10-5-2017

High Level Synthesis and Evaluation of an Automotive RADAR Signal Processing algorithm for FPGAs

Siddhant Luthra
University of Windsor

High Level Synthesis and Evaluation of an Automotive RADAR Signal Processing Algorithm for FPGAs

by

Siddhant Luthra

A Thesis

Submitted to the Faculty of Graduate Studies

through the Department of Electrical and Computer Engineering

in Partial Fulfillment of the Requirements for

the Degree of Master of Applied Science

at the University of Windsor

Windsor, Ontario, Canada

2017

High Level Synthesis and Evaluation of RADAR Signal Processing algorithm for FPGAs

by

Siddhant Luthra

APPROVED BY:

T. Bolisetti

Department of Civil and Environmental Engineering

E, Abdel-Raheem

Department of Electrical and Computer Engineering

M. Khalid, Advisor

Department of Electrical and Computer Engineering

August 3rd, 2017

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone’s copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office and that this thesis has not been submitted for a higher degree of any other University or Institution.

Abstract

High Level Synthesis (HLS) is a technology used to design and develop hardware (HW) using high-level languages such as C/C++. An HLS model of an automotive RADAR signal processing algorithm has been developed for the purpose of comparison between the HLS model and the existing HDL model. Register Transfer Level (RTL) programming is a technology used to design and develop hardware at the register transfer level (or low level) using Hardware description languages such as Verilog and VHDL. FPGA development usually requires the knowledge of RTL technologies. HLS gives software (SW) developers the ability to design and implement their designs on an FPGA without requiring the knowledge of RTL technologies and HDL.

Even though HLS is currently gaining popularity, the applications used to evaluate HLS tend to remain small. We synthesize an automotive RADAR signal processing system using HLS-based design methodology, which has mid to high complexity, and compare our synthesis results to that of the RTL-based design. We used many techniques used to make the high-level program model ready for synthesis while optimizing for both speed and resource usage using Xilinx Vivado HLS Computer-Aided Design (CAD) tool. We achieved a speed up of 2X compared to the RTL-based design while reducing the design time from approximately 16 weeks to 6 weeks. The FPGA resource utilization increased but it was still under 5% of the total resources available on the FPGA.

Acknowledgements

With utmost sincerity, I express my gratitude and respect to my advisor Dr. M. Khalid, who gave me the wonderful opportunity to work under his supervision and inspired me to work with honesty, integrity and discipline.

I would also like to thank my committee members Dr. T. Bolisetti and Dr. E. Abdel-Raheem, who provided me with insightful suggestions to improve my research.

I would like to dedicate my work to my parents, as their ever-encouraging faith has kept me going and gave me the strength to overcome any obstacles that have come my way.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
HLL	High Level Language
HW	Hardware
SW	Software
RTL	Register Transfer Level
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
RADAR	Radio Detection And Radio Ranging
CAD	Computer-Aided Design
IC	Integrated Circuit
OTP	One-Time Programmable
ASIC	Application-Specific Integrated Circuit
SSI	Stack Silicon Interconnect
CPU	Central Processing Unit
GPU	Graphics Processing Unit
DSP	Digital Signal Processing
IP	Intellectual Property
FFT	Fast Fourier Transform
I/O	Input/Output
SoC	System On Chip
FPS	Frames Per Second
FM-CW	Frequency Modulated – Continuous Wave
LFM-CW	Low Frequency Modulated – Continuous Wave
MEMS	Microelectromechanical System
RF	Radio Frequency
MMIC	Monolithic Microwave Integrated Circuit
VCO	Voltage Controlled Oscillator
TLC	Top-Level Control
RPU	Radar Processing Unit
SP3T	Single Pole Three Throw
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
CFAR	Constant False Alarm Rate
CA-CFAR	Cell-Averaging Constant False Alarm Rate
SNR	Signal to Noise Ratio
RAM	Random Access Memory

ROM	Read Only Memory
FIFO	First-In First-Out
R&D	Research and Development
LUT	Look Up Table
FF	Flip-Flop

Chapter 1. Introduction

1.1. Motivation

High Level Synthesis is gaining popularity as a primary design methodology when it comes to hardware design due to a number of advantages such as higher productivity, faster time to market, etc. Even though HLS provides many advantages over the current RTL-based design methodologies, a lot of work remains to be done in effectively utilizing HLS Computer-aided Design (CAD) tools for hardware design targeting complex real world applications.

There have been a few case studies in the past which focus on the High-Level Synthesis of various hardware designs [1, 8] but they generally focus on the design itself and not the comparative evaluation of HLS results with the older RTL-based design methodologies. A case study which focuses on the comparison and evaluation of HLS and HDL synthesis results is presented in [2]. The H.264 video decoding algorithm was synthesized using Vivado HLS CAD tool [3]. Some previous case studies which focus on the HLS technology itself have shown promise that HLS is ready for mainstream implementation [10]. A detailed overview of currently available HLS CAD tools from the industry and academia is presented in [11, 13]

The main motivation for this thesis is to compare HLS and HDL-based design methodologies for designing a mid to high complexity design which is the automotive RADAR signal processing algorithm to detect a target's range and velocity with respect to the unit the system resides in [4, 5].

1.2. Objective

The main goal of this thesis is to evaluate and compare HLS and HDL-based design methodologies by simulating and synthesizing an automotive RADAR signal processing algorithm for Xilinx Virtex 7 FPGA. The metrics used for the evaluation and comparison are time-to-market, speed (timing analysis), and area (resource utilization). The CAD tool used for this purpose is the Xilinx Vivado HLS. The HDL model used in our comparison is described in [4, 5]. For fair comparative analysis, the existing HDL model was updated to target the Xilinx Virtex 7 FPGA using Xilinx Vivado since the original design was for the Xilinx Virtex 5.

A few other questions which will be answered in this thesis are:

- Is HLS ready for complex real world applications such as the automotive RADAR signal processing algorithm?
- What are the advantages of HLS over HDL-based synthesis?

1.3. Thesis Outline

This remainder of this thesis is organized as follows:

In Chapter 2, the background of FPGAs, hardware design methodologies (HDL and HLS), tools for hardware design, a brief comparison of the technologies, and related research are briefly discussed. Chapter 3 describes the automotive RADAR signal processing algorithm to be synthesized using HLS. This includes a brief introduction to target detection, the actual algorithm model, and the HDL modeling of this algorithm.

In Chapter 4, implementations of the algorithm at different levels of abstraction, namely, MATLAB, C++, and Verilog are discussed. Chapter 5 discusses the results obtained from various implementations which are then used for the evaluation and comparison of the HLS and the HDL-based design methodologies. We finally conclude in Chapter 6 with a summary and suggestions for future work.

Chapter 2. Background

This chapter provides background information on FPGAs and HLS. Previous related works are also discussed.

2.1. FPGAs

Field Programmable Gate Arrays (FPGAs) are Integrated Circuits (ICs) which consists of large amounts of programmable logic resources, programmable routing and embedded resources such as memories and DSP blocks. FPGAs enable rapid and custom hardware design for many real-world applications. They are also used for prototyping various hardware designs before they are sent for IC fabrication. FPGAs enables us to reprogram the hardware architecture to mimic a custom IC which allows us to do thorough testing before the design is sent for fabrication. Although one-time programmable (OTP) FPGAs are available which perform specific tasks, reprogrammable FPGAs are more popular because they can handle design changes as the design evolves.

Application specific integrated circuits are custom hardware ICs which are manufactured for specific applications. Although ASICs are faster and consume less power, the recent advances in FPGAs push the limits of speed, complexity, and physical size in comparison to older FPGAs.

The FPGA used for this research is the Xilinx Virtex 7, which is one of the four Xilinx 7 series FPGA families. These FPGAs are optimized for highest system performance and capacity with a 2X improvement in system performance, these are the highest capability devices enabled by stack silicon interconnect (SSI) technology [12]. An overview comparison between the four different families of Xilinx 7-series FPGAs is shown in Table 1.

Max. Capability	Spartan-7	Artix-7	Kintex-7	Virtex-7
Logic Cells	102K	215K	478K	1,955K
Block RAM	4.2 Mb	13 Mb	34 Mb	68 Mb
DSP Slices	160	740	1,920	3,600
DSP Performance	176 GMAC/s	929 GMAC/s	2,845 GMAC/s	5,335 GMAC/s
Transceivers	–	16	32	96
Transceiver Speed	–	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Serial Bandwidth	–	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	–	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	800 Mb/s	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	400	500	500	1,200
I/O Voltage	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V
Package Options	Low-Cost, Wire-Bond	Low-Cost, Wire-Bond, Lidless Flip-Chip	Lidless Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip

Table 1 Xilinx 7-series FPGA family comparison [12]

2.2. Hardware Design Methodologies

Hardware design can be done either for a specific task (Single purpose) or to execute different multiple tasks (General purpose). General purpose hardware (or IC) is designed to be able to execute multiple different tasks, the best example would be a CPU in a personal computer. Single purpose hardware are ICs which are designed only to perform one task and do it well, a common example for this would an IC for encryption and decryption. For our research, we will focus on single purpose HW.

There are a few ways to design hardware, it all depends on what the intention of the design is. For this thesis, we are more concerned about FPGA prototyping which means that we would program an FPGA to perform certain tasks. Traditionally, a Hardware description language (HDL) is used to program the FPGA at the RTL level, two most popular examples of HDLs are Verilog and VHDL. A relatively new technology to program FPGAs which allows us to do so using High Level Languages (HLLs). This allows rapid hardware design targeting ASICs and FPGAs.

2.2.1. Hardware Design using HDLs

As mentioned earlier one of the traditional and most popularly used technology to program FPGAs is using HDLs. They allow the designers to program at the RTL level. VHDL and Verilog, being the two most popular HDLs, had a huge impact on hardware development when they were created.

The main feature of HDLs is the ability to model concurrent operations which is important to reduce the run time of a specific task. It can provide faster speed but is hard to code and can have stalling issues due to dependencies.

An example of concurrent operations is discussed here:

```
Assumptions:
We have 6 variables: A, B, C, D, E, F, and O.

Lets assume B = 3, C = 4, E = 5, and F = 6.

*/
Line 1: A = B + C
Line 2: D = E + F
Line 3: O = A + D
```

Figure 1 Concurrent Operations Example pseudo-code

For the example in Figure 1, let’s assume that each addition takes 1 clock cycle and assigning operations have no delays. If this code is to be run sequentially, Line 1, Line 2, and Line 3 will take 1 clock cycle each since they have one addition (+) each which means the whole process will take 3 clock cycles. Now, to run this code in parallel, we come across with an issue of dependency on Line 3. What this means is to run Line 3, we need the values of variables “A” and “D”. Therefore, there is a limit on how the concurrent statements run. In this case, Line 1 and Line 2 can be run in parallel since both the lines have no dependencies. To sum it all up, executing Line 1 and Line 2 will take only 1 clock cycle, and executing Line 3 will take another 1 clock cycle, so it will take 2 clock cycles instead of 3 when compared to the sequential (or non-concurrent) execution. Even though there isn't much difference in the time consumed, this gap increases as the designs get bigger and more complex. Concurrent programming is one of the most important techniques a hardware designer requires but can get very complicated while using HDLs.

Another significant feature RTL based designs provide us is access to arbitrary precision data types, what this implies is every input, output, internal registers, etc. can have the desired number of bits. In HLLs, however, we usually only have access to data types bound by 8-bit boundaries. The arbitrary precision data types we have access to has an effect on the size, speed, precision, flexibility and power consumption of the final design.

There are quite a few tools available for FPGA programming using HDLs, the one we will be focusing on is Xilinx Vivado.

Currently, Xilinx Vivado is part of the Vivado Design Suite, created by Xilinx. Since the FPGA we are using in our thesis is made by Xilinx. This would be the best tool for us to use. All the Xilinx Vivado Design Suite tools are written with a native tool command language (Tcl) which offers us access to all the tools via command line which is available in the GUI. Xilinx Vivado Design suite currently supports the Xilinx® UltraScale™ and 7 series devices, Zynq® UltraScale+™ MPSoC device, and Zynq®-7000 All Programmable (AP) SoC. [16]

As discussed earlier, the FPGA in question for our thesis is from the Xilinx 7 series called the Virtex 7. Xilinx Vivado is a very broad tool which allows us to synthesize, implement, simulate and analyze our design. It supports both VHDL and Verilog but we will be focusing on Verilog since the existing HDL model of the automotive RADAR signal processing algorithm is written in Verilog.

2.2.2. Hardware Design using HLS

Even though the technologies mentioned in the previous section are very advanced, recent increases in logic capacity of FPGAs and other similar hardware are making these technologies somewhat tedious and time consuming. This would mean a very long time-to-market along with large code size which further implies that the code is more complex and not particularly readable. Eventually, it comes down to the cost of the product in development. Higher time-to-market would mean the development cost will be higher. To overcome the ever-increasing cost and delays in product release, we look for alternatives to these technologies.

An alternative which we will be considering is High Level Synthesis (HLS). HLS allows designers to program hardware using HLLs (C/C++/SystemC) which are usually used for software development. HLS tools, in essence, synthesize the HLL code into optimized RTL level models but there are substantial differences between the models created by the code written in HLL and HDL. As seen from a broader perspective, there are two main types of HLS technologies, one allows us to design the entire hardware in HLLs, while the other is used to accelerate certain parts of a software using hardware also known as heterogeneous computing. For example, the automotive RADAR signal processing algorithm is an independent hardware which detects target range and velocity.

A basic example of heterogeneous computing would be: a GPU is used to accelerate 3-D rendering of graphics for a program running on the CPU. There are various tools available for both types of applications. A few popular ones are shown in Table 2.

Company	Tool
Xilinx	Xilinx Vivado HLS
Intel	Intel FPGA SDK for OpenCL
Mentor Graphics	DK Design Suite
Synopsys	Synphony C
Calypto Design Systems (Mentor Graphics)	Catapult C
Cadence	C to Silicon

Table 2 Popular HLS tools

The HLS tool we will be using is Xilinx Vivado HLS which is a part of the Vivado Design Suite by Xilinx. This tool essentially synthesizes the code written in C, C++ or SystemC and produces an optimized RTL level model. One important feature of this CAD tool is the ability to use different C/C++ compilers like GCC to compile the HLL code and then convert it into the RTL model, this includes the verification for the design using test benches created in C/C++ or SystemC. The testbench created in HLLs are highly productive in the sense they require very little time when compared to test benches created in Verilog, VHDL or SystemVerilog. Since we are using the Xilinx Virtex 7 FPGA board for our thesis. This tool provides us with all the necessary sub-tools required to successfully program the automotive RADAR signal processing algorithm for the Xilinx Virtex 7.

