ABSTRACT

A Comparison of Field Programmable Gate Arrays and
Digital Signal Processors in Acoustic Array Processing

Jeremy C. Stevenson

Mentor: Russell W. Duren, Ph.D.

The Field Programmable Gate Array's (FPGA) constant growth in computing power has given embedded system developers a choice to replace their current processors with a FPGA. However, most systems continue to use the original processor due to familiarity and design speed. Design tools, such as Simulink for MATLAB, have created a potential for significantly reducing FPGA development time. This potential was explored by developing an acoustic array processing system on both a FPGA and a DSP (Digital Signal Processor). The system includes a filtering stage, a correlation stage, and a trigonometric math stage. All of these stages are computationally intensive which provide an accurate portrayal of the chips' capabilities. The paper documents the comparison of the FPGA and the DSP implementations in regards to the performance of each implementation, the design time of each implementation and the capability of the design tools used in each implementation.

A Comparison of Field Programmable Gate Arrays and
Digital Signal Processors in Acoustic Array Processing

by

Jeremy C. Stevenson

A Thesis

Approved by the Department of Electrical and Computer Engineering

James B. Farison

James B. Farison, 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

Russell W. Duren, Ph.D., Chairperson

Michael W. Thompson

Michael W. Thompson, Ph.D.

Peter M. Maurer

Peter M. Maurer, Ph.D.

Accepted by the Graduate School
May 2006

Larry Lyon

J. Larry Lyon, Ph.D., Dean

Copyright © 2006 Jeremy Stevenson

All rights reserved

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Dr. Duren and Dr. Thompson for giving me a chance to come back to Baylor as a graduate student to work with them on this project. Without their support, I would not have had this opportunity nor would I have finished this project. I would like to thank the rest of the Baylor Engineering faculty. They provided me with an excellent education, and they also helped improve my faith through their many student and faculty Christian studies.

CHAPTER ONE

Introduction

The Field Programmable Gate Array, FPGA, is a highly customizable chip often used for logic functions. The FPGA is programmed using a hardware description language, which is used to program the connections for the individual gates in the FPGA. An FPGA design is usually limited by the amount of gates available on the chip. As the number of available gates increase on the FPGA, more and more complex designs can be placed on the chip. The FPGA can also be limited by the time it takes a signal to travel from one gate to another as well as the time it takes to pass through a single gate [1]. Advances in FPGA technology have increased the number of available gates while reducing the time it takes a signal to travel through and between the gates.

The Digital Signal Processor, DSP, unlike the FPGA has a set hardware configuration. A DSP processor is similar to other types of microprocessors yet is distinguished by an architecture designed to efficiently implement many standard DSP operations. A DSP processor works well in signal processing applications since it is optimized to efficiently process signals, is relatively inexpensive, and has a well-defined development path. Since the different types of digital signals require only one set of hardware, a DSP processor can be mass produced so that the hardware is constant for all chips and functionality is defined through software [2].

The advances in FPGA speed and design complexity are now allowing the FPGA's to be able to perform the same signal processing applications as the DSP's [3].

In fact, the computational speed of FPGA's make their use an attractive alternative to DSP processors for computationally intensive applications. In this project, it will be shown that the FPGA is capable of implementing the same design as the DSP with the advantage of achieving much higher computational speeds.

Another major step that FPGA's are taking towards the DSP in regards to signal processing is the utility of the tools used to create the designs. Software/Hardware development tools are essential for creating a functional design and create either an optimized hardware layout when working with a FPGA or an optimized software routine when working with a DSP. The tools used in the DSP software design have been modified so that the software used in the DSP runs efficiently using a pipeline structure that offers only a limited degree of parallelism. However, the tools used in the FPGA design optimize the design to run in a highly parallel manner. The tools for the DSP generally only require knowledge of the C language, while the FPGA tools require the user to know less common hardware description languages such as VHDL or Verilog. Tools like System Generator from Xilinx have reduced the users need to know VHDL and Verilog by using a block representation for large pieces of hardware design code. Advances in the FPGA design tools, along with the performance increase the FGPA provides, have opened the door for signal processing designers to increasingly use FPGA's over the traditional DSP's.

This paper will show the comparisons of the DSP and FPGA implementations in regards to their performance as well as the design time and capability of the tools of each implementation. A sound localization algorithm was used as the test implementation for the DSP and FPGA systems.

CHAPTER TWO

Algorithm

Physical System

The application used as the test bed for the DSP/FPGA comparison is an implementation of a sound localization system. The system consists of an array of two or more microphones with the direction of an incident sound wave calculated from the difference in arrival times at the respective microphones in the array. The design of the array of microphones is such that it allows for comparisons along the three major Cartesian axes. Multiple microphones along a single axis allow for redundant calculations to take place, which helps reduce the uncertainty in the determination of the angle.

In the microphone array, one microphone is considered to be a point of reference and the sound observed on that microphone is used as a base comparison for the sounds observed on the other microphones. Therefore to find the point of origin of an incoming sound, two microphones are needed to locate sound in planar system, and three or more microphones are needed to locate a sound in a three dimensional space. However, in this study we restrict attention to locating an impulsive sound emanating from a point source that is geographically located in a known plane. Fig. 1 shows how the horizontal approach angle is determined. This determination is valid if the distance from the point of origin of the sound is much greater than the distance between the two microphones. For this experiment, the microphones are places 10 cm apart. The 10 cm distance allows sounds originating as close as 10 m away to be located with an

error margin of less than 5 percent. The sounds captured for this experiment are listed below in Table 1. More tests have been run and documented in Appendix K. These tests were sampled at 96 kHz and run over a larger spectrum of angles. The three sounds used in the experiment originated 10 m from the two microphone array.

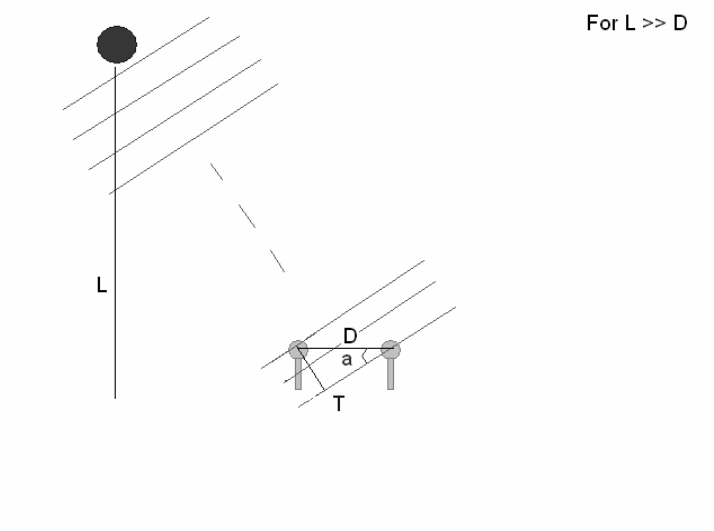

Fig.1. Diagram of Incoming sound waves to acoustic array. **L** is the distance of the origin of the sound from the microphone array. **D** is the distance between the two microphones used in the comparison. **T** is the calculated distance the sound wave travels after reaching the first microphone before the second microphone detects it. The angle of approach of the sound wave is **a**.

Table 1. List of Sound Files and Their Properties

File name	Type of Sound	Distance	Angle
clap.wav	hand clap	10 m	110°
clap1.wav	hand clap	10 m	70°
clap2.wav	hand clap	10 m	75°

Data Capture and Offset Removal

A threshold is used to determine when a signal is to be processed. The signals from each microphone array are stored in circular buffers hat hold 1000 samples; that is once 1000 samples have been stored the next sample overwrites the first position in the buffer. When a sample's value reaches a threshold value, a start signal is sent to the

computational portion of the system and 500 more samples are written to the buffer. After those 500 samples are written, the data in the buffer is held and no further data is written to the buffer. The signal from the microphone that is used as reference is the only signal where threshold value is monitored. The other microphone signals are written to similar buffers that are controlled by the reference signal; they start and stop writing when the reference signal starts and stops writing.

