COMPUTATIONAL ACCELERATION FOR NEXT GENERATION CHEMICAL STANDOFF SENSORS USING FPGAS

A Thesis

Submitted to

the Temple University Graduate Board

In Partial Fulfillment

Of the Requirements for the Degree

MASTER OF SCIENCE

in ELECTRICAL ENGINEERING

John Ruddy

May, 2012

Thesis Approval(s):

Dennis Silage, PhD., Thesis Adviser, Electrical and Computer Engineering

John Helferty, PhD, Electrical and Computer Engineering

Chen-Huan Chiang, PhD, Electrical and Computer Engineering

Abstract

This research provides the real-time computational resource for three dimensional tomographic chemical threat mapping using mobile hyperspectral sensors from sparse input data. The crucial calculation limiting real-time execution of the algorithm is the determination of the projection matrix using the algebraic reconstruction technique (ART). The computation utilizes the inherent parallel nature of ART with an implementation of the algorithm on a field programmable gate array. The MATLAB Fixed-Point Toolbox is used to determine the optimal fixed-point data types in the conversion from the original floating-point algorithm. The computation is then implemented using the Xilinx System Generator, which generates a hardware description language representation from a block diagram design.

Table of Contents

List of Figures

1 Introduction

1.1 Motivation

The goal of this research is to provide the computational resources for a real-time three dimensional (3D) mapping system. The mapping system uses data from multiple moving hyper-spectral sensors to make 3D chemical threat cloud maps. A computed tomography method, based on a modified version of the algebraic reconstruction technique (ART), is used to produce a 3D concentration map from the limited sensor data [4]. Figure 1.1 shows the sensor data before and after processing with the algorithm. Computing this concentration map requires the calculation of a projection matrix, which maps the sparse sensor measurement vectors to the grid space. This calculation is computationally intensive, and can currently require 30 minutes to compute using a multithreaded version of the calculation developed by MESH, Inc. that utilizes an 8 core CPU. In order to develop a real-time mapping system, a fast, practical alternative is needed.

The projection matrix calculation is an iterative one, where the calculation for each element of the matrix is independent of the others, allowing for a parallel determination of the matrix. In order to take advantage of this characteristic, parallel computation platforms—graphical processing units (GPU) and field programmable gate arrays (FPGA)—are considered [3]. The GPU applications researched focused on utilizing single instruction-multiple data (SIMD) operations. It also was appar-

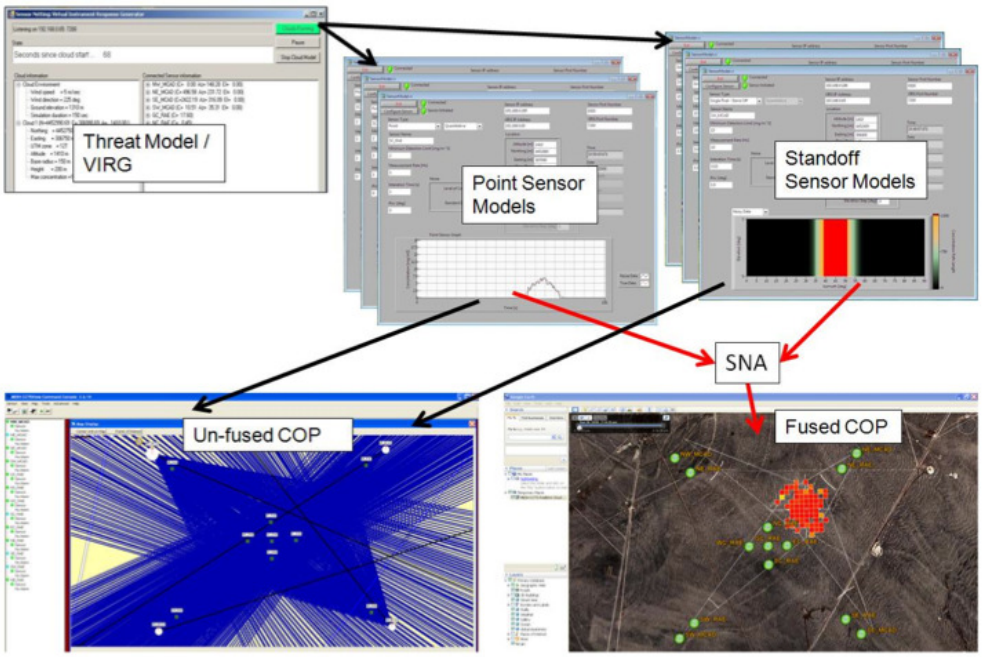

Figure 1.1: Unfused vs. fused common operating picture (COP). The point sensor models and the standoff sensor models are fused using the sensor netting algorithm described.

ent that algorithms needed to be tailored in order to perform correctly on the GPU, adding a possibility of error in the output, as well as increasing development time. The fine grained FPGA allows for massively parallel complex computational logic and arithmetic to be performed, making it suitable for this work.

1.2 Research Objectives

The objective of this research is to develop a fast projection matrix calculation using FPGA technology for use in a real-time hyper-spectral 3D mapping system and rapid tomography simulations. The algorithm for calculating the matrix must be converted

from its native floating point C code to a fixed point representation for simulations. These simulations will help determine the optimal fixed point word length and number of integer and fractional bits that will reproduce the floating point results. Once simulations are completed and optimal values are found, the algorithm will be implemented on an FPGA. This will be done using Xilinx System Generator and Verilog HDL. The final product will be an HDL IP core that can be instantiated to calculate a single projection matrix element.

1.3 Organization of Thesis

This thesis document is organized as follows. A background is given in order to lay the foundation for this work. The projection matrix and its relation to the 3D tomographic calculation is described, followed by an outline of key features of the FPGA architecture. Then, fixed point arithmetic is covered, in order to give the mathematical background for the fixed point simulations. Following the background section, the simulation work is described. The simulation procedures and results are examined. After reviewing the simulation, the implementation details are given. Finally, there is a conclusion.

2 Background

2.1 Projection Matrix Calculation

The projection matrix calculation is part of the sensor amalgamation mapping by Mesh, Inc (SAMMI, www.meshoxford.com), a modified algebraic reconstruction technique used in computed tomography (CT). This method is designed to interpret path integrated sensor measurements from a network of both standoff and point sensors and output a concentration map, as shown in Figure 2.1. ART was chosen since it works best for noisy, sparsely sampled data from a limited number of sensors [4]. The modified CT technique developed by MESH is described in this section with a focus on the projection matrix calculation.

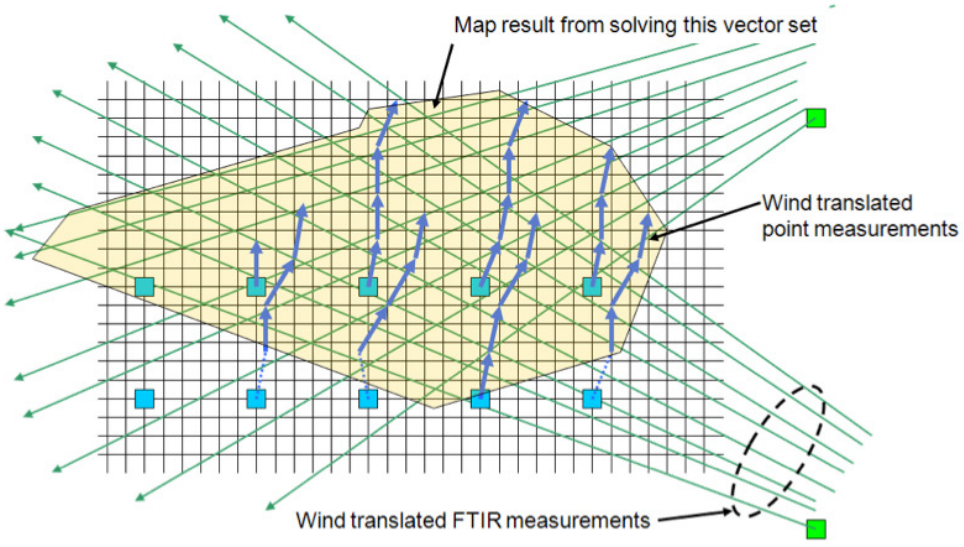

Figure 2.1: Standoff sensor measurements (long diagonal lines) and point sensor measurements (short upward lines) are processed to create the cloud map shown.

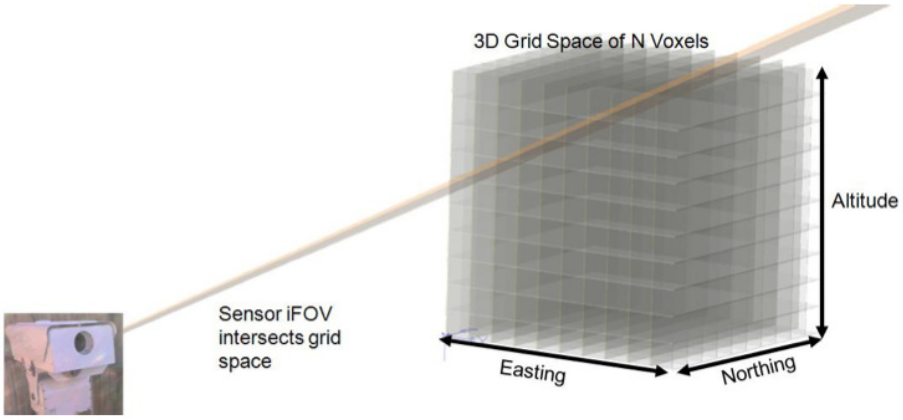

Figure 2.2: Standoff sensor path integrated measurement is shown passing through the grid space.

The three dimensional space that is the target of the sensors is made up of voxels, which are volumetric pixels, and is called the grid space, as shown in Figure 2.2 with a sensor. Two types of hyper-spectral sensors are used, point and standoff sensors. Point sensors measure locally; they need to come into direct contact with the agent being measured. The standoff sensors measure threats at a distance as long as there is a clear line-of-sight. The standoff sensor measurements are path integrated concentration lengths (CL) and are considered vectors. The point sensor measurements, on the other hand, need to be converted to concentration lengths. This is done by multiplying the sensor integration time by the wind velocity, resulting in a measured strip of air that is a path integrated concentration length vector, as shown in Figure 2.3 [8].

The ART method's objective is to produce an object vector, which contains the

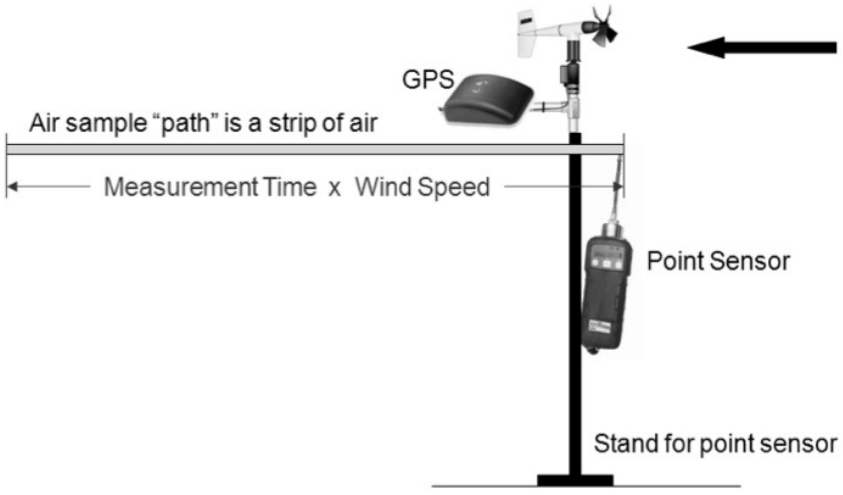

Figure 2.3: Conversion of point sensor measurement to a path integrated concentration length (CL) line segment vectors using wind velocity and integration time.

concentration measurements for each voxel, used to create the map, that can reproduce the inputted CL measurements. This is given below in Equation 2.1:

$$\begin{pmatrix} \uparrow \\ \text{CL "Vector"} \\ \text{P x 1} \\ \downarrow \end{pmatrix} = \begin{pmatrix} \uparrow \\ \leftarrow \quad \text{Projection Matrix} \quad \rightarrow \\ \text{P x N} \\ \downarrow \end{pmatrix} \begin{pmatrix} \uparrow \\ \text{Object Vector} \\ \text{N x 1} \\ \downarrow \end{pmatrix} \tag{2.1}$$

The units for the CL vector, projection matrix, and the object vector are, respectively, units of concentration-distance, units of distance, and units of concentration. The object vector is solved for iteratively, adjusting the part that is affected by the current CL measurement on each iteration, as given by Equation 2.2.

$$\begin{pmatrix} \uparrow \\ \text{Object Vector} \\ \text{N x 1} \\ \downarrow \end{pmatrix}_{q+1} = \begin{pmatrix} \uparrow \\ \text{Object Vector} \\ \text{N x 1} \\ \downarrow \end{pmatrix}_{q} + Adjust_i \begin{pmatrix} \leftarrow \quad \text{Proj. Matrix, row i} \quad \rightarrow \\ \text{1 x N} \end{pmatrix}^T \tag{2.2}$$

where the adjustment is equal to the CL measurement minus the current CL estimate divided by the current projection matrix row squared, as shown in Equation 2.3.