Some other features of this tool include the ability to create Intellectual Property (IP) cores which means we can import sub-designs from other models into our current design easily. Xilinx provides us with some IP cores like Fast Fourier Transform (FFT), Finite Impulse Response (FIR) filters etc. These become very important since programming these models take time and are usually standard therefore they do not need to be programmed manually. With these IP cores, Xilinx also provides us with a variety of options in these cores, since not all designs are going to use the same type of cores. A simple example is the FFT size, it can be adjusted by setting up the FFT size parameter to suit the design.

The way Xilinx Vivado HLS works is that it synthesizes the HLL code into an optimized RTL level model. An overview of Vivado HLS design flow is shown in Figure 2.

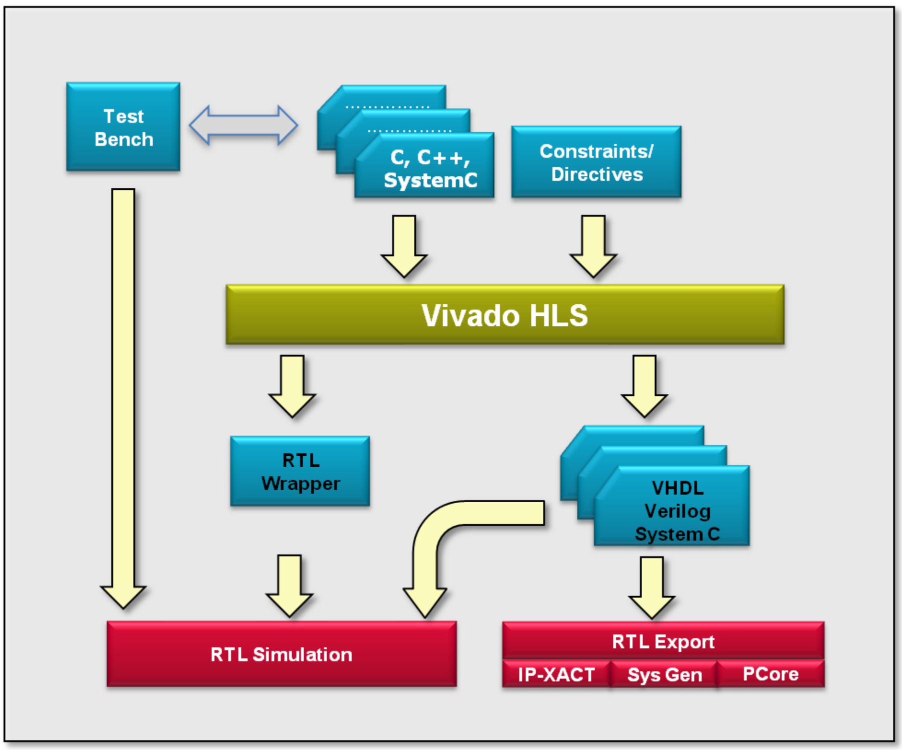

Figure 2 Overview of Xilinx Vivado HLS Design Flow

2.2.2.1. High Level Synthesis Design Methodology

The first step in HLS design methodology is to overcome the limitations HLLs have when it comes to hardware design. Some limitations HLLs have over HDLs are mentioned below:

- In standard HLLs like C/C++, the data types are usually bound by 8-bit boundaries (8, 16, 32, and 64 bits). Whereas HDLs supports data types with arbitrary bit-lengths [17, 18, 19].
- One of the main features of HDLs is concurrent (or parallel) programming. Since HLLs like C/C++ revolve around the concept of sequential programming, special tools are required to program concurrent functions (or modules).
- Standard HLLs do not have the ability for memory management for the entire hardware.

Most HLS CAD tools, including Xilinx Vivado HLS, have pre-built libraries and inbuilt features to overcome most of the limitations HLLs have when it comes to programming for hardware design. Some of the features which are part of the Xilinx Vivado HLS are [13, 16]:

- Vivado HLS automatically generates the Input/output (I/O) interfaces for the design with memories or other communication interfaces.
- It also allocates the necessary registers, memory access, scheduling of operations and binding these operations to the respective functional units.
- Vivado HLS uses pragmas to promote further optimizations using various techniques like flow optimization, loop pipelining, array partition etc., the pragma class supported by the tool can be seen in Table 3 [2].
- Vivado HLS provides us with pre-built libraries which include arbitrary precision (for integers) and arbitrary fixed point precision (for fractional numbers) data types for C/C++. This allows variables of arbitrary widths ranging from 1-bit to 1024-bits to be programmed and used in the design [17, 18, 19].

Pragma Class	Operation
Interface	Define function interface
Function Call	Function Inlining/off Flow Optimization Separate instantiation of functions
Loop Optimization	Loop pipelining Loop unrolling Loop merge operations
Memory Control	Array partition, etc.

Table 3 Pragma Class supported by Vivado HLS [2]

The features mentioned above and the automatically applied optimizations by Vivado HLS are the key design optimization techniques for HLS. Many hardware designers are moving towards HLS with C/C++ as a primary design language since C-level design and verification is relatively easy to use, the time-to-market is significantly lower and the final design is more optimized. The automotive RADAR signal processing algorithm is synthesized to work as a stand-alone hardware due to the nature of the design. Although this design is somewhat complex, we will be using top-down approach since all the functions/modules in our design will be tailored to output the desired results efficiently. We present details of our HLS based methodology to synthesize the automotive RADAR signal processing system in Chapter 4.

2.3. Related Research

There have been a few evaluations of HLS or HLS based CAD tools [2, 10, 11, 13, 16]. Most of them evaluate the tool itself using either a particular benchmark [11], or they conduct surveys and collect information on how HLS has been implemented. However, these evaluation benchmarks tend to stay small in terms of size and complexity. Since we are focusing on complex applications, we will briefly summarize HLS based design of an H.264 Decoder [2].

2.3.1. High Level Synthesis of an H.264 Decoder

Many major platforms like YouTube use H.264 as a video coding standard, due to this the H.264 is present in most of the common embedded SoCs such as Apple's mobile processors, and the popular Qualcomm Snapdragon processors. H.264 being a very complex application, demands HLS-based design so that the designers can achieve accurate results without the tedious and time consuming nature of HDL-based designs.

An H.264 decoder has been synthesized using HLS techniques in [2]. The desired design was required to achieve a throughput of 542 frames per second (fps) at 176x144 resolution and 34 fps at 640x480 (480p) resolution. The results obtained [2] using HLS satisfy the targeted throughput, which shows that HLS-based design methodologies are effective even with complex applications like video decoding. Due to the rapid advancements in HLS-based CAD tools, HLS-based design methodology is increasingly becoming popular and may become a standard for hardware design in future.

A top-down approach for the open-source C-reference model for the H.264 decoder was used. Code restructuring and performance optimizations were performed on the C-reference model in [2], ensuring efficiency while obtaining the desired results. Since the desired throughput was successfully achieved in [2], we used similar optimization techniques to get the desired result for the automotive RADAR Signal Processing System.

Chapter 3. Automotive RADAR Signal Processing

3.1. Target Detection using RADAR systems

Detecting targets for various scenarios in vehicles plays an important role in collision avoidance for automobiles. Although Collison avoidance in vehicles is one of the many applications for target detection, it is an important one since on-road safety has always been a high priority. All the global auto industries are extensively pursuing RADAR based target detection for various purposes like collision warning, automatic braking, blind spot monitoring, parking aid, adaptive cruise control, lane change assistance, and rear crash Collison warning and avoidance, etc. Target detection using various methods like RADAR are extensively being researched while designing autonomous vehicles.

Earlier, a high-power Pulsed Doppler RADAR technique was relied upon for target detection, although this technique was criticized due to the failure of the Mercedes-Benz pulsed RADAR assisted Distronic cruise control system [5]. Therefore, new techniques like the Frequency Modulated-Continuous Wave (FM-CW) RADAR was introduced. Automotive RADAR systems have proved very reliable in recent years in reducing the number of fatal accidents. Initially, these systems were expensive to implement and were only available in high-end luxury cars. Due to the advances in hardware technologies, the costs of implementing these systems in the automotive industry have been reduced significantly. This enables lower-end vehicles to be equipped with the collision avoidance systems as well.

3.2. An Automotive RADAR Signal Processing system

The automotive RADAR signal processing system we will be focusing on is presented in [5]. The system is based on the Long Range Automotive system developed at the University of Windsor. It measures the target range and velocity based on the Linear FM-CW (LFM-CW) approach using a Microelectromechanical system (MEMS) Rotman Lens, MEMS Radio Frequency (RF) switches and phased array antennae for transmission and reception of the signal which the algorithm will process to get the desired output. Table 4 provides the initial system specifications of the automotive RADAR signal processing system [5].

Parameter	Value
RADAR Type	LFM-CW
Operating Frequency	77 GHz
Voltage Controlled Oscillator (VCO)	TLC77xs*
Target Model(s) considered	Swerling I, III, and V type targets
Beamformer	Rotman Lens
Number of Beams	3
Processing duration per beam	2 ms
Beam Width	±4.5°
Antenna Type	Phased Array Antenna
RADAR Processing unit (RPU) platform	FPGA

* 76.5 GHz Monolithic Microwave Integrated Circuit (MMIC) VCO by TLC Precision Wafer Technology

Table 4 Initial System Specifications [5]

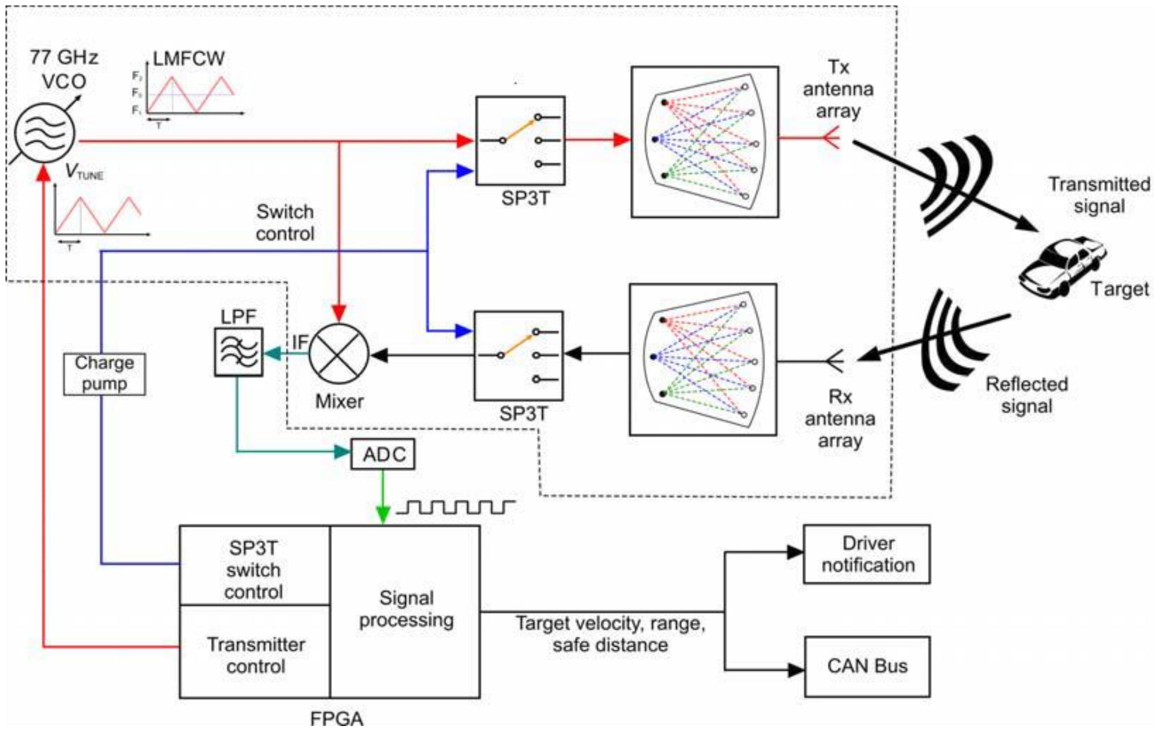

Figure 3 Conceptual Diagram of the RADAR system [5]

A conceptual diagram of the entire RADAR system, showing the major components can be seen in Figure 3. In this design, we will be focusing on the RADAR processing unit which is the FPGA. As we can see, the RPU or the FPGA has 3 main outputs and 1 main input.

The three main digital outputs are:

1. Output to the 77 GHz VCO.

2. Output to the Single Pole 3 Throw (SP3T) switch.

3. Output representing the target velocity and range.

The only main input is the time-domain sample received from the Analog-to-Digital Converter (ADC).

The RADAR Signal Processing algorithm [5] which is to be synthesized can be divided into two parts:

1) RADAR Transmitter control and SP3T switch control.

2) RADAR Receiver Flow Control and Signal Processing.

3.2.1. RADAR Transmitter and SP3T switch Control

This section of the algorithm provides the necessary outputs to the VCO and the SP3T switch. The VCO controls the signal to be transmitted by the RADAR system, and the SP3T switch is responsible for switching between the MEMS Rotman lens beam ports (Beam port 1, Beam port 2, Beam port 3).

