588 signal PSEUD_Di4 : std_logic_vector(n-1 downto 0);
589 signal PSEUD_we4 : std_logic;
590 signal PSEUD_Wa5,PSEUD_RA51,PSEUD_RA52 : std_logic_vector(m-1 downto 0);
591 signal PSEUD_Di5 : std_logic_vector(n-1 downto 0);
592 signal PSEUD_we5 : std_logic;
593 signal PSEUD_Wa6,PSEUD_RA61,PSEUD_RA62 : std_logic_vector(m-1 downto 0);
594 signal PSEUD_Di6 : std_logic_vector(n-1 downto 0);
595 signal PSEUD_we6 : std_logic;
596 signal PSEUD_Wa7,PSEUD_RA71,PSEUD_RA72 : std_logic_vector(m-1 downto 0);
597 signal PSEUD_Di7 : std_logic_vector(n-1 downto 0);
598 signal PSEUD_we7 : std_logic;
599 signal PSEUD_Wa8,PSEUD_RA81,PSEUD_RA82 : std_logic_vector(m-1 downto 0);
600 signal PSEUD_Di8 : std_logic_vector(n-1 downto 0);
601 signal PSEUD_we8 : std_logic;
602
603 signal PSEUD_fin : std_logic;
604 signal PSEUD_rdy : std_logic;
605
606 ----- PHI MATRIX SIGNALS -----
607 signal phi_D01,phi_D02 : std_logic_vector(n-1 downto 0);
608 signal phi_RA1 : std_logic_vector(m-1 downto 0);
609 signal opm21 : std_logic;
610 ----- PHIT MATRIX SIGNALS -----
611 signal phit_D01,phit_D02 : std_logic_vector(n-1 downto 0);
612 ----- U vector -----
613 signal u_D01,u_D02 : std_logic_vector(n-1 downto 0);
614 signal u_RA1 : std_logic_vector(m-1 downto 0);
615 signal opm22 : std_logic;

```

```

616 ----- r vector -----
617 signal r_D01,r_D02 : std_logic_vector(n-1 downto 0);
618 signal r_we,opm23 : std_logic;
619
620 signal r_di : std_logic_vector(n-1 downto 0);
621 signal r_wa : std_logic_vector(m-1 downto 0);
622 signal opmr1,opmr2,opmr3,r_we_aux : std_logic;
623
624 ----- a vector -----
625 signal a_D01,a_D02,a_Di : std_logic_vector(n-1 downto 0);
626 signal a_WA,a_RA1,a_RA2 : std_logic_vector(m-1 downto 0);
627 signal a_we : std_logic;
628 signal opm25,opm26 : std_logic;
629 signal opm27,opm24,opm28 : std_logic_vector(1 downto 0);
630 ----- b vector -----
631 signal b_wa,b_ra1 : std_logic_vector(m-1 downto 0);
632 signal b_we,opm60 : std_logic;
633 signal opm30,opm29,opm31 : std_logic_vector(1 downto 0);
634 signal b_D01,b_D02,b_Di : std_logic_vector(n-1 downto 0);
635 ----- T support vector -----
636 signal T_D01,T_D02,T_WA,T_RA1,T_Di : std_logic_vector(m-1 downto 0);
637 signal T_We : std_logic;
638 signal opm33,opm34,opm32,opm35 : std_logic_vector(1 downto 0);
639 ----- OHM support vector -----
640 signal OHM_D01,OHM_D02,OHM_RA1,OHM_RA2,OHM_DI,OHM_Wa : std_logic_vector(m-1
    downto 0);
641 signal OHM_we,opm36,opm57,OPM56,opm62 : std_logic;
642 signal opm37 : std_logic_vector(1 downto 0);
643 ----- P support vector -----
644 signal P_D01,P_D02, P_RA1,P_RA2,P_WA,P_Di : std_logic_vector(m-1 downto 0);
645 signal P_WE,opm38,opm39,opm58,opm59 : std_logic;
646 signal opm40 : std_logic_vector(1 downto 0);
647 ----- MEM1 -----
648 signal Do11,Do12 : std_logic_vector(n-1 downto 0);
649 signal RA11,RA12 : std_logic_vector(m-1 downto 0);
650 signal opm41,opm42 : std_logic;
651 ----- MEM2 -----
652 signal D021,D022,Di2 : std_logic_vector(n-1 downto 0);
653 signal WA2,RA21,RA22 : std_logic_vector(m-1 downto 0);
654 signal We2,opm44,opm45 : std_logic;
655 signal opm43,opm47 : std_logic_vector(1 downto 0);
656 signal opm46 : std_logic_vector(2 downto 0);
657 ----- MEM3 -----
658 signal D031,D032,Di3 : std_logic_vector(n-1 downto 0);

```

