

Spring 6-17-2015

Implementation of Compressive Sensing Algorithms on Arm Cortex Processor and FPGAs

Dinesh Veeramachaneni

Follow this and additional works at: https://scholarworks.uttyler.edu/ee_grad



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Veeramachaneni, Dinesh, "Implementation of Compressive Sensing Algorithms on Arm Cortex Processor and FPGAs" (2015).
Electrical Engineering Theses. Paper 27.
<http://hdl.handle.net/10950/284>

This Thesis is brought to you for free and open access by the Electrical Engineering at Scholar Works at UT Tyler. It has been accepted for inclusion in Electrical Engineering Theses by an authorized administrator of Scholar Works at UT Tyler. For more information, please contact tbianchi@uttyler.edu.

IMPLEMENTATION OF COMPRESSIVE SENSING ALGORITHMS
ON ARM CORTEX PROCESSOR AND FPGAs

by

DINESH VEERAMACHANENI

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering
Department of Electrical Engineering

Hector A. Ochoa, Ph.D., Committee Chair

College of Engineering

The University of Texas at Tyler
May 2015

The University of Texas at Tyler
Tyler, Texas

This is to Certify that the Master's Thesis of

DINESH VEERAMACHANENI

has been approved for the thesis requirements on

APRIL 22, 2015

for the Master of Science in Electrical Engineering

Approvals



Thesis Chair: Hector A. Ochoa, Ph.D.



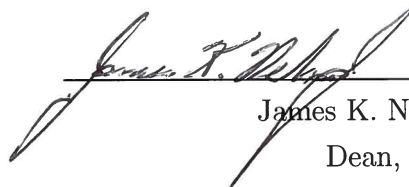
Member: Mukul V. Shirvaikar, Ph.D.



Member: Ron J. Pieper, Ph.D.



Chair and Graduate Coordinator: Hassan El-Kishky, Ph.D.



James K. Nelson, Jr., Ph.D., P.E.,
Dean, College of Engineering

Acknowledgements

It gives me immense pleasure to express my sincere and heartfelt thanks to all the people who have supported and encouraged me. Firstly, I would like to thank my parents and my sister for supporting me and encouraging me in every phase of my life to achieve my goals. A special thanks for all the sacrifices that you have made for me from my behalf.

I would like to express my profound gratitude to Dr. Hector A. Ochoa for his constant support and encouragement. I would also like to thank him for his patience and his timely suggestions throughout the research. I would like to thank Dr. David H.K. Hoe for his support and ideas throughout the research. I would like to thank Dr. Mukul V. Shirvaikar for guiding me and having his valuable suggestions. I would like to express my sincere thanks to Dr. Ron Pieper for advising me, taking time to be part of my committee and reviewing my work.

Finally, I would like to thank my well-wishers and friends for their constant support and encouragement. Last but not the least. I would like to thank everyone who have been directly or indirectly involved for making this research successful.

Table of Contents

List of Tables	iii
List of Figures	iv
Abstract	v
Chapter One: Introduction	1
1.1 Organization of Thesis	2
Chapter Two: Background	3
2.1 Introduction to Compressive Sensing	3
2.1.1 Signal Representation and Measurement Matrix	4
2.2 Ordinary Least Squares Solution of Underdetermined Systems via QR Decomposition	5
2.3 Optimization Techniques	7
2.3.1 Optimal Matching Pursuit (OMP)	7
2.3.2 Compressive Sampling Matching Pursuit (CSMP)	8
2.3.3 Stagewise Orthogonal Matching Pursuit (StOMP)	9
2.4 Advanced RISC Machine (ARM)	10
2.5 Field Programmable Gate Array (FPGA)	11
Chapter Three: Proposed Solution	14
3.1 Implementation in MATLAB	14
3.2 Design Flow for Zynq System on Chip (SoC)	14
3.3 Implementation on ARM processor	16

3.3.1	AXI Standard	16
3.3.2	Vivado Integrated Development Environment (IDE)	17
3.4	Implementation on ARM processor and FPGA	19
3.4.1	Operation of a Direct Memory Access (DMA)	21
3.4.2	Floating Point Unit Operation	21
3.5	Implementation of Matrix Multiplication Block on FPGA	21
Chapter Four:	Results	24
4.1	MATLAB Results	24
4.1.1	MATLAB Results for Optimal Matching Pursuit (OMP)	24
4.1.2	MATLAB Results for Compressive Sampling Matching Pursuit (CSMP)	24
4.1.3	MATLAB Results for Stagewise Orthogonal Matching Pursuit (StOMP)	25
4.2	Execution Times in MATLAB	26
4.3	Execution Times in ARM Processor	29
4.4	Execution Times for a 12×12 Matrix Multiplication Using ARM and Floating Point Units	32
4.5	Performance of Matrix Multiplication IP core	33
4.6	Resource Utilization Reports	33
Chapter Five:	Conclusion and Future Work	39
5.1	Conclusion	39
5.2	Future Work	40
References	41

List of Tables

Table 4.1	Execution Times in MATLAB for a 18×512 Measurement Matrix and Three Targets.	27
Table 4.2	Execution Times in MATLAB for a 24×512 Measurement Matrix and Three Targets.	27
Table 4.3	Comparison of Execution Times of the algorithms for a 700×1000 Radar Measurement Matrix and Three Targets.	28
Table 4.4	Number of execution clock cycles of ARM Processor for an 18×512 measurement matrix and three targets.	29
Table 4.5	Number of Execution Clock Cycles of ARM Processor for a 24×512 Measurement Matrix with Three Targets.	30
Table 4.6	Comparison of the Execution Times in MATLAB and ARM for a 24×512 Measurement Matrix with Three Targets.	30
Table 4.7	Number of Execution Clock Cycles of ARM Processor for a 700×1000 Radar Measurement Matrix with Three Targets.	31
Table 4.8	Comparison of the Execution Times in MATLAB and ARM considering a 700 by 1000 Measurement Matrix with Three Targets.	31
Table 4.9	Performance Results of a 12×12 Matrix Multiplication using an architecture with an ARM Processor, and ARM with Floating Point Units.	32
Table 4.10	Comparison of Execution Clock Cycles for a Matrix Multiplication IP.	33

List of Figures

Figure 2.1	Compressive Sensing Measurement Matrix [5]	4
Figure 2.2	Zed Board Development Kit XC7Z020CLG484-1 with Zynq Processing System [22]	13
Figure 3.1	Design Flow for a Zynq SoC [26].	15
Figure 3.2	Architectural Block Diagram of a Zynq Processing System [26] . .	17
Figure 3.3	Architectural Design of a Zynq Processing System connected to a Timer.	18
Figure 3.4	Architectural Design of a Zynq Processing System with Floating Point IP blocks connected to a Timer.	20
Figure 3.5	Architectural Design of a Zynq Processing System with Matrix Multiplication IP core connected to a Timer.	22
Figure 4.1	Reconstructed Sparse Signal using OMP.	25
Figure 4.2	Reconstructed Sparse Signal using CSMP.	26
Figure 4.3	Reconstructed Sparse Signal using StOMP.	27
Figure 4.4	Reconstructed Sparse Signal using Optimization Techniques. . . .	28
Figure 4.5	Synthesis Resource Utilization Report of ARM Processor Architecture.	34
Figure 4.6	Implementation Resource Utilization Report of ARM Processor Architecture.	34
Figure 4.7	Synthesis Resource Utilization Report of ARM and Floating Point Units Architecture.	35
Figure 4.8	Implementation Resource Utilization Report of ARM and Floating Point Units Architecture.	35
Figure 4.9	Synthesis Resource Utilization Report of the Matrix Multiplication IP core for Matrix Inputs of 18×512 and 512×1	36
Figure 4.10	Synthesis Resource Utilization Report of the Matrix Multiplication IP core for Matrix Inputs of 24×512 and 512×1	37
Figure 4.11	Synthesis Resource Utilization Report of the Matrix Multiplication IP core for Matrix Inputs of 700×1000 and 512×1	38

Abstract

IMPLEMENTATION OF COMPRESSIVE SENSING ALGORITHMS ON ARM CORTEX PROCESSOR AND FPGAs

DINESH VEERAMACHANENI

Thesis Chair: Hector A. Ochoa, Ph.D.

The University of Texas at Tyler

May 2015

Nowadays, communication systems require huge amounts of data to be processed. Some examples of these systems include radar systems, video streaming, and many other multimedia applications. These systems require large amounts of bandwidth to satisfy the Nyquist rate. Compressive Sensing is proposed as a way to reduce their bandwidth requirements.

Compressive Sensing algorithms are generally implemented at the receiver to reconstruct the original signal from a reduced set of samples. This methodology eliminates data which is relatively insignificant. It possesses the potential to eliminate the use of large bandwidth, cost effective matched filters, and high-frequency analog-to-digital converters at the receiver in the case of radar systems. Compressive Sensing is widely used in areas such as Digital Image Processing, Digital Signal Processing, Radars, and Wireless Sensor Networks.

This research investigates on three main optimization techniques commonly used in Compressive Sensing: Optimal Matching Pursuit (OMP), Compressive Sampling Matching Pursuit (CSMP) and Stagewise Orthogonal Matching Pursuit (StOMP). These algorithms were implemented and tested on an ARM processor, and on a Field

Programmable Gate Array (FPGA).

During the first stage of this research, the optimization techniques were implemented in MATLAB. In the second stage, they were implemented on an ARM processor to accelerate their performance. The algorithms show a considerable acceleration on the ARM processor compared to MATLAB. In the final stage of the research, linear algebra operations were implemented on an FPGA to further accelerate their performance. The results show further improvement when part of the code was implemented on an FPGA.

Chapter One

Introduction

Data processing applications like video streaming, multimedia, and radars have been gaining popularity with advancing technology. These applications require data to be sent through a communication channel. They also require high accuracy; therefore, large amounts of samples or data points need to be transmitted. As the number of samples required for transmission increases, the operating frequencies of these systems also need to be increased. It is well known that in order for these signals to be properly reconstructed, they need to be sampled at the Nyquist rate. Therefore, the operating frequencies of systems such as matched filters, and Analog-to-Digital converters need to be increased, placing a high load on the components at the receiver.

The above mentioned problems can be overcome by implementing a technique called Compressive Sensing (also called Compressive Sampling). This technique aims at reconstructing the signal even when it has been sampled at a sub-Nyquist rate. Compressive Sensing algorithms take advantage of non-adaptive linear projections and greedy approaches that iterate until the required solution is obtained.

One of the requirements of Compressive Sampling is that the signal needs to be sparse, meaning that the information rate of the signal is much smaller than suggested by its bandwidth [1]. In Compressive Sensing, only the most important information is acquired, and all the remaining values are discarded or are replaced by a predefined default value. At the end, the transmitter does not transmit the discarded values, saving bandwidth at the receiver and the transmitter.

Advantages of Compressive Sensing include (but are not limited to) reduction in bandwidth by sending fewer samples, and elimination of expensive components in systems like radars. Since the algorithms convert the signal through an iterative

process via software, the process is much faster than the traditional solution which includes physical hardware chips or circuits like the matched filters, and Analog to Digital Convertor's (ADC's).