$$Adjust_i = \frac{CL_i - \begin{pmatrix} \leftarrow \quad \text{Projection Matrix, row i} \quad \rightarrow \\ \text{1 x N} \end{pmatrix} \begin{pmatrix} \uparrow \\ \text{Object Vector} \\ \text{N x 1} \\ \downarrow \end{pmatrix}}{\left| \begin{pmatrix} \leftarrow \quad \text{Projection Matrix, row i} \quad \rightarrow \\ \text{1 x N} \end{pmatrix} \right|^2} \tag{2.3}$$

The projection matrix calculation is the focus of this work. The elements of the matrix W_{ij} are calculated by integrating a basis function in three dimensions over the field of view FOV_i of CL_i. The integration is normalized by dividing the average cross sectional area of the FOV_i spot size at the distance to the center of the voxel. The index i is from 0 to P-1 and index j is from 0 to N-1, where P is the number of CL vectors and N is the number of voxels in the grid space as given in Equation 2.4.

$$\text{Proj. Matrix} = \begin{pmatrix} \frac{1}{A_0} \int\limits_{iFOV_0} Basis(S - Voxel_0)dS & \frac{1}{A_0} \int\limits_{iFOV_0} Basis(S - Voxel_1)dS & \dots \\ \\ \dots & \frac{1}{A_{P-1}} \int\limits_{iFOV_{P-1}} Basis(S - Voxel_{N-1})dS \end{pmatrix} \tag{2.4}$$

The basis function, which was chosen by MESH, is a non-linear, piecewise function of the voxel size, Δ, extending ± 2 voxels in all three dimensions. The arguments of the basis function is the difference between a point, S, on the line-of-sight (LOS) path of the FOV_i of CL_i and the voxel center point. The definition of the basis function is shown in Equations 2.5 and 2.6 and the plot of the basis function for one dimension is shown in Figure 2.4.

$$b1(\Delta, d) := \frac{(2 \cdot \Delta - |d|)^3 - 4 \cdot (\Delta - |d|)^3}{4 \cdot \Delta^3} \qquad b2(\Delta, d) := \frac{(2 \cdot \Delta - |d|)^3}{4 \cdot \Delta^3} \tag{2.5}$$

$$bf(\Delta, d) := \begin{cases} b \leftarrow b1(\Delta, d) & if \quad |d| < \Delta \\ b \leftarrow b2(\Delta, d) & if \quad \Delta \leq |d| \leq 2 \cdot \Delta \\ b \leftarrow 0 & if \quad |d| > 2 \cdot \Delta \end{cases} \tag{2.6}$$

The line of sight is broken up into sub-LOSes for the integration of the projection matrix elements, performed as a discrete summation. The LOS requires a unit direction vector d_u,

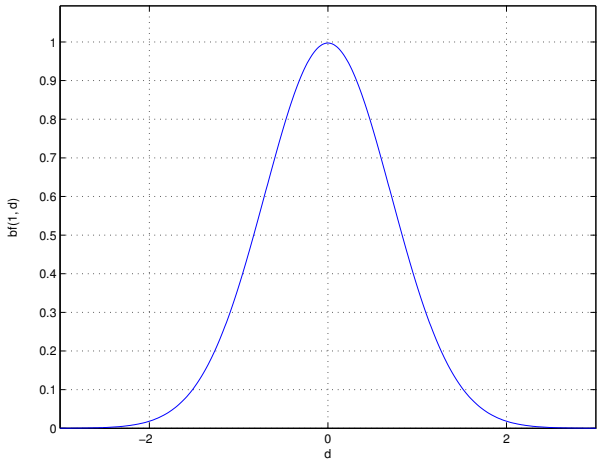

Figure 2.4: One Dimensional Basis Function Plot

given by Equation 2.7.

$$\begin{pmatrix} sin(az) \\ cos(az) \\ tan(el) \end{pmatrix} \cdot \frac{1}{(sin(az)^2 + cos(az)^2 + tan(el)^2)^{0.5}} \tag{2.7}$$

The point S along the LOS at a distance L for the FOV_i of CL_i in three dimensions is given by Equation 2.8.

$$S = S_o + d_u \cdot L \tag{2.8}$$

The summation for each element of the projection matrix is for k discrete steps, where k is from 0 to the integer ratio of the range to a step size. The elements are averaged for all sub-LOSes by dividing by the number of steps. The output is the elements W_{ij} of the projection matrix [12].

2.2 FPGA Architecture

2.2.1 Introduction

The coarse-grained field programmable gate array (FPGA) lends itself well to calculation of the projection matrix, of which each element is computed independently, allowing for parallel execution. An FPGA is a device containing reconfigurable combinational logic blocks (CLBs) and interconnects, which can be configured to perform complex combinational logic computations. The architecture for these is shown in Figure 2.5. They also contain block RAM, memory controllers, and hardware multipliers. The inherent parallelism of the logic blocks allows the FPGA a higher throughput at much lower clock frequencies than a standard multicore processor. This architecture facilitates the type of high speed, parallel computations needed to calculate the projection matrix.

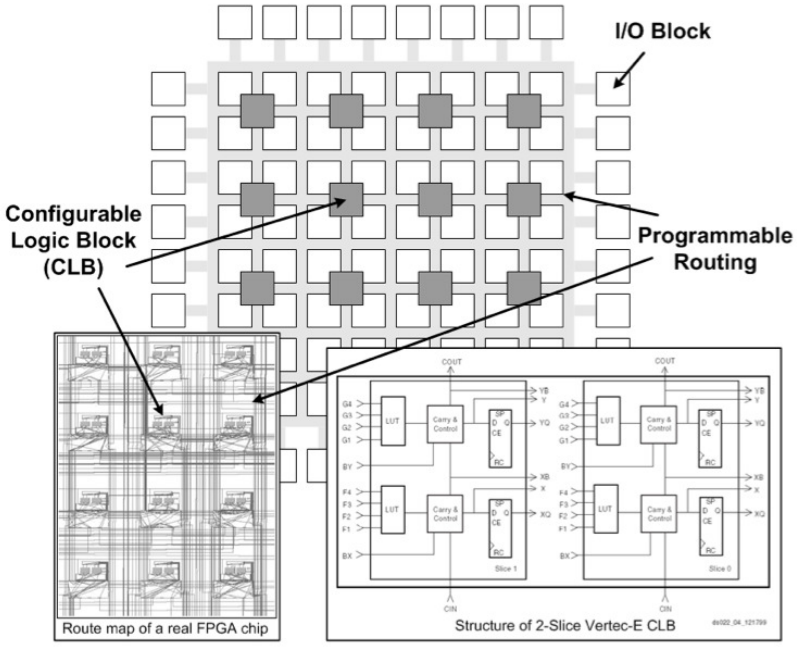

Figure 2.5: Configurable Logic Block and Routing Matrix Diagram

2.2.2 Development Methodologies

FPGAs are generally designed using a hardware description language (HDL), the most common being Verilog and VHDL. Current alternatives to this development cycle are the use of Mathworks MATLAB/Simulink with the Xilinx System Generator, as well as many higher level languages that attempt to raise the level of abstraction by combining HDL with a high level language such as C. The higher level languages can allow for a faster development cycle in some situations, but here it obfuscates the underlying hardware implementation, which may require precise design in order to calculate the projection matrix accurately.

The MATLAB/Simulink design methodology allows for a graphical modeling environment, using blocks and interconnections to build a functional block diagram for a system. This is automatically translated to the FPGA hardware using the Xilinx System Generator, which includes a Simulink library consisting of arithmetic, logic, and DSP blocks. The library of Xilinx blocks is bit and cycle-true to the FPGA implementation, meaning the simulation will behave exactly as the implemented logic. A previous design utilizing the Xilinx System Generator blockset is a Turbo Coding implementation on an FPGA. System Generator was used to build the entire FPGA model and generate a bitstream used to program an FPGA [5].

The design of an FPGA architecture with HDL uses the Xilinx Integrated Development Environment (IDE), which provides several Xilinx IP cores, termed LogiCore Blocks. The LogiCore blocks provide cores that can be instantiated using HDL, which range in application from basic logic to DSP arithmetic, bus interfacing, and communications constructs.

An example project using this type of development is an image steganography FPGA

implementation optimizing the execution time of a parallel image sharing algorithm [6]. The hardware implementation was performed in five stages using hardware multipliers and block RAM to compute the matrices. The speedup from a standard 3.2 GHz clock microprocessor to the FPGA implementation was approximately 320 times, without including the slower 100 MHz clock speed of the FPGA.

Another option is use both System Generator and HDL together. System generator has an option to generate HDL from a system model. This can then be instantiated in a Xilinx ISE project, where custom HDL can be written to provide the communication between the model, the inputs and outputs, and any other custom logic. The approach used in this research is to generate the projection matrix calculation module using System Generator, which will allow it to be instantiated in an HDL project, where custom logic can be written to allow communication to a computer or microprocessor, which will be running the remaining segment of the sensor netting algorithm.

2.2.3 Data Representation

In order to limit hardware resources and increase speed, most signal processing and algorithm accelerating FPGA implementations utilize fixed word-length arithmetic. This is especially important in real-time, mobile applications, due to the need for speed and power efficiency. Floating point arithmetic is not easily implemented on FPGAs, creating the need for IP cores for such computation. Floating point arithmetic uses more power, time, and is much more computationally expensive than fixed point arithmetic. Fixed point arithmetic can usually be modeled to give results with similar accuracy and range to that of floating point. In fact, many algorithms and applications use IEEE single or double precision float-

ing point out of convenience, since almost all common CPUs have floating point hardware units [11].

This work uses fixed point arithmetic in order to utilize the hardware resources efficiently, increase computation speed and be compatible with the Xilinx System Generator blockset. An example of fixed point arithmetic's resource efficiency is the DSP48 block, which provides a way to directly instantiate the DSP48A1 slices on the FPGA. These slices contain an $18x18$ multiplier, an adder, and an accumulator and lend themselves to many DSP and arithmetic applications.

Using fixed point, with an optimized word-length using simulation, two 18 bit numbers can be multiplied. The output of the DSP48A1 slice is 48 bits wide in order to accommodate accumulation after multiplications. If more bits are needed to represent the data, multiple DSP48 blocks can be used for a single multiplication. In contrast, single precision floating point uses two DSP48A1 slices, while double precision requires ten slices. Thus, the reduced word-length of fixed point representation from floating point format can greatly reduce resource usage.

The caveat to using fixed point representation is the need to determine optimal word-length and precision in order to maintain the accuracy of the original algorithm. The direct relationship between word-length and logic utilization provides motivation for minimizing the word-length. The placement of the radix point determines the precision and is a trade-off between range and quantization noise. Determination of these optimal properties is generally done using simulation in some type of procedural language, such as C. Recently, MATLAB has become popular for simulations of FPGA designs, due to the release of the

Fixed Point Toolbox and the ability to rapidly iterate the algorithm and re-simulate [2]. The design flow for this type of development is shown in Figure 2.6.

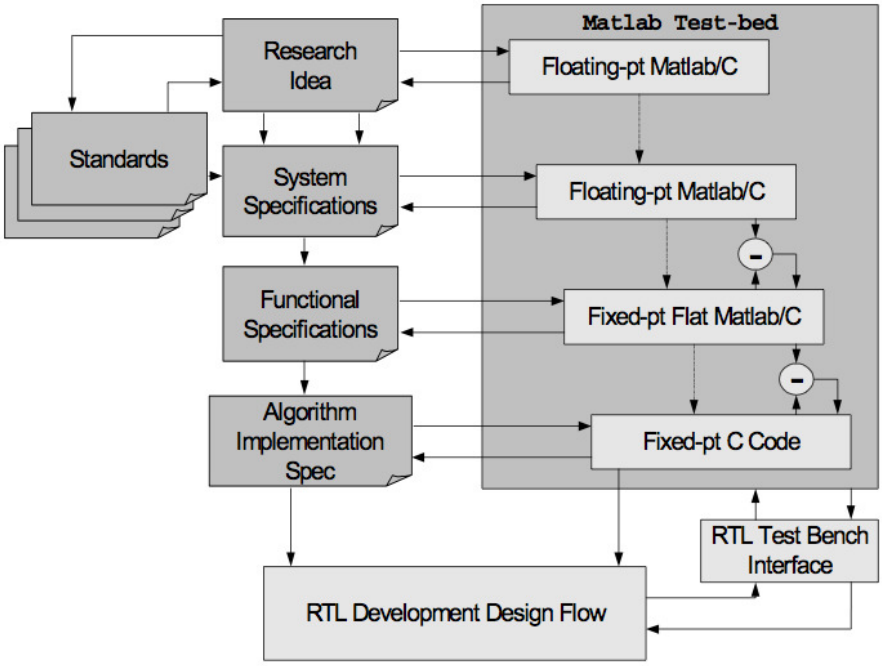

Figure 2.6: Algorithm Development in a MATLAB based design flow.

