Donal James Conneely

FPGA Implementation of LiDAR Terrain Mapping

System based on High-Level Synthesis

M.E. *Electronic & Computer Engineering Project Thesis*

EE5115 Project Thesis

May 2020

Academic Supervisor: Dr Fearghal Morgan
Co-Assessor: Mr Liam Kilmartin

Abstract (Max 200 words)

The purpose of this project was to develop a Terrain Mapping system on an FPGA using a process known as High-Level Synthesis (HLS). The primary aim of this project is to demonstrate the capabilities of HLS, to encode complex systems onto FPGAs. The project consists of a single PYNQ-Z2 board connected to a LiDAR sensor which performs the Terrain Mapping calculations using IP Blocks created through the HLS process. The system constructs a map of the surrounding environment and displays the map using an external client running on a computer. The FPGA system communicates with the external client using Ethernet.

The project is based around the use of a SLAM algorithm, which is an algorithm relating to the problem of constructing a map of an unknown environment while simultaneously calculating the system's location. For this project, the system focuses on the use of the tinySLAM algorithm which is a low complexity form of SLAM. The system was correctly encoded onto the FPGA board and generated the desired map. The system was compared to a standard coding implementation and was found to be significantly faster while operating on the same FPGA device.

Declaration

I declare that this progress report is my original work except where stated:

Date	**Signature**
20h of May 2020	

Acknowledgements

The author would like to acknowledge the contributions of my project supervisor Dr Fearghal Morgan. I would also like to acknowledge my co-assessor Mr Liam Kilmartin, and the Lab technicians Mr Myles Meehan & Mr Martin Burke.

Table of Figures

Table of Contents

Table of Acronyms

- HLS: High-Level Synthesis.
- SLAM: Simultaneous Localization And Mapping.
- FPGA: Field Programmable Gate Array.
- HDL: Hardware Description Language.
- LiDAR: Light Detection And Ranging.
- SOC: Systems on Chip.
- AXI: Advanced eXtensible Interface.
- PS: Processing System.
- PL: Programmable Logic.
- DMA: Direct Memory Access.
- LFSR: Linear Feedback Shift Register.
- SDK: Software Development Kit.

Table of Equations

Section 1. Introduction

The purpose of this project is the development of a Terrain Mapping system on an FPGA using a process known as High-Level Synthesis (HLS). The goal of this project is to demonstrate the capabilities of HLS, to encode complex systems onto FPGAs. This project is designed to construct a map of the surrounding environment and then display the map using an external client running on a Laptop or Desktop computer. The FPGA system communicates with the external client using Ethernet communication.

In recent years, the electronics industry has continuously developed new methods to increase computer efficiency, precision, and flexibility. With this increase in computing power, the application of robotic systems has expanded and thus, the field of autonomous systems has become increasingly important. As the demand for autonomous robots has risen, so too has the demand for their increased understanding of the world around us. Therefore, different perception systems have been developed for these devices to allow them to accurately understand the world and react accordingly. Of these different applications, the area of Terrain perception greatly improves the performance of autonomous machinery, allowing machinery to correctly adapt to the environment they encounter. Terrain Mapping is a method of recording the perceived terrain and using this data to construct a "map" of the perceived environment.

In the field of computer perception, devices must have some method of "perceiving" or "viewing" the world around them. To do this, systems must be designed to operate using a variety of different sensors in order to perceive different elements of the world. One type of sensor commonly used is a remote sensor, one the most common of these being the **Li**ght **D**etection **A**nd **R**anging (**LiDAR**) sensor. Remote Sensing can be defined as *"the science of obtaining information about objects or areas from a distance, typically from aircraft or satellites"* [1]. LiDAR is a specific form of remote sensing, in which the system uses light in the form of a laser to measure distances between the sensor and an object. LiDAR is particularly useful because of its ability to provide high-resolution measurements of 3D objects.

As shown by numerous studies, such as *FPGA based accelerated 3D affine transform for real-time image processing applications* [2] and *Fast Real-Time LIDAR Processing on FPGAs* [3], Field Programmable Gate Arrays (FPGAs) have quickly become one of the primary implementation platforms of computer perception applications. This is often due to the complexity of the systems involved having processing requirements better suited to custom hardware implementations, which FPGAs can implement, versus general-purpose software processors. Previously, FPGAs were typically programmed using Hardware Description Languages (HDLs); however, over the last number of years, many of the inefficiencies of HDLs have been brought into focus. Therefore, an alternative method was developed: High-Level Synthesis (HLS). HLS is a method of converting more standard high-level programming languages, such as C/C++, into HDLs for FPGA development. While this does reduce some of the customization potential of standard HDLs, it allows for increased speed in both programming and error correction.

This report details the development and implementation of a Terrain Mapping system on an FPGA using HLS programming. The project itself is intended to demonstrate the capabilities of HLS as well as compare this FPGA based implementation to the to a pure software implementation on a Microcontroller. The remainder of the report breaks down as follows:

- Section 2 reviews the relevant Literature for Terrain Mapping and HLS.
- Section 3 presents a brief description of the project Architecture and Societal Impacts.
- Section 4 reviews the Project Planning and Development.
- Section 5 details the standard High-Level Synthesis development procedure.
- Section 6 details the code of the tinySLAM algorithm and development of a Python version of the code. This section also details the use of the LiDAR system.
- Section 7 deals with the Integration of the tinySLAM algorithm into HLS methodology.
- Section 8 covers the Results of the system and Comparisons between HLS and pure Python implementations.
- Section 9 details Possibilities for Future work.
- Section 10 is the Conclusions of the project.

An illustration of the system layout of the project can be seen below (***Figure 1.1***).

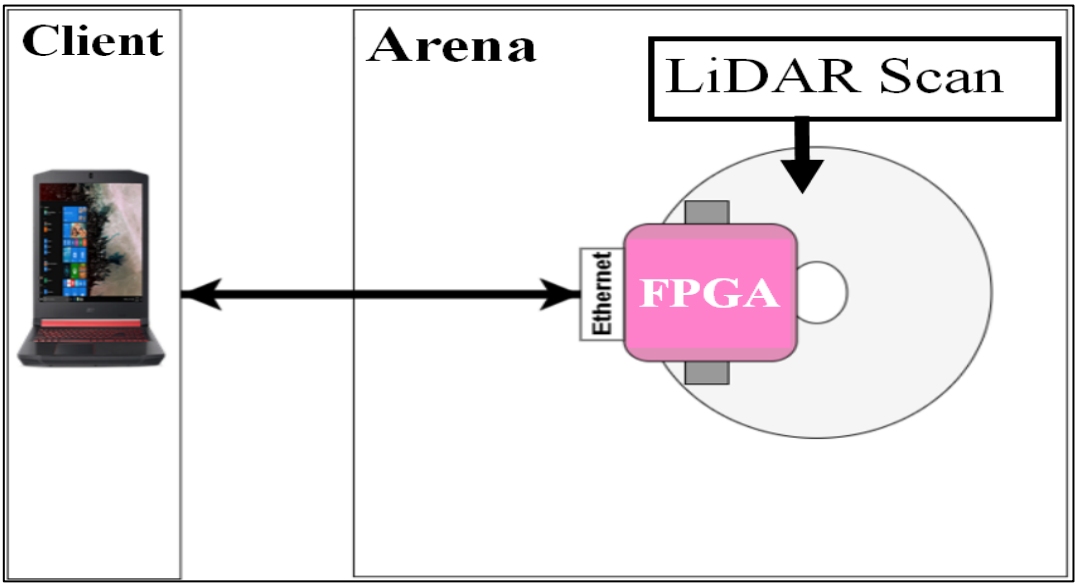


Figure 1.1 System Diagram

Section 2. Literature Review

Section 2.1 Introduction

The purpose of this literature review is to provide background information on some of the technologies incorporated into this project. This section will also detail many of the documents and research methods used in this project. By understanding the key concepts behind Terrain Mapping, it will be possible to develop terrain mapping devices using new, and or different, technologies. This, in turn, will enable the development of multiple terrain mapping applications.

The remainder of the chapter details the Terrain Mapping methods to be used in this project along with an in-depth look at the methodology of HLS.

Section 2.2 Terrain Mapping and Computer Perception

Section 2.2.1 Introduction to Computer Perception

Over the last number of years, computer perception has become an increasingly important data analysis technique. From consumer devices to high-end machinery the field of real-time computer perception has become an important area of development in the electronics industry. Computer Perception, also known as Machine Perception, is the technology which enables devices to sense their surrounding environment. Machine Perception relates to systems with the ability to infer something about the real-world based on data gathered through the attached hardware. The field of Machine Perception can be defined as *"a term that is used to identify the capability of a computer system to interpret data in a manner that is similar to the way humans use their senses to relate to the world around them."* [4].

The field of machine perception includes multiple types of sensors and thus multiple fields of analysis. These include computer vision, machine hearing, machine touch, and machine smell. Computer Vision is the most widely known of these and is used in a variety of fields with many different applications. The field of computer vision relates to a system which takes in an image or video as an input and uses this image to infer something about the image's contents. Computer Vision can be defined as a *"Field of robotics in which programs attempt to identify objects represented in digitized*

images provided by video cameras, thus enabling robots to "see"" [2]. Initially, the project was to be designed using a computer vision-based approach; however, the terrain mapping algorithm involved with a computer vision-based approach was deemed too costly, and inefficient for implementation through High-Level Synthesis and thus, alternative methods were explored.

Upon further investigation, it was decided that the system would instead utilise remote sensing technology. Remote sensing is the field of electronics which involves obtaining data about objects or areas from a distance. Of these types of sensors **L**ight **D**etection **A**nd **R**anging (LiDAR) sensor, is one of the most widely used. LiDAR is a specific form of remote sensing which *"uses light in the form of a pulsed laser to measure ranges (variable distances)"* [5]. In LiDAR sensor the sensor emits a laser which travels to the desired object or area. The laser is then reflected back to the source and the time taken for the laser to be reflected is then used to calculate the distance.

Terrain Mapping is a specific application of computer perception systems. While general machine perception is focused on the observation of the world in general, terrain mapping relates to the perception of the terrain an autonomous system would need to navigate. Terrain Mapping is a method of recording the perceived terrain and using this data to construct a "map" of the perceived environment. Such applications can be useful for autonomous systems which are designed to operate without user input.

Section 2.2.2 Terrain Mapping Methods

Terrain mapping has become a common form of computer perception today. As we develop new autonomous systems it is important that these systems can understand the terrain which they encounter. This understanding allows machinery to react appropriately to the environment, whether that be obstacle avoidance or in cases of object identification. It is important that these autonomous machines can reliably perform these tasks in order to match the desired application. A variety of different studies have been performed into methods of terrain mapping.

Section 2.2.2.1 Mean Shift Methodology

Initial research of this project considered the use of *Terrain perception for a reconfigurable biomimetic robot using monocular vision* [6] and *Terrain Perception in a Shape Shifting Rolling-Crawling Robot* [7]. These reports, written by the same authors, both relate to the development of a terrain mapping robot designed to mimic the biometrics of a spider. For the purposes of this project, we were only interested in the terrain mapping technique used rather than the specifications relating to the mechanical design of the spider-based robot. In these reports the terrain perception system was based on a mean shift algorithm. The mean shift algorithm is based on "*a simple iterative procedure that shifts each data point to the average of data points in its neighbourhood*" [8]. This means that the mean shift method relates to the calculation of maxima of a density function from a discrete sample data set. The function loops iteratively calculating the difference between the current output and the previous output until the output array converges. A simplified version of the mean shift equation can be seen below (Equation 2.1 & Equation 2.2).

$$m(x) = \frac{\sum_{s\varepsilon S} K(s-x)s}{\sum_{s\varepsilon S} K(s-x)}$$

Equation 2.1 Sample Mean

$$mean\ shift = m(x) - x$$
$$x \leftarrow m(x)$$

Equation 2.2 Mean Shift

Where $K(x - y)$ is the Kernel Function, which determines the weight of nearby data points in the estimation of the mean, and S is the finite data set the system operates on. The system then iterates through this equation multiple times, causing the data to converge to the mode value. The convergence relates to the bandwidth of the signal, also known as the smoothness parameter. As stated in the report "*as one increases the value of this smoothness parameter, the convergent mode becomes more global, that is, it smoothes out local perturbations"* [6]. However, this system utilises a process known as Mean Shift Segmentation. In Mean Shift Segmentation the system utilises the Mean Shift in order to perform filtering of the image. After applying the filter, all convergence points are found, and clusters are built

from them. The system uses the smoothness factor to group convergence points together to determine local modes or cluster areas, i.e. areas that are similar, in order to group elements of the terrain together. This results in a final terrain perception algorithm as seen below (Equation 2.3 & Equation 2.4) along with an image of the result from the report (***Figure 2.1***).

$$\mathrm{P}(\mathrm{M}_s) = \frac{1}{|\Omega_T|} \sum_{s_t \in \Omega_T} K_H \left(\mathrm{M}_{s_t} - \mathrm{M}_s\right)$$

Equation 2.3 Pixel Value

$$\mathrm{T}_{th} = \min \mathrm{P}(\mathrm{M}_{s_t}); s_t \in \Omega_T$$

Equation 2.4 Threshold

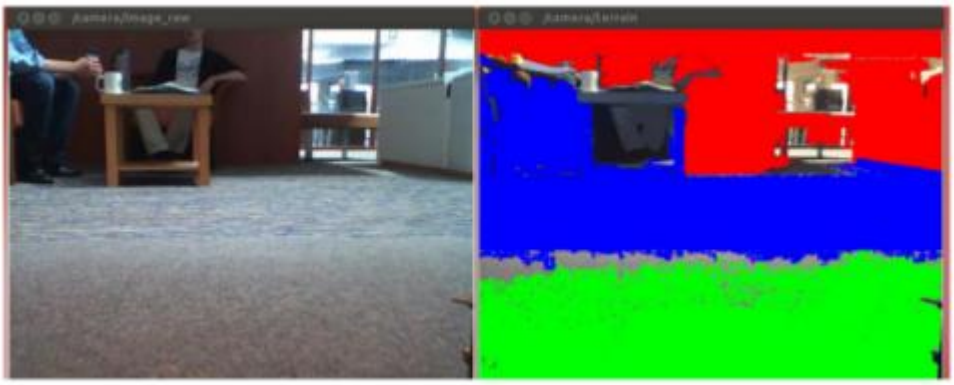


Figure 2.1 Mean Shift Terrain Perception

Upon further investigation; however, it was found that this form of terrain perception was only adequate for determining the quality of the terrain directly in the robot's line of sight. The data from this type of algorithm was not suitable for creating a map of the environment. Therefore alternative methods were investigated.

Section 2.2.2.2 <u>TinySLAM</u>

Following the decision to replace the use of Mean Shift it was decided that the system would move to the use of a Simultaneous Localization And Mapping (SLAM) algorithm can be used. These algorithms relate to the problem of constructing a map of an unknown environment while simultaneously keeping track of the location of the system within said environment. In typical mapping

systems deriving a location or a map can be easy, provided one of the two is already known: Localization relates to inferring location given a map, while Mapping involves the development of a map given locations. However, in a SLAM problem neither the location or the map is known, resulting in a "chicken-or-egg" style problem.

There are multiple types of SLAM algorithms using a variety of different sensors. Initial research focused on Computer Vision or camera-based SLAM algorithms such as FastSLAM [9] and ORBSLAM [10]. Specifically, the research focused on the use of the Monocular, i.e. single camera, SLAM detailed in *A comparison of monocular and stereo visual FastSLAM implementations* [11] and *ORB--SLAM: A Versatile and Accurate Monocular SLAM System* [10]. Upon further investigation, however, it was found that these algorithms would be both too expensive to implement and that they would be too large to implement onto systems with limited resources such as an FPGA. Therefore, it focused shifted to the methods detailed in *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12].

The tinySLAM algorithm is a low complexity implementation of SLAM designed for use on systems with limited resources. The tinySLAM algorithm is a Laser-based version of SLAM using LiDAR to scan for new elements of the map. This system is based on C Code which is particularly useful as the Vivado HLS [13] tool is designed for parsing code from C, C++, System C or OpenCL API C kernel code. The algorithm, therefore, could be easily implemented through HLS with the correct modifications. The tinySLAM algorithm itself breaks down into three primary elements. Firstly, there is the ***Map_Init*** phase. This phase creates a blank map before the system starts scanning the environment.

Secondly, there is the ***Distance_Scan_To_Map*** function. This function is designed to calculate the distance between the LiDAR sensor and the location where the laser is stopped, i.e. the location of an object which stops the laser. This particular function is used to keep track of the system's position as it moves. To do this, the system loops the ***Distance_Scan_To_Map*** method in what is known as a Monte Carlo Method [14]. A Monte Carlo algorithm is a computational algorithm which utilizes random sampling to determine some value. Within the Monte Carlo method of this system, the algorithm selects

a random position within a certain range of the previous position. The system then uses the ***Distance_Scan_To_Map*** to get a value for the system's distance to previously elements of the map. The loops through a series of these random positions and sets the position with the lowest distance value as the current position of the system. A diagram of the process can be seen below (***Figure 2.2***).

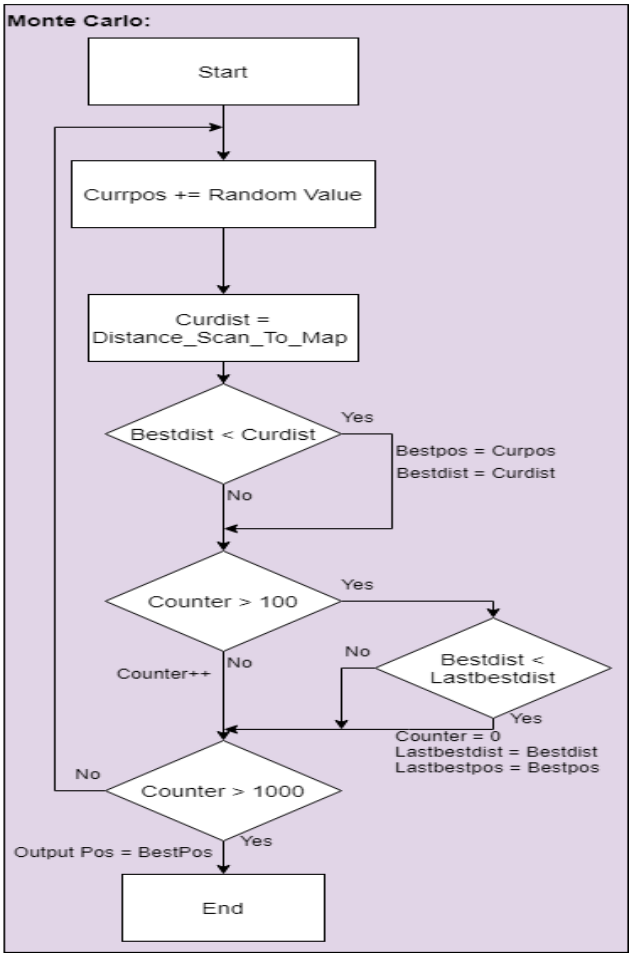


Figure 2.2 Monte Carlo Flow Chart

Finally, there is the ***Map_Update*** function. This function adds the values from the LiDAR scan to the map based on the current position of the system in the map, as determined by the Monte Carlo method.

Section 2.3 FPGAs and High-Level Synthesis (HLS)

Section 2.3.1 Introduction to High-Level Synthesis

Field Programmable Gate Arrays, or FPGAs, are a common electronics device used in a large variety of different applications, both consumer and industrial. An FPGA is a semiconductor-based device built around the use of an array of programmable logic blocks. These logic blocks are connected via a series of reconfigurable interconnects, that allow the logic blocks to simulate the effects of different logic gates. These logic gates can then be used to construct the desired application of the system. These logic blocks also often contain memory elements such as Block Ram (BRAM) or Flip-Flops allowing for data storage. The benefit of coding directly onto an FPGA is their ability to recreate hardware structures allowing for increased energy efficiency along with increased system response times.

FPGAs are commonly programmed using different Hardware Description Languages (HDLs), such as VHDL or Verilog. These HDLs are computer-based languages that describe the hardware of digital systems in text form. These languages are designed to resemble common programming languages, such as C, but are specifically designed for describing hardware structures and logic behaviour. These languages allow for a high degree of system control; however, these systems are often very verbose, time-consuming and can often be difficult to debug for errors. Therefore, an alternative method was developed: High-Level Synthesis (HLS).

HLS is a method of parsing existing high-level programming languages, such as C/C++, into HDLs for FPGA development. HLS operates as a form of design automation, allowing the designer to focus on the specific application requirements instead of focusing on the specific requirements of the target platform. HLS has been found to spend 30% fewer hardware multipliers than the HDL implementations as shown in *Comparative of HLS and HDL Implementations of a Grid Synchronization Algorithm* [15]. While this automation does reduce some of the customization potential of standard HDLs, it allows for the use of the FPGAs hardware implementations without the time delays associated with converting an algorithm to an HDL by hand.

Section 2.3.2 High-Level Synthesis Methodology

For this project, the FPGA must be programmed using HLS. To do so, the system is programmed using the Vivado HLS design suite [13]. This application allows the user to code HLS files, using C/C++ code, which will then be converted into HDL files, such as VHDL or Verilog. The converted code will then be accessible from within the Vivado Design environment [16]. In order to optimize this application, the system is based upon the fundamentals of HLS generation outlined in *Parallel Programming for FPGAs, The HLS Book* [17]. This book details many of the basics of HLS development and goes over details relating to system optimization using HLS. Additionally, the system is also based on the research detailed in *High-level synthesis for FPGAs: code optimization strategies for real-time image processing* [18].

Vivado HLS is a High-Level Synthesis Design Suite, designed to work alongside the existing Vivado Design Suite. High-Level Synthesis creates an RTL (Register Transfer Level) implementation of a design from C, C++, System C or OpenCL API C kernel code. The design is implemented based on defaults and user applied directives. The system takes the developed code and implements the design into VHDL and Verilog, two commonly used HDLs. Vivado HLS is particularly notable for its support of many C/C++ libraries, notably the OpenCV library [19]. OpenCV is an open-source library designed for computer vision applications. This library is designed to assist with the development of computer vision applications such as Terrain Mapping and has been widely used/supported over the last number of years. The system also supports conversion to Semiconductor intellectual property core (IP core) for integration purposes. An image of the system operation can be seen below (***Figure 2.3***).

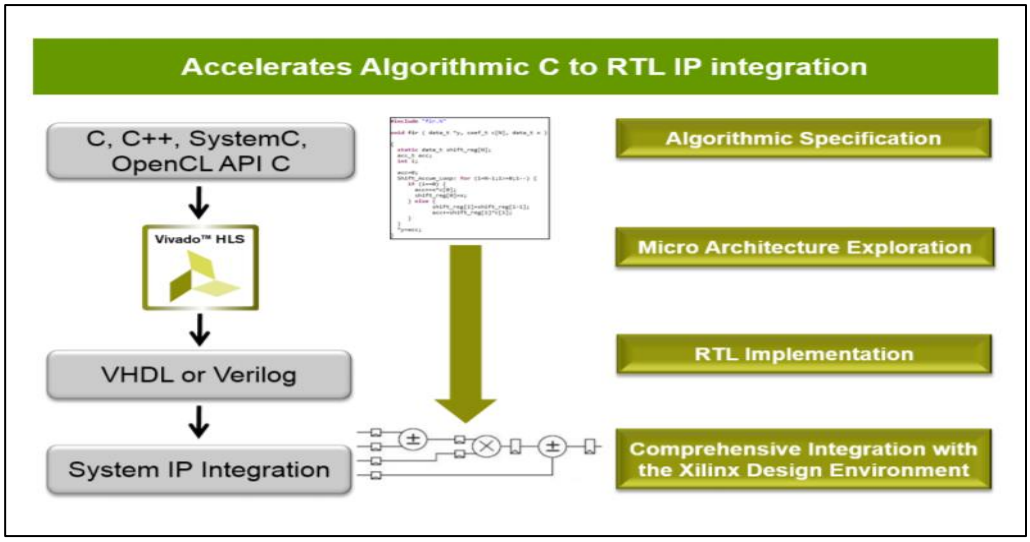


Figure 2.3 Vivado HLS

Based on this understanding further research can be conducted into the field of HLS. The book *Parallel Programming for FPGAs, The HLS Book* [17] was designed as a teaching tool for HLS. As stated in the book: "*this book is a practical guide for anyone interested in building FPGA systems*" and "*Our goal is to give the reader an appreciation of the process of creating an optimized hardware design using HLS*". The goal of said book is to detail the different optimization techniques available to use using Vivado HLS including: Loop Unrolling, System Pipelining, Array Optimization and Co-Simulation. The book itself does not create any new algorithms, instead, the book teaches the user how to code using HLS by using commonly used digital signal applications including Fast-Fourier Transforms (FFTs), Signal Filters, Matrix Multiplication and Sorting Algorithms. Additional optimization techniques have also been identified in *High-level synthesis for FPGAs: code optimization strategies for real-time image processing* [18].

Section 3. Project Description & Societal Impact

Section 3.1 Project Description and System Architecture

The purpose of this project is the development of a Terrain Mapping system on an FPGA using a process known as High-Level Synthesis (HLS). The primary aim of this project is to demonstrate the capabilities of HLS, to allow complex systems to be encoded onto FPGAs more easily. The system consists of an FPGA connected to a remote sensor (LiDAR), which communicates with an external client via Ethernet. The FPGA handles the bulk of the calculations while the client merely acts as a console, displaying the map and allowing the user to Start or Stop the program.

For this system, it was decided that a Xilinx PYNQ-Z2 board should be used as the systems FPGA device. The PYNQ boards are an open-source project developed by Xilinx designed to allow for easy implementation of embedded systems using Systems on Chips (SOCs). The PYNQ boards are designed to be programmed using the Python coding language and libraries. There are two different PYNQ boards, the Z1 and the Z2. While the boards are largely the same, the boards primarily differ based on audio capabilities and expansion headers. In terms of audio the Z1 has an integrated MIC with PWM input, and mono PDM audio out, while the Z2 has a full ADI audio codec with Headphones out, Mic, and line-in. However, given the application requirements, no audio functionality is required, therefore the differences presented in this area are negligible. In terms of the expansion headers, the Z1 has a ChipKit header, while the Z2 provides a 40-pin Raspberry Pi header. While both headers can be useful it was decided that the Raspberry Pi header, and thus the Z2, was superior due to its greater options and higher levels of documentation. Images of the PYNQ-Z2 and PYNQ architectures can be seen below (***Figure 3.1*** & ***Figure 3.2***).

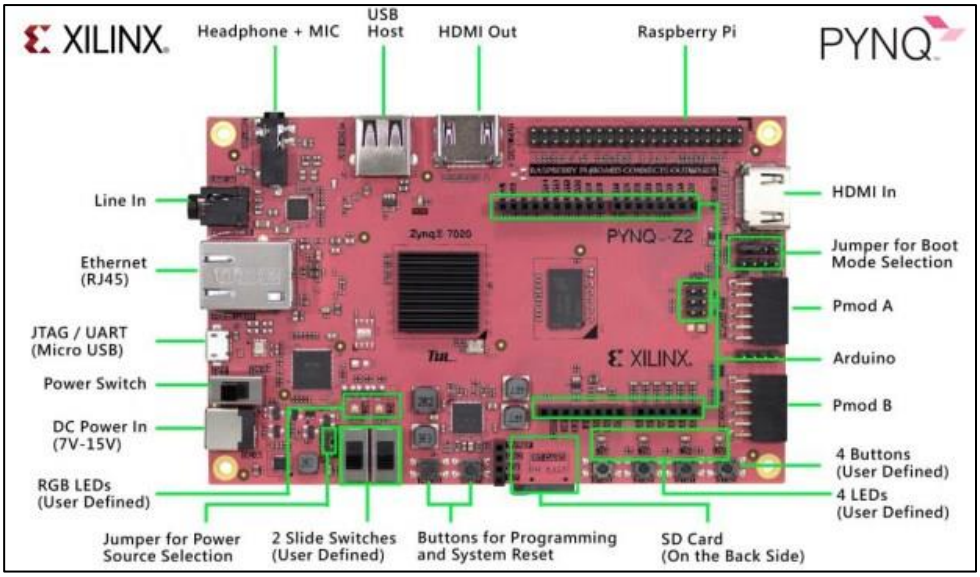


Figure 3.1 PYNQ-Z2 Board

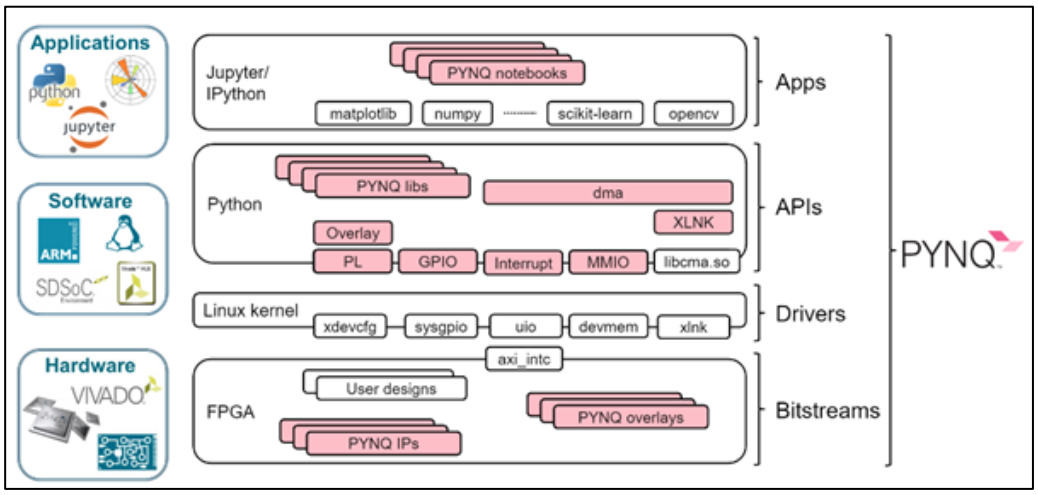


Figure 3.2 PYNQ Architecture

For this project, the FPGA must be programmed using HLS. To do so, the system is programmed using the Vivado HLS design suite [13]. This application allows the user to code HLS files, using C/C++ code, which will then be converted into HDL files, such as VHDL or Verilog. These HDLs can then be used within the Vivado design environment. The system is based upon the fundamentals of HLS generation outlined in *Parallel Programming for FPGAs, The HLS Book* [17]. This book outlines some of the fundamental techniques for HLS development along with examples of real-time data processing using HLS. The code is also planned to incorporate elements from the OpenCV [19] library. The system is designed based on the methodology laid out in *tinySLAM: a SLAM*

Algorithm in less than 200 lines C-Language Program [12] and thus uses a modified version of the C code found in the report.