- *Applications of Compressive Sensing in Digital Image Processing:* A digital image is compressed and under sampled at the transmitter and the reconstruction algorithms are applied at the receiver to recover the digital image. Applications include face recognition and image enhancement [2].
- *Applications of Compressive Sensing in Digital Signal Processing:* A digital compressed and under sampled signal is sent at the transmitter, and the reconstruction algorithms are applied at the receiver to recover the digital signal. Applications include signal reconstruction from fewer samples.
- *Applications of Compressive Sensing in Radar:* For radar systems, parameters like altitude and angular location of the target need to be tracked. This makes radar signals more flexible for Compressive Sensing [3]. Compressive Sensing algorithms are applied on a echo signal captured by the radar to recover the details of the target.
- *Applications of Compressive Sensing in Sensor Networks:* Compressive Sensing is also used in sensor networks, which requires data from the sensor to be recorded over long periods of time. In this case, Compressive Sensing allows the user to take a few readings and recover the data, saving power for readings which are not being taken.

1.1 Organization of Thesis

This thesis is divided into five chapters. Chapter 2 discusses Compressive Sensing, and the ordinary least squares solution which will be used in the optimization techniques. Chapter 3 describes the design methodology and proposed solution. Chapter 4 presents results showing the execution times and resource utilization for each architecture. Chapter 5 includes the conclusions and future work.

Chapter Two

Background

2.1 Introduction to Compressive Sensing

Compressive Sensing, also known as Compressive Sampling (CS) is a technique developed by Emmanuel Candes, Terence Tao and David Donoho around the year 2004 [4]. Using this technique it is possible to recover a signal which is sampled below the Nyquist rate. Traditional sampling theory states that the sampling rate or the number of samples in a signal should be at least twice the maximum frequency in an analog signal.

The general procedure for non-Compressive Sensing signals is to acquire the signal and transmit these data points or projections onto a communication channel. Because the receivers on the end can receive only signals sampled according to Nyquist rate, the signal needs to be sampled at the Nyquist rate. As a result, every single data point needs to be transmitted.

Compressive Sensing is considered an effective solution as long as the signal of interest is sparse, or it can be made sparse by means of a transformation mapping. Compressive Sensing utilizes the techniques of ordinary least squares, and QR decomposition which are used to find the solution for underdetermined systems of equations. One of the conditions for Compressive Sampling to work is that the signals need to be sparse. Sparsity refers to the idea that the amount of information in the signal is much smaller than that suggested by its bandwidth [1].

2.1.1 Signal Representation and Measurement Matrix

Signal representation is a key factor in understanding the functionality of any system. In Compressive Sensing, the observed signal is defined as the product of a measurement matrix and the original signal. This idea is represented in equation 2.1.

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

where Φ is the uniform random measurement matrix, x is the solution vector, and y is the received vector.

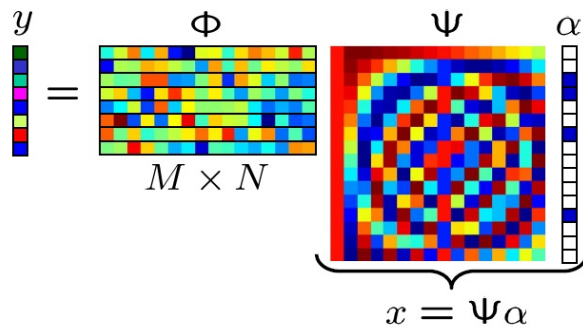


Figure 2.1: Compressive Sensing Measurement Matrix [5]

Figure 2.1 shows an example of a measurement matrix, where $M < N$, x is the solution vector and Ψ is the transformation mapping to ensure that x is sparse. Observe that the number of measurements or samples (M) is less than the length of the under sampled signal (N).

For example, in case of a radar system, the measurement matrix would be constructed out of the transmitted signal, and y would be the observed signal reflected from the target. Using this information it is possible to recover the vector x using Compressive Sensing techniques [3].

Designing a measurement matrix is a very important factor in Compressive Sensing. There are two types of measurement matrices that can be used: the random measurement matrix and the predefined measurement matrix. The measurement matrix Φ shown in Figure 2.1 must allow the reconstruction of the length- N signal x from $M < N$ measurements. As the number of measurements made is less than

the length of the signal, reconstruction by traditional methods is impossible. As mentioned before if x is K -sparse the problem can be solved provided that $M \geq K$ [6]. However, the following condition should be satisfied for any vector v sharing the same K nonzero entries as the vector s .

$$1 - \epsilon \leq \frac{\|\Theta v\|_2}{\|v\|_2} \leq 1 + \epsilon \quad (2.2)$$

where ϵ is greater than zero, $\Theta = \Phi\Psi$ is an $M \times N$ matrix and $\|\cdot\|_2$ represents the second norm of the vector. A sufficient condition for a stable solution for both K -sparse and compressible signals is that Θ should satisfy equation 2.2 for an arbitrary $3K$ -sparse vector v . This condition is referred to as the Restricted Isometric Property (RIP) [7]. If the rows ϕ_j from the measurement matrix Φ cannot sparsely represent the columns of ψ_i of Ψ and vice versa, it is referred to as *incoherent*. If Φ is sufficiently “incoherent” and satisfies the “Restricted Isometric Property (RIP)”, then the information of s will be implanted in y such that it can be perfectly recovered with high probability. Both the RIP and incoherence can be achieved with high probability simply by selecting the measurement matrix as a random matrix. As mentioned above, if a signal x composed of N samples is sparse, then the actual signal can be perfectly reconstructed using $M \geq cK \log(N/K) \ll N$ linear projections of x onto another basis.

2.2 Ordinary Least Squares Solution of Underdetermined Systems via QR Decomposition

Ordinary least squares solution is an effective method used to calculate the solution of underdetermined system of equations. Let A and B be two different matrices. Even more, matrix A is a non-square matrix, in which the number of rows is less than the number of columns. For that reason, it is an undetermined system of equations, and needs to be solved using the ordinary least squares. The ordinary least squares procedure is defined by equation 2.3 [8]-[9].

$$X_k = ((A^T * A)^{-1} * A^T) * B \quad (2.3)$$

The first part of the equation is calculating the product of the matrix A and the transpose of the matrix A . The inverse of the product of both matrices is calculated using the procedure as described below [10]-[13].

$$(A^T * A) = Q * R \quad (2.4)$$

equation 2.4 shows the decomposition of the product matrix into Q and R where Q is an orthogonal matrix and R is an upper triangular matrix. An orthogonal matrix is a matrix in which the transpose of Q and the inverse of Q are the same.

The Q and R decomposition methodology using modified gram schmidt is as follows [10].

1. The product matrix can be decomposed into a set of columns or vectors as shown in equation 2.5

$$C = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_n \end{bmatrix} \quad (2.5)$$

2. The a_1 vector is assigned to a new vector called u_1 , and the e_1 vector is calculated by dividing the u_1 vector by the norm of u_1 . This procedure is summarized in equations 2.6 and 2.7

$$u_1 = a_1 \quad (2.6)$$

$$e_1 = \frac{u_1}{\|u_1\|_2} \quad (2.7)$$

3. In the next step, the projection of the vector a_2 onto the vector e_1 is subtracted from the vector a_2 . The vector u_2 is calculated by dividing the vector u_2 by its norm. The following equations 2.8 and 2.9 summarize this procedure.

$$u_2 = a_2 - (a_2 \cdot e_1)e_1 \quad (2.8)$$

$$e_2 = \frac{u_2}{\|u_2\|_2} \quad (2.9)$$

4. This process is repeated until all the corresponding vectors of e are calculated. The resultant QR factorization is given by equation ??

$$\begin{aligned} C &= \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_n \end{bmatrix} \\ &= \begin{bmatrix} e_1 & e_2 & e_3 & \cdots & e_n \end{bmatrix} \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix} = QR \end{aligned} \quad (2.10)$$

5. The next step is to calculate the inverse of QR as shown in equation 2.11

$$(QR)^{-1} = R^{-1} * Q^{-1} = R^{-1} * Q^T \quad (2.11)$$

The computed value from equation 2.11 is substituted back in equation 2.3 to get the ordinary least squares solution.

2.3 Optimization Techniques

Optimization techniques are a quick and easy way to find solutions of sparse signals, because most of the optimization techniques take advantage of non-adaptive linear projections to preserve the structure of the signal. The following sections describe and explain the functionality of three of the most common optimization techniques. These are the Optimal Matching Pursuit (OMP) [14]-[15], Compressive Sampling Matching Pursuit (CSMP) [14][16], Stagewise Orthogonal Matching Pursuit (StOMP) [17].

2.3.1 Optimal Matching Pursuit (OMP)

1. The measurement matrix Φ and received vector or observed vector u are the inputs to the system. A residual vector is defined in order to check the precision of the solution.
2. Initially at the start of the algorithm the error is equal to the observed vector i.e. u defined by the following equation 2.12

$$r = u \tag{2.12}$$

3. The correlation between the columns of measurement matrix and the residual vector is calculated using equation 2.13

$$|\langle r, \Phi \rangle| \tag{2.13}$$

4. The column which is most correlated is identified and a *support* matrix is updated with the index position of the column number which is most correlated with the residual vector.
5. A *phiactive* matrix is updated with the most correlated column on every iteration.
6. An ordinary least squares solution is calculated between the *phiactive* matrix and the observed vector u to give a sparse vector result x_k as shown in equation 2.14

$$x_k = LS(\Phi_{active}, u) \tag{2.14}$$

7. The residual is re-calculated and is updated for the next iteration using equation 2.15

$$r = u - \Phi x_k \tag{2.15}$$

8. The process is repeated from step 3 until the stopping condition is met. The stopping condition can be one of following: 1. The number of iterations 2. The error

value defined by the user.

9. After the process ends, a sparse vector x_{sparse} is created with the length of the columns of the measurement matrix and is initialized with all zeros, the values of x_k are placed in the x_{sparse} vector at places defined by the *support* matrix.

2.3.2 Compressive Sampling Matching Pursuit (CSMP)

The Optimal Matching Pursuit (OMP) described previously extracts one most correlated column on each iteration. On the other hand, the Compressive Sampling Matching Pursuit is capable of extracting more than one column on each iteration defined by a tuning parameter. The functioning of the Compressive Sampling Matching Pursuit is as follows.

1. The measurement matrix Φ and received vector or observed vector u are the inputs to the system. A residual vector is defined in order to check the degree of the closeness or error.
2. Initially since this is the start of the algorithm the error is equal to the observed vector i.e. u defined by the following equation 2.16

$$r = u \tag{2.16}$$

3. Define a target sparsity s and tuning parameter α . An *n sel* parameter is calculated using equation 2.17 which describes the number of columns to be extracted on every iteration.

$$n\text{sel} = \alpha * s \tag{2.17}$$

4. The *n sel* most correlated columns from the measurement matrix are identified using equation 2.18

$$|\langle r, \Phi \rangle| \tag{2.18}$$

5. Define 'omega' vector to have the indexes of the most correlated *n sel* columns.
6. For the first iteration define T matrix to be the *omega* matrix. For subsequent iterations this matrix will get updated with the union of the *omega* and *support* matrix from the previous iterations.
7. Define a *phiactive* matrix which contains all the most correlated columns specified by T .
8. Perform an ordinary least squares solution of the *phiactive* matrix and observed

vector u to give a result x_t as shown equation 2.19

$$x_t = LS(\Phi_{active}, u) \quad (2.19)$$

9. Sort the x_t vector in a descending order and define a x_{loc} variable which consist the indexes of the sorted columns.