The toolbox presents many fixed point functions and rules in order to craft the simulation to exact specifications, and MATLAB itself can be used to analyze errors between floating point and fixed point calculations. The Fixed Point Toolbox and MATLAB FPGA development cycle is further described in Section 3. The next section introduces fixed point arithmetic as a background for the needed computation algorithms for the fixed point simulation.

2.3 Fixed Point Arithmetic

2.3.1 Introduction

In this section a brief introduction to fixed point arithmetic and related concepts are given. There are several different types of binary number representations, including signed and unsigned integer and fixed point representations [15]. Signed and unsigned fixed point representations are described first, since the integer representations are just cases of fixed point where there is no fractional bits, leaving only the integer bits. Before describing the representations, a few concepts are defined. The resolution of a fixed point number is the smallest non-zero number able to be represented. The range is the difference between the highest and lowest values representable, where the lowest value is negative for the signed representations. Accuracy is defined as the maximum difference between a real number and its representation, and is equal to the resolution divide by two.

Unsigned fixed point numbers are represented by an N-bit binary word with a fixed number of integer and fractional bits. The notation used here is $U(a, b)$, where U means the number is unsigned with a integer bits and b fractional bits, and, in the unsigned case, $a = N - b$. Each bit, starting from the right and beginning at $n = 0$, has a weight of 2^{n-b}, with the decimal point being located between the bits weighted 2^1 and 2^0. The value of a number x, with N-bits, is then given by Equation 2.9,

$$x = \frac{1}{2^b} \sum_{n=0}^{N-1} 2^n x_n \tag{2.9}$$

where x_n is bit n of x. The resolution is equal to 2^{-b} and the range of representable numbers is from 0 to $2^a - 2^{-b}$. For example, an eight bit unsigned fixed point number with $U(6,2)$ representation has resolution $2^{-2} = 1/4$ and range 0 to $2^6 - 2^{-2} = 64 - 1/4 = 63.75$. Its form is given by Equation 2.10.

$$a_5 a_4 a_3 a_2 a_1 a_0 . b_{-1} b_{-2} \tag{2.10}$$

The unsigned integer representation is simply the case where an N-bit number has representation $U(N,0)$ with a resolution of 1 and a range of 0 to $2^N - 1$.

Signed fixed point numbers are represented using two's complement. In order to obtain the two's complement of a number, every bit is inverted and then one is added to the resulting number. In a signed representation, the leftmost bit is considered the sign bit, and is equal to 1 for negative numbers and 0 for positive numbers. An N-bit binary word in signed fixed point representation uses notation $A(a,b)$, where A means a signed number with a integer bits and b fractional bits and $a = N - b - 1$. The precision of the magnitude, a, is reduced by one in the signed case since the leading bit is used to denote the sign and does not contribute to the magnitude of the number. The value of a number x, with N-bits, is given by Equation 2.11,

$$x = \frac{1}{2^b}\left[-2^{N-1}x_{N-1} + \sum_{0}^{N-2} 2^n x_n\right] \tag{2.11}$$

where x_n is bit n of x. The resolution is still equal to 2^{-b}, and the range is given by Equation

2.12.

$$-2^a \leq x \leq 2^a - \frac{1}{2^b} \tag{2.12}$$

For example, an eight bit unsigned fixed point number with representation $A(5,2)$ has resolution $2^{-2} = 1/4$ and range given by Equation 2.13.

$$\begin{aligned} -2^5 \leq x \leq 2^5 - 2^{-2} \\ -32 \leq x \leq 31.75 \end{aligned} \tag{2.13}$$

The work proposed here will utilize the signed, two's complement fixed point representation. The following Section provides a brief summary of the basic arithmetic in this representation[15].

2.3.2 Arithmetic

Signed Addition. In order to add two binary numbers, they need to have or be converted to the same representation. For example, $A(a,b) + A(c,d)$ can be added if $a = c$ and $b = d$. If the representations are different, the numbers will need to be shifted in order to align the radix point. The operand with the larger number of fractional bits needs to be right shifted by the difference $b - d$, where b is the larger number of fractional bits, and sign extended. The sum of the result requires $c + 1$ bits to store any overflow. The resulting representation is $A(c + 1, d)$.

Signed Multiplication. The result of full precision signed multiplication is given by Equation 2.14.

$$A(a,b) \times A(c,d) = A(a+c+1, b+d) \tag{2.14}$$

The hardware available for the computation may not allow for full precision, in which case the result of the multiplication needs to be truncated or rounded. This has effects on the accuracy of the results, which will be discussed in the next section.

A special case of multiplication is when one of the operands is a constant value that is also a power of two. This case allows the multiplication to be performed by simply shifting the radix point to the right by a number of bits equal to the exponential value of the power of two constant.

Signed Division. The result of signed division is given by Equation 2.15

$$\frac{A(a_1,b_1)}{A(a_2,b_2)} = A(a_1+b_1+1, a_2+b_2) \tag{2.15}$$

The division result may also need truncation or rounding. The same special case exists for division where one of the operands is a constant power of two value. The radix point is shifted left by the number of bits equal to the exponential value of the power of two.

Another special case for division allows for logic utilization and computation time reduction. If the divisor of a division operation is constant throughout the algorithm, then the operation can be replaced with a multiplication of the reciprocal of the constant value. Since a division operation usually requires more hardware and has a higher latency than a multiplication operation, a significant amount of resources can be saved by using this

technique wherever possible.

2.3.3 Trigonometric Functions

Fixed point trigonometric functions represent a challenging computation that is usually performed using an approximation. Two types of approximations used are lookup tables (LUTs) and the Coordinate Rotation Digital Computer (CORDIC) algorithm [1]. Lookup tables allow the use of block RAM to store the outputs of trigonometric functions. A larger amount of block RAM can be used to store more precise outputs. This is a tradeoff decision between area utilization and precision.

The CORDIC algorithm uses iterative vector rotation to compute trigonometric functions, a computation that utilizes only shift and add operations, requiring less hardware but more clock cycles. The CORDIC algorithm requires a tradeoff to be made between clock cycles and precision. The number of iterations used determines the precision of the result.

2.3.4 Finite Wordlength Effects

Due to the finite register size allowed for fixed point data and operations, some form of quantization is required to store the result into a smaller register. There are two types of quantization that are generally used in fixed point arithmetic – rounding and truncation. Rounding results in the closest quantized value to the unrounded result. This quantized value differs by no more than half of the quantization step size. Truncation simply drops the lower bits, resulting in a quantized value that is always less than the initial result. The tradeoff between rounding and truncation is logic utilization. Truncation has no hardware

cost while rounding requires the use of a comparator to determine the closest quantized value.

The need to quantize the data results in some amount of error at every quantization in the algorithm. The error is reduced by optimizing the fixed point data types and operations used. Choosing the proper wordlength and radix point can allow for a solution with the desired accuracy and optimized hardware utilization. Chapter 3 details the simulation process used to optimize the fixed point algorithm design.

3 Simulation

3.1 Introduction

The intent of simulation is to find an appropriate fixed point word length and radix point so that the FPGA results agree with the floating point algorithm results with as small an error as possible. Because of this, most of the focus of simulation research is on the difference between the floating point output and the fixed point output, while overlooking the error between the fixed point model and the FPGA, caused by differences in scaling or bit width due to limited bit width options in C, as well as different operation rules. Previous research shows that the fixed point design process, shown in Figure 3.1 in three stages, can model FPGA technologies better when the simulation portion is performed in MATLAB, instead of C [9].

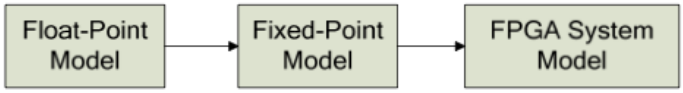

Figure 3.1: FPGA Fixed Point Design Process

As the research shows, when using C, there are many shortcomings when it comes to an exact representation of the FPGA performance. The data-type bit widths when using C are powers of 2, while the bit width on the FPGA is completely customizable. There is also no data type in C that represents fixed point, leaving the tracking of the radix point to the programmer. There is also a lack of operating rules for fixed point, such as type casting, overflow modes, and rounding modes.

MATLAB is already a proven simulation platform for many types of systems. The Fixed-Point Toolbox allows FPGAs to be simulated just as any other system. This toolbox offers fixed point data types and arithmetic in MATLAB, which can be compiled and run at the same speed as a C program. Some of the key features, as noted by The Mathworks, are [10]:

- Fixed point data types with up to 65,535 bit word lengths
- Global and local settings for operating and arithmetic rules
- Fixed point data types can also be used in Simulink
- Data logging and tools for floating point conversion are included

A short introduction on the use of the toolbox is given here. Fixed point numbers are represented using an object called an Fi object. This object holds the information about the number, such as the data type, whether it is signed or unsigned, word length and the number of fractional bits. It is defined in MATLAB as:

$$fp_obj = fi(v, s, w, f) \tag{3.1}$$

where v is the value, s is binary, representing signed or unsigned, w is the word length, and f is the number of fractional bits.

For example $fi(pi, 1, 16, 12)$ has value pi, with 3 integer bits and 12 fractional bits, and a leading sign bit. Note that the word length includes the integer bits, fractional bits, and the sign bit. Using these fixed point objects, the toolbox has constructs that allow for basic

arithmetic, statistical functions, square root and convolution, and user selectable settings governing all arithmetic functions . These settings include cast before sum, overflow mode, sum/product mode, sum/product word length, sum/product, fraction length, and rounding mode. According to the previous research, there are six general steps to MATLAB fixed point simulation[9]:

1. Design and verify the floating point model
2. Segment the algorithm into individual computations
3. Perform a range analysis of each variable and operation
4. Design the fixed point model and simulation
5. Perform signal to quantization noise ratio (SQNR) analysis and iterate step 4
6. Optimize the word length by comparing errors between fixed and floating point and adjusting

The previous research on the Fixed Point Toolbox shows that if the proper settings are chosen for the operating modes, the fixed point to FPGA errors can be minimized to close to zero [9]. These are global settings that can be set in MATLAB to govern all fixed point arithmetic. These are known as $fimath$ properties, and are set as follows:

1. The castBeforeSum attribute should be set to 1, enabling typecasting of input values for addition and subtracton
2. Rounding mode is set to 'floor'

3. Product/Sum Mode is set to 'SpecifyPrecision' so that the output scaling of operations can be specified

4. Overflow mode is set to 'wrap'

3.2 Floating Point Verification

The algorithm used for the projection matrix calculation was originally implemented by MESH, Inc. in floating point C code. The conversion from floating point C to floating point MATLAB code required changing the syntax and functions to be MATLAB compatible. All of the MATLAB functions in the projection matrix calculation are algorithmically identical to the C code implementation, and both utilize double precision variables. Since MATLAB contains its own versions of libraries with similar functions to the included C header libraries, as well as uses double precision floating point by default, the conversion is very straightforward.

In order to test the floating point model, a testbench is written to loop through the possible input values. These input values should be the expected values of the inputs likely to be seen during field use in order to give an accurate comparison to the fixed point implementation. Testing for both floating and fixed point implementations should use the same testbench inputs, but the fixed point inputs are truncated versions of the floating point inputs.

For this work, an actual data set from multiple sensors was used as a testbench input. This data set was used for the calculation of an 841-element projection matrix from 1,146

sensor measurements. Since each sensor measurement must be taken into account for every grid voxel of the projection matrix, the data set provides $841 \times 1146 = 963786$ different inputs for the testbench. After verifying that the floating point MATLAB output matches the C code output, the input data sets and outputs are saved for later comparison to the fixed point implementation.

3.3 Conversion to Fixed Point

3.3.1 Algorithm Segmentation

In order to properly analyze the range of the variables, the algorithm needs to be broken down into individual operations to allow for analysis of the intermediate results. Normally, when an algorithm is coded, many operations are grouped into a single statement, as shown in the following MATLAB expression:

$$\text{basisDomain} = 4 * sqrt(2 * vxSizNE^2 + vxSizA^2); \tag{3.2}$$

This code statement contains multiple squaring and multiplication operations, an addition operation and a square root operation. Analyzing the inputs and output to this statement will allow a determination of the range that these variables take on, but the characteristics of the intermediate data between operations is unknown. To analyze the intermediate results, the statement is broken up into the individual operations that comprise it, as shown in Listing 3.1.

Listing 3.1 Broken up Expression

```
NEsq = vxSizNE^2;
Asq = vxSizA^2;
sqs = 2*NEsq + Asq;
root = sqrt(sqs);
basisDomain = 4*root;
```

Here, the result of each operation is stored into a separate variable. Each of these variables can then be analyzed to determine the word length and precision required for the intermediate results.

This procedure is performed for every mathematical expression in the algorithm, and each expression's individual operations are grouped together. This will allow every expression and its intermediate values to be examined separately from the rest, simplifying the optimization process. Once every operation is rewritten to save the intermediate results, range analysis can be performed each variable.