The first portion of the computations is an offset removal method. The incoming data contains a direct current (DC) offset that can cause the correlation used later to drift, the value constantly increases when it should remain constant. The removal of the DC offset also allows for the assumption of a zero mean signals, which is needed for the correlation methods to follow. Table 2 shows the effect the DC offset can produce on a correlation versus the correlation with the absence of a DC offset. The signals x and y represent impulse- like signals with short peaks that have no DC offset. The signals xn and yn are signals x and y with a small DC offset, 0.01, added to the signals. The offset removal is accomplished by finding the average value of the signal and then subtracting that average value from each sample of the signal.

Correlation

After the data has the DC offset removed, a correlation method is used. A typical discrete time correlation uses the entire signal length; however, this is too long computationally to be completed in a real time setting. The sample auto-correlation of a zero mean sequence is defined as:

$$R_{xx}(n) = \sum_{k=-\infty}^{\infty} x(k)x(n+k) \quad . \tag{1}$$

Table 2. Effect of DC Offset on Correlation

Without Offset			With Offset		
x	y	Correlation	xn	yn	Correlation
0	0	0	0.1	0.1	0.0001
0	0.5	0	0.1	0.6	0.0002
0.5	0.5	0	0.6	0.6	0.0053
0.5	0	0	0.6	0.1	0.0154
0	0	0	0.1	0.1	0.0205
0	0	0.25	0.1	0.1	0.2706
		0.5			0.5205
		0.25			0.2704
		0			0.0153
		0			0.0052
		0			0.0001

The sample cross-correlation of two zero mean sequences is defined as:

$$R_{xy}(n) = \sum_{k=-\infty}^{\infty} x(k)y(n+k) \quad . \tag{2}$$

$R_{xx}(n)$ is the n^{th} lag of the auto-correlation and $R_{xy}(n)$ is the n^{th} lag of the cross-correlation

Two measures have been taken to speed up the correlation. The first is the buffer used to store the incoming signal. The buffer gives the signals to be correlated a known length, 1000. The 1000 sample size was chosen since an impulse-like sound, such as a handclap or a gunshot, can be contained within the 1000 samples for most sampling rates. At rates of 44.1 kHz and 96 kHz the sound contained in the buffer is 22 milliseconds and 10.5 milliseconds, respectively. The reduced size of the signal reduces the possible number of multiplies in each pass of the correlation from *n* multiplies needed, where *n* is the number of samples in the signal, to 1000 multiplies needed.

The second method to speed up the correlation is to reduce the number of multiplies and additions in the method. In other words, the correlation could be limited in the number of shifts since there is a physical maximum delay in the system. This maximum delay occurs when the sound's origin is along the line created by two microphones, i.e. in Fig. 1 if the sound origin was on the line labeled D. This maximum delay was found to be 20 samples at a sampling rate of 44.1 kHz. With a known maximum delay in the system, the correlation could be reduced to fit about this delay. The features of the signals from the other microphones match the reference signals features no more than 20 samples before or after they occurred in the reference signal. This reduced the number of lags needed in the correlation from 2000 to 40. This reduction in the number of lags created an approximation of the sample auto-correlation and cross-correlation defined as the following:

$$R_{xx}(n) \cong \sum_{k=-20}^{20} x(k)x(n+k) \quad . \tag{3}$$

$$R_{xy}(n) \cong \sum_{k=-20}^{20} x(k)y(n+k) \quad . \tag{4}$$

A comparison of the sample cross-correlation with the approximations of the sample cross-correlation can be seen in Fig. 2. The upper left plot is the sample cross-correlation used in MATLAB, and the lower left plot is a zoom of the same cross-correlation about the 980^{th} to 1020^{th} lags. The lower right plot shows the results of the approximation of the sample cross-correlation. A visual inspection of the two plots shows that there are few discrepancies between them. A plot of the difference of the two correlations, upper right plot, shows that the difference of the two is negligible.

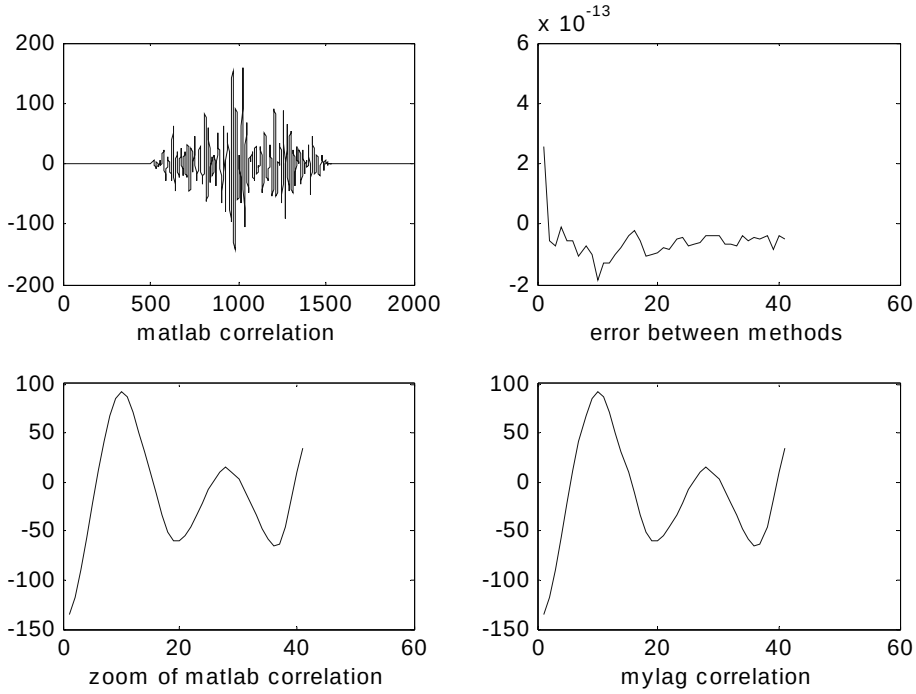

Fig. 2. Plots of Sample Cross-Correlation and Approximation of Sample Cross-Correlation

The correlations that are performed in this system include an auto-correlation of the reference signal and a cross-correlation of the reference signal with each of the other microphone signals. The auto-correlation is performed once; if more than one non-reference microphone signals are to be correlated then the same auto-correlation is used for each comparison. The comparisons of the correlations are done by a simple method to find the index values of the maximum values of the correlations. Once the index values have been found, the index value of the auto-correlation is subtracted from the other index values. This gives the number of samples each signal is delayed from the reference signal.

Trigonometry

Once all of the signal delays have been found, trigonometry is used to find the approach angle of the incoming sound based upon the physical design of the system. Firstly, the number of samples in the delay is converted to an amount of time by dividing by the sampling rate of the system. This time is then multiplied by the speed of sound to find the distance the sound wave traveled. The speed of sound was found by placing an impulse-like noise along the line created by the two microphones. The speed of sound was calculated using the known distance between the two microphones and the sample delay to create a time delay. As shown in Fig. 1, the legs D and T of the triangle shown have been found and the third leg is calculated using the Pythagorean Theorem. This step is needed since an arctangent is used to calculate the angle rather than an arcsine. The arctangent gives a greater numerical accuracy in the range of $-\pi/2$ to $\pi/2$, as well as giving the ability to locate which quadrant the angle is in.

Two limitations of this design were its angular resolution and its range of angle detection. The resolution of the system was limited by the sampling rate used and the distance between the microphones. For the system used in this experiment, 44.1 kHz sampling and 25 cm between microphones, the resolution is 1.73°. The range of the system is directly related it its resolution. The correlations used can detect a delay of up to 20 samples, which resulted in a range of 69.32°. The angles the system could detect ranged from 55.34° to 124.66°.