10. Define a sparse vector x_k with the length of the columns of the measurement matrix and is initialized with all zeros.

11. Insert the first s largest values given by $x_t(x_{loc}(i))$ at indexes defined by $x_k(T(x_{loc}(i)))$ as shown in equation 2.20

$$x_k(T(x_{loc}(i))) = x_t(x_{loc}(i)) \quad (2.20)$$

12. Update the residual by subtracting the product of the sparse vector and the measurement vector with the observed vector as shown in equation 2.21

$$r = u - \Phi x_k \quad (2.21)$$

13. Create a *support* matrix if it is the first iteration, update support for subsequent iterations by the following equation 2.22

$$support = T(x_{loc}(1 : s)) \quad (2.22)$$

The process is repeated from step 4 until the stopping condition is met. The stopping condition can be one of following: 1. The number of iterations, 2. The error value required by the user.

2.3.3 Stagewise Orthogonal Matching Pursuit (StOMP)

The Stagewise Orthogonal Matching Pursuit (StOMP) uses a slightly different approach. A threshold parameter is defined and all the columns with a correlation coefficient greater than the threshold are extracted. The functionality of the StOMP is as follows.

1. The measurement matrix Φ and received vector or observed vector u are the inputs to the system. A residual vector is defined in order to check the degree of the closeness or error.

2. Initially since this is the start of the algorithm the error is equal to the observed vector i.e u defined by the following equation 2.23

$$r = u \quad (2.23)$$

3. The threshold parameter σ_{T_s} is calculated as demonstrated in the equation 2.24

$$\sigma_{T_s} = \frac{\|r\|_2}{\sqrt{n}} T_s \quad (2.24)$$

4. The correlation between the columns of measurement matrix and the residual vector is calculated as demonstrated in equation 2.25

$$|\langle r, \Phi \rangle| \quad (2.25)$$

5. All the columns which have a correlation coefficient greater than the set threshold σ_{T_s} are selected and all the indexes are stored into another vector named *omega*.

6. For the first iteration define a *support* vector which is initialized with *omega*, for subsequent iterations update the *support* with the union of *omega* and *support*.

7. Define a *phiactive* matrix and update the *phiactive* matrix with the most correlated columns obtained in step 6.

8. Calculate the ordinary least squares solution of *phiactive* and *u* as demonstrated in equation 2.26

$$x_k = LS(\Phi_{active}, u) \quad (2.26)$$

9. A sparse vector x_{sparse} is created with the length of the columns of the measurement matrix and is initialized with all zeros, the values of x_k are placed in the x_{sparse} vector at locations or indexes defined by the *support* matrix as shown in equation 2.27.

$$x_{sparse}(supp(e)) = x_k(e) \quad (2.27)$$

where e is a variable that runs through the length of x_k .

10. Update the residual by subtracting the product of the sparse vector and the measurement vector with the observed vector as shown in equation 2.28

$$r = u - \Phi x_k \quad (2.28)$$

11. The process is repeated from step 5 until the stopping condition is met. The stopping condition can be one of following like the number of iterations or the error value required by the user.

2.4 Advanced RISC Machine (ARM)

The Advanced RISC Machine (ARM) is a family of processors launched in November 1990 by Advanced RISC Machine (ARM). The ARM processor has a wide range of

capabilities starting from simple arithmetic calculations to controlling several other peripherals. ARM is available in both 32-bit and 64-bit architectures. The ARM processor is widely used in daily applications ranging from hard disk drives, printers, washing machines, smart meters, gaming and television [18]. One of the most popular application of ARM is its use in cellphones. Because of its wide range of capabilities starting from signal processing to multimedia applications, it is best suited for today's smartphones which are capable of not only receiving signals, but also capable of processing multimedia applications like video streaming and gaming.

One of the most powerful processors from the ARM family is the "ARM cortex A9" processor [19]. The ARM cortex A9 uses an architecture v7-A with an eight stage pipeline. The architecture v7-A of the ARM has features like memory management unit, multitasking, and trust zone (a feature for secure applications like payments), and is capable of high performance at low power. The ARM cortex A9 has one of the fastest (AMBA-AXI) interconnects which makes the movement of data faster between modules [20].

One of the most important features of the cortex-A9 processor is the NEON Media Processor Engine (MPE) technology. The NEON MPE is a separate hardware unit on Cortex-A series processors, together with a vector floating point (VFP) unit [20]. The overall idea of having a dedicated NEON engine is that if an algorithm can be designed to exploit dedicated hardware, performance can be maximized. This means fewer clock cycles are required to process data and therefore more stand-by time which will decrease the power consumption and increase the performance. The NEON engine uses a Single Instruction Multiple Data (SIMD) technique in order to process many data values. The NEON engine is particularly useful for digital signal processing or multimedia algorithms such as Block-based data processing, FFTs, matrix multiplication, and audio video codecs like MPEG-4.

2.5 Field Programmable Gate Array (FPGA)

Field Programmable Gate Array (FPGA) is an integrated circuit which can be configured or programmed on the fly depending on the specific application. A Field Programmable Gate Array consists of basic logic circuits like encoders, decoders, multiplexer's and several Look Up Tables(LUT's) integrated into a block called as a

Configurable Logic Block (CLB). The CLB performs the operations as specified by the designer. First, a program needs to be written for the design in VHDL, Verilog, or System C to obtain proper functionality on the FPGA. The software program describes connections and interface signals for each module, and the functionality of the design.

In recent years, FPGA's have advanced from basic adder, multiplier, DSP blocks to having a variable precision floating point DSP block on the chip. Nowadays, FPGA's are being manufactured with inbuilt hardware in which can one add systems intelligence through software, data processing and decisions can be executed in real time with programmable hardware and system interfaces through programmable Input Output (I/O) that make the FPGA a complete System-on-a-Chip solution [21]. One of the most popular and advanced FPGA's is the Xilinx Zynq 7000 Series.

Figure 2.2 shows the Zynq Evaluation and Development (ZED) Board evaluation kit with an embedded Zynq 7020 FPGA [22]-[23].

The programmable logic of this device is built on a highly advanced 28nm technology and is packed with 85,000 logic cells, 53,200 LUT's, 106,400 Flip-Flops, 560KB of extensible block RAM, 220 Programmable DSP slices, MSPS ADC's with up to 17 differential inputs [24]-[25].

The ZedBoard development kit is also embedded with a dual ARM Cortex-A9 processor (Processing System) with an L1 Cache of 32KB and L2 cache of 512KB and on chip memory of 256KB [22]. The maximum processor clock frequency is 866 MHz. The processor is supported with an external DDR3 memory support, external memory static memory with 2x Quad-SPI, NAND, NOR and 8 DMA channels. There are also some other peripherals like two UART, two CAN, two I2C, two SPI, 4*32 GPIO attached to the Processing System [24]-[25].

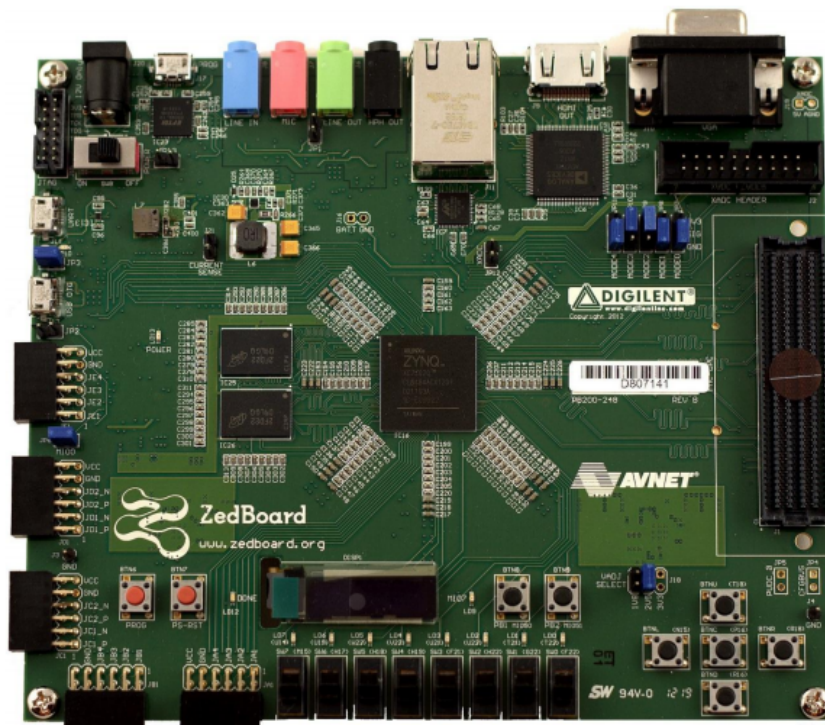


Figure 2.2: Zed Board Development Kit XC7Z020CLG484-1 with Zynq Processing System [22]

Chapter Three

Proposed Solution

The first step in application development is understanding the basic functionality of the algorithm or the process. The next step is to simulate its functionality and the last step is to implement it according to the application.

In order to get a better understanding of the functionality of the optimization techniques, it was decided to program them in MATLAB. MATLAB is a high level language and simulation tool used extensively for finding solutions to the mathematical problems by programming them and then visualizing them. The Optimal Matching Pursuit (OMP), Compressive Sampling Matching Pursuit (CSMP) and Stagewise Orthogonal Matching Pursuit (StOMP) have been programmed and implemented.

3.1 Implementation in MATLAB

The MATLAB implementation of the optimization techniques is used to envision, and verify the functionality of the optimization techniques. The inputs to the algorithms are a measurement matrix Φ and an observation matrix u . The algorithms are required to find out the best fit solution i.e. x . Multiple submodules like matrix multiplication, ordinary least squares, and QR decomposition are implemented in MATLAB. Their functionality has been checked with a predefined solution defined at the beginning of the algorithms.

3.2 Design Flow for Zynq System on Chip (SoC)

It is essential for any application development to have proper design flow in order to have a fully functional and error free application. Figure 3.1 demonstrates the steps

starting from the design specification stage to the final testing stage [26].