This part of the algorithm is responsible for the following:

- Generation of the RADAR frequency chirp by tuning the VCO with a voltage sweep through a Digital-to-Analog Converter (DAC).
- The synchronization of the chirp generation with the signal processing done after receiving the appropriate signal.
- Keeping track of when every down sweep ends so the appropriate output can be sent to the SP3T switch control to switch to the next beam port changing the beam direction.
- Modifying the output to the SP3T to switch between the MEMS Rotman lens beam ports.
    - Beam port 1 to Beam port 2
    - Beam port 2 to Beam port 3
    - Beam port 3 back to Beam port 1

The flowchart for this part of the algorithm can be seen in Figure 4 [5]. Some key information regarding this part of the algorithm is listed as follows [5]:

- The sensor begins with beam port 1 of the Rotman Lens, and after system reset.
- The system starts with the up sweep or a positive frequency chirp.
- Based on the market availability of fast DACs, a 10-bit DAC with a 900 nanoseconds refresh period should be suitable for the target sweep duration of 1 millisecond.
- The DAC is configured to output voltage range from 4.5 V to 6.1 V based on the 10-bit modulating output to the DAC from the FPGA which ranges from 0 to 1023.
- A clock signal is also sent to the DAC for the DAC clock.
- A sampling clock for the ADC will be sent to the ADC.
- The output to the 3-pin MEMS RF switch control will be a 3-bit output from the FPGA:
    - $(100)_2$ (Decimal equivalent of 4): For beam port 1
    - $(010)_2$ (Decimal equivalent of 2): For beam port 2
    - $(001)_2$ (Decimal equivalent of 1): For beam port 3

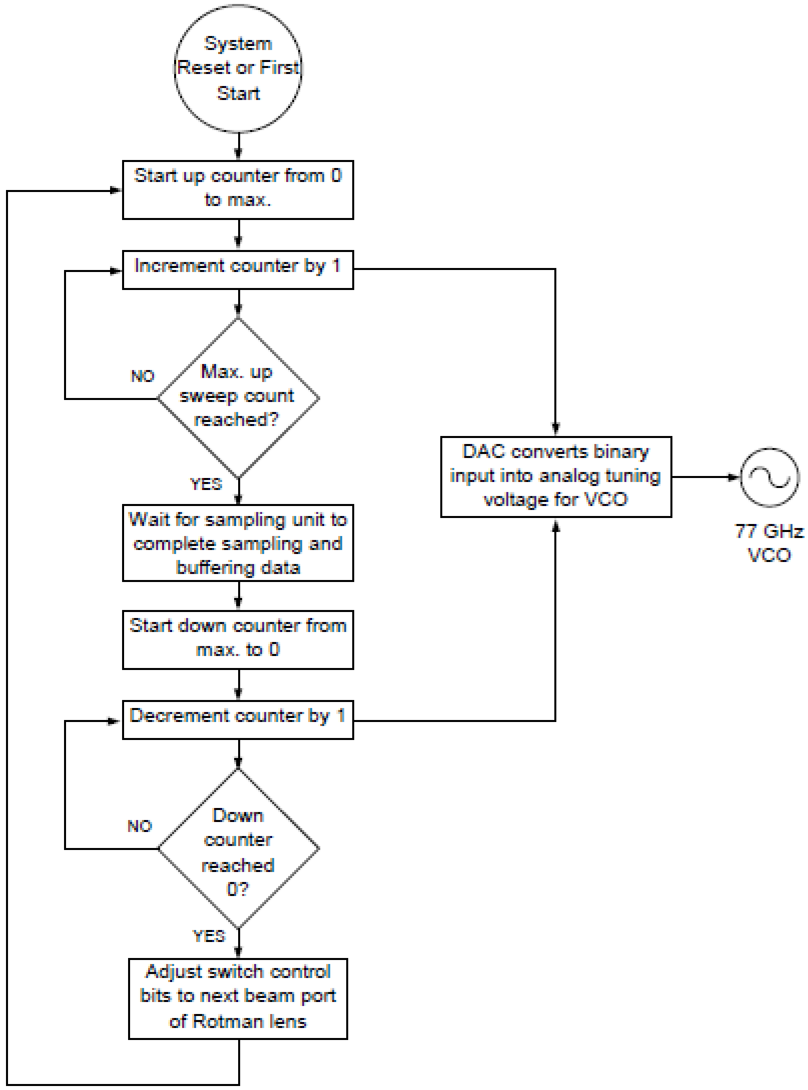

Figure 4 Flowchart for the RADAR transmitter and SP3T switch control [5]

3.2.2. RADAR Receiver Flow Control and Signal Processing

This is the main part of the automotive RADAR signal processing algorithm, which outputs the target information. The input to this are the time-domain samples received from the ADC. The responsibilities of this part of the algorithm are as follows:

- Applying the Hamming window to the received time-domain samples from the ADC.
- Perform Fast Fourier Transformation (FFT) on the windowed time-domain samples to convert into frequency-domain samples.
- Calculating the peak intensity for every frequency bin of the frequency-domain samples.
- Neglecting noise, clutter, and individual target detection by running a Constant false alarm rate (CFAR) algorithm for both up and down sweeps.
- Calculate the final target information by peak pairing, after the CA-CFAR algorithm is done.

The flowchart for the RADAR receiver flow control and Signal Processing can be seen in Figure 5 [5]. Some key information about this part of the algorithm is listed as follows [5]:

- The bandwidth of the system was chosen to be 800 MHz for a respectable range resolution.
- The sampling frequency of 2 MHz was calculated to be used over 1.024 ms for 2048 samples.
- The FFT size, which is the number of samples will be 2048, therefore, 2048 samples will be collected in 1.024 ms.
- Cell-Averaging CFAR (CA-CFAR) was chosen for the CFAR algorithm.

- Since the output from the FFT is symmetrical, only half of the FFT output is considered therefore the CA-CFAR algorithm processes 1024 frequency-domain peaks.
- The probability of false alarm was selected as 10^{-6}, the averaging depth of 4 cells on either side of the CUT and 2 guard bands on either side of the CUT were chosen, which generates a scaling constant of approximately 1.3714.
- Spectral proximity and Power level were the two criteria used for peak pairing.

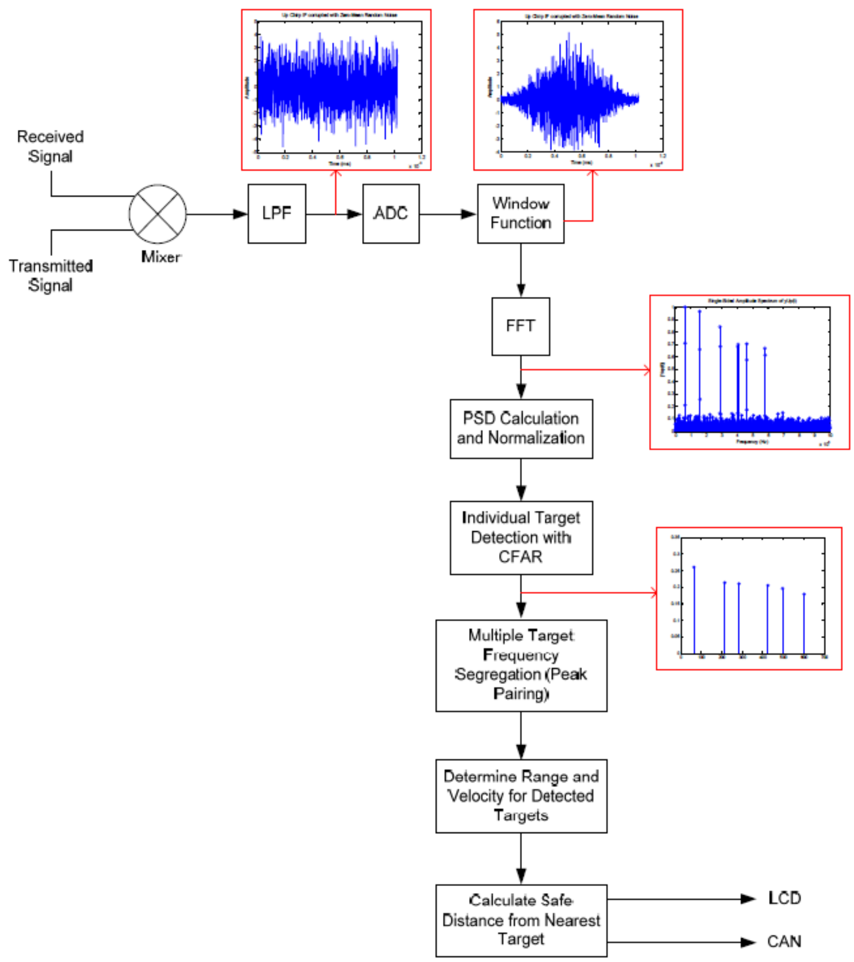

Figure 5 Flowchart of Radar Receiver flow control and Signal Processing [5]

The final design specifications for the complete algorithm can be seen in Table 5. These specifications will be used for the three implementations of the automotive RADAR signal processing system in Chapter 4.

Parameter	Value
LFM-CW sweep bandwidth	800 MHz
FFT size	2048
FFT type	Radix-4 DIT
Up/Down sweep duration	1 ms
ADC resolution / Sampling rate	11 bits / 2.2 MSPS
DAC resolution / refresh period	10 bits / 2.2 MSPS
Target range	0.40m – 200m
Target relative velocity	±300 km/h
CFAR Algorithm	CA-CFAR
CFAR Parameters	One-sided cell-averaging depth = 4 One-sided guard band count = 2

Table 5 Final Parameters for the Algorithm [5]

Chapter 4. Implementations of the RADAR Signal Processing System

This chapter describes the implementations of the automotive RADAR signal processing algorithm at different levels of abstraction.

1. **MATLAB Implementation:** This section discusses the MATLAB implementation of the algorithm, focusing only on the 2nd part of the algorithm which is the RADAR receiver flow control and signal processing.
2. **Existing Verilog Implementation:** This section briefly explains the current HDL implementation of the algorithm [5, 6] targeted for the Xilinx Virtex 5, and the changes we made to this implementation to target the Xilinx Virtex 7.
3. **HLS Implementation and Optimizations:** Here we describe the design methodology we used to implement the automotive RADAR signal processing algorithm using HLS. This section also covers the various HLS code optimization techniques we used to make our design more efficient.

4.1. MATLAB Implementation

A MATLAB implementation of the algorithm was used for testing the correctness of the algorithm using MATLAB version R2016b [20]. The MATLAB design from [5], creates a MATLAB model to do two things:

1. Check the error percentages for the calculated range and velocity.

2. To create sample 10-bit input data which is to be sent to the HDL/HLS based implementations for simulation.

In [5], three tests have been conducted on MATLAB, the first one is the test to see if the hamming window function is necessary, and 2 highway test scenarios to check the accuracy of the signal processing algorithm. A brief explanation of these tests is discussed in this section. The MATLAB implementation does not test the first part of the algorithm which is the RADAR transmitter control and SP3T switch control.

Before we discuss the test scenarios, some parameters used for the MATLAB implementation are as follows:

1. Frequency sweep bandwidth (B) = 800 MHz
2. Sampling Frequency = 2 MHz
3. Sampling duration (T) = 1.024 ms
4. Number of time-domain samples = 2048
5. FFT size = 2048
6. FFT frequency resolution = 2 MHz / 2048 = 976.5625 Hz/bin
7. Rate of change of frequency over a single sweep (k) = Bandwidth/Sampling duration (B/T).

A flowchart of the sequential MATLAB implementation used for the simulation of the test scenarios is shown in Figure 6.

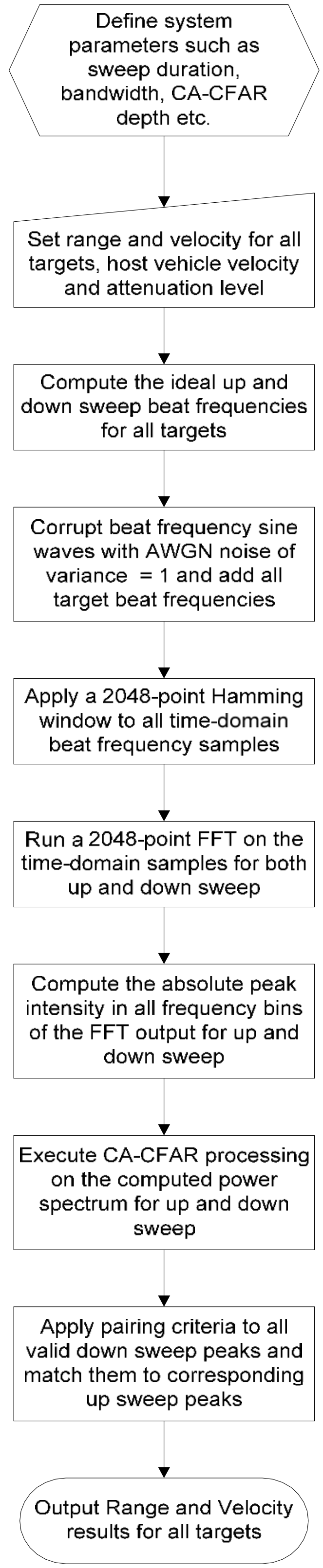

Figure 6 Flowchart of the MATLAB implementation [5]

Scenario to tests the need for Windowing [5]:

This test verifies the requirement of the windowing stage of the RADAR signal processing algorithm.

The test scenario used for this is 1 target at 142 meters with the velocity of 165 km/h, while the host velocity is 70 km/h. Due to the receding target, the relative velocity will be (70 - 165) = -95 km/h (negative Doppler shift) [5]. Assuming c = 2.973 x 10^8 m/s^2.

Results without windowing:

Calculated Up-sweep frequency (f_{up}): 734375.00 Hz

Calculated Down-sweep frequency (f_{down}): 761718.75 Hz

Calculated target range:

$$r = \frac{f_{up} + f_{down}}{2} \times \frac{c}{2k}$$

Therefore, the range r = 142.33 m

Calculated relative target velocity:

$$v_r = \frac{f_{up} + f_{down}}{4} \times \frac{c}{f_0}$$

Therefore, the velocity v_r = -26.497 m/s = -95.39 km/h.

Results with windowing:

Calculated Up-sweep frequency (f_{up}): 733398.44 Hz

Calculated Down-sweep frequency (f_{down}): 760742.11 Hz

Calculated target range:

$$r = \frac{f_{up} + f_{down}}{2} \times \frac{c}{2k}$$

Therefore, the range r = 142.15 m

Calculated relative target velocity:

$$v_r = \frac{f_{up} - f_{down}}{4} \times \frac{c}{f_0}$$

Therefore, the velocity v_r = -26.497 m/s = -95.39 km/h.

Based on these results, there was no change in the velocity measurement of the algorithm. However, the measured range distance had a difference of 18 cm, between with and without windowing. Calculating error percentages for both:

Without windowing: (142.33 – 142)/142 x 100 = 0.23%.

With windowing: (142.15 - 142)/142 x 100 = 0.11%.

Therefore, applying the hamming window to the time-domain samples is a significant improvement in terms of range measurement. These results also agree with the results of the similar test in [5].

Scenarios for the verification of the algorithm [5]:

Test 1 (3-Lane Highway with narrow beam): The scenario for this is illustrated in Figure 7.

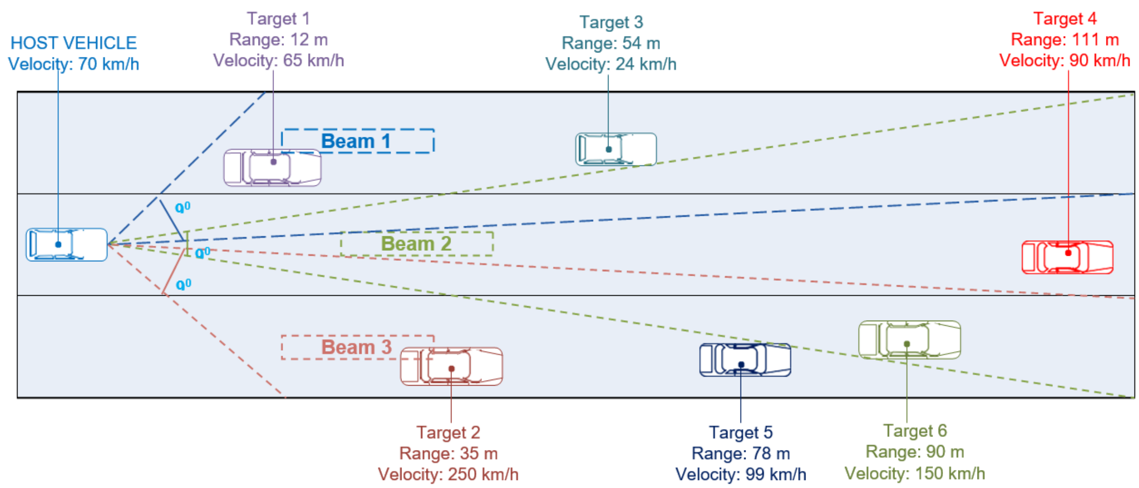

Figure 7 Illustration of test scenario 1 (3-Lane Highway with narrow beam) [5].