CHAPTER THREE

Hardware

Sound Acquisition

There are two sets of equipment that have been used to acquire the sounds to process. The initial hardware included a Yamaha MG10/2 mixer and two Nady SCM 1000 condenser microphones. The Yamaha MG10/2 mixer has four XLR balanced microphone inputs and several built in amplifiers [4]. However, the major short coming of the mixer was its lack of onboard analog to digital converters (ADC). In order to create a digital signal, the mixer had to be connected to a computer using RCA cables and use the computers sound card to sample the signal, which was limited to a maximum sampling rate of 44.1 kHz.

The Nady SCM 1000 condenser microphones have three polar patterns: cardioid, figure eight, and omni-directional. The polar pattern used in the experiment was the omni-directional pattern so that the microphones did not need to be rotated in order to pick up sounds from behind the microphone. The Nady microphones have a frequency response from 20 Hz to 20 kHz and sound levels of 134 dB were the maximum acceptable inputs for the microphones. The signal to noise ratio, a ratio of signal strength to noise strength in decibels, was 76 dB for the Nady microphones. The main detractor of these microphones was the size; the Nady microphones were 6.5” in length with a diameter of 2.5” and weighing almost a pound (15oz) [5].

The Yamaha mixer was replaced with a Presonus Firepod. The Firepod lacked the amplifiers that the mixer had, but its features were of more benefit to the system than the mixer's amplifiers. The Firepod has eight XLR balanced microphone inputs, each of which has its own ADC. The eight XLR inputs would allow a full array of microphones to be connected to a single card, rather than the two mixers that would be needed previously. The ADCs on the Firepod can sample at a rate of 96 kHz, twice the rate of the computer's onboard soundcard. The Firepod sends the sampled signal to the computer via an IEEE 1394 connection at up to a rate of 400 mbps [6].

Audio-Technica MT830R lavaliere microphones replaced the Nady SCM 1000 microphones. The audio specifications of the Audio-Technica microphones were strikingly similar to the Nady microphones. The frequency response of the Audio-Technica microphones was 30 Hz to 20 kHz, a minor decrease from the range of the Nady microphones. The signal to noise ratio, 70 dB, and maximum input level, 131 dB, were similar to that of the Nady microphones. The polar pattern of the Audio-Technica, shown in Fig. 3, is a constant omni-directional pattern. The sizes of the Audio-Technica microphones were .62" in length, .33" in width, and .19" in thickness with a weight of .05 ounces [7]. The much smaller size of the Audio-Technica microphones made them more appealing to use in an array than the larger Nady microphones.

Fig. 3. Polar Pattern of MT830R Microphone

DSP

A TMS320C6711 DSK board made by Texas Instruments was used for the DSP implementation. The board operates at 100 MHz, 10 ns per cycle, and is capable of completing eight 32-bit instructions per cycle. The DSP on the board has eight independent functional units: four floating point ALUs (arithmetic logic unit), two fixed point ALUs, and two fixed/floating point multipliers. The DSP has 8 KB of L1 cache and 64 KB of L2 cache. The DSK board also has an additional 16 MB of SDRAM and 128 KB of Flash memory on board for code and data storage [8]. The DSK board connected to a computer using a parallel port connection. One analog input and one analog output are included on the board.

FPGA

The Nallatech XtremeDSP board was used for the FPGA implementation. The XtremeDSP board consists of two of Nallatech's DIME-II system boards. The BenONE-Kit motherboard is configured to only link with the packaged BenADDA DIME-II module. The motherboard also contained a Spartan-II FPGA which is used

for the PCI and USB interfaces; in this experiment only the PCI interface was used. The BenADDA DIME-II module houses the Virtex-II XC2V3000 FPGA, which is the FPGA that is programmed in this experiment. The BenADDA module also has two ADC channels that sample at a rate up to 105 MHz and two DAC, digital to analog converter, channels that can operate at rates of 160 Mhz [9].

The Virtex-II FPGA included on the BenADDA board contains 96 18-bit x 18-bit multipliers, 96 18 KB block RAMs, and 3,584 combinational logic blocks (CLBs). Xilinx rates the part as the equilavent of 3,000,000 logic gates. The clocking available to the FPGA is onboard the BenADDA board can run at speeds up to 120 MHz [10].

CHAPTER FOUR

Software

System Modeling

MATLAB version 7 R14 was used to develop the initial system model. The MATLAB language has been designed for quick algorithm development. The MATLAB language doesn't require memory allocation or variable type declarations. It has built in mathematical functions and vector support. MATLAB also removes the need to compile the code in order to be run; the program or sections of the program can be run immediately after changes have been made. The previously mentioned algorithm was coded in MATLAB and tested by importing a sound file recorded by the microphone array. Plots of the sound files used in this experiment can be seen in Fig. 4, Fig. 5, and Fig. 6 as the imported digital signals in MATLAB. Once the algorithm was verified, the modified correlation method was developed. This correlation method is contained in the mylags.m file listed in Appendix B. This method was tested alongside with MATLAB's built in correlation routine. The similarities of the MATLAB language and the C language provided a basis for the code written in the DSP implementation.

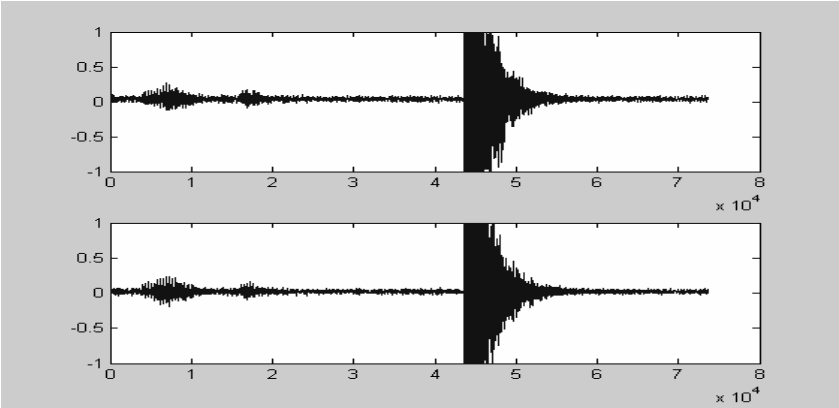

Fig. 4. Plot of clap.wav

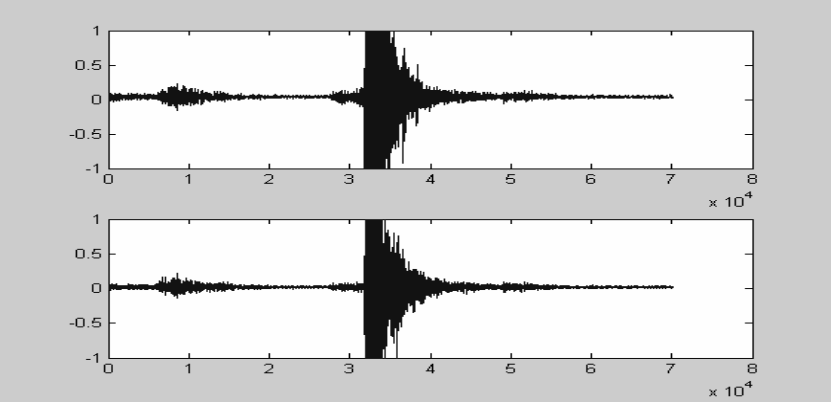

Fig. 5. Plot of clap1.wav

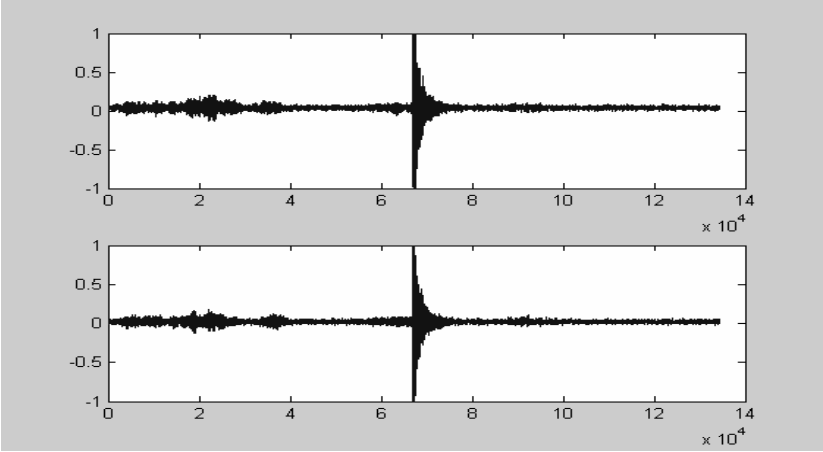

Fig. 6. Plot of clap2.wav

Another feature of MATLAB is its visual modeling program, Simulink. Simulink uses functional blocks to build and simulate systems. These blocks can be built in blocks provided with Simulink to perform common functions or blocks that import code to perform functions not provided with Simulink. The code used to build the algorithm in MATLAB was used to create a system model in Simulink using both built-in and custom blocks. These blocks were used as the basis for the different modules used in the FPGA design.