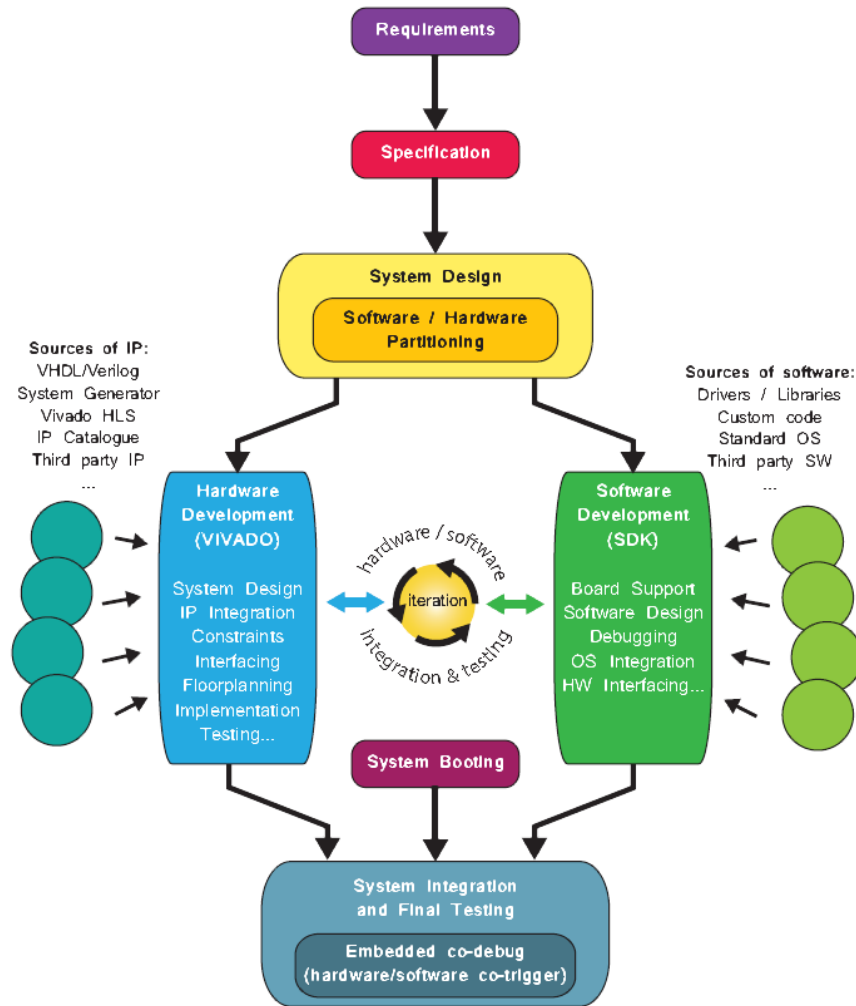


Figure 3.1: Design Flow for a Zynq SoC [26].

Requirements and specifications: A starting point of any project is defining the requirements and desired outcomes. System specification contains details such as the intended functionality, interfaces required and performance criteria.

System design: Most of the design architectures are top-down approaches. The top-level functionality and flow are designed first, and the internal submodules and functions are designed later. Depending on the system complexity, the design may be

partitioned into hardware, and software. The processor is generally responsible for task scheduling, and general-purpose tasks. Other computationally intensive and parallel tasks can be partitioned and implemented on hardware, i.e. Programmable Logic (PL). Vivado Integrated Development Environment (IDE) is used to start building the design and make the necessary hardware connections [27]-[32]. After the design is built and necessary connections are made, it is exported to Software Development Kit (SDK) for software development, which involves exporting the addresses of the individual components of the design and their respective connections [33]-[35].

System integration and testing: The final step is to integrate the hardware and software, initialize the design and check the functionality of the design.

3.3 Implementation on ARM processor

The first architectural design consists of the Zynq processing system which has an in built dual core ARM cortex A9 processor. The Zynq processing system not only encompasses the ARM processor, but is also embedded with several other peripheral memory interfaces, general purpose ports, high performance ports and Accelerator Coherency Port (ACP) [24]-[25]. Figure 3.2 shows the architectural block diagram of a Zynq processing system. The Application Processing Unit (APU) highlighted in the block diagram consists of 2 ARM cores, associated NEON Media Processing engine (MPE), Vector Floating Point Unit (VFU), level 1 cache, level 2 cache, scoop control unit and an associated On Chip Memory (OCM). The Processing System (PS) and the Programmable Logic (PL) are connected by high performance AXI standard, which is based on the ARM AMBA 3.0 open standard [25].

3.3.1 AXI Standard

AXI stands for Advanced eXtensible Interface (AXI) which is used to connect the processor, peripherals, and several other IP blocks on the Programmable Logic (PL) in an embedded system. The current version of AXI is AXI4. There are three different AXI4 bus protocols namely AXI4, AXI4-Lite and AXI4-Stream [26].

AXI4: This is used for memory mapped links providing the highest performance. In this case an address is supplied, and a data burst size up to 256 data words or

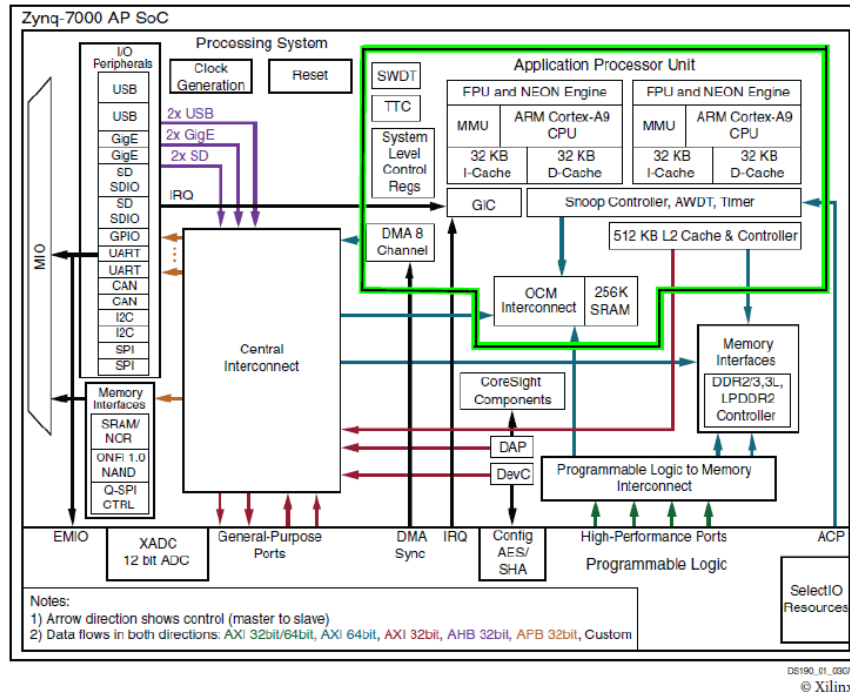


Figure 3.2: Architectural Block Diagram of a Zynq Processing System [26]

data beats can be achieved.

AXI4-Lite: This is a simplified version of the AXI4 Lite which is also memory mapped, accepts an address and allows only one data transfer or data word per connection (no bursts).

AXI4-Stream: This is used for high speed streaming data. This mechanism does not have an address mechanism and just streams in and streams out the data between source and destination (non-memory mapped).

3.3.2 Vivado Integrated Development Environment (IDE)

The Vivado Integrated Development Environment (IDE) is an environment build for creating and making connections to the hardware part of the SoC design [27]-[29].

The development environment is capable of making connections between processor, memories, peripherals, external interfaces using the above mentioned AXI standard. The choice of the AXI bus depends on the particular properties of that connection.

The first architectural design consists of the Zynq processing system. The algorithms were processed initially on the ARM processor. A timer was also included in the design to calculate the amount of clock cycles required to process the algorithm.

The timer is implemented on a programmable logic (PL) through an AXI interconnect via AXI4 memory mapped interface. The AXI interconnect essentially acts as a switch which manages and directs traffic between AXI interfaces. Figure 3.3 shows the architecture of a Zynq processing system with the attached timer.

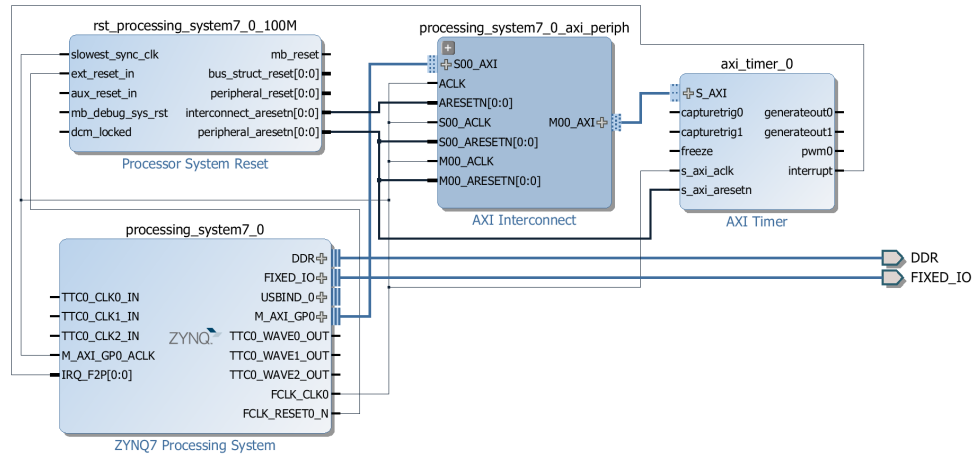


Figure 3.3: Architectural Design of a Zynq Processing System connected to a Timer.

This architecture was designed in Vivado Integrated Development Environment (IDE) [30]-[32]. After the necessary connections had been made in Vivado, the hardware connections are converted to a stream of zero's and one's understandable by the processor. This is called as a bitstream. The bitstream, and the hardware address map (*xparameters.h*) containing the address of every component is exported to Software Development Kit (SDK). The algorithms were written in C code, and implemented on the ARM cortex A9 processor. The code is written in Xilinx -Software

Development Kit (SDK), and the output is printed on the serial terminal.

Xilinx Software Development Kit (SDK) is an editor to write C code and execute it on the ARM processor. It is a complete application development, and debugging environment built on eclipse [33]-[35]. It includes GNU based compiler tool chain namely GCC compiler, GDB debugger, utilities and exclusive library support for ARM and NEON extensions using C, C++ languages. It also includes a JTAG debugger, flash programmer, drivers for Xilinx IPs and bare metal application [26].

3.4 Implementation on ARM processor and FPGA

The second design consists of both, the ARM processor, and FPGA in order to leverage the capability of the FPGA, and increase the performance of the design. Therefore it was designed to implement the computationally intensive matrix multiplication algorithms on the FPGA using the Xilinx floating point IP blocks. The floating point blocks use AXI4-Stream input data and AXI4-Stream output data interfaces.

In order to transfer the data in and out of the processing system a Direct Memory Access IP block was used in the design. It is also used to convert the memory mapped data coming out of a memory mapped IP block like a processing system into a stream data, and vice versa.

Figure 3.4 shows the architecture of a Zynq processing system with associated floating point blocks. It can be seen that two floating point blocks are used, one for multiplication and one for addition.

Two Direct memory access IP blocks are used in the circuit in order to transfer the data to the first floating point unit and one Direct memory access IP block to transfer the data to the second floating point unit and get back the result. This design is connected to the high performance (HP) AXI port of the processing system to have a high performance for transferring the data in and out.

