Design & Performance Analysis of Wireless Radar Receiver for Joint Radar Communication Systems on RFSoC

Shragvi Sidharth Jha
2019207

May 2022

Student's Declaration

I hereby declare that the work presented in the report entitled **"Design & Performance Analysis of Wireless Radar Receiver for Joint Radar Communication Systems on RFSoC"** submitted by me for the partial fulfillment of the requirements for the degree of *Bachelor of Technology* in *Electronics & Communication Engineering* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of my work carried out under guidance of **Dr. Sumit J. Darak**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

Shragvi Sidharth Jha
IIIT, Delhi

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Dr. Sumit J. Darak
IIIT, Delhi

Abstract

Autonomous vehicles are becoming prevalent day-by-day, and with the advent of such technology comes its various safety risks and challenges. The primary objective is to improve road safety and mobility for all users. The key enabling technology that has shown viable promise is millimeter wave communication, but has its limitations: Communication links are line-of-sight and directional. They experience very high propagation loss due to atmospheric absorption. A possible solution would be to use a Joint radar-communication transceiver where radar waveforms are embedded within the communication frame. Utilizing advanced machine learning framework to characterize the features of the JRC, such as range, doppler, angle of arrival of targets from corresponding radar echoes. The positives of such an approach is shared hardware, common waveform generation, no channel estimation necessary. The advantages we would receive are no additional cost for hardware, no requirement for separate radar spectrum leading to no interference, lower latency as channel estimation is not done. A hardware implementation for multiple target detection is discussed, the details of the implementation of the algorithms such as Matched-filtering, MUSIC and CLEAN are presented. A complete end-to-end system for multiple target detection using a Python based GUI is presented. Analysis and design details are thoroughly discussed, using RMSE plots for various fixed-point word-lengths along with Hardware Software Co-Design(HSCD).

Keywords: Joint Radar Communication (JRC), Radar Target Detection, Matched-filtering, MUSIC, CLEAN, FPGAs

Acknowledgments

I would like to thank my mentor Dr. Sumit J. Darak (Associate Professor, IIIT Delhi) for giving me this opportunity to take up the project. I thank him for his immense guidance, patience and encouragement.

Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Background 2

1.2.1 FPGA 2

1.2.2 Radar Signal Processing Algorithm 2

1.3 Objectives 4

1.4 Work Flow 4

1.5 Awards 5

1.6 Organization 5

2 Implementation of Radar Signal Processing Algorithm **6**

2.1 Python Based Implementation 6

2.2 Algorithms Overview 8

3 Radar Signal Processing Architecture **11**

3.1 HLS Based Implementation 11

3.1.1 Idea to RTL Flow 11

3.1.2 FFT IP 12

3.1.3 Hardware Optimization 12

3.2 Proposed Architecture 14

3.3 Matched Filtering Architecture 16

3.4 Optimizing RTL & Analysis 21

3.4.1 Optimizing RTL 21

3.4.2 Analysis of Pragma 21

4 Word Length Analysis **23**

4.1 Approach 23

4.2 Resource Utilization 24

4.3 Functionality Results 25

5 Hardware Software Co-Design **27**
5.1 Approach 27
5.2 Results 27

6 Conclusion & Future Works **29**
6.1 Conclusion 29
6.2 Future Works 29

Chapter 1

Introduction

This chapter covers the introduction to the area of research, objective and work done. Later give a background regarding the radar signal processing algorithm, and an introduction to FPGAs is provided.

1.1 Problem Statement

Autonomous vehicles and self-driving cars has seen great improvement in terms of technological advancement in the past decade. Various field such as environment sensing, vehicular control, and communication among various vehicles subsequently has also seen, massive improvement. However, there is still a challenge when it comes to target detection, and environment awareness, such leads to issues in terms of road safety. Proposed solutions involving cameras and lidar have their merits, but there drawbacks in adverse weather conditions is still unanswered. Radar offers the necessary feature set to tackle the issues stated. An Intelligent Transportation System(ITS), which consists of Vehicle-to-Everything(V2X) communication, X could be other vehicles(V2V), pedestrians(V2P), or other infrastructure present in the vicinity(V2I). Such a framework would result in the increased road safety, due to data being accessible to all entities present in the area.

That is where the proposed Joint Radar Communication(JRC) System comes in to the picture, an architecture that not only provides the benefits of Radar but also that of the standardized mm-Wave. Radio-Detection-and-Ranging (RADAR) is a detection system based on Radio Waves, used to calculate and determine the range, angle and velocity of the target. A complete radar system consists of transmitter antenna, a receiver antenna-which is commonly the transmitter antenna itself, and a processor that takes in the transmitter and receiver data to calculate the desired parameters. The underlying principle of radar systems rests on the transmitting various wave forms and the received signal being processed via the receiver antenna, where the processing of DOA, velocity and Range of the target can be done. Although there are many benefits to the mm-Wave Radar architecture, one major downside is the Range capability of the Radar. The current JRC architecture is suited for ultra-short range. Which are able to operate for distances up to 40-m of range.

The complete JRC architecture consists of various blocks, such as a transmitter, which generates the Radar waveform, which in this is a IEEE 802.11 ad based waveform, a receiver block and a signal processing block. The receiver block, essentially consists of a delayed version of the transmitting waveform but with additional information about the targets velocity and the DOA. This report although talks about the signal processing block, which takes the transmitted

waveform and the received waveform as inputs and provides the target data, range, velocity and DOA. The methods and algorithms used are matched filtering for Range & DOA calculation and the MUSIC algorithm for Doppler velocity estimation.

A baseline MATLAB version which was developed by PhD students under Dr.Shobha Sundar Ram is as reference. We first started with implementing the MATLAB design in Python using various packages such as NumPy and SciPy so that maximum optimization could be achieved via these packages and bench-marking the system on the RFSoC's Processing System(PS) this provides our baseline results. Similarly, the radar signal processing blocks have implemented on the FPGA using Vivado High-Level-Synthesis(HLS). Later on additional optimizations, software and hardware were performed to measure the performance and resource utilization and latency, finally deciding on an architecture that can be implemented on the RFSoC to achieve the desired results in terms of area utilization, latency and throughput. Other optimizations were added to improve area utilization and latency such using fixed-point arithmetic over floating point precision, Hardware Software Co-design was performed to improve power consumption and reduce area utlization of the Xilinx Ultrascale+ RFSoC.

1.2 Background

1.2.1 FPGA

Field-programmable-gate-array (FPGA) is a semiconductor device that is designed to be configured as per the designers after the manufacturing stage is completed. This is achieved by the architecture of the FPGA, which is a sea of configurable logic blocks(CLB), connected to each other via programmable interconnects. FPGA have broken into many different markets due to their programmable nature.

In order to map our RSP algorithms effectively on to the FPGA, we follow a technique called hardware-software co-design, for this to be possible we have chosen the Zynq Ultrascale+ RFSoC, which is a complete System-on-Chip(SoC), consisting of Hard IP processors such as the Quad Core ARM Cortex-A53, along with 16nm FinFet+ Programmable Logic.
The RFSoC integrates multi-gigasample RF data converters in the SoC architecture itself, making the industries only single-chip, re-configurable radio platform.

On top of all the processing capabilities the RFSoC has it can also run a complete Linux based operating system, which is called the PYNQ framework. PYNQ stands for Python on ZYNQ, it is a fully functional Linux based operating system that has a Jupyter Notebook based interface to communicate with the board. The web hosted Jupyter Notebook enables us to run our designs on the PL part of the RFSoC, while the Processing System handles the challenges that occur while running the operating system. This also enables us to make great visualizations of the models we are trying to process. Which is used to create a Graphical-User-Interface(GUI).