```

659 signal WA3,RA31,RA32 : std_logic_vector(m-1 downto 0);
660 signal we3,opm50 : std_logic;
661 signal opm48,opm49,opm51,opm52 : std_logic_vector(1 downto 0);
662 ----- MEM4 -----
663 signal Do41,Do42 : std_logic_vector(n-1 downto 0);
664 ----- MEM5 -----
665 signal Do51,Do52 : std_logic_vector(n-1 downto 0);
666 ----- MEM6 -----
667 signal Do61,Do62 : std_logic_vector(n-1 downto 0);
668 ----- MEM7 -----
669 signal Do71,Do72 : std_logic_vector(n-1 downto 0);
670 ----- MEM8 -----
671 signal Do81,Do82,Di8 : std_logic_vector(n-1 downto 0);
672 signal WA8 : std_logic_vector(m-1 downto 0);
673 signal We8,opm53,opm55 : std_logic;
674 signal opm54 : std_logic_vector(1 downto 0);
675
676 ----- AUXILIARES -----
677 signal aux : std_logic_vector(n-1 downto 0);
678 signal dim1_aux : std_logic_vector(d-1 downto 0);
679 signal aux_wa : std_logic_vector(m-1 downto 0);
680 begin
681
682 aux <= (others => '0');
683 dim1_aux <= aux(d-1 downto 1) & '1'; --- dimension = 1
684 ----- FSM -----
685 cosamp_fsm1 : cosamp_fsm generic map(m)port map(rst,clk,cosamp_str,mult_fin,
        g2s_Fin,matcre_fin,pseud_fin,vec_cop2_fin,vec_cop1_fin,verifier_o,
        fsm_rom_add,fsm_fin,fsm_rdy);
686
687 ----- MEMORIES -----
688 --- FSM_ROM
689 opm61 <= fsm_rom_dat(105);
690 opm62 <= fsm_rom_dat(104);
691 opm60 <= FSM_ROM_dat(103);
692 FSM_ROM1 : FSM_ROM port map(FSM_ROM_Add,FSM_ROM_dat);
693 COSAMP_mult_str <= FSM_ROM_dat(102);
694 opm1 <= FSM_ROM_dat(101); opm2 <= FSM_ROM_dat(100 downto 99); opm3
        <= FSM_ROM_dat(98 downto 97);
695 opm4 <= FSM_ROM_dat(96 downto 95); opm5 <= FSM_ROM_dat(94); opm6 <=
        FSM_ROM_dat(93 downto 92);
696 opm7 <= FSM_ROM_dat(91 downto 90); opm8 <= FSM_ROM_dat(89 downto 88); opm9 <=
        FSM_ROM_dat(87 downto 86);
697 g2s_Str <= FSM_ROM_dat(85);

```



```

698   opm10 <= FSM_ROM_dat(84 downto 83 ) ; opm11 <= FSM_ROM_dat(82 downto 81); opm12
      <= FSM_ROM_dat(80 downto 79);
699   opm13 <= FSM_ROM_dat(78 downto 77) ; matcre_str <= FSM_ROM_dat(76);
      vec_cop1_str <= FSM_ROM_dat(75);
700   opm14 <= FSM_ROM_dat(74);          opm15 <= FSM_ROM_dat(73);          opm16
      <= FSM_ROM_dat(72);
701   opm17 <= FSM_ROM_dat(71);          vec_cop2_str <= FSM_ROM_dat(70);
702   opm18 <= FSM_ROM_dat(69);          opm19 <= FSM_ROM_dat(68);          opm20
      <= FSM_ROM_dat(67);
703   T_res_str <= FSM_ROM_dat(66);      P_res_str <= FSM_ROM_dat(65);
      ohm_res_str <= FSM_ROM_dat(64);
704   y2_res_str <= FSM_ROM_dat(63);      b_res_str <= FSM_ROM_dat(62);
      subs_ver_rst <= FSM_ROM_dat(61);
705   pseud_str <= FSM_ROM_dat(60);
706   opm21 <= FSM_ROM_dat(59);          opm22 <= FSM_ROM_dat(58);
707   opm23 <= FSM_ROM_dat(57);          opm24 <= FSM_ROM_dat(56 downto 55); opm25
      <= FSM_ROM_dat(54);
708   opm26 <= FSM_ROM_dat(53);          opm27 <= FSM_ROM_dat(52 downto 51); opm28
      <= FSM_ROM_dat(50 downto 49);
709   opm29 <= FSM_ROM_dat(48 downto 47); opm30 <= FSM_ROM_dat(46 downto 45); opm31
      <= FSM_ROM_dat(44 downto 43);
710   opm32 <= FSM_ROM_dat(42 downto 41); opm33 <= FSM_ROM_dat(40 downto 39); opm34
      <= FSM_ROM_dat(38 downto 37);
711   opm35 <= FSM_ROM_dat(36 downto 35); opm36 <= FSM_ROM_dat(34);          opm37
      <= FSM_ROM_dat(33 downto 32);
712   opm38 <= FSM_ROM_dat(31);          opm39 <= FSM_ROM_dat(30);          opm40
      <= FSM_ROM_dat(29 downto 28);
713   opm41 <= FSM_ROM_dat(27);          opm42 <= FSM_ROM_dat(26);          opm43
      <= FSM_ROM_dat(25 downto 24);
714   opm44 <= FSM_ROM_dat(23);          opm45 <= FSM_ROM_dat(22);          opm46
      <= FSM_ROM_dat(21 downto 19);
715   opm47 <= FSM_ROM_dat(18 downto 17); opm48 <= FSM_ROM_dat(16 downto 15); opm49
      <= FSM_ROM_dat(14 downto 13);
716   opm50 <= FSM_ROM_dat(12);          opm51 <= FSM_ROM_dat(11 downto 10); opm52
      <= FSM_ROM_dat(9 downto 8);
717   opm53 <= FSM_ROM_dat(7);          opm54 <= FSM_ROM_dat(6 downto 5);   opm55
      <= FSM_ROM_dat(4);
718   opm56 <= FSM_ROM_dat(3);          opm57 <= FSM_ROM_dat(2);          opm58
      <= FSM_ROM_dat(1);
719   opm59 <= FSM_ROM_dat(0);
720   --- PHI MATRIX
721   phi_ram1      : ram_s2p generic map(n,m,532)port map(clk,phi_wa,phi_RA1,
      mult_add_row2,phi_we,phi_di,phi_D01,PHI_D02);

```