The programmed FPGA was originally planned to be built into an autonomous robot; however, due to time constraints, this was not possible. As previously stated in Section 2.2.2, the current system is designed around a terrain perception completely based on the use of remote sensing technology and thus the FPGA is connected to an external LiDAR via USB. LiDAR is a common form of remote sensor using light in the form of a laser to measure the distance between the sensor and an object. In the original tinySLAM system [12], the system utilised a highly precise laser scanner called a Hokuyo URG04 laser scanner [20]; however, this scanner was found to be much too expensive for the purposes of this project. Therefore, an alternative known as a Slamtec RPLiDAR A1M8 [21] was chosen. This low-cost LiDAR provides a 360-degree 2D laser scanner with a range of than 6 meters at a frequency 5.5Hz. This LiDAR has already been shown to work well within the limits of a tinySLAM system as seen in *A low-cost indoor mapping robot based on TinySLAM algorithm* [22].

The external client of the project is an external system to the FPGA running on a Laptop or Desktop computer. This system communicates with the FPGA via Ethernet. The external client is designed to handle the display of the map and to allow the user to Start/Stop the program. The system is planned to pass the calculated terrain information to the client allowing the client to construct a map of the "arena". Originally, the client was to be developed as a unique independent system; however, due to time constraints, the client was instead the client run on a Jupyter Notebooks Browser [23]. Jupyter Notebooks is a server-client application that allows editing and running notebook documents via a web browser. Jupyter Notebooks is the supported IDE of the PYNQ boards and runs on the bootable Linux operating system. It supports multiple coding languages including Python, C, and Pearl, and allows for simplified installation of libraries, graph data and annotation of code. A diagram detailing the basic arena structure as well as an overview of the system architecture can be seen below (***Figure 3.3*** & ***Figure 3.4***).

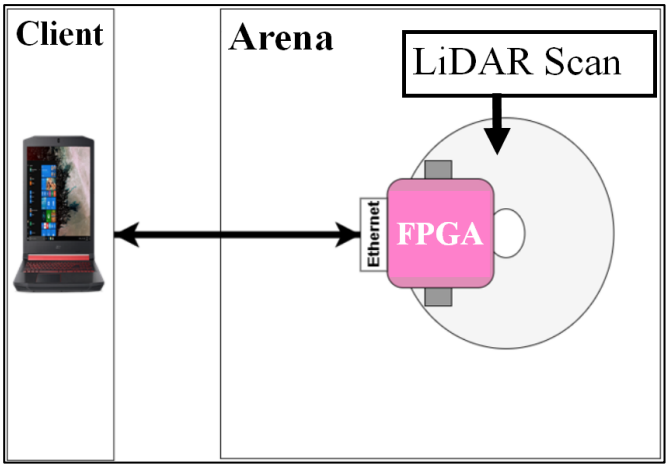


Figure 3.3 Arena Structure

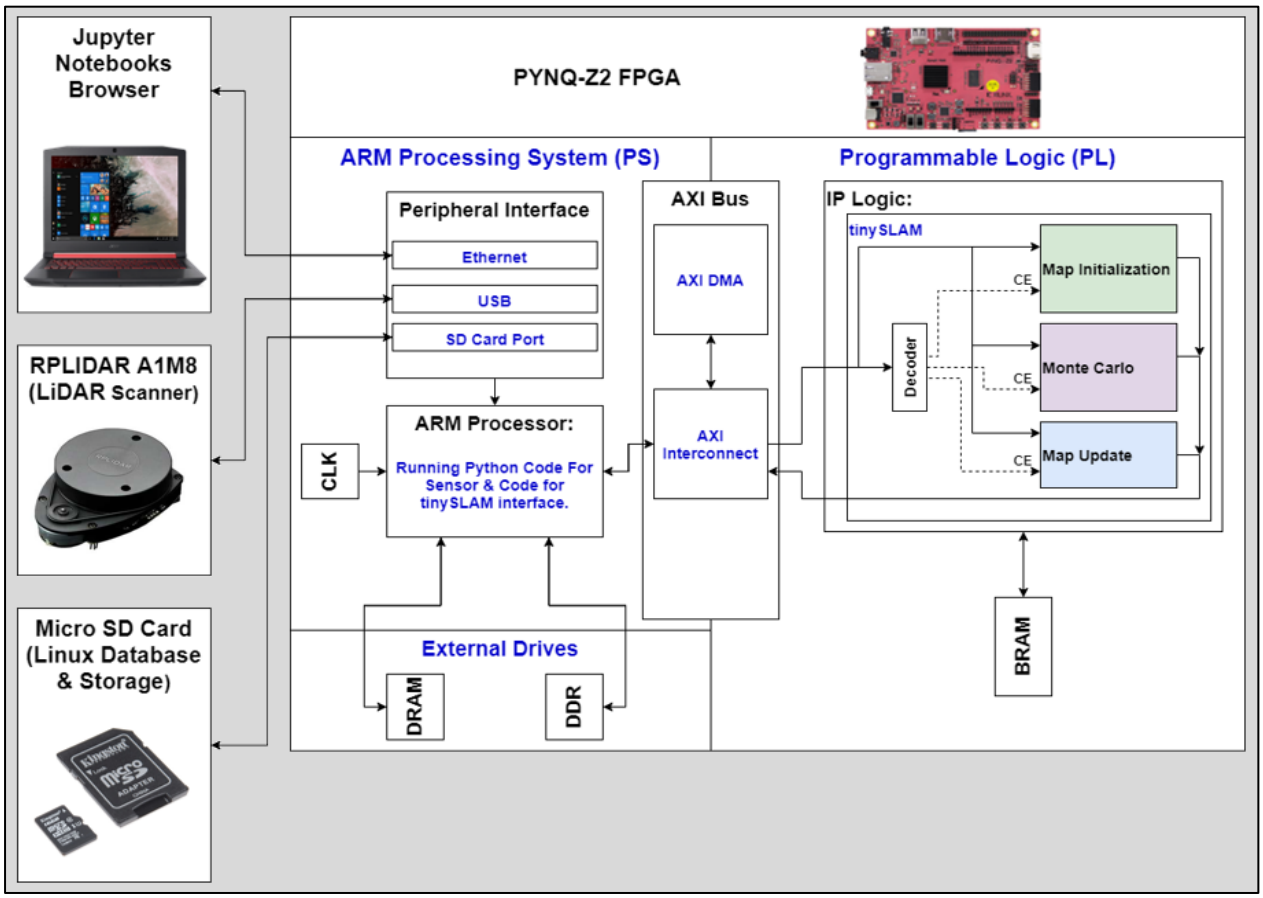


Figure 3.4 System Architecture

Section 3.2 Risk to Life and Limb

As this project deals with an autonomous system with little to no human interaction there are very few dangers involved with the development of such a project. However, it is still important to

consider potential risks brought about by the device. In order to prevent harm to people, it is necessary that all risks be noted and alleviated if necessary.

Firstly, as this system is based on a LiDAR, which is a form of laser-based technology, we must assess the risks involved in the use of such a laser. As noted in the RPLiDAR datasheet [24], this LiDAR qualifies as for Class I laser safety standard, due to its low power laser (< 5mW) and the fact that due to the motorised spinning of the laser (5.5 Hz) the laser emits in a very short time frame. According to the Laser Safety Standards, as defined in IEC-60825 [25], a laser of Class I laser is safe under all conditions of normal use. This means that even under direct viewing with a naked eye, the maximum permissible exposure (MPE) will never be exceeded. Therefore, this laser should never pose any risk to users. However, it is still recommended that some eye protection is worn in case of a fault.

Another risk to consider is the possibility of a tripping hazard in areas of high population. This project utilises a small FPGA and LiDAR system connected to a Laptop or Desktop via an Ethernet cable. Therefore, if a person was unaware of the system's presence, they may walk into it, or the Ethernet cable extending between the two components and potentially injure themselves by tripping. Currently, as the system does not have an autonomous method of motion it must be carried, allowing the user to warn those around it of the hazard. In the case where the system has been developed into a robot, the robot could denote its presence in some way to those nearby. This could be in the form of a sound the robot makes or by introducing a visual queue to note it is active e.g. flashing lights. These signifiers would help inform a passer-by that the robot is potentially obstructing their path.

Finally, we must consider the risk of the use of the maps generated by the system. This project is designed to map a specific area of space. Therefore, if the map were to be faulty in any way a user may be unaware of obstacles they encounter when entering the area. While this project is focused around a small indoor setting, if this design were to be scaled and used in outdoor terrain environments any errors in the mapping process could lead to dangers for a person using the described map, e.g. the user is unaware of tripping hazards and other obstacles or the slight inaccuracies of the mapping cause the user to take an incorrect route.

Section 3.3 Broader Impacts

In relation to the Broader Impacts on society, the project is designed so that it can be used in multiple situations. The primary goal of the project is to demonstrate the capabilities of High-Level Synthesis based FPGA implementations versus system implementations on more standard software platform such as a microcontroller.

Firstly, this device has the potential to be used at home in a consumer capacity. Over the last number of years, the popularity of autonomous robotic systems has increased rapidly, both in industry and at a consumer level. As the demand for autonomous robots has risen, so too has the demand for their increased understanding of the world around us. Terrain Mapping greatly improves the performance of autonomous machinery, allowing machinery to correctly adapt to the environment they encounter. Therefore, development of the exploration of this HLS vs microcontrollers in relation to this technology may serve to improve the performance of these consumer products in the future.

Additionally, the concept of privacy must be considered. As this robot records a map of the area surrounding it, there are privacy concerns which would need to be noted if this were made into a full consumer device. As this device transmits the map data to the client over Wi-Fi, there is the possibility that this data could be hacked and thus the user's privacy may be compromised. In order to ensure this does not happen, high levels of encryption would need to be implemented on a full release of this system, to ensure it meets EU General Data Protection Regulations.

Section 4. Project Planning & Milestones

Section 4.1 Initial Project Plan

At the beginning of this project, we were tasked with the creation of a preliminary report to detail an overall explanation of our project. For said report, an initial timeline for the project was developed and detailed in a Gantt Chart. This timeline can be seen below (***Figure 4.1*** & ***Figure 4.2***).

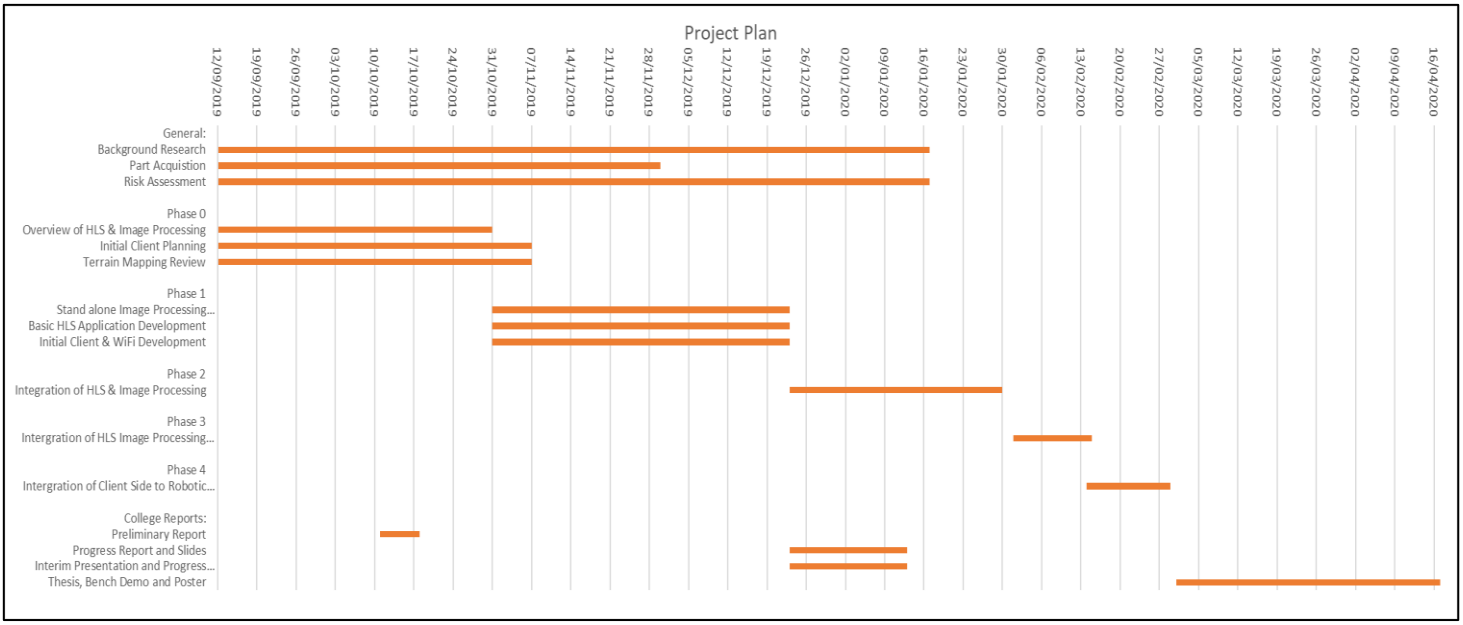


Figure 4.1 Initial Project Timeline (Gantt Chart)

Task Description	Start Date	Duration (Days)
General:		
Background Research	12-Sep-19	127
Part Acquistion	12-Sep-19	79
Risk Assessment	12-Sep-19	127
Phase 0		
Overview of HLS & Image Processing	12-Sep-19	49
Initial Client Planning	12-Sep-19	56
Terrain Mapping Review	12-Sep-19	56
Phase 1		
Stand alone Image Processing Application	31-Oct-19	53
Basic HLS Application Development	31-Oct-19	53
Initial Client & WiFi Development	31-Oct-19	53
Phase 2		
Integration of HLS & Image Processing	23-Dec-19	38
Phase 3		
Intergration of HLS Image Processing into Robotic System	01-Feb-20	14
Phase 4		
Intergration of Client Side to Robotic System	14-Feb-20	15
College Reports:		
Preliminary Report	11-Oct-19	7
Progress Report and Slides	23-Dec-19	21
Interim Presentation and Progress Review Meeting	23-Dec-19	21
Thesis, Bench Demo and Poster	01-Mar-20	76

Figure 4.2 Initial Project Milestones

The initial timeline predicted that most of the first Semester would be spent reviewing the basics of HLS development and the Terrain Mapping System. It predicted that an initial version of the Terrain Mapping System working by Christmas and that the project would be beginning the integration of the Terrain Mapping Code into HLS.

Section 4.2 Semester 1 Progress

At the beginning of Semester 2, we were tasked with the creation of a progress report to outline progress that had been made on the project. For this report, an updated timeline reflecting the changes to the project timeline was created. This timeline can be seen below (***Figure 4.3*** & ***Figure 4.4***).

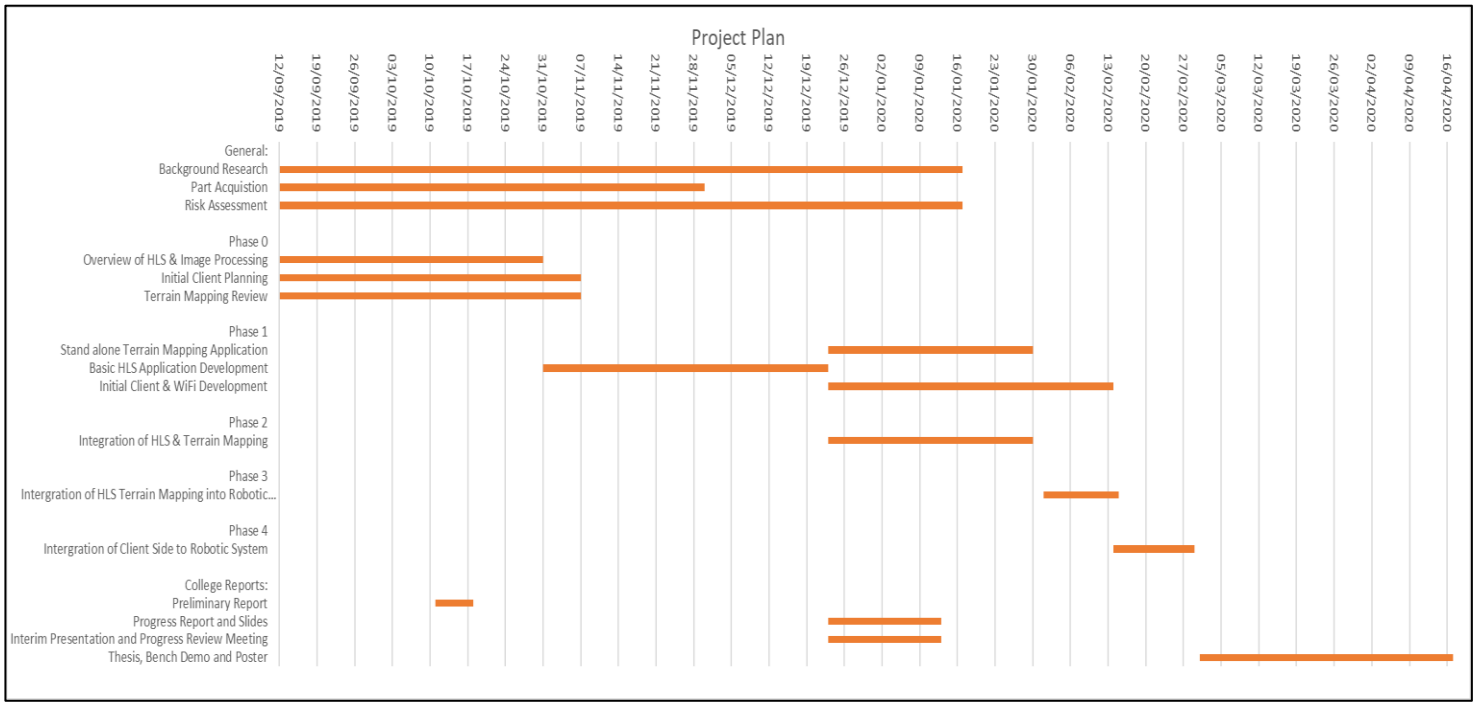


Figure 4.3 Semester 1 Project Timeline (Gantt Chart)

Task Description	Start Date	Duration (Days)
General:		
Background Research	12-Sep-19	127
Part Acquistion	12-Sep-19	79
Risk Assessment	12-Sep-19	127
Phase 0		
Overview of HLS & Image Processing	12-Sep-19	49
Initial Client Planning	12-Sep-19	56
Terrain Mapping Review	12-Sep-19	56
Phase 1		
Stand alone Terrain Mapping Application	23-Dec-19	38
Basic HLS Application Development	31-Oct-19	53
Initial Client & WiFi Development	23-Dec-19	53
Phase 2		
Integration of HLS & Terrain Mapping	23-Dec-19	38
Phase 3		
Intergration of HLS Terrain Mapping into Robotic System	01-Feb-20	14
Phase 4		
Intergration of Client Side to Robotic System	14-Feb-20	15
College Reports:		
Preliminary Report	11-Oct-19	7
Progress Report and Slides	23-Dec-19	21
Interim Presentation and Progress Review Meeting	23-Dec-19	21
Thesis, Bench Demo and Poster	01-Mar-20	76

Figure 4.4 Semester 1 Milestones

As can be seen above there have been significant changes to the timeline to reflect the different issues relating to making the project. By the start of Semester 2, the project had maintained its initial schedule regarding Phase 0. This includes the overview of the HLS, Image Processing and Terrain Mapping Algorithms. However, Phase 1 had significant changes. While the initial projections for the basic HLS development have been maintained, development of both the standalone Terrain Mapping System and the Initial Client development have been substantially delayed. This is due to several outside factors unrelated to the project itself. Additionally, due to research carried out throughout Semester 1, it was decided that the system would move to a LiDAR-based terrain mapping algorithm rather than a Camera based on. This change was due to the camera-based systems being too large to adequately implement onto systems with limited resources such as an FPGA. This shift in focus caused many unforeseen delays.

Section 4.3 Final Timeline of the Project

By the end of the project, the timeline had changed once again. These changes were as a result of changes to the overall project structure as well as changes in the documentation used as a basis for the project. The updated finalised timeline of the project can be seen below (***Figure 4.5*** & ***Figure 4.6***).

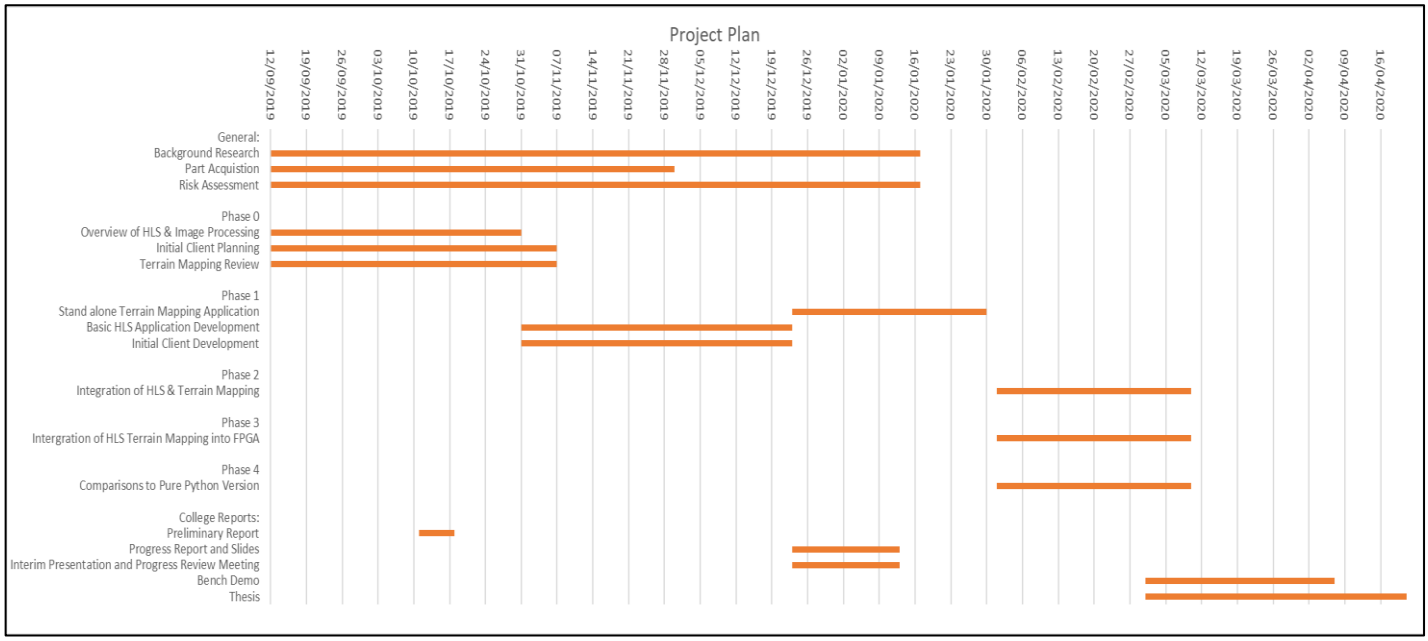


Figure 4.5 Finalised Project Timeline (Gantt Chart)

Task Description	Start Date	Duration (Days)
General:		
Background Research	12-Sep-19	127
Part Acquistion	12-Sep-19	79
Risk Assessment	12-Sep-19	127
Phase 0		
Overview of HLS & Image Processing	12-Sep-19	49
Initial Client Planning	12-Sep-19	56
Terrain Mapping Review	12-Sep-19	56
Phase 1		
Stand alone Terrain Mapping Application	23-Dec-19	38
Basic HLS Application Development	31-Oct-19	53
Initial Client Development	31-Oct-19	53
Phase 2		
Integration of HLS & Terrain Mapping	01-Feb-20	38
Phase 3		
Intergration of HLS Terrain Mapping into FPGA	01-Feb-20	38
Phase 4		
Comparisons to Pure Python Version	01-Feb-20	38
College Reports:		
Preliminary Report	11-Oct-19	7
Progress Report and Slides	23-Dec-19	21
Interim Presentation and Progress Review Meeting	23-Dec-19	21
Bench Demo	01-Mar-20	37
Thesis	01-Mar-20	80

Figure 4.6 Finalised Project Milestones

As can be seen above the final timeline remains mostly unchanged from the timeline described at Christmas. Phase 0 of the project had been completed according to schedule. As the project developed it was decided that the system would use a Jupyter Notebooks Browser [23] as a client due to time constraints. Therefore, all the client development had been covered during the investigations into HLS as these involved the use of Jupyter Notebooks clients already. Finally, due to unforeseen circumstances from outside the college, the due dates for the Bench Demo and the Thesis were changed.

Section 5. Overview of High-Level Synthesis

Section 5.1 Introduction to High-Level Synthesis

Field Programmable Gate Arrays, or FPGAs, are a common device used in the electronics industry. FPGAs are semiconductor-based devices built around the use of an array of programmable logic blocks. These logic blocks allow the FPGA to recreate the functionality of different Hardware implementations as specified by the programmer. FPGAs are programmed using Hardware Description Languages (HDLs), the most common of these being VHDL or Verilog. HDLs are specific languages designed to describe hardware systems to be implemented by the FPGAs logic blocks. These languages are very precise, allowing a skilled user to build optimised hardware structures for their applications; however, these systems are often very verbose and time-consuming to program. Due to this an alternative was developed known as High-Level Synthesis (HLS).

HLS is a method of converting other existing high-level programming languages, such as C/C++, into HDLs for deployment on FPGAs. As opposed to HDLs in which every element must be uniquely specified, HLS operates as a form of design automation, using pre-set elements to create the desired design. While this automation does reduce some of the customization potential of standard HDLs, it allows the designer to benefit from the FPGAs hardware implementations, without the time delays associated with converting an algorithm to an HDL by hand.

The primary goal of this project was to develop a system which would demonstrate the capabilities of High-Level Synthesis to encode more complex systems onto an FPGA. This project is also to serve as an introductory guide to the use of HLS as a system development platform.

Section 5.2 Vivado HLS Introduction and Setup

For this project, the FPGA had to be programmed using HLS. To do so, it was decided that the system would be programmed using the Xilinx Vivado tools. There were two primary tools used in the development of this project: Vivado HLS Design Suite [13], and Vivado Design Suite [16]. Vivado HLS is an application designed to allow the user to code HLS files which will then be converted into HDL files for use in FPGAs. Vivado HLS is designed to support multiple coding languages including

C, C++, System C or OpenCL API C kernel code and is designed to support many C/C++ libraries, notably the OpenCV library [19]. As previously stated, OpenCV is an open-source library designed for computer vision and image processing systems. This library is designed to assist with the development of computer vision and image processing applications has been widely used/supported over the last number of years. The system also supports conversion to Semiconductor intellectual property core (IP core) for integration purposes. An IP core is a reusable unit of logic used in FPGAs.

In order to use HLS correctly, the user must add the specific FPGA board being used to both the Vivado HLS libraries and the Vivado Design Suite libraries. This is done so that the system may correctly deploy the designed IPs on the desired technology. Both automatically come packaged with many of these board files in-built; however, some need to be added manually. For this project, a Xilinx PYNQ-Z2 board was used, which is not within the Vivado libraries by default. Therefore, the board support files must be manually added. To add a board to these tools the following must be done:

For Vivado Design Suite:

- Close Vivado.
- Download the Board Files Online
- Extract and Copy Board Files to: *Directory\Xillinx\Vivado\Version\data\boards\board_files*

For Vivado HLS Design Suite:

- Close Vivado HLS.
- Go to: *Directory\Xillinx\Vivado\Version\common\config*
- Open **VivadoHls_boards.xml** using Text Editor. This file stores the default boards supported by Vivado HLS.
- Add Board Details (Example Details of PYNQ-Z2 seen below (***Figure 5.1***)).

Where *Directory* is the Directory Xilinx is installed on your Computer and *Version* is the Version of Vivado on your computer, e.g. 2018.2.

```
<board name="Xilinx_PYNQ-Z2" display_name="Xilinx PYNQ-Z2" family="zynq" part="xc7z020clg400-1"  device="xc7z020" package="clg400" speedgrade="-1" vendor="pynq.io" />
```

Figure 5.1 Board Details Example

Section 5.3 Basic HLS Application Development

The project began by developing a basic HLS example to demonstrate how Vivado HLS functioned. For this process, it was decided that a basic multiplier would be developed from the initial code to the deployment phase on the FPGA. Development began on the HLS side using the Vivado HLS design suite [13]. As previously stated, this application allows the user to code HLS files, using a variety of different high-level programming languages, which will then be converted into HDL files, such as VHDL or Verilog. For this project, it was decided that the system would be coded using C/C++ as the chosen Terrain Mapping algorithm, tinySLAM [12], was originally developed for C/C++ systems, thus minimizing the required changes for HLS deployment.

Section 5.3.1 Vivado HLS Basics

In the Vivado HLS environment the system requires three file types:

1. The Design File (.C or .CPP), which details the code for the desired application
2. The Testbench File (.C or .CPP), which is used to test the functionality of the system
3. A Header File (.H), which acts as the link to match the Testbench to the Design File.

