ABSTRACT

Accelerating Path Planning Algorithms with High Level Synthesis Tools and FPGAs

John Trower, M.S.E.C.E

Chairperson: Russell W. Duren, Ph.D.

Accelerating path planning algorithms with field programmable gate arrays (FPGA) allows the designer to achieve significant performance increases over using a traditional central processing unit (CPU). Converting an algorithm to run on an FPGA is a complicated and time consuming process. This thesis develops and verifies a design framework that demonstrates how to design a path planning algorithm in a high level language, then convert the algorithm into hardware description languages using high level synthesis tools. This design framework will be used to demonstrate the acceleration of a genetic algorithm.

Accelerating Path Planning Algorithms with High Level Synthesis Tools and FPGAs

by

John Trower, B.S.E.C.E

A Thesis

Approved by the Department of Electrical and Computer Engineering

Kwang Y. Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

Russell W. Duren, Ph.D., Chairperson

Michael W. Thompson, Ph.D.

Randal L. Vaughn, Ph.D.

Accepted by the Graduate School
December 2012

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

I would like thank the members of my committee – Dr. Russell Duren, Dr. Michael Thompson, and Dr. Randal Vaughn – and especially Dr. Duren who worked with me while I created and validated this design framework. I would also like to thank Dr. Duren and the Electrical and Computer Engineering Department at Baylor University for purchasing the hardware necessary to complete this research. Finally, I would like to thank my parents for providing encouragement and advice while working on this thesis.

CHAPTER ONE

Introduction

Designing an optimized path for a vehicle to take through an unknown environment is a problem that has been researched for years and many different algorithms have been developed to solve this problem. Most of these algorithms are designed to produce a path for the main vehicle. The path planning algorithm that our research team is developing is to solve the path for an escort vehicle that is providing protection to the main vehicle. The Navy has asked Baylor University to investigate potential improvements to their Electronic Attack (EA) path planning algorithm that was previously developed for use in the Navy's mission planning system. The input for this system is the path for the protected entity (PE). Based on the path of the PE, our algorithm has to create the path for an EA aircraft which will provide suppression of enemy air defense systems through different jamming techniques.

To solve this problem, the research team started by creating a MATLAB program based on the algorithm described in patent [1]. The algorithm that is described in this patent is for an onboard computer inside an EA aircraft. Our implementation of this algorithm analyzes the Electronic Order of Battle (EOB) library, along with the path for the PE, to determine where the EA needs to be positioned in order to provide adequate protection to the PE. The area where the EA can be and provide adequate protection is known as the Jam Acceptability Region (JAR). It should be noted that the best position for the EA to be located, from a jamming point of view, is next to the PE. Unfortunately, this is usually not feasible for two reasons; to avoid revealing the position of the PE and to stay outside the lethal range of any enemy air defense systems such as surface-to-air missiles (SAMs) and anti-aircraft artillery.

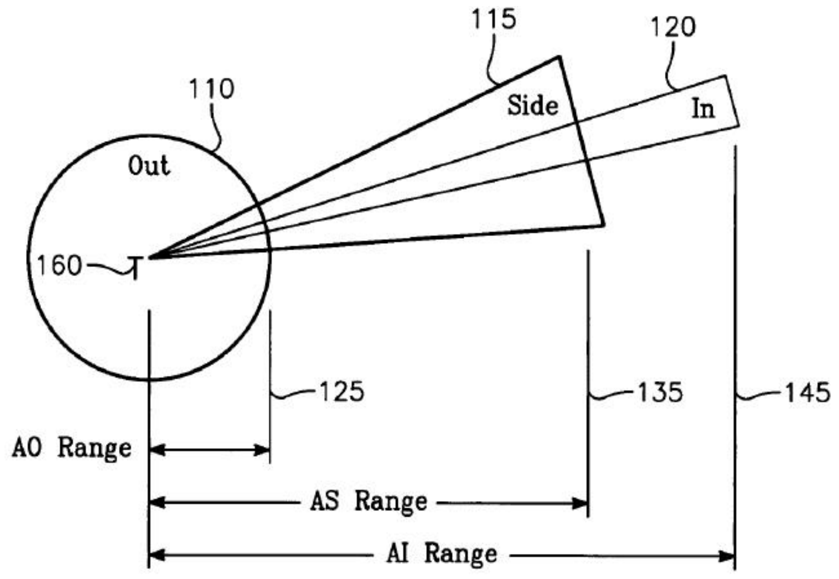

Figure 1: Jam Acceptability Region Contours Model (Figure from [1])

The JAR is defined as "the family of positions an EA may occupy and still provide effective jamming to protect the PE." [1] The figures in this section are from a patent [1] and the bold numbers refer to labels in the figures. Figure 1 shows the three JAR contours that graphically represent each radar (**160**). Each of the three contours relates to the antenna pattern of the radar system, where **120** represents the main-lobe ellipsoid, **115** represents the side-lobe ellipsoids, and **110** represents the back-lobe ellipsoids. The distance between the PE and the enemy radar will determine where the EA has to be and what type of jamming technique will be

Table 1: Jamming Techniques (Details from [1])

Abbreviations	Jamming Approaches
PAO	Preemptive Assignment - Out of Alignment
PAS	Preemptive Assignment - In Side-Lobe Alignment
PAI	Preemptive Assignment - In Main-Lobe Alignment
RAO	Reactive Assignment - Out of Alignment
RAS	Reactive Assignment - In Side-Lobe Alignment
RAI	Reactive Assignment - In Main-Lobe Alignment

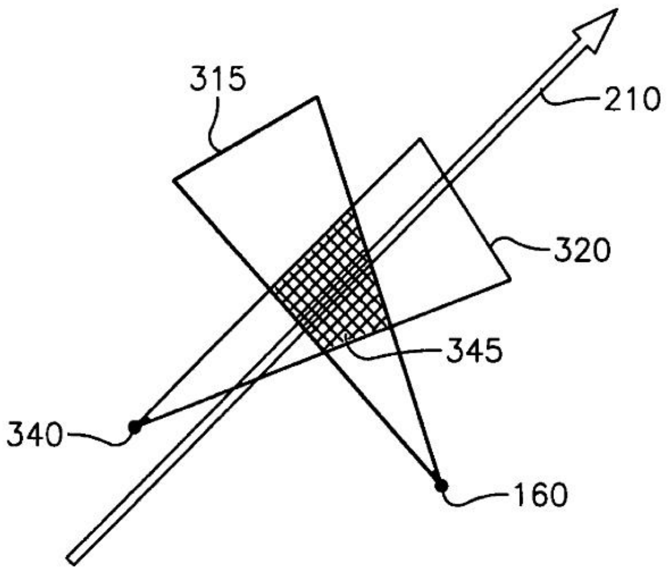

Figure 2: Jam Acceptability Region Overlap Area (Figure from [1])

required to provide adequate suppression. A list of the six jamming techniques outlined in [1] and used in our algorithm is provided in Table 1.

If multiple enemy radar systems are near each other, it is possible to have an overlapping JAR. This can be seen in Figure 2. When this occurs, the area where the EA needs to be is smaller than if there was only one radar, and it also requires the EA to use possibly more than one technique to suppress the enemy radar. In Figure 2, the enemy radar stations are represented by points **160** and **340**. The path of the PE is shown by the arrow labeled **210**. When the PE first enters within the range of radar **340**, the EA can be anywhere within JAR **320** as long as the EA stays outside the lethal range of any threats. When the PE enters within the range of radar **160**, the EA now has to be in the overlapping area of both **315** and **320** JARs. Once the PE has left the range of radar **160**, the EA is able to fly anywhere within JAR **320** as long as the EA is outside the lethal range of any threats.

Once our algorithm has analyzed the entire PE flight path and determined the JARs and jamming tactics at each time step, this information is passed off to one or more path planning algorithms. The path planning algorithms are used to create a flight path for the EA aircraft that will make sure the EA is in the correct area to provide protection for the PE.

The part of the project that this thesis is focused on is creating and verifying a design framework where different path planning algorithms can be created and tested in high level languages, like MATLAB and C/C++. The design framework then provides a path to convert the high level code into hardware description languages that run on Field Programmable Gate Arrays (FPGAs) with the goal of greatly decreasing the time required to find a good flight path for the EA.

Thesis Outline

Chapter One provided a background of the project and the problem that this thesis is addressing. Chapter Two will provide background on current research in the field of path planning as it relates to accelerating path planning algorithms with FPGAs. Chapter Three will introduce the hardware and software development environment chosen for the research. It will also provide an overview of how the algorithm was created and show the steps necessary to convert the algorithm into code that is able to run on an FPGA. Then Chapter Four will provide some design considerations and show the performance of the algorithm on the FPGA. The last chapter will summarize the project and outline some future research that needs to be performed to fully accelerate path planning algorithms in FPGAs.