```

722 cosamp_mux21 : mux2_1_cosamp generic map(m)port map(matcre_RA,MULT_add_row1,
      opm21,phi_RA1);
723 --- TRANSPOSE PHI MATRIX
724 phit_ram1 : ram_s2p generic map(n,m,532)port map(clk,phi_wa,MULT_add_row1,
      MULT_add_row2,phi_We,phit_di,phit_D01,phit_D02);
725 --- U VECTOR (measurements)
726 u_ram1 : ram_s2p generic map(n,m,19)port map(clk,u_wa,u_RA1,
      mult_add_col2,u_we,u_di,u_D01,u_D02);
727 cosamp_mux22 : mux2_1_cosamp generic map(m)port map(mult_add_col1,subs_ver_add
      ,opm22,u_RA1);
728 --- R vector residuo
729 r_ram1 : ram_s2p generic map(n,m,19)port map(clk,r_wa,mult_add_col1,
      mult_add_col2,r_we_aux,r_di,r_d01,r_d02);
730 cosamp_mux23 : mux2_1_log_cosamp port map('0',subs_Ver_we,opm23,r_We);
731
732
733
734 mux_r : mux2_1_cosamp generic map(m)port map(u_wa,subs_ver_add,opmr1,r_wa)
      ;
735 mux_r2 : mux2_1_log_cosamp port map(u_we,r_we,opmr2,r_we_aux);
736 mux_r3 : mux2_1_cosamp generic map(n) port map(u_di,subs_ver_o,opmr3,r_Di);
737
738
739 --- a vector
740 a_ram1 : ram_s2p generic map(n,m,28)port map(clk,a_WA,a_RA1,a_RA2,a_WE,
      A_Di,a_D01,a_D02);
741 cosamp_mux24 : mux4_1_cosamp generic map(m)port map(vec_Cop2_wa,g2s_wa,
      b_res_add,ohm_do2,opm24,a_WA);
742 cosamp_mux25 : mux2_1_cosamp generic map(m)port map(mult_Add_col1,g2s_RA1,
      opm25,a_RA1);
743 cosamp_mux26 : mux2_1_cosamp generic map(m)port map(mult_Add_col2,g2s_RA2,
      opm26,a_RA2);
744 cosamp_mux27 : mux4_1_log_cosamp port map('0',vec_cop2_we,g2s_we,b_res_we,
      opm27,a_we);
745 cosamp_mux28 : mux3_1_cosamp generic map(n)port map(vec_cop2_dat_o,g2s_dat_o,
      b_res_dat_o,opm28,a_Di);
746 --- b vector
747 b_ram1 : ram_s2p generic map(n,m,28)port map(clk,b_WA,b_ra1,g2s_ra2,b_we
      ,b_Di,b_Do1,b_Do2);
748 cosamp_mux29 : mux3_1_cosamp generic map(m)port map(b_res_add,T_d01,g2s_wa,
      opm29,b_WA);
749 cosamp_mux30 : mux4_1_log_cosamp port map('0',b_res_we,mult_we,g2s_we,opm30,
      b_we);

```