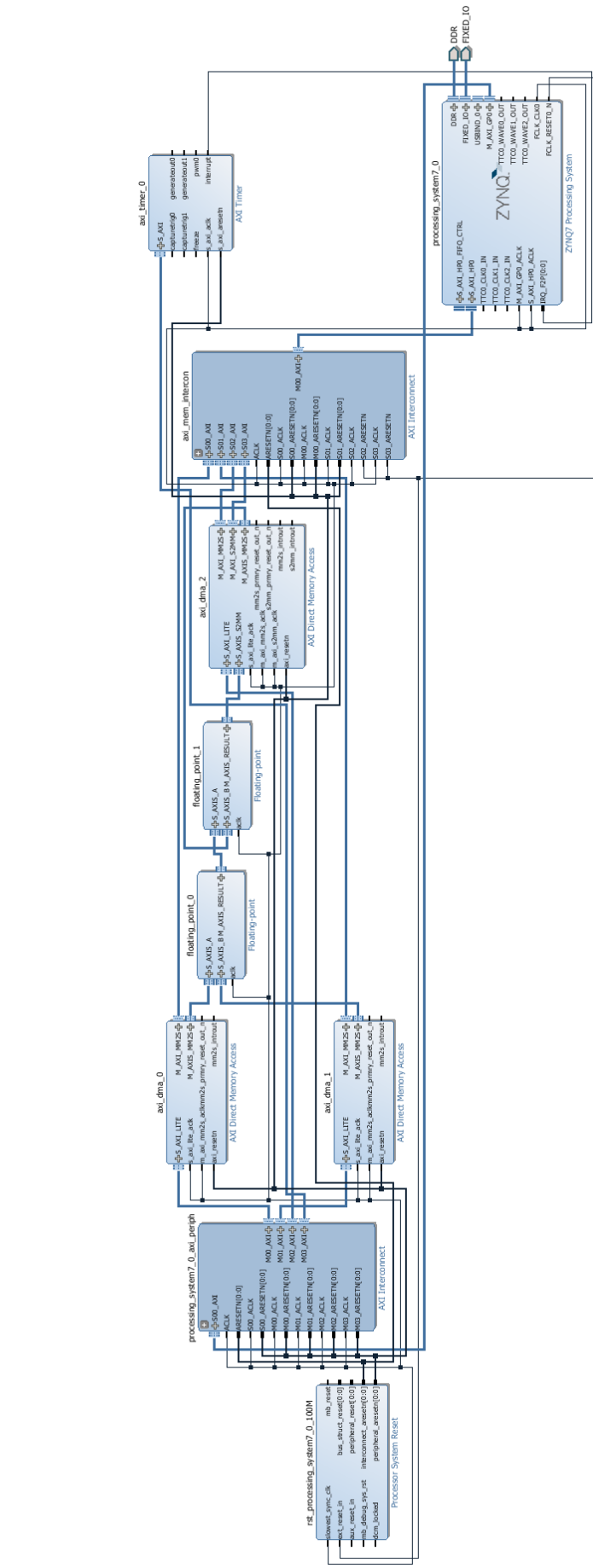


Figure 3.4: Architectural Design of a Zynq Processing System with Floating Point IP blocks connected to a Timer.

3.4.1 Operation of a Direct Memory Access (DMA)

Converting a memory mapped data into a stream data: The memory mapped data that needs to be converted into stream data is sent as an input to the M_AXI_MM2S, and the converted stream data is sent out through M_AXIS_MM2S [36].

Converting a stream data into a memory mapped data: The stream data that needs to be converted into memory mapped data is sent as an input to the S_AXIS_S2MM, and the converted memory mapped data is sent out through M_AXI_S2MM [36].

3.4.2 Floating Point Unit Operation

The *floating_point_0unit* receives the stream data from the processing unit. It is configured in the multiply mode which multiplies the two inputs and sends it on to another floating point unit named *floating_point_1*. The second floating point unit is configured in the addition mode. The *floating_point_1* gets the second input from the processing system which is the previously accumulated data. This accumulated sum is added at every iteration to each sub-product coming from the *floating_point_0*. After all iterations are complete the result is obtained from the output of the second floating point and is transferred to the processing system (PS). More details on the floating point IP can be found in the Xilinx Website [37].

3.5 Implementation of Matrix Multiplication Block on FPGA

In this architecture, it was designed to implement a matrix multiplication block on an FPGA with input stream and output stream in order to increase the performance [38]. Vivado High Level Synthesis (HLS) is a design tool used to convert a code written in high level languages like C, C++ to a hardware description language like VHDL, Verilog and system C. Vivado HLS is inbuilt with a lot of optimization directives. Few of them are pipeline, array partition, and array reshape [39]. Vivado HLS is also inbuilt with a test bench for verification of our design, synthesis and RTL generation tool to implement the design. After generating the required matrix multiplication block, the necessary connections are done in Vivado IDE as shown in the Figure 3.5. The matrix multiplication block is connected to the Accelerator Coherency Port (ACP) of the processing system to have an accelerated performance over the previous design.

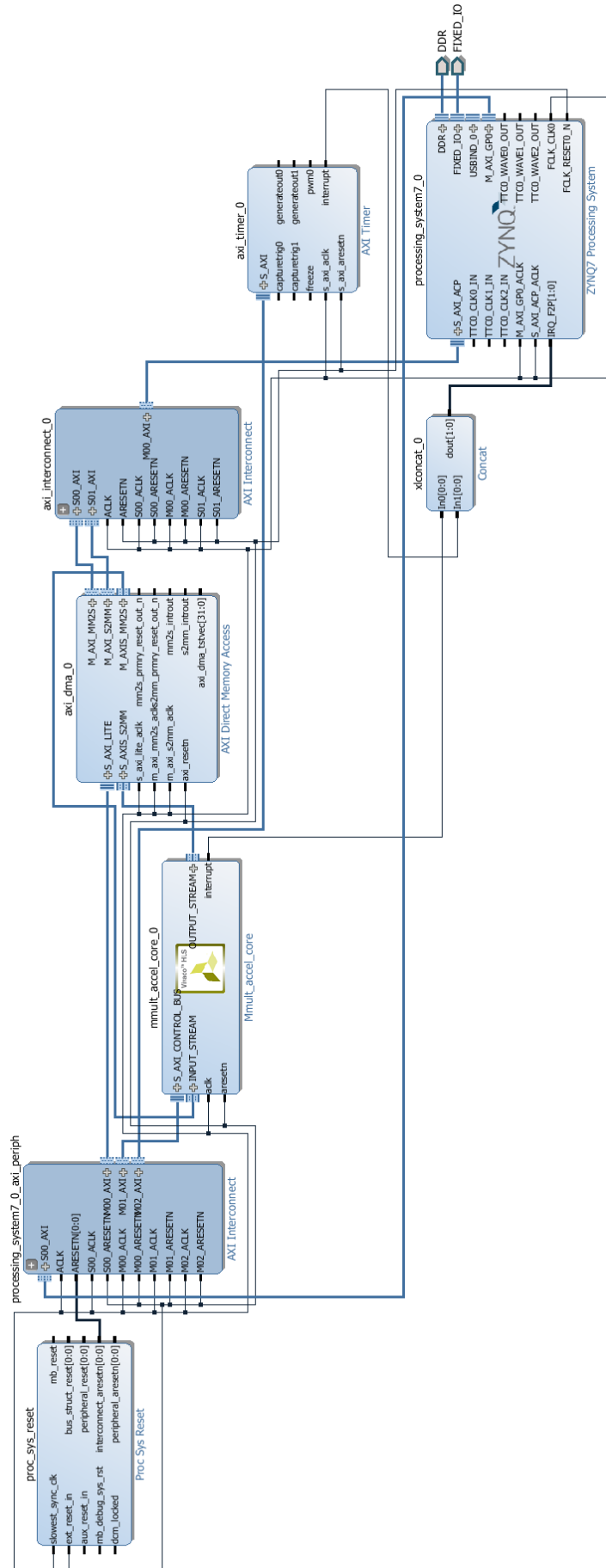


Figure 3.5: Architectural Design of a Zynq Processing System with Matrix Multiplication IP core connected to a Timer.

This design eliminates the use of two DMA's compared to the previous architecture. This design uses one DMA to stream in both inputs one after the other, process the matrix multiplication at once, and streams out the result using the same DMA. As a result, this architecture reduces the overall processing time.

Chapter Four

Results

The main goal of this research is the implementation of Compressive Sensing algorithms in real time systems. Embedded systems are the perfect candidates because of their parallel processing capabilities. The following are the results for the implementations discussed in the earlier chapters.

4.1 MATLAB Results

4.1.1 MATLAB Results for Optimal Matching Pursuit (OMP)

Initially a MATLAB code was implemented for the Optimal Matching Pursuit (OMP) with an 18×512 measurement matrix and an 18×1 observation matrix as inputs.

Figure 4.1 shows the reconstructed sparse signal, which is composed of 512 samples using Optimal Matching Pursuit (OMP). The simulation results in MATLAB show that the OMP is working as intended, and has perfectly reconstructed the solution vector from the measurement matrix and the observed matrix. Even more, Figure 4.1 shows that all samples other than the targets are zero. Therefore, OMP was able to reconstruct the radar scene perfectly.

4.1.2 MATLAB Results for Compressive Sampling Matching Pursuit (CSMP)

A 24×512 measurement matrix and a 24×1 observation vector are taken as inputs for the Compressive Sampling Matching Pursuit (CSMP) algorithm.

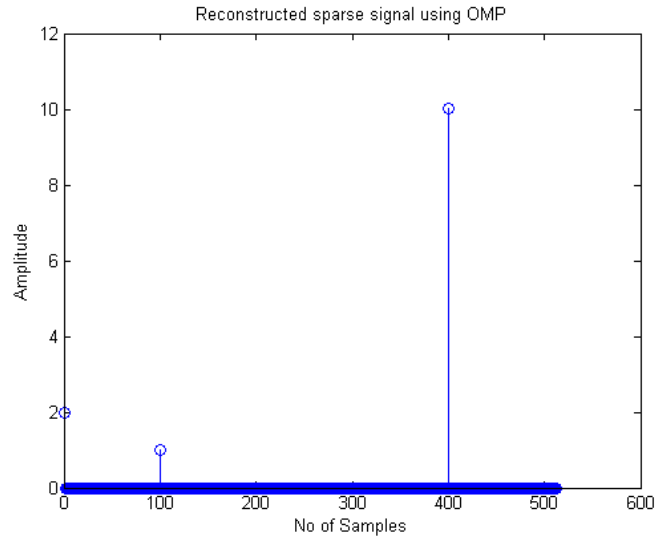


Figure 4.1: Reconstructed Sparse Signal using OMP.

Figure 4.2 shows a perfect reconstruction of the sparse vector of 512 samples using CSMP algorithm.

4.1.3 MATLAB Results for Stagewise Orthogonal Matching Pursuit (StOMP)

An 18×512 measurement matrix, and an 18×1 observation vector are taken as inputs for the Stagewise Orthogonal Matching Pursuit (StOMP) algorithm.

Figure 4.3 shows the reconstructed signal using StOMP. The simulation results in MATLAB show that the algorithm functioned as intended, and had reconstructed the signal with good accuracy.

However, a careful observation at the reconstructed vector indicates a small blip at sample 310. This is due to the way StOMP works. StOMP considers a preset threshold, and all correlations greater than the preset threshold are extracted in every iteration. As a result, a careful and accurate selection of the threshold parameter is required for the algorithm to work properly.