3.3.2 Dynamic Range Analysis

When converting to fixed-point, an understanding of the range of the intermediate and output variables is required to choose starting values for the word length and radix point. The ideal word length would allow for the full range of values to be represented with a radix point set to minimize quantization error. The dynamic range of a variable can be used to determine the number of integer bits needed to represent the full range. The optimal number of fractional bits is determined by underflow logging analysis, described in Section 3.3.4.

The Fixed-Point toolbox provides two methods for analyzing the dynamic range of

variables to determine optimal word lengths – a numeric type scope and an instrumented MEX file. The numeric type scope logs the ranges and values of individual variables. MEX stands for MATLAB executable. MEX files are compiled C programs from MATLAB code, and an instrumented MEX file contains instrumentation for logging minimum and maximum values for all variables. Both methods are used in this research. The numeric type scope will be described first.

The Fixed-Point toolbox provides a *NumericTypeScope* object that can log the ranges of values that a variable takes on and then displays a histogram of the range, some statistics and the recommended fixed-point data type. The MATLAB code in Listing 3.2 shows how the scope is defined and used for each variable. Lines 4-6 each define a scope, while lines 23-25 use the *step* command to process the current variable's value.

This code example uses the direction calculation segment of the algorithm. The inputs are the variables x and i, which are varied from 0 to 2π to simulate the possible inputs. It was found to be faster to run the simulations in batches, analyzing only one or two variables at a time, than analyzing all of the variables at once. The MATLAB simulations run significantly slower when there are too many *step* functions to process the data.

The output of the scope for the *dirx* variable, which is the output direction for the x-axis, is shown in Figure 3.2. The data values that the variable assumes are plotted in a histogram verses its number (or percentage) of occurrences. Each bin represents a power of two that values lie within. The word length can be set to be chosen automatically and then the fractional bits can be graphically controlled. The display gives the percentage of output values that fall outside of the range of the current numeric type being displayed.

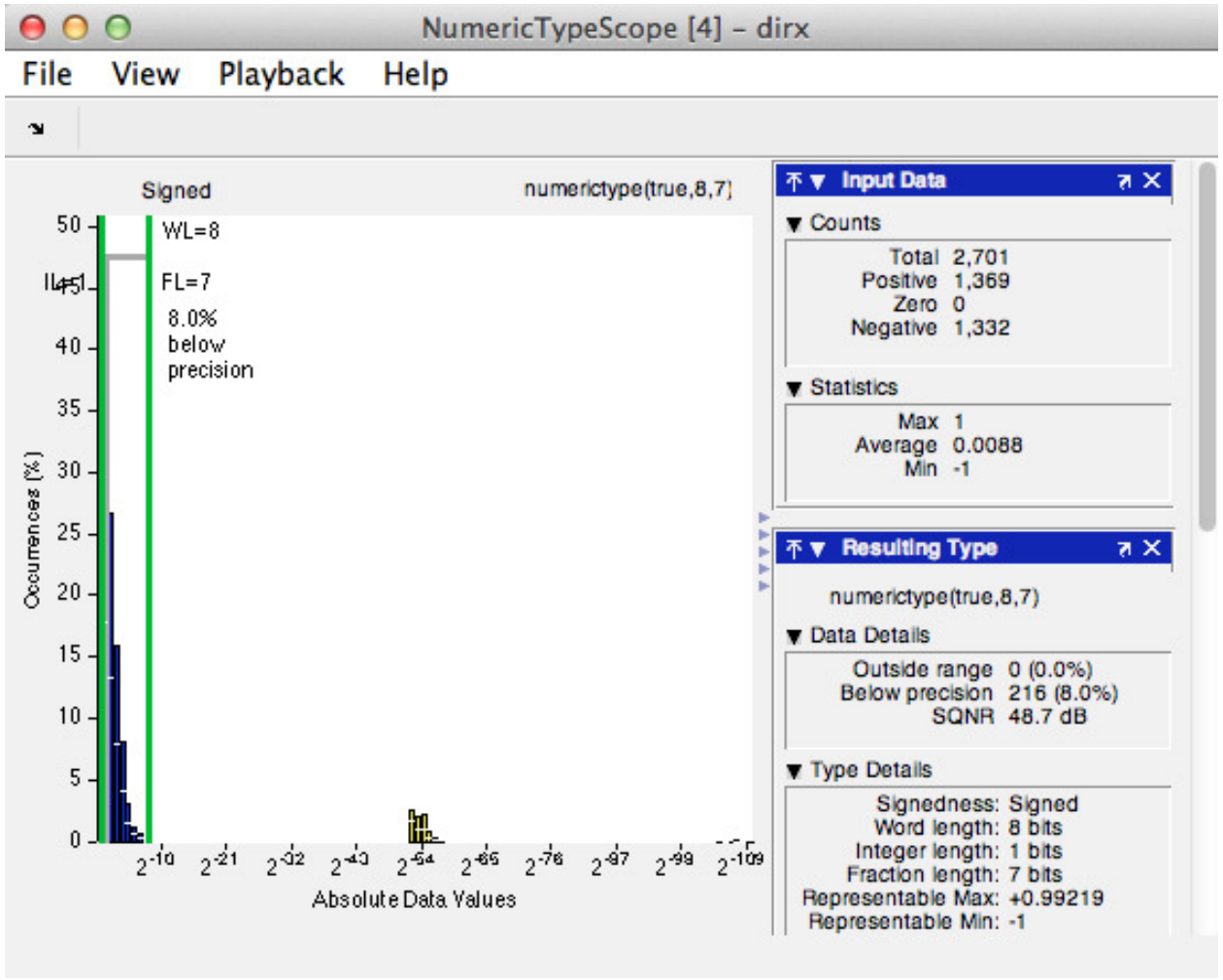

Figure 3.2: MATLAB NumericTypeScope Display for *dirx*

Listing 3.2 Direction Calculation Analysis

```
% Direction function range analysis

%Define scopes
hscopex = NumericTypeScope;
hscopey = NumericTypeScope;
hscopez = NumericTypeScope;

%Loop for different elevation(i) and azimuth(x) values
 for i = 0:10:360
    for x = 0:5:360
        dir1x = cos(degtorad(x));
        dir1y = sin(degtorad(x));
        dir1z = tan(degtorad(i));

        sumSqrs = dir1x^2 + dir1y^2 + dir1z^2;

        %Outputs
        dirx = dir1x/sqrt(sumSqrs);
        diry = dir1y/sqrt(sumSqrs);
        dirz = dir1z/sqrt(sumSqrs);

        %Process data and update scope
        step(hscopex, dirx);
        step(hscopey, diry);
        step(hscopez, dirz);
end; end
```

In Figure 3.2, it can be seen that 8% of values fall below the numeric type chosen with a word length of 8 and a fractional length of 7. Dragging the rightmost vertical line allows control over the displayed numeric type. Here, if dragged so that the bins in the middle of the graph are included, the percentage of numbers outside the range falls to 0.1%, but the word length is increased to 59 bits. This gives two clear options, use a word length of 8 or 59, depending on the precision required and the hardware available.

Utilizing the numeric type scope tool on the variables in the floating point algorithm is useful for acquiring a detailed visual description of the range of data assumed. Another

tool in the fixed point toolbox is the instrumented MEX file tool, which allows for a range analysis of all the variables in a function at once, but without the histogram for each variable.

To compile an instrumented MEX file, the *buildInstrumentedMEX* command is used with name of the function to be analyzed as an argument. This creates a compiled MEX file with the name $funct\_MEX$, where $funct$ is the function name. This MEX file can be called just like any other MATLAB function. In order to utilize the instrumentation, the MEX file needs to be called using different input data sets. A testbench is created for each function to loop through the possible inputs. An example of this is shown in Listing 3.3, where the $filimits\_MEX$ function is called on line 17 with many different inputs.

Listing 3.3 Limits Calculation Testbench

```
1  | resNE = fi(25,T,F);
2  | resA = fi(25,T,F);
3  | losLo = [3692489.8 590730 1330];
4  | losLoc = fi(losLo, T,F);
5  | gVox = [3692751.8 589587.5 1322.4];
6  |
7  | for az = 0:45:360
8  |    azi = fi(az, T,F);
9  |    for el = -90:18:89;
10 |        ele = fi(el,T,F);
11 |        for j = (gVox(1)-1000):160:(gVox(1)+1000)
12 |           gj = fi(j,T,F);
13 |           for jj = (gVox(2)-1000):180:(gVox(2)+1000)
14 |              gjj = fi(jj,T,F);
15 |              for jjj = (gVox(3)-1000):160:(gVox(3)+1000)
16 |                 gvx = [gj gjj fi(jjj,T,F)];
17 |                 filimits_MEX(resNE, resA, azi, ele, losLoc, gvx);
18 |               end
19 |            end
20 |         end
21 |     end
22 | end
```

The MEX function is nested inside several for loops, each iterating one of the inputs to the function. With normal MATLAB code, calling a function this many times would take an inordinate amount of time. The compiled MEX function allows for a 100 times or greater speedup compared to the normal code. This makes simulating a larger number of inputs possible, giving a more thorough analysis of the data types.

After running the testbench script, the instrumentation results can be displayed using the following command.

```
showInstrumentationResults filimits_MEX -proposeWL
```

The $-proposeWL$ argument enables the option to include a suggested word length for each variable. After executing the command, a code generation report window appears containing the MEX function. In this display, every variable can be highlighted for information, such as word length, fraction length, proposed word length, minimum value, maximum value and more. This is shown in Figure 3.3.

Utilizing these two data type tools allows a full analysis of every variable in the algorithm. The range of the variables define the integer length of the fixed point objects, which will be used to implement the fixed point version of the algorithm.

3.3.3 Data Type Conversion

The next step is to convert the input data types to fixed point, and modify any math operations that do not have built in support for fi objects. The math operations should

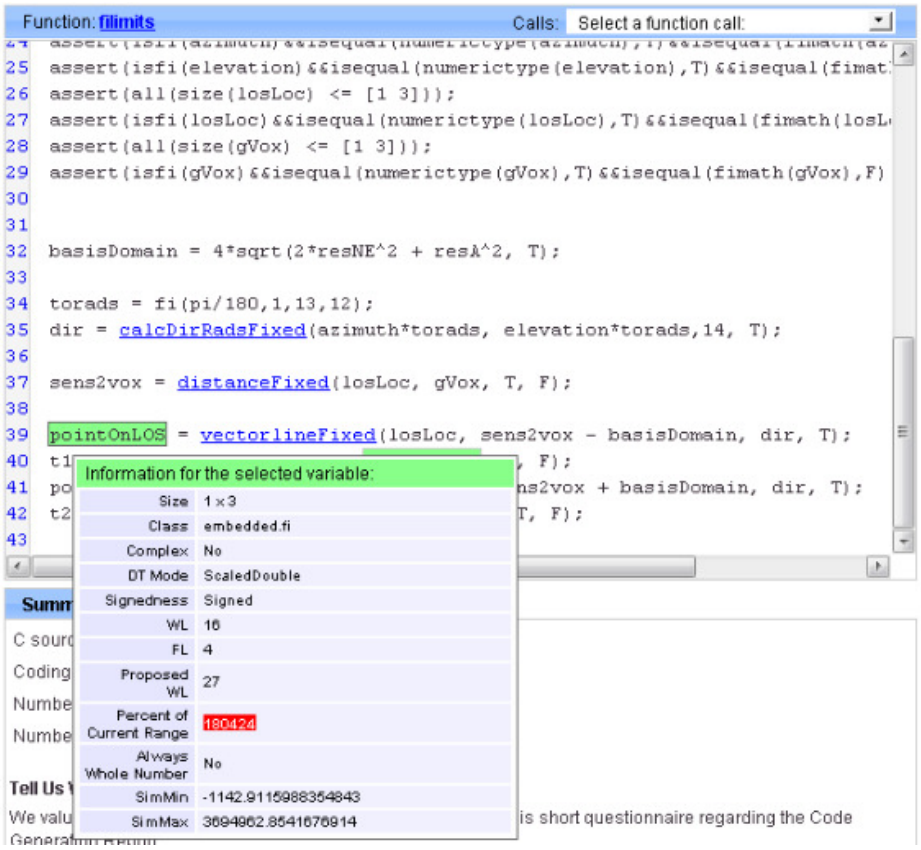

Figure 3.3: MEX Function Instrumentation Results

be computed using the same technique that will be used in hardware in order to minimize error between simulation and implementation. Listing 3.4 shows the direction calculation from the earlier listing using fi objects.

In the fixed-point version, the math operation divide needed to be changed from / to $divide(T, a, b)$, where T is the numeric type, a is the dividend, and b is the divisor. The numeric type property defines the range and precision for the output of the division. Also, the built-in trigonometric functions need to be replaced with either CORDIC functions or lookup tables. The CORDIC is an approximation of sin and cos in fixed point using vector rotation. The function is used to generate the sine and cosine values for the azimuth input.