```

750 cosamp_mux31 : mux3_1_cosamp generic map(n)port map(b_res_dat_o,mult_o,
      G2s_dat_o,opm31,b_Di);
751 cosamp_mux60 : mux2_1_cosamp generic map(m)port map(vec_cop2_ra,G2s_ra1,opm60,
      b_ra1);
752
753 ---- T support vector
754 T_ram          : ram_s2p generic map(m,m,28)port map(clk,T_WA,T_RA1,g2s_ra2,T_we
      ,T_Di,T_D01,T_D02);
755 cosamp_mux32 : mux3_1_cosamp generic map(m)port map(T_RES_wa,g2s_WA,
      vec_cop1_Wa,opm32,T_WA);
756 cosamp_mux33 : mux3_1_cosamp generic map(m)port map(g2s_RA1,matcre_sup_add,
      mult_add_o,opm33,T_RA1);
757 cosamp_mux34 : mux4_1_log_cosamp port map('0',T_res_we,g2s_we,vec_Cop1_we,
      opm34,T_we);
758 cosamp_mux35 : mux3_1_cosamp generic map(m)port map(T_res_wa,g2s_ind_o,
      vec_Cop1_dat_o,opm35,T_Di);
759 ----- OHM support vector
760 OHM_ram1       : ram_s2p generic map(m,m,28)port map(clk,OHM_WA,OHM_RA1,ohm_RA2,
      OHM_we,OHM_DI,OHM_D01,OHM_D02);
761 cosamp_mux36 : mux2_1_cosamp generic map(m)port map(G2s_RA1,vec_cop1_RA,opm36,
      OHM_RA1);
762 cosamp_mux37 : mux3_1_log_cosamp port map('0',OHM_RES_we,G2s_we,opm37,OHM_we);
763 cosamp_mux56 : mux2_1_cosamp generic map(m)port map(OHM_RES_wa,G2s_WA,opm56,
      OHM_WA);
764 cosamp_mux57 : mux2_1_cosamp generic map(m)port map(OHM_RES_WA,g2s_ind_o,opm57
      ,OHM_DI);
765 cosamp_mux62 : mux2_1_cosamp generic map(m)port map(g2s_ra2,vec_cop2_wa,opm62,
      ohm_ra2);
766 ---- P support vector
767 p_ram1         : ram_s2p generic map(m,m,28)port map(clk,P_WA,P_RA1,P_RA2,P_we,
      P_di,P_D01,P_D02);
768 cosamp_mux38 : mux2_1_cosamp generic map(m)port map(G2s_RA1,vec_cop2_WA,opm38,
      P_RA1);
769 cosamp_mux39 : mux2_1_cosamp generic map(m)port map(G2s_RA2,vec_cop1_RA,opm39,
      P_RA2);
770 cosamp_mux40 : mux3_1_log_cosamp port map('0',P_RES_we,g2s_we,opm40,P_we);
771 cosamp_mux58 : mux2_1_cosamp generic map(m)port map(P_Res_wa,g2s_wa,opm58,P_WA
      );
772 cosamp_mux59 : mux2_1_cosamp generic map(m)port map(P_res_Wa,g2s_ind_o,opm59,
      P_Di);
773 ---- MEM1
774 MEM1_ram       : ram_s2p generic map(n,m,532)port map(clk,pseud_WA1,RA11,RA12,
      pseud_we1,pseud_di1,D011,D012);

```

```

775 cosamp_mux41 : mux2_1_cosamp generic map(m)port map(pseud_RA11,mult_add_row1,
      opm41,RA11);
776 cosamp_mux42 : mux2_1_cosamp generic map(m)port map(pseud_RA12,mult_add_row2,
      opm42,RA12);
777 ---- MEM2
778 MEM2_ram1 : ram_s2p generic map(n,m,532)port map(clk,WA2,RA21,RA22,WE2,Di2,
      Do21,Do22);
779 cosamp_mux43 : mux4_1_cosamp generic map(m)port map(y2_res_add,PSEUD_WA2,P_Do1
      ,g2s_Wa,opm43,WA2);
780 cosamp_mux44 : mux2_1_cosamp generic map(m)port map(PSEUD_RA21,G2s_ra1,opm44,
      RA21);
781 cosamp_mux45 : mux2_1_cosamp generic map(m)port map(PSEUD_RA22,G2s_RA2,opm45,
      RA22);
782 cosamp_mux46 : mux5_1_log_cosamp port map('0',y2_res_we,PseUD_WE2,vec_cop2_we,
      G2s_we,opm46,we2);
783 cosamp_mux47 : mux4_1_cosamp generic map(n)port map(y2_res_dat_o,PSEUD_di2,
      vec_cop2_dat_o,g2s_dat_o,opm47,Di2);
784 ---- MEM3
785 MEM3_ram2 : ram_s2p generic map(n,m,532)port map(clk,WA3,RA31,RA32,We3,Di3,
      Do31,Do32);
786 cosamp_mux48 : mux3_1_cosamp generic map(m)port map(mult_add_o,g2s_wa,
      PSEUD_wa3,opm48,WA3);
787 cosamp_mux49 : mux3_1_cosamp generic map(m)port map(g2s_RA1,PSEUD_RA31,P_do1,
      opm49,RA31);
788 cosamp_mux50 : mux2_1_cosamp generic map(m)port map(g2s_RA2,PSEUD_RA32,opm50,
      RA32);
789 cosamp_mux51 : mux4_1_log_cosamp port map('0',mult_we,g2s_we,PSEUD_we3,opm51,
      WE3);
790 cosamp_mux52 : mux3_1_cosamp generic map(n)port map(mult_o,g2s_dat_o,pseud_di3
      ,opm52,Di3);
791
792 ---- MEM4
793 MEM4_ram1 : ram_s2p generic map(n,m,532)port map(clk,PSEUD_WA4,PSEUD_RA41,
      PSEUD_RA42,PSEUD_WE4,PSEUD_DI4,D041,D042);
794 ---- MEM5
795 MEM5_ram1 : ram_s2p generic map(n,m,532)port map(clk,PSEUD_WA5,PSEUD_RA51,
      PSEUD_RA52,PSEUD_WE5,PSEUD_DI5,D051,D052);
796 ---- MEM6
797 MEM6_ram1 : ram_s2p generic map(n,m,532)port map(clk,PSEUD_WA6,PSEUD_RA61,
      PSEUD_RA62,PSEUD_WE6,PSEUD_DI6,D061,D062);
798 ---- MEM7
799 MEM7_ram1 : ram_s2p generic map(n,m,532)port map(clk,PSEUD_WA7,PSEUD_RA71,
      PSEUD_RA72,PSEUD_WE7,PSEUD_DI7,D071,D072);
800 ---- MEM8

```