1.2.2 Radar Signal Processing Algorithm

The algorithms that are implemented on the RFSoC are matched filtering, MUSIC and CLEAN. Matched filters are often used in signal detection to correlate a known signal, or template, with an unknown signal to detect the presence of the template in the unknown signal. This is equivalent to convolving the unknown signal with a conjugated time-reversed version of the template. In order to judge the range of an object we can send the transmitted signal and wait for the reflected signal, matched filtering is done to determine whether or not the received

signal was in fact reflected off an object of our interest. In this architecture matched filtering provides the Range and the DOA or Azimuth angle of the target. By first applying an azimuth angle delay on the received signal and the performing matched filtering we can figure out the precise angle for which the target is present as the result of matched filtering will peak at one of the possible DOAs. This operation is typically done in the time domain using correlation, but this process for larger matrices will lead to massive memory requirements, therefore here the matched filtering is done in the frequency domain, which essentially is matrix multiplication. At the cost of Fourier Transform & inverse Fourier Transform blocks we save a lot of area. Multiple Signal Classification (MUSIC) is used in order to estimate the Doppler veloctiy of our targets. The basic idea of MUSIC algorithm is to conduct characteristic decomposition for the covariance matrix of any array output data, resulting in a signal subspace orthogonal with a noise subspace corresponding to the signal components. Then these two orthogonal subspaces are used to constitute a spectrum function, be got though by spectral peak search and detect DOA signals. Typically MUSIC is utilized to perform DOA estimation, but using the similar concept behind DOA estimation it can be used for Doppler Estimation. The reason for such a choice is Performance vs Area. In order to accurately detect the targets velocity using matched filtering roughly 4096 packets would be needed, each packet can be considered some to have a small amount of information about the transmitted signal. As matched filtering would be used for calculate the Doppler velocity performing a matrix multiplication which has 4096 columns efficiently would not be feasible leading to large area utilization. This is where the MUSIC algorithm comes into play, in order for the MUSIC algorithm to be able to detect the Doppler velocity 100 packets are required. This drastically reduced the area utilization allowing more parallelism hence improving our latency and throughput. Once a certain number of pacckets have been collected, an index based maximum search is done across all packets, generating the input to the MUSIC IP. Auto-correlation is performed on the input vector and its transpose generating a square matrix of size equivalent the number of packets across the rows and columns. QR Decomposition is followed swiftly afterwards to generate the appropriate eigen-vectors, using the eigen-vectors a process known as noise subspace estimation is done. After which we generate the pseudo spectrum consisting of the MUSIC result. The peak of the spectrum corresponds to the Doppler velocity of the target we are trying to detect. CLEAN is the final part of the Radar Signal Processing block, CLEAN is used in order for multiple target detection to be feasible. The idea behind CLEAN is to remove the currently existing peak that has been obtained from matched filtering and MUSIC, this corresponds to one of the many targets that may be present in the environment. Once the peak is removed we perform matched filtering and MUSIC again to obtain the next peak. CLEAN is then performed in a iterative fashion until we have obtained all of targets.

To our Radar Signal Processing block we consider the Transmitted signal(XMAT) and Received Signal(RXMAT) to be inputs. Given the number of Antennas in the System we create an Azimuth Delay matrix. In our case the number of antennas are 32. The azimuth delay is computed for each angle ranging from -90°to 90°with a resolution of 1°using the formula $exp(-1j * 2 * \pi * \sin\{\theta\} * d/\lambda)$ Here θ is the angle under consideration, d is the antenna spacing for the multi-antenna system in the JRC system the antenna spacing is $\lambda/2$. Now for every possible combination of packet and angle we perform the matched filtering in the frequency domain followed by MUSIC. Repeat the process after CLEAN for each target present in the environment.

1.3 Objectives

The objective of the project is to design and analyze the performance of the radar receiver that is implemented in the JRC system using the HLS design approach. In HLS, the aim is to explore various user directives and analyze the trade-off between various factors while optimising the hardware architecture. The signal processing algorithms being implemented here would be the matched filtering, MUSIC & CLEAN, we do that to extract information such as Range, Doppler velocity and DOA of the target. One of the major goals is to be able to detect multiple targets located at different positions varying in size. All of these operations need to be performed in conjunction while achieving low latency and high throughput. Design of an intuitive GUI which has the capabilities of setting the Range, Doppler velocity and Azimuth of different targets and presenting the output plots of matched filtering, MUSIC and CLEAN. Development of new architectures at the algorithm level to further improve the target detection performance.

1.4 Work Flow

This semesters work is in continuation of last semester.
Semester 1:

- Implementation of Range-Doppler matched filtering on RFSoC PS
- Development of Hardware blocks for matched filtering and testing Hardware Software Co-Design
- Analysis of different user directives in HLS and the impact it has on resource utilization and latency.

Semester 2:

- Implementation of matched filtering, MUSIC, CLEAN on Processing System of RFSoC to detect Range-Doppler-Azimuth.
- Development of matched filtering hardware for RFSoC using HLS.
- Analyzing the latency of different matched filtering configurations and parallelism it incorporates.
- Integrating the matched filtering design with the PS implementation consisting of MUSIC and CLEAN, and testing for different HSCD.
- Creating a prototype GUI for a complete demonstration of the working system.

Semester 3:

- Word length analysis of the Matched Filtering and MUSIC IP was done through Vivado HLS.
- Functionality results of the complete system across various Signal-to-Noise Ratios was performed.
- Hardware Software Co-Design analysis was conducted, demonstrating the Area-Performance trade-offs.

- Completion of the GUI, making it more user accessible, allowing various settings to be incorporated.
- Development of newer architectures to further improve performance of Range, Azimuth, Doppler detection.

1.5 Awards

The work presented here has recieved a couple of awards over the past year,

- Winner of the Qualcomm Innovation Fellowship(QIF) 2022.
- VLSID 2022 Conference Design Contest Winner
- Recipient of the Chanakya Fellowship by iHub-Anubhuti IIIT-Delhi

1.6 Organization

Implementation and results of the report has organized in the following manner.

- Chapter 2 covers the implementation of the Radar Signal Processing Algorithm in Python, and in HLS. It also discusses optimization that we made to make the run-time smaller. Hardware Optimizations were also discussed. Proposed architecture and representations using block diagrams are also provided.
- Chapter 3 covers the results and compares the purely software approach of solving the issue to a hardware-software co-design approach. Architecture choices were made their results were discussed in accordance to the various optimization that were possible, as discussed in chapter 2.
- Chapter 4 goes over the Word length analysis in detail
- Chapter 5 covers the Hardware Software Co-design, the approach taken to the results have been discussed.
- Chapter 6 concludes the entire paper and the future scope of the work.

Chapter 2

Implementation of Radar Signal Processing Algorithm

We discuss the implementations of the radar signal processing algorithm. The first implementation in Python was used to get a baseline results. These results were identical to the MATLAB implementation, and the performance of the code was also tested on the RFSoCs Processing System(PS). Later an architectural discussion of the entire Radar Signal Processing is discussed. With emphasis on the hardware blocks implemented on the RFSoCs Programmable Logic(PL) for hardware acceleration.

2.1 Python Based Implementation

The Python based implementation was done using NumPy and SciPy. The simulations in Python are done with multiple point targets in mind present at different locations for a common. A radar with fixed parameters was taken, these parameters include PRI, PTM, Golay Sequence Length, transmitted signal power, antenna gain. The antenna setup taken into consideration is multi-antenna setup consisting of 32 antennas, with antenna spacing being $\lambda/2$.

In order to create the transmitting signal we have to firstly decide on a waveform, in this case that wave form is the IEEE 802.11 ad wave form. From the entire waveform we extract the data component leaving us with 512x100 matrix, 512 being the length of the data sequence, 100 being the number of packets, let us call this the XMAT. As each antenna will be transmitting the same XMAT we can just create a 3D matrix of size 32x1024x100 where the contents of the different antennas are the same.

Once we have the transmitted wave form and have modelled our targets, the generation of the received signal needs to be done. Let us call this signal RXMAT. As we have multiple targets, the same XMAT is used to generate that targets corresponding RXMAT. Additional RADAR parameters have been adjusted in the generation the RXMAT, the parameters include:

- Channel: Friis Free Space
- Transmitted Power: 40dBm
- Range resolution: 0.08m
- Azimuth resolution: 1°

- Doppler Velocity resolution: 0.3 m/s
- No. of Packet for Doppler Processing: 100 Packet
- SNR: varied from -25dB to +10dB

All the targets RXMATs are then super-positioned to create the complete RXMAT that will be sent to the matched filtering blocks. So for a particular target in order to incorporate range we delay XMAT, the way we calculate the delay is by using $2 * TargetRange/c$, essentially zero padding the start of the XMAT, next in order to include Range and Azimuth, we create a Doppler Delay and Antenna Matrix respectively and multiply our delayed signal with those matrices. The Doppler delay is $exp(1j * 2 * \pi * 2 * TargetVelocity * TimeArray/\lambda)$, where time delay is the sum of the slow time axis and fast time axis. After doppler delay incorporation we have a 1024x100 matrix. The antenna matrix is $exp(-1j * 2 * \pi * \sin(TargetDOA) * d)$, where d is the antenna spacing. In order to include the effect of antenna matrix we multiply the post doppler delay matrix with antenna matrix for every antenna. This finally creates the RXMAT for a single target whose dimensions are 1024x100x32.