The host velocity considered is 70 km/h, 6 targets will be tested, and a 3-beam Rotman lens RADAR sensor is being used. A Signal to Noise Ratio (SNR) of 4.73 dB was used. The specifications for this test scenario are mentioned in Table 6.

Beam Port	Target	Range (m)	Velocity (km/h)	Theoretical Up Sweep IF (Hz)	Theoretical Down Sweep IF (Hz)
1	1	12	65	63784	62358
	3	54	24	290397	277280
2	4	111	90	580509	586212
	6	90	150	461541	484354
3	2	35	250	158148	209477
	5	75	99	405783	414053
	6	90	150	461541	484354

Table 6 Specifications for Test 1 (3-Lane Highway with narrow beam)

The results for this scenario with the error percentages of the algorithm are shown in Table 7.

Beam Port	Target	Real Range (m)	Real Velocity (km/h)	Calculated Range (m)	Range Error (m)	Calculated Velocity (km/h)	Velocity Error (km/h)
1	1	12	65	12.36	0.36	66.59	1.59
	3	54	24	54.35	0.35	25.71	1.71
2	4	111	90	111.30	0.30	90.44	0.44
	6	90	150	90.21	0.21	148.36	1.64
3	2	35	250	35.30	0.30	247.15	2.85
	5	75	99	78.32	0.32	100.66	1.66
	6	90	150	90.30	0.30	151.76	1.76

Table 7 Results for Test 1 (3-Lane Highway with narrow beam)

The maximum errors for this scenario are:

Range: For target 1 detected at beam port 1 = 0.36 m or (0.36/12) x 100 = 3%.

Velocity: For target 2 detected at beam port 3 = 2.85 km/h or (2.85/250) x 100 = 1.14%.

Test 2 (3-Lane Highway with a single wide beam): The scenario for this is illustrated in Figure 8.

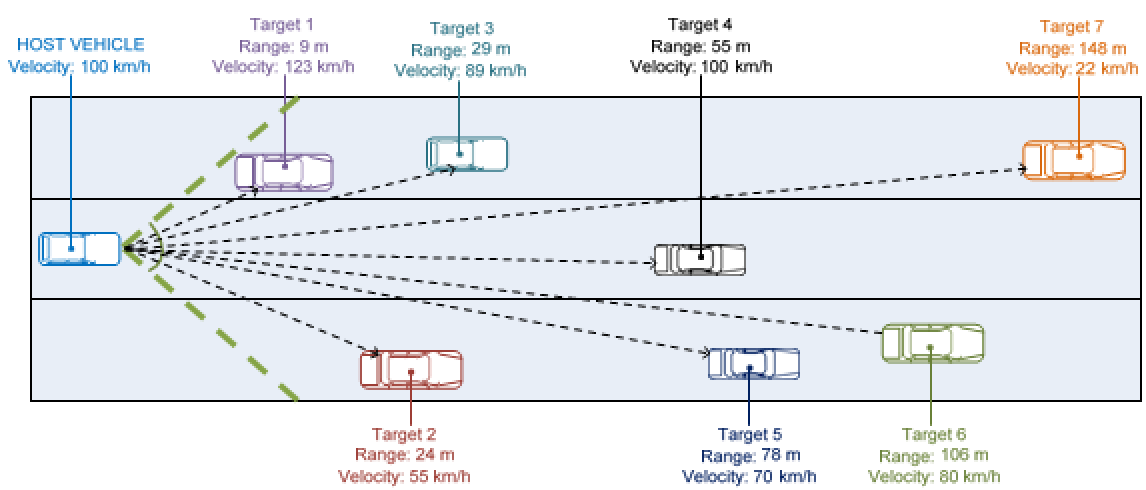

Figure 8 Illustration of test scenario 2 (3-Lane Highway with single wide beam) [5].

The host velocity considered is 70 km/h, 7 targets will be tested with a single wide beam. A Signal to Noise Ratio (SNR) of 4.73 dB was used. The specifications for this test scenario are mentioned in Table 8.

Target	Range (m)	Velocity (km/h)	Theoretical Up Sweep IF (Hz)	Theoretical Down Sweep IF (Hz)
1	9	123	44004	50563
2	24	55	132585	119753
3	29	89	153990	150853
4	55	100	289060	289060
5	78	70	414239	405684
6	106	80	559964	554261
7	148	22	789013	76671

Table 8 Specifications for Test 2 (3-Lane Highway with a single wide beam)

The results for this scenario with the error percentages of the algorithm are shown in Table 9.

Target	Real Range (m)	Real Velocity (km/h)	Calculated Range (m)	Range Error (m)	Calculated Velocity (km/h)	Velocity Error (km/h)
1	9	123	9.38	0.38	123.85	0.85
2	24	55	24.34	0.34	52.31	2.69
3	29	89	29.27	0.27	89.78	0.78
4	55	100	55.37	0.37	100.00	0.00
5	78	70	78.32	0.32	69.34	0.66
6	106	80	106.28	0.28	79.56	0.44
7	148	22	148.37	0.37	21.64	0.36

Table 9 Results for Test 2 (3-Lane Highway with a single wide beam)

The maximum errors for this scenario are:

Range: For target 1 = 0.38 m or (0.38/12) x 100 = 3.167%.

Velocity: For target 2 = 2.69 km/h or (2.69/250) x 100 = 1.076%.

4.2. HDL-Based Implementation

The HDL-Based model's overview can be seen in Figure 9, the HDL for this design is Verilog. The design is synthesized and simulated using Xilinx Vivado. The original design from [5] was originally designed for the Xilinx Virtex-5 FPGA but has been updated for the Xilinx Virtex-7.

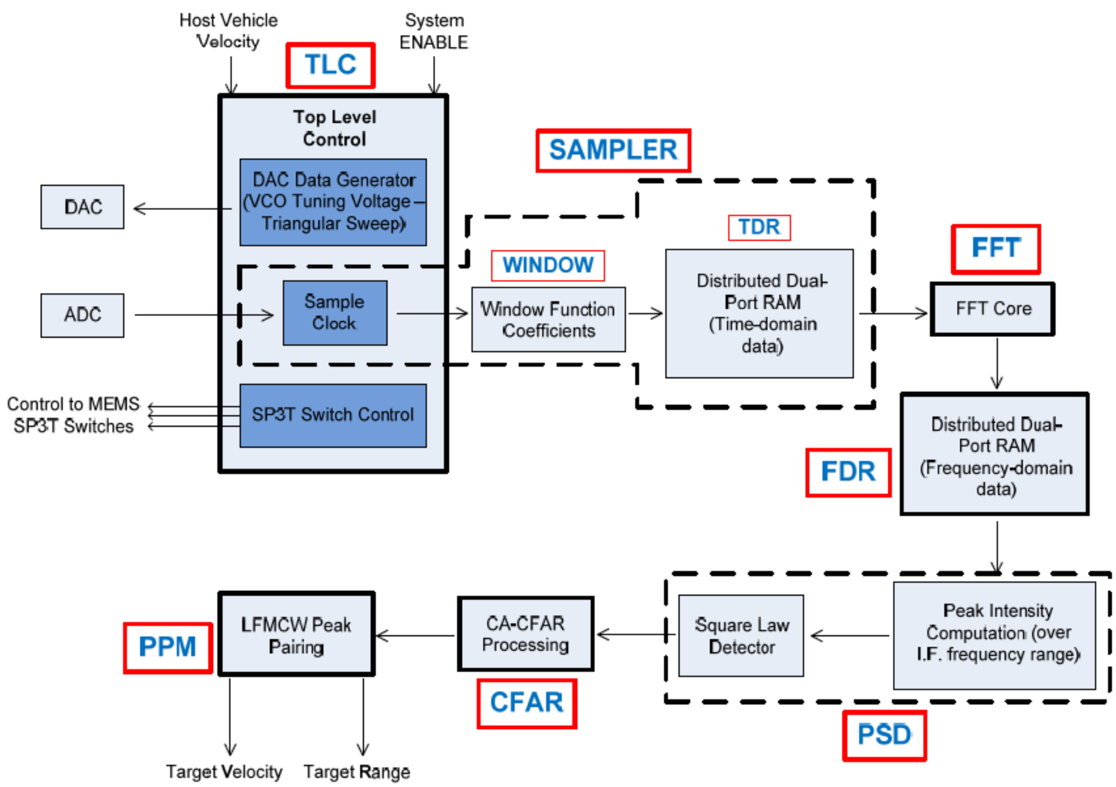

Figure 9 HDL-based design overview [5]

4.2.1 TLC – Top Level Control

The Top-Level Control (TLC), is responsible for providing the basic control for the design. This includes the VCO tuning based on the modulating clock, the sampling clock to the ADC and the sampler, and the SP3T switch control. The TLC is the main control block for the design and an overview can be seen in Figure 10.

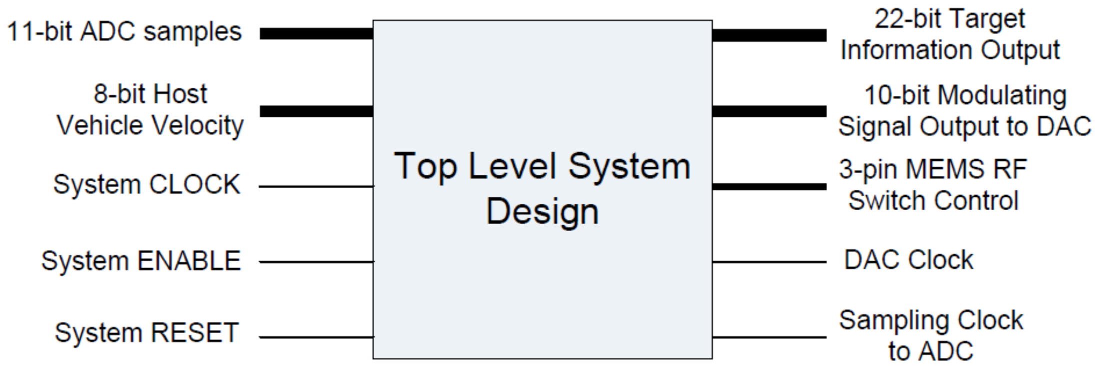

Figure 10 TLC overview

The **22-bit target information output** consists of:

- Most significant 10 Bits: 10-bit target velocity, where 9 bits are used for the integer part and 1 bit for the fraction, therefore, the velocity is [9-bits].[1-bit].
- The next 10 bits: 10-Bit target range, where 8 bits are used for the integer part and 2 bits for the fraction part, therefore, the range is [8 bits].[2-bits].
- The last 2 bits: The last 2 bits of the final information includes the beam port number from which the target was detected.
    - 01: For beam port 1 (100).
    - 10: For beam port 2 (010).
    - 11: For beam port 3 (001).

The **10-bit modulating output** to DAC controls the VCO voltage based on the DAC clock.

The **3-pin MEMS RF switch control** controls the beam port from which the next up/down sweep data will be received from the ADC.

The **DAC clock provides** the operating clock to the DAC for VCO tuning.

The **sampling clock to ADC** is a 2 MHz clock which is the operating clock frequency of the ADC.

4.2.2 Sampler

The sampler or the ADC-control is responsible for receiving the data from the ADC, while also providing the necessary logic for the 2 MHz sampling clock to the ADC. The sampler is also responsible for applying the hamming window function to the time-domain inputs received from the ADC. The data is then stored in a dual-port RAM which allows us to access the stored windowed data from the next module.

4.2.3 FFT

The FFT module is responsible for performing the Fast Fourier transform on the windowed time-domain data from the Sampler. Due to the symmetrical nature of the Frequency domain data from the FFT, the first 1024 values are ignored by the system since it also contains more noise. The last 1024 frequency-domain output from the FFT core is then stored in another dual-port RAM. The specifications of the FFT core can be seen in Table 10.

Parameter	Value
FFT size	2048
Architecture type	Burst I/O
Radix	Radix-4
Input word length	12 bits
Output word length	12 bits (scaled)
Scaling type	Rounding
I/O data type	2’s complement
Internal phase factor length	16 bits

Table 10 FFT Specifications [5].

4.2.4 Peak Intensity Calculator (PSD)

The peak intensities for all the 1024 frequency domain data from the FFT are calculated here. Before sending the peak intensities forward to the CFAR processor, they are passed through a square-law detector unit which ensures positive peak intensities for the entire data.

4.2.5 Constant False Alarm Rate Processor (CFAR)

The CA-CFAR algorithm is implemented in this block. The CFAR processor receives the data from the PSD in batches of 4 and then stored in a Block-Ram. Once 32 frequency-domain values are received by this block. The CA-CFAR algorithm removes the unwanted clutter and noise due to the various reasons like system noise and weather conditions.

4.2.6 Peak Pairing Module

This module is responsible for pairing the peak intensities to detect valid targets from the CFAR processed frequency domain data. The criteria used for peak paring are Spectral proximity and power level comparison. The output from this module contains the target information (velocity, range, and beam port) and is sent to the TLC for the final adjustments and then sent as an output to the final design.

4.2.7 Usage/Timing analysis for the design

The resource utilization of the original RTL-based design for the Xilinx Virtex-5 board is shown in Table 11 and the timing analysis for the same can be seen in Table 12.

Resource	Used	Available	% Usage
Slice Registers	1357	32640	4 %
Slice LUTs	7445	32640	22 %
DSP48E slices	17	288	5 %
Fully used LUT-FF pairs	705	8097	8 %
BUFG/BUFGCTRLs	1	32	3 %
FPGA fabric area ratio	21	100	21 %

Table 11 Resource utilization on the Virtex-5 board. [5]

Operation	Effective Clock cycles per beam	Latency per Beam with operating Clock at 100 MHz (in ms)
Up sweep sampling	204756	2.047560 ms
Window and feed time-domain samples to FFT core	2072	0.020720 ms
FFT calculation	3960	0.039600 ms
Peak Intensity Calculations	10743	0.10743 ms
CFAR processing and Peak Pairing	4388	0.060460 ms
Total Signal Processing Latency	21163	0.211630 ms
Overall Latency	225928	2.259280 ms

Table 12 Timing/Latency analysis for the HDL-based design

4.3. High Level Synthesis of the RADAR system

The basic structure for the HLS-based design was kept the same to ensure a fair comparison between the HLS and HDL-based design. The basic structure for the design is shown in Figure 9. This section discusses the design methodology used for HLS. The design was synthesized and simulated using Xilinx Vivado HLS and the FPGA we selected was the Xilinx Virtex-7.

The list of the inputs and outputs for our design is similar to the ports for the RTL-based design and are shown in Table 13.