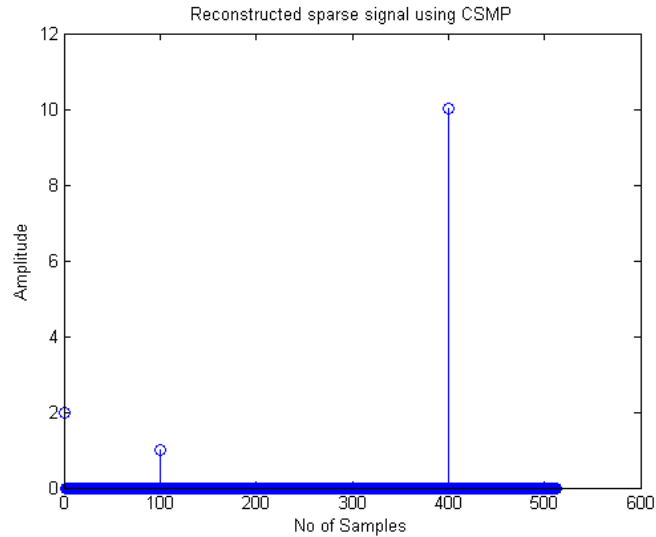


Figure 4.2: Reconstructed Sparse Signal using CSMP.

4.2 Execution Times in MATLAB

In this section, the execution times for the OMP, CSMP, and the StOMP in MATLAB will be presented. The codes were tested for different size matrices in order to compare their performance. Table 4.1 shows the execution times for an 18×512 measurement matrix. It is clear by looking at the data that the OMP is the fastest. It needs to be noted that it was not possible to perform the CSMP algorithm for an 18×512 measurement matrix, because the measurement matrix was not random enough for the CSMP to work. Table 4.1 and all further results in MATLAB discussed in this section are obtained when the algorithms were implemented in MATLAB on Intel corei7 processor running on a 64 bit operating system with 16GB RAM.

Table 4.2 shows the MATLAB results for a 24×512 input measurement matrix. In this case, the three different algorithms are working perfectly and they were able to reconstruct the signal.

By looking at the execution times it is evident that the OMP is faster than the

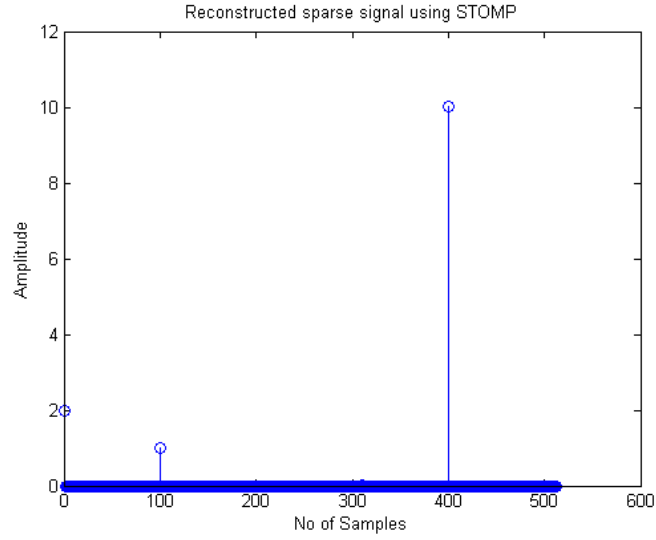


Figure 4.3: Reconstructed Sparse Signal using StOMP.

Table 4.1: Execution Times in MATLAB for a 18×512 Measurement Matrix and Three Targets.

	OMP	CSMP	StOMP
Execution Time	8.24 ms	*	36.889 ms

Table 4.2: Execution Times in MATLAB for a 24×512 Measurement Matrix and Three Targets.

	OMP	CSMP	StOMP
Execution Time	8.28 ms	39.39 ms	37.05 ms

CSMP and StOMP. However, it is very difficult to judge the performance of the algorithms for a small number of targets, especially the CSMP and the STOMP. It is also very difficult in MATLAB to measure accurately the time for algorithms which run in milliseconds, because MATLAB produces different execution time each time the algorithm is run. Therefore, the execution time of every algorithm was averaged out for 100 iterations. Finally, a measurement matrix of 700×1000 elements was considered. This matrix was generated using a chaotic radar system, and three targets

were placed randomly [40]. Figure 4.4 shows the reconstructed radar scene through optimization techniques.

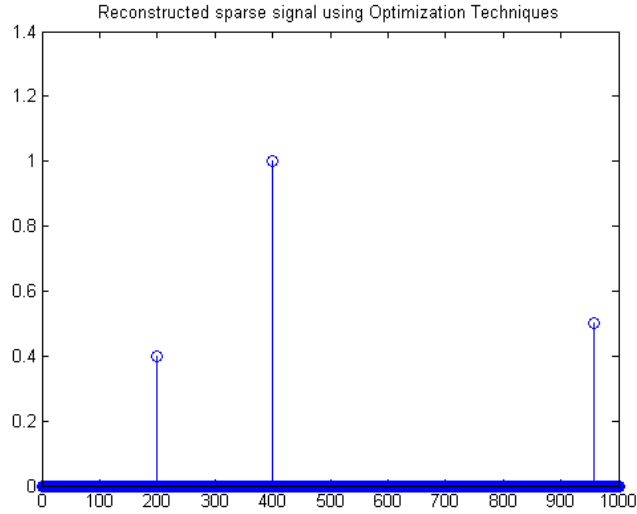


Figure 4.4: Reconstructed Sparse Signal using Optimization Techniques.

Table 4.3: Comparison of Execution Times of the algorithms for a 700×1000 Radar Measurement Matrix and Three Targets.

	Execution Times	Acceleration Factor against CVX
OMP	11.19 ms	536.19
CSMP	9.15 ms	655.73
StOMP	9.26 ms	647.94
CVX	6 sec	1

Table 4.3 shows the execution times for a 700×1000 radar data measurement matrix with three targets. The performance acceleration is obtained by comparing the codes running in MATLAB, against Convex optimization technique (CVX).

It is clear that the execution time for CSMP and StOMP for the bigger data set is lesser than the smaller data set. This is because these algorithms are highly dependent

on the data being processed, and the amount of information. In other words, the larger the amount of data to be processed, the more random the measurement matrix is, increasing the performance of the algorithms.

4.3 Execution Times in ARM Processor

There are three basic optimization levels as discussed below:

The first level optimization performs jump and pop optimizations. The second level optimization enhances nearly all optimizations that do not involve a speed-space trade-off. This kind of optimization is the general standard optimization from program deployment. The third level of optimization is the highest optimization level. This one increases more expensive options like using more resources, including those that increase code size. These optimization levels need to be applied with extreme caution, if not the optimization could lead to less efficient results than the actual implementation.

Table 4.4: Number of execution clock cycles of ARM Processor for an 18×512 measurement matrix and three targets.

Optimization level	OMP	CSMP	StOMP
No optimization (-O0)	251,347	*	434,846
Optimize (-O1)	47,851	*	85,001
Optimize more (-O2)	53,126	*	88,400
Optimize most (-O3)	4,907,613	*	5,024,685

Table 4.4 shows the number of clock cycles required to execute the algorithms for different optimization levels. It can be seen that the first level optimization works better for the algorithms. The second level optimization is worse than the first level optimization; However, it is better than the actual non optimized implementation. Third level optimization yielded much worse results than the actual implementation.

Table 4.5 shows the number of clock cycles required for each algorithm to be executed. It is clear that the optimizations depend on the algorithms, and are not same for all algorithms. The results show that the OMP and the StOMP require less number of clock cycles in the first level (-O1). However, CSMP required lesser

Table 4.5: Number of Execution Clock Cycles of ARM Processor for a 24×512 Measurement Matrix with Three Targets.

Optimization level	OMP	CSMP	StOMP
No optimization (-O0)	330,386	1,022,681	561,124
Optimize (-O1)	94,385	3,42,478	153,440
Optimize more (-O2)	96,993	319,479	160,381
Optimize most (-O3)	96,784	331,295	160,255

number of execution cycles in the second optimization level (-O2).

The execution time of the algorithms can be calculated using the formula equation 4.1

$$t = n \times T \quad (4.1)$$

where t is the execution time, n is the number of clock cycles and T is the time period of each clock cycle. In this specific case the timer is implemented on a Programmable Logic (PL) which is running at 100 MHz. The time period T can be calculated using the formula in equation 4.2

$$T = \frac{1}{f} \quad (4.2)$$

where f is the frequency. Table 4.6 shows the execution time calculated using equation 4.1 and equation 4.2, and a comparison between these values and the ones from using MATLAB are presented.

Table 4.6: Comparison of the Execution Times in MATLAB and ARM for a 24×512 Measurement Matrix with Three Targets.

Implementation	OMP	CSMP	StOMP
MATLAB	8.28 ms,	39.39 ms	37.05ms
ARM (Non-Optimized)	3.30 ms	10.22 ms	5.61 ms
ARM (Optimized)	0.94 ms	3.19 ms	1.53 ms
Accelaration Factor (Non-Optimized)	2.50	3.85	6.60
Accelaration Factor (Optimized)	8.80	12.34	24.21

A 700×1000 measurement matrix was taken as an input to the algorithms. Table 4.7 shows the number of clock cycles for different optimization levels.

Table 4.7: Number of Execution Clock Cycles of ARM Processor for a 700×1000 Radar Measurement Matrix with Three Targets.

Optimization level	OMP	CSMP	StOMP
No optimization (-O0)	17,198,286	12,812,619	45,143,213
Optimize (-O1)	*	5,160,452	11,081,729
Optimize more (-O2)	*	5,230,980	10,851,392
Optimize most (-O3)	*	5,236,724	10,828,840

The execution times for the results in Table 4.7 can be calculated using equation 4.1 and equation 4.2 The calculated execution times using equation 4.1 and equation 4.2 are shown in 4.8

Table 4.8: Comparison of the Execution Times in MATLAB and ARM considering a 700×1000 Measurement Matrix with Three Targets.

Implementation	OMP	CSMP	StOMP
MATLAB	11.19 ms	9.15 ms	9.26 ms
ARM No Optimization (-O0)	171.98 ms	128.12 ms	451.43 ms
ARM Optimize (-O1)	*	51.60 ms	110.81 ms
ARM Optimize more (-O2)	*	52.30 ms	108.51 ms
ARM Optimize most (-O3)	*	52.36 ms	108.28 ms

It is clear from the results that the MATLAB implementation is better than the ARM implementation. At this moment, there is no a clear reason for this result. However, it is conjectured that it is due to the following reasons.

1. MATLAB is being run on a computer with high configuration and also a key factor is that the ARM performs element by element processing whereas MATLAB process the data in vectorized way which means it considers the data as vectors or arrays and process it. Vectorization typically uses pre-compiled, optimized functions to execute it instead of running through the interpreter. Vectorization is typically useful in most of the matrix manipulations like matrix multiplication, dot product etc. Therefore this process is much faster than the traditional approach of doing element by element processing.
2. The performance of the algorithms also might depend on the kind of input data.
3. It also might depend on the proper selection of stack and heap sizes which can

alter the performance of the algorithms.