Once both RXMAT and XMAT have been, the radar signal processing can begin. There are three major blocks that need to be discussed, firstly the matched filtering block. Typically matched filtering is performed in the time domain where cross correlation is done, but as the spacial complexity for that algorithm is quite resource intensive we decided to perform the matched filtering in the frequency domain. That consists of first performing a Fast Fourier Transform(FFT) on the XMAT & RXMAT. Then for each packet we iterate over each possible angle, where the target could be located, we calculate the azimuth delay using the formula $exp(-1j * 2 * \pi * \sin\{\theta\} * d/\lambda)$, then performing a series of matrix multiplications that involves incorporating the azimuth delay in the RXMAT echo and its subsequent multiplication with XMAT to find the changes in the transmitted and received signal, and finally doing the Inverse Fast Fourier Transform(IFFT) to get the output vector in the time domain. Doing this entire process for all the angles and packets generates an output matrix of size No. of Packet * No. of Possible Angles * Radar Length (100 * 181 * 1024), this matrix will be called Y for future purposes.

Once we have our Y matrix, we have the necessary information to estimate the Range and Azimuth/DOA of the target, now to calculate the Doppler velocity we use the MUSIC algorithm. First we locate the peak of Y[i] where i $\in \{1, 2, 3, ... 100\}$. This creates an vector of size Packet Length, whose values are the peak of that corresponding packet in Y. Now MUSIC algorithm is performed on this array. Auto-correlation is performed on the input vector and its transpose generating a square matrix of size equivalent the number of packets across the rows and columns. QR Decomposition is followed swiftly afterwards to generate the appropriate eigen-vectors, using the eigen-vectors a process known as noise subspace estimation is done. After which we generate the pseudo spectrum consisting of the MUSIC result. The output of the MUSIC algorithm provides a vector of size 200 this constitutes of the power spectral densities, called pmusic. The location of this peak would correspond to the Doppler velocity of the target.

Next we need to CLEAN, so that we can find the next peak. The CLEAN algorithm takes our 3D Y Matrix that we got from matched filtering and the corresponding peaks that would include the Peak Range, Azimuth and Velocity. Using this information we can model a situation where only one target is present in the environment which has this Peak Range, Azimuth and Velocity, effectively creating a new Dummy RXMAT. The method to create the dummy signal

is similar to how we created the RXMAT array, we do not include RADAR parameters in the generation of the signal as that the Hardware dependent, as well as not adding additional noise to the dummy signal. Using this new dummy RXMAT we can do the steps of matched filtering to get the Dummy Y matrix, take the superposition of our Original Y matrix and -Dummy Y Matrix, essentially leaving us with residue after the CLEAN operation has been completed. For the residue Y matrix we can find the new peak range and azimuth, and to calculate Doppler velocity perform MUSIC on the residue Y matrix and get its peak. Finally repeating the same CLEAN algorithm steps as many times we want corresponding to the number of targets we want to detect in the environment. The Figure 2.1 details the points mentioned through a diagram.

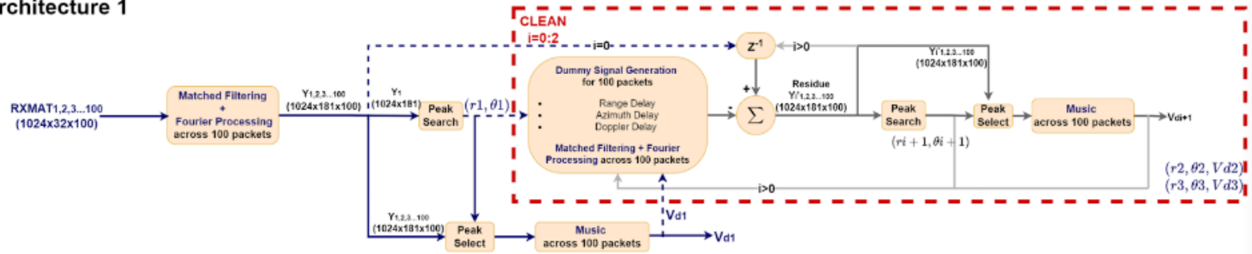

Figure 2.1: RSP Algorithm Overview

2.2 Algorithms Overview

Data: Radar Parameters, Target Parameters, XMAT
Result: Range, Doppler, Azimuth of Detected Targets
RXMAT = RXMAT Signal Modelling(Radar Parameters, Target Parameters)
Y = Matched Filtering(RXMAT, XMAT)
PMUSIC = MUSIC(Y)
Range, Doppler, Azimuth = Peak Extraction(Y, PMUSIC)
new_Y, new_PMUSIC = CLEAN(Y)
new_Y, new_PMUSIC = CLEAN(new_Y)
new_Range, new_Doppler, new_Azimuth = Peak Extraction(new_Y, new_PMUSIC)
Repeat CLEAN until all targets are visible

Algorithm 1: Overall Radar Signal Processing Algorithm

Data: RXMAT, XMAT
Result: Y
FFT_RXMAT = FFT{RXMAT}
FFT_XMAT = FFT{XMAT} **for** *Each Packet* **do**
for $\theta \in$ {*-90° to 90°*} **do**
azimuth delay = $exp(-1j * 2 * \pi * \sin\{\theta\} * d/\lambda)$
Delayed_RXMAT = RXMAT * Azimuth Delay
Y = IFFT{DIAG-Matrix-Multiplication{Delayed RXMAT, XMAT}}
end
end

Algorithm 2: Optimized Matched Filtering Algorithm

Data: Y
Result: PMUSIC
Find the Peak for any one Packet
x, y = find-peak(Y[0])
for $i \in \{1, 2, 3 \ldots 100\}$ **do**
  music_input[i] = Y[i][x][y]
end
Cov-Matrix = music_input * music_input'
eigen-vector-music = Calculate-Eigen-Vector(Cov-Matrix)
for $i \in$ *Doppler Axis* **do**
  NSS = Generate Noise Subspace
  PMUSIC-c[i] = NSS * eigen-vector-music * eigen-vector-music' * NSS'
  PMUSIC-c[i] = abs(1/PMUSIC-c[i])
end
PMUSIC = 10 * log10(PMUSIC-c)

Algorithm 3: MUSIC Algorithm

Data: Y, PMUSIC
Result: CLEAN Y, CLEAN PMUSIC
x, y = argmax(Y)
z = argmax(PMUSIC)
DummyTargetData = Generate_RXMAT(x, y, z)
Dummy_Y = matchedfiltering(DummyTargetData, XMAT)
CLEAN Y = Y - Dummy_Y
CLEAN PMUSIC = MUSIC(CLEAN Y)

Algorithm 4: CLEAN Algorithm

The entirety of the Radar Signal Processing Algorithm took 1,037.5 seconds on the RFSoC's PS after software optimizations and reducing computational complexity wherever feasible. In the next chapter improved architectures are discussed which is able to drastically reduce the execution time on the Xilinx Ultrascale+ we are able to achieve this by taking certain assumptions of the targets present, and remove redundancies in the processing.

Chapter 3

Radar Signal Processing Architecture

This chapter goes over the optimizations that were made in HLS so that, better and improved performance can be achieved. It also goes over the architecture of the matched filtering algorithm implemented using HLS.

3.1 HLS Based Implementation

High-Level-Synthesis(HLS) was used to create hardware accelerators for the Matched filtering & MUSIC IP and indirectly the CLEAN algorithm as it utilizes the matched filtering and MUSIC IPs. Using the IPs, benchmarking various HSCD cases are analysed.