```

801 MEM8_ram1      : ram_s2p generic map(n,m,532)port map(clk,WA8,PSEUD_RA81,
      PSEUD_RA82,WE8,DI8,D081,D082);
802 cosamp_mux53 : mux2_1_cosamp generic map(m)port map(matcre_WA,PSEUD_Wa8,opm53,
      WA8);
803 cosamp_mux54 : mux3_1_log_Cosamp port map('0',matcre_we,PSEUD_we8,opm54,We8);
804 cosamp_mux55 : mux2_1_cosamp generic map(n)port map(matcre_dat_o,PSEUD_di8,
      opm55,Di8);
805 ----- PSEUDOINVERTER-----
806 pseudoinverter1: pseudoinverter generic map(n,m,d)port map(rst,clk,PSEUD_str,
      dim_i,S3(d-1 downto 0),PSEUD_mult_str,PSEUD_mult_dim_i1,PSEUD_mult_dim_j1,
      PSEUD_mult_dim_i2,PSEUD_mult_dim_j2,PSEUD_MULT_row1, PSEUD_MULT_row2,
      PSEUD_MULT_col1,PSEUD_MULT_col2,MULT_add_row1,MULT_add_row2,MULT_add_col1,
      MULT_add_col2,MULT_o,MULT_add_o,MULT_diag_sign, MULT_we,MULT_fin,MULT_rdy,
      trans_str,trans_dim_i,trans_dim_j,trans_add_in,trans_add_out,trans_we,
      trans_fin,trans_rdy,inv_str,inv_dim,inv_Do_11,inv_Do_12,inv_Do_21,inv_Do_22
      ,inv_Do_31,inv_Do_32,inv_Do_41,inv_Do_42,inv_Do_51,inv_Do_52 ,inv_Do_61,
      inv_Do_62, inv_mult_o,inv_mult_add_row1,inv_mult_add_row2,inv_mult_add_col1
      ,inv_mult_add_col2,inv_mult_add_o,inv_diag_sign,inv_mult_we,inv_mult_fin,
      inv_Wa1,inv_RA11,inv_RA12,inv_Di1,inv_we1,inv_Wa2,inv_RA21,inv_RA22,inv_Di2
      ,inv_we2,inv_Wa3,inv_RA31,inv_RA32,inv_Di3,inv_we3,inv_Wa4,inv_RA41,
      inv_RA42,inv_Di4,inv_we4,inv_Wa5,inv_RA51,inv_RA52,inv_Di5,inv_we5,inv_Wa6,
      inv_RA61,inv_RA62,inv_Di6,inv_we6,inv_mult_dat_row1,inv_mult_dat_row2,
      inv_mult_dat_col1,inv_mult_dat_col2,inv_mult_str,inv_fin,inv_rdy,PSEUD_Wa1,
      PSEUD_RA11,PSEUD_RA12,PSEUD_Di1,PSEUD_we1,PSEUD_Wa2,PSEUD_RA21,PSEUD_RA22,
      PSEUD_Di2,PSEUD_we2,PSEUD_Wa3,PSEUD_RA31,PSEUD_RA32,PSEUD_Di3,PSEUD_we3,
      PSEUD_Wa4,PSEUD_RA41,PSEUD_RA42,PSEUD_Di4,PSEUD_we4,PSEUD_Wa5,PSEUD_RA51,
      PSEUD_RA52,PSEUD_Di5,PSEUD_we5,PSEUD_Wa6,PSEUD_RA61,PSEUD_RA62,PSEUD_Di6,
      PSEUD_we6,PSEUD_Wa7,PSEUD_RA71,PSEUD_RA72,PSEUD_Di7,PSEUD_we7,PSEUD_Wa8,
      PSEUD_RA81,PSEUD_RA82,PSEUD_Di8,PSEUD_we8,Do11,Do12,Do21,Do22,Do31,Do32,
      Do41,Do42,Do51,Do52,Do61,Do62,Do71,Do72 ,Do81,Do82,PSEUD_fin,PSEUD_rdy);
807 ----- INVERTER -----
808 inverter1      : inverter  generic map(n,m,d)port map(rst,clk,inv_str,inv_dim,
      inv_Do_11,inv_Do_12,inv_Do_21,inv_Do_22,inv_Do_31,inv_Do_32,inv_Do_41,
      inv_Do_42,inv_Do_51,inv_Do_52,inv_Do_61,inv_Do_62,inv_mult_o,
      inv_mult_add_row1,inv_mult_add_row2,inv_mult_add_col1,inv_mult_add_col2,
      inv_mult_add_o,inv_diag_sign,inv_mult_we,inv_mult_fin,inv_Wa1,inv_RA11,
      inv_RA12,inv_Di1,inv_we1,inv_Wa2,inv_RA21,inv_RA22,inv_Di2,inv_we2,inv_Wa3,
      inv_RA31,inv_RA32,inv_Di3,inv_we3,inv_Wa4,inv_RA41,inv_RA42,inv_Di4,inv_we4
      ,inv_Wa5,inv_RA51,inv_RA52,inv_Di5,inv_we5,inv_Wa6,inv_RA61,inv_RA62,
      inv_Di6,inv_we6,inv_mult_dat_row1,inv_mult_dat_row2,inv_mult_dat_col1,
      inv_mult_dat_col2,inv_mult_str,inv_fin,inv_rdy);
809 ----- TRANSPOSER -----

```