CHAPTER TWO

Related Work

Path planning is a topic that has been researched for years and there are many resources available that explain in detail many different algorithms. For this research a couple of criteria were established to evaluate potential path planning algorithms. The first criterion was that the path planning algorithm had to be designed to work in a three dimension space and have been applied to path planning for aircraft. The second criterion was that the algorithm had to be known to work on an FPGA. With this criteria established a literature review was performed to find potential algorithms to use to accelerate our path planning system.

The main resource that was used to determine which path planning algorithm this thesis would focus on was based on a master's thesis written by Cocaud [2]. The main objective of Cocaud's thesis was to design a control scheme that could be used to improve the autonomous controller's trajectory planning capabilities that are currently used in unmanned aerial vehicles (UAVs). To do this Cocaud starts by providing a summary of the different path planning algorithms that have been developed and provides the advantages and disadvantages for each algorithm. Cocaud concludes based on the literature that genetic algorithms provide the best solution. Here is Cocaud's justification:

> Genetic Algorithms (GA) belong to the randomized search category. Compared to the other search algorithms of the same category, they have the advantage of not requiring the gradient of the objective function, nor the constraint functions. Also, its highly randomized nature drastically reduces the chances of being trapped in local optimums. Moreover, the time complexity of GAs are usually much lower than conventional search algorithms such as A*, Dijkstra, or even the simplex algorithm when linear problems are considered. [2]

After Cocaud determined that the best algorithm to use is the genetic algorithm, he outlined the details of his improved autonomous controller design and provided results based on a computer program that he created. The multi-objective genetic algorithm that Cocaud developed was later converted to run on an FPGA in [3]. In this paper, Allaire et al. outlined how they took the design that Cocaud presented in his thesis and converted the algorithm to run on an FPGA. For the algorithm to be converted to work on the FPGA, the population characteristics had to be predefined. The population was set to a predefined value to help limit the amount of FPGA resources that would be required for the design. This also led to predefining the number of children generated and the number of best solutions kept from each generation.

Allaire et al. designed the FPGA to interface with the computer program over a UART. In this design, the FPGA only computes the selection, population update, crossover, and mutation. The authors designed each of these units to run in parallel, which means that there are multiple instances of each unit on the FPGA. Each of these units was also designed to only take one clock cycle. The mutation unit is the last unit to be executed and when it completes, the current population data is sent to the computer and the computer program computes the cost of each population, sorts the population based on the cost, and then transmits the sorted population to the FPGA. This cycle repeats until one of the exit conditions is met.

The results that Allaire et al. were able to achieve were also presented. They were only able to get three of the four units implemented on the FPGA. The three units that were implemented were the selection, crossover, and population update units. The reason they list for not implementing the mutation unit was that there were not enough resources available on the selected Virtex 2 FPGA board, the ML310. They concluded in the end, that if they were able to acquire an FPGA with more resources to where they could compute the fitness of all the children solutions at the same time and add the evaluation unit to the FPGA, that they could compute a new path every 100ms. The current software implementation takes approximately 50 to 60 seconds. If they were able to achieve a new path every 100ms then this design could be used to perform real-time path planning.

In [4], Kok et al. also implemented Cocaud's multi-objective genetic algorithm on an FPGA with the goal of creating a completely synthesizable design. In contrast to the previous mentioned implementation, Kok et al. not only included the basic genetic algorithm tasks in their FPGA design but they also included a main control unit which "provides a memory interface, sorting algorithm, population update mechanism, and monitors the termination criterion of the evolutionary algorithm." [4] This design also includes Light Detection And Ranging (LiDAR) terrain data onboard the FPGA. The authors met their goal of being able to synthesize their entire genetic algorithm design for a Virtex 4 FPGA board but they provide no indication of testing the design on the selected hardware platform. They provide timing results comparing the software implementation and the hardware implementation but the hardware implementation is based on simulator results and not actual hardware results. Based on the simulated hardware implementation, the authors' design was able to achieve a 260,850x improvement over the software implementation.

Kok et al. in the end conclude that one of the main difficulties in the design was designing the algorithm in low level hardware description languages. The authors state:

> One development difficulty encountered was that the development time for the Hardware-EA using a low level hardware description language (HDL) such as VHDL is very time consuming. Future work will explore utilizing a high level HDL such as Mitrion-C, Handel-C or Impulse C. [4]

This provides the basis for this thesis, the creation of a design framework that allows the developer to design an algorithm in a high level language and convert it to run on an FPGA. The design framework that was created for this thesis will be presented in the next chapter.

CHAPTER THREE

Motivation and Design Framework

Converting algorithms written in high level languages, like C/C++ or MATLAB, into a hardware description language (HDL) that is able to run on an FPGA is a complicated task. The use of high level synthesis tools can help make this process easier. The purpose of this thesis was to develop a design framework to guide this process. The framework that was developed is shown in Figure 3. In this research, the path planning algorithm was originally designed in MATLAB. After the algorithm was optimized, the algorithm was converted into C/C++ with the use of MATLAB Coder Toolbox. The generated C/C++ code was then converted into Verilog code with the use of Xilinx Vivado HLS. In the following sections, the motivation for creating this framework will be presented and the design framework will be outlined in greater detail.

Motivation

There are many reasons why a designer would want to design an algorithm in a higher level language and then compile the design into lower level languages. The higher level language used in this thesis was MATLAB. MATLAB was selected for the following reasons. First, MATLAB it is a very powerful environment for the development and verification of algorithms. Second, MATLAB code is at a higher level of abstraction than code written in C/C++. Third, MATLAB is widely used in academia and industry because it has many built-in features that make it easy to (a) work with array and matrix variables, (b) work with a wide range of data types including complex (imaginary) variables, and (c) develop graphical interfaces and displays and perform complicated 3-D data visualization. Fourth, MATLAB can also be supplemented with a wide array of toolboxes that provide commands to

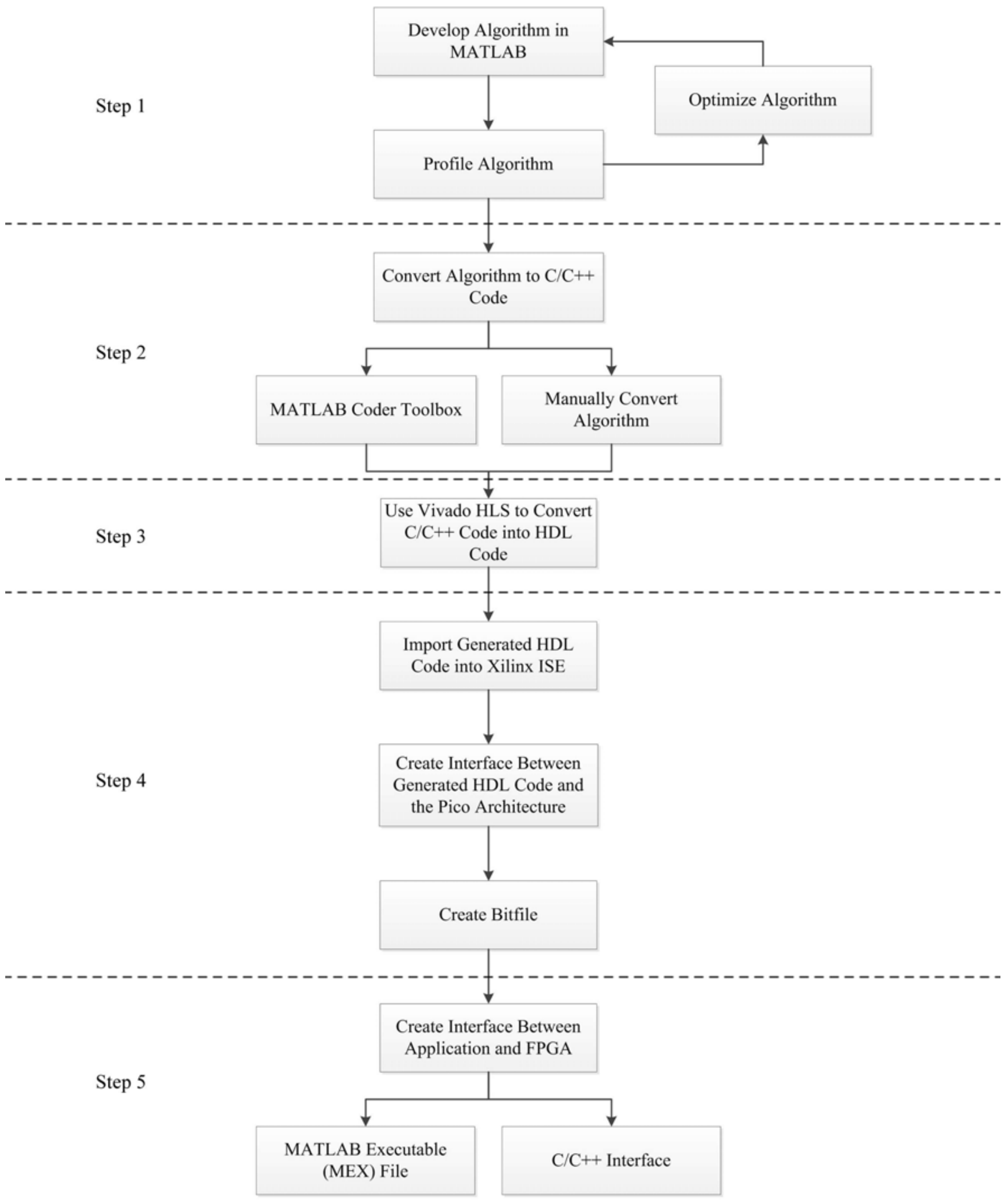
Develop Algorithm in MATLAB
Optimize Algorithm
Step 1
Profile Algorithm
Convert Algorithm to C/C++ Code
Step 2
MATLAB Coder Toolbox
Manually Convert Algorithm
Use Vivado HLS to Convert C/C++ Code into HDL Code
Step 3
Import Generated HDL Code into Xilinx ISE
Create Interface Between Generated HDL Code and the Pico Architecture
Step 4
Create Bitfile
Create Interface Between Application and FPGA
Step 5
MATLAB Executable (MEX) File
C/C++ Interface