3.1.1 Idea to RTL Flow

The steps involved in designing the matched filtering IP were:

1. Specification of the algorithm in C/C++
2. Designing the IP with AXI Stream Interface
3. Optimizing the design with User defined directives provided by Vivado HLS
4. C++ to RTL Flow
   (a) Simulation: Compiling and testing of the algorithm using C/C++ test-bench
   (b) Synthesis: Check synthesisability of design and map it to corresponding hardware. Timing results and hardware utilization is reported after synthesis.
   (c) C/RTL Simulation: Verifies the RTL generated for functionality through the C test-bench
   (d) Export IP: Export the RTL design as an IP along with specification of the driver files, used to integrate the IP with Processor Subsystem

3.1.2 FFT IP

Starting with the FFT IP, to implement the FFT operation we choose to use the Xilinx LogiCORE FFT IP. The IP can implement forward as well as inverse Fourier transform in an optimised manner. Some key features of the IP include:

- The transform size can range from 2^3 to 2^{16}.
- Fixed point precision of the data can range from 8 bits to 34 bits
- Phase factor can too range from 8 bits to 34 bits
- Both Fixed-point and floating-point arithmetic are supported.
- Block RAM or distributed RAM can be used for data storage.
- Multiple parameters can be set during run-time
- Bit-reversed and Natural ordering of Data is supported.
- Multiple architectures are supported each having its performances benefits.

Next we wanted to configure the FFT IP as per our requirement.

We can decide the number of channels for the FFT IP, which is essentially the number of FFT operations that will take place in parallel. For our use case we want a one Inverse Fourier Transform so we will choose a single channel. The transform length is the point-size of the FFT operation. The FFT core is optimized to perform the operation for the chosen point-size, hence deciding it before hand is preferred. As we want to convert our 1024 sized vector to the time-domain we will choose a transform size to be the same. As for the nature of the data, we can choose between floating and fixed point. One major challenge we faced is the implementation of the fixed point FFT IP, due to the lack of documentation, various trials were made to get the fixed point version working, additionally there are restriction to the total number of bits in the fixed point word length. Only multiples of 8 are permitted, the number of integer bits are restricted to 1, more information on fixed point implementation is provided in later chapters. Our final choice was the natural ordering of the output data, the core an option to present the data in a bit-reversed manner or in the natural ordering. For our use case we decided to settle with the natural ordering option, even though it utilizes more resources.

3.1.3 Hardware Optimization

The first challenge we needed to tackle in order to implement the complete matched filtering IP, was reduce BRAM utilization, as the FPGA can only store a certain amount of data, carefully managing the usage was critical. The idea was to send data packet wise, this led to a 1024x32 matrix being sent 100 times, and the resultant output was a 1024x181 matrix, both matrices being complex in nature. Certain optimizations were made, for the hardware accelerator we have decided to compute one packet at a time, send the new packet data sequentially for all the 100 packets. That means matched filtering will be done for RXMAT[:][:][PACKET] & XMAT[:][:][PACKET] in the PL. As for a single packet all the antennas are transmitting the same data, meaning XMAT[0][:][PACKET] == XMAT[1][:][PACKET] therefore in the PL we will store 1024 complex numbers in the BRAM. Hence the size requirement reduces from 1024 * 32 to 1024 drastically. The hardware IP has the Azimuth delay vectors for all the possible range

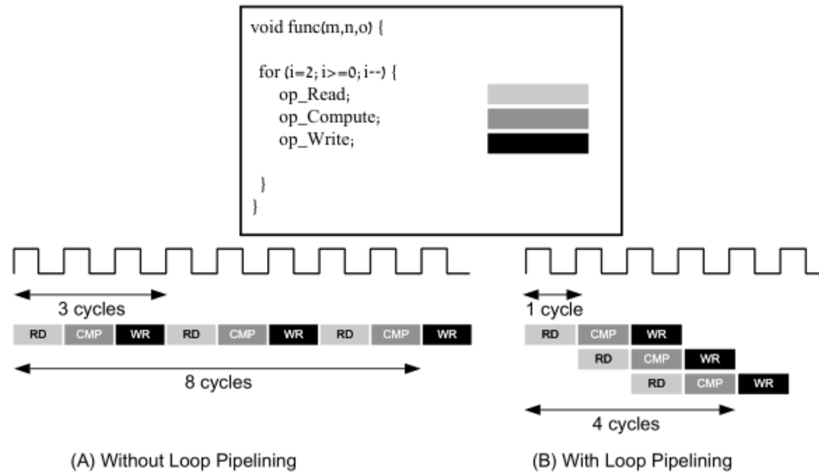

Figure 3.1: HLS Pipeline Diagram

of angles stored at the time of creation in the form of Look-Up-Table(LUT). As for incorporating the azimuth delay in the RXMAT using HLS user directives, maximum possible parallelism has been incorporated. The matrix multiplication that was required is not reduced to a matrix vector multiplication where only diagonal elements are calculated, as those will be the elements we desire for the final IFFT operation that needs to be done. Therefore the accelerator will take in RXMAT[PACKET] & XMAT[PACKET][0] and produce Y[i][181][1024] and just repeat this process for all packets.

Other optimization are Vivado HLS specific, these include Pipelining for-loops,Unrolling for-loops and partitioning of Block-RAMs(BRAMs). All these optimizations are directives that can be applied in Vivado HLS, the tool is used to synthesize the C++ code to RTL so that we can program the PL of the FPGA. These directives, also known as pragmas can be used so that the generated RTL can reduced latency, improved throughput, and lower the overall area usage of the PL.

HLS PIPELINE, is pragma directive that is used to lower the initiation interval of a function or loop by allowing concurrent execution of operations. The initiation interval is the number of clock cycles that is needed for a new input of the function to be processed. Ideally we would want an initiation interval of 1, meaning that we processing a new input at every clock cycle. But this is sometimes not feasible due to the complexity of the algorithm we want to implement. In our design we also took this factor in mind and decided the other optimization parameters to achieve a lower initiation interval.

HLS Unroll, is a pragma directive that allows us to create multiple independent copies of a certain operation. This would essentially create multiple copies of the loops body in RTL, allowing us to perform some if not all the iterations of the loop in parallel. Normal C++ loops are rolled, converting this to RTL would mean that the resources needed in the loops body would only be created once but reused for all the iterations. By adding unroll we are enabling greater throughput and increasing the need for data access. The next choice we need to make is whether we want to completely unroll the loop, making 'number of iteration' copies of the loop body or partially unroll to a certain factor, N. Fully unrolled loops let us compute the entire for loop in parallel, while partially unrolling limits this to N concurrent operations which are then performed sequentially until all the necessary operations are done.

HLS Array-partitioning, as we are increasing the data access required by unrolling the loops, we

also need to increase the number of ports that can be used to read the data from the buffers. As the buffers we want to implement will be synthesized using Block-RAMs(BRAMs) and they have two ports to read from we need to divide this large chunk of memory into smaller divisions that allows us to meet the increased data access requirement. Array-partitioning essentially does this, the RTL synthesis will create multiple smaller memories, effectively improving the throughput along the way. Array partitioning can be done in three ways, block, cyclic or complete. In block partitioning we are creating N separate blocks, N is a factor decided by us, then splitting the entire buffer in these N blocks equally. Cyclic also creates N blocks but the method of assigning values to those blocks is different, if we have chosen an N of 5 what will happen is that first element is assigned to block 1, second element to block 2, third to block 3 and so on. Once we reached the fifth element the cycle starts again.

HLS DATAFLOW, it is equivalent to pipelining the function calls and the loops to increase concurrency of the operations. It can be applied to the top level function only. By default, HLS performs all the function calls sequentially. For example, in a case when a loop or function writes data to a variable which is to be read by the next function call or loop, the latter will not start its execution until the former completes its execution. Using DATAFLOW directive here, the second function will start its operation before the previous operation completes its execution.

3.2 Proposed Architecture

A complete PL implementation of the radar signal algorithm that is used to determine Range, Doppler velocity and Azimuth is shown below in Figure 3.2.