4.4 Execution Times for a 12×12 Matrix Multiplication Using ARM and Floating Point Units

The previous architecture with an ARM processor is implemented only on the processing system, and it doesn't use the Programmable Logic (PL). A timer is only implemented on the Programmable Logic (PL) for calculating the number of clock cycles.

The current architecture with an ARM and floating point units uses the Programmable Logic(PL) to implement the floating point units and therefore this architecture utilizes more resources than the previous architecture. It might be obvious that an architecture which utilizes more resources is faster than the architecture with lesser resources.

However, a careful observation of the architecture reveals that each and every element is being transmitted, processed, and sent back which increases the transmit and receive delays. Also, the calculation is made element by element processing and computationally intense task is not being processed in each iteration. Only one multiplication and one addition is performed in one iteration. Therefore this architecture is slower than the previous implementation of the ARM processor.

Table 4.9: Performance Results of a 12×12 Matrix Multiplication using an architecture with an ARM Processor, and ARM with Floating Point Units.

Architecture		Clock Cycles	Accelaration
ARM Processor	Non-Optimized	16,512	*
	Optimized	3,649	*
ARM and Floating Point Units	Non-Optimized	1,491,690	0.011
	Optimized	1,474,260	0.002

Table 4.9 shows the performance results for a matrix multiplication using an ARM processor and the architecture with an ARM and floating point units. As shown in Table 4.9, the number of clock cycles for a 12×12 matrix multiplication using an ARM processor is less than the architecture with an ARM processor and floating

point units.

4.5 Performance of Matrix Multiplication IP core

Table 4.10 shows the execution clock cycles and the corresponding acceleration factor for each optimization level in the ARM. Table 4.10 also shows the number of execution clock cycles from the synthesis report of the matrix multiplication IP core.

It can be observed that the acceleration factor is most for the non-optimized (a) case and is less for the second level of optimization (-O2). However, it is evident from the results that the FPGA implementation is accelerated even though the ARM is optimized.

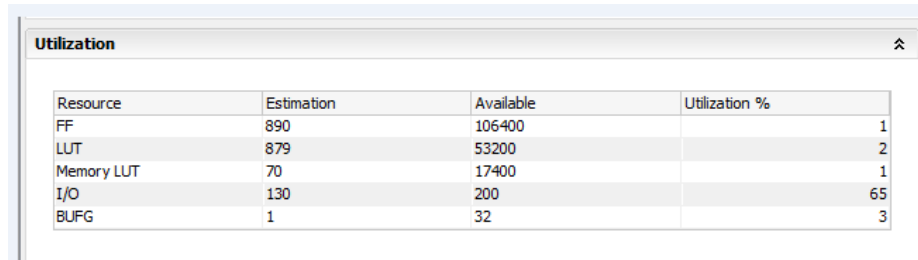
Table 4.10: Comparison of Execution Clock Cycles for a Matrix Multiplication IP.

Implementation	Optimization level	512x18	512x24	1000x700
FPGA matrix multiplication IP core (A)		10,363	13,471	706,213
ARM	No optimization (-O0) (a)	68,025	90,074	5,106,373
	Acceleration Factor (a/A)	6.56	8.69	7.23
	Optimize (-O1) (b)	19,602	27,778	16,19,757
	Acceleration Factor (b/A)	1.89	2.06	2.29
	Optimize more (-O2) (c)	13,100	19,829	1,149,934
	Acceleration Factor (c/A)	1.26	1.47	1.62
	Optimize most (-O3) (d)	20,404	27,851	1,618,986
	Acceleration Factor (d/A)	1.96	2.06	2.29

4.6 Resource Utilization Reports

Figure 4.5 shows the synthesis report showing the number of resources utilized of the ARM architecture. Figure 4.5 also shows that only few resources are utilized in the ARM architecture because we only have a timer implemented on the Programmable Logic. Notice that this is only a synthesis report and it takes into account the expected number of resources required. When the actual logic is implemented, the number of resources may increase or decrease depending on the complexity of the

design.



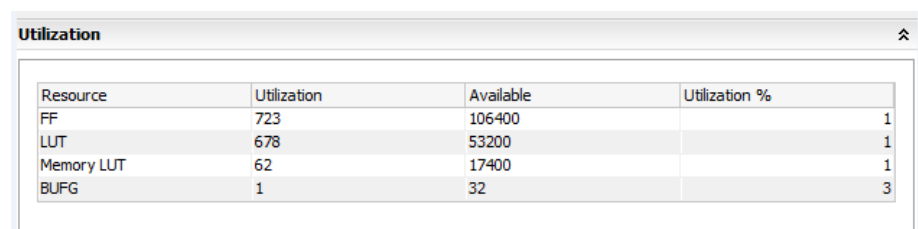
The image shows a screenshot of a software window titled "Utilization". It contains a table with the following data:

Resource	Estimation	Available	Utilization %
FF	890	106400	1
LUT	879	53200	2
Memory LUT	70	17400	1
I/O	130	200	65
BUFG	1	32	3

Figure 4.5: Synthesis Resource Utilization Report of ARM Processor Architecture.

Figure 4.6 shows the implementation resource utilization report of the ARM architecture. The implementation report takes into account the actual transmit delays, placement and routing, timing issues into account and updates the number of resources required accordingly.

Notice that the resource utilization of the implemented design is less than the resources required for the synthesized design.



The image shows a screenshot of a software window titled "Utilization". It contains a table with the following data:

Resource	Utilization	Available	Utilization %
FF	723	106400	1
LUT	678	53200	1
Memory LUT	62	17400	1
BUFG	1	32	3

Figure 4.6: Implementation Resource Utilization Report of ARM Processor Architecture.

Figure 4.7 shows the synthesis report of the resource utilization of the ARM and floating point units architecture. However, the actual resource utilization is found

from the implementation report of the implemented design.

Resource	Estimation	Available	Utilization %
FF	7019	106400	7
LUT	5626	53200	11
Memory LUT	185	17400	1
I/O	130	200	65
BRAM	4	140	3
DSP48	4	220	2
BUFG	1	32	3

Figure 4.7: Synthesis Resource Utilization Report of ARM and Floating Point Units Architecture.

Figure 4.8 shows the implementation resource utilization of the ARM and floating point units architecture. Figure shows that the implemented design resource utilization is less than the synthesized design resource utilization.

Resource	Utilization	Available	Utilization %
FF	6553	106400	6
LUT	5054	53200	10
Memory LUT	153	17400	1
BRAM	4	140	3
DSP48	4	220	2
BUFG	1	32	3

Figure 4.8: Implementation Resource Utilization Report of ARM and Floating Point Units Architecture.

The current ARM and floating point units architecture utilizes more resources than the previous ARM architecture. However, the Table 4.9 shows the current architecture of ARM and floating point units is less efficient and requires more clock cycles to process the floating point matrix multiplication. This is because the ARM

processor has a built in NEON Media Processor Engine (MPE) and Vector Floating Point unit (VFP) unit which is a dedicated hardware for floating point arithmetic [41]. Also transmit and receive delays need to be eliminated by streaming large data at once.

Figure 4.9 shows the synthesis report of the matrix multiplication IP core for input matrices of 512×18 and 18×1 . It is evident from the synthesis report that there are huge number of LUT's, Flip Flop's, and DSP blocks utilized which are key elements for matrix multiplication.

Performance Estimates

- ⊕ **Timing (ns)**
- ⊖ **Latency (clock cycles)**
 - ⊖ **Summary**

Latency		Interval		Type
min	max	min	max	
10362	10362	10363	10363	none

- ⊕ **Detail**

Utilization Estimates

- ⊖ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	19	90	8836	17397
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
ShiftMemory	-	-	-	-
Total	19	90	8838	17397
Available	280	220	106400	53200
Utilization (%)	6	40	8	32

Figure 4.9: Synthesis Resource Utilization Report of the Matrix Multiplication IP core for Matrix Inputs of 18×512 and 512×1 .

Figure 4.10 shows the synthesis utilization report for matrix multiplication with inputs 24×512 and 512×1 and it is evident from the synthesis report that this matrix multiplication block utilizes more resources than the previous matrix multiplication IP core with 18×512 and 512×1 input matrices. It is also obvious that

a matrix multiplication with more number of elements to be processed require more resources. It is observed in Table 4.10 that there is a considerable acceleration with this architecture implementation.

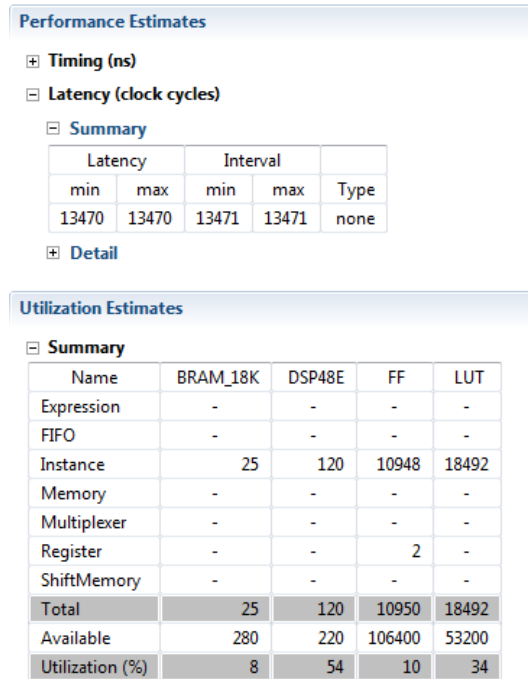


Figure 4.10: Synthesis Resource Utilization Report of the Matrix Multiplication IP core for Matrix Inputs of 24×512 and 512×1 .

Figure 4.11 shows the synthesis resource utilization of the matrix multiplication IP core with input matrices 1000×700 and 700×1 . It is observed from Table 4.10 that there is a considerable improvement with the current architecture of the matrix multiplication IP core. However, the practical implementation would not be possible because of the practical limitations of the available resources on the FPGA board. Therefore an efficient solution for this problem would be to use multiplexers and demultiplexers to reuse the resources by the process of resource sharing and resource reutilization.

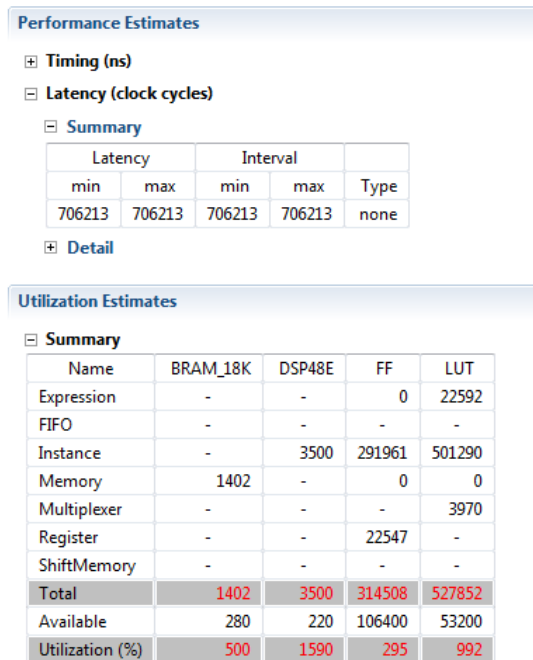


Figure 4.11: Synthesis Resource Utilization Report of the Matrix Multiplication IP core for Matrix Inputs of 700×1000 and 512×1 .