Figure 3: Design Framework

perform signal processing, optimization, etc. The most important toolboxes for this research were the Mapping Toolbox and the MATLAB Coder Toolbox. MATLAB is an interactive environment that allows the designer to quickly see the impact of changes made to the algorithm making verification much easier to perform.

MATLAB does have one disadvantage that is the subject of this thesis. That is its execution speed. There are multiple ways a designer can increase the execution speed in MATLAB. The first is the designer can make algorithm improvements by finding better ways to compute their solution. A second approach would be to improve the MATLAB code. This would include fixing any part of the code that MATLAB warns about like preallocating memory for variables that change sizes within loops. Another change a designer can do is to suppress any unnecessary output. Third, a designer can use different MATLAB toolboxes to increase the execution speed of their algorithm. MATLAB provides the MATLAB Coder Toolbox that the designer can use to compile the time critical routines into a MEX file which MATLAB can execute like a normal MATLAB function. Lastly, hardware acceleration can be used to increase the speed. MATLAB provides the Parallel Computing Toolbox which allows the designer to take advantage of multicore processors and GPUs on a single machine or on a cluster of machines. The other hardware acceleration platform that a designer can use is an FPGA which is the subject of this thesis.

In particular, the thesis seeks to create and verify a framework that allows the designer to quickly port portions of an algorithm from high level MATLAB code to an FPGA accelerator and be able to utilize the FPGA accelerator from the MATLAB environment. One of the research goals was to determine if Vivado HLS can work with MATLAB generated C/C++ code. Would many changes be required to get MATLAB C/C++ code to work with Vivado HLS? If so, the goal was to identify the changes that are required. The next goal of this research was to create as simple an interface as possible between MATLAB and the code running on the

FPGA and to then characterize how efficient the resultant system is and compare timing to the MATLAB code. If satisfactory results are obtained, the objective was to document how to repeat the process. Otherwise the goal was to determine if there are additional lines of research that might result in an improved system.

Hardware Platform

In addition to developing the design framework, this research involved determining which FPGA board would be a good platform for hardware acceleration. This involved not only picking the FPGA board but also evaluating the software API that was provided with the board, the available resources onboard, and the interface that was available between the FPGA and the computer. The target FPGA board that was selected for this thesis was the M-503 module connected to the EX-500 backplane. Both of these boards were developed by Pico Computing, Inc. A picture of both the FPGA module and backplane can be seen in Figure 4 and Figure 5 respectively.

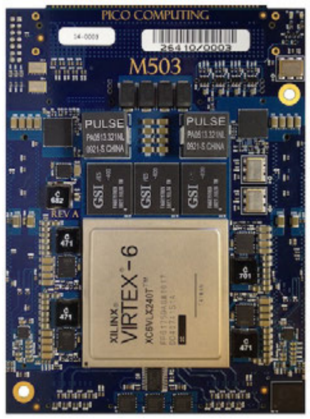

Figure 4: M-503 Module (Figure from [5])

The M-503 FPGA module and the EX-500 backplane were selected because of their compatibility with both Linux and Windows, the C++ API that was provided, and for the PCI Express interface. These three benefits allowed the research to be performed in the research lab and also provides the capabilities for this research to be used in a variety of different environments as the project moves forward.

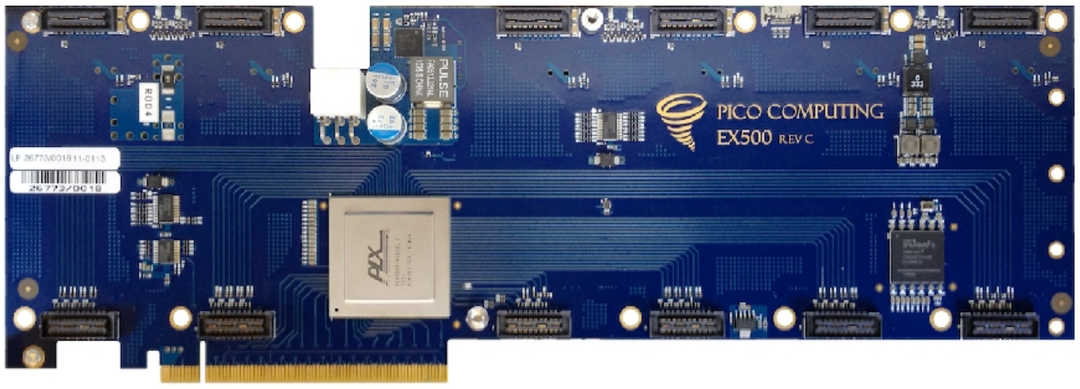

Figure 5: EX-500 Backplane (Figure from [6])

The M-503 module was also selected because of its FPGA resources. The FPGA's specifications are listed in Table 2. The backplane that was selected supports up to two more M-503 modules. This would allow the designer to have multiple different path planning algorithms running at the same time or completely different programs utilizing different FPGAs.

Table 2: Pico M-503 Specifications (Specifications from [7])

Resource	Specification
Device	XC6VLX240TFF1759-2
Logic Cells	241,152
Configurable Logic Blocks	
Slices	37,680
Max Distributed RAM (Kb)	3,650
DSP Slices	768
BRAM Blocks	
18 Kb	832
36 Kb	416
Max (Kb)	14,976
CMTs	12
PCIe	2
GTXs	4

The computer that this research was performed on was a custom built computer for this research project. The specifications of the computer are listed in Table 3.

The different software packages that were selected and used to develop and convert the genetic algorithm are listed in Table 4.

Table 3: Computer Specifications

Components	Specification
Operating System	Ubuntu Release 10.04 (lucid)
Kernel	Linux 2.6.32-42-generic
Processor	Intel Core i7 CPU 960 @ 3.2GHz
Memory	20GB @ 1067 MHz

Table 4: Software Versions

Software	Version
MATLAB	7.12.0 (R2011a)
MATLAB Coder Toolbox	2.0 (R2011a)
Xilinx ISE Design Suite	14.2
Vivado HLS Standalone	2012.2
GNU Compiler Collection (GCC)	4.4.3-4ubuntu5.1
Pico Computing Framework	5.0.6.1

Design Framework

Step 1: Initial Algorithm Development in MATLAB

The design framework starts with the algorithm being developed in MATLAB. MATLAB was selected because it provides a very robust way to visualize data, is very efficient at performing complex operations on matrices, and provides multiple ways to convert MATLAB code into lower level languages. As the algorithm is being developed, the designer needs to make sure to correct any warnings that MATLAB marks as the algorithm is developed. Fixing these warnings as the algorithm is being developed will result in the algorithm running faster and this will prevent problems later on in the framework.

Once the algorithm has been developed, the algorithm needs to be profiled to see if there are time consuming sections in the code that can be optimized by the designer. The profile results also let the designer know how many times different sections of the code are being executed. The most time consuming sections are highlighted with red lines. To fix these computational intensive sections, the designer might look for a MATLAB function that performs the necessary function or try a different approach to solving the time consuming section. There will not always be a better solution, but it is worth trying to optimize the algorithm now before continuing on with the design framework.

After the algorithm has been optimized, the data type of all the variables needs to be specified. MATLAB defaults to all numbers being represented as a double. Most algorithms do not need that level of precision, so the designer needs to determine what level of precision is needed and set each variable accordingly. By doing this step, the execution time of the algorithm might take longer in MATLAB than when everything was set as a double. The potential performance loss in MATLAB is not a real problem, because the algorithm is not being optimized to run in MATLAB, but to run on an FPGA where having everything set to the proper precision will help speed up the algorithm and require fewer resources.