It is also used for the elevation input, but the result is divided, sin divided by cos, to give a result of tangent. These values are then used to calculate the direction.

Listing 3.4 Fixed-point direction calculation.

```
1 |  function dir = calcDirRadsFixed(azimuth, elevation, iters, T)
2 |  % Calculates direction vector using azimuth and elevation
3 |  % inputs as radians in fixed point format.
4 |  %
5 |  % 'T' is a numerictype for the fixed-point divisions
6 |  % defining the wordlength and fractionlength for the result.
7 |  %
8 |  % 'iters' defines how many iterations the cordic calculation uses.
9 |
10| if nargin<3, iters = 14; end;
11| if nargin<4
12|   T = numerictype('Wordlength', 33, 'FractionLength', 32);
13| end
14|
15| [dir(2), dir(1)] = cordicsincos(azimuth, iters);
16| [tmpsin, tmpcos] = cordicsincos(elevation, iters);
17| dir(3) = divide(T, tmpsin, tmpcos);
18|
19|  sumSqrs = dir(1)^2 + dir(2)^2 + dir(3)^2;
20|  dir(1) = divide(T,dir(1),sqrt(sumSqrs));
21|  dir(2) = divide(T,dir(2),sqrt(sumSqrs));
22|  dir(3) = divide(T,dir(3),sqrt(sumSqrs));
```

The function in the listing calculates the outputs using the inputted values numeric type and the divide function's numeric type. The numeric types of the direction outputs should be pre defined with the numeric types determined in the range analysis. This can be done using $dir(1) = fi(0,'numerictype',T)$, where T is the desired numeric type.

The remaining data types and operations are converted in the same way as the direction calculation. Squaring and cubing operations are converted to multiplications. The MATLAB square root function, $sqrt()$, is compatible with fixed point data types, simplifying the

conversion. The basis function arithmetic is replaced with a lookup table that stores the possible results for the range of possible inputs. Once all of the data types and operations are converted, the algorithm is ready to be tested for overflow and underflow characteristics.

3.3.4 Overflow/Underflow Logging

The next step is to turn on overflow and underflow warning messages along with data logging. This will allow for an understanding of what operations have quantization effects on the data. At this point, overflow messages should be minimal, since the integer word length was determined in the previous section. Underflow warnings, on the other hand, will represent where the fraction length is not sufficient enough to hold the required precision. The first three lines of Listing 3.5 are used to turn on the overflow and underflow warnings. The output to the warning message implementation is shown in Listing 3.6. This output is shown in the MATLAB console during simulation runtime.

Backtrace is turned on to see what line numbers the warnings occur at. The line $fipref('LoggingMode','on')$ turns on data logging. A report can be printed with the $logreport(var1, var2, ...)$ function. An output of the log report function is shown in Figure 3.4.

In Figure 3.4, the minimum and maximum numbers logged, and the lower and upper bounds are given in a table. In order to gather a better log of the values, the datatype can be overridden to a scaled double data type, which is a fi object that logs the values it takes on with double precision. The output of this log is given in Figure 3.5.

Listing 3.5 Overflow/Underflow Warning Message Implementation

```
warning on  fi:overflow
warning on fi:underflow
warning off backtrace
fp = fipref;

fipref('LoggingMode', 'on');
iters = 14;
T = numerictype('Wordlength', 33, 'FractionLength', 32);
% Loop for different elevation(i) and azimuth(x) values
for i = 0:10:360
    for x = 0:5:360
        azimuth = fi(deg2rad(x), 1, 6, 5);
        elevation = fi(deg2rad(i), 1, 6, 5);
        [dir(2), dir(1)] = cordicsincos(azimuth, iters);
        [tmpsin, tmpcos] = cordicsincos(elevation, iters);
        dir(3) = divide(T, tmpsin, tmpcos);

        sumSqrs = dir(1)^2 + dir(2)^2 + dir(3)^2;
        dir(1) = divide(T,dir(1),sqrt(sumSqrs));
        dir(2) = divide(T,dir(2),sqrt(sumSqrs));
        dir(3) = divide(T,dir(3),sqrt(sumSqrs));
    end
end
```

Listing 3.6 Overflow/Underflow Warning Messages

```
...
Warning: 1 underflow(s) occurred in the fi bitshift operation.
Warning: 1 underflow(s) occurred in the fi bitshift operation.
Warning: 1 underflow(s) occurred in the fi bitshift operation.
Warning: 1 underflow(s) occurred in the fi bitshift operation.
Warning: 1 overflow(s) occurred in the divide operation.
Warning: 1 overflow(s) occurred in the fi assignment operation.
Warning: 1 underflow(s) occurred in the fi bitshift operation.
Warning: 1 underflow(s) occurred in the fi bitshift operation.
Warning: 1 underflow(s) occurred in the fi bitshift
...
```

Knowing where the underflows occur allows for those values to be analyzed for quantization errors. The numeric type scope from earlier includes a measurement of the SQNR, so the fraction lengths can be adjusted and iterated until an acceptable SQNR is reached.

```
> logreport(azimuth, elevation)
                minlog         maxlog     lowerbound     upperbound
   azimuth     0.96875        0.96875             -1        0.96875
 elevation       0.875          0.875             -1        0.96875
>
```

Figure 3.4: Logreport output

```
-> logreport(azimuth, elevation)
                minlog         maxlog     lowerbound     upperbound
   azimuth    2.268928       2.268928             -1        0.96875
 elevation   0.5235988      0.5235988             -1        0.96875
->
```

Figure 3.5: Logreport output with Scaled Double data type

3.4 Results

In order to verify that the final fixed-point implementation can accurately represent the floating point algorithm, the fixed-point algorithm needs to be tested. The same testbench input data sets used in the floating point verification are used here so that the results can be compared directly for error.

The testbench data sets are from one collection of data from a complete projection matrix calculation, provided by MESH, Inc. For the purpose of testing, the data sets are split up into the individual sensor angle inputs, as described in Section 2.1. Each set of data is the calculation for every grid voxel in the grid for one sensor angle measurement. There are 1,146 sensor angles, and therefore data sets, each calculated against all 841 grid voxels. These are real-world values from sensor measurements, which means that the simulation results should accurately represent the actual implementation.

While analyzing the range and precision of the data types throughout the conversion to fixed-point, special attention is paid to the inputs and outputs from the basis function

described in section 2.1. Every data type setting is important, but the basis function is the value that will be accumulated thousands of times as the algorithm loops through the grid voxels and measurement rays, so quantization errors will grow over time. This is implemented using lookup tables for the basis function, so not only is the precision of the input address important, but also the precision of the stored values and the fixed point representation they will assume in memory.

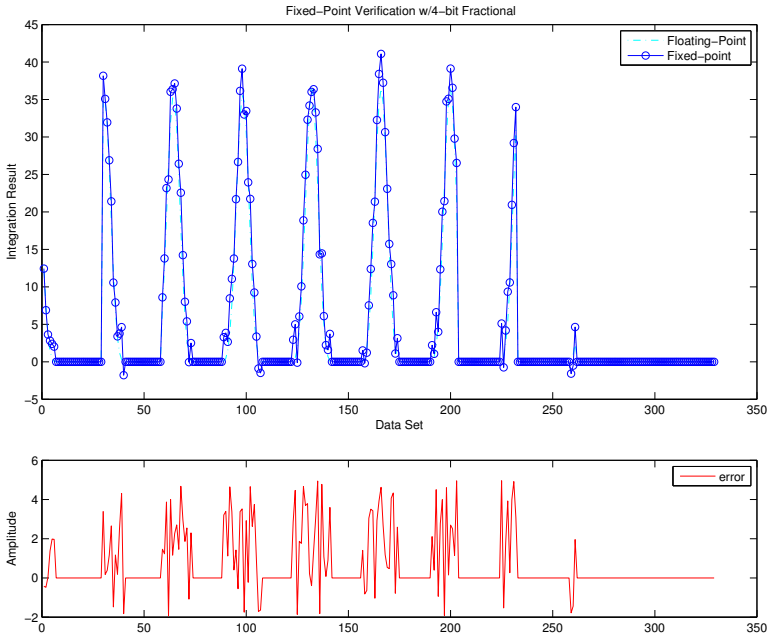

Figure 3.6: Fixed-Point simulation for basis value with 4 fractional bits

Figures 3.6, 3.7 and 3.8 show the error in the output for one of the test data sets for different numbers of fraction length for the output of the basis function.The range analysis showed that a word length of 15 could represent the range of the basis function result. The fractional width was determined to be optimal with 14 bits using the SQNR analysis in the previous section.

In Figure 3.6, the number of fractional bits was chosen to be only 4, causing a high quantization error. The maximum error magnitude for this data type was approximately 5.

This is $5/40 = 12.5\%$ error, which is an unacceptable amount of error. Figure 3.7 shows the same data set but using 8 fractional bits to represent the basis value. Here, the maximum magnitude of the error is reduced to about 10×10^{-3}, or 2.5% error.

The optimal fractional bits, 14, are chosen for the data set simulated in Figure 3.8. Here it can be seen that the maximum error magnitude is approximately 6×10^{-4}, or 0.0015%. This error is small enough that the error propagation has a minimal effect on the output.

This computational model is then executed with a sufficient number of the input data sets in order to verify that the error level is small for all of the different inputs. A random selection of 250 of the 1146 data sets for simulation produces an error of approximately 1.6×10^{-3}, which is 0.0041% of the maximum integration result.

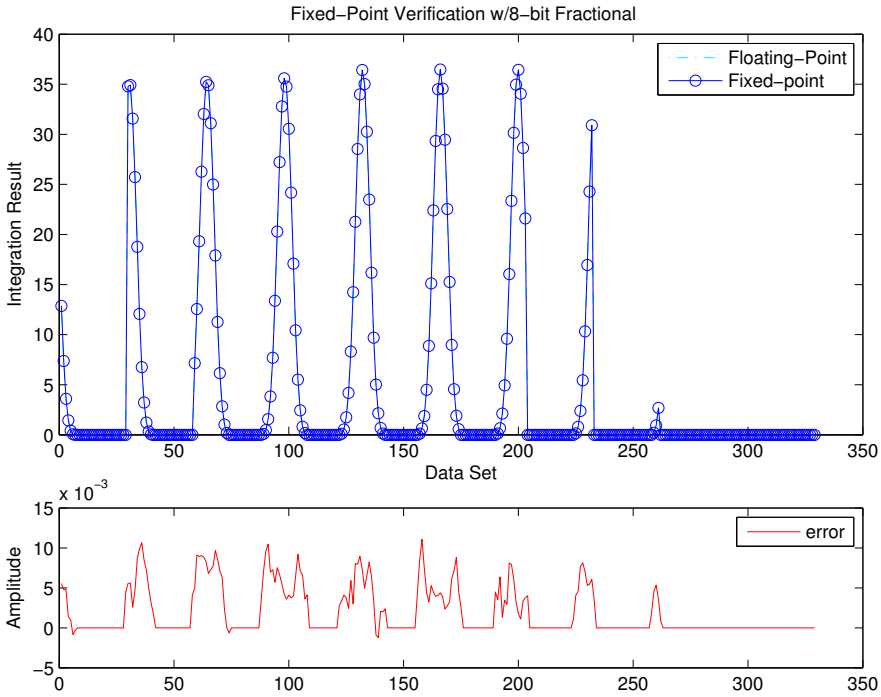

Figure 3.7: Fixed-Point simulation for basis value with 8 fractional bits

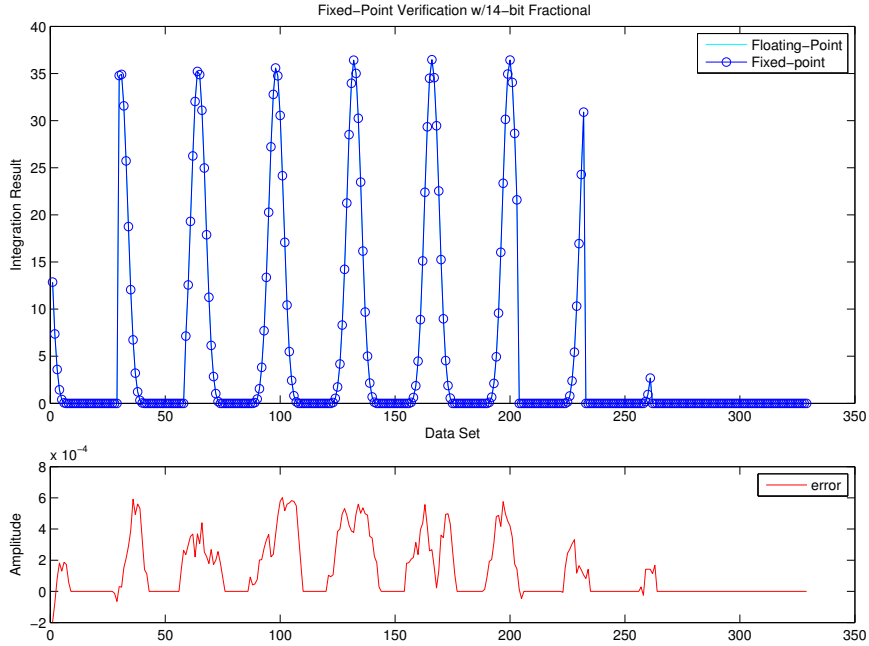

Figure 3.8: Fixed-Point simulation for basis value with 14 fractional bits

4 Implementation

4.1 Introduction

After simulating the algorithm and optimizing the fixed point data representation to be used, the design of the logic begins. As stated earlier, the implementation design methodology used here is MATLAB/Simulink with the Xilinx System Generator Blockset. This methodology allows the generation of HDL from a Simulink model. The generated HDL can be used in a Xilinx ISE project, where custom logic can be designed to communicate with the model.