```

810 transposer1      : transposer generic map(n,m,d)port map(rst,clk,trans_str,
    trans_dim_i,trans_dim_j,trans_add_in,trans_add_out,trans_we,trans_fin,
    trans_rdy);
811 ----- SUBTRACTOR VERIFIER -----
812 subs_Ver1        : subs_Ver generic map(n,m,d)port map(rst,clk,MULT_we,MULT_o,
    u_D01,MULT_add_o,subs_Ver_rst,subs_ver_o,subs_ver_add,verifier_o,
    subs_ver_we);
813 ----- VECTOR RESET -----
814 B_reset1          : bc generic map(n,m,d)port map(rst,clk,B_RES_str,S3,dim_j,
    B_RES_add,B_RES_dat_o,B_RES_we,B_RES_fin,B_RES_rdy);
815 Y2_reset1         : bc generic map(n,m,d)port map(rst,clk,Y2_RES_str,S3,dim_j,
    y2_RES_add,y2_RES_dat_o,y2_RES_we,y2_RES_fin,y2_RES_rdy);
816
817 ----- INDEX RESET -----
818 T_reset1          : ind_res generic map(m,d)port map(rst,clk,T_RES_str,dim_j,
    T_RES_WA,T_RES_we,T_RES_fin,T_RES_rdy);
819 P_reset1          : ind_res generic map(m,d)port map(rst,clk,P_RES_str,dim_j,
    P_RES_WA,P_RES_we,P_RES_fin,P_RES_rdy);
820 OHM_reset1        : ind_res generic map(m,d)port map(rst,clk,OHM_RES_str,dim_j,
    OHM_RES_WA,OHM_RES_we,OHM_RES_fin,OHM_RES_rdy);
821 ----- vector copier assign -----
822 vec_Cop_support1 : ver_cop generic map(m,m,d)port map(rst,clk,VEC_COP1_str,
    VEC_COP1_RA_str,VEC_COP1_RA_end,VEC_COP1_WA_str,VEC_COP1_dat_i,
    VEC_COP1_dat_o,VEC_COP1_RA,VEC_COP1_WA,VEC_COP1_we,VEC_COP1_fin,
    VEC_COP1_rdy);
823 cosamp_mux14       : mux2_1_cosamp generic map(m)port map(aux(m-1 downto 0),s1,
    opm14,VEC_COP1_RA_str);
824 cosamp_mux15       : mux2_1_cosamp generic map(d)port map(s1(d-1 downto 0),s3(d
    -1 downto 0),opm15,VEC_COP1_RA_end);
825 cosamp_mux16       : mux2_1_cosamp generic map(m)port map(aux(m-1 downto 0),s1,
    opm16,VEC_COP1_WA_str);
826 cosamp_mux17       : mux2_1_cosamp generic map(m)port map(P_D02,OHM_D01,opm17,
    VEC_COP1_dat_i);
827
828 vec_Cop_dat2        : ver_cop generic map(n,m,d)port map(rst,clk,VEC_COP2_str,
    VEC_COP2_RA_str,vec_cop2_RA_end,VEC_COP2_WA_str,VEC_COP2_dat_i,
    VEC_COP2_dat_o,VEC_COP2_RA,VEC_COP2_WA,VEC_COP2_we,VEC_COP2_fin,
    VEC_COP2_rdy);
829 cosamp_mux18       : mux2_1_cosamp generic map(m)port map(aux(m-1 downto 0),s1,
    opm18,VEC_COP2_RA_str);
830 cosamp_mux19       : mux2_1_cosamp generic map(m)port map(aux(m-1 downto 0),s1,
    opm19,VEC_COP2_WA_str);
831 cosamp_mux20       : mux2_1_cosamp generic map(n)port map(b_D01,D031,opm20,
    VEC_COP2_dat_i);

```