Chapter Five

Conclusion and Future Work

5.1 Conclusion

Signal processing and signal reconstruction is an extremely important factor in communication systems. In fact most under sampled signals could not be recovered back with the regular reconstruction methods. Compressive Sensing is proposed as a method to recover under-sampled signals; the only condition is that the signal of interest should be sparse. There are a number of optimization techniques to recover the under sampled signal. In this research work, it was decided to use the Optimal Matching Pursuit (OMP), Compressive Sampling Matching Pursuit (CSMP) and Stagewise Orthogonal Matching Pursuit (StOMP) to recover back the sparse signal and improve their performance.

Three architectures were designed in order to improve the performance of the optimization techniques. The first architecture of the ARM processor shows a reduction in computation time when compared to the software implementation in MATLAB. The second architecture utilizes the Programmable Logic (PL) section to implement a multiply and accumulate unit on the PL. This architecture utilizes more resources than the first one. However, its execution time is much more than the first architecture. The reason for this is that the second architecture perform operations element by element, destroying any possibility of parallel processing.

A third architecture was designed in order to leverage, and utilize the features of the FPGA. The third architecture utilizes more resources than the second architecture and includes pipelining that improves resource utilization. This architecture streams in a whole matrix, does matrix multiplication at a time and streams out the data at once. Data delays for transmitting and receiving each element has been reduced and also element by element processing had been eliminated by processing

the whole matrix multiplication in the FPGA.

5.2 Future Work

The current design implements only a portion of the code on the Programmable Logic (PL). Future work includes all time consuming submodules and tasks like the ordinary least squares to be implemented on a PL and improve the performance. Finally, future work can also include the implementation of the whole optimization techniques on the Programmable Logic (PL) and build a custom IP for each of the optimization techniques. It is expected to have a 100 times acceleration factor if the whole optimization technique could be implemented as an IP. However, implementing the whole optimization techniques on the Programmable Logic (PL) might pose a serious challenge due to limited number of resources on the Programmable Logic (PL). Therefore an efficient solution like pipelining which involves multiplexers and de-multiplexers needs to be incorporated so that we can reutilize the resources effectively.

References

- [1] E. J. Candes, and M. B. Wakin, “An Introduction To Compressive Sampling,” *Signal Processing Magazine, IEEE*, vol 25, no. 2, pp. 21-30, March 2008.
- [2] J. Romberg, “Imaging via Compressive Sampling,” *Signal Processing Magazine, IEEE*, vol.25, no.2, pp.14-20, March 2008.
- [3] R. G. Baraniuk and P. Steeghs, “Compressive Radar Imaging,” in *IEEE Radar Conf.* (Boston, U.S.A, April 17-20, 2007), pp.128-133.
- [4] D. L. Donoho. (2004, Sept-14). Compressed Sensing. Stanford University. Stanford, CA, U.S.A. [Online]. Available: <http://statweb.stanford.edu/%7edonoho/Reports/2004/CompressedSensing091604.pdf>
- [5] J. Romberg and M. Wakin, “Compressed sensing: A Tutorial,” presented at the IEEE Statistical Signal Processing Workshop, Madison, WI, Aug. 26, 2007, power point presentation.
- [6] R. G. Baraniuk, “Compressive Sensing [Lecture Notes],” *Signal Processing Magazine, IEEE*, vol 24, no. 4, pp. 118-121, July 2007.
- [7] E. J. Candes, J. Romberg, and T. Tao, “Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information”, *IEEE Trans. on Inform. Theory*, vol. 52, no.2, pp. 489-509, Feb. 2006.
- [8] A. Rzhetsky and M. Nei, “Statistical Properties of the Ordinary Least-Squares, Generalized Least-Squares, and Minimum-Evolution Methods of Phylogenetic Inference,” *Journal of Molecular Evolution*, vol.35, no.4, pp. 367-375, Oct. 1992.
- [9] Ordinary Least-Squares. Stanford University. [Online]. Available: graphics.stanford.edu/~jplewis/lscourse/ols_slides.ppt
- [10] I. Yanovsky. Math 151B QR Decomposition with Gram-Schmidt. University of California Los Angeles. [Online]. Available: <http://www.math.ucla.edu/%7eyanovsky/Teaching/Math151B/handouts/GramSchmidt.pdf>

- [11] L. Ma, K. Dickson, J. McAllister, and J. McCanny, "QR Decomposition-Based Matrix Inversion for High Performance Embedded MIMO Receivers," *IEEE Trans. on Signal Processing*, vol.59, no.4, pp.1858-1867, Jan. 2011.
- [12] A. Rosado, T. Iakymchuk, M. Bataller, and M. Wegrzyn, "Hardware-efficient matrix inversion algorithm for complex adaptive systems," in *2012 19th IEEE International Conf. on Electronics, Circuits and Systems (ICECS)* (Seville, Spain, Dec. 9-12 ,2012), pp.41-44.
- [13] J. L. V. M. Stanislaus and T. Mohsenin, "High performance compressive sensing reconstruction hardware with QRD process," in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)* (Seoul, Korea, May 20-23, 2012), pp.29-32.
- [14] J. A. Tropp and S. J. Wright, "Computational Methods for Sparse Solution of Linear Inverse Problems", *Proceedings of the IEEE*, vol.98, no.6, pp.948-958, April 2010.
- [15] A. Septimus and R. Steinberg, "Compressive sampling hardware reconstruction," *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, (Paris, France, May 30, 2010 - June 2, 2010), pp.3316-3319.
- [16] D. Needell and J. A. Tropp, "CoSaMP: Iterative signal recovery from incomplete and inaccurate samples", *Applied and Computational Harmonic Analysis*, vol.26, no.3, pp.301-321, May 2009.
- [17] D. L. Donoho, Y. Tsaig, I. Drori, and J-L Starck, "Sparse Solution of Underdetermined Systems of Linear Equations by Stagewise Orthogonal Matching Pursuit," *IEEE Transac. on Inform. Theory*, vol.58, no.2, pp.1094-1121, Feb. 2012.
- [18] Texas Instruments Inc., Dallas, TX, USA, "ARM Applications," Available: <http://www.ti.com/lscs/ti/arm/applications.page> Accessed: May 7, 2015.
- [19] ARM Inc., Cambridge, England, UK, "ARM Processors," Available: <http://www.arm.com/products/processors/> Accessed: May 7, 2015.
- [20] ARM Inc., Cambridge, England, UK, "ARM Features," Available: <http://www.arm.com/support/university/index.php> Accessed: May 7, 2015.
- [21] Xilinx Inc., San Jose, CA, USA, "A Generation ahead for Smarter Systems: 9 reasons why the Xilinx ZYNQ-7000 All Programmable SoC Platform is the

- smartest solution,” Available: http://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-background.pdf Accessed: May 7, 2015.
- [22] Avnet Inc., Phoenix, AZ, USA. *ZedBoard Hardware User’s Guide*. (2014) [Online]. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2.2.pdf, Accessed on: May 7, 2015.
- [23] Avnet Inc., Phoenix, AZ, USA. *Getting Started Guide*. (2012) [Online]. Available: <http://zedboard.org/sites/default/files/GS-AES-Z7EV-7Z020-G-14.1-V5.pdf>, Accessed on: May 7, 2015.
- [24] Xilinx Inc., San Jose, CA, USA, “Data specifications of the Zynq Series,” Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf Accessed: May 7, 2015.
- [25] Xilinx Inc., San Jose, CA, USA. *Zynq Technical Reference Manual*, (2015) [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, Accessed on: May 7, 2015.
- [26] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book*. Dept.of Electron. and Elect.Eng, University of Strathclyde, Glasgow, Scotland, UK, 2014. [Online]. Available: <http://www.zynqbook.com/downloads.html>
- [27] Xilinx Inc., San Jose, CA, USA. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator*, (2013) [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_3/ug994-vivado-ip-subsystems.pdf, Accessed on: May 7, 2015.
- [28] Xilinx Inc., San Jose, CA, USA. *Vivado Design Suite User Guide: Using the Vivado IDE*, (2014) [Online]. Available: www.xilinx.com/support/documentation/sw_manuals/xilinx2014.1/ug893-vivado-ide.pdf, Accessed on: May 7, 2015.
- [29] Xilinx Inc., San Jose, CA, USA. *Vivado Design Suite User Guide: Embedded Processor Hardware Design*, (2014) [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014.1/ug898-vivado-embedded-design.pdf, Accessed on: May 7, 2015.
- [30] Xilinx Inc., San Jose, CA, USA, “Targeting Zynq Using Vivado IP Integrator,” Available: <http://www.xilinx.com/training/vivado/targeting-zynq-using-vivado-ip-integrator.htm>. Accessed: May 7, 2015.

- [31] Xilinx Inc., San Jose, CA, USA. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (2013) [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug940-vivado-tutorial-embedded-design.pdf, Accessed on: May 7, 2015.
- [32] Xilinx Inc., San Jose, CA, USA. *Vivado Tutorial Using IP Integrator* (2014) [Online]. Available: http://www.xilinx.com/support/documentation/university/Vivado-Teaching/Digital-Design/2014x/docs-pdf/Vivado_tutorial.pdf, Accessed on: May 7, 2015.
- [33] Xilinx Inc., San Jose, CA, USA, “Xilinx Software Development Kit,” Available: <http://www.xilinx.com/tools/sdk.htm>. Accessed: May 7, 2015.
- [34] Xilinx Inc., San Jose, CA, USA. *Xilinx Software Development Kit Help Contents* (2014) [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/SDK_Doc/index.html, Accessed on: May 7, 2015.
- [35] Xilinx Inc., San Jose, CA, USA. *Xilinx Software Development Kit (SDK) User Guide: System Performance Analysis* (2014) [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug1145-sdk-system-performance.pdf, Accessed on: May 7, 2015.
- [36] Xilinx Inc., San Jose, CA, USA. *AXI DMA v7.1: LogiCORE IP Product Guide* (2015) [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf, Accessed on: May 7, 2015.
- [37] Xilinx Inc., San Jose, CA, USA. *LogiCORE IP Floating-Point Operator v7.0 Product Guide* (2014) [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf, Accessed on: May 7, 2015.
- [38] Xilinx Inc., San Jose, CA, USA. *Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HLS* (2013) [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf, Accessed on: May 7, 2015.
- [39] Xilinx Inc., San Jose, CA, USA. *Vivado Design Suite User Guide High-Level Synthesis* (2012) [Online]. Available: <http://www.xilinx.com/support/>

documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf, Accessed on: May 7, 2015.

- [40] C. T. Enugula, "A Compressive Radar System with Chaos Based FM Signals Generated Using The Bernoulli Map, M.S. thesis, Dept. Elect.Eng., Univ. of Texas at Tyler, Tyler, Texas, USA, 2012.
- [41] Xilinx Inc., San Jose, CA, USA. *Boost Software Performance on Zynq-7000 AP SoC with NEON* (2014) [Online]. Available: http://china.xilinx.com/support/documentation/application_notes/xapp1206-boost-sw-performance-zynq7soc-w-neon.pdf, Accessed on: May 7, 2015.