Step 2: Convert MATLAB Algorithm to C/C++ Code

The next step is to convert the MATLAB code into C/C++ code. According to Figure 3, there are two options for this step. The designer can convert that MATLAB code by hand or use the MATLAB Coder Toolbox to convert the code into a C/C++ Static Library. The option that will be discussed in this thesis will be using the MATLAB Coder Toolbox. The flowchart for how to use MATLAB Coder Toolbox is provided in Figure 6. To run the MATLAB Coder Toolbox, type '`coder`' in the MATLAB Command Window.

Once the MATLAB Coder Project has been set up, the designer is ready to create a MATLAB Executable (MEX) file. The reason that the MATLAB Coder Toolbox flowchart suggests creating a MEX file before converting the code to C/C++

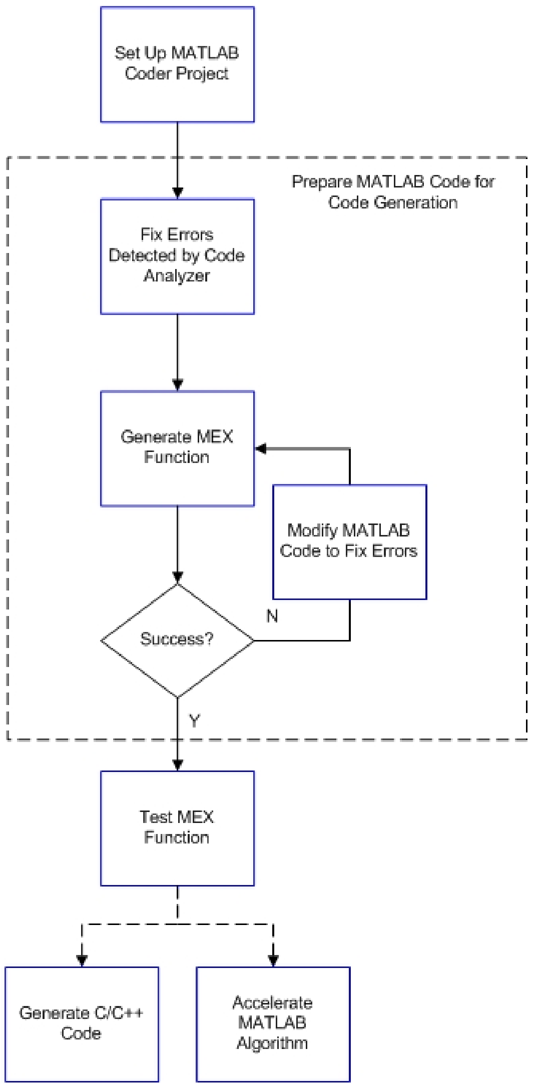
Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Fix Errors Detected by Code Analyzer
Generate MEX Function
Modify MATLAB Code to Fix Errors
Success?
N
Y
Test MEX Function
Generate C/C++ Code
Accelerate MATLAB Algorithm

Figure 6: Converting MATLAB Code to C/C++ Code (Figure from [8])

Static Library is to allow the designer to verify that MATLAB has converted the algorithm properly. The MEX file may also result is significant and perhaps sufficient speed up as compared to the original MATLAB code. A MEX file is compiled C/C++ code with a wrapper around it so that MATLAB is able to execute the MEX function just like the original function. This allows the designer to verify that with the same inputs the MEX file generates the same output as the original algorithm. Converting the MATLAB code to a MEX file is not guaranteed to work initially since not all

MATLAB functions are able to be converted by the MATLAB Coder Toolbox. To fix this, the designer will have to recode that section of the code using a different MATLAB function or create the required functionality in the designer's MATLAB code.

After all of the error messages have been taken care of and MATLAB Coder Toolbox reports a successful build, the designer is able to view the report. In the report, the designer is able to explore how MATLAB converted the code. The designer is also able to verify that all of the variables are set to the proper precision by using the mouse and hovering over all of the variables in the original MATLAB code.

Once the MATLAB Coder Toolbox has successfully built the MEX file, the designer needs to test the MEX file. The MEX file is executed just like a normal MATLAB function, except now that it is compiled, the file will execute faster than the original algorithm. If the compiled MEX file is fast enough to meet the design requirements, then the designer is finished. If the performance is still not where it needs to be, then instead of creating a MEX file, change the "Output Type" to "C/C++ Static Library" and select "Generate code only". With the algorithm now converted into C/C++ code, the designer is ready to import the code into Vivado HLS to convert the code to HDL.

Step 3: Convert C/C++ Code into HDL Code

The Xilinx Vivado HLS tool, previously known as Xilinx AutoESL, was used to convert C/C++ code into HDL code that can be implemented on an FPGA. Vivado HLS tool was selected for use in this thesis because this tool is designed to optimize HDL code to run on Xilinx FPGAs. The target platform for this thesis is a Xilinx Virtex 6 FPGA which means this tool is able to optimize the code for the amount of BRAM, DSP48E, Flip-Flops, Look-Up Tables, and other resources available on the target FPGA. The version of the software that was used in this thesis is Vivado HLS

Standalone. The standalone version allows the designer to target Xilinx FPGAs that were developed before the Xilinx 7 Series FPGAs were released.

The Vivado HLS flowchart is shown in Figure 7. Start by creating a project with the C/C++ code that was generated by MATLAB Coder Toolbox. When prompted for the C/C++ code, only select C/C++ source files and not any header files. The designer might want to first copy the source files to the directory where the Vivado HLS project is being created so that the source code will not change unexpectedly if the MATLAB Coder Toolbox is run again. This step is required if the designer plans to insert directives into the C/C++ code as they will be lost if MATLAB Coder Toolbox is rerun. When prompted to select a testbench file, unless the designer has previously created a testbench file, the designer can leave this blank and add one later or create a testbench file now while setting up the project. MATLAB Coder Toolbox does not create this testbench file so the designer will need to be familiar with coding in the same language as the generated source code (i.e. C or C++). Vivado HLS requires that the testbench file be coded in the same language as the top level source code file.

Now that the project is setup, the designer can verify that the generated C/C++ code can be synthesized into HDL by selecting "Synthesis" from the menu. After verifying that the code is synthesizable, the designer will want to create a testbench file. In Vivado HLS, this testbench file is not just for testing the generated C/C++ code but it can also be used to test the synthesized HDL code as well. This allows the designer to write a single testbench in a higher level language and use it to not only verify the C/C++ code but also to verify that the generated HDL code functions the same as the C/C++ code. If the testbench does not generate the same results before and after HLS, the designer knows that the problem is most likely not with the testbench but in the way the code was converted.

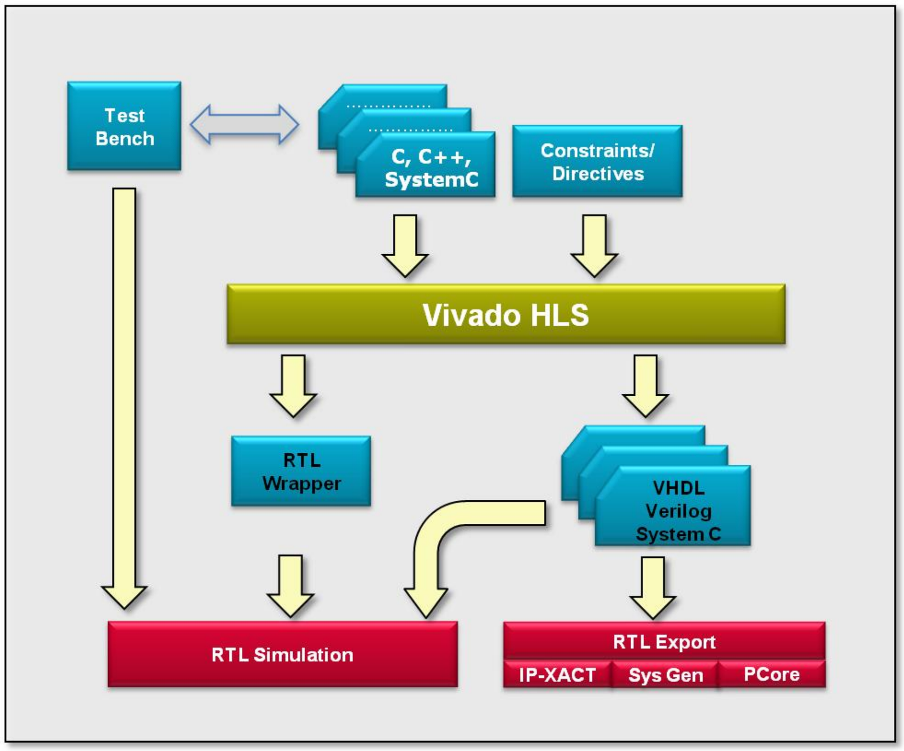

Figure 7: Vivado HLS Flowchart (Figure from [9])

For this thesis, the testbench file was written in C because all of the source files that were generated by MATLAB Coder Toolbox were C files. To simplify creating the testbench, the inputs that were used in MATLAB were converted into a hexadecimal number and written to a CSV file. The reason the input arguments were converted into hexadecimal numbers before writing them to a CSV file was to preserve the data type that was defined in MATLAB. With all of the inputs written to CSV files, they can easily be read into a C/C++ testbench. This allows the designer to use the same inputs in Vivado HLS that were used in MATLAB. The CSV files need to be added to the Vivado Project by selecting "Test Bench" in the design hierarchy and adding the files to the project.