DSP

The DSP board was programmed using Code Composer Studio (CCS). CCS uses C coding to generate code for the DSP boards built by Texas Instruments. The CCS package includes a C complier, an assembler, a debugger, and a profiler. The debugging tools included allow the user to step through each assembly instruction and observe the data stored in all of the chip's registers. The profiler examines the time spent in each marked section of code as well as the number of instructions unique to each section of code.

FPGA

Xilinx ISE 6.3i and System Generator 3 were both used to develop the FPGA program. System Generator, SysGen, is an add-on for Simulink that was developed by Xilinx. SysGen contains blocks for common FPGA function such as input/output buffers, accumulators, counters, RAM, and multipliers. SysGen also has tools to view the FPGA gate usage as well as to generate or cosimulate the design. Cosimulation is

simulation where part of the simulation is run on the FPGA with the rest of the simulation run on the computer.

ISE is a platform where code in VHDL or Verilog can be written to create a FPGA design. ISE provides more freedom than SysGen when creating designs, but the effort needed to make the design is more than that needed in SysGen. ISE contains a synthesizing tool, a place and route tool, a timing analysis tool, and a bit file generation tool. The synthesizer takes the VHDL or Verilog code and calculates which parts, combinational logic gates, registers, lookup tables, etc., are needed to build the design. The place and route tool takes the part list from the synthesizer and the timing constraints and connects the parts within the FPGA to meet both specifications. The timing tool analyzes the amount of delay from a value entering to a part to when it leaves the part, the travel time of a value between parts, and other factors such as clock fan-out. The bit file generation tool takes the results from the place and route tool and creates a binary file that contains the FPGA configuration.

CHAPTER FIVE

Implementation

DSP

The DSP design begins with getting the signals from the microphones to the board. With the DSK's design, the most appropriate way to import signal is to use the CCS Probe Point system. The Probe Point system fills an array that has been defined in the code by parsing a data file and recording entries to consecutive locations in the array. In this design, two arrays of 1000 32-bit floating point numbers were declared in memory and two microphone signals were read into the arrays. The data files that contained the microphone signals only contained the 1000 samples surrounding the peak of the signal. By placing the sound data into memory, the board's lack of sufficient ADC was overcome. The rest of the algorithm could be and was implemented; this section of the algorithm was enough to test the aspects of the DSP and the CCS utilities, without taking into the shortcomings of the DSK board.

The offset removal portion of the algorithm was accomplished in the smoothing function by moving a pointer down each array and summing the values. Once all the values in each array had been accumulated, the totals were divided by 1000 to obtain the offset to be removed from each sample. Pointers were once again used to recall each sample from memory, remove the offset from the sample, and the place the new value of the sample back in the memory location the original sample came from.

The correlation that takes place in the DSP implementation is in the correlate function. This function contains two while loops. The inner loop increments a pointer to each array so that all 1000 samples in the arrays are multiplied together and then accumulated. The loop counting variable takes into account the number of shifts that have occurred so that the pointers of each array do not point to invalid data. Once the program exits the inner loop, value of that pass of the correlation is written to memory by the outer loop. The outer loop also resets the pointers to the beginning of the array and increments certain pointers to do the shifting each pass. In this implementation three correlations are done simultaneously; an auto-correlation, a correlation where it assumed the reference's features occur before the other signal's features, and a correlation where it is assumed the reference's features occur after the other signal's features. The data from each of these correlations is shown as a plot in Fig. 7. The left plot is the auto-correlation with a peak at the beginning of the plot. The right plot is the correlation between the two signals and the peak occurs later on in the plot.

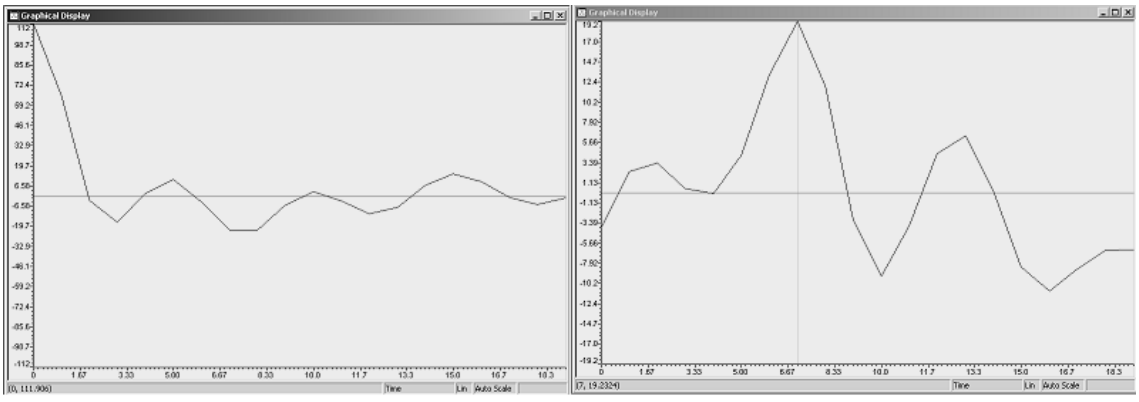

Fig. 7. Plot of Correlation from DSP Implementation. These plots show the streamed output of the auto-correlation and cross-correlation. The auto-correlation (left) shows a peak at the initial lag with a peak value of 112. The cross-correlation (right) shows a peak at the seventh lag with a peak value of 19.2.

The rest of the algorithm was combined into a single function, the findangle function. The memory holding the output from the correlation function was examined sequentially until all twenty of the output values had been examined and the maximum found. Again, all three maximums were found simultaneously. After the maximums had been found the values from the two correlations that were not the auto-correlation were compared, and the maximum of those two was used since it represented the maximum of the correlation for that signal. The arithmetic to determine the delay and finally the angle of approach of the sound were implemented, and the value of the angle was saved into memory.

FPGA

The FPGA design was implementing using SysGen with some blocks made from imported code made in ISE. In order to match the design implemented in the DSP, the system began after a peak had been detected in the reference signal. The signal was sent to the FPGA directly from MATLAB using the GatewayIn block in SysGen. This block took a double precision floating point value from MATLAB and converted it to a desired fixed point value, in this case a signed 16-bit number with 15 decimal points. After the number had been converted to a fixed point number, the signal was routed to the offset removal circuitry. This circuitry consisted of an accumulator and to a register to delay the signal until the average offset had been found. The accumulator operated continuously with a reset sent to the accumulator every 1,000 clock cycles. On the output of the accumulator a register was used to hold the $1{,}000^{th}$ value, the final sum, by triggering a clock enable on the $1{,}000^{th}$ clock. This value was then multiplied but a value of 0.001 to complete the average. By the time

the average was found, the signals started to leave the delay register to go to a subtractor that removed the average offset from each sample.