Images of a basic multiplier code can be seen below (***Figure 5.2***, ***Figure 5.3*** & ***Figure 5.4***):

```
#include "mult.h"
#include <stdio.h>

void mult(int A, int B, int *C)
{
    *C = A * B;
}
```

Figure 5.2 Design File

```
#include "mult.h"
#include <stdio.h>

int main()
{
    printf("\n\n");
    int a, b, c;

    a = 10;
    b = 10;

    printf("Basic Multiplication\n");
    printf("%d * %d", a, b);

    mult(a, b, &c);
    printf(" = %d", c);

    printf("\n\n");
    return 0;
}
```

Figure 5.3 Testbench File

```
#ifndef __MULT_H__
#define __MULT_H__

#include <stdio.h>

void mult(int A, int B, int *C);

#endif
```

Figure 5.4 Header File

It is important to note the lack of the **return** statement in the above code. In typical C/C++ programming the return statement tells the program to leave a subroutine and go to a return address, directly after where the subroutine was called. In most languages, the return statement is either return or return value, where value is a variable or other information coming back from the subroutine as seen

below (***Figure 5.5***). However, in Vivado HLS the code is meant to represent the structure of a hardware implementation of the application. In hardware there is no such thing a blanket return statement, i.e. the system cannot be old to just return to a previous value, any "returns" must use specified lines or ports. Therefore, the use of the return statement is not effective in the HLS environment. Instead, the system must use a pointer (address) to the desired output location. This can be seen in ***Figure 5.2*** where the output address is represented by *C, where * before a variable signifies the address of the value.

```
#include "mult.h"
#include <stdio.h>

int mult(int A, int B)
{
    int answer = 0;
    answer = A*B;
    return answer;
}
```

Figure 5.5 Return Statement Example

Once the files have been created the C synthesis function can be run. This is the function which converts the C/C++ code into HDL models for FPGA deployment. If the synthesis completes without any errors the system will display an estimate of the FPGA resources to be used by the design. If any of these estimates exceed the resources available to the FPGA, then the application cannot be implemented. The resource estimates of this system can be seen below (***Figure 5.6***).

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2	2	2	2	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	3	0	20
FIFO	-	-	-	-
Instance	0	-	144	232
Memory	-	-	-	-
Multiplexer	-	-	-	21
Register	-	-	99	-
Total	0	3	243	273
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

Figure 5.6 Resource Estimates Example

As can be seen above, this simple program uses very few resources. All four of the major resources have usage estimates below 2%. The Block RAM (BRAM) handles system memory. The DSP48 blocks act as arithmetic calculation units, performing mathematical calculations. The Flip Flops (FF) are binary shift registers used an alternate form of memory in the system. Finally, the Lookup-Tables (LUT), is a form of truth table which defines combinatorial logic behaves.

Following C Synthesis, the system can perform C simulation, which runs the testbench code in terms of pure C/C++ code. This test is useful in determining programming errors in the model. This test will only analyse the system in terms of C/C++ code, therefore elements which are acceptable in code but have no hardware equivalent, such as the **return** statement, will not show any errors during this stage. Once C simulation is completed the C/RTL Co-Simulation can be run. This simulation runs both the C simulation and either the VHDL or Verilog model of the system (users' choice), comparing system outputs to determine if both the outputs match.

Once the system has been finalised the user can export the system into the Vivado Design Environment for deployment onto an FPGA. There are two methods of doing this:

1. Copy the HDL files generated by Vivado HLS into the Vivado Design Suite manually. This method, while slow, would allow the user to modify the files if they desired.
2. Use Vivado HLS' built-in RTL Export Function. This function uses the HDL files to create an IP Block which can be imported into the Vivado Design Suite. This method is much easier and more reliable; however, it removes the user's ability to edit the HDL files.

This project focuses on the use of HLS instead of modifying HDLs; therefore, the RTL Export Function is used.

Section 5.3.2 AXI (Advanced eXtensible Interface)

FPGAs are typically broken into two primary parts: The Processing System (PS) and The Programmable Logic (PL). The PS is a processing core, similar to those found on microcontroller devices. This component runs code and handles communication with external devices, such as sensors. In the case of the PYNQ-Z2 board, the PS is comprised of an ARM-Linux processor.

The PL is where the IP Blocks generated by the HLS process are stored. These IP Blocks are programmed by the HDLs to create the functionality of Hardware implementations, such as the calculations for the application. These two components operate independently of each other. In order to communicate between the two components, the system must use what is known as the Advanced eXtensible Interface (AXI) bus [26]. A simplified illustration of the AXI system can be seen below (***Figure 5.7***).

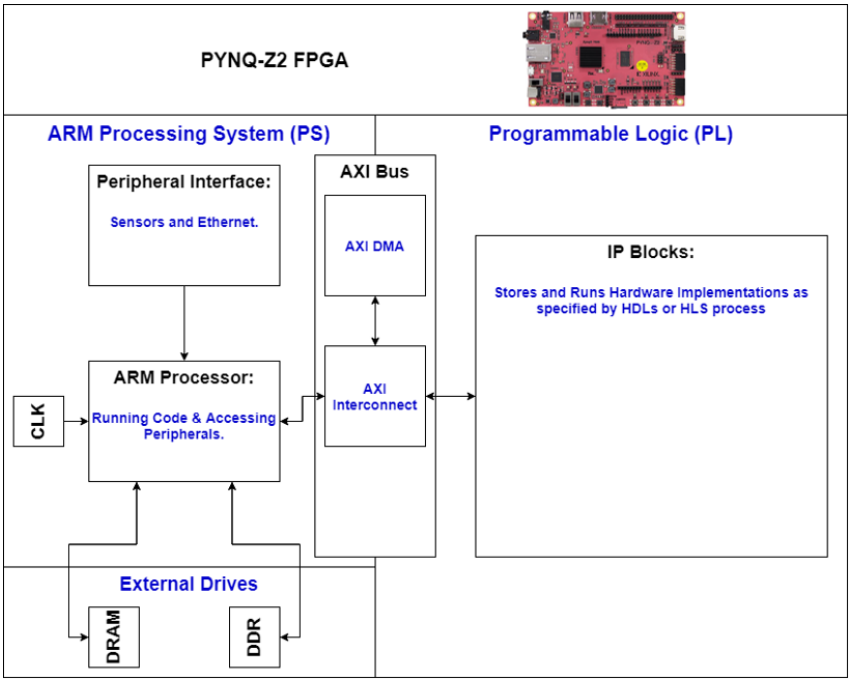


Figure 5.7 AXI Diagram

The AXI bus is the primary gateway between the two parts of the FPGA transferring and translating data between the two components. There are two types of AXI: AXI4 Memory Map and AXI-Stream. In AXI4 Memory Map, the system describes communication between a single AXI master and an AXI slave. Memory Mapped AXI are connected via separate AXI Interconnects. In a memory-mapped system, the master must select the specific memory address that they wish to send data to. This

allows the system to send large amounts of data quickly. However, the sender and receiver must keep track of the exact memory location in use. AXI4 Memory Map breaks down into two types:

1. AXI4 deals with memory mapping of large values and are designed for high-performance memory-mapped requirements.
2. AXI4-Lite meanwhile, is designed for more simplistic data, typically single integer values and is utilized for the single-bit memory map transactions.

The other type of AXI interface is the AXI-Stream interface. This interface relates to the transmission of data that is continuous or must be kept together in some way, e.g. voice or video data. This data is sent in the form of packets rather than single values. Therefore, AXI-Stream must handle the entire stream rather than individual data bytes. To do this AXI-Stream removes the requirement for specific memory addressing found in memory-mapped AXI. Instead, the data passes through what is known as the AXI Direct Memory Access (AXI-DMA). This system acts as a form of register buffer, transferring streams of bytes back and forth between the PS and a pre-set location in memory. The advantage of AXI-Stream over standard AXI Memory Mapped is that the system does not need to manage the specific memory address for each byte of data, and the system is guaranteed to wait for all values to complete before moving to the next step. However, the disadvantage of AXI-Stream is while AXI-Stream allows for unlimited burst transactions, each individual AXI-Stream transmission (packets) have a set maximum size, as opposed to AXI Memory Mapped which has no maximum provided that each address is correctly accessed.

In Vivado HLS the use of the AXI bus is controlled using different pragmas. In C/C++ code "*pragma*" directives are the methods specified by the C standard for providing additional information to the compiler. These allow the user to specify values or conditions which may be beyond what can be conveyed using the programming language itself. These are built-in to the C compiler. Vivado HLS has several HLS exclusive pragmas which denote aspects of converting code to HDLs. These include pragmas for Kernel Optimization, Interface Synthesis, Task-level Pipeline and Data Structures. The full list of pragmas can be found online [27]. In order to use the AXI bus, the system must set the input and

output lines of the system to the relevant AXI pragmas. An updated version of the previous multiplier code (Section 5.3.1) can be seen below (***Figure 5.8***).

```
void mult(int A, int B, int *C)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=A
#pragma HLS INTERFACE s_axilite port=B
#pragma HLS INTERFACE s_axilite port=C

    *C = A * B;
}
```

Figure 5.8 Multiplier AXI-Lite

In the example above, the system uses the AXI4-Lite to send single integer values between the PS and PL. The inputs *A* and *B* as well as the output address **C* are set to the AXI-Lite port using the `#pragma HLS INTERFACE s_axilite`. Additionally, you can see that the return port is set to ap_ctrl. The ap_ctrl port is a port built into all IP blocks allowing block-level interfaces to be accessed. These are used to control the timing of a block independently from the main clock. As such control is not required in this system the pragma `#pragma HLS INTERFACE ap_ctrl_none port=return` which is a default setting disabling ap_ctrl.

In the case of AXI-Stream however, the system is very different. AXI-Stream is designed to work with arrays and does not work with single integers. Therefore, the following array multiplier was made to accommodate it (***Figure 5.9***, ***Figure 5.10*** & ***Figure 5.11***)

```
#include  "smult.h"
#include  <ap_axi_sdata.h>

void hls_mult(stream_type in_data_A[10], stream_type in_data_B[10], stream_type out_data[10])
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=in_data_A
#pragma HLS INTERFACE axis port=in_data_B
#pragma HLS INTERFACE axis port=out_data

    for(int i = 0; i < 10; i++)
    {
        out_data[i].data = in_data_A[i].data * in_data_B[i].data;
        out_data[i].strb = in_data_A[i].strb;
        out_data[i].keep = in_data_A[i].keep;
        out_data[i].user = in_data_A[i].user;
        out_data[i].last = in_data_A[i].last;
        out_data[i].id = in_data_A[i].id;
        out_data[i].dest = in_data_A[i].dest;
    }
}
```

Figure 5.9 AXI-Stream Array Multiplier

```
#ifndef __SMULT_H__
#define __SMULT_H__

#include <stdio.h>
#include <assert.h>
#include <ap_axi_sdata.h>

typedef ap_axiu<32,1,1,1> stream_type;

void hls_mult(stream_type in_data_A[10], stream_type in_data_B[10], stream_type out_data[10]);

#endif
```

Figure 5.10 AXI-Stream Header File

```
#include <stdio.h>
#include <stdlib.h>
#include "smult.h"

int main()
{
    printf("Stream Multiplication\n");
    stream_type A[10], B[10], C[10];

    for(int i = 0; i < 10; i++)
    {
        A[i].data = i;
        A[i].strb = -1;
        A[i].keep = 15;
        A[i].user = 0;
        A[i].last = (i == 9) ? 1 : 0;
        A[i].id = 0;
        A[i].dest = 0;

        B[i].data = i;
        B[i].strb = -1;
        B[i].keep = 15;
        B[i].user = 0;
        B[i].last = (i == 9) ? 1 : 0;
        B[i].id = 0;
        B[i].dest = 0;
    }

    hls_mult(A, B, C);

    for(int i = 0; i < 10; i++)
    {
        printf("%d * %d = %d\n", (int) A[i].data, (int) B[i].data, (int) C[i].data);
    }

    return 0;
}
```

Figure 5.11 AXI-Stream Testbench File

As can be seen above AXI-Stream has a very different process. Instead of simple integers, the AXI-Stream uses must be defined using an AXI stream type. For this example, the in-built *ap_axiu* was used. These stream types store the data value as well as several other ap values including strb, user, last and id. These are values assigned to the stream at the PS to ensure the stream is read in the correct order at the other side. In the case of the multiplier when creating the output stream these values must be set to relevant numbers in order for data to be correctly processed. The most simplistic approach to this as seen in ***Figure 5.9*** is to set the values of the output line equal to one of the input lines values. Both the output and input lines are set to AXI-Stream using the pragma `#pragma HLS INTERFACE axis`.

Section 5.3.3 Vivado Design Suite

Once the system has been successfully coded using Vivado HLS the system must be then imported into the Vivado Design Suite for Deployment onto the FPGA. The Vivado Design Suite [16] is a software application designed to allow users to write HDL files and convert them into Bitstream

files for deployment. As this project focuses on the use of HLS, the system imports the HDL files generated by Vivado HLS rather than create unique ones.

To use the designs created in Vivado HLS, the user must first create an empty project in Vivado using the desired technology (FPGA). Once that has been done the HLS files can be imported into the Design Suite using two different methods as described in Section 5.3.1. As this project focuses on the use of the HLS designs the ability to edit the HDLs manually is not be needed, therefore the RTL export method is used. The IP can be imported using the IP Catalog as seen in the images below (***Figure 5.12*** & ***Figure 5.13***)

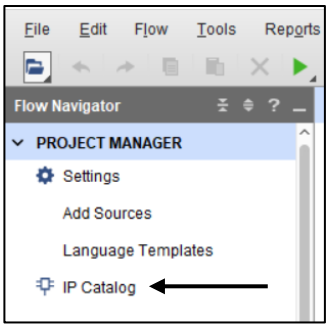


Figure 5.12 IP Catalog

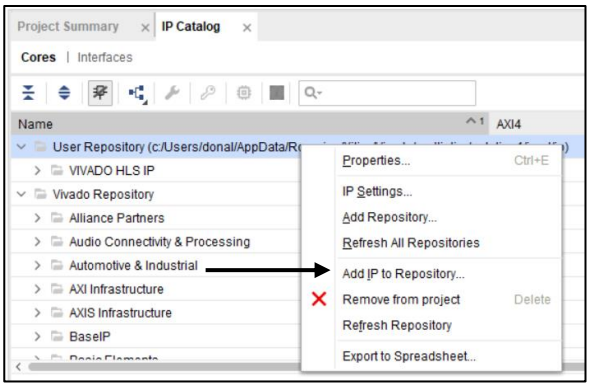


Figure 5.13 Add IP

Once the IP has been imported the system can be built. To do this the user creates a Block Design. In Vivado the Block Design of a system is used to connect different IP together to form a complete system. This process is used to connect the system to the PS via the AXI. For this project, the system uses a ZYNQ Processing System. As discussed in Section 5.3.2, in a memory-mapped AXI Interface the system must communicate through an AXI Interconnect between the PS and the IP. The diagram of the basic multiplier using AXI-Lite can be seen below (***Figure 5.14***)

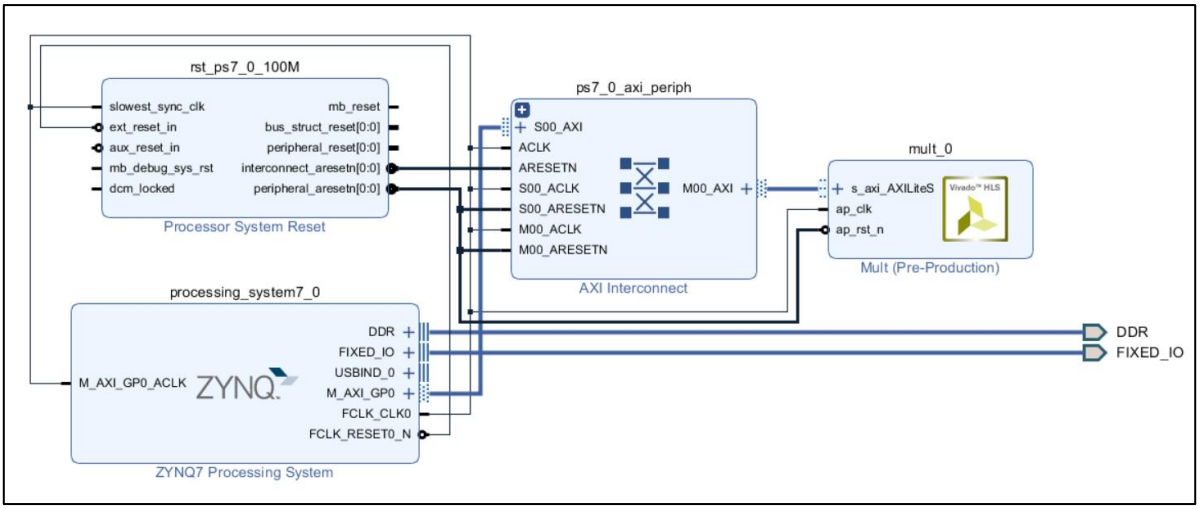


Figure 5.14 Basic Multiplier AXI-Lite

The multiplier is connected to the processing system via the AXI interconnect. Additionally, ZYNQ Processing System can be modified to change the clock speed of the system; however, the clock speed of the IP must be changed from within Vivado HLS to match the speed of the ZYNQ. In AXI-Lite systems, the design shows a single AXI-Lite connection between the AXI Interconnect and the Multiplier. This single connection carries all three data line (A, B & C).

In the case of the AXI-Stream multiplier, mentioned previously, the system must use the AXI DMA in communication. While in an AXI-Lite implementation the system uses a single connection, for AXI-Stream each input requires a unique connection. These connections are connected to an AXI DMA. A DMA can support two lines, an input, and an output line. Therefore, as this multiplier has two inputs (A & B) and a single output, two DMAs were required. Additionally, as this data deals with single integers, the system must buffer the stream before accessing the data. To do this the system passes each line (A. B & Output) through AXI-Stream Data FIFO. Finally, the inclusion of an AXI DMA requires that an AXI-Interrupt Controller also be added. As there are two DMAs their interrupt lines are concatenated together before being passed to the Interrupt Controller. This results in a much more complex system as seen below (***Figure 5.15***).

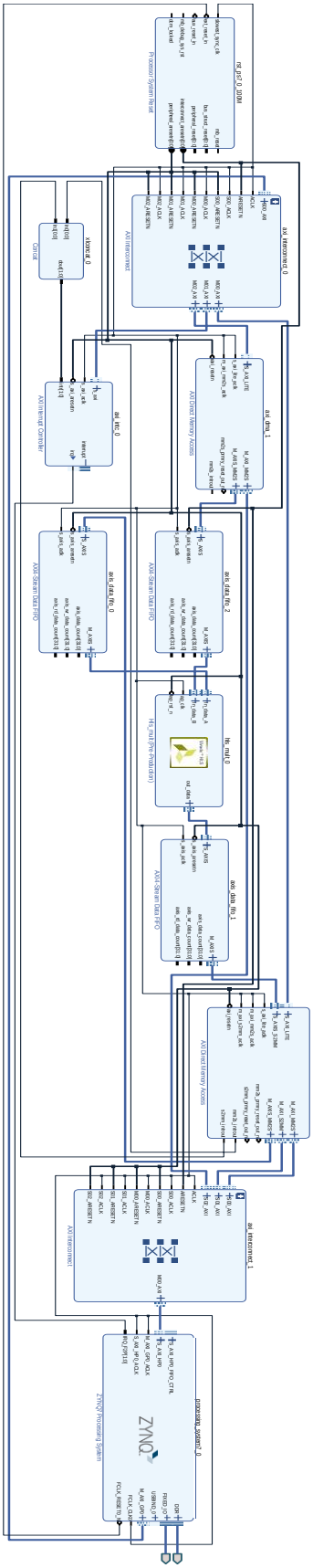


Figure 5.15 AXI-Stream Multiplier

Before finishing the Block Design, all unmapped connections must be handled. To do this the user can use the automatic addressing button as seen below (***Figure 5.16***). This function automatically connects any unmapped drives to default values. Finally, to check if the Block Design has been correctly connected the user can use the validate design button (***Figure 5.17***). If no errors are detected, then the design has all components connected sufficiently. Once the block design has been completed the system can be converted into a bitstream for deployment. To do this the user must create an HDL wrapper for the design (***Figure 5.18)***. This wrapper creates an HDL version of the complete design, which denotes all the connections of your design. Once the wrapper has been created the bitstream can be generated by pressing the Generate Bitstream button, which will perform Synthesis, Implementation and Bitstream Generation (***Figure 5.19***).

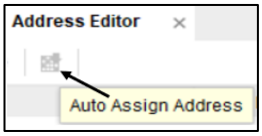


Figure 5.16 Auto Assign Address

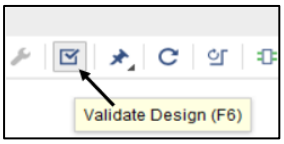


Figure 5.17 Validate Design

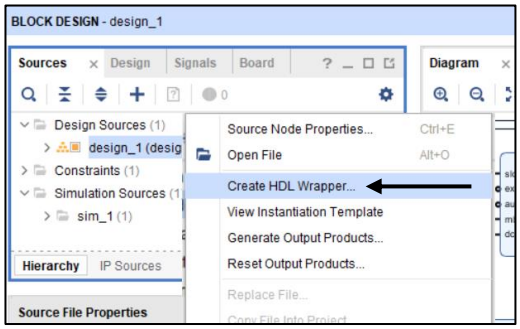


Figure 5.18 HDL Wrapper

Figure 5.19 Bitstream Generation

Section 5.3.4 Jupyter Notebooks and System Deployment

Once the Bitstream has been generated the system can be deployed on the FPGA. For this project, this process is done using the Jupyter Notebooks Browser [23]. Jupyter Notebooks is a server-client application that allows editing and running notebook documents via a web browser. Jupyter Notebooks is the supported IDE of the PYNQ boards and runs on the bootable Linux operating system. It supports multiple coding languages including Python, C, and Pearl. For this project, all elements of the Jupyter Notebooks side are implemented using Python 3. Before deployment can begin the FPGA

must be setup on Jupyter Notebooks. This can be done by following the Jupyter Notebooks Setup Guide [28].

Once set up the Jupyter Notebooks system can be used to run the custom design on the FPGA. To do this the Bitstream files created by the Vivado Design Suite must be copied to the PYNQ board. The following files must be copied to a **single shared folder** on the FPGA:

1. Copy the **.hwh** file found at: *Directory\projectname.srcs\sources_1\bd\design\hw_handoff*
2. Copy the **.bit** and **.tcl** files found at: *Directory\projectname.runs\impl_1*
3. Rename the **.bit** and **.tcl** files so that their names match the **.hwh** file.

Where *Directory* is the project directory, *projectname* is the name of the project used in the Vivado Design Suite and *design_1* is the name of the Block Design. Once this has been done the files can be accessed from with the Jupyter Notebooks browser. To run the system, create a new Python 3 notebook. For the AXI-Lite multiplier, the system must first import the PYNQ Overlay library and set the overlay to the bit file location. Once the Overlay has been set you must then select the IP you wish to test. As seen from the Vivado Block Diagram the multiplier has been labelled as mult_0 (***Figure 5.14***). Finally, the user can read/write to the input/output of the IP. As this system uses AXI-Lite which is a memory-mapped interface the program must specify the exact memory address for each value. These memory addresses can be found in the VHDL created by the HLS process, where the values 16#XX is the equivalent to the hexadecimal value 10xXX as seen below (***Figure 5.20*** & ***Figure 5.21***).

```
constant ADDR_A_DATA_0 : INTEGER := 16#10#;
constant ADDR_A_CTRL   : INTEGER := 16#14#;
constant ADDR_B_DATA_0 : INTEGER := 16#18#;
constant ADDR_B_CTRL   : INTEGER := 16#1c#;
constant ADDR_C_DATA_0 : INTEGER := 16#20#;
constant ADDR_C_CTRL   : INTEGER := 16#24#;
constant ADDR_BITS         : INTEGER := 6;
```

Figure 5.20 HDL Memory Addresses

```
from pynq import Overlay
mult = Overlay('/home/xilinx/MyFiles/Demo/mult_demo.bit')
add_ip = mult.mult_0
add_ip.write(0x10, 10)
add_ip.write(0x18, 100)
add_ip.read(0x20)
```

```
1000
```

Figure 5.21 AXI-Lite Multiplier Notebook

The AXI-Stream implementation is again more complex. Once again, the system imports the PYNQ Overlay library and set the overlay to the bit file location. However, unlike the AXI-Lite implementation, instead of targeting the IP itself the program instead selects the two DMAs (axi_dma_0

and axi_dma_1 respectively). In order to send data to the DMA, it must be stored in a DMA compatible array known as a CMA Array. Once the arrays of data have been filled the data is sent to the PL system using the **dma.sendchannel.transfer** command. Following this, the output is read using the **dma.recv.channel.transfer** command. These commands send and receive data to/from the DMAs, which removes the need to access the specific memory addresses. An example of AXI-Stream multiplier and images of the PYNQ to Jupyter Notebooks setup can be seen below (***Figure 5.22***, ***Figure 5.23*** & ***Figure 5.24***).

```
from pynq import Xlnk
import numpy as np
import pynq.lib.dma
from pynq import Overlay

overlay = Overlay('/home/xilinx/MyFiles/AXI-Stream-Demo/design_1.bit')
dmaA = overlay.axi_dma_0
dmaB = overlay.axi_dma_1

xlnk = Xlnk()

input_buffer_A = xlnk.cma_array(shape=(10,), dtype=np.uint32)
input_buffer_B = xlnk.cma_array(shape=(10,), dtype=np.uint32)

for i in range(10):
    input_buffer_A[i] = i
    input_buffer_B[i] = 2*i

print("Inputs")
for i in range(10):
    print(input_buffer_A[i]," * ",input_buffer_B[i])

print("")
dmaA.sendchannel.transfer(input_buffer_A)
dmaA.sendchannel.wait()
dmaB.sendchannel.transfer(input_buffer_B)
dmaB.sendchannel.wait()

output_buffer = xlnk.cma_array(shape=(10,), dtype=np.uint32)

print("")
dmaA.recvchannel.transfer(output_buffer)
dmaA.recvchannel.wait()

print("Outputs")
for i in range(10):
    print(output_buffer[i])

del input_buffer_A, input_buffer_B, output_buffer
```

Figure 5.22 AXI-Stream Multiplier Notebook

```
Inputs
0  *  0
1  *  2
2  *  4
3  *  6
4  *  8
5  *  10
6  *  12
7  *  14
8  *  16
9  *  18


Outputs
0
2
8
18
32
50
72
98
128
162
```

Figure 5.23 AXI-Stream Outputs

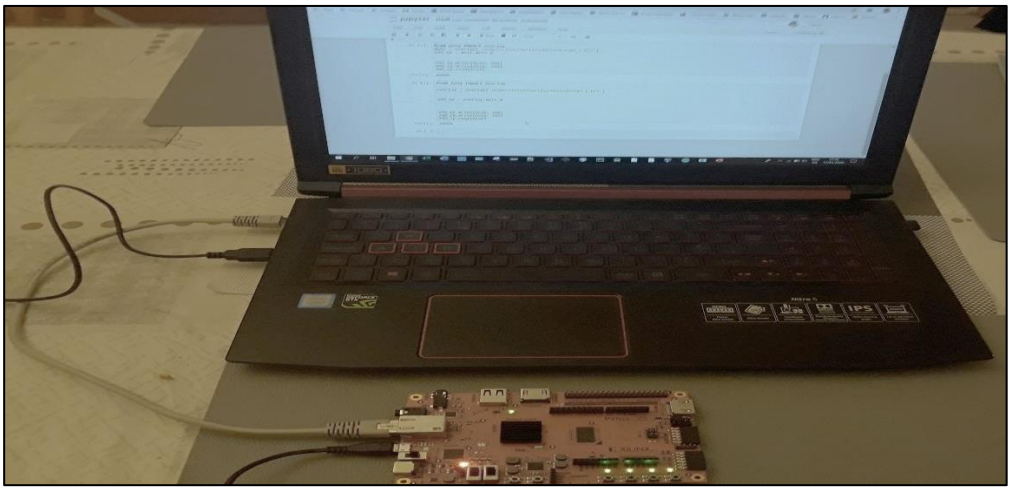


Figure 5.24 PYNQ to Jupyter Notebooks

Section 6. Terrain Mapping Methodology: TinySLAM

Section 6.1 Introduction to Terrain Mapping

The primary task of this project is the development of a system which can accurately create a map of the surrounding environment using an FPGA. In order for the system to create a map of the surrounding terrain, the system must have some method of "perceiving" the terrain. The field of Computer Perception, also known as Machine Perception, has become an increasingly important field of electronics over the last number of years. This process relates to the concept of devices which can use sensors to analyse the world around them in order to infer some aspect of the real world. The goal of machine perception is to create a system that can perceive the world in a similar manner to how humans can in order to allow for more complex applications.

Terrain Perception is a specific form of computer perception systems. The goal of terrain perception is to analyse the structure of the terrain or environment. Terrain Mapping relates to the method of using this "perception" data in order to construct a map of the perceived environment. This type of application can be particularly helpful for mapping areas unsafe for human interaction or in systems designed for little to no user input.

As discussed in Section 2.2.2.2, this project is based around the use of the tinySLAM algorithm. Simultaneous Localization And Mapping (SLAM) algorithms are algorithms which relate to the problem of constructing a map of the surrounding environment while simultaneously keeping track of the system location within said environment. In most mapping or location systems, the system has one of these two pieces of information and uses this knowledge to determine the other value, i.e. it knows its location and uses that to determine the map or vice versa. In a SLAM problem, the system has neither value, resulting in what is commonly referred to as a "chicken-or-egg" scenario. SLAM algorithms relate methods which solve this problem.