With the testbench file now created with the same inputs that were used in MATLAB, the designer is now able to verify the generated C/C++ code and simulate the synthesized HDL code. The designer will notice that testing the C/C++ code will be considerably faster than testing the synthesized HDL code. For the algorithm that was developed in this thesis, to test the entire synthesized HDL code would take days if not weeks. In contrast, testing the C/C++ code takes less than ten minutes. When simulating the design, Vivado HLS defaults to using a built-in simulator if the designer does not specify one of the three supported simulators. For this thesis the built-in simulator was used. While the design is being simulated, the built-in simulator is creating Value Change Dump (VCD) files so that the designer is able to analyze the waveforms to verify that the design is working properly. The waveform files will be located in '`<Project_Name>/<Current_Solution>/sim/systemc/`'.

Step 4: Create Interface for Generated RTL Code

Now that the Vivado HLS project has been successfully tested, the designer has two options on how to proceed with the design. The designer can export the project to ISE for implementation on the FPGA or the designer can start to optimize the algorithm through the use of directives. The thesis is going to discuss how to export the project to ISE since Xilinx provides detailed instruction on how to use directives to optimize the algorithm in [10]. To export the design for integration into an ISE project, the design needs to be exported as a "Pcore for EDK". The option to have the RTL (register-transfer level) evaluated in Vivado HLS does not need to be selected since the design will be implemented in ISE. The files that need to be added to the ISE project will be located in '`<Project_Name>/<Current_Solution>/impl/verilog`' or '`<Project_Name>/<Current_Solution>/impl/vhdl`'. The designer has the option to use either the generated VHDL or Verilog code. In the rest of this section, the thesis is going to refer to the selected code as RTL. Now that the designer has imported the generated RTL code into the ISE project, the designer needs to instantiate the top level function. The top level function will have the same name as the top level MATLAB function and the top level function in the generated C/C++ code. The designer can use Xilinx ISE to create an instantiation template for the top level function. The designer then needs to connect the ports of the instantiated block to the appropriate signals.

Vivado HLS requires all input and output signals in a C or C++ design to be assigned to one of ten different interface types. The ten different interface types are outlined in Table 5. Depending on what input and output arguments the developer had designed in their MATLAB algorithm, Vivado HLS will determine which interface will need to be used. The designer can override the interface that Vivado HLS selects with the use of directives. Vivado HLS will create the correct signals that are needed for each interface and it is up to the designer to connect these signals to their custom code.

If the MATLAB function had input or output arguments that were arrays, Vivado HLS will determine whether to implement the arrays as either single or dual-port RAMs. The designer can force Vivado HLS to implement a certain type of memory interface with the use of directives. The generated RTL code will include the interface signals for the RAMs but will not have instantiated the RAMs in the generated RTL. Table 6 shows the signals that Vivado HLS creates to interface with a dual-port RAM. An example of how the BRAM would be instantiated and how these signals would be connected to the BRAM can be seen in Listing 1. As can be seen from the generated signals that Vivado HLS created and from the example listing, there is no interface to initialize the BRAM. Vivado HLS did not generate signals for writing to the BRAM because this BRAM is used as an input to the algorithm. It is up to the designer to determine how the BRAM is initialized. The BRAM could be initialized with custom HDL code or the BRAM could be initialized by a program running on the computer. In the code developed for this thesis, the way that the BRAMs are initialized is to connect each BRAM to a stream from the

Table 5: Vivado HLS Supported Interface Types (Descriptions from [10])

I/O Protocol	Description
ap_ack	Can be specified on any function argument except arrays and provides an additional acknowledge port to indicate input data has been read by this RTL block or confirm output data has been read by a downstream RTL block.
ap_bus	This interface implements pointer and pass-by-reference variables as a general purpose bus access similar to a typical DMA interface.
ap_ctrl_none and ap_ctrl_hs	These interface types can only be specified on the function return argument. The ap_ctrl_hs is the default and adds function level control signals: an input start signal, output idle and done signals. If there is a function return argument, the done signal signifies when the return value is valid. The ap_ctrl_none type ensures these control signals are not added to the design.
ap_fifo	An ap_fifo interface can be specified for pointer, array or pass-by-reference arguments. An ap_fifo interface implements the data accesses as reads and writes to a FIFO, with associated empty, full and data valid signals.
ap_hs	Implements each argument with an RTL port supported by a full two-way acknowledge and valid handshake. This can be specified for any function argument except arrays.
ap_memory	This interface is the default type for arrays arguments and can only be specified on array arguments. An ap_memory interface results in an RTL implementation which accesses the array elements as data values in a RAM, with associated address, chip enable and write enable control signals. The set_directive_resource command should be used to identify which RAM resource in the technology library is used for the array: this will in turn specify the number of ports available and which control signals are implemented.
ap_none	This interface provides no additional handshake or synchronization ports and can be applied to any function argument type except arrays. This is the default interface type for all read-only arguments (input ports), except array arguments.
ap_ovld	Identical to the ap_vld interface except that it only applies to write-only arguments (RTL output ports). This is the default interface type for all write-only arguments, except array arguments.
ap_vld	Can be specified on any function argument except arrays and provides an additional data-valid port to indicate when input data is valid and can be read or when output data is valid.

```
parameter TIME_R_RAM_WIDTH = 32;
parameter TIME_R_RAM_ADDR_BITS = 7;

(* RAM_STYLE="{BLOCK |  BLOCK_POWER1 | BLOCK_POWER2}" *)
reg [TIME_R_RAM_WIDTH-1:0] TIME_R_RAM [(2**TIME_R_RAM_ADDR_BITS)
    -1:0];

always @(posedge ap_clk) begin
    if (time_r_ce0) begin
        time_r_q0 <= TIME_R_RAM[time_r_address0];
    end
    if (time_r_ce1)
        time_r_q1 <= TIME_R_RAM[time_r_address1];
end
```

Listing 1: Block RAM Instantiated in User Logic

computer.[1] Listing 2 shows code that was developed for this thesis and demonstrates how a BRAM was initialized with a stream. The added logic does not interfere with the signals that Vivado HLS generated, so both the generated code and the BRAM initialization code will work without interfering with each other.

Table 6: Example of Signals Vivado HLS Creates to Interface with a BRAM

Signals	Object	Type	IO Protocol	Direction	Bits
`time_r_address0`	`time_r`	`array`	`ap_memory`	out	7
`time_r_ce0`	-	-	-	out	1
`time_r_q0`	-	-	-	in	32
`time_r_address1`	-	-	-	out	7
`time_r_ce1`	-	-	-	out	1
`time_r_q1`	-	-	-	in	32

After the generated RTL has been instantiated in the ISE project and the designer has created the necessary custom logic to interface with the generated RTL code, the designer is ready to synthesize the design for the first time. When the designer synthesizes the design for the first time, Xilinx ISE might prompt the

[1] The Pico Computing board connects to the computer using a PCI Express interface. The Pico API supports two software models for communicating over this PCI Express link. The methods are a PicoBus model and a Streaming model. For this thesis the Streaming model was used because it provides a higher throughput than the PicoBus model.

```
parameter TIME_R_RAM_WIDTH = 32;
parameter TIME_R_RAM_ADDR_BITS = 7;

(* RAM_STYLE="{BLOCK |  BLOCK_POWER1 | BLOCK_POWER2}" *)
reg [TIME_R_RAM_WIDTH-1:0] TIME_R_RAM [(2**TIME_R_RAM_ADDR_BITS)
    -1:0];

always @(posedge ap_clk) begin
    if (time_r_ce0 || (time_r_initialize && s2i_valid && s2i_rdy))
        begin
        if (time_r_initialize && s2i_valid && s2i_rdy)
            TIME_R_RAM[time_r_write_address] <= s2i_data[31:0];
        time_r_q0 <= TIME_R_RAM[time_r_address0];
    end
    if (time_r_ce1)
        time_r_q1 <= TIME_R_RAM[time_r_address1];
end

always @(posedge ap_clk) begin
    if (ap_rst || reset_bram_write_address) begin
        time_r_write_address <= 0;
    end
    else if (time_r_initialize && s2i_valid && s2i_rdy) begin
        time_r_write_address <= time_r_write_address + 1;
    end
end

assign s2i_rdy = 1;
```

Listing 2: Block RAM Instantiated with Stream Interface in User Logic

designer to regenerate all of the IP Cores that Vivado HLS instantiated before the design can be synthesized. Once all of the IP Cores have been generated the design can be fully synthesized.