I/O	Port Name	Width (in bits)	Description
Inputs	Reset	1	System Reset
	Unit_vel	8	Host Velocity in km/h
	Data_in	11	Input Data from the ADC
Outputs	Sclk	1	Clock for ADC
	Beamport	3	The beam port signal to switch the S3PT switch
	Modulate	10	Modulation output for VCO tuning
	Final_target_info	22	Target information which includes target velocity, target range and the beam port in which the target was detected.
	Final_info_valid	1	Flag to indicate valid final_target_info

Table 13 Inputs and Outputs of the main design.

4.3.1. HLS programming techniques

Some features which are important for hardware design are not present in standard HLLs. Since we are using C++, Xilinx Vivado HLS provides us with some useful techniques to overcome this limitation. Some of the features [8] we used to program the design are:

1. Arbitrary Precision

- Data types in standard HLLs like C++ only allow the programmer to use data with 8-bit boundaries, for example, integers can be represented as either 32 bits or 64-bits for newer systems.
- Due to the importance given to bit length in hardware design to preserve memory resource usage and to reduce time delays for operations, arbitrary

precision allows us to use variables with arbitrary precision. Xilinx Vivado HLS currently supports bit-widths from 0 to 1024.

- There are two basic data types for arbitrary precision
    - **i.** For Integers: ap_int or ap_uint are two data types which are available to us. The data width (in bits) is mentioned when these data types are initialized, for example: ap_int<5> would be a 5 bit signed integer.
    - **ii.** For Non-integer numbers: ap_fixed and ap_ufixed are two data types which are available to us for fraction numbers. The data width (in bits) for the entire number and the data width of the integer part are mentioned during initialization of the variable. For example: ap_fixed<20, 12> would be a 20-bit value with 12 bits for the integer part and 8 bits for the fraction part.
- These data types allow us to work with variables which are not bounded by the 8-bit boundary. These data types also include member function which returns the value in different data types for type casting or performing operations. A few of these member functions are:
    - **i.** to_[u]int(): This function returns the [un]signed integer represented by the arbitrary precision type.
    - **ii.** to_[u]double(): This function returns the [un]signed floating point value.
    - **iii.** range(A, B): This function returns the value represented by the bits A to B. The return type for this function is the same data type as the original variable but is resized to (B-A) bits.

2. **Multiple outputs from a function**

- In standard C++, functions are only allowed to return one type of data, this data could be an integer, floating point number, or an array etc. In hardware design, however, there are usually more than 1 outputs, therefore, there are techniques which are used to support multiple output behavior.
- In Xilinx Vivado HLS, the compiler detects when the values to a function are passed by value or by reference.
    i. The data passed by value to a function are automatically assigned as Input ports.
    ii. The data passed by reference can be assigned as inputs, outputs, or bi-directional input/output ports based on how they are being accessed inside the function. If there are writes to this particular reference and no reads, they are assigned as output ports. If there are reads and no writes, they are assigned as input ports, and if they are being read from and written to, they are assigned as I/O ports.
- Therefore, we use pass by value for the inputs and pass by reference for outputs and I/O ports.

3. **Registers**

- In C++, there are no available data types which mimic a register in a hardware. These registers are important since they retain information which were acquired during the last function run.
- Xilinx Vivado HLS uses static variables for register initialization so the important data which was stored during the last function call can be accessed.

- To use this, “static” keyword is used before the variable initialization. For example, “static int a = 0;”. The variable “a” in this example will only initialize once. Once this variable is written to, the value stored in “a” will be maintained for the next function run.

4. Memory Interface

- In C++, the memory interface equivalent are arrays. Arrays are blocks of data which can store multiple values of the same data type.
- Random Access memory (RAM) equivalent, these would be regular arrays and Vivado HLS automatically recognizes if there are both reads and writes to the array. If there are only read operations for an array they would be treated as Read-only memory (ROM).
- Read-Only Memory (ROM): Even though Vivado HLS automatically recognizes if an array is RAM or ROM based on the type of access it has, some arrays can be forced to be treated as ROMs by adding the keyword “const” before the array initialization. For example: “const int[5];” would be an array which stored 5 integer values and will be treated as ROM which implies that this array will only be written to once. The reason to use this memory interface is that it takes less time to access data from a ROM than the time taken to access data from a RAM.

All these additional features help us design efficient and proper hardware using HLLs since they were not originally designed to develop hardware.

4.3.2. Top-Level Function/Module

The top module for our design is a function called "RPU()", the inputs and outputs for this function are shown in Table 13. We used arbitrary precision data types for all the inputs, outputs, and the internal registers. For 1 bit internal flags, data type "bool" was used which are 1-bit data types by default and can have "true" or "false" as their values. The reason for this is that "bool" types in C++ are very well managed when it comes to condition management like if-else statements etc. This function, similar to the RTL-based design, is responsible for VCO tuning, beam port switching, final target information management. The clock for the ADC is also managed by this function.

This function also calls the necessary functions for further calculations. The function calls from this function are:

- Adc_control(): This function is called to store the input data from the ADC.
- Fft_control(): This function is called after data for each sweep is collected and the FFT computation can be started.
- Fft_absolute(): This function is called after the FFT computation has been completed to store and forward the frequency domain data for further calculations.
- Cfar(): This function is called for the CA-CFAR computations [5].
- Peak_pairing(): This function is called to detect valid targets and calculations for the range and velocity of the target [1, 5].

After these functions, the final target information is filled and the final information valid flag is set to 1 stating that valid output is available.

The input and output of this function are the main I/O for our design which are shown in Table 13.

4.3.3. Adc_control() function

This function is responsible to store the input data from the ADC. The Hamming window is applied to the input data and sent to the fft_control() function to be stored. The Hamming window coefficients are stored in a constant array (ROM) of size 1024 and stores 10-bit window coefficients. These values are acquired using MATLAB. This function multiplies the input data by the window coefficient and then rounded to 12 bit signed values and sent to the fft_control() for further computations. The inputs and outputs for this function can be seen in Table 14.

I/O	Port	Width (in bits)	Description
Inputs	Reset	1	System Reset
	Data_in	11	Input Data from the ADC
Outputs	Xn_index	11	Index to maintain the number of data acquired from ADC.
	Xn_value	12	Windowed time-domain data
	Fft_start	1	Flag to start FFT computation
	Fft_record	1	Flag to indicate fft_control needs to store the windowed data

Table 14 I/O for adc_control() function

4.3.4. Fft_control() function

This function is responsible for storing the windowed time-domain data from the adc_control(). Once 2048 values have been stored in a First-In-First-Out(FIFO) memory interface, the FFT computation is started and the output is stored in another FIFO. The output data which is the frequency-domain values are complex variables. Since we only need the magnitude (or absolute value) of this data, the absolute value is computed as the square root of the sum of squares of the real and the imaginary part. These absolute values are then sent to fft_absolute() function for further computations. The inputs and outputs for this function can be seen in Table 15.

I/O	Port	Width (in bits)	Description
Inputs	Reset	1	System Reset
	Fft_record	1	Flag to indicate that this function should store the windowed data from adc_control()
	Fft_start	1	Flag to indicate FFT computations can be started
	Xn_index	11	The index where the data will be stored in the FIFO
	Xn_value	12	The value which will be stored at xn_index in the FIFO
Outputs	Xk_abs	13	The absolute value of the frequency domain data.
	Xk_index	11	The index indicating the number of data currently been sent in xk_abs
	Fft_done	1	Flag to indicate FFT computations are finished

Table 15 I/O for fft_control() function

4.3.5. Fft_absolute() function

This function is responsible for storing 4 absolute values of the frequency-domain data and are sent to the cfar() function for CA-CFAR computation. Since CFAR works with bins of 32 values at a time, we send 4 values at a time so this function runs 8 times before CFAR computation begins. The inputs and outputs for this function can be seen in Table 16.

I/O	Port	Width (in bits)	Description
Inputs	Reset	1	System Reset
	Fft_done	1	Flag to indicate FFT computations are finished
	Xk_index	11	The index indicating the number of data currently been sent in xk_abs
	Xk_abs	13	The absolute value of the frequency domain data.
Outputs	outA, outB, outC, outD	13	4 absolute values to be sent to the cfar() function
	Abs_done	1	Flag to indicate that 4 values are being sent.

Table 16 I/O for fft_absolute() function

4.3.6. Cfar() function

This function is responsible for CA-CFAR computations which store the left and right averages of a particular frequency bin of 32 frequency-domain data to remove noise and to detect a new target. Once these computations are done, a new target flag is set to true if a target has been detected. The absolute value of the particular target and the information regarding where it was detected is then sent to peak pairing for the range and velocity calculations. The inputs and outputs for this function can be seen in Table 17.

I/O	Port	Width (in bits)	Description
Inputs	Reset	1	System Reset
	inA, inB, inC, inD	13	4 absolute values to be sent to the cfar() function
	Start	1	Flag to indicate that 4 values are being sent. (abs_done output from fft_absolute())
Outputs	Target_abs	13	The absolute value of the frequency domain data of the corresponding detected target
	Target_pos	10	The position of the target in the frequency bin where it was detected.
	New_target	1	Flag to indicate a new target has been detected and the values in target_abs and target_pos are valid.
	complete	1	Flag to indicate that the CFAR computations have been completed for the particular frequency bin.

Table 17 I/O for cfar() function

4.3.7. Peak_pairing() function

This function is responsible to validate the detected targets from the previous functions and calculate the range and velocity of the target in respect to the host vehicle. This function waits for the previous computation of each bin and calculates the ranges and velocity of the detected targets in each bin. Peak pairing allows us to validate the targets and avoid the spectral copies of a valid target to prevent multiple outputs for the same target. This function uses spectral proximity and power level comparison as criteria for the necessary computations. The calculated range and velocity are adjusted and sent to the top level function (RPU()) for the final output. The inputs and outputs for this function can be seen in Table 18.

I/O	Port	Width (in bits)	Description
Inputs	Reset	1	System Reset
	Target_abs	13	The absolute value of the frequency domain data of the corresponding detected target
	Target_pos	10	The position of the target in the frequency bin where it was detected.
	New_target	1	Flag to indicate a new target has been detected and the values in target_abs and target_pos are valid.
	complete	1	Flag to indicate that the CFAR computations have been completed for the particular frequency bin.
	Updown	1	Flag to indicate if up-sweep or down-sweep.
	Unit_vel	8	Host Velocity in km/h
Outputs	Target_info	20	Calculated range and velocity of the target
	Info_valid	1	Flag to indicate if the target_info is valid.

Table 18 I/O for peak_pairing() function

4.3.8. Optimizations and Final Design

In the HLS code, these functions are called sequentially but they are controlled using Boolean flags which help us control when a particular function is to be started. Along with that, while the completion of the previous task is in progress, the initialization process such as recording data etc. are done to maximize throughput for each function call. Although Xilinx Vivado HLS analyzes and optimizes our design automatically, we also have the option to optimize it manually giving us flexibility [6, 7, 14, 15]. A few automatic and manual optimizations which were implemented are:

- **Memory optimizations:**
    - Memory usage, in general, includes internal registers, RAMs, and ROMs.

- For Internal registers: The concept of arbitrary precision as mentioned in Section 4.3.1 is used to reduce the resource usage for the registers. Instead of storing the values in 8-bit boundaries (8, 16, 32, etc), we store the data in the necessary bit-width using arbitrary precision. For example: variable xn_index has a max value of 2047 and is always greater than 0, instead of storing this value in an int type which is 32-bits, we use ap_uint<11> which is unsigned 11-bits, therefore we save 21-bits to store this value.
- For Arrays (RAM/ROM): We mentioned the concept of memory management in HLS in Section 4.3.1, using this technique we use keywords like "const" or pragmas for FIFO to optimize memory. For example: "window" array is read only since we only need to read the window coefficients for multiplication, therefore, instead of using this array as a RAM, we use the "const" keyword to initialize the array which implies that this array will be implemented as a ROM in our design.

- **Timing-Based optimizations**
  - Both software and hardware developers focus on speeding up the design, although implementing these optimizations are very different when it comes to software and hardware.
  - The first step to optimize the timing of a certain design is top optimize the algorithm itself, we used Boolean flags to control when the actual calculations of a particular function needs to be started, although during

the time when the flag for previously called function is false and hence is in progress, the next functions which are called start initialization necessary data to prepare for when the previous function is completed. This saves time since the initializations are already done before the actual computation starts.

  - Xilinx Vivado HLS automatically recognizes when pipelining is necessary by analyzing the operations in a particular function. For example: when the FFT-core starts the FFT computations, the rest of the design continues to work since there are no more dependencies for the FFT- core.
  - Some other internal optimizations include designing the system in such a way that every snippet of code is written to maximize the operations so the entire design would require fewer clock cycles overall. For example: As soon as the FFT-computation is finished, the frequency domain data's absolute value is calculated and sent to the next function, instead of waiting for the next run.

- **Automatic optimizations:**
  - Vivado HLS analyzes the entire design and implements the operations with no dependencies in parallel, although this increases resource utilization of the final design, it decreases the latency significantly and hence the trade-off between resource utilization and time delay is reasonable. Although this behavior can be disabled by

using pragmas, we keep this enabled since the newer FPGAs have very high capacity.

- Vivado HLS also recognizes the array usage and change the access type of each array based on how, when and where the array is accessed. For example: The output of the FFT-core is set as a ROM automatically since once the output is stored, we are only reading from this array. Vivado HLS also recognizes that we are accessing the values from this array one by one starting from index 0, it implements this ROM as a FIFO which implies that as soon as the value is ready, it can be read from which provides us with significant decrease in design latency.

These optimizations techniques combined with the already optimized algorithm from [5] gives us a very efficient design. The final comparison and results are discussed in the next chapter.

Chapter 5. Synthesis Results

This chapter discusses the results of our synthesized designed which includes resource and timing analysis of our design and the comparison between the RTL-based design and the HLS-based design.

5.1. Approximate Time-To-Market

Time-To-Market for a product is the time taken from the starting of the project to the end of development. Even with the availability of newer, more advanced tools, Research and Development(R&D) cost and time are usually very high. Based on the design we are considering, designing hardware using HLS requires a lot less time to program when compared to RTL-based designs. The main reason for this is the use of HLLs, high level of abstraction enables us to design hardware faster and more efficiently, decreasing the R&D time and cost. Based on our experience, an approximate comparison on the number of weeks it will take for an average hardware developer to develop our selected algorithm using the RTL-based methodology and HLS-based methodology is shown in Figure 11.