There are many types of SLAM algorithms, each focussed on the use of different sensors or data. Originally, the project was to focus on the use of camera-based SLAM algorithms such as FastSLAM [9] and ORBSLAM [10]. However, it was decided that the system would focus on the use of the LiDAR-based tinySLAM algorithm as detailed in *tinySLAM: a SLAM Algorithm in less than 200 lines C-*

Language Program [12]. This algorithm was chosen due to its low complexity being perfect for systems with limited resources such as an FPGA. Additionally, the algorithm was designed to operate using C/C++ code which the HLS process is based on allowing for a more direct conversion of the system.

Section 6.2 TinySLAM Methodology

The tinySLAM algorithm also referred to as CoreSLAM, is a low complexity SLAM implementation. The system is based around the use of a remote sensing technology called LiDAR. **L**ight **D**etection **A**nd **R**anging sensors, or LiDAR sensors, are a form of remote sensing which uses a laser in order to determine the distance between the sensor and an object. In LiDAR sensors, the sensor emits a pulsed laser towards the object. The laser is then reflected by the object back towards the sensor. The time taken for the laser to return is used to determine the distance between the LiDAR and the object.

The tinySLAM algorithm is designed to be a simplified SLAM implementation using C/C++ code. In the paper *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12] the three primary elements of the tinySLAM algorithm are described. These three processes describe the individual calculations required to operate the tinySLAM algorithm; however, these processes are independent and are not linked together. Therefore, in order to implement the tinySLAM system, the **main** method function must be established. For this project, the system is designed around a modified version of the tinySLAM using the code found on the OpenSLAM website [29] as a basis. OpenSLAM.org is an online repository where researchers can publish different SLAM techniques to make them more accessible. The OpenSLAM website acts as a repository where the authors of the algorithm [12] have stored a copy of their system to allow others to examine their findings. Using this repository as a basis, the implementation of tinySLAM used in this project was developed.

Section 6.2.1 Initial TinySLAM Testing and Modification

Initial testing of the tinySLAM algorithm was performed using the test data sets found in the OpenSLAM [29] repository. This data set is based on the use of a highly precise laser scanner called a

Hokuyo URG04 laser scanner [20]. This laser scanner is a high precision LiDAR with 240° area scanning range with 0.36° angular resolution. Additionally, the scanner has a 5.6-metre range and operates at a 10Hz scanning frequency. While this scanner was deemed much too expensive for the purposes of this project, the sample data provided by the scanner in the repository was deemed useful in determining the functionality of the system.

Using this data, the version of the tinySLAM algorithm found on the OpenSLAM repository could be tested. It was decided that all testing of the pure C/C++ versions of this code would be performed using the Visual Studio Software [30]. Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to program different systems and supports a wide range of languages and Microsoft Development platforms. For this test, the **test_lab_reverse.c** from the OpenSLAM repository was used as the main method using the test data file **test_lab.dat**. In order to view the map created by the system, the code has been modified using the OpenCV library [19]. OpenCV is an open-source library designed for computer vision and machine learning functions. OpenCV was "*built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products*" [19]. This library is designed to assist programmers in the creation of all kinds of image processing applications. This library is used to display the map created by the algorithm and to add a marker at the location of the device. An image of the resulting map and the added code can be seen below (***Figure 6.1*** & ***Figure 6.2***).

```
//Display Map
Mat img = imread(filename);
namedWindow("image", WINDOW_NORMAL);
imshow("image", img);
waitKey(0);

circle(img, CvPoint(x, TS_MAP_SIZE - y), 10, (0, 255, 0), 10);
imshow("image", img);
waitKey(0);
destroyWindow("image");
```

Figure 6.1 OpenCV Image Display and Marker

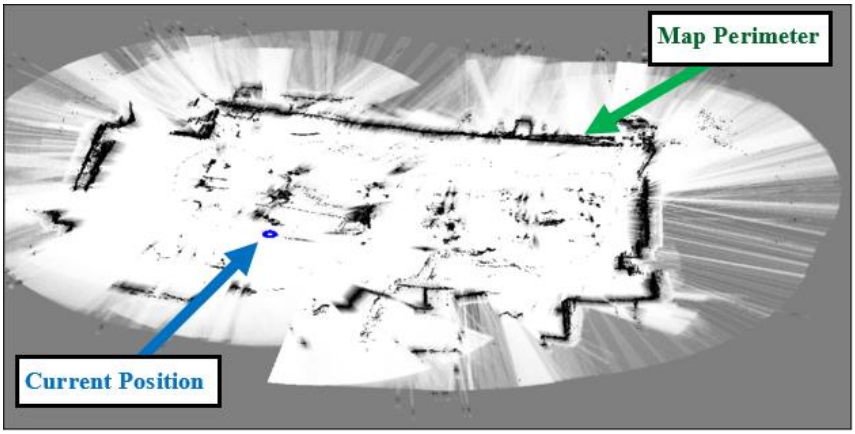


Figure 6.2 TinySLAM Test Map

As can be seen above the tinySLAM algorithm creates a map using Black and White Colours. The Black in the image represents walls or objects detected in the sensor's path. The White in the image denotes open space, i.e. a lack of obstruction. Additionally, using the OpenCV the system adds an additional blue marker around the location of the LiDAR device during the scan. The system also displays the scan number and the position value for each scan in the command window.

In the OpenSLAM repository, the system includes many additional functions and extensions not required for the implementation of the tinySLAM algorithm itself. For the purposes of this project, the system focuses solely on the primary tinySLAM algorithm defined in the files **CoreSLAM.c**, **CoreSLAM.h** and the testing methods defined in **test_lab_reverse.c**. This decision was made due to the time constraints of the project, and the limited resources on the FPGA not being suitable for implementing unnecessary functionality. In this regard the tinySLAM header file included in the OpenSLAM repository, **CoreSLAM.h** included references to additional C Files found on the library. These additional functions, relating to *CoreSLAM_ext.c, CoreSLAM_loop_closing.c, CoreSLAM_random.c* and *CoreSLAM_state.c*, were removed as they were not required for the terrain mapping algorithm and would take too much space on the limited FPGA.

Additionally, the original tinySLAM code from OpenSLAM included calculations for position based on odometers. An odometer is an instrument used for measuring the distance travelled by an object, typically a vehicle such as a bicycle or car. The original system was built into a robot system to move around the area, thus allowing the odometers to be used when calculating the position. However,

while the odometer-based calculation was indeed included in the OpenSLAM code, it is overwritten by the LiDAR-based position calculation, effectively ignoring this element. As this project does not include the use of odometers this excess code was removed.

Finally, the original code includes a method of "spanning" the test scans across multiple degrees. This process created multiple copies of the same data and causing the system to run the calculations multiple times. This was done to improve the system accuracy by repeating data to prevent errors. This process caused all data to be repeated three times, which causes space issues for FPGA and reduces system speed. The accuracy added by the process was deemed minimal and thus the spanning value was set to 1 instead of the original 3. Comparison of the effect of spanning can be seen below (***Figure 6.3*** & ***Figure 6.4***)

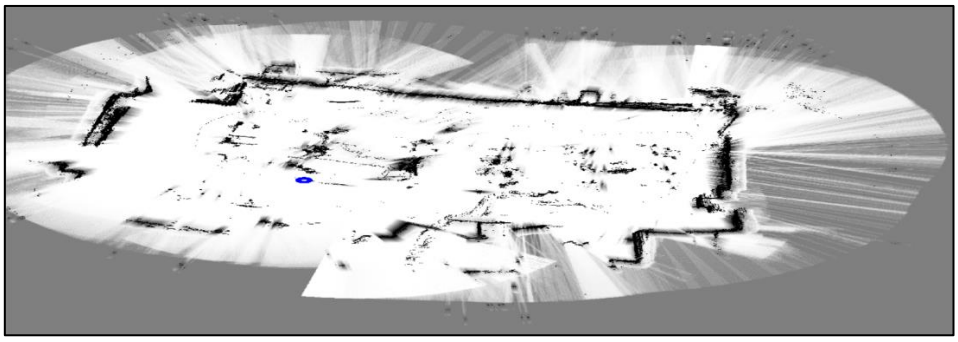


Figure 6.3 TinySLAM Span = 3

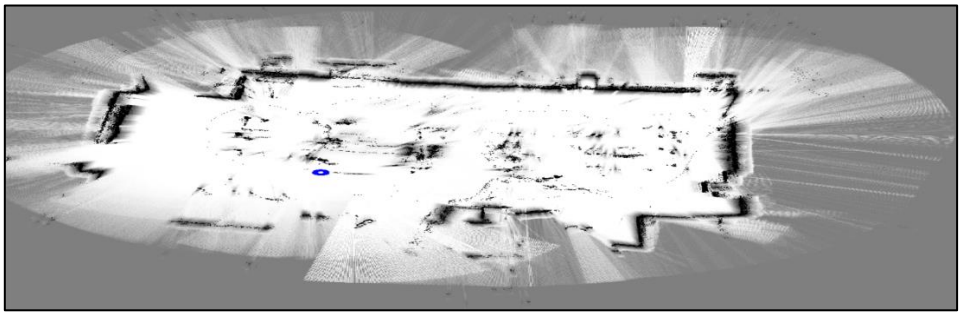


Figure 6.4 TinySLAM Span = 1

Section 6.2.2 Modified TinySLAM Breakdown

Following the initial testing of the existing tinySLAM algorithm and subsequent modification the finalised version of tinySLAM to be used in this project was decided. A flowchart describing the tinySLAM algorithm can be seen below (***Figure 6.5***).

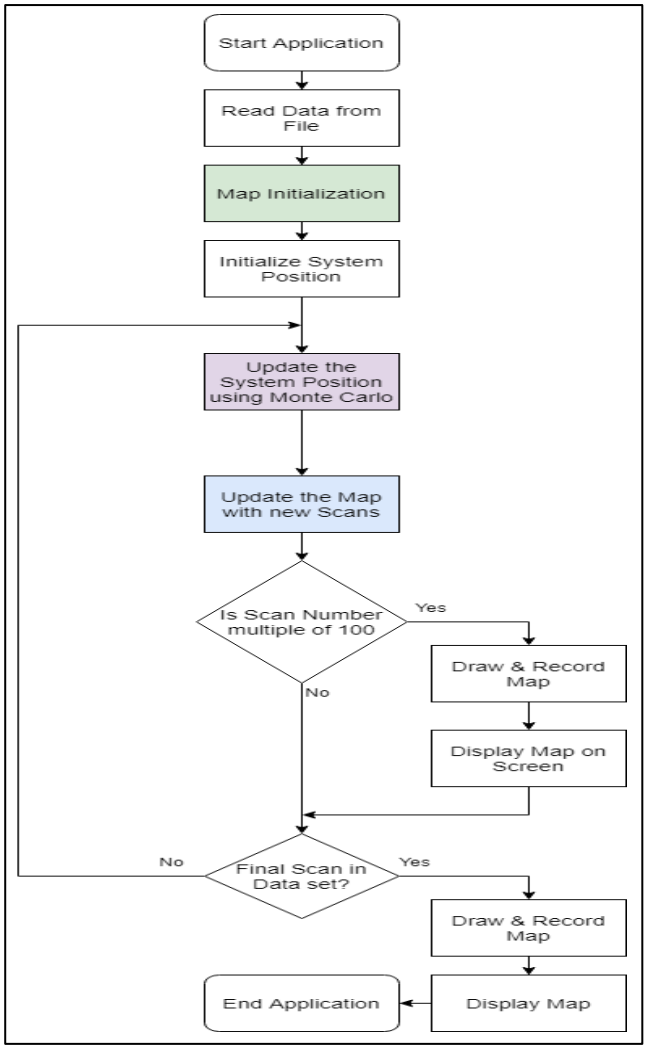


Figure 6.5 TinySLAM Flowchart

Section 6.2.2.1 Map Initialization

The tinySLAM algorithm breaks down into three primary components as described in *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12]. Firstly, there is the Map Initialization phase. This phase loops through each pixel in the image to create a blank map. The initialization is only run once before the system enters the loop. The code for the map initialization phase can be seen below (***Figure 6.6***)

```
void ts_map_init(ts_map_t *map)
{
    int x, y, initval;
    ts_map_pixel_t *ptr;
    initval = (TS_OBSTACLE + TS_NO_OBSTACLE) / 2;
    for (ptr = map->map, y = 0; y < TS_MAP_SIZE; y++) {
        for (x = 0; x < TS_MAP_SIZE; x++, ptr++) {
            *ptr = initval;
        }
    }
}
```

Figure 6.6 TinySLAM Map Init Code

Section 6.2.2.2 Monte Carlo and Updating Position

Following this, the system calculates the LiDARs current position in the map using a process known as Monte Carlo. A Monte Carlo method [10], is a computational algorithm which utilizes random sampling to determine some value. For this system the Monte Carlo method is based on the ***Distance_Scan_To_Map*** described in *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12]. This method uses the LiDAR data to calculate the distance between the sensor and some object on the map, i.e. where the LiDAR laser would have been reflected. The Monte Carlo method uses this calculation to determine the system's position. The Monte Carlo method of this system selects a random position within a certain range of the previous position. The system then uses the distance value as calculated by ***Distance_Scan_To_Map*** to compare the current location to the previous location. This method loops through a series of these random positions and selects the position with the lowest change in distance as the current position of the system. The code for the Monte Carlo method and the ***Distance_Scan_To_Map*** can be seen below (***Figure 6.7*** & ***Figure 6.8***).

```
int ts_distance_scan_to_map(ts_scan_t *scan, ts_map_t *map, ts_position_t *pos)
{
    double c, s;
    int i, x, y, nb_points = 0;
    int64_t sum;

    c = cos(pos->theta * M_PI / 180);
    s = sin(pos->theta * M_PI / 180);

    //Translate and rotate scan to robot position and compute the distance.
    for (i = 0, sum = 0; i != scan->nb_points; i++) {
        if (scan->value[i] != TS_NO_OBSTACLE) {
            x = (int)floor((pos->x + c * scan->x[i] - s * scan->y[i]) * TS_MAP_SCALE + 0.5);
            y = (int)floor((pos->y + s * scan->x[i] + c * scan->y[i]) * TS_MAP_SCALE + 0.5);

            //Check Boundaries
            if (x >= 0 && x < TS_MAP_SIZE && y >= 0 && y < TS_MAP_SIZE) {
                sum += map->map[y * TS_MAP_SIZE + x];
                nb_points++;
            }
        }
    }
    if (nb_points) sum = sum * 1024 / nb_points;
    else sum = 2000000000;
    return (int)sum;
}
```

Figure 6.7 Distance_Scan_To_Map Code

```
ts_position_t monte_carlo_move(ts_scan_t *scan, ts_map_t *map, ts_position_t *start_pos, int debug)
{
    ts_position_t cpp, currentpos, bestpos, lastbestpos;
    int currentdist;
    int bestdist, lastbestdist;
    int counter = 0;

    currentpos = bestpos = lastbestpos = *start_pos;
    currentdist = ts_distance_scan_to_map(scan, map, &currentpos);
    bestdist = lastbestdist = currentdist;

    do {
        currentpos = lastbestpos;
        currentpos.x += 50 * (((double)rand()) / RAND_MAX - 0.5);
        currentpos.y += 50 * (((double)rand()) / RAND_MAX - 0.5);
        currentpos.theta += 50 * (((double)rand()) / RAND_MAX - 0.5);

        currentdist = ts_distance_scan_to_map(scan, map, &currentpos);

        if (currentdist < bestdist) {
            bestdist = currentdist;
            bestpos = currentpos;
            if (debug) printf("Monte carlo ! %lg %lg %lg %d (count = %d)\n", bestpos.x, bestpos.y, bestpos.theta, bestdist, counter);
        }
        else {
            counter++;
        }
        if (counter > 100) {
            if (bestdist < lastbestdist) {
                lastbestpos = bestpos;
                lastbestdist = bestdist;
                counter = 0;
            }
        }
    } while (counter < 1000);
    return bestpos;
}
```

Figure 6.8 Monte_Carlo Code

Section 6.2.2.3 Map Update

Finally, the last of the major components of tinySLAM is the Map Update code. This function is designed to add the values from the scans to the map, based on the position value calculated during the Monte Carlo. To do this the system loops through each element of the scan, determining the position of each element using Sine and Cosine calculations. Once this is done the Map Update function calls the ***Map_Laser_Ray*** function. This function as described by the tinySLAM authors *"uses a Bresenham algorithm to draw the laser rays in the map, with another enhanced bresenham algorithm inside to compute the right profile"* [12]. This method uses the previously calculated coordinate geometry functions to determine the location of the scan on the map and then determines if the scan registers as an obstacle (perimeter line) or empty space based on a threshold value. These values are then added to the map by looping through each pixel of the map using pointers. The Map Update function performs this same method for all scans. The code for the ***Map_Laser_Ray*** and Map Update can be seen below (***Figure 6.9*** & ***Figure 6.10***).

```
void ts_map_laser_ray(ts_map_t *map, int x1, int y1, int x2, int y2, int xp, int yp, int value, int alpha)
{
    int x2c, y2c, dx, dy, dxc, dyc, error, errorv, derrorv, x;
    int incv, sincv, incerrorv, incptrx, incptry, pixval, horiz, diago;
    ts_map_pixel_t *ptr;

    if (x1 < 0 || x1 >= TS_MAP_SIZE || y1 < 0 || y1 >= TS_MAP_SIZE)
        return;

    x2c = x2;
    y2c = y2;

    //Clipping
    if (x2c < 0) {
        if (x1 == x2c) return;
        y2c += (y2c - y1) * (-x2c) / (x2c - x1);
        x2c = 0;
    }

    if (x2c >= TS_MAP_SIZE) {
        if (x1 == x2c) return;
        y2c += (y2c - y1) * (TS_MAP_SIZE - 1 - x2c) / (x2c - x1);
        x2c = TS_MAP_SIZE - 1;
    }

    if (y2c < 0) {
        if (y1 == y2c) return;
        x2c += (x1 - x2c) * (-y2c) / (y1 - y2c);
        y2c = 0;
    }

    if (y2c >= TS_MAP_SIZE) {
        if (y1 == y2c) return;
        x2c += (x1 - x2c) * (TS_MAP_SIZE - 1 - y2c) / (y1 - y2c);
        y2c = TS_MAP_SIZE - 1;
    }
    dx = abs(x2 - x1); dy = abs(y2 - y1);
    dxc = abs(x2c - x1); dyc = abs(y2c - y1);
    incptrx = (x2 > x1) ? 1 : -1;
    incptry = (y2 > y1) ? TS_MAP_SIZE : -TS_MAP_SIZE;
    sincv = (value > TS_NO_OBSTACLE) ? 1 : -1;

    if (dx > dy) {
        derrorv = abs(xp - x2);
    }
    else {
        SWAP(dx, dy); SWAP(dxc, dyc); SWAP(incptrx, incptry);
        derrorv = abs(yp - y2);
    }

    error = 2 * dyc - dxc;
    horiz = 2 * dyc;
    diago = 2 * (dyc - dxc);
    errorv = derrorv / 2;
    incv = (value - TS_NO_OBSTACLE) / derrorv;
    incerrorv = value - TS_NO_OBSTACLE - derrorv * incv;
    ptr = map->map + y1 * TS_MAP_SIZE + x1;
    pixval = TS_NO_OBSTACLE;

    for (x = 0; x <= dxc; x++, ptr += incptrx) {
        if (x > dx - 2 * derrorv) {
            if (x <= dx - derrorv) {
                pixval += incv;
                errorv += incerrorv;
                if (errorv > derrorv) {
                    pixval += sincv;
                    errorv -= derrorv;
                }
            }
            else {
                pixval -= incv;
                errorv -= incerrorv;
                if (errorv < 0) {
                    pixval -= sincv;
                    errorv += derrorv;
                }
            }
        }

        //Integration into Map
        *ptr = ((256 - alpha) * (*ptr) + alpha * pixval) >> 8;
        if (error > 0) {
            ptr += incptry;
            error += diago;
        }
        else error += horiz;
    }
}
```

Figure 6.9 Map_Laser_Ray Code

```
void ts_map_update(ts_scan_t *scan, ts_map_t *map, ts_position_t *pos, int quality)
{
    double c, s, q;
    double x2p, y2p;
    int i, x1, y1, x2, y2, xp, yp, value;
    double add, dist;

    c = cos(pos->theta * M_PI / 180);
    s = sin(pos->theta * M_PI / 180);
    x1 = (int)floor(pos->x * TS_MAP_SCALE + 0.5);
    y1 = (int)floor(pos->y * TS_MAP_SCALE + 0.5);

    //Translate and rotate to robot position
    for (int i = 0; i != scan->nb_points; i++)
    {
        x2p = c * scan->x[i] - s * scan->y[i];
        y2p = s * scan->x[i] + c * scan->y[i];
        xp = (int)floor((pos->x + x2p)* TS_MAP_SCALE + 0.5);
        yp = (int)floor((pos->y + y2p)* TS_MAP_SCALE + 0.5);

        dist = sqrt(x2p * x2p + y2p * y2p);
        add = TS_HOLE_WIDTH / 2 / dist;

        x2p *= TS_MAP_SCALE * (1 + add);
        y2p *= TS_MAP_SCALE * (1 + add);
        x2 = (int)floor(pos->x * TS_MAP_SCALE + x2p + 0.5);
        y2 = (int)floor(pos->y * TS_MAP_SCALE + y2p + 0.5);

        if (scan->value[i] == TS_NO_OBSTACLE) {
            q = quality / 2;
            value = TS_NO_OBSTACLE;
        }
        else {
            q = quality;
            value = TS_OBSTACLE;
        }
        ts_map_laser_ray(map, x1, y1, x2, y2, xp, yp, value, q);
    }
}
```

Figure 6.10 Map Update Code

Section 6.2.3 Conversion to Python

One of the primary goals of this project is to perform a comparison of an HLS based FPGA implementation of a system versus a more standard coding implementation of said system. To do this the system makes use of the FPGAs ARM processor to run a software-only implementation of the desired application. This project uses a Jupyter Notebooks Browser [23] as an external client for sending data or commands to the FPGA. Jupyter Notebooks is a server-client application that allows editing and running notebook documents via a web browser. For this project, all elements of the Jupyter Notebooks side are implemented using Python 3 as Python is an in-built function of the PYNQs ARM Processor. Therefore, following the testing of the original tinySLAM algorithm, development began on a Python Version of the tinySLAM algorithm.

The goal of this Python Version is to act as a reference point for measuring the functionality of the HLS version on the board. Additionally, because Jupyter Notebooks implementations of HLS functions require some external code to operate, this system acts as the foundation for these elements. While Jupyter Notebooks does allow for programming in Python, its connection to the FPGA made it

somewhat slow for development purposes. Therefore, the code was developed using the PyCharm IDE [31] running on a laptop and was transferred to Jupyter Notebooks upon completion. PyCharm is an integrated development environment used in computer programming, specifically for the Python language. When converting the code to Python, the primary algorithm followed the same structure as discussed in Section 6.2.2. However, each element of the algorithm was individually brought converted to Python. Due to the differences between C code and Python some changes were required. For this system, the methods or functions are divided into two types: HLS functions, the functions to be implemented using HLS, and External Functions, which denote functions which are to remain on the Jupyter Notebooks client when the HLS implementation had been developed.

Firstly, in C/C++ code the system uses elements known as structs in order to describe each scan. Structs are a form of composite data type, in which a group of different variables are allocated under one shared block of memory. In the C code the struct details the scan values and is made up of a timestamp, two odometer values q1 and q2, and values relating to the scan itself including arrays for it is x and y coordinates. However, there is no equivalent to a struct in python, instead, the system creates a new class to represent the struct. This allows the data to be accessed in a similar manner as class creation like a struct can define multiple values under a single heading. However, the use of classes requires more memory than structs. Images of the C structs versus the Python class can be seen below (***Figure 6.11*** & ***Figure 6.12***)

```c
typedef struct {
    double x[TS_SCAN_SIZE], y[TS_SCAN_SIZE];
    int value[TS_SCAN_SIZE];
    int nb_points;
} ts_scan_t;

typedef struct {
    int timestamp;
    int q1, q2;
    ts_scan_t scan;
} ts_sensor_data_t;
```

Figure 6.11 C Struct

```python
class ts_sensor_data_t():
    def __init__(self, TS_SCAN_SIZE):
        self.TS_SCAN_SIZE = TS_SCAN_SIZE
        self.timestamp = 0
        self.q1 = 0
        self.q2 = 0
        self.scan = []

        for i in range(0, self.TS_SCAN_SIZE):
            elements = [0]*4
            self.scan.append(elements)
```

Figure 6.12 Python Class

Following this, changes were required in the ***read_sensor_data*** method. This is the method that reads in the values from the data file for use in the program. In the original C version of the system, each line of the file was read in using the *sscanf* function. The sscanf function reads data from an input

file to a formatted string. This allows each value of the line to be read individually. The *strtok* function is then used, which breaks strings into a series of tokens using the delimiter. By repeating these two calls, *sscanf* and *strtok*, the system loops through each piece of data individually and add their values to their respective arrays. It is also important to note that due to the structure of the data file the system must skip the first 10 values as these values contain no data, i.e. whitespace.

However, in Python, the *sscanf* and *strtok* function cannot be used. Instead, the system reads each line in as a single string and splits each element of the string into an array, using the spaces as the delimiters. The system then sets each value as the value at the array location, i.e. timestamp = str[1], q1 = str[2] etc. This method while effective is also susceptible to the whitespaces and therefore must skip irrelevant lines. Additionally, while the C system can use the pointers to create copies of arrays, Python has no equivalent. Thus, a custom method called *passValues* was created to serve that purpose. The ***read_sensor_data*** of both C and Python can be seen below (***Figure 6.13*** & ***Figure 6.14***).

```c
int read_sensor_data(ts_sensor_data_t *data)
{
    int i, j, nb_sensor_data = 0;
    int d[TS_SCAN_SIZE];
    ts_scan_t *scan;
    char *str, line[4000];
    double angle_deg, angle_rad;

    FILE *input = fopen(TEST_FILENAME, "rt");
    do {
        // Read the scan
        str = fgets(line, 4000, input);
        if (str == NULL) break;
        str = strtok(str, " ");
        sscanf(str, "%d", &data[nb_sensor_data].timestamp);
        str = strtok(NULL, " ");
        sscanf(str, "%d", &data[nb_sensor_data].q1);
        str = strtok(NULL, " ");
        sscanf(str, "%d", &data[nb_sensor_data].q2);
        data[nb_sensor_data].q2 = -data[nb_sensor_data].q2;
        for (i = 0; i < 10; i++)
            str = strtok(NULL, " ");
        for (i = 0; i < TEST_SCAN_SIZE; i++) {
            if (str) {
                sscanf(str, "%d", &d[i]);
                str = strtok(NULL, " ");
            }
            else d[i] = 0;
        }

        // Change to (x,y) scan
        scan = &data[nb_sensor_data].scan;
        scan->nb_points = 0;
#define SPAN 1
        // Span the laser scans to better cover the space
        for (i = 0; i < TEST_SCAN_SIZE; i++) {
            for (j = 0; j != SPAN; j++) {
                angle_deg = TEST_ANGLE_MIN + ((double)(i * SPAN + j)) * (TEST_ANGLE_MAX - TEST_ANGLE_MIN) / (TEST_SCAN_SIZE * SPAN - 1);
                angle_rad = angle_deg * M_PI / 180;
                if (i > 45 && i < TEST_SCAN_SIZE - 45) {
                    if (d[i] == 0) {
                        scan->x[scan->nb_points] = TS_DISTANCE_NO_DETECTION * cos(angle_rad);
                        scan->y[scan->nb_points] = TS_DISTANCE_NO_DETECTION * sin(angle_rad);
                        scan->value[scan->nb_points] = TS_NO_OBSTACLE;
                        scan->x[scan->nb_points] += TEST_OFFSET_LASER;
                        scan->nb_points++;
                    }
                    if (d[i] > TEST_HOLE_WIDTH / 2) {
                        scan->x[scan->nb_points] = d[i] * cos(angle_rad);
                        scan->y[scan->nb_points] = d[i] * sin(angle_rad);
                        scan->value[scan->nb_points] = TS_OBSTACLE;
                        scan->x[scan->nb_points] += TEST_OFFSET_LASER;
                        scan->nb_points++;
                    }
                }
            }
        }
        nb_sensor_data++;
    } while (1);

    fclose(input);
    return nb_sensor_data;
}
```

Figure 6.13 C Read_Sensor_Data

```
#Copy array from one to the other
def passValues(self, iscan):
    oscan = iscan[:]
    return oscan

#Read Data from data file
def read_sensor_data(self, data):
    nb_sensor_data = 0
    d = [0] * self.TS_SCAN_SIZE
    num = 0

    # Read in data
    input = open('C:/Users/donal/PycharmProjects/tinySLAM_Python/test_lab.dat', 'r')
    while True:
        rstr = input.readline()
        if not rstr:
            break

        str = re.split(" ", rstr)
        data[nb_sensor_data].timestamp = float(str[0])
        data[nb_sensor_data].q1 = float(str[1])
        data[nb_sensor_data].q2 = float(str[2])
        data[nb_sensor_data].q2 = -data[nb_sensor_data].q2

        for i in range(0, self.TEST_SCAN_SIZE):
            try:
                if (str[i + 18]):
                    d[i] = int(str[i + 18])
                else:
                    d[i] = 0
            except IndexError:
                d[i] = 0
    scan = self.passValues(data[nb_sensor_data].scan)
    nb_points = 0
    SPAN = 1
    for i in range(0, self.TEST_SCAN_SIZE):
        for j in range(0, SPAN):
            angle_deg = self.TEST_ANGLE_MIN + float((i * SPAN + j)) * (self.TEST_ANGLE_MAX - self.TEST_ANGLE_MIN) \
                        / (self.TEST_SCAN_SIZE * SPAN - 1)
            angle_rad = angle_deg * self.M_PI / 180
            if ((i > 45) and (i < self.TEST_SCAN_SIZE - 45)):
                if (d[i] == 0):
                    scan[nb_points][1] = self.TS_DISTANCE_NO_DETECTION * math.cos(angle_rad)
                    scan[nb_points][2] = self.TS_DISTANCE_NO_DETECTION * math.sin(angle_rad)
                    scan[nb_points][3] = self.TS_NO_OBSTACLE
                    scan[nb_points][1] += self.TEST_OFFSET_LASER
                    nb_points += 1
                if (d[i] > (self.TEST_HOLE_WIDTH / 2)):
                    scan[nb_points][1] = d[i] * math.cos(angle_rad)
                    scan[nb_points][2] = d[i] * math.sin(angle_rad)
                    scan[nb_points][3] = self.TS_OBSTACLE
                    scan[nb_points][1] += self.TEST_OFFSET_LASER
                    nb_points += 1

    for i in range(0, self.TS_SCAN_SIZE):
        scan[i][0] = nb_points

    data[nb_sensor_data].scan = self.passValues(scan)
    nb_sensor_data += 1
    num += 1

input.close()
return nb_sensor_data, data
```

Figure 6.14 Python Read_Sensor_Data

The remaining changes to the system were mostly syntactical changes, relating the specific syntaxes of C code versus Python Code, e.g. array syntax. Once these changes had been completed the

system was then run to test its functionality. The final image of the system can be seen below (***Figure 6.15***)

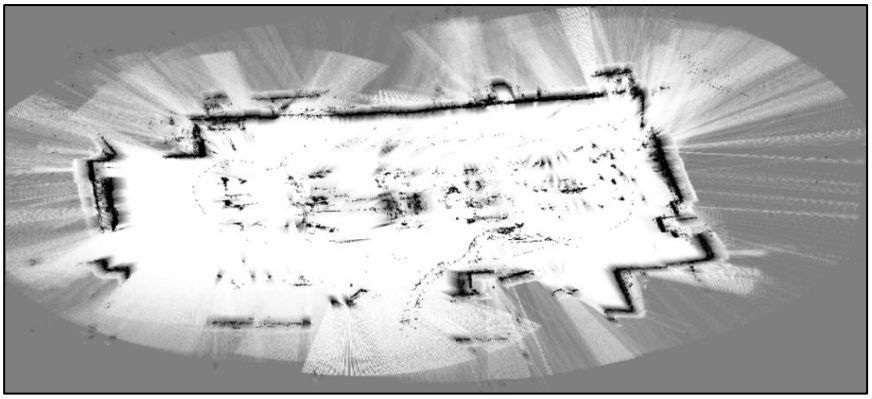


Figure 6.15 Python TinySLAM Test DataSet

As can be seen above, the Python Version was able to successfully run the tinySLAM algorithm and produce the map. When comparing the Python Version to the C Version it was found that the C version was much faster. Both systems were run on the same laptop (Acer Nitro 5) using the same test data set. It was found that the C/C++ version running in Visual Studio [30] was much faster than the Python Version running in PyCharm [31]. These differences in speed appear to come from C/C++ being a statically typed programming language, while Python is a dynamically typed language. In statically typed languages variable types are specified by the user and therefore the system does not need to waste time calculating the variable type. This allows for speed optimization relevant to the data type. Python meanwhile is a dynamically typed language, where variable types are associated with run-time values. This means that the user does not need to specify the variable type making them quicker to program but slower during execution as the system must calculate the relevant data type. The times for both systems, in minutes and seconds, can be seen below.