The signal then entered the correlation block, which contained imported code from ISE. The signal spent 1,000 cycles once it entered the correlation block to be written into RAMs defined in the code. The correlation block then performed 19,800 multiplies and accumulates, with an output asserted every 980 multiplies and adds. The multiples and accumulates were defined in modules to add fixed point support. The multiply module takes the signed 16-bit numbers with 15 decimal points and converted the numbers into 15-bit unsigned numbers. The two numbers were then multiplied and the 15 most significant bits of the resultant 30-bit number were saved. The number was then converted back into a signed 16-bit number based upon the exclusive or of the sign bits of both numbers. The accumulation was accomplished by shifting 1s or 0s for eight bits to each number. This created a signed 16-bit number with 7 decimal points. The right shift by eight bits was to ensure that the eventual sum of 980 numbers would not cause an overflow. The sign bit of the input number was used to determine whether to add or subtract the value from the current sum. At this stage, there were two processes operating, an auto-correlation and a signal lags reference correlation. Plots of the outputs of the two correlations can be seen in Fig. 8. Like the plots in Fig. 6, the plots here are the auto-correlation, dashed, showing a peak at the beginning and the correlation between the two signals, solid with a peak after several passes of the correlation.

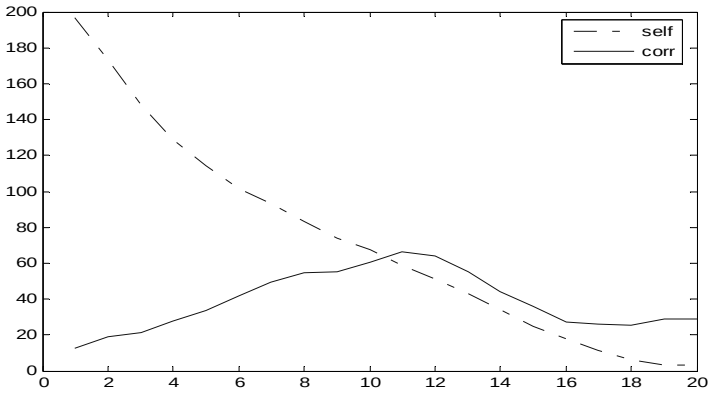

Fig. 8. Plot of Correlation from FGPA Implementation. These plots show the streamed output of the auto-correlation and cross-correlation. The auto-correlation (dashed) shows a peak at the initial lag with a peak value of 200. The cross-correlation (solid) shows a peak at the eleventh lag with a peak value of 74.3.

The output of the correlation blocks were fed to blocks also written in ISE that held the maximum values it was given. These blocks were examples of a simple function that was not included in the SysGen blockset and was added to customize the SysGen library. The held value from the "findmax" block was sent to trigonometry subsystem of the FPGA design. The values from the two "findmax" blocks were subtracted so that the value from the auto-correlation was subtracted from the signal lags reference correlation value. This new value was then multiplied by the reciprocal of the sampling rate and by the speed of sound. That output was fed as both inputs of a 16-bit multiplier to square it, and the subtracted from the square of the distance between the microphones. A CORDIC square root block was then used to compute the square root so value of the third leg of the triangle shown in Fig. 1 could be known. The CORDIC square root block, like the CORDIC arctangent block used next, are IP cores included in the SysGen blockset library. An IP core is a module that generally fulfills a high level function that was not originally designed by Xilinx. Once the

square root block is finished, its output is sent to the Y input of a CORDIC arctangent block. The X input of this block is the value was the value of difference of the correlations after they had been scaled by the sampling rate and the speed of sound.

CHAPTER SIX

Conclusions

The DSP system was tested using the three sound files listed in Table 1. The DSP found the three angles to be: 109.46°, 72.36°, and 77.75° respectively. The results of each test, sampled at each pass of the correlation, are plotted in Fig. 9. In each of the tests, the calculated angle was within a five percent margin mentioned previously. The FPGA system was tested with the three sound files from Table 1 as well. The FPGA located the sounds at 111.32°, 72.66°, and 77.03° respectively. Plots of the angles found by the FPGA are shown in Fig. 10; the plots are sampled after each pass of the correlation since the value remains the same throughout a pass. Both Fig. 9 and Fig. 10 show the determination of the angle as time passes.

In terms of performance, the FPGA outperformed the DSP by a wide margin. The FPGA took 23005 clock cycles to produce a final answer running at a clock rate of 40 MHz. The DSP took 25725060 clock cycles to produce its final answer running at a clock rate of 100 MHz. This resulted in operating times of 0.575 ms and 257.3 ms respectively, a difference of 256.7 ms. A time difference in favor of the FGPA was expected due to its increased capabilities to perform complex calculations as well as its increased processing speed.

The other focus of this experiment was to see how the available tools affected the design development time and the design performance. For a comparison of design, a metric of code length, number of lines, was used. The DSP code, shown in Appendix C, totaled to 86 lines of code. In order to get a lines of code total for the FPGA design,

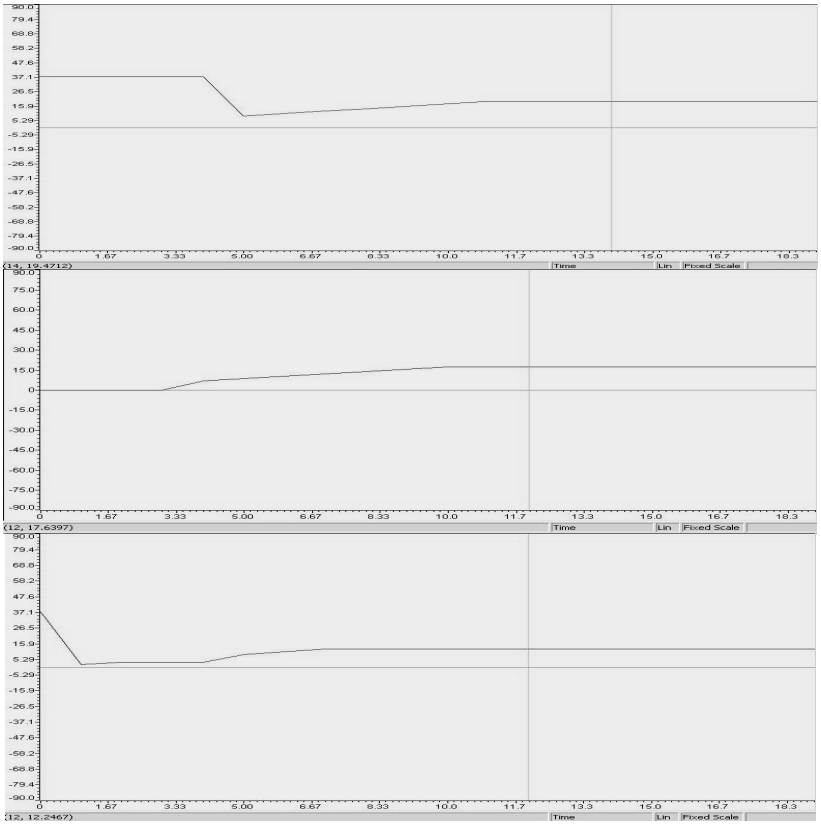

Fig. 9. Plots of the DSP Determination of Angles for Sounds Listed in Table 1. These plots show the output of the trigonometry potion of the code after each pass of the correlation.