```

832 cosamp_mux61      : mux2_1_cosamp generic map(d)port map(dim_j,s1(d-1 downto 0)
      ,opm61,vec_cop2_RA_end);
833 ----- Matrix creation assign -----
834 matcre1          : matcre generic map(n,m,d)port map(rst,clk,MATCRE_str,dim_i,
      dim_j,s3,T_D01,phi_D01,MATCRE_dat_o,MATCRE_RA,MATCRE_WA,MATCRE_sup_add,
      MATCRE_we ,MATCRE_fin,MATCRE_rdy);
835 ----- Great2small assign -----
836 G2s1             : g2s generic map(n,m,d)port map(rst,clk,G2s_str,dim_j,
      G2s_dat1,G2s_dat2,G2s_ind1,G2s_ind2,G2s_dat_o,G2s_ind_o,G2s_RA1,G2s_RA2,
      G2s_WA,G2s_we,G2s_fin,G2s_rdy);
837 cosamp_mux10     : mux4_1_cosamp generic map(n)port map(D031,a_D01,D021,b_do1,
      opm10,G2s_dat1);
838 cosamp_mux11     : mux4_1_cosamp generic map(n)port map(D032,a_D02,D022,b_do2,
      opm11,G2S_dat2);
839 cosamp_mux12     : mux3_1_cosamp generic map(m)port map(T_D01,P_D01,OHM_D01,
      opm12,G2S_ind1);
840 cosamp_mux13     : mux3_1_cosamp generic map(m)port map(T_D02,P_D02,OHM_D02,
      opm13,G2S_ind2);
841 ----- multiplication-----
842 multiplication1 : Multiplication generic map(n,m,d)port map(rst,clk,MULT_str,
      MULT_dim_i1,MULT_dim_j1,MULT_dim_i2,MULT_dim_j2,MULT_dat_row1,MULT_dat_row2
      ,MULT_dat_col1,MULT_dat_col2,MULT_add_row1,MULT_add_row2,MULT_add_col1,
      MULT_add_col2,MULT_o,MULT_add_o,MULT_diag_sign,MULT_we,MULT_fin,MULT_rdy);
843 cosamp_mux1      : mux2_1_log_cosamp port map(COSAMP_mult_str,PSEUD_mult_Str,
      opm1,MULT_str);
844 cosamp_mux2      : mux4_1_cosamp generic map(d)port map(dim_j,
      PSEUD_mult_dim_i1,s3(d-1 downto 0),dim_i,opm2,MULT_dim_i1);
845 cosamp_mux3      : mux3_1_cosamp generic map(d)port map(dim_i,PSEUD_mult_dim_j1,
      dim_j,opm3,MULT_dim_j1);
846 cosamp_mux4      : mux3_1_cosamp generic map(d)port map(dim_i,
      PSEUD_mult_dim_i2,dim_j,opm4,MULT_dim_i2);
847 cosamp_mux5      : mux2_1_cosamp generic map(d)port map(dim1_aux,
      PSEUD_mult_dim_j2,opm5,MULT_dim_j2);
848 cosamp_mux6      : mux4_1_cosamp generic map(n)port map(phit_D01,
      PSEUD_mult_row1,D011,phi_do1,opm6,MULT_dat_row1);
849 cosamp_mux7      : mux4_1_cosamp generic map(n)port map(phit_D02,
      PSEUD_mult_row2,D012,phi_do2,opm7,MULT_dat_row2);
850 cosamp_mux8      : mux4_1_cosamp generic map(n)port map(r_do1,PSEUD_mult_col1,
      u_do1,a_do1,opm8,MULT_dat_col1);
851 cosamp_mux9      : mux4_1_cosamp generic map(n)port map(r_do2,PSEUD_mult_Col2,
      u_do2,a_do2,opm9,MULT_dat_col2);
852 end cosamp;

```

Bibliography

- [1] Single-pixel imaging via compressed sampling. *IEEE Signal Processing Magazine*, 25(2):83–91, March 2008.
- [2] R. Baraniuk. Compressive sensing. *IEEE Proc Mag*, 24(4):118–124, 2007.
- [3] H. Nyquist. Certain topics in telegraph transmission theory. *Proceedings of the IEEE*, 90(2), February 2002.
- [4] C. E Shannon. Communication in the presence of noise. *Proceedings of the IEEE*, 86(2):447–457, February 1998.
- [5] R. H. Walden. Analog-to-digital converter survey and analysis. *IEEE Journal on Selected Areas in Communications*, 17(4):539–550, April 1999.
- [6] A. M. Bruckstein, D. L. Donoho, and M. Elad. From sparse solutions of systems of equations to sparse modeling of signals and images. *Society for Industrial and Applied Mathematics*, 51(1):34–81, February 2009.
- [7] D. L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4), April 2006.
- [8] E. J. Candes, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory*, 52(2):489–509, February 2006.
- [9] Y. C. Eldar and G. Kutyniok. *Compressed Sensing Theory and Applications*. Cambridge University Press, 2012.
- [10] L. Du, R. Wang, W. Wan, and X. Q. Yu. Analysis on greedy reconstruction algorithms based on compressed sensing. pages 783–789, Shangai, China, July 2012.
- [11] L. Jicheng, Z. Hao, and M. Huadong. Novel hardware architecture of sparse recovery based on FPGAs. volume 1, pages 302–306, Dalian, China, July 2010.

- [12] P. Blache, H. Rabah, and A. Amira. High level prototyping and FPGA implementation of the orthogonal matching pursuit algorithm. pages 1336–1340, Montreal, Canada, July 2012.
- [13] L. Bai, P. Maechler, M. Muehlberghuber, and H. Kaeslin. High-speed compressed sensing reconstruction on FPGA using OMP and AMP. pages 53–56, Seville, Spain, December 2012.
- [14] J. L. V. M. Stanislaus and T. Mohsenin. Low-complexity FPGA implementation of compressive sensing reconstruction. pages 671–675, San Diego, CA, January 2013.
- [15] J. L. V. M. Stanislaus and T. Mohsenin. High performance compressive sensing reconstruction hardware with QRD process. pages 29–32, Seoul, Korea, May 2012.
- [16] K. Moriya and T. Nodera. A new scheme of computing the approximate inverse preconditioner for reduced linear systems. *Journal of Computational and Applied Mathematics*, (199):345–352, 2007.
- [17] G. Schulz. Iterative berechnung der reziproken matrix. *Angew. Math. Mech.*, (13): 57–59.