- Python: 08:46.04
- C/C++: 00:20.05

Section 6.2.4 RPLiDAR & Live Sensor Integration

The purpose of this project was to develop a system which could perform real-time SLAM operations to create a map of the environment. Following the successful implementation of the test data set in both C and Python code, the project moved to the integration of a real-time sensor.

The original tinySLAM system [12] utilised a high precision LiDAR called a Hokuyo URG04 laser scanner [20]. This laser scanner is a high precision LiDAR with 240° area scanning range with 0.36° angular resolution. However, this scanner is very expensive and therefore was deemed unsuitable for this project. Therefore, an alternative known as a Slamtec RPLiDAR A1M8 [21]. This low-cost LiDAR provides a 360-degree 2D laser scanner with a range of than 6 meters at a frequency 5.5Hz. This LiDAR has already been shown to work well within the limits of a tinySLAM system as seen in *A low-cost indoor mapping robot based on TinySLAM algorithm* [22]. These are 2D laser scanners meaning that the system will only map the area at the same height as the device, i.e. at ground level a table will be seen by the table legs rather than the surface area. In order to successfully implement this LiDAR, the RPLiDAR Software Development Kit (SDK) was used [32]. In programming an SDK refers to a collection of software development tools for a product or library distributed in one installable package. The RPLiDAR SDK contains the code and libraries relevant to RPLiDAR execution. Using this SDK library as a basis a modified version of the tinySLAM algorithm was developed to incorporate live sensor data. One of the sample RPLiDAR programs known as *ultra_simple* was used as a basis for RPLiDAR implementation. A flowchart describing the new system can be seen below (***Figure 6.16*** & ***Figure 6.17***).

Figure 6.16 RPLiDAR

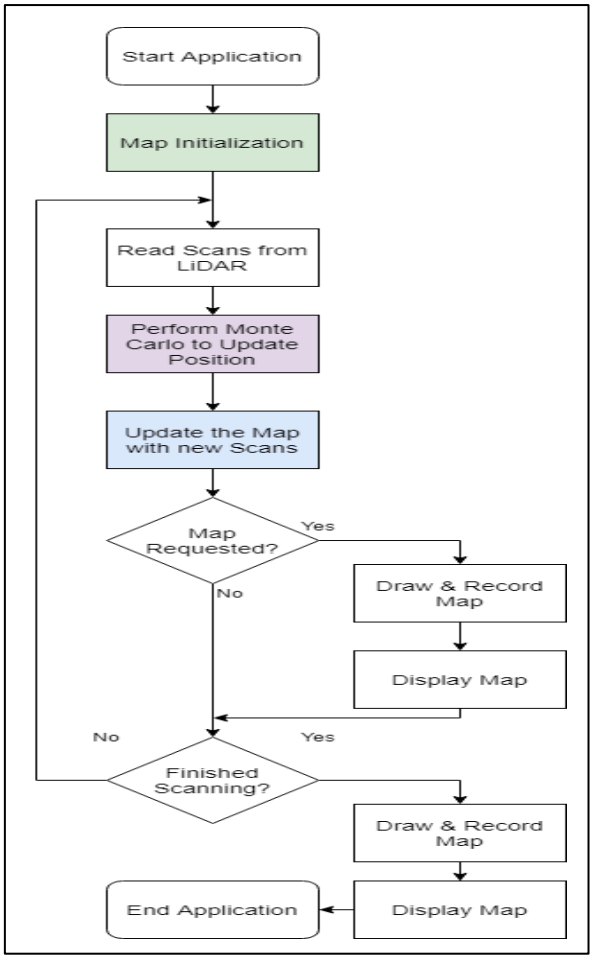


Figure 6.17 RPLiDAR TinySLAM Flowchart

Section 6.2.4.1 C/C++ Version

Using the SDK example the tinySLAM algorithm discussed in Section 6.2.1 & Section 6.2.2, was modified. Three additional methods were added to the code: *checkRPLIDARHealth*, *StartSensor* and *StopSensor*. The *checkRPLIDARHealth* method determines if the LiDAR connection is stable, i.e. if the LiDAR is connected or not. The *StartSensor* method enables the sensor scanning feature, and *StopSensor* method turns off the sensor. Additionally, the **read_data** method was altered. In the previous test data set version, this method read the entire dataset into an array for use in the algorithm. The new version is run once every loop and read the current scans of the sensor. The code of these new elements can be seen below (***Figure 6.18***, ***Figure 6.19***, ***Figure 6.20*** & ***Figure 6.21***).

```
bool checkRPLIDARHealth(RPlidarDriver * drv)
{
    u_result     op_result;
    rplidar_response_device_health_t healthinfo;


    op_result = drv->getHealth(healthinfo);
    if (IS_OK(op_result)) { // the macro IS_OK is the preperred way to judge whether the operation is succeed.
        //printf("RPLidar health status : %d\n", healthinfo.status);
        if (healthinfo.status == RPLIDAR_STATUS_ERROR) {
            fprintf(stderr, "Error, rplidar internal error detected. Please reboot the device to retry.\n");
            return false;
        }
        else {
            return true;
        }

    }
    else {
        fprintf(stderr, "Error, cannot retrieve the lidar health code: %x\n", op_result);
        return false;
    }
}
```

Figure 6.18 CheckRPLiDARHealth

```
void StopSensor()
{
    drv->stop();
    drv->stopMotor();

    RPlidarDriver::DisposeDriver(drv);
    drv = NULL;
}
```

Figure 6.19 StopSensor

```
bool StartSensor()
{
    const char * opt_com_path = NULL;
    _u32         baudrateArray[2] = { 115200, 256000 };
    _u32         opt_com_baudrate = 0;
    u_result     op_result;
    double angle_deg, angle_rad;
    bool setup = true;

    opt_com_path = "\\\\.\\com3";

    // create the driver instance
    drv = RPlidarDriver::CreateDriver(DRIVER_TYPE_SERIALPORT);
    if (!drv) {
        fprintf(stderr, "insufficent memory, exit\n");
        exit(-2);
    }

    rplidar_response_device_info_t devinfo;
    bool connectSuccess = false;

    size_t baudRateArraySize = (sizeof(baudrateArray)) / (sizeof(baudrateArray[0]));
    for (size_t i = 0; i < baudRateArraySize; ++i)
    {
        if (!drv)
            drv = RPlidarDriver::CreateDriver(DRIVER_TYPE_SERIALPORT);
        if (IS_OK(drv->connect(opt_com_path, baudrateArray[i])))
        {
            op_result = drv->getDeviceInfo(devinfo);

            if (IS_OK(op_result))
            {
                connectSuccess = true;
                break;
            }
            else
            {
                delete drv;
                drv = NULL;
            }
        }
    }
    if (!connectSuccess)
    {
        fprintf(stderr, "Error, cannot bind to the specified serial port %s.\n", opt_com_path);
        setup = false;
    }
    else if (!checkRPLIDARHealth(drv))
    {
        printf("LIDAR Health Error\n");
        setup = false;
    }
    else
    {
        drv->startMotor();
        drv->startScan(0, 1);
    }

    return setup;
}
```

Figure 6.20 StartSensor

```
void read_scan(ts_scan_t *scan)
{
    u_result     op_result;
    double angle_deg, angle_rad;

    // fetech result and print it out...
    rplidar_response_measurement_node_hq_t nodes[8192];
    size_t   count = _countof(nodes);

    //Get the scan
    float map[TEST_SCAN_SIZE];
    op_result = drv->grabScanDataHq(nodes, count);

    if (IS_OK(op_result))
    {
        drv->ascendScanData(nodes, count);
        for (int pos = 0; pos < (int)count; ++pos)
        {
            map[pos] = (nodes[pos].dist_mm_q2 / 4.0f);
        }
    }

    scan->nb_points = 0;
#define SPAN 1
    // Span the laser scans to better cover the space
    for (int i = 0; i < TEST_SCAN_SIZE; i++) {
        for (int j = 0; j != SPAN; j++) {
            angle_deg = TEST_ANGLE_MIN + ((double)(i * SPAN + j)) * (TEST_ANGLE_MAX - TEST_ANGLE_MIN) / (TEST_SCAN_SIZE * SPAN - 1);
            angle_rad = angle_deg * M_PI / 180;

            if (i > 45 && i < TEST_SCAN_SIZE - 45) {
                if (map[i] == 0) {
                    scan->x[scan->nb_points] = TS_DISTANCE_NO_DETECTION * cos(angle_rad);
                    scan->y[scan->nb_points] = TS_DISTANCE_NO_DETECTION * sin(angle_rad);
                    scan->value[scan->nb_points] = TS_NO_OBSTACLE;
                    scan->x[scan->nb_points] += TEST_OFFSET_LASER;
                    scan->nb_points++;
                }
                if (map[i] > TEST_HOLE_WIDTH / 2) {
                    scan->x[scan->nb_points] = map[i] * cos(angle_rad);
                    scan->y[scan->nb_points] = map[i] * sin(angle_rad);
                    scan->value[scan->nb_points] = TS_OBSTACLE;
                    scan->x[scan->nb_points] += TEST_OFFSET_LASER;
                    scan->nb_points++;
                }
            }
        }
    }
}
```

Figure 6.21 RPLiDAR Read Scan

In the test data set the system is set to loop through each value until completion. However, in a live sensor scenario, this cannot be done as the sensor will continually search for new data. Therefore, a new end condition was necessary. For this system, it was decided that a user-controlled end condition would be suitable. This version of the system is set to loop until the Escape Key on the computer keyboard is pressed and held. Additionally, the RPLiDAR is a 360° LiDAR, while the Hokuyo-URG04 only dealt with 240°. Therefore, the angular range of the system also needed to be changed. This is accomplished using the following code (***Figure 6.22***, ***Figure 6.23*** & ***Figure 6.24***).

```
//Pressed and Hold the Escape Key to Exit the Loop
if (GetAsyncKeyState(VK_ESCAPE) & MSB)
{
    go = false;
}
```

Figure 6.22 Escape Loop

```
#define TEST_ANGLE_MIN -120
#define TEST_ANGLE_MAX +120
```

Figure 6.23 Old Angles

```
#define TEST_ANGLE_MIN -180
#define TEST_ANGLE_MAX 180
```

Figure 6.24 New Angles

With these modifications made the system could be tested. As no robotic system has been developed for this project, the test was carried out while the user was carrying the LiDAR in hand and

moving around the environment. The finalised scan of the room can be seen below (***Figure 6.25*** & ***Figure 6.26***).

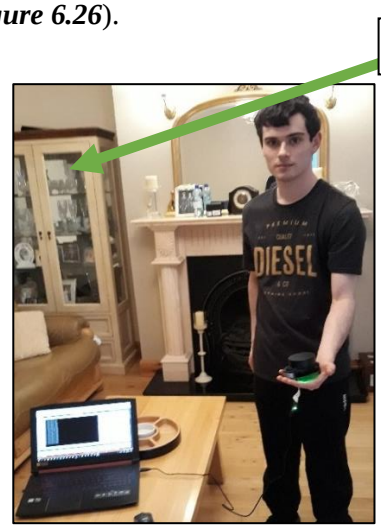


Figure 6.25 System Operation

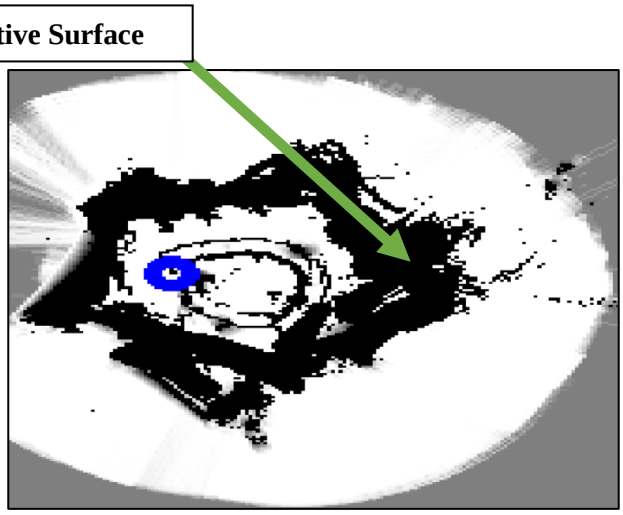


Figure 6.26 RPLiDAR tinySLAM Output

The system creates a map of the room successfully. The corner of the room contains a reflective surface, which results in a noise being detected as shown above. It is important to note the system's dependence on the Monte Carlo method. The Monte Carlo method is used to calculate the systems current position and relies on comparisons to previous data. Therefore, the user must move at slow enough speed to allow the system to scan previous objects for comparison. If the user were to move too fast the system may lose track of its position and therefore result in an incorrect map. This error could be solved through the use of an external position tracking method, such as the odometers mentioned by the tinySLAM authors. However, due to time constraints development of such an alternative was not possible.

Section 6.2.4.2 Python Version

The SDK for the RPLiDAR is designed to operate using a C/C++ system. However, this causes issues when attempting to translate the system to a Python-based implementation. While there are custom made libraries online which translate the SDK features to a Python setting, initial testing found that these libraries were unreliable or gave different results than the original SDK. Therefore, it was decided that the SDK would be directly ported to Python by creating a library to reference the SDK. To do this the SDK is referenced through a shared library. The computer this project was developed on

used a Windows 10 operating system. Therefore, in order to reference the library using a Python script the SDK was converted into a dynamic link library (.DLL). DLLs are libraries that contain code or data that can be used by multiple programs. Most Windows libraries involve the use of a DLL. For this project, a custom DLL was developed which acts as a Python wrapper for the existing RPLiDAR library.

To simplify development of the DLL a shortened version of the RPLiDAR library containing only the required methods was created. To do this a copy of the SDK was created and the *ultra_simple* method was modified to denote the functions required. This file contained three methods *StartSensor*, *StopSensor* and *read_scan*, which started, stopped, and read data from the sensor, respectively. Once this was done the system was built as a static library (.LIB) using the Visual Studio tools. This static library allows the RPLiDAR methods to be called in other C/C++ projects at runtime.

Using the static libraries **rplidar.lib** (rplidar functions) and **sensor.lib** (three methods above), a new project was created called LiDAR_Lib. This project acts as a wrapper project to convert the static libraries into a single DLL for use in Python. To do this the **extern** "C" **__declspec(dllexport)** method was used. This method defines the code as functions that can be used externally to C code. The *declspec(dllexport)* portion of the code defines the methods as a DLL file. Both the .lib files were imported into this project and used to define the wrapper methods. Once this was done the project was built as a DLL file. The code for the wrapper project can be seen below (***Figure 6.27***).

```
#include <iostream>
#include "LiDAR.h"

using namespace std;
#define DLLIMPORT_EXPORT extern "C" __declspec(dllexport)

//Export Sensor Start-up to Python
DLLIMPORT_EXPORT int BeginSensor(const char* port)
{
    //"\\\\.\\com3"
    string vals = port;
    if (StartSensor(vals))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

//Export Sensor End to Python
DLLIMPORT_EXPORT void EndSensor()
{
    StopSensor();
}

//Export Read Scan to Python
DLLIMPORT_EXPORT void ReadSensor(float map[682])
{
    read_scan(map);
}
```

Figure 6.27 RPLiDAR DLL Wrapper

Once the DLL had been successfully created, the RPLiDAR functions code be imported into the Python script as seen below(***Figure 6.28***)

```
self.my_dll = 'C:/Users/donal/source/repos/LiDAR_Lib/x64/Debug/LiDAR_Lib.dll'
self.myLiDAR = cdll.LoadLibrary(self.my_dll)
```

Figure 6.28 Python Import DLL

Once the library had been imported to the Python script, the system was modified to match the changes described in the Section 6.2.4.1 for the C/C++ version. The program was then run and returned a similar image to the C output as seen below (***Figure 6.29***).

Figure 6.29 Python RPLiDAR

Once again the Python Version was found to be much slower than the C/C++ version due to the differences in coding languages. Additionally, due to the previously discussed restrictions caused by the Monte Carlo method, the user must move at an appropriate speed to ensure object detection. This means that due to the slowdown caused by the Python Interpreter the user must move even slower in order to ensure the system can accurately calculate the current position. Otherwise, the system will lose track of position and start mapping incorrect values.

Section 7. Combined System Development

Section 7.1 Introduction to Combined System Analysis

Following the completion of the standalone tinySLAM system and development of basic High-Level Synthesis (HLS) programs, development shifted towards combining these two applications into a single system. This process combination is the primary goal of the entire project and involves the development and deployment of the tinySLAM algorithm on the PYNQ-Z2 FPGA.

Development of the combined system is split into two parts, development of HLS tinySLAM using test data set and the development of the HLS tinySLAM using real-time data. This Section will also cover the deployment of pure Python tinySLAM for both types of data to compare system functionality.

Section 7.2 Development HLS TinySLAM with Test Data

The first stage of combining the two-system relates to the integration of the tinySLAM algorithm into the HLS format. This project focused on the deployment of the modified tinySLAM system detailed in Section 6.2.2. As stated in Section 5, HLS development involves communication between the Processing System (PS) and Programmable Logic (PL) components of the FPGA. This project focuses on implementing the three primary components of tinySLAM as described in *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12] using HLS. The remaining elements of the algorithm, the image processing and display code, are deployed using Pure Python on the PS component of the FPGA. A system flowchart of the project can be seen below (***Figure 7.1*** & ***Figure 7.2***).

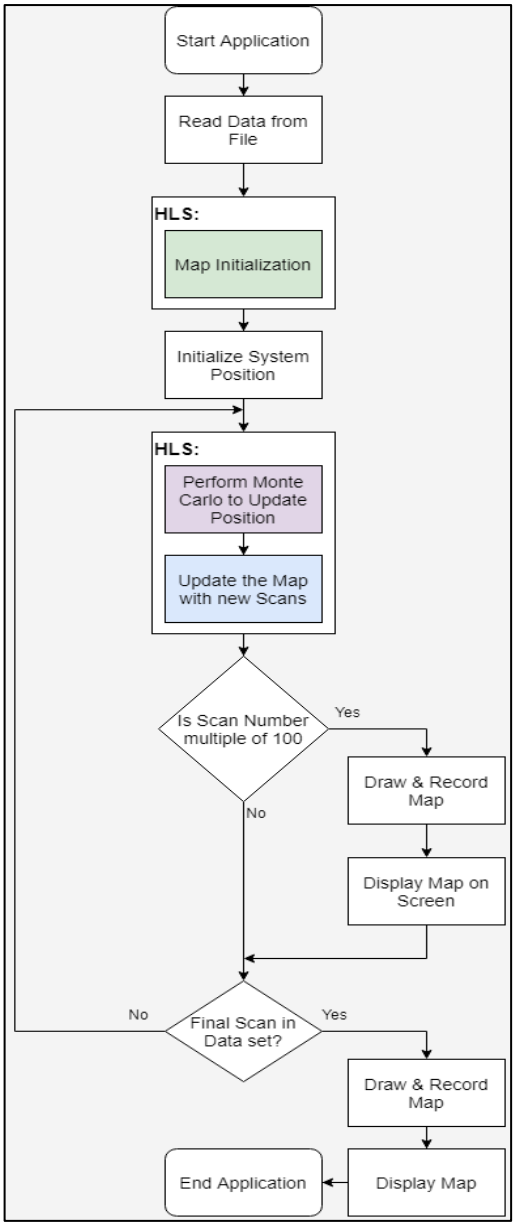


Figure 7.1 HLS tinySLAM Test Data Set

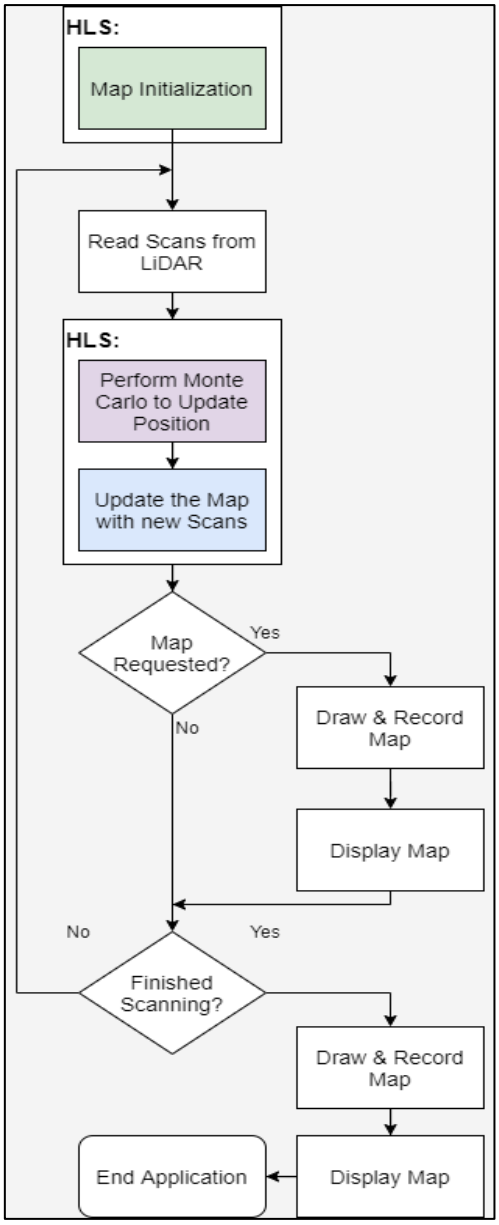


Figure 7.2 HLS tinySLAM Live Data

As can be seen above the systems for both the Test Data Set and the Live Data are very similar with the primary difference relating to when the data is read. In FPGAs, external peripherals are managed by the Processing System (PS). Therefore, the reading of data from the LiDAR occurs in the Python side and will be transferred across the AXI to the tinySLAM algorithm in the Programmable Logic (PL) of the board. Additionally, as both the Monte Carlo and Map Update components of the tinySLAM algorithm occur immediately after each other, the system has been designed to combine these into one reference called ***HLS_Main***. This is done to reduce the amount of data being sent across the AXI and thus increase the overall speed of the system while reducing the number of memory

resources required. The map initialization code ***HLS_Map*** remains independent as it only runs once at the beginning of the program, It must be noted that this code could be further optimized as the ***HLS_Map_Init*** function would be better suited to a pure Python implementation; however, the goal of this project was to demonstrate the capabilities of HLS to encode complex systems and therefore the system uses HLS for as much code as possible rather than what is considered optimal distribution.

Section 7.2.1 HLS Modifications to TinySLAM

Although the original tinySLAM does use C/C++ code, there are many elements which are acceptable in standard coding which cannot be used in HLS. This is normally due to HLS converting systems into Hardware where certain programming features have no hardware equivalent. An example of this can be seen with the **return** statement as discussed in Section 5.1. Therefore, the tinySLAM algorithm required some modifications before it could be used in the HLS process.

Section 7.2.1.1 General Modifications

Before any specific elements of the tinySLAM algorithm were analysed a series of general modifications were required. Firstly, there is the size of the system. The original tinySLAM operates on a laptop or desktop computer, which would have a lot of space for developing a large map. However, the FPGA is a much smaller device with much less memory. Therefore, the size and resolution of the map had to be reduced in order to fit onto the FPGA. The tinySLAM algorithm already includes a scaling factor to convert the data into a map of the desired size. Therefore, all that was required was changing the scale and map size of the system. The scale was changed such that the image maintained the same scale as the original image on the smaller map. Images of the scale change can be seen below (***Figure 7.3*** & ***Figure 7.4***)

```
#define TS_SCAN_SIZE 683
#define TS_MAP_SIZE 2048
#define TS_MAP_SCALE 0.1
#define TS_DISTANCE_NO_DETECTION 4000
#define TS_NO_OBSTACLE 65500
#define TS_OBSTACLE 0
#define TS_HOLE_WIDTH 600
```

Figure 7.3 Original Scale

```
#define TS_SCAN_SIZE 683
#define TS_MAP_SIZE 360
#define TS_MAP_SCALE 0.017578125
#define TS_DISTANCE_NO_DETECTION 4000
#define TS_NO_OBSTACLE 65500
#define TS_OBSTACLE 0
#define TS_HOLE_WIDTH 600
```

Figure 7.4 HLS Scale

Following the scale changes the next changes relates to the format of the sensor data and the map array. In the original tinySLAM algorithm both the sensor data and the map array were stored as structs. These structs contained the arrays of elements. However, for this system, the scans and the map must be sent back and forth across the AXI bus. As these elements are very large AXI Stream was found to be incompatible as AXI Stream has a set limit on the amount of data it can carry. Therefore, AXI-Lite, a memory-mapped AXI was used. This AXI would allow for the transmission of the large data arrays; however, AXI-Lite cannot support structs only single value data types, integers, or bytes. Therefore, the arrays were modified to convert the data into arrays of single values rather than structs containing these arrays. Images of the changes can be seen below (***Figure 7.5*** & ***Figure 7.6***)

```
typedef unsigned short ts_map_pixel_t;

typedef struct {
    ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE];
} ts_map_t;

typedef struct {
    double x[TS_SCAN_SIZE];
    double y[TS_SCAN_SIZE];
    int value[TS_SCAN_SIZE];
    int nb_points;
} ts_scan_t;
```

Figure 7.5 Original Format

```
float scan[TS_SCAN_SIZE][4],ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE]
```

Figure 7.6 New Format

As can be seen above the map remains mostly unchanged as the struct for map merely contained the array while the HLS version deals with the array directly. However, the scan is significantly changed. The original scan contains 3 arrays, one for X, Y, and the Value along with a value called nb_points which represents the number of points in the scan. However, the HLS version uses a 2D array to represent all the data. The array is split so that scan[?][1] is X[?], scan[?][2] is Y[?], scan[?][3] is Value[?]. All values for scan[?][0] contain the same number nb_points.