The system design is made easier by the fixed-point simulations. The algorithm has already been broken down into its individual operations, which is how it needs to be implemented in hardware, and the rounding and truncation at each intermediate stage of the calculation have been determined. The resulting single stream of logic that computes a single element of the projection matrix can then be instantiated multiple times on the FPGA. With enough area and resources, the entire matrix can be computed at once.

This may require one very large FPGA or multiple FPGAs with interconnections. Commercially available Off-The-Shelf (COTS) FPGA boards are being marketed as "algorithm accelerator" platforms, such as the Dini Group's (www.dinigroup.com) DNBFC_S12_PCIe, shown in Figure 4.1, which contains 12 Spartan 6 FPGAs for data processing and 1 for data flow control.

Without a board such as this one, the research work cannot demonstrate complete

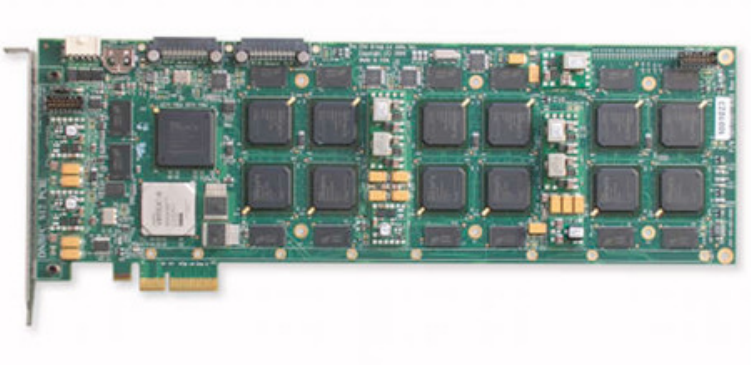

Figure 4.1: DNBFC_S12_PCIe by Dini Group. Contains 12 Spartan 6 FPGAs for processing and one to handle data flow [7].

parallelism. In such a case, the maximum number of parallel operations permitted on the FPGA are used, and the operation is performed iteratively, N/x times. N is the total number of elements in the matrix and x is the number of parallel computations able to be performed. For example, a 100 element matrix can be processed in 20 cycles if the possible parallel computations was 5, and 10 cycles if it was 10.

4.2 System Generator

The Xilinx System Generator is an extension to Simulink that offers high-level graphical design of high-performance DSP systems on FPGAs [5]. The System Generator blockset contains functions for digital logic, arithmetic and signal processing, all of which are bit and cycle accurate. Figure 4.2 shows the basic setup for System Generator designs.

System Generator allows for the use of Xilinx blocks and Simulink blocks in one model, separated by a gateway block. The gateway blocks convert the data from the Simulink default double data type to a specified fixed point data type, and vice versa. All Xilinx

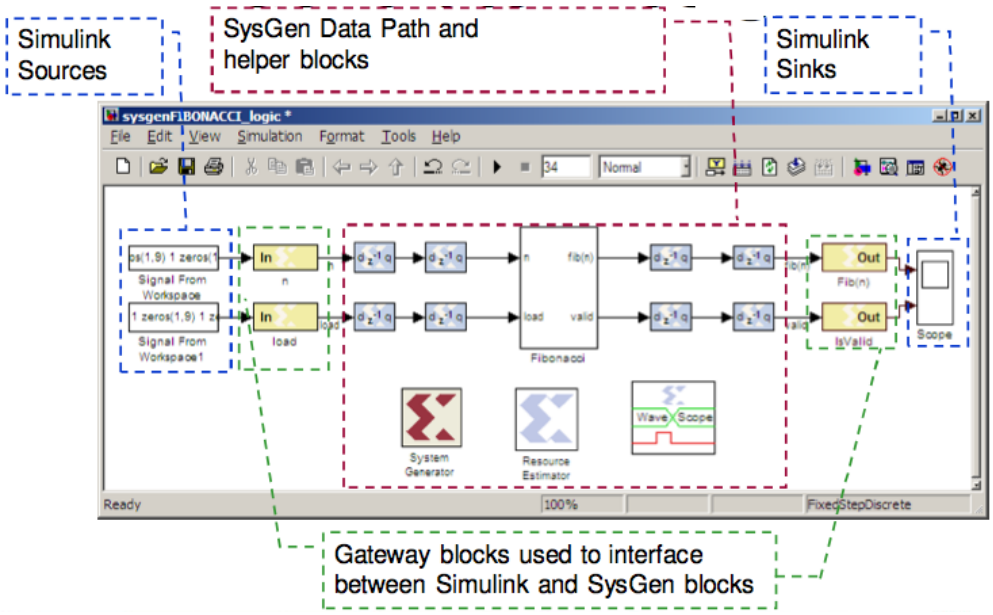

Figure 4.2: Basis System Generator Design, Xilinx 2010©

blocks must be inside the gateways and all Simulink blocks must be outside the gateways for the model to run properly.

4.3 Hardware Design

4.3.1 Distance Calculation

A distance calculation function is used multiple times in the algorithm to calculate the integration limits. This operation computes the distance between two three dimensional points using Equation 4.1.

$$dist = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2} \tag{4.1}$$

The distance operation is performed in hardware using addition, subtraction and mul-

tiplication blocks for the expression inside the square root. The square root operation is performed using the CORDIC algorithm, which is available in the Xilinx Blockset. The CORDIC Square Root block has a parameter for the number of iterations the algorithms uses, which allows a tradeoff to be made between accuracy and clock cycles. The distance calculation subsystem is shown in Figure 4.3.

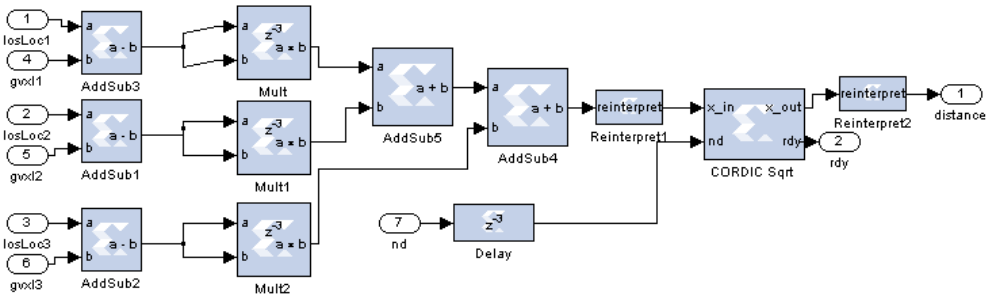

Figure 4.3: Distance Calculation Implementation

The only blocks in the distance calculation subsystem that have latency are the multiplier blocks and the CORDIC square root block. The CORDIC block has a latency given by Equation 4.2.

$$\text{Latency} = 7 + \text{integer bits} + \text{modulo2(integer bits)} \tag{4.2}$$

The multiplier blocks each have a latency of 3. Since they execute in parallel, the delay for the distance calculation is the CORDIC square root latency plus 3.

4.3.2 VectorLine Function

The algorithm uses a function called *vectorLine* to calculate a new position on a line when iterating through the integration, as well as when finding the integration limits. The inputs to the function are the origin of the line, its direction and its step size. This gives a point on a line that is a distance from its origin equal to the step size. The expression representing this function is given by Equation 4.3.

$$position(1:3) = origin(1:3) + t * dir(1:3) \tag{4.3}$$

The hardware implementation for the vector line function is shown in Figure 4.4. The subsystem utilizes three multiplier blocks and three addition blocks. The multiplier blocks have a latency of one, while the addition blocks have a latency of zero, so the *vectorLine* implementation has a latency of only one clock cycle.

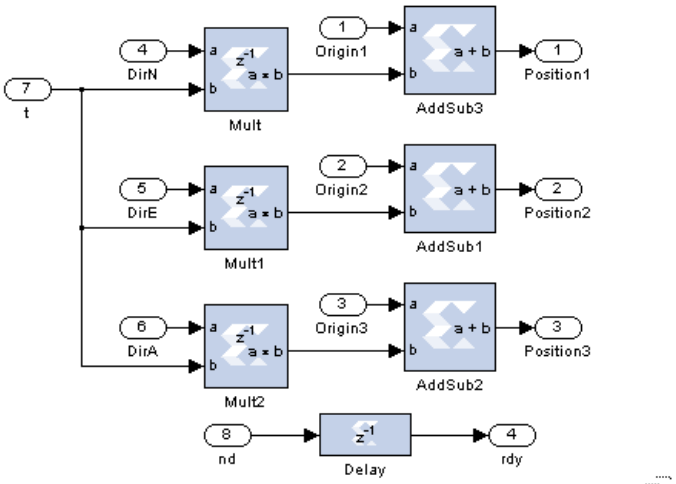

Figure 4.4: Vectorline Function Implementation

4.3.3 Direction Calculation

A direction calculation function is used by the algorithm to aid in the calculation of the limits as well as for the sub direction calculations. This function has two inputs, azimuth and elevation, that correspond to the angle of the sensor. The azimuth is the 360 degrees of horizontal freedom, while the elevation is the vertical angle. Code Listing 4.1 shows the function of the direction operation.

Listing 4.1 Direction Calculation

```
dir(1) = cos(azimuth*(pi/180));
dir(2) = sin(azimuth*(pi/180));
dir(3) = tan(elevation*(pi/180));

sumSqrs = dir(1)^2 + dir(2)^2 + dir(3)^2;

dir(1) = dir(1)/sqrt(sumSqrs);
dir(2) = dir(2)/sqrt(sumSqrs);
dir(3) = dir(3)/sqrt(sumSqrs);
```

This function requires the use of CORDIC blocks to calculate sine, cosine, and tangent. Since tangent is not available as a direct CORDIC implementation, it is calculated from sine and cosine. The inputs to the CORDIC block also must be within the range $-\frac{\pi}{2} \leq \frac{\pi}{2}$. The inputs to the direction calculation may range from -2π to 2π, so this also needs

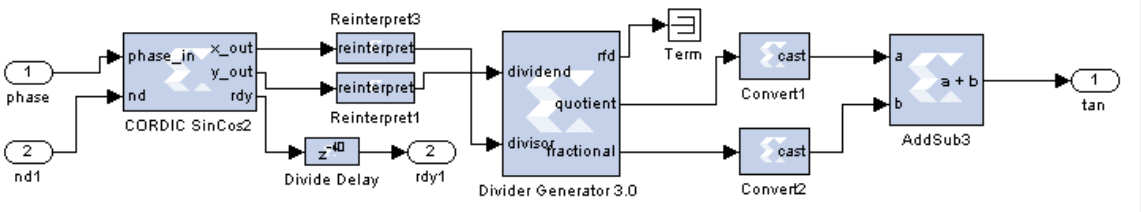

Figure 4.5: Tangent Implementation

to be corrected in hardware. The solution to this is to use an *mcode* block from the Xilinx Blockset. An *mcode* block allows a subset of the MATLAB language to be used and implemented in hardware. Listing 4.2 shows the MATLAB code used to correct the input phase.

Listing 4.2 Input Phase Correction M-Code Block

```
function [x, caz, cel] = correctphase(az, el)
       if(az > 180)
                caz = 180 - az;
                x = 1;
       elseif(az <-180)
                caz = -180 + az;
                x = 1;
       else
                x = 0;
                caz = az;
       end

       if(el > 180)
                 cel = el - 180;
       elseif(el < -180)
                 cel = el + 180;
       else
                 cel = el;
       end
end
```

The sine and cosine calculations are implemented using the available CORDIC SIN-COS block, but the tangent function needs to be implemented using a combination of the CORDIC block and a divider, as shown in Figure 4.5.

The denominator for the three dimensions of the calculation is equal to the square root of the sum of the squares of the three trigonometric functions. The denominator block is shown in Figure 4.6 and utilizes the CORDIC algorithm to perform the square root

operation.