Now if the designer wants to simulate the entire design to verify that they have connected the generated RTL correctly into their ISE project, then the designer needs to create an RTL testbench. Unfortunately Vivado HLS cannot create an RTL testbench file from the C/C++ testbench file that was created to test the code in Vivado HLS. The designer will have to create their own testbench. To simplify writing this testbench to verify that the algorithm is working properly, the designer will be able to use the same CSV files that were created in MATLAB and used to test the algorithm in Vivado HLS. This will allow the designer to verify the design with real input data which will allow the designer to verify the entire design.

After the designer has been able to verify that the entire design works by simulating it, the designer is ready to implement the design and create a bitfile. Once the bitfile has been created, the designer needs to verify that the bitfile is working. To do this, the designer needs to create an interface between MATLAB and the FPGA. This will allow the designer to pass inputs from MATLAB to the FPGA and the designer will be able to verify the results in MATLAB by retrieving the solution from the FPGA. How this interface is created will be discussed in the next section.

Step 5: Create Interface Between MATLAB and FPGA

To incorporate this accelerated path planning algorithm into our path planning code, an interface had to be created between MATLAB and the FPGA. Pico Computing, the company that makes the FPGA board that was selected for this thesis, provided a C API that allows C/C++ code to interface with the FPGA. To create the interface between MATLAB and the FPGA, a MATLAB Executable (MEX) file was created that include calls to Pico's API. This allows the entire mission planning software that is written in MATLAB to program the FPGA, pass variables into the FPGA, and retrieve the solution from the FPGA. By creating this MEX file to be the interface between MATLAB and the FPGA no code in the original mission planning software had to be modified because the MEX function is called just like the original MATLAB function.

Recall from Step 2 earlier in this chapter, a MEX file is a compiled C/C++ file that includes a gateway function that allows MATLAB to execute the MEX function like a normal MATLAB function. The MEX file that the designer has to write to interface MATLAB to the FPGA will be similar to the testbench file that was written in Vivado HLS. Instead of reading inputs from CSV files, they will be passed in as arguments to the MEX function.

An example of what a MEX file looks like can be seen in Listing 3. The only two things that are required in a MEX file is including the `mex.h` header file on line 13 and including the `mexFunction` on line 86. The `mexFunction` is required to be the last function in the MEX file. The `mexFunction` is the function that adds the capabilities that allow MATLAB to interface with C/C++ code. The `mexFunction` is typically the function that the designer will use to verify that the data being passed to the MEX function is of the right data type and number of elements. No other function is required, but a good programming practice would be to create a separate function for the computational routine which can be seen on line 16. The computational function will be the function that will interface with the FPGA.

```
 1 /*****************************************************************
 2  * This example is based on the StreamLoopback128 reference
 3  * design that was release with the Pico Computer Framework
 4  * 5.0.6.1.
 5  *****************************************************************/
 6
 7 #include <stdio.h>
 8 #include <string.h>
 9 #include <stdio.h>
10 #include "picodrv.h"
11 #include "pico_errors.h"
12
13 #include <mex.h>
14
15 // Computational Function
16 int fpgaFunction(const char *bitFileName, uint32_t *a, uint32_t *b,
       uint32_t *sum)
17 {
18     int          err, stream;
19     int          numElements = 256;
20     uint32_t     input[1024], output[1024];
21     char         ibuf[1024];
22     PicoDrv      *pico;
23
24     // Program the FPGA
25     err = RunBitFile(bitFileName, &pico);
26     if (err < 0)
27     {
28         printf("RunBitFile Error: %s\n", PicoErrors_FullError(err,
               ibuf, sizeof(ibuf)));
29         delete pico;
30         return -1;
```

```
    }

    // Create Stream 1
    stream = pico->CreateStream(1);
    if (stream < 0)
    {
        printf("CreateStream Error: %i\n", stream);
        delete pico;
        return -1;
    }

    // Copy Input Data from MATLAB into Input Data Buffer
    for (int i = 0; i < numElements/16; i++)
    {
        input[4*i+3] = 0;
        input[4*i+2] = 0;
        input[4*i+1] = b[i];
        input[4*i]   = a[i];
    }

    // Write Input Data Buffer to Stream 1
    err = pico->WriteStream(stream, input, numElements);
    if (err != numElements)
    {
        printf("WriteStream Error: %i\n", err);
        delete pico;
        return -1;
    }

    // Clear Ouput Buffer
    memset(output, 0, sizeof(output));

    // Read Output Data from Stream 1
    err = pico->ReadStream(stream, output, numElements);
    if (err != numElements)
    {
        printf("ReadStream Error: %i\n", err);
        delete pico;
        return -1;
    }

    // Output Data to MATLAB
    for (int i=0; i < numElements/16; ++i)
    {
        sum[i] = output[4*i];
    }

    // Close Opened Stream and Destroy the PicoDrv Object
    pico->CloseStream(stream);
    delete pico;

    return 0;
}

```

```
// Gateway Function
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray*
    prhs[])
{
    char     *bitFileName;
    int       bitFileNameLength;
    int       status;
    uint32_t *a, *b, *sum;

    // Check for the Proper Number of Arguments
    if (nrhs != 3)
    {
        mexErrMsgTxt("Three Input Arguments Are Required");
    }
    if (nlhs != 1)
    {
        mexErrMsgTxt("One Output Argument Is Required");
    }

    // Get Bitstream File Name (Input Argument #0)
    bitFileNameLength = mxGetN(prhs[0])*sizeof(mxChar)+1;
    bitFileName = (char *)mxMalloc(bitFileNameLength);
    if (mxIsChar(prhs[0]))
    {
        if (mxGetString(prhs[0], bitFileName, bitFileNameLength) !=
            0)
        {
            mexErrMsgTxt("Input Argument 0 Error: Cound Not Convert
                String Data");
        }
    }
    else
    {
        mexWarnMsgTxt("Input Argument 0 Must be a String");
    }

    // Get Pointer to 'a' (Input Argument #1)
    if(mxGetM(prhs[1]) != 1)
    {
        mexErrMsgTxt("Input Argument 1 Must be a Row Vector.");
    }
    a = (uint32_t *)mxGetPr(prhs[1]);

    // Get Pointer to 'b' (Input Argument #2)
    if(mxGetM(prhs[2]) != 1)
    {
        mexErrMsgTxt("Input Argument 2 Must be a Row Vector.");
    }
    b = (uint32_t *)mxGetPr(prhs[2]);

    // Get Pointer to 'sum' (Output Argument #0)
    plhs[0] = mxCreateNumericMatrix(1,mxGetN(prhs[1]),mxUINT32_CLASS
        , mxREAL);
    sum = (uint32_t *)mxGetPr(plhs[0]);

    // Call the FPGA routine
    status = fpgaFunction(bitFileName, a, b, sum);
    printf("Status: %d\n\n", status);

    // Free memory
    mxFree(bitFileName);
}
```

Listing 3: Example MEX File with Pico API

With the algorithm now accelerated on hardware and the interface between MATLAB and the FPGA created, the designer is finished. In the next chapter, the details of how the design framework was applied to the genetic algorithm will be presented along with solutions to problems that were encountered while working through the design framework.

CHAPTER FOUR

Implementation and Results

This chapter presents the implementation details and results of converting a genetic algorithm using the design framework that was described in Chapter Three. The genetic algorithm was originally written in MATLAB, and with the use of MATLAB Coder Toolbox and Vivado HLS, the algorithm was converted to run on an FPGA. In addition to converting the algorithm, a MEX file was written that allowed MATLAB to use the accelerated algorithm running on the FPGA with the same calling syntax as the original genetic algorithm.

Implementation

Figure 8 shows the flowchart of the different steps involved in computing a genetic algorithm. The genetic algorithm that was implemented in this research is based on the work done in [2]. The MATLAB implementation of the genetic algorithm is based on the continuous genetic algorithm presented in [11] and the fitness function is based on the work done in [2].

For this thesis the entire genetic algorithm was implemented on the FPGA. This implementation was chosen even though the most time consuming section of the algorithm is computing the fitness function. With the possibility that the genetic algorithm will have to go through many generations before converging on a viable flight path, just implementing the fitness function on the FPGA could lead to the overhead in sending each generation across the PCI Express bus becoming the bottleneck of the algorithm. Implementing the entire genetic algorithm on the FPGA resulted in communicating with the FPGA only to initialize the inputs and to receive the final path that is computed by the genetic algorithm.