Finally, an HLS wrapper method has been added to the system. Normally HLS programs only contain a single function; however, this project contains two ***HLS_Main*** and ***HLS_Map_Init***. In order to operate an HLS system, the AXI references must be placed on the outermost function (top function) of the program. Therefore, in order to handle two functions, the system includes a wrapper class which takes values in from the AXI and passes them to the correct function and passes the results back to the AXI upon completion. The system is based on the transmission of an AXI-Stream value called Run. This Run value acts as a form of chip enable and decode, where the value determines which function is used. The Run value is sent across AXI-Stream, the advantage of this is that AXI-Stream forces the system to wait until a value has been sent across. This prevents the system from continuously calculating data and prevents the system from running the methods before the entire AXI-Lite array (scan) has been transferred. An image of the wrapper code can be seen below (***Figure 7.7***).

```
//HLS Wrapper Function (Switch Based System)
void HLS_tinySLAM(int scan[TS_SCAN_SIZE][4], ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE], stream_pos in_pos[3], stream_pos out_pos[3], stream_type Run[1])
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=map
#pragma HLS INTERFACE s_axilite port=scan
#pragma HLS INTERFACE axis port=in_pos
#pragma HLS INTERFACE axis port=out_pos
#pragma HLS INTERFACE axis port=Run
#pragma HLS allocation instances=hls::cosf limit=1 function
#pragma HLS allocation instances=hls::sinf limit=1 function
#pragma HLS allocation instances=hls::abs limit=1 function
#pragma HLS inline region recursive

    ap_int<32> check = Run[0].data;
    converter convert;
    stream_pos temp_pos[3];
    float scanner[TS_SCAN_SIZE][4];

    //Convert Bits to Floats
    for(int i = 0; i < TS_SCAN_SIZE; i++)
    {
        for(int j = 0; j < 4; j++)
        {
            convert.i = scan[i][j];
            scanner[i][j] = convert.f;
        }
    }

    //If Run == 1-> HLS_map_init
    if(check == 1)
    {
        HLS_map_init(map);
    }
    //If Run == 2-> Main
    if(check == 2)
    {
        ////////////////////////////////Monte Carlo/////////////////////////////////////////
        HLS_monte_carlo(scanner, map, in_pos, temp_pos);

        ////////////////////////////////Update Map/////////////////////////////////////////
        //Transfer Back Position
        for(int i = 0; i < 3; i++)
        {
            out_pos[i].data = temp_pos[i].data;
            out_pos[i].strb = temp_pos[i].strb;
            out_pos[i].keep = temp_pos[i].keep;
            out_pos[i].user = temp_pos[i].user;
            out_pos[i].last = temp_pos[i].last;
            out_pos[i].id = temp_pos[i].id;
            out_pos[i].dest = temp_pos[i].dest;
        }

        int value = 50;
        HLS_map_update(scanner, map, temp_pos, value);
    }
}
```

Figure 7.7 HLS tinySLAM Wrapper

As can be seen in the code above the system wraps the two methods. When Run equals 1 the program runs ***HLS_Map_Init***, but when Run equals 2 the system runs the Monte Carlo and Map Update Methods, referred to as ***HLS_Main*** in the Python code. Additionally, you can see that the system takes in the previous position and outputs the updated position via AXI-Stream. As stated previously, the scan method requires float values for the X and Y components; however, the AXI-Lite interface cannot handle floating-point data. Therefore, the data has been converted into unsigned integer data at the Python side. In order to use the data, the system must convert the bits of the unsigned integers back into floating-point numbers. This is done using a function of C/C++ code called a Union. A Union is a data type in C programming that allows different data types to be stored in the same memory locations. The data is stored as bits allowing the union to convert the data between the different data types. In this system the Union, called **convert** takes in the integer value and stores it. The program then reads the bits of this integer as a float and stores it in a new scanning array. The Union code and the use of the converter can be seen below (***Figure 7.8*** & ***Figure 7.9***).

```
typedef union {
  int i;
  float f;
} converter;
```

Figure 7.8 Union Example

```
float scanner[TS_SCAN_SIZE][4];
converter convert;

//Convert Bits to Floats
for(int i = 0; i < TS_SCAN_SIZE; i++)
{
    for(int j = 0; j < 4; j++)
    {
        convert.i = scan[i][j];
        scanner[i][j] = convert.f;
    }
}
```

Figure 7.9 Union Conversion

The HLS wrapper also includes many pragmas used in the system. These primarily consist of the different AXI interface pragmas discussed in Section 5. Additional pragma functionality was identified using the methods laid out in the book *Parallel Programming for FPGAs, The HLS Book* [17]. Firstly, the system defines the `#pragma HLS allocation instances=XXX limit=1 function` which limits the system into creating a single copy of function XXX. For example, the Cosine and Sine functions in HLS require a lot of space. Therefore, when this function is defined the system prevents multiple Cosine and Sine elements being created, instead, all uses of Cosine and Sine use the same hardware elements. Finally, the system sets the `#pragma HLS inline region recursive`. Normally when a program wishes to run a method the function is “called”, i.e. the system

moves to the location of that function in memory and runs the code. Function inlining is a process where the system removes functions calls and replaces them with copies of the desired function. This process removes a function as a separate entity in the hierarchy instead merging the functions together. Inlined functions run a little faster than the normal functions as calling overheads are saved; however, system memory suffers due to multiple copies of the same code. However, in this project the system only calls each method once, and hence function inlining is used in order to ensure the system meets the timing requirements of the hardware without any memory penalty. Additionally, the use of the previously mentioned allocation pragmas removes all other repeating data.

Section 7.2.1.2 Map Initialization Modifications

Following the general modifications, modifications to the primary tinySLAM elements began. These components were based on the updated tinySLAM methods mention in Section 6. Firstly, there is the Map Initialization phase. This phase loops through each pixel in the image to create a blank map. In the original system, this process calculates the initval and loops through each value in the map setting each value equal to the initval. In the HLS tinySLAM, the initval was instead stored as a constant value reducing resource usage. Additionally, as previously mentioned, the struct for the map array has been removed thus the system operates on the direct map location in memory. The changed code can be seen below (***Figure 7.10*** & ***Figure 7.11***).

```
void ts_map_init(ts_map_t *map)
{
    int x, y, initval;
    ts_map_pixel_t *ptr;
    initval = (TS_OBSTACLE + TS_NO_OBSTACLE) / 2;
    for (ptr = map->map, y = 0; y < TS_MAP_SIZE; y++) {
        for (x = 0; x < TS_MAP_SIZE; x++, ptr++) {
            *ptr = initval;
        }
    }
}
```

Figure 7.10 Old Map Init

```
//Map Initialisation
void HLS_map_init(ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE])
{
    for (int i = 0; i < (TS_MAP_SIZE * TS_MAP_SIZE); i++)
    {
        map[i] = initval;
    }
}
```

Figure 7.11 New Map Init

Section 7.2.1.3 Monte Carlo Modifications

The map position calculation or Monte Carlo method also required additional changes. Firstly, the Monte Carlo method used is dependent on the use of the ***Distance_Scan_To_Map*** described in *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12]. Therefore, this algorithm needed to be modified to better suit the HLS implementation. The majority of this function remains unchanged. The scan, map values have been changed to use their AXI-Lite Versions as previously stated, and the position value has been changed to use AXI-Stream. As discussed in Section 5.1, the use of a **return** statement is not applicable in HLS, therefore, the distance value is returned using a pointer implementation. The primary change to the distance calculation is that the system has been redesigned to use floats instead of doubles. This was done to reduce the memory usage of the data. Additionally, the Cosf, Sinef and Floorf built into HLS were used instead of the more standard Cos, Sine and Floor functions found in the C math library. This was done as these HLS specific implementations have been optimized for hardware rather than the more standard math libraries. Images of the new code can be seen below (***Figure 7.12*** & ***Figure 7.13***).

```
//////////////////////////////////////////////////MAP DISTANCE MEASURE//////////////////
int ts_distance_scan_to_map(ts_scan_t *scan, ts_map_t *map, ts_position_t *pos)
{
    double c, s;
    int i, x, y, nb_points = 0;
    int64_t sum;

    c = cos(pos->theta * M_PI / 180);
    s = sin(pos->theta * M_PI / 180);

    //Translate and rotate scan to robot position and compute the distance.
    for (i = 0, sum = 0; i != scan->nb_points; i++) {
        if (scan->value[i] != TS_NO_OBSTACLE) {
            x = (int)floor((pos->x + c * scan->x[i] - s * scan->y[i]) * TS_MAP_SCALE + 0.5);
            y = (int)floor((pos->y + s * scan->x[i] + c * scan->y[i]) * TS_MAP_SCALE + 0.5);

            //Check Boundaries
            if (x >= 0 && x < TS_MAP_SIZE && y >= 0 && y < TS_MAP_SIZE) {
                sum += map->map[y * TS_MAP_SIZE + x];
                nb_points++;
            }
        }
    }
    if (nb_points) sum = sum * 1024 / nb_points;
    else sum = 2000000000;
    return (int)sum;
}
```

Figure 7.12 Old Map Distance

```
//Distance Calculation
void HLS_distance(float scan[TS_SCAN_SIZE][4], ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE], stream_pos pos[3], int *output)
{
    float scanx, scany, posx, posy, postheta;
    float c, s;
    int i, x, y, nb_points = 0;
    int scanvalue, scannb_points;
    int64_t sum;

    posx = pos[0].data;
    posy = pos[1].data;
    postheta = pos[2].data;


    c = hls::cosf(postheta * M_PI / 180);
    s = hls::sinf(postheta * M_PI / 180);

    scannb_points = ((int) scan[0][0]);

    //Translate and rotate scan to robot position and compute the distance.
    for (i = 0, sum = 0; i != scannb_points; i++)
    {
        scannb_points = ((int) scan[i][0]);
        scanx = scan[i][1];
        scany = scan[i][2];
        scanvalue = (int) scan[i][3];

        if (scanvalue != TS_NO_OBSTACLE)
        {
            x = (int)hls::floorf((posx + c * scanx - s * scany) * TS_MAP_SCALE + 0.5);
            y = (int)hls::floorf((posy + s * scanx + c * scany) * TS_MAP_SCALE + 0.5);

            //Check Boundaries
            if (x >= 0 && x < TS_MAP_SIZE && y >= 0 && y < TS_MAP_SIZE) {
                sum += map[y * TS_MAP_SIZE + x];
                nb_points++;
            }
        }
    }
    if (nb_points) sum = sum * 1024 / nb_points;
    else sum = 2000000000;
    *output = (int)sum;
}
```

Figure 7.13 New Map Distance

Once the Distance Scanning method had been updated the Monte Carlo method could also be changed. Firstly, due to the HLS wrapper inlining pragma, all function calls in the system would be removed and replaced with copies of the function hardware. The Monte Carlo method involves the use of multiple calls to the Distance Scan method. Therefore in order to prevent memory waste, the system was limited to the use of one copy of the Distance method using the allocation method described in Section 7.2.1.1. This prevents multiple copies of the Distance Scan being created. Following this, the Monte Carlo method was modified to use the AXI-Lite and AXI-Stream data types for the scan, map, and position values. The Monte Carlo method operates by creating and updating internal position values and using the finalised value as the output. By default, these internal values are created using Distributed RAM. Distributed RAM, otherwise known as LUT RAM, is a form of system memory which makes use of logic blocks implemented as a lookup table (LUT), i.e. a memory where the address signal are the inputs and the outputs are stored. This RAM is primarily used for temporary values in a system. However, it was found that the system did not have enough LUT RAM to support these elements,

therefore the system was set to use Block RAM instead using the methods defined in *Parallel Programming for FPGAs, The HLS Book* [17]. Block RAM, or BRAM, is used for storing large amounts of data inside of an FPGA. They are the primary source of memory for the IP Logic Blocks and are typically used to store large datasets or data of a set size. The BRAM also handles storage for much of the data coming through the AXI System. The pragma `#pragma HLS RESOURCE variable=XXX core=RAM_1P_BRAM` forces the Vivado HLS tool to use a single port BRAM for storing the variable XXX.

Following storage management, it was found that the **random** method used in the C code was not acceptable. In HLS there is no automatic equivalent to a random number generator, and therefore all C/C++ random library functions cannot operate. To circumvent this issue a custom pseudo-random number generator was created. This random number generator is based off a Linear Feedback Shift Register (LFSR). In computing, an LFSR is a form of shift register which uses a version of its previous output as the system input. An LFSR *"is a shift register that, when clocked, advances the signal through the register from one bit to the next most-significant bit"* [33]. LFSRs are a commonly used element of computing both in hardware and in software. LFSR have commonly been used as pseudo-random number generators using the value of the last output to create a new output. The larger the bit rating of the LFSR the more "random" the numbers appear. Upon further research, it was decided that the pseudo-random number generator would use the LFSR method outlined in the book *The C Programming Language 2nd Edition* [34]. This method was chosen as it appears to be what the C random function, **rand()**, is based off, using a similar calculation as seen in the GNU C Library [35] [36]. This random method uses the C long data type, creating a 64-bit value. This 64-bit value is then passed through the random function to create a new number. When the random method is called again the previous output of the random method is used as the input. For the purposes of this project the first value, sometimes referred to as a seed value, is set to 1. The code for the updated Monte Carlo and the LFSR random method can be seen below (***Figure 7.14*** & ***Figure 7.15***).

```
//Use the Monte Carlo Method
void HLS_monte_carlo(float scan[TS_SCAN_SIZE][4], ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE], stream_pos in_pos[3], stream_pos out_pos[3])
{
#pragma HLS allocation instances=HLS_distance limit=1 function
    int currentdist, bestdist, lastbestdist;
    int counter = 0;
    stream_pos curpos[3], bestpos[3], lastbestpos[3];

#pragma HLS RESOURCE variable=curpos core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=bestpos core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=lastbestpos core=RAM_1P_BRAM

    //Default Position from Input Position
    for (int i = 0; i < 3; i++)
    {
        curpos[i].data = bestpos[i].data = lastbestpos[i].data = in_pos[i].data;
        bestpos[i].strb = in_pos[i].strb;
        bestpos[i].keep = in_pos[i].keep;
        bestpos[i].user = in_pos[i].user;
        bestpos[i].last = in_pos[i].last;
        bestpos[i].id = in_pos[i].id;
        bestpos[i].dest = in_pos[i].dest;
    }

    //Get Distance using Input Position and set Default Distance
    HLS_distance(scan, map, curpos, &currentdist);
    bestdist = lastbestdist = currentdist;

    //Loop 1000 times for clarity
    do {
        //Update Current Position with last best position
        for (int i = 0; i < 3; i++)
        {
            curpos[i].data = lastbestpos[i].data;
        }
        //Random position
        curpos[0].data += 50 * (((double)myrand()) / RAND_MAX - 0.5);
        curpos[1].data += 50 * (((double)myrand()) / RAND_MAX - 0.5);
        curpos[2].data += 50 * (((double)myrand()) / RAND_MAX - 0.5);

        //Calculate Distance
        HLS_distance(scan, map, curpos, &currentdist);

        //If Distance Less than Best Value, Best Value = Current Value
        if (currentdist < bestdist)
        {
            bestdist = currentdist;
            for (int i = 0; i < 3; i++)
            {
                bestpos[i].data = curpos[i].data;
            }
        }

        //Otherwise Increase Counter
        else
        {
            counter++;
        }

        //Ignore first 100 values for precision purposes
        if (counter > 100) {
            //Update Best Position
            if (bestdist < lastbestdist)
            {
                for (int i = 0; i < 3; i++)
                {
                    lastbestpos[i].data = bestpos[i].data;
                }
                lastbestdist = bestdist;
                counter = 0;
            }
        }
    } while (counter < 1000);

    //Transfer Back Position
    for(int i = 0; i < 3; i++)
    {
        out_pos[i].data = bestpos[i].data;
        out_pos[i].strb = bestpos[i].strb;
        out_pos[i].keep = bestpos[i].keep;
        out_pos[i].user = bestpos[i].user;
        out_pos[i].last = bestpos[i].last;
        out_pos[i].id = bestpos[i].id;
        out_pos[i].dest = bestpos[i].dest;
    }
}
```

Figure 7.14 HLS Monte Carlo Method

```
//Seed Value
static unsigned long val = 1;

//Pseudo Random Number Generator (Based on C::rand() and LFSR)
int myrand()
{
    val = (val * 1103515245) + 12345;
    return((unsigned)(val / 65536) % RAND_MAX);
}
```

Figure 7.15 LFSR Random Code

Section 7.2.1.4 Map Update Modifications

Finally, the last component of the tinySLAM algorithm to be implemented in HLS is the Map Update Method. This function deals with the method of adding new scans to the map. As discussed in Section 6.2.2.3, the Map Update function is dependant on the ***Map_Laser_Ray*** function. This function loops through the map and uses the coordinate geometry value of each scan, calculated in the Map Update method, to calculate the location of each scan value and add said value to the map. The ***Map_Laser_Ray*** function remains mostly unchanged, the only changes required were the use of the HLS versions of the Cosine, Sine and Absolute methods rather than the use of the C Math libraries, as discussed in Section 7.2.1.3, and changes relating to the map format. As the map is no longer contained within a struct the system can no longer use C pointers to reference the map. Instead, the system uses a standard array reference to call the map value. The difference in map calls can be seen below(***Figure 7.16*** & ***Figure 7.17***)

```
ptr = map->map + y1 * TS_MAP_SIZE + x1;
                 ↓
*ptr = ((256 - alpha) * (*ptr) + alpha * pixval) >> 8;
```

Figure 7.16 Old Map Format

```
ptr = y1 * TS_MAP_SIZE + x1;
      ↓
map[ptr] = ((256 - alpha) * (map[ptr]) + alpha * pixval) >> 8;
```

Figure 7.17 New Map Format

Following the changes to the ***Map_Laser_Ray*** method, the main Map Update method renamed ***HLS_Map_Update*** could be modified. Similar to the Monte Carlo method, the Map Update method includes the use of an allocation pragma to limit the number of copies of the Map Laser Ray method to one, for memory efficiency purposes. Following this, all the Cosine, Sine, Sqrt and Floor methods of the code are replaced with the HLS versions instead of the C Math libraries to for system optimization. The only other change required involved replacing the old scan referencing with the new AXI

compatible referencing, due to the removal of the scan struct. The system reads the value scan value from the array at the start of each loop. It is important to note that in the original system there was only a single nb_point value, while the current system uses an array of nb_points due to the 2D array used to replace the struct. This scan array is the same array that crosses the AXI. In AXI arrays the system **must** read all values of the AXI array **sequentially**, i.e. value [1][1] must be read before [2][1] or [1][2]. Due to this even though all values of the nb_points array, scan[i][0], are the same, the value must be read in regardless. The code for the HLS version of the Map Update can be seen below (***Figure 7.18***).

```
//Map Update
void HLS_map_update(float scan[TS_SCAN_SIZE][4], ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE],  stream_pos pos[3], int quality)
{
#pragma HLS allocation instances=HLS_map_laser_ray limit=1 function

    //Initialize Variables
    float c, s, q, x2p, y2p, add, dist;
    float scanx, scany, posx, posy, postheta;
    int i, x1, y1, x2, y2, xp, yp, value;
    int scanvalue, scannb_points;

    posx = pos[0].data;
    posy = pos[1].data;
    postheta = pos[2].data;

    //Calculate Cosine, Sine and X Y coordinates
    c = hls::cosf(postheta * M_PI / 180);
    s = hls::sinf(postheta * M_PI / 180);
    x1 = (int)hls::floorf(posx * TS_MAP_SCALE + 0.5);
    y1 = (int)hls::floorf(posy * TS_MAP_SCALE + 0.5);

    scannb_points = ((int) scan[0][0]);
    scanx = scan[0][1];
    scany = scan[0][2];
    scanvalue = ((int) scan[0][3]);

    //Translate and rotate can to robot position
    for (i = 0; i != scannb_points; i++)
    {
        //Cannot Read Data from same location twice
        if(i > 0)
        {
            scannb_points = ((int) scan[i][0]);
            scanx = scan[i][1];
            scany = scan[i][2];
            scanvalue = ((int) scan[i][3]);
        }
        //Calculate Positional Values
        x2p = c * scanx - s * scany;
        y2p = s * scanx + c * scany;

        xp = (int)hls::floorf((posx + x2p)* TS_MAP_SCALE + 0.5);
        yp = (int)hls::floorf((posy + y2p)* TS_MAP_SCALE + 0.5);
        dist = hls::sqrtf(x2p * x2p + y2p * y2p);

        add = TS_HOLE_WIDTH / 2 / dist;
        x2p *= TS_MAP_SCALE * (1 + add);
        y2p *= TS_MAP_SCALE * (1 + add);

        x2 = (int)hls::floorf(posx * TS_MAP_SCALE + x2p + 0.5);
        y2 = (int)hls::floorf(posy * TS_MAP_SCALE + y2p + 0.5);

        //If No Obstacle Found Set to White
        if (scanvalue == TS_NO_OBSTACLE)
        {
            q = quality / 2;
            value = TS_NO_OBSTACLE;
        }

        //Otherwise Set to Black
        else
        {
            q = quality;
            value = TS_OBSTACLE;
        }

        HLS_map_laser_ray(map, x1, y1, x2, y2, xp, yp, value, q);
    }
}
```

Figure 7.18 HLS Map Update

Section 7.2.2 HLS Synthesis and Vivado Design Suite

Once the C code had been modified to effectively use HLS the system was then put the HLS process for deployment. For this project, the system relies on displaying images in order to see the map and verify if the system is working correctly. While Vivado HLS does support the OpenCV library for image processing applications, the C simulator in Vivado HLS does not support the *imshow* or *imread* needed to display the image. Therefore in order to test the modified C code, the code was copied into a Visual Studio [30] project. This project would run the code found in Vivado HLS, with the HLS specific pragmas and methods replaced or removed. This test resulted in the following map (***Figure 7.19***)

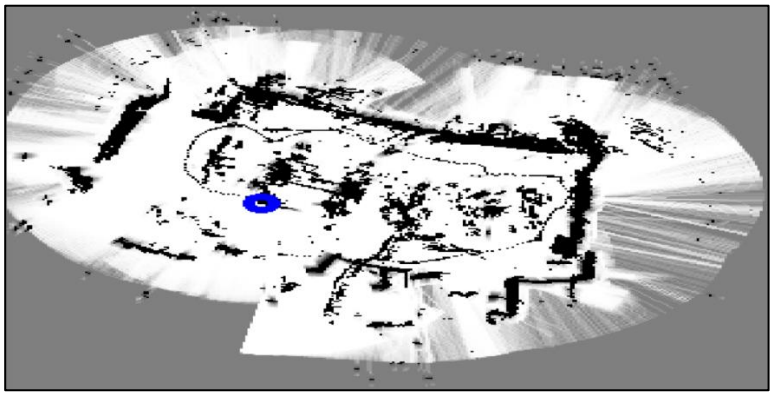


Figure 7.19 HLS C Simulation

The C simulation of the HLS code is able to accurately map the test data to the new scale of the map. Following this test, the files are run through the Vivado HLS C synthesis function. This is the function which converts the C/C++ code into HDLs for FPGA deployment. At first, this system was synthesised using the default clock period of 10ns or 100MHz clock frequency. This is the default frequency set by Vivado HLS for all projects. The system was able to synthesis correctly under these conditions; however, the system was noted as being too slow in relation to the system clock timing, with the system requiring 9.79ns out of the 10ns period. This was value was too high for the standard uncertainty measures of +/- 20% of the target period, in this case between 11.25 ns and 8.75ns. It is important that the timing value is below the minimum value as the Synthesis report only displays the ideal version of the system when the system is exported as RTL the system generally suffers some slowdown which in this case would result in errors. Following further testing, it was found that reducing the clock speed of the system to 25ns or 40MHz provided the most optimal clock conditions as higher

frequencies (50~100MHz) encounter timing errors, while lower frequencies were deemed to slow for normal use. Additionally, reducing the clock speed also reduces the utilization requirements of the system. This is important as the system requires enough resources for the required AXI interconnections. The performance estimates of both the 100Mhz and 40Mhz values can be seen below (***Figure 7.20*** & ***Figure 7.21***).

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.792	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	10	-	-
Expression	-	58	360	24189
FIFO	-	-	-	-
Instance	136	98	14544	19987
Memory	17	-	260	305
Multiplexer	-	-	-	4583
Register	-	-	13209	-
Total	153	166	28373	49064
Available	280	220	106400	53200
Utilization (%)	54	75	26	92

Figure 7.20 100MHz Timing Estimates

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	25.00	21.614	3.13

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	10	-	-
Expression	-	103	360	24464
FIFO	-	-	-	-
Instance	136	50	10472	19233
Memory	17	-	260	305
Multiplexer	-	-	-	4610
Register	-	-	9647	-
Total	153	163	20739	48612
Available	280	220	106400	53200
Utilization (%)	54	74	19	91

Figure 7.21 40MHz Timing Estimates

Finally, once the system has been correctly built the project was then exported into the Vivado Design Environment for deployment onto an FPGA. This was done using RTL Export Function. This function uses the HDL files to create an IP Block which can be imported into the Vivado Design Suite. The Vivado Design Suite [16] is a software application designed to allow users to write HDL files and convert them into Bitstream files for deployment. As this project focuses on the use of HLS, the system imports the IP Block generated by Vivado HLS rather than create unique ones. The IP Block was imported using the method described in Section 5.3.3. As stated previously, the clock period of this IP Block was changed to better suit the application requirements. In order to handle this, the clock frequency of the ZYNQ Processing system must be set to match the IP Block. This can be done by

using the customize block function to change the ZYNQ IP block. It is important to note that if the user uses the auto-configuration buttons at the top of the Block Design, this particular change must be made after the user runs the ***Run Block Automation*** command, as the Block Automation defaults the clock to 100Mhz, but before they run the ***Run Connection Automation*** command as the Processing Reset System created by Connection Automation is dependent on the clock speed of the ZYNQ at the time of creation. This project uses a combination of both AXI-Lite and AXI-Stream interfaces. Thus, both were required for system operation. The final block design of the system can be seen below (***Figure 7.22***), please note that due to the size of the image the text is legible if zoomed in.

Following the Block Design completion, the system could be converted into a bitstream for deployment. To do this an HDL wrapper must be created for the design. This wrapper creates an HDL version of the complete design, which denotes all the connections of your design. Once the wrapper had been created the bitstream was generated by pressing the Generate Bitstream button, which performs Synthesis, Implementation and Bitstream Generation. The project was successfully converted into a bitstream for system deployment on the PYNQ-Z2 FPGA.

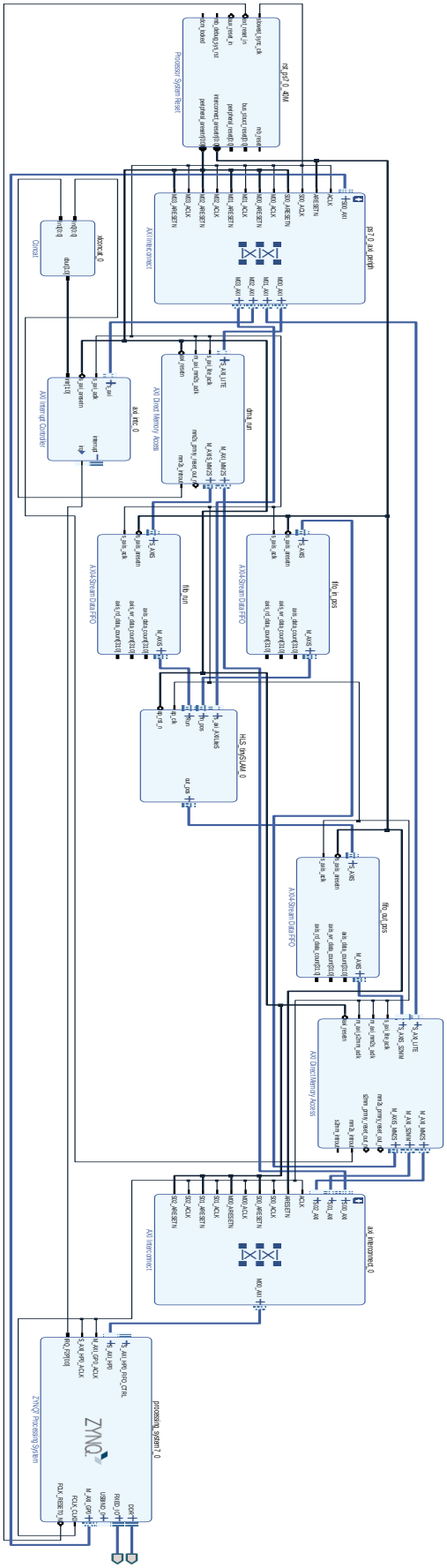


Figure 7.22 Finalised Block Design

Section 7.2.3 HLS TinySLAM Deployment and Testing

Once Bitstream had been generated the system was deployed on the FPGA board. For this project, this process is done using the Jupyter Notebooks Browser [23]. The Bitstream was loaded onto the board using the methods described in Section 5.3.4. Once this was done the tinySLAM algorithm could be used on the FPGA. As this version is dependent on the test data set, a copy of the test data had to be loaded onto the board as well.

In order to run the system, a new Python 3 notebook was created. This notebook uses the Python code created for the Python Version of tinySLAM described in Section 6.2.3 as a basis for the system. This notebook is divided into three parts. The first part initialises the Extended Functions class and the Sensor Data type. This class stores all the elements of the tinySLAM program not handled by the IP Block, i.e. reading data from the data file and converting the map array into an image. These functions remain unchanged from the original Python versions.