- [18] H.-B. Li, T.-Z. Huang, Y. Zhang, X.-P. Liu, and T.-X. Gu. Chebyshev-type methods and preconditioning techniques. *Applied Mathematics and Computation*, 218(2): 260–270, September 2011.
- [19] S. Amat, S. Busquier, and J. M. Gutierrez. Geometric constructions of iterative functions to solve nonlinear equations. *Applied Mathematics and Computation*, 157(1): 197–205, August 2003.
- [20] M. Lusting, D. L. Donoho, and J. M. Pauly. Rapid MR imaging with compressed sensing and randomly under-sampled 3DFT trajectories. *Proceedings of ISMRM*, 2006.
- [21] S. S. Vasanawala, M. T. Alley, B. A. Hargreaves, R. A. Barth, J. M. Pauly, and M. Lusting. Improved pediatric MR imaging with compressed sensing. *Radiology*, 256(2):607–616, August 2010.
- [22] M. Mishali, Y. C. Eldar, O. Dounaevsky, and E. Shoshan. Xampling: analog to digital at sub-nyquist rates. *IET Circuits, Devices and Systems*, 5(1):8–20, January 2013.
- [23] Y. Oike and A. E. Gamal. CMOS image sensor with per-column sigma-delta ADC and programmable compressed sensing. *IEEE Journal of Solid-State Circuits*, 48(1), January 2013.

- [24] M. Balouchestani. Increasing the reliability of wireless sensor network with a new testing approach based on compressed sensing theory. Paris, France, May 2011.
- [25] M. Balouchestani. Low-power wireless sensor network with compressed sensing theory. Montreal, Canada, June 2011.
- [26] C. Hongzhi, X. Ning, S. Bo, C. Liangliang, Z. Jianguo, C. L. King Wai, and Y. Ruiguo. Infrared camera using a single nano-photodetector. *IEEE Sensors Journal*, 13(3), March 2013.
- [27] L. Jacques and P. Vandergheynst. *Compressed Sensing: “When Sparsity Meets Sampling”, in Optical and Digital Image Processing: Fundamentals and Applications*. Wiley, KGaA, Weinheim, Germany, 2011.
- [28] D. L. Donoho and M. Elad. Optimally sparse representation in general (nonorthogonal) dictionaries via l_1 minimization. *The National Academy of Sciences*, 100(5): 2197–2202, 2003.
- [29] E.J. Candes and T. Tao. Decoding by linear programming. *IEEE Transactions on Information Theory*, 51(12), December 2005.
- [30] R. A. DeVore. Deterministic constructions of compressed sensing matrices. *Journal of Complexity*, 23(4-6):918–925, August-December 2007.
- [31] R. Calderbank, S. Howard, and S. Jafarpour. Construction of a large class of deterministic sensing matrices that satisfies a statical isometry property. *IEEE Journal of Selected Topics in Signal Processing*, 4(2), April 2010.
- [32] S. Li, F. Gao, G. Ge, and S. Zhang. Deterministic construction of compressed sensing matrices via algebraic curves. *IEEE Transactions on Information Theory*, 58(8), August 2012.
- [33] J.A. Tropp and S. J. Wright. Computational methods for sparse solution of linear inverse problems. *Proceedings of the IEEE*, 98(6):948 – 958, June 2010.
- [34] Y. Zhang. Theory of compressed sensing via l_1 -minimization: a non-RIP analysis and extensions. *Journal of the Operations Research Society of China*, 1(1):79–105, March 2013.
- [35] S.G. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12), December 1993.
- [36] J.A. Tropp and A.C. Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on Information Theory*, 53(12):4655 – 4666, December 2007.

- [37] D. Needell and J. A. Tropp. CoSaMP: Iterative signal recovery from incomplete and inaccurate samples. *Information Theory and Applications*, January 2008.
- [38] T. Blumensath and M.E. Davies. Gradient pursuits. *IEEE Transactions on Signal Processing*, 56(6):2370 – 2382, June 2008.
- [39] F. Toutounian and F. Soleymani. An iterative method for computing the approximate inverse of a square matrix and the Moore–Penrose inverse of a non-square matrix. *Applied Mathematics and Computation*, 224:671–680, November 2013.
- [40] S. I. Grossman. *Algebra lineal*. Mcgraw Hill, 6 edition, 2008.
- [41] S. Mirzaei A. Irturk and R. Kastner. An efficient FPGA implementation of scalable matrix inversion core using QR decomposition. Technical Report CS2009-0938, University of California San Diego (UCSD), March 2009.
- [42] C.K. Singh, H. P Sushma, and P.T. Balsara. VLSI architecture for matrix inversion using modified gram-schmidt based QR decomposition. pages 836 – 841, January 2007.
- [43] B. A. Shenoi. *Introduction to Digital Signal Processing and Filter Design*. wiley-Interscience, 2006.
- [44] J. L Semlow. *Biosignal and Biomedical Image Processing Matlab-based applications*. Marcel Dekker, 2004.
- [45] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90 – 93, 1974.
- [46] A. Bovik, editor. *Handbook of image and video processing*. Academic Press, 2000.
- [47] H. D. Rico-Aniles, J. M. Ramirez-Cortes, and J. de J. Rangel-Magdaleno. FPGA-based inversion matrix using an iterative chebyshev-type method in the context of compressed sensing. Montevideo, Uruguay, May 2014.
- [48] A Irturk, B. Benson, S Mirzaei, and R Kastner. An FPGA design space exploration tool for matrix inversion architecture. Anaheim, CA, June 2008.