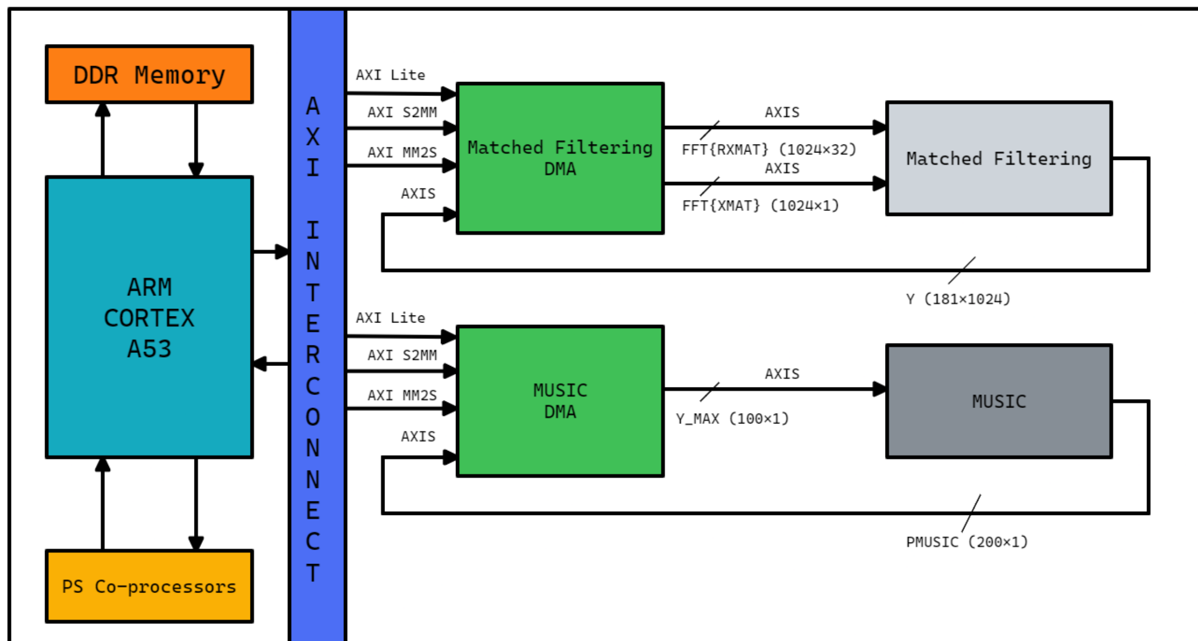

Figure 3.2: Proposed Architecture Overview

This architecture visualisations gives the big picture of the radar signal processing blocks that are implemented in the PL. As the CLEAN algorithm utilizes these blocks we can reuse this design for that algorithm. The AXI signals from the interconnect to the DMAs are used initializing the DMA and for allowing data communication with the DDR Memory. The AXI Stream protocol

is used for transferring the data from the Hardware Accelerator to the DMA.

The flow of the design, is to utilize the PYNQ OS running on the ARM Cortex A53 to generate the RXMAT that will be sent to the Hardware Accelerator based on the radar parameters used for modelling the entire environment. This includes all the parameters we mention in the previous chapter as well the SNR for the environment. Then using the PYNQ drivers for reading and writing to the DDR memory via the DMA for efficiently sending the large amount of RXMAT data. As the DMA utilizes AXI Stream interface, the Matched filtering and MUSIC IPs use the same interface for easy communication among the protocols. Internally the Matched filtering consists of the FFT IP and the Matrix Vector IP to create 1024 vectors are the various angles we are interested in finally converting the data to the time domain using an IFFT IP. As the data we are sending is complex in nature for single precision floating point each data point in the matrix would take 64bits, this becomes a challenge when designing the IP, as to not go over the resource limit present on the Xilinx Ultrascale+. Once we have received all of the data from the Matched filtering IP we perform a peak search on the ARM Processor. The peak search consists of taking the first packet finding the index for which the max is present, then over the 99 other packets create 1x100 vector consisting of all points corresponding to the max index. Once we have that vector we send the data using the other DMA to the MUSIC IP where we are able to generate a 1x200 pseudo spectrum for detecting the Doppler Velocity of the target of interest. The data for the Range, Azimuth and Doppler is present on the ARM Processor which we then plot on the PYNQ OS' Jupyter Notebook as well verify the targets location. Then we perform the CLEAN operation, the dummy signal generation is done on PS, the dimension of the dummy signal is same as the original RXMAT. This dummy signal is sent to the Matched filtering IP to generate the corresponding Y array, that we use to create the Residue matrix. The residue matrix present on the PS is used to calculate the new target location. Over next few section I will discuss the Matched Filtering IP in detail, the optimizations made in Vivado HLS, the resulting effect on the latency and utilization.

3.3 Matched Filtering Architecture

This sections illustrates the implementation of the matched filtering implementation in floating point in Vivado HLS. The matched filtering IP consists of FFT & IFFT blocks, the azimuth delay block and the matrix-vector multiplication block. The azimuth delay block contains the delayed matrix, the values were calculated before hand using,

$AzimuthDelay = exp(-1j * 2 * \pi * \sin\{\theta\} * d/\lambda)$

here θ ranges from $-90°$ to $90°$ and d is the antenna spacing. This is generates a 181x32 matrix that is stored in the LUT of the IP.

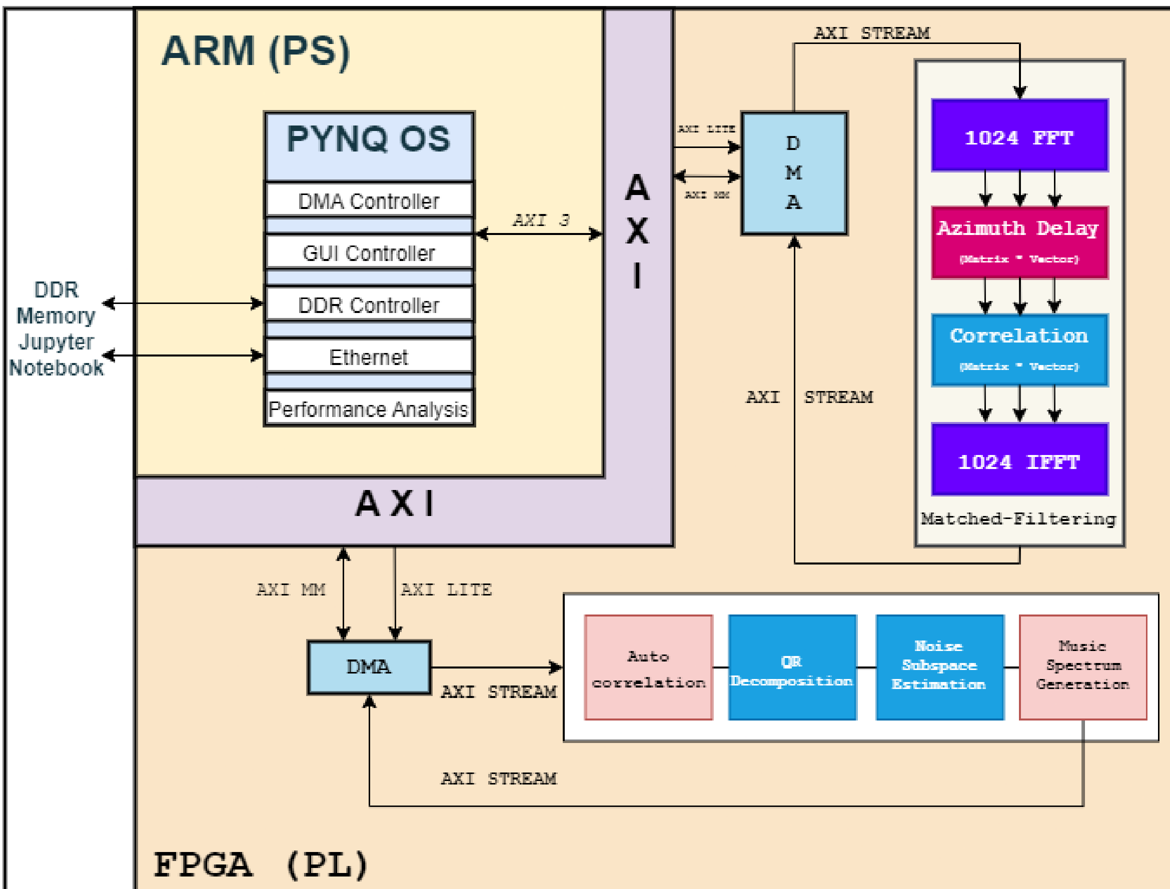

Figure 3.3: Detailed Matched Filtering and MUSIC Architecture

1. Specification in C++: Here we will go over the C++ specification for the different blocks that make up the Matched filtering IP.