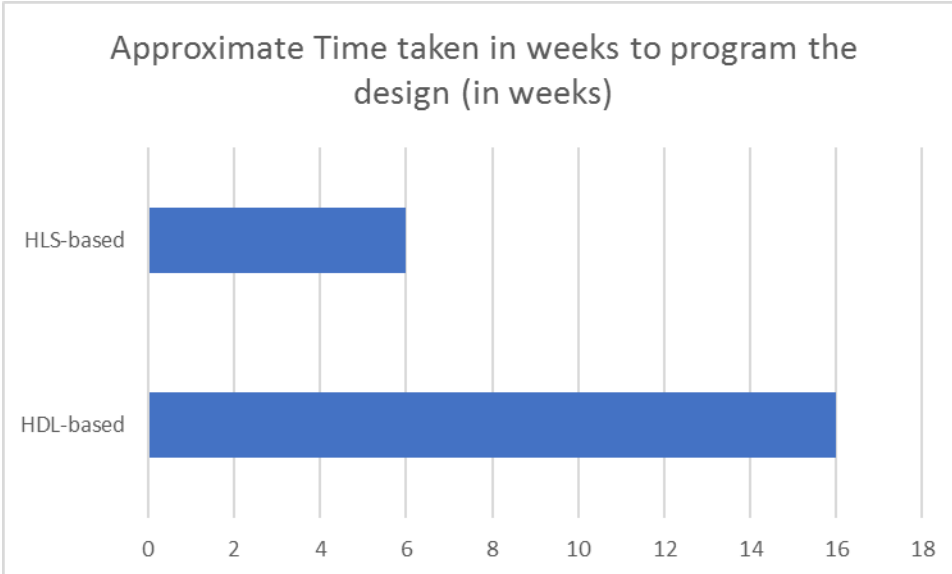

Figure 11 Approximate time taken to write HDL and HLS code.

As we can see from the graph, the time taken to program using HLLs is significantly less, this is due to the high level of abstraction HLLs provides and are comparatively faster to code in than HDLs.

5.2. Resource Utilization

The resource utilization for HLS-based designs is usually significantly higher than the corresponding RTL-based design due to the high focus on reducing time latency for the overall design. Due to the advancements in current hardware, this becomes less of an issue since the newer hardware like FPGAs have very high capacity when it comes to LUTs, registers, RAM/ROM, and other resources. A general resource utilization breakdown for our design is shown in Table 19.

Module/Resource	BRAM	DSP	FF	LUT
RPU (Top-Level)	37	39	15551	16749
FFT_ABSOLUTE	0	0	39	25
FFT_CONTROL	35	34	14490	14976
RPU_CFAR	1	1	307	663
PEAK_PAIRING	0	2	395	608

Table 19 Resource utilization breakdown

This section compares the resource utilization of our design for the Xilinx Virtex-7 FPGA between the HLS-based and HDL-based design.

- **RAM:** the comparison of overall RAM usage of the design is shown in Table 20 and Figure 12. Even though the usage suggests that the HLS-based uses about 4.75 times more RAM, it is still significantly less than the available RAM available in the Virtex-7[12] making the RAM utilization of only 1.3%.

RAM usage in (Kb)		
Available	**RTL-based design usage**	**HLS-based design usage**
52920	144	684

Table 20 RAM utilization comparison.

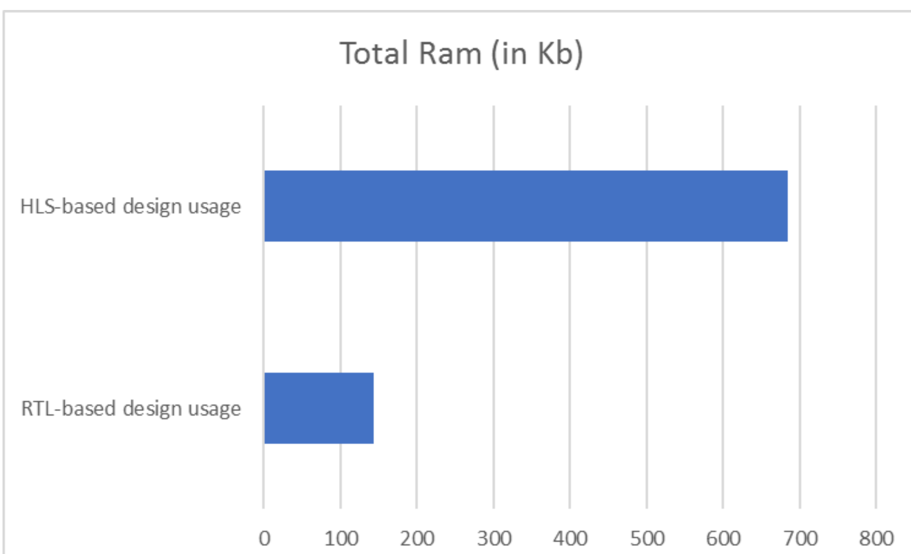

Figure 12 RAM utilization comparison graph

- **Slice LUTs / Slice Registers:** The comparison of the number of Slice LUTs and registers used in our design for the Xilinx Virtex-7 FPGA is shown in Table 21 and Figure 13. Although the HLS-based design utilization for both Slice LUTs and registers are significantly higher than the RTL-based design, the utilization percentage based on the available resources are around 4% for the Slice LUTs and around 2% for Slice registers.

Slice LUTs and Slice Registers usage			
Resource	**Available**	**RTL-based design usage**	**HLS-based design usage**
Slice LUTs	433200	8662	16749
Slice Registers	866400	3381	15551

Table 21 Slice LUTs / Registers utilization comparison.

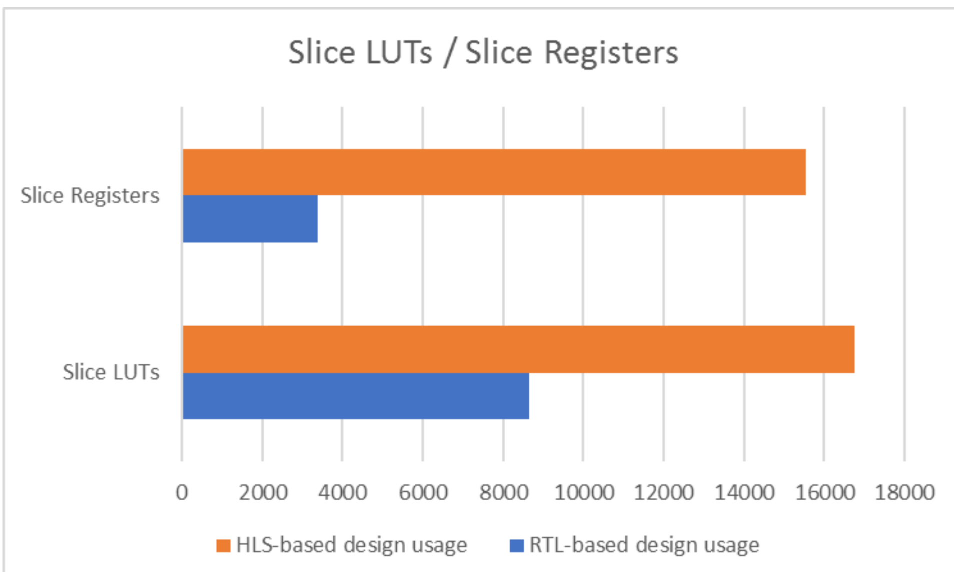

Figure 13 Slice LUTs / Registers utilization comparison graph

- **BRAM_18K / DSP48E1 resources:** The comparison of the number of embedded BRAM and the DSP blocks used in our design for the Xilinx Virtex-7 FPGA is shown in Table 22 and Figure 14. Although the HLS-based design utilizations are significantly higher than the RTL-based design, the utilization percentage based on the available resources are around 1.3% for the BRAM and around 1.1% for the DSP blocks.

BRAM and DSP block utilization			
Resource	**Available**	**RTL-based design usage**	**HLS-based design usage**
BRAM_18K	2940	8	38
DSP48E1	3600	29	39

Table 22 BRAM and DSP blocks utilization comparison.

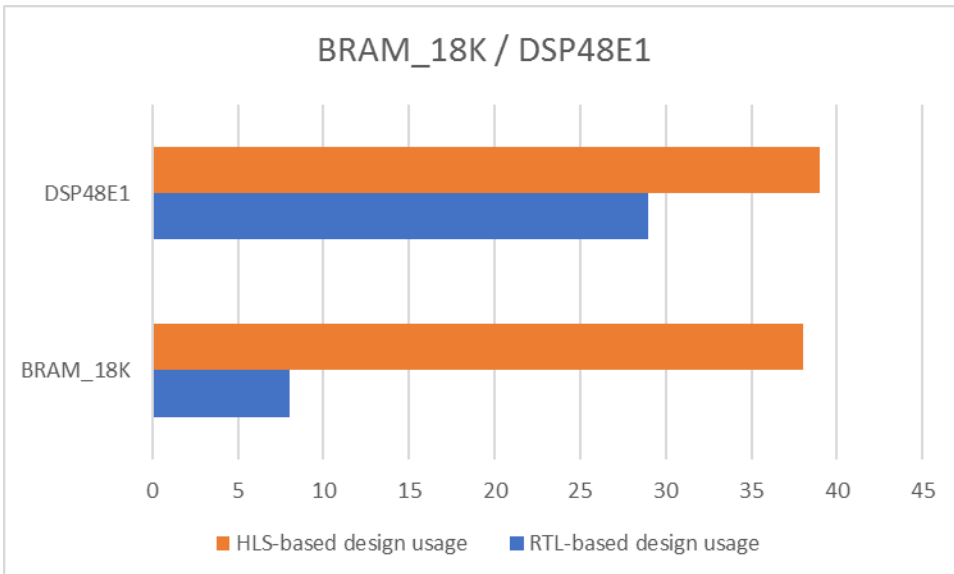

Figure 14 BRAM and DSP blocks utilization comparison graph

Based on the utilization numbers, we conclude that although the HLS-based design utilizes a lot more resources than the RTL-based design, the newer FPGAs have significantly large amount of available resources and hence the overall usage of these resources are still very low. In applications where time to market is crucial, HLS is the preferred design methodology even though it uses up more resources compared to HDL based design methodology.

5.3. Performance

Performance is usually measured as the time taken for a design to provide a valid output. The time difference between when the 1st input was taken into the design and when the valid output was outputted from the design is called latency (time difference between stimulus and response). We use the computed latency for both the RTL-based design and the HLS-based design to provide a comparative analysis, which can be seen in Table 23 and Figure 15.

Latency in	RTL-based design	HLS-based design
Clock Cycles	22018	10863
Time in milliseconds	2.2018 ms	1.0863 ms

Table 23 Latency comparison

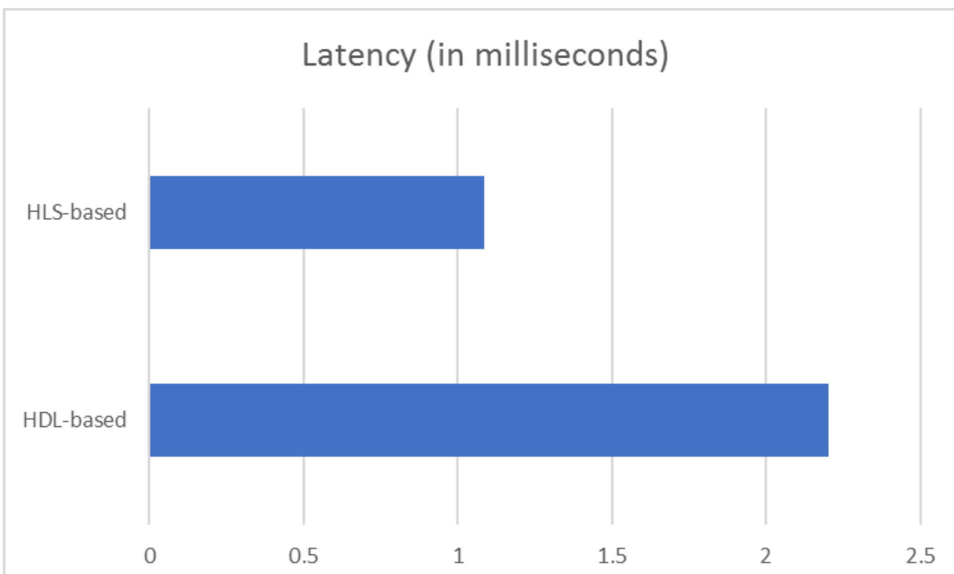

Figure 15 Latency comparison graph

Based on these results, we can see that the HLS-based model is around twice as fast as the RTL-based design. Achieving around 2X speed-up on the HLS-based design implies that even though the comparative resource utilization is a lot higher for the HLS-based design, the performance of the design has a significant improvement. If we take into consideration how many resources are available to us, this area vs speed-up trade-off is very good as our overall resource utilization is still under 5% of the total available resources.

Chapter 6. Conclusions and Future Work

This section concludes our work by providing some information including summary and future work.

6.1. Summary

High Level Synthesis of an automotive RADAR signal processing system was performed which included the RADAR signal processing algorithm to detect targets in automobiles including the target distance from the host and the velocity of the target. Using an existing HDL-model of this system and modifying it for a fair comparison between the two models, we successfully synthesized the design for the Xilinx Virtex-7 FPGA using the Xilinx Vivado Design Suite. For our HLS, we achieved an overall speed up of about 2X when compared to the HDL-based design which is a significant improvement while keeping the overall resource utilization at under 5% with respect to the available resources. The C++ code for our design is available in the appendix section of this thesis.

6.2. Future Work

For future research, there are a few places where the design can be made more efficient. A few of them are:

- Optimizing the algorithm itself: The algorithm used for this thesis is somewhat optimized but can be further improved.

- Using newer and more advanced hardware: For future improvements, the current ADC, DAC, and the VCO can be updated to newer ones to remove the data transfer speed bottleneck issue.
- Pipelined Design: Although Xilinx Vivado HLS automatically analyzes the design to implement pipelined designs. The HLS code written in C++ can be further modified to maximize this feature which will include research on how the pipelining is done in Vivado HLS. This topic by itself is a significantly large project.

Future research can also include HLS of even more complex applications.

References

[1]. E. Casseau and B. Le Gal. “High-level synthesis for the design of FPGA-based signal processing systems”. In SAMOS, pages 25–32, 2009.

[2]. X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, D. Chen. “High Level Synthesis of Complex Applications: An H.264 Video Decoder”. FPGA’16, Monterey, California, USA, February 2016.

[3]. Xilinx. Xilinx Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html

[4]. S. Zereen, S. Lal, M. Khalid, S. Chowdhury S, "An FPGA-Based Controller for a 77 GHz MEMS Tri-Mode Automotive Radar", Proc. of 24th ACM/SIGDA International Symposium on FPGAs (FPGA 2016), Monterey, California, United States, February 2016.

[5]. S. Lal, “An FPGA-based 77Ghz RADAR Signal Processing System for Automotive Collision Avoidance”, University of Windsor, Windsor, Ontario, Canada, May 2010.

[6]. Xilinx, UltraFAST. “UltraFast High-Level Productivity Design Mothodology Guide”, UG1197, 2017.