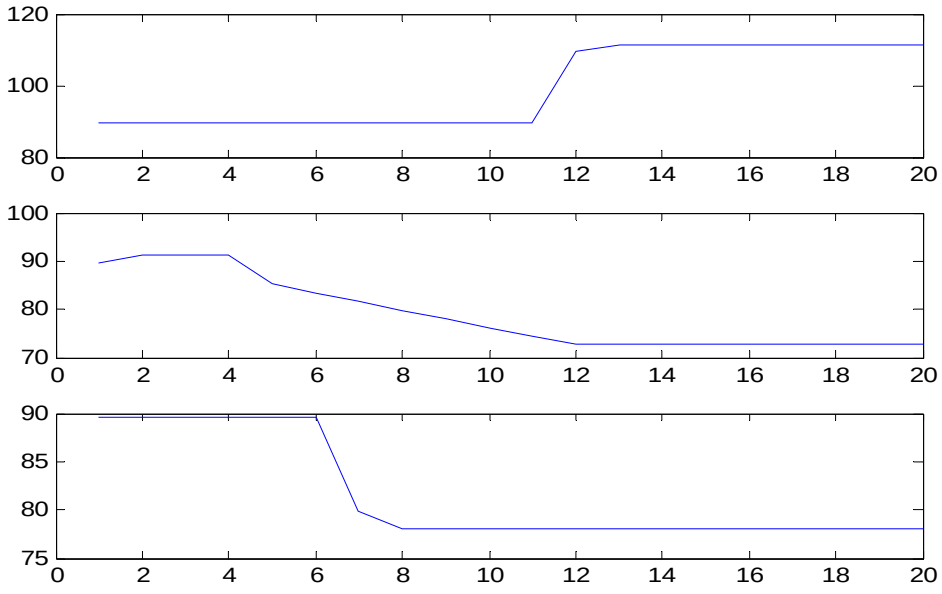

Fig. 10. Plots of the FPGA Determination of Angles for Sounds Listed in Table 1. These plots show the output of the trigonometry potion of the code after each pass of the correlation.

an assumption must be made about the blocks used in the design. The blocks, while all varying in actual code size to define there function, all require the user to know three basic details to instantiate the block: input size, output size, and function. Since the developer does not need to know the code used to generate the block, but the port sizes and function of the block, each block was assigned a code length of three lines. The FPGA design line count totaled to 198 lines of code, nearly twice that of the DSP design. In the DSP design, the CCS tool was the sole tool used to make the design. As mentioned earlier, the CCS program takes C code and creates an assembly program to send to the DSP. This is one of the CCS program's strengths; it uses a widely known language with many years of support refining it. The familiarity of the C language with most signal processing designers greatly reduces the time needed to create a design for the DSP. The debugging process in CCS went smoothly as well. Any errors that were found were clearly identified. Also, the data could be monitored through each step of the program easily using the Watch Window function of the debugger. The profiler of CCS allows the user to see how the complier uses the code to generate an assembly program to implement on the DSP. The results of the profiler can be seen in Table 3. The table shows that most of the codes operation time is spent performing the correlation, while minimal time is needed to calculate the angle once the correlation has completed.

Table 3. Number of Cycles per Section of Code in DSP Implementation

	Code Size	Inclusive Cycles	Exclusive Cycles
main	288	25725060	527
smoothing	428	1515318	1511694
correlation	580	24175314	24175314
findangle	592	33668	23834

The FPGA design was created using both SysGen and ISE. The portions of the design created in SysGen were developed quickly. The act of placing blocks and connecting them with wires greatly reduced development time compared to coding each block and routing the signals using Verilog. A problem with using the blocks is the limited functionality in the block set. Although many functions have been made into blocks, some functions, such as correlation, have not been included as blocks. SysGen did provide for expansion of the block set by giving the developers a block to incorporate Verilog code into a custom block. The customizable "black box" does allow developers to add new functionality to the block set; the "black boxes" do not have the same support as the rest of the blocks in SysGen. While fixed point numbers are supported by the blocks that came with SysGen, the "black boxes" must have fixed point support coded into each file. Also, in design simulations the blocks native to SysGen do not require external simulation. The "black boxes" require a call to an external simulator or to be run in a hardware co-simulation in order for any outputs to be generated.

The error reporting in SysGen is cryptic at best. Since the code is embedded into blocks, it is difficult to point to a specific point in the design where an error occurred. Often times, errors are noted to have occurred and the design simulation halts. Another problem when testing designs in SysGen is the inability to view any signal as the design runs. The sole way to watch signals that are not already outputs is to add another output connected to the signal needed to be watched. In order to view the device usage of the design, the designer must find a results file deep within the hierarchy of folders created by SysGen in the design creation. SysGen utilizes the

same tools as ISE to translate the user's design into a hardware implementation. Calls to the synthesis, place and route, timing analysis, and bit file generation tools are made and recorded in this log file. Table 4 shows the device utilization from the log file generated by the FPGA implementation.

Table 4. Part Usage of FPGA Implementation

Part	Number Used	Number Available
External IOB	110	484
MULT18X18	3	96
RAMB16	4	96
SLICE	3123	14336
BUFGMUX	2	16
TBUF	224	7168

The portion of the FPGA design written in ISE was the "black box" section in the SysGen design. The code was written in Verilog. The Verilog language is a powerful language, but requires more effort to produce a design than a design made in C would need. The error reporting in ISE was similar to that found in CCS, but some errors were not easily identified. The signals used in ISE were able to be monitored even if they were not outputs. This allowed for an easier method to ensure the design was functioning properly. The device utilization reports, the same ones as SysGen uses, were much easier to access in ISE. The reports were displayed as each tool completed its process.

Overall, despite the ability to produce a faster implementation in the FPGA the DSP was still a much faster and more streamlined process to create the design. The FPGA design could be quickly laid out in SysGen, but the testing and debugging time overshadow the time saved by developing the system visually. The time needed to create the design using solely ISE was also much greater than the time needed when

using CCS for the DSP. This is due to the amount of detail that must be observed when coding a hardware design in ISE. With continued improvements on SysGen's functionality, the FPGA should not only perform the signal processing application faster than the DSP but also have a shorter developmental stage.

APPENDICES

APPENDIX A

Findangle.m

```
function theta=findangle

sos=1095.81818181818;
micd=9.84/12;

%Reads claps in and seperates the channels

[y fs]=wavread('h:\simulink\clap.wav');
[m p]=max(y(:,1));
temp=1:1000;
temp=temp';
yright=y(p-500:p+499,2);
yrm=yright-mean(yright);
yleft=y(p-500:p+499,1);
ylm=yleft-mean(yleft);

%Does correlations to determine time delay

% [clead, clag]=mylag(yrm,yrm,20);
% [m1,p1]=max(clead);
% [clead, clag]=mylag(yrm,ylm,20);
% [m3,p3]=max(clead);
% [m4,p4]=max(clag);
% maxv=max(m3,m4);
% if (maxv == m3)
%     p2 = p3;
% else p2 = p4; end
% td=p2-p1;

ini=xcorr(yrm,yrm);
[m1,p1]=max(ini);
subplot(2,1,1);plot(ini);
fin=xcorr(yrm,ylm);
[m2,p2]=max(fin);
subplot(2,1,2);plot(fin);
td=p2-p1;

%Calculates angle
time=td/fs;
```

```
dist=time*sos;
temp=sqrt(micd^2-dist^2);
ratio=dist/temp;
theta=atan2(temp,dist);
theta=theta*180/pi;
```

APPENDIX B

mylag.m

```
function [clead, clag]=mylag(x,y,Mlags)

% This routine does correlation of x with y over a limited range of lag
% values.
% Use:  [clead clag]=mylag(x,y);
% x and y are input vectors;
% clag returns the nonnormalized "lag" estimates for the correlation with the
interpretation that x lags y.
% clead returns the nonnormalized "lead" estimates for the correlation with
% the interpretation that x leads y.

[xx yx]=size(x);
[xy yy]=size(y);
if ((xx>1) & (yx>1) )|((xy>1) & (yy>1))
   error('x and y must be vectors');
end

if yy==1
   y=y';
end

if yx==1
   x=x';
end


Ny=length(y);
Nx=length(x);

if Nx>Ny
   y=[y zeros(1,Nx-Ny)];
end

if Nx<Ny
   x=[x zeros(1,Ny-Nx)];
end

N=max(Nx,Ny);
```

```
for M=0:Mlags;
    kx=1:N-M;
   dex=M+1;
    ky=M+1:N;
    clead(dex)=sum(x(kx).*y(ky));
end

for M=0:Mlags;
    kx=M+1:N;
    ky=1:N-M;
    dex=M+1;
    clag(dex)=sum(x(kx).*y(ky));
end
```