Following this, the next segment of the notebook deals with the communication and usage of the tinySLAM methods inside the IP Block. First, the system imports the PYNQ Overlay library and set the overlay to the bit file location. Then the IP and the DMA used by the IP are selected for AXI transfers. This segment of the notebook consists of four different functions. The first two relate to the converting floating-point numbers back and forth between Bit Data. As stated in Section 7.2.1.1, the scanning function uses floating-point data; however, the AXI-Lite system can only transfer integers or bits and cannot handle floating points. In order to circumvent this issue, the system converts the floating-point numbers into a 64-bit value which is then sent across the AXI-Lite to the IP Block. This value is then decoded back to a floating-point number using the previously discussed Union method. In order to send the value as a 64-bit value, the system uses the Python struct module, different from the C struct, which deals with data at the binary level. The **floatToBits** function takes in a floating-point number and "packs" the value into the struct module. The function then "unpacks" the value as a 64-bit long, i.e. reads the bits into the long value without the decimal e.g. 14.59 would become 1459. The **BitsTofloat** method performs this process in reverse reading in a 64-bit long and outputting a floating-point value. The code for both processes can be seen below (***Figure 7.22***).

```python
def floatToBits(self, f):
    s = struct.pack('>f', f)
    return struct.unpack('>l', s)[0]

def BitsTofloat(self, i):
    s = struct.pack('>l', i)
    return struct.unpack('>f', s)[0]
```

Figure 7.23 Python Bit Conversions

Following this, the system describes with the ***HLS_Map_Init*** function which handles communication with the IP Blocks Map Initialisation method. As this method relates to initialising the map no values scans, position values or previous maps need to be sent for system operation. Instead, the system simply sends a Run value of equal to 1 across the DMA. As described in the HDL Wrapper, Section 7.2.1.1, setting the Run value equal to 1 activates the Map Init Function. The system is then told to sleep for 0.1 seconds. This is done in order to give the system time to complete the calculation and acts as a buffer for any potential timing errors found. Finally, the system can then read the Map values from the AXI-Lite by looping through each value in memory. The memory location of the map array can be found in the VHDL code generated by the HLS process. When examining the VHDL code it was found that due to the size of the Map array, each memory element of the array stores two 16 values for the array, i.e. the system converted the 32-bit integers into 16-bit values and combined them to save space. Therefore in order to read the array correctly, the system must split each value from the AXI into two 16 bit values and use those values in the map. The VHDL describing the memory format and the Python code for the ***HLS_Map_Init*** function can be seen below (***Figure 7.24*** & ***Figure 7.25***).

```
-- ------------------------Address Info-------------------
-- 0x04000 ~
-- 0x07fff : Memory 'scan' (2732 * 32b)
--           Word n : bit [31:0] - scan[n]
-- 0x40000 ~
-- 0x7ffff : Memory 'map_r' (129600 * 16b)
--           Word n : bit [15: 0] - map_r[2n]
--                    bit [31:16] - map_r[2n+1]
-- (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

architecture behave of HLS_tinySLAM_AXILiteS_s_axi is
    type states is (wridle, wrdata, wrresp, wrreset, rdidle, rddata, rdreset);  -- read and write fsm states
    signal wstate  : states := wrreset;
    signal rstate  : states := rdreset;
    signal wnext, rnext: states;
    constant ADDR_SCAN_BASE  : INTEGER := 16#04000#;
    constant ADDR_SCAN_HIGH  : INTEGER := 16#07fff#;
    constant ADDR_MAP_R_BASE : INTEGER := 16#40000#;
    constant ADDR_MAP_R_HIGH : INTEGER := 16#7ffff#;
    constant ADDR_BITS         : INTEGER := 19;
```

Figure 7.24 VHDL TinySLAM

```
def HLS_map_init(self):
    #Initialize Variables
    Run = xlnk.cma_array(shape=(1,), dtype=np.uint32)
    map1 = []

    #print("Activate Map Init...")
    Run[0] = 1
    dma_run.sendchannel.transfer(Run)
    dma_run.sendchannel.wait()

    #Delay to Allow time for calculation to complete
    time.sleep(0.1)

    #print("Reading Map...")
    for v in range(0, 64800):

        #Read in the 32 bit value
        location = map_start + 4*v
        val = tiny_ip.read(location)

        #Split into two 16 bit values
        upper = (val >> 16) & 0xFFFF
        lower = val & 0xFFFF

        #Append
        map1.append(lower)
        map1.append(upper)

    #print(map1)
    return map1
```

Figure 7.25 HLS_Map_Init Jupyter Notebooks

Following the development of the ***HLS_Map_Init*** function, development of the ***HLS_Main*** function began. This function would run the main tinySLAM code on the IP Block, which consisted of both the Monte Carlo position update system and the Map Update system. In order to run this function, the system first had to transfer all relevant data to the IP Blocks via the AXI interface. The scans and the map were written directly into the IP Block memory addresses using AXI Lite. As discussed before, the scan data which was in terms of floating-point numbers needed to be converted into Bits or Unsigned long values using the **floatToBits** method before it could be transferred. The map array was found to be too large and HLS automatically converted the array such that each memory location held a single 32-bit value, which consists of a combination of two 16-bit values. Therefore in order to send the data to the board every two values of the map were converted into two 16 bit values and combined together before AXI transfer.

Once the map and scan data had been sent over the AXI-Lite, the position value was sent using AXI-Stream. Finally, the Run value, this time equal to 2, was sent to activate the system. The Run value is transferred last as is operates as both a system decoder, i.e. what function is run in the IP, and is used as a chip enable system, forcing the program to wait for AXI-Stream data to be sent before it can run. Once again the system is then told to sleep for 0.1 seconds as a form of buffer. Once all the input data has been sent the system then reads the outputs. First, the system reads the new Position value, as calculated by the Monte Carlo method, from the AXI-Stream interface. Following this, the system reads

in the updated map using the same method of AXI-Lite reading described for the ***HLS_Map_Init*** function. An image of the ***HLS_Main*** function can be seen below (***Figure 7.26***).

```
def HLS_Main(self, scan, mp, pos):
    #Initialize Variables
    Run = xlnk.cma_array(shape=(1,), dtype=np.uint32)
    pos2 = xlnk.cma_array(shape=(3,), dtype=np.float32)
    map1 = []

    x = 0
    #print("Writing Scan...")
    for i in range(0, TS_SCAN_SIZE):
        for j in range(0, 4):
            location = scan_start + 4*x
            val = scan[i][j]
            var = self.floatToBits(val)
            tiny_ip.write(location,var)
            x += 1

    #print("Writing Map...")
    for v in range(0, 64800):
        #Combine two 16 bit values into a single value
        location = map_start + 4*v
        #val = (mp[2*v] << 16) | mp[(2*v)+1]
        val = (mp[(2*v)+1] << 16) | mp[2*v]

        #Write the 32 bit value
        tiny_ip.write(location,val)

    #print("Streaming Position")
    dma_pos.sendchannel.transfer(pos)
    dma_pos.sendchannel.wait()

    #print("Activate Main...")
    Run[0] = 2
    dma_run.sendchannel.transfer(Run)
    dma_run.sendchannel.wait()

   #Delay to Allow time for calculation to complete
   time.sleep(0.1)

   #print("Recieve New Position")
   dma_pos.recvchannel.transfer(pos2)
   dma_pos.recvchannel.wait()

   #print("Reading Map...")
   for v in range(0, 64800):

        #Read in the 32 bit value
        location = map_start + 4*v
        val = tiny_ip.read(location)

        #Split into two 16 bit values
        upper = (val >> 16) & 0xFFFF
        lower = val & 0xFFFF

        #Append
        map1.append(lower)
        map1.append(upper)

   return map1, pos2
```

Figure 7.26 HLS_Main Jupyter Notebooks

Finally, the last segment of the notebook covers the main method of the Python script. This script runs the code and calls the other functions. This code remains mostly unchanged from the main method of the Python Version of HLS from Section 6.2.3. The only changes required were replacing the references to the old tinySLAM methods with the HLS tinySLAM methods. The only other change relates to the display of the image. The previous Python version of the system used the OpenCV library in order to display the image. While OpenCV library operates correctly in Jupyter Notebooks the library is unable to display the image using the *imshow* method. Therefore, the matplotlib library is used to replace the *imshow* method. This library is designed to display data plots of X vs Y and is compatible with Jupyter Notebooks. This library is used to display the image. The change in code and the output image can be seen below (***Figure 7.27***, ***Figure 7.28*** & ***Figure 7.29***).

```
image = cv2.imread(filename)
plt.imshow(image)
plt.show()
```

Figure 7.27 Matplotlib Version

```
img = cv2.imread(filename)
cv2.namedWindow("image", cv2.WINDOW_NORMAL)
cv2.imshow("image", img)
cv2.waitKey()
```

Figure 7.28 OpenCV Version

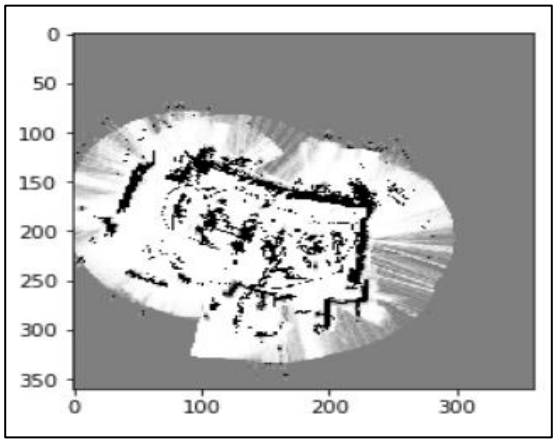


Figure 7.29 Final Map

As seen in the image above, the HLS implementation of the tinySLAM algorithm succeeds in generating the map of the test data set. The map shown displays the same map generated by the C simulation.

Section 7.3 HLS TinySLAM with Live Data

Following the completion of the tinySLAM using the test data, development of a system using Live LiDAR data began. As discussed previously in Section 6.2.4, this system uses the RPLiDAR as the LiDAR scanner. The primary tinySLAM code created using HLS required no changes, as both the test data and the live data system use the same functions in HLS. The only modifications required for system operation was the inclusion of the RPLiDAR into the Python code on Juypter Notebooks.

In order to use the RPLiDAR, the RPLiDAR Software Development Kit (SDK) was used [32]. The RPLiDAR SDK contains the code and libraries relevant to RPLiDAR execution. This SDK was used in the development of the C and Python versions of the tinySLAM system as seen in Section 6.2.4. This SDK is designed to operate on a variety of different operating systems including Windows and Linux OS. The PYNQ-Z2 board is designed to uses an ARM-Linux operating system; however, the computer used for primary system development used a Windows OS. Therefore, the SDK could not be

directly deployed from the Windows system. In order to circumvent this issue, it was necessary to examine the SDK in a Linux OS.

Section 7.3.1 RPLiDAR SDK using Virtual Machine

Initial examination of the SDK was performed using a virtual machine. A Virtual Machine (VM) in computing is an emulation of a computer system. Virtual machines are based on computer architectures and provide the functionality of a physical computer. For this project, the Oracle VM VirtualBox [37] was used. Virtual Box is a free and open-source hosted hypervisor for x86 virtualization, which is used to run virtual machines with different operating systems. For this project, a 32-bit Ubuntu Linux OS was used. This OS allowed for testing the Linux version of the RPLiDAR library.

In order to use the sensor in the VM Virtual Box must be set up to link the LiDAR USB to the VM. This can be done by using the Oracle VM Extension Pack, which add functionality for external ports such as USB 2.0 and 3.0. To install the Extension Pack, the user needs to download the package, then add the settings as follow:

1. File -> Preferences -> Extensions, Add Extension file
2. Settings of Desired VM -> USB -> Add Desired USB device.

Once this has been done the device is connected to the VM for development. The LiDAR must be connected during the VM boot-up to ensure it connects correctly. Once inside the VM, the user must use the ***sudo chmod 666 /dev/ttyUSB0*** and ***sudo sysctl kernel.dmesg_restrict=0*** in the command terminal to allow for access to the RPLiDAR USB port.

The library was tested on the VM using the *ultra_simple* method included in the SDK. To do this the SDK was downloaded onto the VM. Then the library was deployed in Linux following the method described in the RPLiDAR SDK setup guide [38]. This setup uses the Linux **make** command along with Makefiles which when run automatically construct the library as required. Finally, a Python script was created to run the *ultra_simple* method. The output and the Python script can be seen below (***Figure 7.30*** & ***Figure 7.31***).

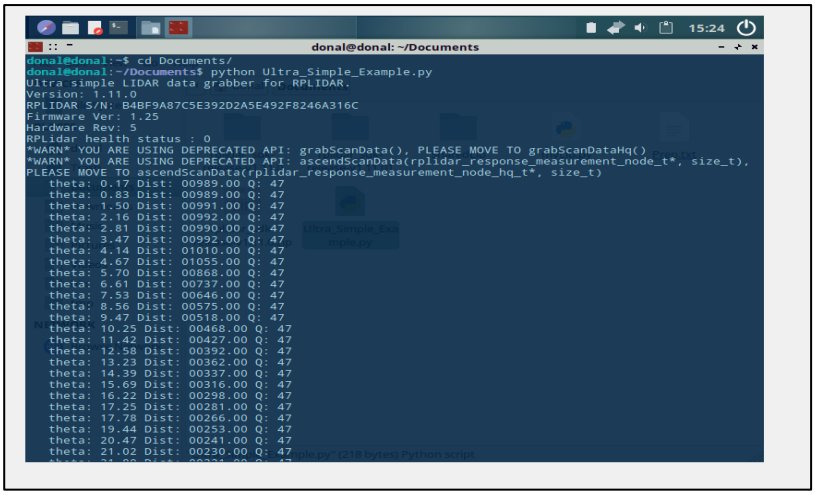


Figure 7.30 Ultra Simple on VM

```
from ctypes import *

my_so = "/home/donal/Documents/rplidar_sdk-release-v1.11.0/sdk/output/Linux/Release/ultra_s
lib = cdll.LoadLibrary(my_so)

mp = (c_float * 682)()
port = c_char_p("/dev/ttyUSB0")

lib.main()
```

Figure 7.31 Ultra Simple Python Script

Section 7.3.2 RPLiDAR SDK on the PYNQ-Z2

Following the examination of the SDK on the Linux VM, the *ultra_simple* method was modified as described in Section 6.4.2 for FPGA deployment. However, it was soon discovered that the ARM Linux operating system on the PYNQ-Z2 did not function the same as the Linux VM, and thus the RPLiDAR library could not be instantiated using the methods described by the RPLiDAR SDK setup guide [38]. This appears to be caused by the differences between ARM Linux and more standard Linux formats resulting in minor differences which prevent the automated Makefile procedure from creating the Library from correctly, i.e. while the Makefile system did create the library correctly, attempting to use these libraries gave an OS error. Therefore, the library needed to be manually modified before implementation.

Firstly, the Makefile system needed to be run. While the final library generated by the Makefile is incompatible, many of the files generated by the automated system are still required for deployment. Once this was done a new folder called SharedLibrary was created. This folder contains all relevant

information of the system in one location to remove the need for cross-referencing location used by the Makefile system. The following elements were copied:

1. All files located in *rplidar_sdk\sdk\obj\Linux\Release\sdk\src* with files within subfolders copied as standalone files (subfolders are removed).
2. All files located in *rplidar_sdk\sdk\sdk\src* where subfolders are copied as they are (required to be in subfolders for referencing purposes).
3. All files located in *rplidar_sdk\sdk\sdk\include*.
4. Finally, the static library ***librplidar_sdk.a*** found in *rplidar_sdk\sdk\output\Linux\Release* must be copied.

Once this was done a wrapper C file was created within the SharedLibrary. This wrapper file was a modified version of the *ultra_simple* C file, containing the C code for the 3 primary methods used in this system as described in Section 6.4.2. Once this was done the conversion of the files to a usable library format could begin. Firstly, the C file is converted into an object file. An object file in Linux is a file containing object code or formatted machine code that stores data about a particular function. This form of file is not directly executable. In order to convert the file from C to an object file the command terminal of the FPGA was used. The command terminal or command line of the FPGA can be accessed from Jupyter Notebooks. To convert the file the command gcc -Wall -fPIC -c myLib.cpp was used in the command line converting the file **myLib.cpp** into an object file called **myLib.o**.

Once the wrapper file has been converted into an object file, the final library file can be created. This is done using the g++ -shared -o libLiDAR.so myLib.o net_serial.o net_socket.o thread.o timer.o rplidar_driver.o -Wl,--no-whole-archive librplidar_sdk.a command. This command creates the shared library file **libLiDAR.so**, by combining all the library files of the original system, **net_serial.o, net_socket.o thread.o, timer.o** & **rplidar_driver.o**, with the custom wrapper file **myLib.o**. Additionally, RPLiDAR specific values or references set in the system are read from the static library of the SDK **librplidar_sdk.a**. It is important to note that the static library must be set as no-whole-archive. By default, libraries are set as the whole-archive option which includes every object file in the archive in the link, rather than searching the archive for the required object files. However, the static library contains references to elements already included in the above object files. Therefore, no-whole-

archive must be set to prevent system errors, i.e. the system will fail to build the library if two methods with the same name are found, e.g. main(). Once the shared library has been completed the RPLiDAR library can be imported into Jupyter Notebooks Python Scripts. Images of the Jupyter Notebooks Terminal, Library Files, and a Python example of the RPLiDAR can be seen below (***Figure 7.32***, ***Figure 7.33*** & ***Figure 7.34***).

```
from ctypes import *

my_so = '/home/xilinx/LinLib/SharedLibrary/libLiDAR.so'
my_func = cdll.LoadLibrary(my_so)
mp = (c_float * 682)()
my_string = "/dev/ttyUSB0"

port = c_char_p(my_string.encode('utf-8'))
select = c_int(0)

if(my_func.main(mp, port, select) == 1):
    select = c_int(1)
    my_func.main(mp, port, select)

    select = c_int(2)
    my_func.main(mp, port, select)

for i in range(0, 682):
    print(i, mp[i])
```

```
0 875.0
1 874.0
2 878.0
3 0.0
4 0.0
5 0.0
6 865.0
7 861.0
8 858.0
9 855.0
10 854.0
11 861.0
12 0.0
```

Figure 7.32 RPLiDAR Jupyter Notebooks Python

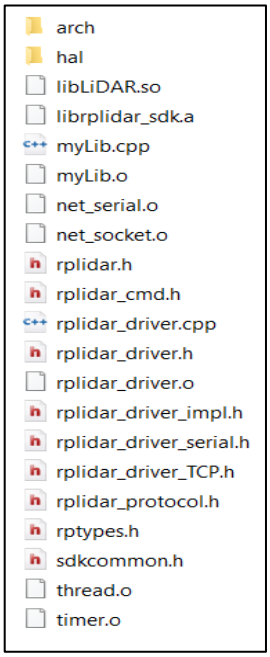


Figure 7.33 RPLiDAR Library Files

```
root@DonalPynq:/home/xilinx/LinLib/SharedLibrary# gcc -Wall -fPIC -c myLib.cpp
myLib.cpp: In function 'int BeginSensor(const char*)':
myLib.cpp:85:15: warning: unused variable 'opt_com_baudrate' [-Wunused-variable]
  _u32         opt_com_baudrate = 0;
               ^~~~~~~~~~~~~~~~
myLib.cpp: At global scope:
myLib.cpp:191:5: warning: first argument of 'int main(float*, const char*, int)' should be 'int' [-Wmain]
 int main(float map[TEST_SCAN_SIZE], const char * mypath, int select)
     ^~~~
myLib.cpp:191:5: warning: second argument of 'int main(float*, const char*, int)' should be 'char **' [-Wmain]
myLib.cpp:191:5: warning: third argument of 'int main(float*, const char*, int)' should probably be 'char **' [-Wmain]
root@DonalPynq:/home/xilinx/LinLib/SharedLibrary# g++ -shared -o libLiDAR.so myLib.o net_serial.o net_socket.o thread.o timer.o rplidar_driver.o
 -Wl,--no-whole-archive librplidar_sdk.a
root@DonalPynq:/home/xilinx/LinLib/SharedLibrary#
```

Figure 7.34 Jupyter Notebooks Terminal

Finally, before the RPLiDAR can be used the FPGA must be granted access to the LiDAR in the USB port. This is done by using the command ***sudo chmod 666 /dev/ttyUSB0*** and the ***sudo sysctl kernel.dmesg_restrict=0*** in the command terminal. Unfortunately, these commands must be run every time the board is powered on as the FPGA resets when turned-off. Therefore, to ease this issue these commands were added to the list of commands to be automatically executed during boot-up. To do this

both commands were added to the Linux crontab file. The crontab file in Linux is a daemon that performs user-edited tasks at specific times and events. The crontab is accessed using the sudo crontab -e. Both of these commands were set as boot-up commands as shown below (***Figure 7.35***)

```
@reboot sudo sysctl kernel.dmesg_restrict=0
@reboot sudo chmod 666 /dev/ttyUSB0
```

Figure 7.35 Crontab

Section 7.3.3 HLS TinySLAM using RPLiDAR

Once the RPLiDAR had been successfully set up the library could be imported into the TinySLAM Python Notebook. As stated previously, the tinySLAM algorithm passed through the HLS process required no modifications, therefore the HLS segment of the Python Script also remained unchanged. The primary change related to the reading scans segment of the Python Script. In the original system, the read scans method read data from a single test data file. This read scans method was run once at the beginning of the program. In the case of live data however, the system reads each scan individually from the sensor and performs a read on every loop. The modifications made to the Python script follow a similar format to the Python version of the the RPLiDAR tinySLAM described in Section 6.2.4.2. The program was then run and returned a similar image as seen below (***Figure 7.36***).

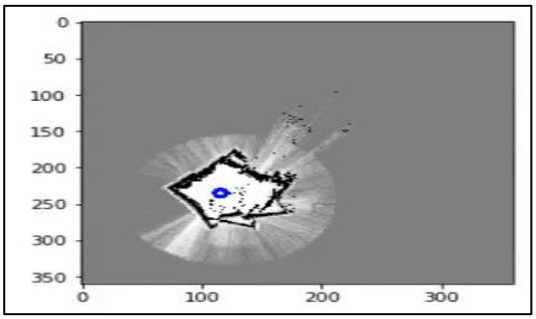


Figure 7.36 TinySLAM HLS Live Sensor Map

As seen in the image above, the HLS implementation of the tinySLAM algorithm succeeds in generating the map of the Live Sensor Data. The map shown displays a similar image to that of the map created using Live Data from the Visual Studio C implementation. An image of the PYNQ RPLiDAR Setup can be seen below (***Figure 7.37***)

Figure 7.37 PYNQ to RPLiDAR Connection

Section 8. System Results & HLS Comparisons

Section 8.1 HLS TinySLAM using Test Data

The purpose of this project was to demonstrate the capabilities of High-Level Synthesis to encode complex systems onto FPGAs. The primary goal was to convert a complex system onto an FPGA and compare said system to a version of the system created through more standard coding procedures. For this project, a terrain mapping system was chosen based on the tinySLAM algorithm. The project involved the development of two separate applications. The first was the development of the tinySLAM algorithm on HLS using a set of test data found in the OpenSLAM [4] repository for the tinySLAM algorithm.

As described in Section 7, the HLS tinySLAM was successfully created. The system created a similar map to the C version of the tinySLAM algorithm. The map had some changes, which were caused due to the different scale of the HLS map versus the C tinySLAM map. When the C version was rescaled to match the HLS version the maps appeared identical as seen below (***Figure 8.1*** & ***Figure 8.2***).

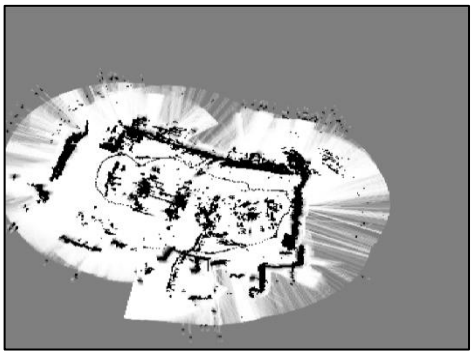


Figure 8.1 Scale C Map

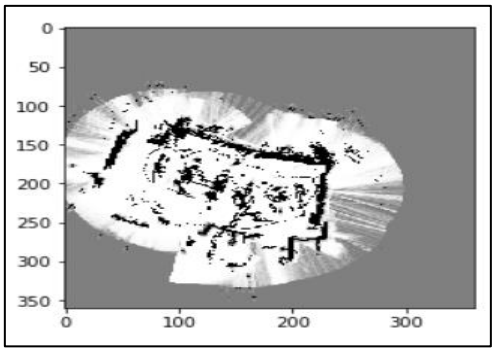


Figure 8.2 HLS Map Output

The HLS tinySLAM was found to be much slower than the C version of tinySLAM, with the C version creating the map within 30 seconds while the tinySLAM calculation requires approximately a full hour. However, the C/C++ implementation was run on the Acer Nitro 5 Laptop, while the HLS version was run on a PYNQ-Z2 Board which is a much smaller and less powerful machine, which accounts for the majority of the slowdown. In order to better compare HLS methodology to more standard coding, a Python version of the tinySLAM algorithm was developed, as seen in Section 6.2.3.

This Python script could then be run on the same PYNQ Board as the HLS implementation using the Jupyter Notebooks Browser allowing for a more direct comparison. The Python tinySLAM was updated to use the same scaling configurations as the HLS tinySLAM resulting in the following map (***Figure 8.3***).

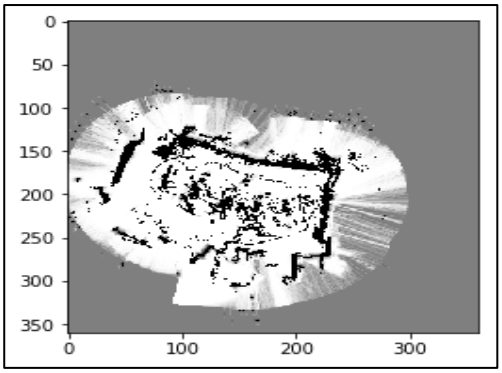


Figure 8.3 Python Map

As can be seen above the Python Version on the FPGA outputs the same map as both the C and the HLS programs. Comparing the Python Version to the HLS tinySLAM we can see that the use of HLS has dramatically increased the efficiency of the system. The times for both systems, hours, minutes and seconds, can be seen in the images below (***Figure 8.4*** & ***Figure 8.5***).

```
Finished!!
Time: 2:59:09.756796
```

Figure 8.4 Python Time

```
Finished!!
Time: 0:55:21.783728
```

Figure 8.5 HLS Time

The pure Python version is approximately three times slower than the HLS implementation. Both of these systems ran on the same FPGA, using the same test data set. Additionally, any methods that had not been passed through the HLS process, i.e. the map display methods, used the same Python code in both versions of the system. This demonstrates the advantage of placing a complex system such as tinySLAM through the HLS process. Additionally, while FPGAs are commonly programmed using Hardware Description Languages (HDLs), the use of HLS allowed for the use of the increased energy efficiency and speed associated with hardware implementations, without the time delays associated with converting an algorithm to an HDL by hand. However, that while the HLS process completed the map much faster than the pure Python method, the Python method began faster, slowing down as the system looped through each iteration, while the HLS method maintained a constant calculation speed

throughout. Therefore, in the case of a much smaller data set, less than 50 values as opposed to the current 590 values, it is possible that the pure Python method may result in a faster system. However, this increase was found to be both minimal and inconsistent while the HLS method gives a much more consistent speed increase for large datasets.

Section 8.2 HLS TinySLAM using RPLiDAR

Once the tinySLAM algorithm had been completed using the test data set the project moved toward the development of the system using a live sensor. For this project, a Slamtec RPLiDAR A1M8 [21] was used. This LiDAR was chosen due to its low cost and evidence demonstrating the LiDARs capabilities in previous tinySLAM systems as seen in *A low-cost indoor mapping robot based on TinySLAM algorithm* [22]. This LiDAR was to be combined with the PYNQ-Z2 Board to result in the final system. A system architecture diagram of the final system can be seen below (***Figure 8.6***).

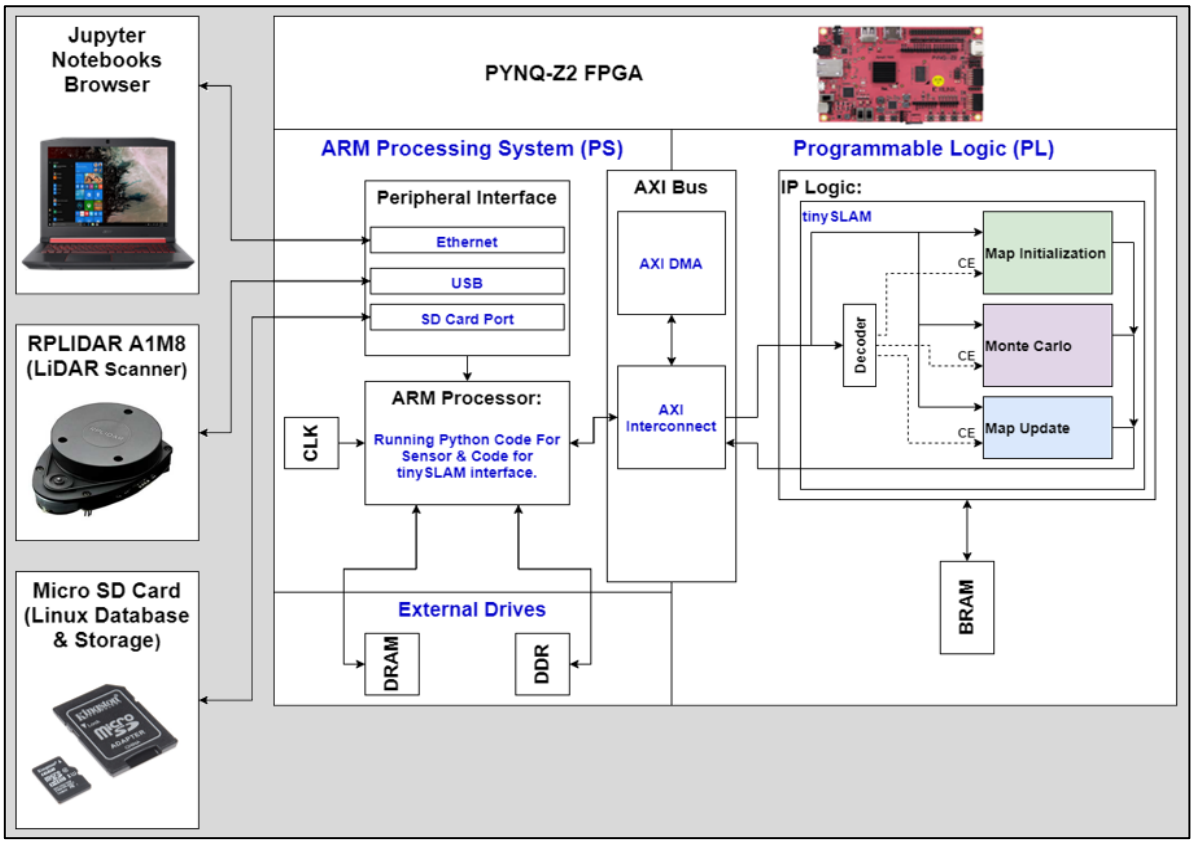


Figure 8.6 System Architecture

As described in Section 7, development of the HLS tinySLAM using the LiDAR was successful. The HLS components of the system remain the same as those seen in the Test Dataset with the changes relating to the Python Script and the integration of the RPLiDAR. The system successfully mapped the environment. Using the same room, the system created a similar map to the C version of the tinySLAM algorithm. The algorithm output a similar map for both systems as seen below (***Figure 8.7*** & ***Figure 8.8***).