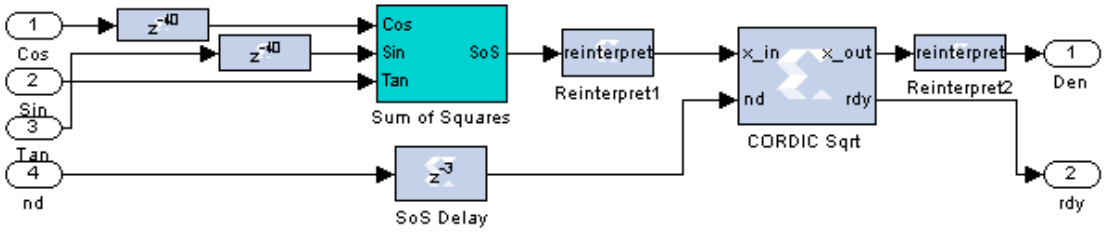

Figure 4.6: Denominator Implementation

The final operation for the direction calculation is to divide the numerators by the denominators. Delays are used on the data lines in parallel with the denominator calculation to make sure the data is aligned. The Xilinx block set contains a divider generator, which is used to perform the division here. This block has a latency of 40 clock cycles. The full implementation of the direction calculation is shown in Figure 4.7.

Adding up the latency of the CORDIC blocks, the tangent and denominator calculations, and the divison operation results in a total latency of 128 clock cycles. The direction calculation has a partially pipelined design, allowing data to be entered in every 88 clock cycles. This means that once the initial 128 clock cycles have passed, data at the output is valid every 88 clock cycles.

4.3.4 Integration Limits

The integration limits calculation utilizes the direction calculation, vectorLine, and distance blocks. The basis domain is the range that the integration is calculated over. This function requires multiple multiplications and a square root operation, and is implemented as a LUT

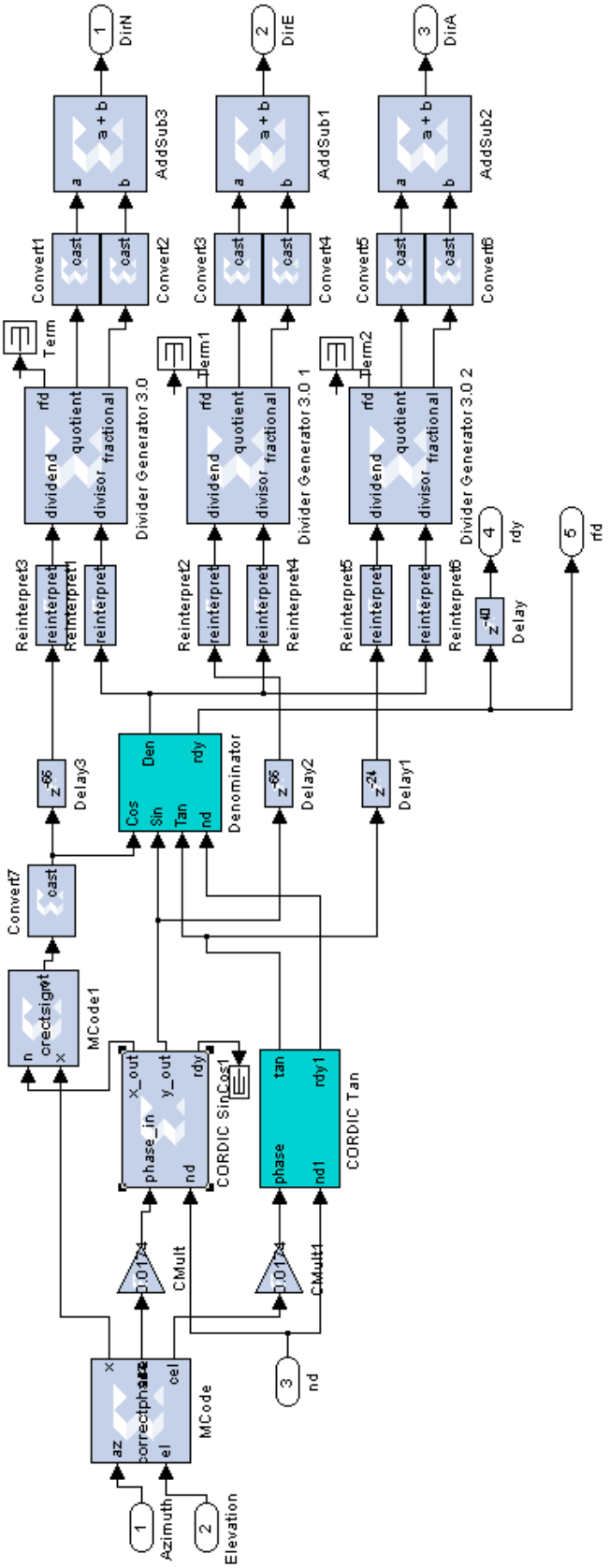

Figure 4.7: Direction Calculation Implementation

to save clock cycles and resources. This is written into a variable by a MATLAB script and loaded into the LUT block. The integration limits block has a latency of 132 clock cycles, and is shown in Figure 4.8.

4.3.5 Subdirection Calculation and RAM

The subdirection calculation splits the sensor field of view into subdirections to be integrated over and added together. The calculation has to loop for every direction outputted, meaning there must be a control block to handle the looping and incrementing. The control block starts a new calculation of the directions and handles the incrementing of the counters using relational and logic blocks. The overall sub direction calculation is shown in Figure 4.9.

The latency for the subdirection block is high and depend on the number of subdirections to be calculated. The default number is 255 subdirections, and the latency for a single direction is 131 clock cycles. Thus, the total number of clock cycles is 33,400 clock cycles and with a 100 MHz clock, this is equivalent to 33.4 microseconds.

The calculated subdirections need to be used on every iteration of the integration loop, so there needs to be some sort of storage for the results. This is implemented using a Dual Port RAM block, which allows reading and writing to different addresses in memory. The three directions calculated are concatenated and stored in one memory address. When read, the data is spliced back into three registers. A memory width of 57 bits and depth of 256 was used. This implementation is shown in Figure 4.10.

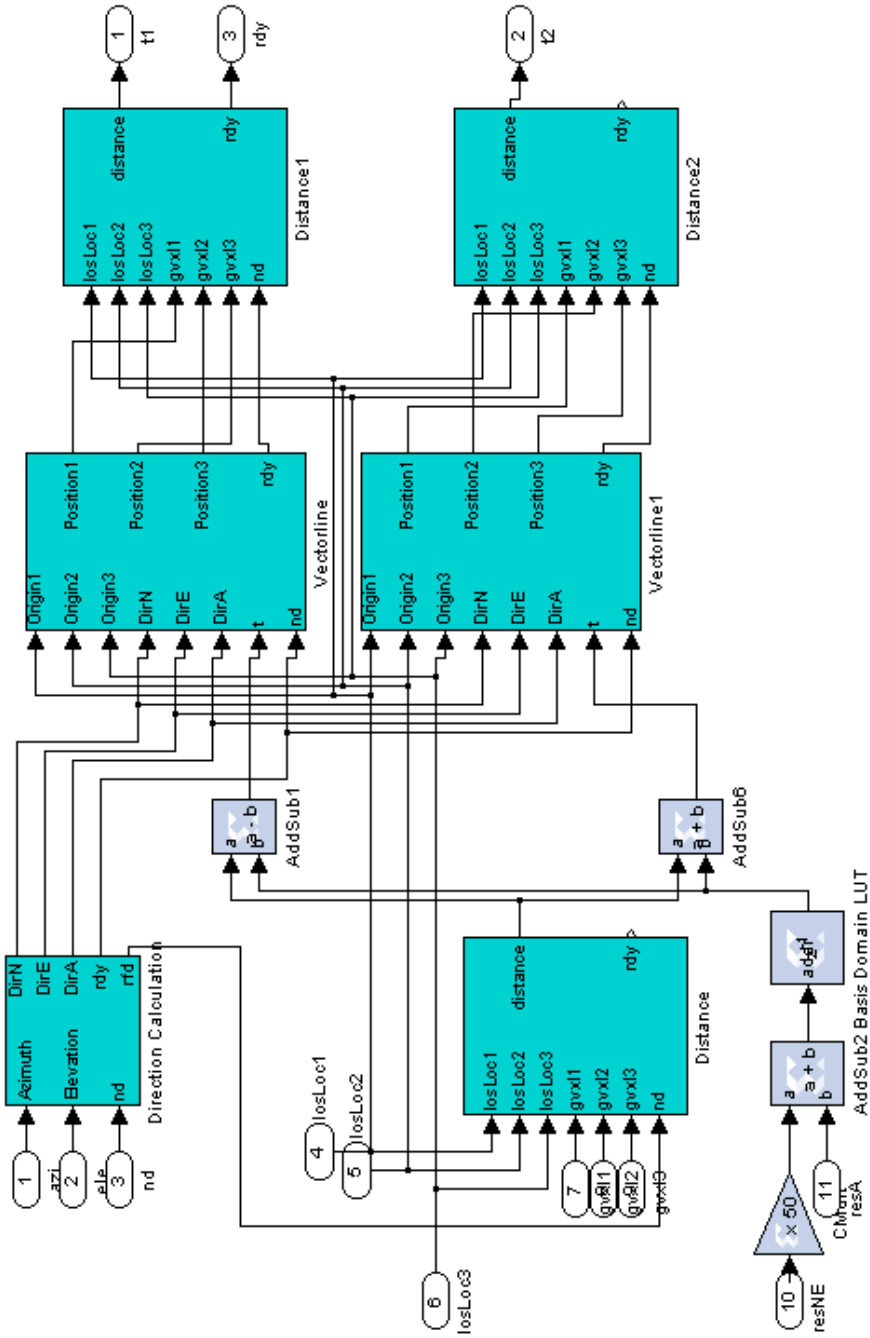

Figure 4.8: Integration Limits Calculation

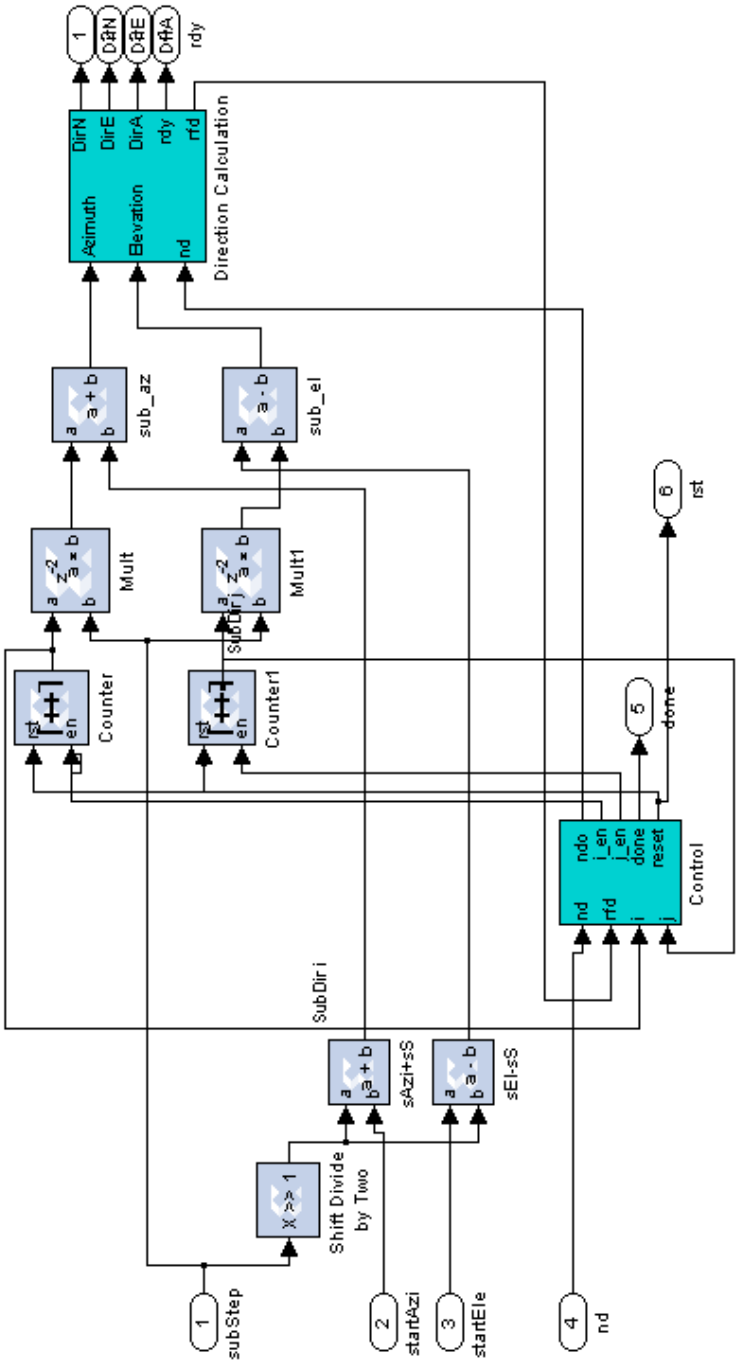

Figure 4.9: Subdirection Calculation Implementation

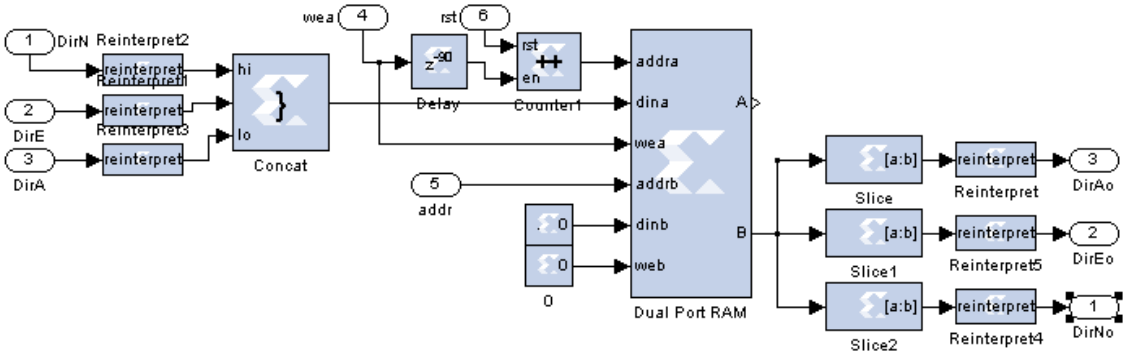

Figure 4.10: Subdirection RAM

4.3.6 Basis Function

The basis function is implemented the same way as in the simulation, with lookup tables. There are three lookup tables, one for each dimension of the basis function, the results of which are multiplied together to form the result. The lookup tables are implemented by writing the values to a variable in MATLAB using a script, and then loading those values into the LUT. This implementation is shown in Figure 4.11.