[7]. Xilinx, UltraFAST. “UltraFast Design Methodology Guide for the Vivado Design Suite”, UG949, 2016.

[8]. Xilinx. “Introduction to FPGA Design with Vivado High-Level Synthesis.”, UG998, 2013.

[9]. E. Oruklu, R. Hanley, S. Aslan, C. Desmouliers, F. M. Vallina, and J. Saniie. “System-on-chip design using high-level synthesis tools”. Circuits and Systems, 3(01):1, 2012.

[10]. G. Inggs, S. Fleming, D. Thomas, and W. Luk. “Is high level synthesis ready for business? A computational finance case study”. In FPT, pages 12–19, 2014.

[11]. Berkeley Design Technology. “An independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs”. http://www.bdti.com, 2010.

[12]. Xilinx, “7 Series FPGAs Data Sheet: Overview”, DS180, 2016.

[13]. R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. Ting Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, K. Bertels. “A Survey and Evaluation of FPGA High Level Synthesis Tools”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2015.

[14]. Xilinx. “Improving Performance – Vivado HLS Version”.

[15]. Xilinx. “C-based Design: High Level Synthesis with Vivado HLx tool”.

[16]. W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, D. Stroobandt. “An overview of today’s high-level synthesis tools”. Springer Science + Business Media, LLC 2012.

[17]. Xilinx, “Vivado Design Suite User Guide”. UG910, 2016.

[18]. Xilinx, “Vivado Design Suite User Guide”. UG902, 2016.

[19]. Xilinx, “Vivado Design Suite User Guide”. UG871, 2016.

[20]. MathWorks. MATLAB. https://www.mathworks.com/product/matlab.html, 2016

Appendix – Source Code

Main header file (main_head.h)

```
#ifndef _MAIN_HEAD_H_
#define _MAIN_HEAD_H_

#include <iostream>
#include <math.h>
#include "ap_fixed.h"

typedef ap_ufixed<1, 1> int1;
typedef ap_ufixed<2, 2> int2;
typedef ap_ufixed<3, 3> int3;
typedef ap_ufixed<4, 4> int4;
typedef ap_ufixed<5, 5> int5;
typedef ap_ufixed<6, 6> int6;
typedef ap_ufixed<7, 7> int7;
typedef ap_ufixed<8, 8> int8;
typedef ap_ufixed<9, 9> int9;
typedef ap_ufixed<10, 10> int10;
typedef ap_ufixed<11, 11> int11;
typedef ap_ufixed<12, 12> int12;
typedef ap_ufixed<13, 13> int13;
typedef ap_ufixed<14, 14> int14;
typedef ap_ufixed<15, 15> int15;
typedef ap_ufixed<18, 18> int18;
typedef ap_ufixed<19, 19> int19;
typedef ap_ufixed<20, 20> int20;
typedef ap_ufixed<22, 22> int22;
typedef ap_ufixed<24, 24> int24;
typedef ap_ufixed<25, 25> int25;

//Signed Fixed Point Variables with 0 fraction bits
typedef ap_fixed<11, 11> sint11;
typedef ap_fixed<12, 12> sint12;
typedef ap_fixed<13, 13> sint13;
typedef ap_fixed<22, 22> sint22;

//Fixed Point with > 0 fraction bits (unsigned)
typedef ap_ufixed<12, 1> fixed_12_1;
typedef ap_ufixed<7, 2> fixed_7_2;
typedef ap_ufixed<22, 11> fixed_22_11;
typedef ap_ufixed<18, 13> fixed_18_13;

void RPU(int1 reset, int1 enable, int8 unit_vel, sint11 data_in,
               int1* sclk,
               int22* final_target_info,
               int1* final_info_valid,
               int3* beamport,
               int10* modulate);

void adc_control(int1 reset, sint11 data_in, int11* xn_index,
                             sint12* xn_value, bool* fft_start, bool* hold,
int1* sclk, bool* fft_record);
```

```
void fft_control(int1 reset, bool fft_record, bool fft_start, int11 xn_index,
sint12 xn_value,
                             int13* xk_abs, int11* xk_index, bool* fft_done);

void fft_absolute(int1 reset, bool fft_done, int11 xk_index, int13 xk_abs,
                                   int13* outA, int13* outB, int13* outC, int13*
outD, bool* abs_done);

void cfar(int1 reset, bool start, int13 inA, int13 inB, int13 inC, int13 inD,
                int13* target_abs, int10* target_pos,
                bool* new_target, bool* complete);

void peak_pairing(int1 reset, bool new_target, int13 target_abs, int10
target_pos,
                              bool complete, bool updown, int8 unit_vel,
                              int20* target_info, bool* info_valid);


#endif
```

Top level function (top.cpp)

```
#include "main_head.h"

void RPU(int1 reset, int1 enable, int8 unit_vel, sint11 data_in,
               int1* sclk, int22* final_target_info,
               int1* final_info_valid, int3* beamport,
               int10* modulate)
{
       //Internal Variables
       static int5 sclk_count = 0; //Count for the ADC clock for the clock
divider

              //Internal Flags (Flags)
              static bool dirchange = false;           //Indicate if the
direction will change
              static bool updown = true;               //Indicates up-sweep
(true) or down-sweep (false), start with up sweep
              static bool fft_start = false;           //Indicates if fft should
start
              static bool hold = false;                //Hold sclk output to adc
(Wait to finish before accepting new data)
              static bool fft_done = false;            //Indicates if fft is
done with computations
              static bool info_valid = false;          //Indicates if the info
is valid after peak pairing
              static bool vel_adjusted = false;  //Indicates if the velocity has
been adjusted due to beam angle
              static bool moddone = false;             //Indicates if tuning
voltage has been updated


              //Clock for VCO Tuning (A)
              static int1 modclock = 0;   // 1-bit clock initially set to 0
              static int6 modcounter = 0; // 6 bit counter for the clock divider
(0 to 49)

              //ADC control and data capture (B)
              static int11 xn_index = 0;
              static sint12 xn_value = 0;

              static int11 xk_index = 0;
              static int13 xk_abs = 0;
              static bool abs_done = false;

                            //CFAR
              static int13 absA = 0;
              static int13 absB = 0;
              static int13 absC = 0;
              static int13 absD = 0;

              static int13 target_abs = 0;
              static int10 target_pos = 0;
              static bool new_target = false;
              static bool cfar_complete = false;
```

```
			//For final_info
		static int20 target_info  = 0;
		static int19 velmulres = 0;
		static int9 velmulfac = 257;

		//UP/DOWN done
		static bool in_done = false;
		static bool fft_record = false;


	if(reset == 1 || enable == 0)		//Synchnorous Reset
	{
		//Main Outputs
		*sclk = 0;
		*final_target_info = 0;
		*final_info_valid = 0;
		*beamport = 1;
		*modulate = 0;


		in_done = false;
		fft_record = false;

		velmulres = 0;
		target_info = 0;
		info_valid = false;
		target_abs = 0;
		target_abs = 0;
		new_target = false;
		cfar_complete = false;

		absA = 0;
		absB = 0;
		absC = 0;
		absD = 0;
		abs_done = false;

		xk_abs = 0;
		fft_done = false;
		xk_index = 0;
		xn_value = 0;
		xn_index = 0;

		fft_start = false;
		fft_record = false;

		in_done = false;

		sclk_count = 0;

		modcounter = 0;
		dirchange = false;
		modclock = 0;
		moddone = false;

		vel_adjusted = false;
		updown = true;		//Start with up sweep
```

```
}
else                    //Reset is 0
{


        if(*final_info_valid == 1)
        {
                *final_info_valid = 0;
                vel_adjusted = false;
        }

        if (modcounter == 49)
        {
                modclock = ~modclock;
                modcounter = 0;
                moddone = false;
        }
        else
        {
                modcounter++;
        }

        if(fft_start && !dirchange)
        {
                dirchange = true;

                if(updown)
                        updown = false;
                else
                {
                        updown = true;
                        *modulate = 0;

                        //Switching the beam ports
                        switch((*beamport).to_int())
                        {
                        case 1: *beamport = 4;
                                      break;
                        case 2: *beamport = 1;
                                      break;
                        case 4: *beamport = 2;
                                      break;
                        }
                        in_done = true;
                }
        }
        else if(modclock == 1 && !moddone)
        {
                if(updown)    //Up sweep
                {
                        if(*modulate < 1023)
                               *modulate += 1;      //Increase tuning voltage
                }
                else   //down sweep
                {
                        if(*modulate > 0)
```

```
                                    *modulate -= 1;
                        }

                        moddone = true;              //Tuning voltage has been
updated
                }


                adc_control(reset, data_in, &xn_index, &xn_value,
                                &fft_start, &hold, sclk, &fft_record);

                fft_control(reset,fft_record, fft_start, xn_index, xn_value,
                                &xk_abs, &xk_index, &fft_done);

                fft_absolute(reset, fft_done, xk_index, xk_abs,
                                &absA, &absB, &absC, &absD, &abs_done);

                cfar(reset, abs_done, absA, absB, absC, absD,
                        &target_abs, &target_pos, &new_target, &cfar_complete);

                peak_pairing(reset, new_target, target_abs, target_abs,
cfar_complete,
                                     updown, unit_vel, &target_info, &info_valid);

                //Clock to ADC
                if(sclk_count == 24)
                {
                        if(*sclk == 1)
                                *sclk = 0;
                        else
                                *sclk = 1;

                        sclk_count = 0;
                }
                else
                {
                        *sclk = 0;
                        sclk_count += 1;
                }

                if(info_valid)
                {
                        switch((*beamport).to_int())
                        {
                        case 1: (*final_target_info).range(21,2) = target_info;
                                        (*final_target_info).range(1, 0) = 2;
        //beamport
                                        *final_info_valid = 1;
                                        break;
                        case 2: velmulres = target_info.range(19, 10) * velmulfac;
                                        (*final_target_info).range(11,2) =
target_info(9, 0);
                                        (*final_target_info).range(1, 0) = 1;
                                        vel_adjusted = true;
                                        break;
                        case 4: velmulres = target_info.range(19, 10) * velmulfac;
                                        (*final_target_info).range(11,2) =
target_info(9, 0);
```

```
                                        (*final_target_info).range(1, 0) = 3;
                                        vel_adjusted = true;
                                        break;
                        }
                }
                if(vel_adjusted && velmulres.range(17, 9) <= 300)
                {
                        (*final_target_info).range(21, 12) = velmulres.range(17,
8);
                        *final_info_valid = 1;
                }
        }
}
```

ADC control (adc_control.cpp)

```
#include "main_head.h"
#include "setWindow.h"


void adc_control(int1 reset, sint11 data_in, int11* xn_index,
                             sint12* xn_value, bool* fft_start, bool* hold,
int1* sclk, bool* fft_record)
{
       //Internal Variables
       static bool sample_read = false;   //Flag to indicate if a sample has been
read

       static sint22 mult_res = 0;        //22 bit data to store the multiplied
value (hamming window)


       if(reset == 1)
       {
              sample_read = false;

              *xn_value = 0;
              mult_res = 0;
              *fft_start = false;

              *hold = false;
              *xn_index = 0;
              *fft_record = false;
       }
       else
       {

              if(*fft_start)
              {
                     *fft_start = false;
                     *hold = false;
              }
              //Capturing data from the adc
              if(*sclk == 1)
              {      if(*xn_index < 1023)
                     {
                            *fft_record = true;
                            mult_res = data_in * window[(*xn_index).to_uint()];
                            *xn_value = mult_res.range(21, 10);

                     }
                     else
                     {
                            mult_res = data_in * window[(2047 -
(*xn_index).to_uint())];
                            *xn_value = mult_res.range(21, 10);
                     }

                     if(*xn_index == 2047)       //If 2048 samples are collected
                     {
                            *xn_index = 0;
```

```
                                        *hold = true;
                                        *fft_start = true;
                                        *fft_record = false;

                                }
                                else
                                {
                                        *xn_index += 1;
                                }
                        }
                }
}
```

FFT Control (fft_control.cpp)

```
#include "main_head.h"
#include "fft_head.h"

void fft_control(int1 reset, bool fft_record, bool fft_start, int11 xn_index,
sint12 xn_value, int13* xk_abs, int11* xk_index, bool* fft_done)
{
	//Internal Variables
	static cmpxDataIn x_in[FFT_LENGTH];
#pragma HLS STREAM variable=x_in depth=2048
	static cmpxDataOut x_out[FFT_LENGTH];
#pragma HLS STREAM variable=x_out depth=2048

	const bool fft_direction = true;
	const double sc = ldexp(1.0, 10);
	static bool send_data = false;


	if(reset == 1)
	{
		*fft_done = false;
		*xk_index = 0;
		*xk_abs = 0;

		send_data = false;

	}
	else
	{
		if(send_data)
			*xk_index += 1;
		else if(*xk_index == 2047)
			*xk_index = 0;

		if(*fft_done)
			*fft_done = false;

		if(!fft_start && fft_record)
		{
			x_in[xn_index.to_uint() - 1] = std::complex<double>
(xn_value, 0);
		}
		else
		{
			//Insert the last value then start FFT
			x_in[2047] = std::complex<double> (xn_value, 0);
			bool ovflo;

			fft_top(fft_direction, x_in, x_out, &ovflo);
			*fft_done = true;
			send_data = true;
			*xk_index = 0;		//Send data starting with xk_index = 0;
		}

		if(send_data)
		{
```

```
                    //Once FFT is done, send the absolute value of the output
forward one by one
                            if(*xk_index > 1023)        //Ignoring the first 1024
values
                            {
                                   double tempR, tempI, tempAbs;
                                   tempR =
x_out[(*xk_index).to_uint()].real().to_double();
                                   tempI =
x_out[(*xk_index).to_uint()].imag().to_double();
                                   tempAbs = sc*sqrt(tempR*tempR + tempI*tempI);
                                   *xk_abs = round(tempAbs);
                            }

                            if(*xk_index == 2047)
                            {
                                   send_data = false;
                            }
              }
       }
}
```

FFT Absolute (fft_absolute.cpp)

```
#include "main_head.h"


void fft_absolute(int1 reset, bool fft_done, int11 xk_index, int13 xk_abs,
                                   int13* outA, int13* outB, int13* outC, int13*
outD, bool* abs_done)
{
       //Internal Variables
       static bool storing = false;

       static int13 data_buffer[4];
       static int2 count = 0;
       static bool enable = false;


       if(reset == 1)
       {
              count = 0;
              enable = false;

              *outA = 0;
              *outB = 0;
              *outC = 0;
              *outD = 0;
              *abs_done = false;
       }
       else
       {
              if(*abs_done == true)
                     *abs_done = false;

              if(fft_done)
              {
                     enable = true;
              }

              if(enable && xk_index > 1023)
              {
                     if(storing)
                     {

                            data_buffer[count.to_uint()] = xk_abs;

                            if(count == 3)
                            {
                                   count = 0;     //Reset
                                   storing = false;
                            }
                            else
                                   count++;

                     }

                     if(!storing)
                     {
```

```
                    *outA = data_buffer[0];
                    *outB = data_buffer[1];
                    *outC = data_buffer[2];
                    *outD = data_buffer[3];

                    *abs_done = true;
                    storing = true;
                }
            }
            if(xk_index == 2047)
            {
                enable = false;
            }
        }
}
```