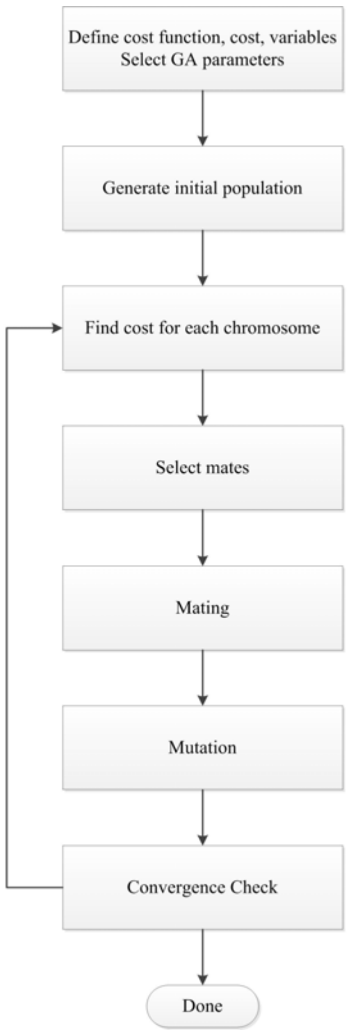
Define cost function, cost, variables
Select GA parameters
Generate initial population
Find cost for each chromosome
Select mates
Mating
Mutation
Convergence Check
Done

Figure 8: Genetic Algorithm Flowchart (Figure from [11])

While converting the entire genetic algorithm, some design modifications and tool configurations had to be changed to prevent errors in converting the algorithm. For example, when converting the algorithm with the MATLAB Coder Toolbox the designer might receive an error message that will suggest declaring a MATLAB function as an extrinsic function or calling the function with `feval`. The solution suggested by the MATLAB Coder Toolbox will not work in this design framework because each time an extrinsic function is called, control is passed back to MATLAB to execute the function. The goal of this design framework is to convert the algorithm to run entirely on an FPGA, so control cannot be returned to MATLAB to execute a function. To fix this error, the designer either needs to find a different MATLAB function that performs the desired task and can be converted by the MATLAB Coder Toolbox or the designer has to code the necessary functionality into their own code.

Another error message that MATLAB Coder Toolbox might produce is "Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [:?x1]. Please consider enabling dynamic memory to allow unbounded sizes." The solution is to set the maximum bounds for all arrays since dynamic memory is not supported in synthesizable hardware. There are two options to set the maximum bounds that will prevent this error message from appearing. The first option is to set the maximum bound to the exact value needed by the algorithm. The other option is to set the maximum bound to the absolute maximum size that will be needed in the worst case scenario. To simplify the complexity of the design while creating this framework, the exact value was used when initializing all of the arrays in this design.

To get the C/C++ code that is generated by the MATLAB Coder Toolbox to work with Vivado HLS, the designer will need to change some default parameters in the MATLAB Coder Toolbox. To change these settings, click on the "Build" tab and select "More Settings". In the "Project Settings" window, select the "Optimization" tab. In the "Optimization" tab, deselect "Use memcpy for vector assignment" and "Use memset to initialize floats and doubles to 0.0". With these options deselected, MATLAB will generate C/C++ code that Vivado HLS can process. If the designer receives the following error message in Vivado HLS "Memory copy is not supported unless used on bus interface (possible cause(s): non-static/non-constant local array with initialization)," then the designer needs to verify that they configured MATLAB Coder Toolbox properly.

With the C/C++ code converted into HDL code by Vivado HLS, it is time to implement the genetic algorithm on the FPGA. The genetic algorithm was implemented on the FPGA using the StreamLoopback32 reference design that was provided with the Pico Computing Framework. This design was used as a template that was modified to incorporate the generated HDL code. The ISE project hierarchy can be seen as a conceptual diagram in Figure 9. Everything in this hierarchy was provided by Pico Computing except for the “Generated RTL Code” and the “User Module” code which instantiates the generated RTL code. The Pico Architecture creates the interface for the computer to communicate with the FPGA over the PCI Express bus, provides the capabilities for the designer to use streams to communicate with the FPGA, and provides the interface to the different peripherals onboard (i.e., the DDR3 memory). The architecture provides a place for the designer to insert their own custom code in the “User Module”. The “User Module” file is defined in the `PicoDefines.v` file with the following definition: `` `define USER_MODULE_NAME UserModule``. The word `UserModule` would be replaced with the name of the designer’s user module.

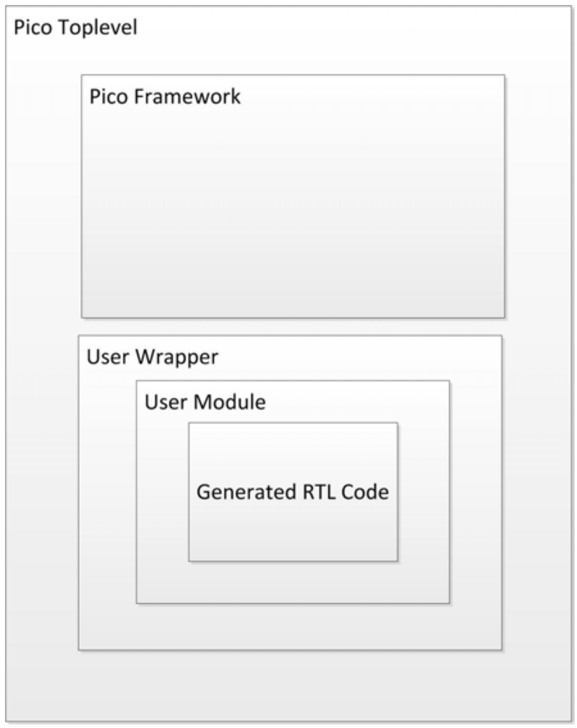

Figure 9: Pico Architecture with Generated RTL Conceptual Diagram

Recall from Chapter Three that when arrays are passed as input arguments into the algorithm or the algorithm generates outputs that are arrays, Vivado HLS implements them as single or dual-port RAMs. The designer can use the `set_directive_resource` command to specify which RAM resource is used onboard the FPGA or let Vivado HLS implement what it thinks will produce the most optimal design. Since one of the goals of this project was to create as simple as possible conversion from MATLAB code to HDL code, the option to let Vivado HLS decided how to implement the RAMs was selected. For this project the RAMs were setup as BRAMs on the FPGA. For the genetic algorithm there were four input arrays and four output arrays. To interface with each array from the computer program a stream was created to interface to each BRAM separately. The streams that were used in this design can be seen in Table 7. To interface the generated HDL code with the BRAM and to allow the computer program to initialize input arrays, the user module contains code similar to what is presented in Listing 2 from Chapter Three. Similar code was used to interface the generated HDL code with the BRAMs and the computer program with the BRAMs for the output arrays.

Table 7: Stream Assignments

Stream Number	Direction	Function
1	IN/OUT	Command Stream
2	IN	Initialize Time BRAM
3	IN	Initialize JARs Incenters BRAM
4	IN	Initialize JARs Incenters Length BRAM
5	IN	Initialize JARs Incenters Offset BRAM
6	OUT	Read Cost BRAM
7	OUT	Read Minimum Cost BRAM
8	OUT	Read Mean Cost BRAM
9	OUT	Read Population (Final Path) BRAM

With the generated RTL code incorporated into the Pico Architecture, the design was then simulated and synthesized. The results of the synthesized design and timing performance that was achieved will be discussed in the next section.

Results

The test case that was used as the algorithm was being developed and taken through the design framework is specified in Table 8. The test case represents the upper bound of how many Jamming Requirements would normally be passed to the genetic algorithm. This test case was chosen because it allowed the tools to be tested thoroughly with a complex design.

Table 8: Test Case Parameters

Parameter	Setting
Jamming Requirement Length	109
Potential Waypoints (JAR Incenters)	13423
Population Size	20
Mutation Rate	15%
Best Paths Kept	10
Maximum Iterations	2500

Initially when the genetic algorithm was being developed, the execution time of the MATLAB algorithm took longer than after the algorithm went through the design framework. Modifying the MATLAB code to avoid all warnings and making it work as a MEX file caused the algorithm to improve its speed. It turns out that after the genetic algorithm went through the entire framework that the hardware implementation was not faster than the MEX file or original MATLAB code. This can be attributable to a couple of different factors. First, the synthesized design was not able to meet timing. The resources used by the synthesized design are outlined in Table 9. The problem that the Xilinx Place and Route tools had with this design is not with the logic delay but with routing between the logic. It is possible that too much of the genetic algorithm was put on the hardware. FPGAs are good at accelerating algorithms that are heavy on computation versus algorithms with high I/O operations. Recall from Chapter Two that all of the FPGA implementations of the genetic algorithm had some portion of the genetic algorithm running on the computer. For this design, the entire genetic algorithm was put on the FPGA to reduce the amount of time data would be sent between the computer program and the FPGA. In addition, no effort was made to utilize the Vivado HLS features that parallelize and pipeline C/C++ thereby speeding up the computations. The generated C/C++ code had no directives added to it in Vivado HLS since the goal of this project was to evaluate how the well the tools convert the algorithm. The performance results of the genetic algorithm code at three different stages of the design framework can be seen in Table 10.