APPENDIX C

volume.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "volume.h"

float yleft[BUFSIZE];       /* processing data buffers */
float yright[BUFSIZE];
float self[20];
float corr[20];
float test[20];
float angle[20];

extern void load(unsigned int loadValue);
void smoothing(float *input, float *input1);
void correlate(float *input, float *input1, float *output, float *output1, float *test);
float findangle(float *output, float *output1, float *test, float *tht);
static void dataIO(void);

void main(){
   int a = 0;
   float *input = yleft;
   float *input1 = yright;
   float *output = self;
   float *output1 = corr;
   float *test = test;
   float *tht = angle;

   while(a<1){
     dataIO();
          dataIO();
          smoothing(input, input1);
          correlate(input, input1, output, output1, test);
          *angle = findangle(output, output1, test);
          a++;
   }
}

void smoothing(float *input, float *input1){
         float offset = 0;
```

```
        float offset1 = 0;
        int n = 0;
        for(n = 0; n < 1000; n++){
                offset = offset + input[n];
                offset1 = offset1 + input1[n];
        }

        offset = offset / 1000;
        offset1 = offset1 / 1000;

        for(n = 0; n < 1000; n++){
                input[n] = input[n] - offset;
                input1[n] = input1[n] - offset1;
        }

}

void correlate(float *input, float  *input1, float  *output, float *output1, float *test){
   int i = 0;
   int n = 0;
   int size = BUFSIZE;
   float *cool = input;
   float *cool1 = input1;
   float temp, temp1, temp2;

        while(n<20){
                temp = 0.0;
                temp1 = 0.0;
                temp2 = 0.0;
                while(i<size-n){
                        temp = temp + input[i] * input1[i];
                        temp1 = temp1 + input[i] * cool[i];
                        temp2 = temp2 + cool[i] * cool1[i];
                        i++;
                }
                output[n] = temp1;
                output1[n] = temp;
                test[n] = temp2;
                n++;
                input++;
                cool1++;
                i=0;
        }
}

float findangle(float *output, float *output1, float *test, float *tht){
```

```
        int n = 0;
        int pos1, pos2, pos3 = 0;
        float max1, max2, max3, delay, value = 0.0;

        while(n<20){
                if(output[n] > max1) {
                        max1 = output[n];
                        pos1 = n;}
                if(output1[n] > max2) {
                        max2 = output1[n];
                        pos2 = n;}
                if(test[n] > max3) {
                        max3 = test[n];
                        pos3 = n;}
                if(max3 > max2) pos2 = pos3;

                delay = 1095.81818181818*(pos2 - pos1)/44100;
                value = sqrt(.82*.82 - delay*delay);
                tht[n] = 180*(atan(delay/value))/3.1416;
                n++;
}
```

APPENDIX D

findanglewithcosim.m

```
clear
sos=1095.81818181818;
micd=9.84/12;
rst=[zeros(1,1003) 1 zeros(1,19999)]';
m=1:length(rst);
rst=[m' rst];
ce=[1 zeros(1,1007)]';
c=1:length(ce);
ce=[c' ce];
[y fs]=wavread('h:\simulink\clap2.wav');
[m p]=max(y(:,1));
temp=1:1000;
temp=temp';
yright=y(p-500:p+499,2);
yleft=y(p-500:p+499,1);
yrm=[temp yright];
ylm=[temp yleft];
micd=[0 micd];
sim('fullpartssim.mdl');
tht=(theta(3987:981:end))/pi*180;
plot(tht)
tht(end)
```

APPENDIX E

SysGen Block Diagrams

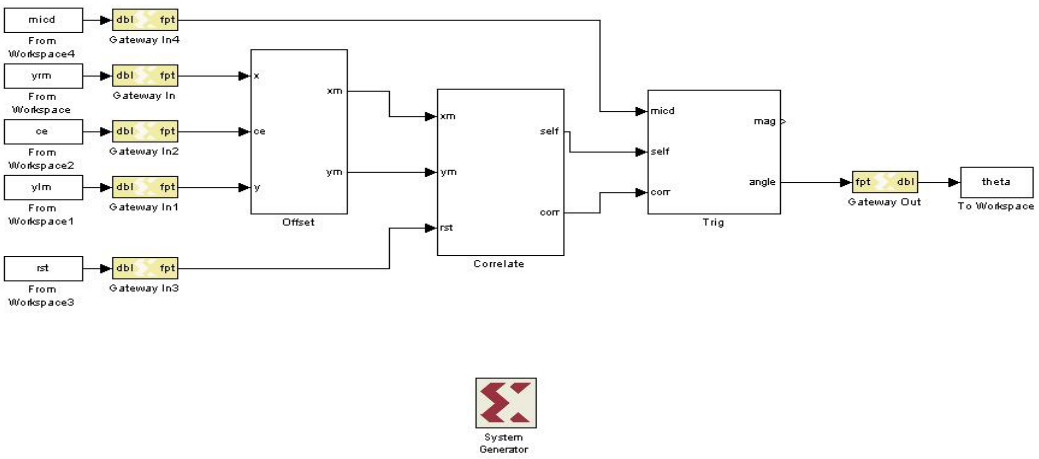

Fig. 11. Diagram of Complete FPGA system

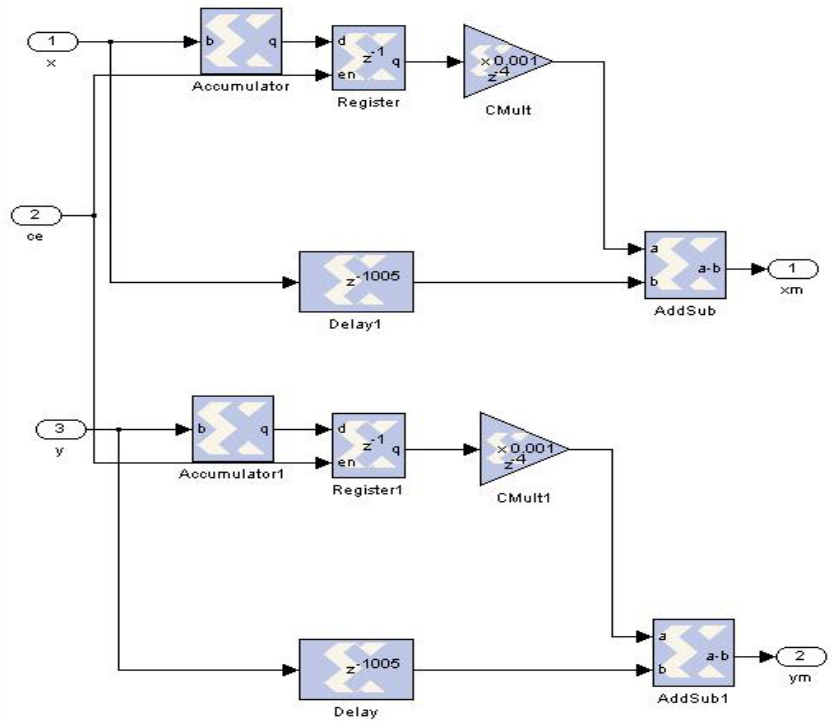

Fig. 12. Diagram of the Offset Removal

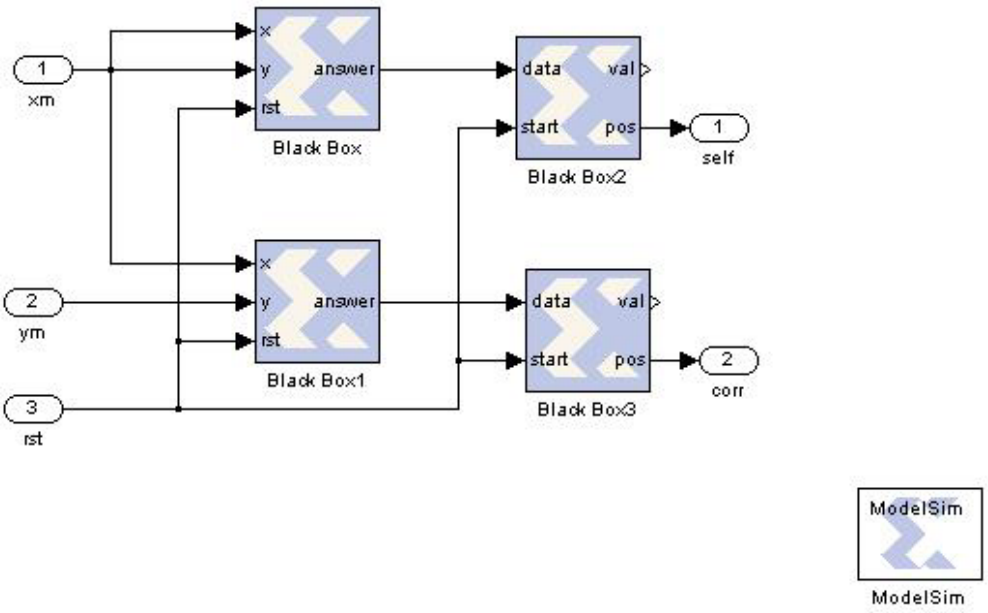

Fig. 13. Diagram of Correlation and Maximum Value Search

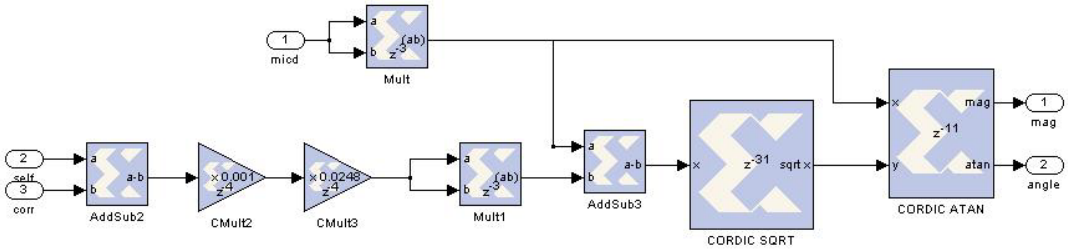

Fig. 14. Diagram of the Trigonometry Section

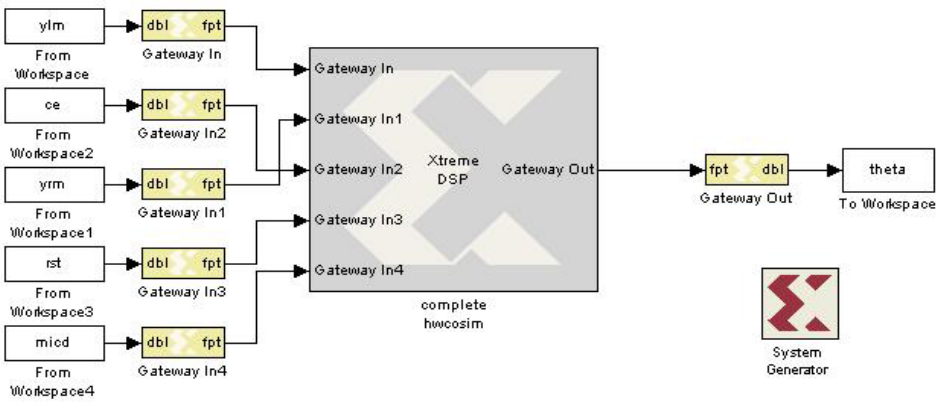

Fig. 15. Diagram of Co-Simulation System

APPENDIX F

correlate.v