CFAR (cfar.cpp)

```
#include "main_head.h"


void cfar(int1 reset, bool start, int13 inA, int13 inB, int13 inC, int13 inD,
                int13* target_abs, int10* target_pos,
                bool* new_target, bool* complete)
{
	//Internal Variables
	static int13 buffer[32];

	static int10 indexa = 0;
	static int5 indexb = 0;
	static int5 indexc = 0;

	static int15 avgL = 0;
	static int15 avgR = 0;
	static int15 avg = 0;
	static bool cfar_done = false;
	static bool start_cfar = false;
	static int2 cfar_step = 0;

	static int18 T = 0;
	static int5 K = 0;
	static int13 CUT = 0;


	if(reset == 1)
	{
		*target_pos = 0;
		*target_abs = 0;
		*new_target = false;
		*complete = false;

		indexa = 0;
		indexb = 0;
		indexc = 0;

		avgL = 0;
		avgR = 0;
		avg = 0;
		cfar_done = false;
		start_cfar = false;
		cfar_step = 0;

		T = 0;
		K = 0;
		CUT = 0;
	}
	else
	{
		if(*complete)
		{
			indexa = 0;
			indexb = 0;
			start_cfar = false;
```

```
		}
		else if(start_cfar)
		{
			if (cfar_done)
			{
				start_cfar = false;
				indexb = 0;
				indexc = 0;
			}
			else
			{
				start_cfar = true;
				indexb = 31;
			}
		}

		if(start && !start_cfar)
		{
			buffer[indexb.to_uint()] = inA;
			buffer[indexb.to_uint() + 1] = inB;
			buffer[indexb.to_uint() + 2] = inC;
			buffer[indexb.to_uint() + 3] = inD;

			if(indexa == 1020)
				indexa = 1023;
			else
				indexa += 4;

			if(indexb == 28)
			{
				indexb = 0;
				indexc = 0;
				start_cfar = true;
			}
			else
				indexb += 4;
		}

		if(*complete)
			*complete = false;
		else if(cfar_done || *new_target)
		{
			indexc = 0;
			cfar_done = false;
			*target_abs = 0;
			*target_pos = 0;
		}
		if(start_cfar)
		{
			switch(cfar_step.to_uint())
			{
			case 0:
					*new_target = false;
					if( indexa >= 0 && indexa <= 511)
						K = 20;
					else if( indexa >= 512 && indexa <= 851)
						K = 17;
					else if( indexa >= 852)
```

```
                                                K = 16;

                                        if(indexc < 6)
                                        {
                                                avgR = buffer[indexc.to_uint()+3] +
                                                        buffer[indexc.to_uint()+4] +
                                                        buffer[indexc.to_uint()+5] +
                                                        buffer[indexc.to_uint()+6];

                                                avgL = buffer[indexc.to_uint()+3] +
                                                        buffer[indexc.to_uint()+4] +
                                                        buffer[indexc.to_uint()+5] +
                                                        buffer[indexc.to_uint()+6];
                                        }
                                        else if(indexc < 25)
                                        {
                                                avgR = buffer[indexc.to_uint()+3] +
                                                        buffer[indexc.to_uint()+4] +
                                                        buffer[indexc.to_uint()+5] +
                                                        buffer[indexc.to_uint()+6];

                                                avgL = buffer[indexc.to_uint()-3] +
                                                        buffer[indexc.to_uint()-4] +
                                                        buffer[indexc.to_uint()-5] +
                                                        buffer[indexc.to_uint()-6];
                                        }
                                        else
                                        {
                                                avgR = buffer[indexc.to_uint()-3] +
                                                        buffer[indexc.to_uint()-4] +
                                                        buffer[indexc.to_uint()-5] +
                                                        buffer[indexc.to_uint()-6];

                                                avgL = buffer[indexc.to_uint()-3] +
                                                        buffer[indexc.to_uint()-4] +
                                                        buffer[indexc.to_uint()-5] +
                                                        buffer[indexc.to_uint()-6];
                                        }
                                        cfar_step = 1;
                                        break;
                                case 1:
                                        avg = avgR.range(14,3) + avgL.range(14, 3) +
1;
                                        cfar_step = 2;
                                        break;
                                case 2:
                                        T = avg*K;
                                        CUT = buffer[indexc.to_uint()];
                                        cfar_step = 3;
                                        break;
                                case 3:
                                        if( CUT.to_int() > (T.range(14, 2).to_int())
&& CUT > 7)
                                        {
                                                *new_target = true;
                                                *target_abs = CUT;
                                                *target_pos = indexa + indexc - 30;
                                                K = 0;
```

```
                                }
                                if(indexc == 31)
                                        cfar_done = true;

                                if(indexc == 31 && indexa == 1023)
                                        *complete = true;


                                indexc += 1;
                                cfar_step = 0;
                                break;
                        }//End of switch
                }       //end of star_cfar
        }
}
```

Peak Pairing (peak_pairing.cpp)

```
#include "main_head.h"

void peak_pairing(int1 reset, bool new_target, int13 target_abs, int10 
target_pos,
                              bool complete, bool updown, int8 unit_vel,
                              int20* target_info, bool* info_valid)
{
       //Internal Variables
       static int13 abs_bufup[8], abs_bufdown[8];
       static int10 pos_bufup[8], pos_bufdown[8], posa, posb;
       static bool upfill = false, downfill = false, start_pairing = false;
       static bool pairing_done = false, stb = false, faster = false, updone = 
false;
       static int3 paircount = 0, indexup = 0, indexdown = 0, tmpindex = 0, 
count = 0;
       static fixed_7_2 vel_fac = 3.40625;
       static fixed_18_13 velocity = 0;
       static fixed_12_1 range_fac = 0.0927734375;
       static fixed_22_11 range = 0;
       static int2 st_pp = 0;
       static int14 absa = 0, absb = 0, absc = 0;
       static int11 sum_pos = 0, diff_pos = 0;
       static bool start_v = false;


       if(reset == 1)
       {
        count = 0;
        paircount = 0;
        abs_bufup[0] = 0;
        pos_bufup[0] = 0;
        abs_bufdown[0] = 0;
        pos_bufdown[0] = 0;
        upfill = false;
        downfill = false;
        start_pairing = false;
        updone = false;

        start_v = false;

        *target_info = 0;
              *info_valid = false;
              pairing_done = false;
              indexup = 0;
              indexdown = 0;
              tmpindex = 0;
              vel_fac = 3.40625;
              range_fac = 0.0927734375;
              st_pp = 0;
              stb = false;
              posa = 0;
              posb = 0;
              absa = 0;
              absb = 0;
              absc = 0;
```

```
            sum_pos = 0;
            diff_pos = 0;
            faster = false;
            velocity = 0;
            range = 0;
        }
        else
        {
            if(complete)
            {
                downfill = false;
            }

            if(new_target)
            {
                start_v = true;
            }
            if(pairing_done)
            {
                start_pairing = false;
                paircount = 0;
                updone = false;
                upfill = false;
            }
            if(updone && !downfill)
            {
                count = 0;
                if(updown == false && updone == 1)
                    start_pairing = true;

            }

            if(!upfill && start_v && !updone)
            {
                if(count == 0 && target_pos > 4)
                {

                    abs_bufup[count.to_uint()] = target_abs;
                    pos_bufup[count.to_uint()] = target_pos;
                    count += 1;
                }
                else if(count >= 1)
                {
                    if(target_pos == (pos_bufup[count.to_uint() - 1] +
1))
                    {
                        if(target_abs > abs_bufup[count.to_uint() -
1])
                        {
                            abs_bufup[count.to_uint() - 1] =
target_abs;
                            pos_bufup[count.to_uint() - 1] =
target_pos;
                        }
                    }
                    else
                    {
                        abs_bufup[count.to_uint()] = target_abs;
```

```
                                    pos_bufup[count.to_uint()] = target_pos;
                                    count += 1;


                                    if(count == 7)

                                    {
                                           upfill = true;
                                           start_v = false;
                                           paircount = count;
                                           updone = true;
                                    }
                             }
                      }
               }
               else if(start_v && !downfill && upfill)
               {
                      if(count == 0 && target_pos > 4)
                      {
                             abs_bufdown[count.to_uint()] = target_abs;
                             pos_bufdown[count.to_uint()] = target_pos;
                             count += 1;

                      }
                      else if (count >= 1)
                      {

                             if(target_pos == pos_bufdown[count.to_uint() - 1] +
1)

                             {
                                    if(target_abs > abs_bufdown[count.to_uint() -
1])

                                    {
                                           abs_bufdown[count.to_uint() - 1] =
target_abs;

                                           pos_bufdown[count.to_uint() - 1] =
target_pos;

                                    }
                             }
                             else
                             {
                                    abs_bufdown[count.to_uint()] = target_abs;
                                    pos_bufdown[count.to_uint()] = target_pos;
                                    count += 1;
                                    if(count == 7)

                                    {
                                           downfill = true;
                                           start_v = false;
                                    }
                             }
                      }
               }

               if(pairing_done)
               {
                      *target_info = 0;
                      *info_valid = false;
```

```
                pairing_done = false;
                indexup = 0;
                indexdown = 0;
                tmpindex = 0;
                st_pp = 0;
                stb = false;
                posa = 0;
                posb = 0;
                absa = 0;
                absb = 0;
                absc = 0;
                sum_pos = 0;
                diff_pos = 0;
                faster = false;
                velocity = 0;
                range = 0;


        }
        else if(start_pairing && indexdown <= (paircount-1))
        {
                *target_info = 0;
                *info_valid = 0;

                switch(st_pp.to_int())
                {
                case 0:
                                if(pos_bufup[indexup.to_uint()] >
pos_bufdown[indexdown.to_uint()])
                                        posa = pos_bufup[indexup.to_uint()] -
pos_bufup[indexdown.to_uint()];
                                else
                                        posa = pos_bufup[indexdown.to_uint()] -
pos_bufup[indexup.to_uint()];

                                if(abs_bufup[indexup.to_uint()] >
abs_bufdown[indexdown.to_uint()])
                                        absa = abs_bufup[indexup.to_uint()] -
abs_bufup[indexdown.to_uint()];
                                else
                                        absa = abs_bufup[indexdown.to_uint()] -
abs_bufup[indexup.to_uint()];


                                if(abs_bufup[indexup.to_uint()] >
abs_bufdown[tmpindex.to_uint()])
                                        absa = abs_bufup[indexup.to_uint()] -
abs_bufup[tmpindex.to_uint()];
                                else
                                        absa = abs_bufup[tmpindex.to_uint()] -
abs_bufup[indexup.to_uint()];


                                if(indexup < paircount - 1)
                                {
                                        if(abs_bufup[indexup.to_uint() + 1] >
abs_bufdown[tmpindex.to_uint()])
```

```
                                                absc =
abs_bufup[indexup.to_uint() + 1] - abs_bufdown[tmpindex.to_uint()];
                                        else
                                                absc =
abs_bufdown[tmpindex.to_uint()] - abs_bufup[indexup.to_uint() - 1];
                                }
                                else
                                        absc = 8191;

                                if(indexup < paircount - 1)
                                {
                                        if(pos_bufup[indexup.to_uint() + 1] >
pos_bufdown[indexdown.to_uint()])
                                                posb =
pos_bufup[indexup.to_uint() + 1] - pos_bufdown[indexdown.to_uint()];
                                        else
                                                posb =
pos_bufdown[indexdown.to_uint()] - pos_bufup[indexup.to_uint() + 1];
                                }
                                else
                                        posb = 1023;
                                st_pp = 1;
                                break;
                        case 1:
                                if(posa < 84 && posa <= posb)
                                {
                                        if(absa <= absb && absa <= absc)
                                        {
                                                tmpindex = indexdown;
                                        }
                                }
                                if(indexdown == paircount - 1)
                                {
                                        st_pp = 2;
                                }
                                else
                                {
                                        indexdown += 1;
                                        st_pp = 0;
                                }
                                break;
                        case 2:
                                indexdown = 0;
                                sum_pos = pos_bufup[indexup.to_uint()] +
pos_bufdown[tmpindex.to_uint()];

                                if(pos_bufdown[tmpindex.to_uint()] > 0)
                                {
                                        if(pos_bufup[indexup.to_uint()] >
pos_bufdown[tmpindex.to_uint()])
                                        {
                                                diff_pos =
pos_bufup[indexup.to_uint()] - pos_bufdown[tmpindex.to_uint()];
                                                faster = false;
                                        }
                                        else
                                        {
```

```
                                                diff_pos =
pos_bufdown[tmpindex.to_uint()] - pos_bufup[indexup.to_uint()];
                                                faster = true;
                                        }
                                        st_pp = 3;


                                }
                                else
                                {
                                        if(indexup < paircount - 1)
                                        {
                                                indexup += 1;
                                                st_pp = 0;
                                        }
                                        else
                                                pairing_done = true;
                                }
                                break;
                case 3:
                                if(!stb)
                                {
                                        if(faster == 0)
                                                velocity = vel_fac * diff_pos;
                                        else
                                                velocity = vel_fac * diff_pos;

                                        range = range_fac * diff_pos;
                                        stb = true;
                                }
                                else
                                {
                                        if(!faster)
                                                (*target_info).range(19, 11) =
unit_vel - velocity.range(13, 5);
                                        else
                                                (*target_info).range(19, 11) =
unit_vel + velocity.range(13, 5);

                                        (*target_info)[10] = velocity[4];
                                        (*target_info).range(9, 0) =
range.range(18, 9);
                                        *info_valid = 1;

                                        tmpindex = 0;
                                        posa = 0;
                                        posb = 0;
                                        absa = 0;
                                        absb = 0;
                                        stb = false;
                                        st_pp = 0;
                                        indexup += 1;

                                        if(indexup == paircount)
                                                pairing_done = true;
                                }
                                break;
                }       //Switch
```

```
            }       //If start_pairing

        }       //If not reset
}       //End of function
```

Window Coefficients (set_window.h)

```
static const ap_ufixed<10,10> window[] = {82,
                     82,
                     .
                     .      //All window coefficients here
                     .
                     1023};
```

Vita Auctoris

NAME:	Siddhant Luthra
PLACE OF BIRTH:	Haryana, India
YEAR OF BIRTH:	1993
EDUCATION:	Bachelor of Applied Science in Electrical and Computer Engineering (Minor: Computer Science) University of Windsor, Windsor, Ontario, Canada 2011 – 2015.
	Master of Applied Science in Electrical and Computer Engineering University of Windsor, Windsor, Ontario, Canada 2015 - 2017