Table 9: FPGA Resource Utilization[*]

Slice Logic Utilization	Used	Available	Utilization
Slice Registers	51,859	301,440	17%
Slice LUTs	46,229	150,720	30%
Slice LUTs Used as Memory	2,612	58,400	4%
RAMB36E1/FIFO36E1s	201	416	48%
RAMB18E1/FIFO18E1s	39	832	4%
DSP48E1s	150	768	19%

[*] The resource utilization numbers are approximate numbers. The numbers are based on a synthesized design that did not meet timing.

Table 10: Genetic Algorithm Performance

Implementation	Execution Time	Performance Increase	Lines of Code
MATLAB Code	2.30018 sec	--	183
Compiled MEX File	1.40379 sec	38.97%	2663
Generated HDL Code[*]	6.39242 sec	-177.9%	194220

[*] The execution time is based on simulation results

The research for this thesis led to three major discoveries. First, design and validation can be performed when the algorithm is being developed in MATLAB. During this research, a case never presented itself where the results produced by the generated HDL code did not match the results produced by the MATLAB code. This finding will allow designers to test their algorithms in higher level languages and have confidence that the tools are accurately converting the algorithm. This finding does not mean that the code does not need to be tested at each stage, it just shows that the tools are mature enough to convert algorithms properly. Second, Vivado HLS is able to convert MATLAB generated C/C++ code into HDL code without having to modify the code. The only thing the design has to do is configure the MATLAB Coder Toolbox as outline in the Implementation section earlier in this chapter. Third, Vivado HLS does not generate the entire interface necessary for the designer to include the generated HDL code into their existing HDL hierarchy. As noted in Chapter Three, the designer is responsible for creating the interface between the generated HDL code and the BRAMs. This is true even if the designer uses a different I/O protocol.

Based on the discoveries that were found when creating and verifying this design framework, future work will be able to take code that is designed in high level MATLAB code and convert it to run on an FPGA. The next chapter will provide a summary of this research and provide some areas for future work.

CHAPTER FIVE
Conclusions

The design framework that was presented in this thesis can be used to accelerate not only path planning algorithms but any algorithm that is written in high level languages, like MATLAB or C/C++, and convert them to run on FPGAs. This design framework was used to convert a genetic algorithm for use in the current mission planning system. Though it turned out that this algorithm was not accelerated compared to the MATLAB implementation, the findings that were discovered when creating and verifying the design framework will provide a good foundation for future work as this project moves forward.

An area that needs to be researched further is the timing problem that was encountered with the HDL code that was generated by Vivado HLS. For simple designs that included basic math operations like addition, subtraction, multiplication, division, and computing the square root everything worked fine. The timing issues appeared once the entire genetic algorithm was ported to the FPGA. Future research will need to be performed to figure out how to prevent the timing issues with more complex designs.

The next area of research that needs to be performed is to work on the current genetic algorithm. The current genetic algorithm is not fully implemented to take into account all of the design requirements that the research team has decided need to be a part of the algorithm. One of the items that the current genetic algorithm does not take into account is whether or not the EA can see the enemy radar station. To incorporate this into the genetic algorithm, the FPGA will have to be loaded with Digital Terrain Elevation Data (DTED). Due to the amount of DTED data that might be needed, the DTED data will most likely be stored in the onboard DDR3 memory on the FPGA module. If this is the method that is selected, an interface has already been created to connect the generated HDL code with the onboard DDR3 memory. Additional MATLAB code and a MATLAB MEX file have been created to load the DDR3 memory with DTED data. The part that still needs to be developed is the addition of an optimized line-of-sight algorithm to the genetic algorithm. The MATLAB line-of-sight algorithm cannot be used because MATLAB Coder Toolbox cannot convert this MATLAB function. Another graduate student in this research team previously worked on creating and optimizing a line-of-sight algorithm written in MATLAB that runs faster than MATLAB's line-of-sight algorithm [12]. This algorithm needs to be tested with this design framework to determine if the tools can convert and optimize it and if the algorithm works efficiently on the FPGA.

Performing a feasibility check is another area that needs to be added to this algorithm. The feasibility check determines if the current flight path can actually be flown by the selected EA aircraft. In [2], Cocaud provides equations for determining the feasibility of a flight path for UAV aircraft. These equations may need to be modified to take into account the parameters of an EA aircraft but they should provide a good starting point. In addition to determining if the EA is capable of flying the current flight path will be ensuring that the EA stays within the JAR in order to provide protection to the PE.

The final area that needs to be researched is to evaluate different tools to convert MATLAB code to HDL code or MATLAB generated C/C++ code to HDL code. Instead of using MATLAB Coder Toolbox to convert the MATLAB code into C/C++ code for Vivado HLS to convert to HDL code, it may be possible to use the MATLAB HDL Coder Toolbox to convert the MATLAB code directly into HDL code. There are also many other C-to-HDL tools available which should be evaluated to see if they produce better results than Vivado HLS. Vivado HLS was chosen for this research since it is developed by Xilinx which is the vendor for the FPGA that was selected for this research and it is designed to optimize the design for the resources of the selected FPGA.

With these improvements to the genetic algorithm, the execution time of the MATLAB program may increase. This is when accelerating the algorithm on hardware may significantly improve the performance of the algorithm. FPGAs allow the designer to perform multiple different operations in parallel or the same operation in parallel to accelerate the speed of the entire algorithm. The line-of-sight calculation will be a prime candidate for instantiating multiple times on the FPGA because the calculation will be performed numerous times every iteration through the genetic algorithm. Having this in parallel increases the amount of resources used on the FPGA which might lead to the design needing to use partial reconfiguration. Partial reconfiguration is where portions of the FPGA get reprogrammed depending on what is currently needing to be calculated. The feasibility check is currently not being envisioned to run every iteration, so the resources that are needed by the feasibility check do not need to be allocated for the entire duration of the execution of the genetic algorithm. Instead those resources could be used by the genetic algorithm to accelerate more of the algorithm and then when the feasibility check needs to be performed, a portion of the FPGA will be programmed to perform the feasibility check calculation. When the feasibility check calculation is completed, that portion of the FPGA would be reprogrammed back to the genetic algorithm.

With the findings from this thesis, future algorithms will be able to be ported to the FPGA to accelerate the mission planning software. In addition to these findings, with the improvements outlined in this chapter implemented, the genetic algorithm may be able to operate near real-time as compared to the software only implementations. This could provide the Navy path planners an EA flight path that will provide adequate protection to the PE to fulfill the EOB mission requirements. If the EOB mission requirement change, the Navy path planners will be able to rerun the mission software and get a new EA flight path in a relatively short amount of time.

BIBLIOGRAPHY

[1] J. Dark, J. Buscemi, and S. Burkholder, "Aircrew Aid to Assess Jam Effectiveness," Patent US007 427 947B1, Sep. 23, 2008.

[2] C. Cocaud, "Autonomous Tasks Allocation and Path Generation of UAV's," Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, Jul. 2006.

[3] F. C. J. Allaire, M. Tarbouchi, G. Labont, and G. Fusina, "FPGA Implementation of Genetic Algorithm for UAV Real-Time Path Planning," *Journal of Intelligent and Robotic Systems*, vol. 54, pp. 495–510, Jul. 2008.

[4] J. Kok, L. Gonzalez, R. Walker, T. Gurnett, and N. Kelson, "A Synthesizable Hardware Evolutionary Algorithm Design for Unmanned Aerial System Real-Time Path Planning," in *Proceedings of the 2010 Australasian Conference on Robotics & Automation*, 2010.

[5] (2012, Oct.) M-503. Pico Computing, Inc. [Online]. Available: http://picocomputing.com/m-series/m-503/

[6] (2012, Aug.) EX-500 EX-Series Backplane. Pico Computing, Inc. [Online]. Available: http://picocomputing.com/brochures/EX-500.pdf

[7] (2012, Jan.) Virtex-6 Family Overview. Xilinx, Inc. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[8] *MATLAB Coder User's Guide*, MATLAB 7.12, MathWorks, Inc., Apr. 2011.

[9] (2012, Aug.) Introduction to High-Level Synthesis with Vivado HLS Standalone - Vivado 2012.2 Version. Xilinx, Inc. [Online]. Available: www.xilinx.com/university/workshops/high-level-synthesis-flow/index.htm

[10] (2012, Jul.) Vivado Design Suite User Guide: High-Level Synthesis. Xilinx, Inc. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[11] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, 2nd ed. John Wiley & Sons, Inc., 2004.

[12] J. Perry, "Line of Sight Algorithm," Jan. 2011, Baylor University - unpublished.