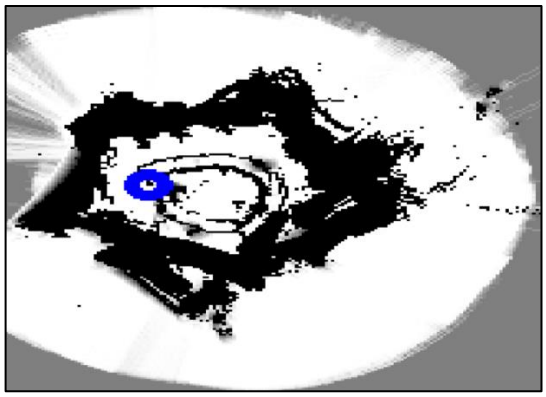


Figure 8.7 tinySLAM RPLiDAR C

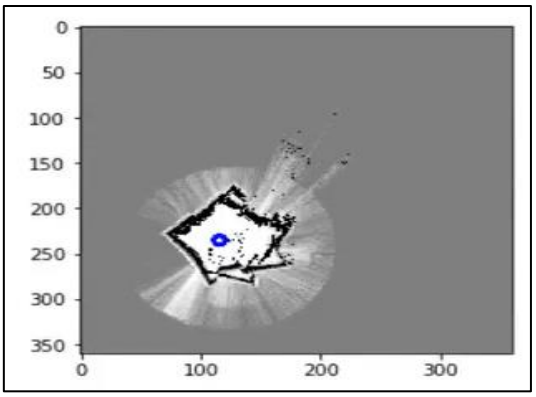


Figure 8.8 tinySLAM RPLiDAR HLS

As this version of the system is dependent on live data the map quality is dependent on the number of scans used to create the map, i.e. the length of time the system scans the environment determines the map quality. As previously seen in Section 8.1, the C/C++ implementation is much faster than the HLS version. In the images seen above both the C and HLS versions were run for approximately the same amount of time, between 5 to 10 minutes, resulting in similar maps. However, as can be seen above the C version has a much higher quality due to its higher scanning speed. An additional element of note is the conflict with the Monte Carlo method for updating the system position. As discussed in Section 6, the Monte Carlo method relies on comparisons to previous data in order to update the system position. The LiDAR sensor must move at slow enough speed to allow the system to scan previous objects for comparison. If the sensor were to move too fast the system may lose track of its position and therefore result in an incorrect map. For the C/C++ implementation, this simply requires the system to move at approximately the average walking speed of a person. However, due to the slower calculation time of the HLS method, the HLS version of the system must be moved at a much slower speed in order to ensure the system operates correctly.

Once again, the majority of this slowdown can be attributed to the C code using a much more powerful machine. In order to better compare HLS methodology to more standard coding, a Python version of the tinySLAM algorithm was developed, as seen in Section 6.2.4.2. This Python script could then be run on the same PYNQ Board as the HLS implementation using the Jupyter Notebooks Browser allowing for a more direct comparison. However, due to the PYNQ-Z2 using a Linux system rather than a Windows system the Dynamic Link Library (DLL) used in Section 6.2.4.2 could not be used. Instead, this code was replaced with the PYNQ-Z2 compatible library developed for the HLS implementation as discussed in Section 7.3.2. Using this system, the pure Python version was able to successfully create a map of the environment resulting in a similar map to the HLS version seen in ***Figure 8.8***. As with the HLS version the map quality is dependent on the number of scans used. The time taken for these systems to complete the same number of scans (40 scans) in hours, minutes and seconds can be seen below:

- Pure Python: 0:11:33.64
- HLS Implementation: 0:04:28.75

As can be seen above the HLS implementation is significantly faster, with the HLS version reducing the Python time by 60%. The slowdown of the Python version also has a knock-on effect on the map position system. As previously stated, due to the Monte Carlo method used for calculating the systems current position the system is required to move slow enough that the system can maintain some of the previous objects between scans. However, as the system calculation speed slows down the movement of the LiDAR must slow down to compensate or the system may lose track of its location and result in incorrect values on the map. Therefore, due to the Python Version being slower than the HLS version, the LiDAR must be moved slower while using the Python Version to maintain map integrity.

Section 9. Possibility for Extension

Following project completion, several additional changes were noted which could be implemented in order to improve overall system design for deliverable user product.

Firstly, in a real-world scenario, this project should be integrated into a fully autonomous system by attaching the system to an autonomous robot. Currently, the user must manually carry the LiDAR during use which is cumbersome and inefficient. The project should be made into an autonomous system, using some form of robotic system to move the LiDAR throughout the environment and thus scan the array. The use of a robot will also allow for precise control of the movement speed, which is useful due to the effects of movement speed on the Monte Carlo method detailed in Section 8.2.

Another aspect to consider is the use of an alternative location management system. This project focus on the calculation of the systems current position through the use of the LiDAR and the Monte Carlo method. While this method is effective it has notable flaws in the limitation of the systems movement speed. Therefore, the accuracy and movement speed of the system could be increased by adding a secondary method of determining location. In the tinySLAM methods detailed in *tinySLAM: a SLAM Algorithm in less than 200 lines C-Language Program* [12] and in *A low-cost indoor mapping robot based on TinySLAM algorithm* [22] odometers are used as a secondary position calculation. An odometer is an instrument used for measuring the distance travelled by an object, typically a vehicle such as a bicycle or car. The tinySLAM code found in the OpenSLAM [29] repository included calculations for position based on odometers; however, due to time constraints, this process was not included in this project.

The project could also be further developed with the creation of an independent external client. In a real-world scenario, this project should be connected to an independent external client, as opposed to the current model which uses a Jupyter Notebooks browser. While the Jupyter Notebooks client is acceptable for development purposes if this project were to be released as a product the system would need to use its own external client. Additionally, the current model communicates with the board via Ethernet limiting the system's distance from the Laptop/Desktop. The system could be connected with

the client via Wi-Fi through the uses of a Wi-Fi adapter connected to the Ethernet port of the PYNQ-Z2 board. This would increase the overall capabilities of the system without any major reduction in system quality, as the majority of system functions are handled within the PYNQ board, where the client merely acts as a user terminal for system activation, termination and to display the map.

Finally, this project could be more optimized to run on the PYNQ board. The system could optimize the allocation of tasks between the Processing System (PS), Python code, and the Programmable Logic (PL), HLS IP, portions of the board to better optimize system speed. The goal of this project was to demonstrate the capabilities of HLS and therefore the system is designed to perform as much of the calculations on the PL side as possible. This can cause bottlenecking as the system must pass all the data back and forth across the AXI bus. Additionally, this causes some slow-down on the HLS side in order for the system to handle the large resource usage. The system could be better optimized with a more balanced distribution of tasks between the PS and PL components of the board, e.g. the *Map_Init* function of the tinySLAM algorithm involves the simple creation of an array and thus would be better suited to a Python implementation, while the *Monte Carlo* and *Map Update* components involve large looping calculations better suited to a hardware or HLS implementation.

Section 10. Conclusions

The purpose of this project was to demonstrate the advantages of High-Level Synthesis. The aim was to show the capabilities of HLS to encode complex system onto FPGAs for system deployment. The project was designed to perform a form of terrain mapping algorithm to create a map of the surrounding area. Originally, this project was to be made into an autonomous system however, due to time constraints and other unforeseen circumstances the project removed the autonomous motion to focus on the primary terrain mapping algorithm itself.

The original goal of the project was to demonstrate HLS as a viable platform for the development of complex systems. To do so a Python version of the system was also required to act as a basis for comparison between the hardware implementations of HLS and the more standard coding implementations performed on C or Python system. The original tinySLAM algorithm ran in C code; however, this was run on a Laptop or Desktop which is a much more powerful machine than an FPGA. Hence, it was determined that a comparison between the HLS on the FPGA and the C code on the Laptop would not be a fair comparison. Thus, the Python version of the system was developed for deployment on the FPGA. This was also beneficial as the elements of tinySLAM not put through the HLS process could use the same code as the pure Python version allowing for a more direct comparison.

As discussed in Section 8, the system involved two tests. First, there were the tinySLAM tests performed using a test data set. Secondly, there were the tinySLAM tests performed using live data from a LiDAR sensor. Both tests were performed correctly, producing the correct maps for each data set. For both systems, it was found that the HLS implementations were significantly more efficient than the pure Python versions. Both HLS versions were roughly 2.5~3 times faster than the pure Python implementations. Both versions used the same PYNQ-Z2 FPGA, the same LiDAR and contained the same Python code for any elements that had not been put through the HLS process. Due to these similarities the system clears shows that the HLS process provides significant benefits for encoding complex systems. Additionally, due to the previously noted Monte Carlo method, the increase in the system calculation speed of the HLS implementation also allows for the system to be moved at a higher speed, due to the Monte Carlo calculation limiting system mobility.

In conclusion, I found this project to be interesting to work on as it allowed me to apply many of the skills I have gained throughout my studies. The development of this project gave me great insight into the areas of FPGA architectures and the field of machine perception. The project allowed me to see the first-hand application ad effectiveness of FPGA hardware implementations and the problems involved with real-time calculations of data. I gained new insight into the application of machine perception technologies and the issues involved with the development of using remote sensor devices. The system also allowed for the development of my understanding of programming libraries and Linux operating systems.

References

[1] N. O. a. A. (NOAA), “Remote Sensing,” 15 04 2020. [Online]. Accessed 19 04 2020: https://oceanservice.noaa.gov/facts/remotesensing.html.

[2] P. K. B. S. B. Pulak Mondal, “FPGA based accelerated 3D affine transform for real-time image,” *Computers and Electrical Engineering,* vol. 49 (2016), pp. 69-83, 2015. Available: https://www.sciencedirect.com/science/article/pii/S0045790615001457.

[3] K. Shih, A. Balachandran, K. Nagarajan, B. Holland, C. Slatton and A. George, “Fast real-time lidar processing on FPGAs,” in *ERSA,* 2008. https://pdfs.semanticscholar.org/d9c5/8794bb1cbed8cd5cb4d8c50b84918ac2cb19.pdf.

[4] “Machine Perception,” wiseGeek, 30 March 2020. [Online]. Accessed 21 April 2020: https://www.wisegeek.com/what-is-machine-perception.htm.

[5] N. O. a. A. A. (NOAA), “LiDAR,” 15 04 2020. [Online]. Accessed 19 04 2020: https://oceanservice.noaa.gov/facts/lidar.html.

[6] A. Sinha, N. Tan and R. E. Mohan, “Terrain perception for a reconfigurable biomimetic robot using monocular vision,” 2014. https://jrobio.springeropen.com/articles/10.1186/s40638-014-0023-2.

[7] F. Masataka, R. E. Mohan, N. Tan, A. Nakamura and T. Pathmakumar, “Terrain Perception in a Shape Shifting Rolling-Crawling Robot,” *Robotics,* pp. 5-19, 2016. https://www.mdpi.com/2218-6581/5/4/19.

[8] C. Y, “Mean shift, mode seeking, and clustering.,” *IEEE Trans Pattern Anal Mach Intell,* vol. 17, no. 8, p. 790–799, 1995. https://pdfs.semanticscholar.org/253a/36f0e1e17d53cedc891e35eb6f091dd8f08a.pdf?_ga=2.261660941.1009659746.1589555305-861096365.1588854529.

[9] M. Montemerlo, S. Thrun, D. Koller and B. Wegbreit, “FastSLAM: A Factored Solution to the Simultaneous,” *Proc. AAAI Nat'l Conf. Artificial Intelligence,* 2002. http://robots.stanford.edu/papers/montemerlo.fastslam-tr.pdf.

[10] J. M. M. M. J. D. T. R. Mur-Artal, “ORB-SLAM: A versatile and accurate monocular SLAM system,” *IEEE Trans. Robot.,* vol. 31, no. 5, pp. 1147-1163, 2015. https://ieeexplore-ieee-org.libgate.library.nuigalway.ie/document/7219438.

[11] R. Giubilato, M. Pertile and S. Debei, “A comparison of monocular and stereo visual FastSLAM implementations,” *IEEE Metrology for Aerospace (MetroAeroSpace),,* pp. 227-232, 2016. https://ieeexplore.ieee.org/document/7573217.

[12] B. Steux and O. E. Hamzaoui, “tinySLAM : a SLAM Algorithm in less than 200 lines of C code ,” in *2010 11th International Conference on Control Automation Robotics & Vision*, Singapore, 2010. https://ieeexplore.ieee.org/document/5707402.

[13] “Vivado HLS,” [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration.html.

[14] “Monte Carlo Method,” Science Direct, [Online]. Accessed 21 04 2020: https://www.sciencedirect.com/topics/engineering/monte-carlo-method.

[15] F. Sanchez, R. Mateos, E. Bueno, J. Mingo and I. Sanz, “Comparative of HLS and HDL Implementations of a Grid Synchronization Algorithm.,” 2013. https://ieeexplore.ieee.org/document/6699478

[16] “Vivado Design Suite,” [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[17] R. Kastner, J. Matai and S. Neuendorffer, Parallel Programming for FPGAs, The HLS Book, 2018. http://kastner.ucsd.edu/hlsbook/.

[18] C. Li, Y. Bi, Y. Benezeth, D. Ginhac and F. Yang, High-level synthesis for FPGAs: code optimization strategies for real-time image processing, vol. 14, Journal of Real-Time Image Processing, 2017. https://link.springer.com/article/10.1007%2Fs11554-017-0722-3.

[19] "OpenCV," [Online]. Accessed 18 10 2019: https://opencv.org/about/.

[20] "HOKUYO URG-04," [Online]. Accessed 22 04 2020: https://www.hokuyo-aut.jp/search/single.php?serial=166..

[21] "Slamtec RPLiDAR A1M8," [Online]. Accessed 22 04 2020: https://www.slamtec.com/en/Lidar/A1.

[22] Z. Gong, J. Li and W. Li, "A low cost indoor mapping robot based on TinySLAM algorithm," *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pp. 4549-4552, 2016. https://ieeexplore.ieee.org/document/7730187.

[23] "Jupyter," [Online]. Available: https://jupyter.org/.

[24] "RPLIDAR Datasheet," [Online]. Accessed 22 04 2020: https://www.robotshop.com/media/files/pdf/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf.

[25] "Laser Safety Standar IEC 60825-1," [Online]. Accessed 22 04 2020: https://webstore.iec.ch/publication/3587..

[26] Xilinx, "Advanced eXtensible Interface," [Online]. Accessed 23 04 2020: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf..

[27] "HLS Pragmas," [Online]. Accessed 23 04 2020: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html.

[28] "Jupyter Setup Guide," [Online]. Available: https://pynq.readthedocs.io/en/v2.3/getting_started.html..

[29] C. Stachniss, U. Frese and G. Grisetti, "OpenSLAM," 2006. [Online]. Accessed 25 04 2020: https://openslam-org.github.io/.

[30] "Visual Studio," [Online]. Available: https://visualstudio.microsoft.com/

[31] "Pycharm," [Online]. Accessed 26 04 2020: https://www.jetbrains.com/pycharm/

[32] "RPLiDAR SDK," [Online]. Accessed 28 04 2020: http://www.slamtec.com/en/support#rplidar-a-series

[33] "LFSR," Texas Instruments, December 1996. [Online]. Accessed 30 04 2020: http://www.ti.com/lit/an/scta036a/scta036a.pdf

[34] B. Kernighan and D. Ritchie, The C Programming Language, Second Edition, Prentice Hall, 1988. http://www2.cs.uregina.ca/~hilder/cs833/Other%20Reference%20Materials/The%20C%20Programming%20Language.pdf.

[35] "Random.c," [Online]. Accessed 30 04 2020: https://code.woboq.org/userspace/glibc/stdlib/random_r.c.html#__random_r

[36] "Rand.c," [Online]. Accessed 30 04 2020: https://code.woboq.org/userspace/glibc/stdlib/rand.c.html

[37] "Virtual Box," Oracle, [Online]. Accessed 02 05 2020: https://www.virtualbox.org/

[38] "RPLiDAR SDK Setup Guide," [Online]. Accessed 02 05 2020: https://github.com/Slamtec/rplidar_sdk

Appendices

Appendix A: Risk Assessment and Standard Operating Procedure:

OÉ Gaillimh
NUI Galway

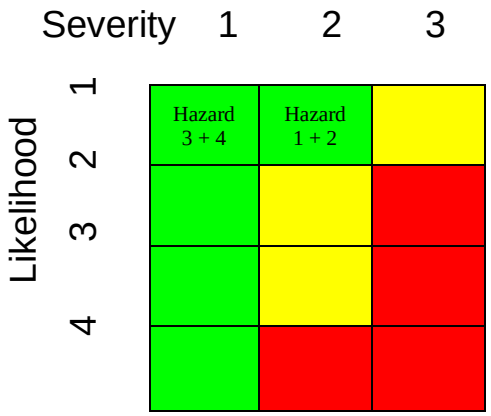
College of Engineering and Informatics, Risk Assessment.

Locations: List **ALL** Lab Location(s): ENG 3001, ENG 3002 or ENG 3003 (Based on Availability of the Lab)	Project Title: **FPGA Implementation of LiDAR Terrain Mapping System based on High-Level Synthesis**	Duration (Dates): Sept 2019 – May 2020
Person(s) at Risk:	Donal James Conneely, MCE Healthy Undergraduate Student	

Hazard(s)	**Likelihood (1-4)** (With Controls in place)	**Severity (1-3)** (With Controls in place)	**Controls** (State if normal Laboratory controls are sufficient per Safety Statement.)
1. Slip, trip or fall	1	2	Cables, personal items and objects shall be removed from the area of use of the system for by all users.
2. Electrical Exposure	1	2 (Severity Lowered due to system battery rather than mains).	Report defects to the laboratory supervisor. Faulty units are not be used to avoid risks to user health.
3. Equipment malfunction	1	1	Report defects to the laboratory supervisor. Faulty units are not be used to avoid risks to user health.
4. Injury Due to Laser	1 (Severity Lowered due to the Class I laser and the rotation of the laser while LiDAR is in use)	1 (Severity Lowered due to the Class I laser)	Wear safety glasses. Use only wire cutting and wire stripping tools provided by the discipline. Ensure only cutters with safety clips are used.

Likelihood:
1= Very Unlikely/Yearly
2= Unlikely/During a Semester
3= Likely/Weekly
4= Very Likely/Daily

Severity:
1= Slight Harm
2= Moderate Harm
3= Extreme Harm



Comments:

Risk Assessment **with** controls (Taken as the highest colour in matrix)	High	Medium	Low

Hierarchy of Controls: (Refer to Lab Safety Statement for basic controls)

Details of Engineering Controls Required:

- Additional equipment or apparatus available and approved.
- Additional Safety Signs in the work area.
- Raw Materials defined and approved.
- Any additional Noise, Vibration hazards.
- Any additional Occupational Health hazards defined and controls in place.

Details of Administrative/Procedural Controls required:

- Standard Operating Procedures in place for lab experiments.
- Provide induction training for all labs.
- Relevant MSDS available.
- Any specific Emergency Procedures are in place and approved.

General Rules:
- No Loose Hair, No Loose Clothes, No Dangling Jewellery
- Adequate Footwear required (No open toe or canvas shoes)
- Wash your hands after the laboratory session.
- No Eating or Drinking.

Prepared By: Donal Conneely
Approved by (Research/Project Supervisor(s):
Signed by Researcher/Student(s):

OÉ Gaillimh NUI Galway	**College of Engineering and Informatics**		
Subject: EE5115 ME Project Muscle Fatigue and Fitness Analyzer	Title: **HLS based FPGA Terrain Mapping System Test**		Lab Number: ENG 3001 /3002 / 3003
Objective	The purpose of this test is to perform an initial test of the HLD implementation of the tinySLAM algorithm on the FPGA.		
Health & Safety Documents	Refer to the Safety Statement for the Laboratory, and any other documents as required. Refer to User Manual relating to RPLiDAR sensors. Refer to User Manual relating to PYNQ-Z2 Board.		
New Safety Hazards			
Additional PPE Required			
Additional Engineering Controls	1. Disconnect laptop from mains for testing. Battery power for all electronics. 2. The subject will be required to move around the environment throughout the test, therefore all obstacles/ trip hazards will be removed. 3. There will be a spotter there for the duration of the test.		
Overall Risk Level Low/Med/High	**See Risk Assessment Above**		
Report all accidents to:	Lab Champion: Safety Representative:		
SOP prepared by: **Donal Conneely**	Contact Details: Email: D.CONNEELY7@nuigalway.ie	Approved By:	Date:

#1	Scope of Work/Activity: The purpose of this test is to perform a full system test of the HLS implementation of the tinySLAM algorithm using a live sensor. This involves the use of a Light Detection And Ranging (LiDAR) sensor connected to a PYNQ-Z2 FPGA. This PYNQ Board communicates via Ethernet with a corresponding Jupyter Notebooks client running on a Laptop. For this test, the system will be based around the use of an Acer Nitro 5 laptop. This laptop will be unplugged from the mains supply for the duration of the test and thus the system will be solely powered from the laptops internal battery. The FPGA, with no mains power supply, is powered using the laptop. The system will be tested by having the subject move around the environment will holding the LiDAR system in their hand. **Equipment List:** PYNQ-Z2 Board with Ethernet cable and USB to micro USB power cable, microSD setup for PYNQ interface, Slamtec RPLIDAR A1M8 with the corresponding micro USB to PH2.54-7P pitch connector and USB to micro USB power cable and an Acer Nitro 5 Laptop.
#2	Reference Documents: **PYNQ-Z2 and Setup Links:** http://www.pynq.io/board.html https://pynq.readthedocs.io/en/latest/getting_started/pynq_z2_setup.html **Slamtec RPLiDAR and SDK:** https://www.slamtec.com/en/Lidar/A1 https://www.robotshop.com/media/files/pdf/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf. http://www.slamtec.com/en/support#rplidar-a-series https://github.com/Slamtec/rplidar_sdk
#3	Environmental or Waste Management Impacts and procedures:
#4	Other Equipment:
#5	Materials:
#6	Maintenance**:**

#7

Procedure:

1. Before testing can begin, the system must be setup. Firstly, the RPLiDAR must be connected to the PYNQ-Z2 Board. This must be done before the FPGA is powered on. An image of the RPLiDAR to PYNQ connection can be seen below:

Figure 1. RPLiDAR to PYNQ-Z2

2. Next, the PYNQ-Z2 Board is connected to the Laptop. The laptop is disconnected from the mains if it is connected. Once the FPGA has been connected, the board is turned on. Once the FPGA has correctly turned on, i.e. the FPGA lights glow green, Juypter Notebooks will be opened using the FPGA and the desired notebook will be selected, as seen below:

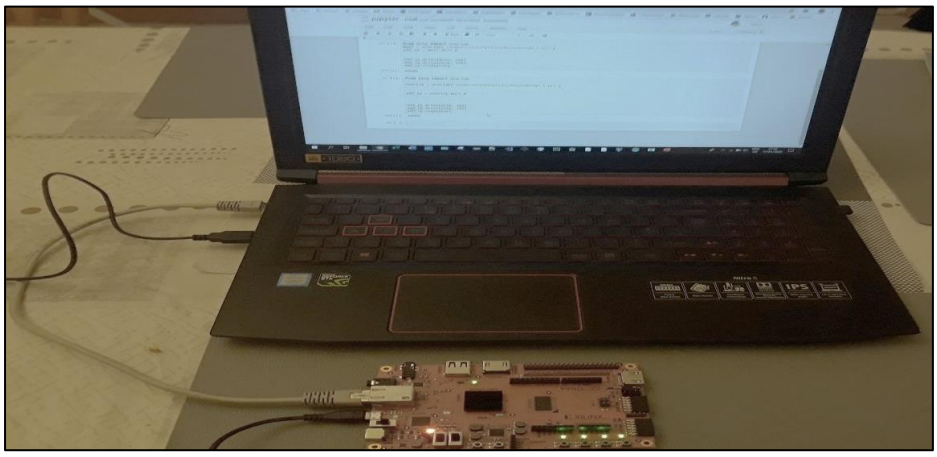


Figure 2. PYNQ-Z2 to Juypter Notebooks Client

3. Once the preparations are complete the subject will run the first two scripts of the Python Notebook. These scripts do not run the system, merely set up the required functions to be used by the system managing the External Functions and the HLS functions respectively as seen below:

```
class ExtendedFunctions():
    def __init__(self):
        self.TEST_SCAN_SIZE = 682
        self.TEST_MIN_DIST  = 20
        self.TEST_ANGLE_MIN = -180
        self.TEST_ANGLE_MAX = +180
        self.TEST_OFFSET_LASER = 145
        self.TEST_HOLE_WIDTH = 600
        self.M_PI = 3.14159265358979323846

        self.TS_SCAN_SIZE = 683
        self.TS_MAP_SIZE = 360
        self.TS_MAP_SCALE = 0.017578125
        self.TS_DISTANCE_NO_DETECTION = 4000
        self.TS_NO_OBSTACLE = 65500
        self.TS_OBSTACLE = 0
        self.TS_HOLE_WIDTH = 600

        self.val = 1
        self.RAND_MAX = 0x7fff

        self.my_so = '/home/xilinx/LinLib/SharedLibrary/libLiDAR.so'
        self.myLiDAR = cdll.LoadLibrary(self.my_so)
        my_string = "/dev/ttyUSB0"
        self.port = c_char_p(my_string.encode('utf-8'))

    #Start the Sensor
    def Start_Sensor(self):
        mp = (c_float * self.TEST_SCAN_SIZE)()
        select = c_int(0)
        if (self.myLiDAR.main(mp, self.port, select) == 1):
            return True
        else:
            return False

    #Stop the Sensor
    def Stop_Sensor(self):
        mp = (c_float * self.TEST_SCAN_SIZE)()
        select = c_int(2)
        self.myLiDAR.main(mp, self.port, select)

    #Read Scans from sensor
    def Read_Sensor(self, scan):
        mp = (c_float * self.TEST_SCAN_SIZE)()
        select = c_int(1)
        self.myLiDAR.main(mp, self.port, select)

        nb_points = 0
        SPAN = 1
        for i in range(0, self.TEST_SCAN_SIZE):
```

Figure 3. External Functions Notebook

```
class HLS_Functions():

    ////////////////////////////////////////////////// Bit Converters //////////////////////////////////////////////////
    def floatToBits(self, f):
        s = struct.pack('>f', f)
        return struct.unpack('>l', s)[0]

    def BitsTofloat(self, i):
        s = struct.pack('>l', i)
        return struct.unpack('>f', s)[0]

    ////////////////////////////////////////////////// MAP INIT //////////////////////////////////////////////////////
    def HLS_map_init(self):
        #Initialize Variables
        Run = xlnk.cma_array(shape=(1,), dtype=np.uint32)
        map1 = []

        #print("Activate Map Init...")
        Run[0] = 1
        dma_run.sendchannel.transfer(Run)
        dma_run.sendchannel.wait()

        #Delay to Allow time for calculation to complete
        time.sleep(0.1)

        #print("Reading Map...")
        for v in range(0, 64800):

            #Read in the 32 bit value
            location = map_start + 4*v
            val = tiny_ip.read(location)

            #Split into two 16 bit values
            upper = (val >> 16) & 0xFFFF
            lower = val & 0xFFFF

            #Append
            map1.append(lower)
            map1.append(upper)

        #print(map1)
        return map1

    ////////////////////////////////////////////// MONTE CARLO + MAP UPDATE ////////////////////////////////////////////////
    def HLS_Main(self, scan, mp, pos):
        #Initialize Variables
        Run = xlnk.cma_array(shape=(1,), dtype=np.uint32)
        pos2 = xlnk.cma_array(shape=(3,), dtype=np.float32)
        map1 = []

        x = 0
```

Figure 4. HLS Functions Notebook

4. Once the setup scripts have been run the main tinySLAM algorithm can be run. Running this code will cause the system to take readings from the LiDAR once every cycle and these readings will be used to update the position and the map. While the code is running the subject will be required to move around the room. The subject must move at a slow speed to ensure the system correctly determines its current location and updates the map appropriately.

Figure 5. Subject moving the LiDAR

5. The system is designed to continuously display location data on the screen. Every 10 scans, the system will display a map. An example of a map constructed using test data, (no live sensor involved) can be seen below:

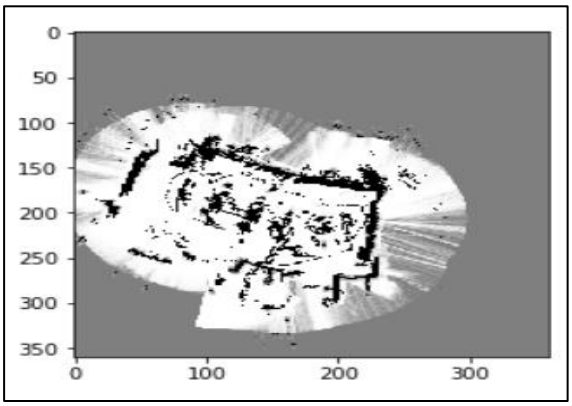


Figure 6. Sample Map

Appendix B: Project Code GitHub:

All the code of the project including the C code, Python code, tinySLAM code and Jupyter Notebooks code can be found on GitHub at the following link

https://github.com/DonalJamesConneely/EE5115-Masters-Project_HLS-Terrain-Mapping.git