   (a) FFT/IFFT: The Xilinx LogiCore IP was chosen as it provided good performance and was highly optimized in terms of resource usage. Below is the implementation in Vivado HLS. The "hls::fft" is used as we are importing the FFT function the HLS header file. Synthesis table 3.1.

```
/*
The main function here is the fft function, the other functions are
    helper functions.
```

```
The fft function uses the hls fft header file. Additional variables such as config_t and status_t are required for the functioning of the fft IP.

@param    'direction' is used to set whether forward or inverse fft is to be done
@param    'in' is the input vector, the input needs to of complex type, the size is dependent on
      the size of the fft we are doing, in this case it is 1024
@param    'out' is the input vector, the input needs to of complex type, the size is dependent on
      the size of the fft we are doing, in this case it is 1024

@return   void
*/

#include "hls_fft.h"

struct config1 : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
    static const unsigned scaling_opt = hls::ip_fft::block_floating_point;
    static const unsigned input_width = 32;
    static const unsigned output_width =32;
    static const unsigned config_width = FFT_CONFIG_WIDTH1;
    static const bool mem_hybrid = true;
    static const unsigned max_nfft = FFT_NFFT_MAX1;
    static const unsigned stages_block_ram = (max_nfft < 10) ? 0 : (max_nfft - 9);
    static const unsigned phase_factor_width=25;
};
// Type definitions for structures present in hls_fft.h
typedef hls::ip_fft::config_t<config1> config_t;
typedef hls::ip_fft::status_t<config1> status_t;

/*
The use of pipeline pragma lets the various iterations run concurrently, this increase the latency tremendously
*/
void dummy_proc_fe_fft(bool direction, config_t* config, cdt in[dim_r], cdt out[dim_r]){
    config->setDir(direction); // setting the direction of transform
    for (int i=0; i< dim_r; i++){
#pragma HLS pipeline
        out[i] = in[i];
    }
}

/*
The use of pipeline pragma lets the various iterations run concurrently, this increase the latency tremendously
*/
void dummy_proc_be_fft(status_t* status_in, cdt in[dim_r], cdt out[dim_r]){
    for (int i=0; i< dim_r; i++){
 #pragma HLS pipeline
        out[i] = in[i];
    }
}

/*
```

```
The IP beforehand is quite optimized, the use of dataflow pragma allows
     for the various function to run
concurrently, in parallel.
*/

void fft(bool direction, cdt in[dim_r], cdt out[dim_r]){
#pragma HLS dataflow
    cdt xn[dim_r];
    cdt xk[dim_r];
    config_t fft_config;
    status_t fft_status;
    dummy_proc_fe_fft(direction, &fft_config, in, xn);
    hls::fft<config1>(xn, xk, &fft_status, &fft_config);
    dummy_proc_be_fft( &fft_status, xk, out);
}
```

Listing 3.1: FFT/IFFT

(b) Azimuth Delay: The matrix for azimuth delay is stored in the LUT of the FPGA, earlier we saw the technique to calculate this matrix. Now we specified in C++ on how to access the individual elements of rxmat in the frequency domain that need to correctly scaled with this new delay factor. From the FFT function we pass the rxmat matrix to the azimuth delay function. Synthesis table 3.2.

```
/*
One of the main blocks of the matched filtering IP. This is used for
    incorporating the azimuth delay in the
RXMAT that is sent from the ARM Processor. The data is complex in
    nature, for that a typedef of cdt is used.
That corresponds to std::complex<float> from the standard library.

dim_c:    32
dim_r:    1024

For each column of rxmat we multiply the corresponding row, with the
    appropriate delay factor.

@param    'rxmat' is the RXMAT matrix sent from the PS in the Frequency
     Domain
@param    'angle' is the angle for which we need to scale the matrix,
    rxmat
@param    'rxmat_dd' is the delayed version of rxmat

@return   void
*/


/*
Here the main optimization is in loop: azi_L1. The use of pipelining
    pragma allows the unrolling of loop azi_L2. Allowing all 32 rows to
     be processed in parallel. This leads to an increase in resource
    such as DSP and BRAMs. As to process those arrays we need to
    provide sufficient ports, that is done through the  array_partition
     pragma
*/

void azi_delay(cdt rxmat[dim_c][dim_r], cdt rxmat_dd[dim_c][dim_r], int
     angle){
  azi_L1:
  for(int i=0; i<dim_r; i++){
#pragma HLS PIPELINE
```

```
    azi_L2:
    for(int j=0; j<dim_c; j++){
      rxmat_dd[j][i] = rxmat[j][i] * cdt(azi_real[angle][j], azi_imag[
    angle][j]);
    }
  }
  return;
}

```

Listing 3.2: Azimuth Delay

(c) Matrix Vector Multiplication: As we know that correlation in the time domain corresponds to multiplication in the frequency domain. Once we have the delayed echo rxmat we need to multiply it with the transmitted signal xmat. In order to efficiently to do this operation, the frequency transformed version of xmat is stored on the board using the LUTs. This saves computation of the FFT of XMAT, we are allowed to do as the RXMAT that we are generating in the PS is based on the XMAT. Hence we can assume that we have access to the XMAT. Synthesis table 3.3.

```
/*
This function performs the matrix vector muliplication between the
    delayed rxmat and
xmat in the frequency domain. The reason we are able to perform a
    multiplication is due to
the fact that accordin the RSP algorithm in MATLAB, once the
rxmat-1024x32 , xmat-32x1024 matrix multiplication is done, this
    creates a 1024x1024 matrix as we know, but from the matrix
we only require the daigonal elements. This is a huge redundancy that
    we can exploint on the FPGA.
By calculating the mutliplications for each element of the final 1024
    row we can reduce the latency
by a huge factor.

Additionally the rows of the matrix xmat are identical, so instead of
    storing the entire 32x1024 matrix
we are store only 1 row and re-use that for the other iterations.

@param    'rxmat_dd' is the delayed version of the rxmat in the
    frequency domain
@param    'out' is the final vector of size 1024 that is sent to the
    IFFT function

@return   void
*/


/*
Here the main optimization is done through the PIPELINE pragma, this
    allows us to perfom all 32 multiplication and addition operations
for a single element of the 1024 long vector at the same time.
*/
template <typename T>
void mv_mult(T rxmat_dd[dim_c][dim_r], T out[dim_r]) {
  L1:
  for(int m = 0; m < dim_r; m++) {
#pragma HLS PIPELINE
    cdt sum = 0;
    L2:
    for(int n = 0; n < dim_c; n++) {
```

```
32          sum+= rxmat_dd[n][m] * xmat_fft[m];
33        }
34        out[m] = sum;
35      }
36      return;
37  }
38
39
```

Listing 3.3: Matrix Vector Multiplication

2. Synthesis Results: Now we shall see how the optimization we made has affected the resource utilization and latency. As we have optimized the various inner for loops the latency should be low overall. We are also trying to meet the 100MHz clock frequency. But due to number of the iteration of each inner loop, the total latency is high.

Clock Period (in ns)	Latency Cycles	BRAM	DSP48	FF	LUT
8.75	3196	19	0	16364	13409

Table 3.1: FFT Synthesis Results

Clock Period (in ns)	Latency Cycles	BRAM	DSP48	FF	LUT
8.49	1033	0	512	41390	31167

Table 3.2: Azimuth Delay Function Synthesis Results

Clock Period (in ns)	Latency Cycles	BRAM	DSP48	FF	LUT
8.419	1193	4	640	64473	45471

Table 3.3: Matrix Vector Multiplication Function Synthesis Results

The reason for the entire matched filtering function to have more utilization than the sum of the individual blocks in table 3.4is that, all the helper functions that were used, some to call the main blocks are the locations where the arrays are declared, hence the utilization for those particular is lower, as only the necessary blocks for those functions are passed through functions parameters.

Clock Period (in ns)	Latency Cycles	BRAM	DSP48	FF	LUT
8.75	828915	1082	1158	139545	111806

Table 3.4: Matched Filtering Synthesis Results

3.4 Optimizing RTL & Analysis

The optimization mentioned in the previous section were necessary in order to meet the target of 100MHz clock, while minimizing the latency.

3.4.1 Optimizing RTL

After writing the baseline functionality version of the C++ code, optimizations to improve performance were mainly done using the following steps:

1. Changing the coding style for the usage of HLS PRAGMAs. There are certain techniques that can be used to code the functionality, in order to achieve better latency or resource utilization.
2. By applying appropriate PRAGMAs to reduce latency in return of using more area. This allows multiple parallel blocks to perform its computation at the same time.
3. Through HLS Analysis tool redundancies can be found as for the case of storing XMAT, and the RXMAT * XMAT multiplication, all of these redundancies can be exploited to improve performance of the hardware IP.