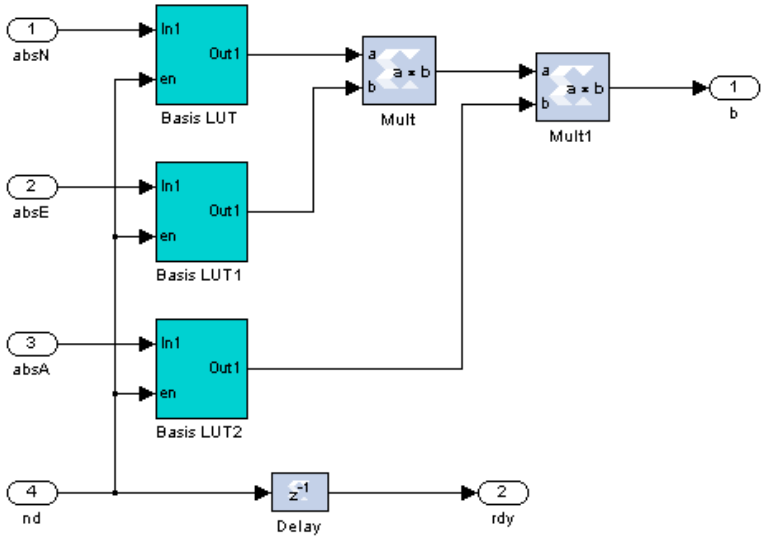

Figure 4.11: Basis Function Implementation

4.3.7 Integration

The integration function is implemented as a discrete summation using an accumulator. It utilizes the previous vectorLine and basis function blocks, as well as two other blocks to calculate the position on the line for the next integration, and to calculate the relative position of the current grid voxel point. The relative position block utilize a subtraction block and a m-code block, which allows MATLAB code to be implemented in hardware. The m-code block was used to output the absolute value of the input. The position block takes the limits as inputs and outputs the next point on the line to be integrated for each iteration.

The integration has to loop through a few hundred steps, depending on the limits, for each subdirection. For the example inputs used, there were 346 steps in the integration and 255 sub directions. That is 88,230 integration steps each taking 6 clock cycles to compute, for a total of 5,823,180 clock cycles, which is 58 milliseconds with a 100MHz clock.

4.4 Simulation Results

Due to the large number of computational resources and RAM for the simulation of the entire integration loop, the implementation was tested in segments. The implementation was apportioned in the same manner as the MATLAB simulation was segmented. This facilitates the direct comparison of results to determine if there is any error.

Since each segment in hardware is built using the exact same implementations as in the MATLAB simulation as well as the same fixed point data types and quantization, the

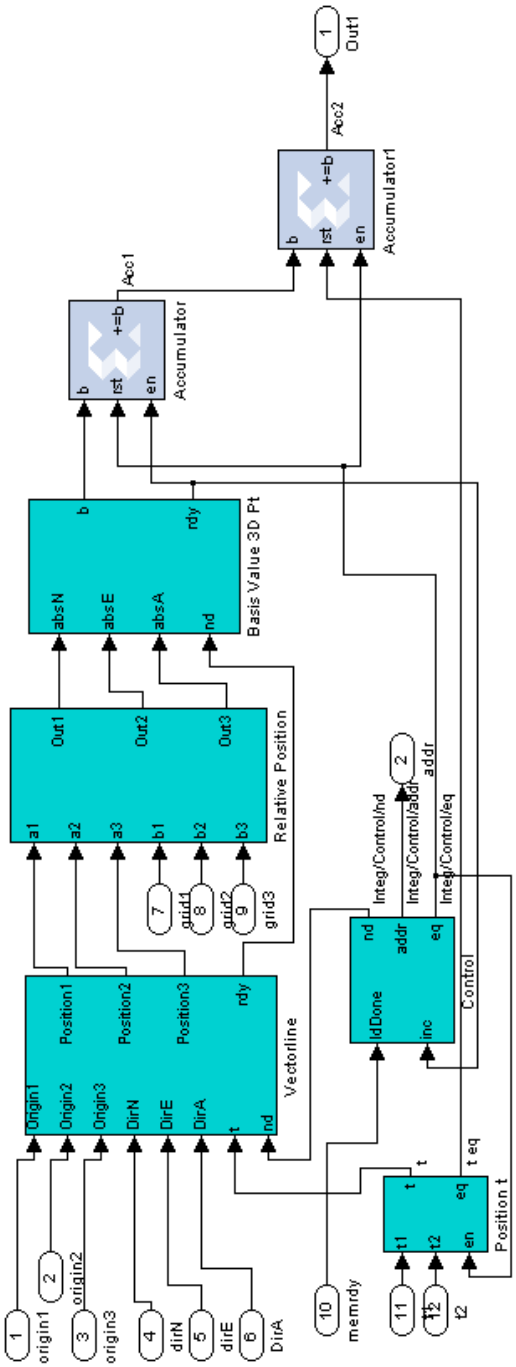

Figure 4.12: Integration Implementation

results of each segment exactly matches the corresponding MATLAB segments. It can then be said here that correctly choosing the *fimath* parameters, described in Section 3.1, in a fixed-point simulation allows the simulation and hardware implementations to correspond.

One final result of the analysis is the resource estimation. The Xilinx System Generator comes with a Resource Estimator block that gives an estimation of the hardware utilization. This is shown in Figure 4.13. Here, the input-output buffers (IOBs) and block RAM (BRAMs) usage is at zero. The IOBs in the simulation are not set up to be FPGA inputs and outputs, so they are not utilized. The block RAM is used to implement the lookup tables, and is contingent on the size of the vectors that are used to initialize the lookup tables.

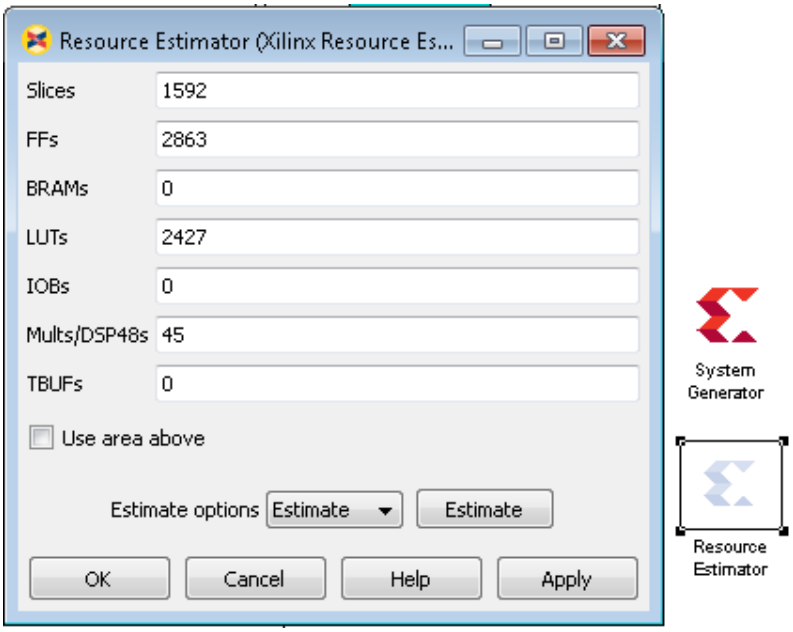

Figure 4.13: Resource Estimation

The Spartan 6 XC6SLX150, which is one of 12 FPGAs utilized on the Dini Group algorithm acceleration board shown in Figure 4.1, contains 23,000 slices, 93,152 LUTS, 184,000 flip-flops, 4,800kb of block RAM and 180 DSP48 multipliers. The limiting factor constraining how many copies of the design can be implemented is the number of multipliers.

This design is estimated to utilize 45 DSP48 multipliers, which means one Spartan 6 can contain up to four implementations of the algorithm. The remaining FPGA resources can be used to implement multipliers in order to increase the number of design copies implemented on each device.

The Xilinx System Generator includes multiplier blocks that can be implemented using LUTs. According the Xilinx, the largest multiplier implementation, using 53-bit inputs, utilizes 150 LUTs [14]. The remaining 90,000 LUTs allow for 604 more multiplier implementations. Dividing by the number of multipliers per design, which is 45, yields 13 more copies, for a total of up to 17 design copies per FPGA.

The number of projection matrix calculation in the dataset given by MESH, Inc. is 964,000. If each computation then executes in 58.6 milliseconds with a 100 MHz clock, this would require 56,490 seconds to calculate the entire dataset using a single design copy. Dividing by 17, the maximal number of designs per FPGA, yields 3323 seconds, and dividing again by 12, the number of FPGAs on the board, yields 276.9 seconds. This is approximately three times larger than the real-time design requirements of 90 seconds. However, this acceleration board can utilize a clock of up 350 MHz, which allows the computation time to meet the required specifications.

5 Conclusion

The completed work for this research includes the simulation of the algorithm in Fixed-Point MATLAB, the optimization of data types, and the hardware implementation utilizing the Xilinx System Generator. The research explores the use of the MATLAB Fixed-Point Toolbox to efficiently convert a floating point algorithm to a fixed-point one using optimal word lengths and radix points, to be implemented on an FPGA. The results show that a fixed-point implementation can be optimized so that the error between floating and fixed point models can be greatly minimized. This research also shows that utilizing the correct data type properties for the MATLAB simulation can result in a hardware-true simulation, with little to no error measured between simulation and implementation.

References

[1] Ray Andraka. "A survey of CORDIC algorithms for FPGA based computers". pages 191–200, 1998.

[2] T. M. Bhatt and D. McCain. "Matlab as a Development Environment for FPGA Design". In *Design Automation Conference*, June 2005.

[3] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach. "Acelebrating Compute-Intensive Applications with GPUs and FPGAs". In *Symposium on Application Specific Processors*, 2008.

[4] D.Verhoeven. "Limited-Data Computed Tomography Algorithms for the Physical Sciences". *Applied Optics*, 32, 20:3736–3754, 1993.

[5] R. Esposito. *"Turbo Coding Implemented in a Fine Grained Programmable Gate Array Architecture"*. PhD thesis, Temple University, 2009.

[6] R. Esposito, J. Mountney, L. Bai, and D. Silage. "Parallel Aarchitecture Iimplementation of a Reliable (k, n) Image Sharing Scheme". In *IEEE International Symposium on Parallel and Distributed Systems*, 2008.

[7] Dini Group. "High Performance Computing Algorithm Acceleration FPGA Board". `http://www.dinigroup.com/new/DNBFC_S12_PCIe.php`.

[8] T. Gruber, L.Grim, C. Keiser, and W. Ginley. "Sensor-Netting Algorithm for CB Threat Mapping". In *Proc. SPIE*, volume 7665, April 2010.

[9] B. Liu, Y. Chai, and X. Zhang. "Research on Fixed-Point Simulation Approaches in the Design of FPGAs". In *Information Engineering and Computer Science (ICIECS), 2nd International Conference on*, 2010.

[10] The Mathworks. "Fixed-Point Toolbox". `http://www.mathworks.com/products/fixed/`.

[11] B. Parhami. *"Computer Arithmetic: Algorithms and Hardware Design"*. Oxford University Press, New York, 2010.

[12] D. Silage. "Architecture for 3D-Tomographic Processing". *Coperative Deployment of Next Generation Chemical Standoff Sensors*. STTR A10A-022-0241.

[13] D. Silage. *"Embedded Densign in Verilog Using Programmable Gate Arrays"*. Bookstand Publishing, Houston, TX, 2008.

[14] Xilinx. "LogiCORE IP Multiplier v11.2". `http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf`.

[15] R. Yates. "Fixed-Point Arithmetic: An Introduction". In *Digital Signal Labs, Technical Reference*, 2009.