3.4.2 Analysis of Pragma

Various PRAGMAs used had different effects on the generated synthesis reports. These are just some of the conclusion from the usage of the PRAGMAs

- PIPELINE
  1. Pipelining improved overall throughput of the design
  2. Pipelining a loop automatically unrolls all the inner loops. If the inner loops involves a lot of computation, unrolling can lead to a significant increase in re-sources.
  3. A region can be pipelined iff its Initiation Interval (II) is less than 256. II is a parameter associated with pipelining which is the number of clock cycles after which the next iteration of the loop can begin.
  4. There are certain cases where PIPELINING should be avoided, when a certain operation takes a single clock cycles then PIPELINING only increases the utilization without any performance benefit, hence it should be avoided in such cases.
- INLINE
  1. It reduces the latency by the various functions in the top function itself
  2. No impact on the resource utilization
- UNROLL

1. Hardware resources increase drastically, as unrolling creates multiple copies of the hardware in order for parallel execution to take place.
2. Latency reduces as the parallel operations are being performed. The number of clock cycles are now reduced to perform the same task.

- DATFLOW

1. At the cost of hardware resources to buffer data, pipelining is done at the function level, thus reducing the number clock cycles in turn reducing the latency.
2. There are some conditions to be met for the data-flow pragma to be used, such as the function where this is applied needs to be of the type, Single Producer-Single Consumer.

Chapter 4

Word Length Analysis

The fixed point implementation has an advantage over the floating point implementation as it requires less hardware and provides improved throughput at the cost of loss in accuracy. This chapter covers the analysis of performance, accuracy, and resource utilization for various word lengths. The fixed point implementation allows the independent handling of fractional and integral arithmetic. The inbuilt ap_fixed data type from library "ap_fixed.h" is used to analyze varying word length in HLS. In ap_fixed data type, a fixed-point value is represented as a sequence of bits with a specified position for the binary point. This data type in HLS is represented as follows

ap_fixed <int W, int I>

- W is the total number of bits used to represent the number
- I is the number of bits used to store the integral part. It determines the position of the binary point in fixed point implementation.
- I bits to the left of the binary point represent the Integer part of the number, W-I bits to the right of the binary point represent the fractional part of the value.

4.1 Approach

For our use case, the absolute error of the output of the matched filtering block may be different than what is expected, but the co-ordinate of the targets are still being detected correctly. Hence in the following graph I have shown the error that was presented when different word lengths were used, only some of them failed to detect the target. While at other word lengths it was detected correctly. In Table 4.1 we can make conclusions on what Word lengths need to be used.

From here we can see that for cases where the azimuth error is greater 0, those wordlengths would not be useful to conduct experiments. Also it is worth noting that RMSE is an important metric but when the range and azimuth error is low for a high RMSE it can still be used for experiments. In the figure 4.1, the x-axis contains the combination of the total width of the matched filtering & FFT IP. The number of integer bits for FFT was set by Xilinx to 1 hence we had to scale our input and output data if more integer were needed. As for the matched filtering block, at each stage of the IP, the maximum and minimum values were calculated and based on that integer bits were chosen.

Wordlength MF	Wordlength FFT	RMSE	range error	azimuth error
16	16	0.0451411	0.028408	5
28	16	0.045137	0.028408	1
32	16	0.0451369	0.028408	1
24	16	0.0451368	0.028408	1
20	16	0.0451365	0.028408	1
16	24	0.00171987	0.028408	0
16	32	0.00169496	0.028408	0
20	24	0.00032408	0.028408	0
24	24	0.00027583	0.028408	0
28	24	0.00027573	0.028408	0
32	24	0.000275723	0.028408	0
20	32	0.000156042	0.028408	0
24	32	9.83E-06	0.028408	0
28	32	1.27E-06	0.028408	0
32	32	1.09E-06	0.028408	0

Table 4.1: Wordlength Selection Table

4.2 Resource Utilization

The word-lengths chosen affect the resource utilization vastly, fixed point implementation use fewer resources than floating point implementation, at a cost of accuracy. There is a trend that can be noticed from figure 4.2

- BRAM Utilization increases with increasing word length as the no. of bits to be stored increases.
- The DSP48 units required for the multiplication and accumulation operations decreases with decreasing word length requirement.
- The no. of FF and LUT decrease with decrease of word length as lower number of bits are needed.
- The execution time for the system also reduces when choosing fixed point word lengths
- Power consumption also reduces when we reduce the word length.

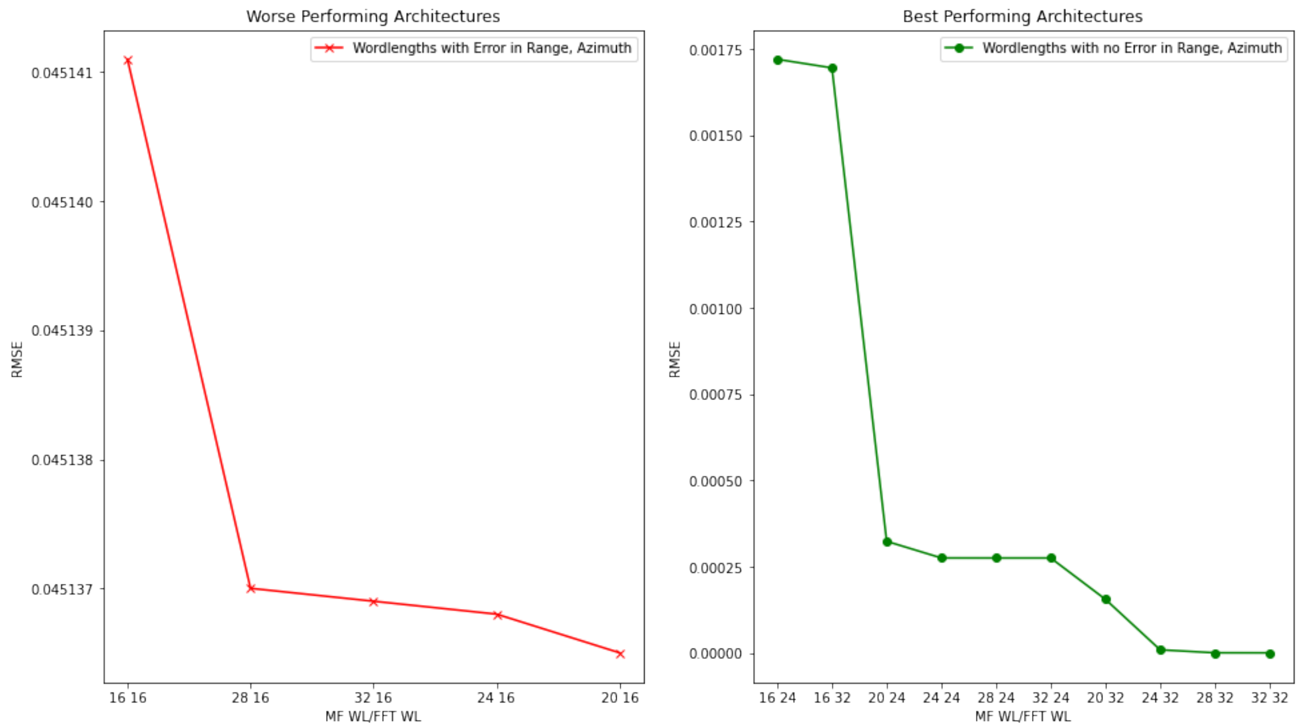

Figure 4.1: RMSE v Wordlength

WL - MF	WL - FFT	WL - MUSIC	System Execution Time (PYNQ) (seconds)	BRAM {Utilization, %}	DSP {Utilization, %}	FF {Utilization, %}	LUT {Utilization, %}	Total & Dynamic Power(W)
FP	FP	FP	15.775727	{1073, 99.35%}	{1655, 38.74%}	{144001, 16.93%}	{151712, 35.67%}	{7.842, 6.595}
32	32	32	13.537432	{818, 75.74%}	{1163, 27.22%}	{41001, 4.82%}	{63382, 14.9%}	{4.888, 3.677}
24	24	24	13.760167	{638, 59.07%}	{600, 14.04}	{35078, 4.24%}	{44152, 10.38%}	{4.454, 3.249}
16	16	24	13.542408	{493, 45.65%}	{340, 7.96%}	{31806, 3.74%}	{34047, 8.01%}	{4.299, 3.097}

Figure 4.2: Word-Length v Resource Utilization

4.3 Functionality Results

For the word lengths chosen from the figure 4.2 we need to verify the RMSE for detecting 1, 2, 3 Target. We can do this by running these word length architectures for multiple iterations for a range of SNR, which is -25dB to +10dB in this case. From the following figures, 4.3, 4.4, 4.5 we can see that the floating point IP performed best, then the fixed point architectures, which is expected. We can also see the effect of noise on the detection of the targets.

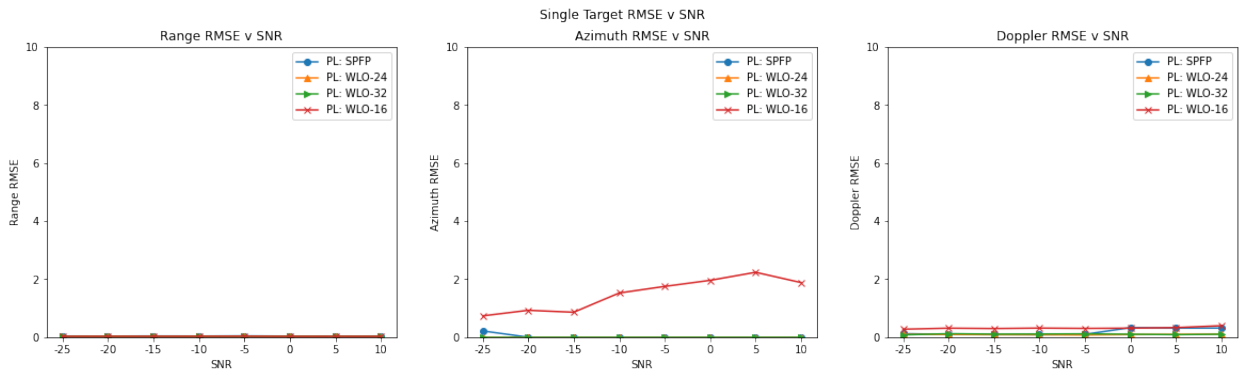

Figure 4.3: Single Target RMSE

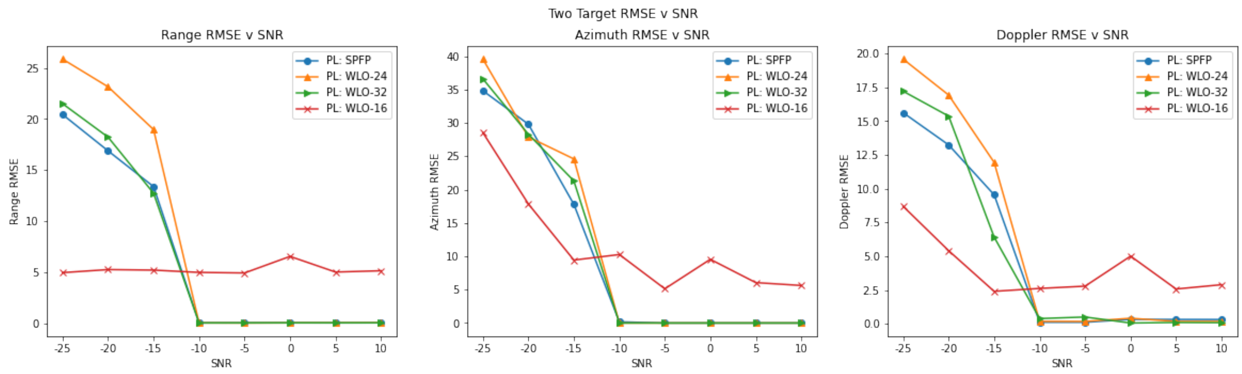

Figure 4.4: Two Target RMSE

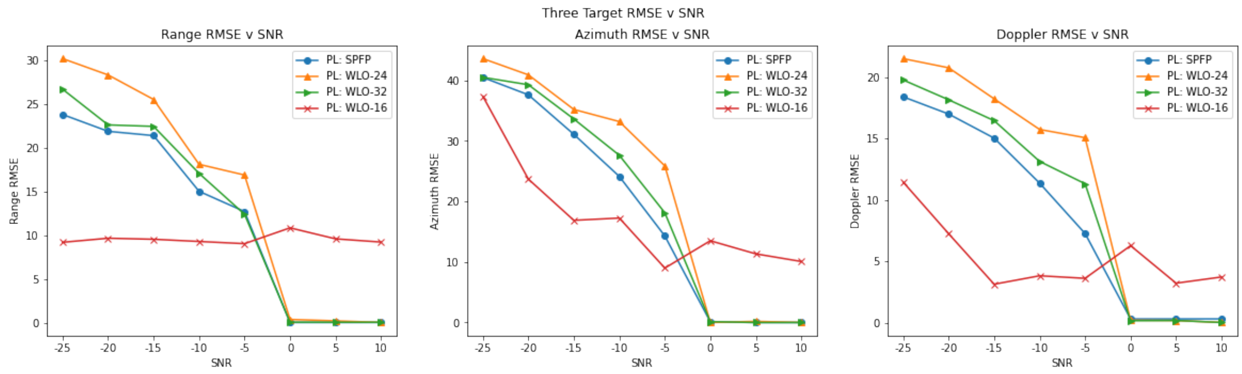

Figure 4.5: Three Target RMSE

Chapter 5

Hardware Software Co-Design

Hardware Software Co-designing(HSCD) is the partitioning of the algorithm such that some part is implemented in the hardware while some part in the software. The technique exploits the trade-offs between hardware and software for any design implementation. It allows the design to be implemented efficiently in terms of performance, power and area. It supports the growing complexity of embedded systems. In this chapter, we have analysed the various HSCD configurations for Matched filtering, FFT & MUSIC, the objective here is to compare the execution time of different combinations and effect on resource utilization and power.

5.1 Approach

Certain parts of the complete algorithm were mapped to differently to PS & PL. Certain parts of the algorithm were put on the FPGA and other parts ran on the PS. The floating point IP was used to test the different configurations. As we used two DMAs, one for Matched filter and one for MUSIC, some configuration get rid of the redundant DMA. The Full-Power High Performance port on the MPSoC was used to send data from the DDR memory to the hardware accelerators. The AXI DMA converts the Memory Mapped(MM) protocol to stream via the MM2S ports. The input stream from DMA is sent to the HLS IP and the output stream is converted to MM by DMA and stored in the PS memory via the FPHP port.

5.2 Results

In figure 5.1, we can see the variation in resources and execution time for detecting 1 and 3 targets. As shift more blocks to the PL the execution time decreases, with an increase in power consumption. For single target results there is a 13,413% improvement in terms of performance of the complete PL based system as compared to the PS based system. While for three target detection there is a 9,892.2% improvement of PL system over PS system.

PL	PS	Three Target Execution Time (PYNQ) (seconds)	Single Target Execution Time (PYNQ) (seconds)	Resource Utilization {BRAM, DSP, FF, LUT}	Power {Total, Dynamic} (W)
MF, FFT, MUSIC	-	15.775727	3.826025	{1073, 1655, 144001, 151712}	{7.842, 6.595}
MF, FFT	MUSIC	20.428827	5.353565	{685, 732, 80849, 60067}	{5.166, 3.953}
MUSIC	MF, FFT	1551.183378	516.49506	{262.5, 350, 28923, 40800}	{4.435, 3.234}
-	MF, FFT, MUSIC	1560.71118	517.01337	-	-

Figure 5.1: Hardware Software Co-Design Results

Chapter 6

Conclusion & Future Works

6.1 Conclusion

The motivation for the project was clearly understood. The steps that were taken to achieve the objects of the project were also put into motion. The tool, Vivado HLS was studied in detail so that usage later on would not be an issue. Understanding and trying to change the C++ coding style was one of the more trickier challenges to tackle. Understanding the algorithm, and finding scope for parallelism was developed, and an overall approach to tackling the problem of mapping algorithms to various efficient architectures was developed over the period of the project. Hardware Software Co-Design was also done so that we can see the impact of different IP on the resource utilization and latency. The impact of different word lengths on the resource utilization and the accuracy of the results was also demonstrated. Given the set of resources present on the Xilinx Ultrascale+ the maximum possible performance was exploited. Variation in different SNR scenarios was also shown. All the objectives that were set in the previous semester have been met, and the results are as per expectation.

6.2 Future Works

Some form of Reconfigurability needs to included so that we can have on the fly switching of architectures for different situations and can achieve different levels of performance. Newer version of the current architecture has developed by us at the algorithm level, and implementing that on the FPGA is really interesting. A lot of redundancies have been removed thus making the system even more performance efficient while lowering resource utilization. Currently we are in the process of writing a Journal Paper for the work done till now.