```
module correlate(x,y,clk,ce,rst,answer);
	input	[15:0]	x,y;
	input	clk,rst,ce;
	output	[15:0]	answer;
	reg	we,arst;
	reg	[9:0]	addrx,addry,count;
	reg	[15:0]	answer;
	wire	[15:0]	xt,yt,temp,corred;

block_ram xram(clk, we, addrx, x, xt);
block_ram yram(clk, we, addry, y, yt);
fixedpt mult(clk, ce, xt, yt, temp);
fixedptacc acc(temp, corred, clk, ce, arst);

always@(posedge clk) begin
	if(rst) begin
		we<=1;
		count<=1;
		addrx<=0;
		addry<=0;
		arst <= 1; end
	else begin
		arst <= 0;
		if(we==0) begin
			if(count<20) begin
				if(addrx<980) begin
					addrx<=addrx+1;
					addry<=addry+1; end
				else begin
					count<=count+1;
					addrx<=0;
					addry<=count;
					answer<=corred;
					arst <= 1; end  end
			else
				we<=1; end end
		if(we==1) begin
			if(addrx<999) begin
				addrx<=addrx+1;
```

```
                addry<=addry+1; end
            else
                if (addrx>=999)begin
                    we<=0;
                    addrx<=0;
                    addry<=0;
                    count<=0; end end end
endmodule
```

APPENDIX G

block_ram.v

```
module block_ram (clk, we, a, di, do);

parameter N = 16;  // Data Bus Width
parameter M = 10;  // Address Bus Width


 input       clk;
 input       we;
 input  [9:0] a;
 input  [15:0] di;
 output [15:0] do;

 reg    [15:0] ram [1023:0];
 reg    [9:0] read_a;

 always @(posedge clk) begin
         if (we)
                 ram[a] <= di;
         read_a <= a;
 end

 assign do = ram[read_a];

 endmodule
```

APPENDIX H

fixedpt.v

```
/*
*         Inputs are 2 16 bit fixed point numbers with
*         1 sign bit, 15 bits for fractions
*         Output is a 16 bit fixed point number with
*         1 sign bit, 15 bits for fractions
*/

module fixedpt(clk,ce,a,b,c);
        input    clk,ce;
        input    [15:0] a,b;
        output [15:0] c;

        wire sign_bit;
        wire [29:0] temp;
        wire [14:0] fract_bits,a_fract,b_fract;

        assign sign_bit = a[15] ^ b[15];
        assign a_fract = a[15] ? ~(a[14:0]-1) : a[14:0];
        assign b_fract = b[15] ? ~(b[14:0]-1) : b[14:0];
        assign temp = a_fract * b_fract;
        assign fract_bits = temp[29:15];
        assign c = sign_bit ? {sign_bit,~fract_bits + 1} : {sign_bit,fract_bits};

endmodule
```

APPENDIX I

fixedptacc.v

```
module fixedptacc(a,sum,clk,ce,rst);
	input clk,ce,rst;
	input	[15:0] a;
	output [15:0] sum;
	wire sign_bit;
	wire [15:0] temp,temp1;
	reg [15:0] sum;

	assign sign_bit = a[15];
	assign temp = sign_bit ? {8'b11111111, a[15:8]} : {8'b00000000, a[15:8]};
	assign temp1 = sign_bit ? ~(temp - 1) : temp;


always@(posedge clk) begin
	if(rst) begin sum <= 16'b0000000000000000; end
	else
		if(sign_bit) sum	<= sum - temp1;
		else sum <= sum + temp1;
end


endmodule
```

APPENDIX J

findmax.v

```
module findmax(data,clk,ce,start,pos,val);
	input	[15:0]	data;
	input	clk,start,ce;
	output	[15:0]	val;
	output	[15:0]	pos;
	reg	[15:0]	count,pos;
	reg	[15:0]	val;
	wire [15:0] temp;

assign temp = data[15] ? ~(data[15:0]-1) : data[15:0];

always@(posedge clk)
begin
	if(start) begin
		val<=0;
		pos<=0;
		count<=0;
	end
	else begin
		if(temp>val) begin
			val<=temp;
			pos<=count;
		end
		count<=count+1;
	end
end
endmodule
```

APPENDIX K

Test samples at 96 kHz

Table 5. Test samples and results for sounds sampled at 96 kHz

File name	Sound	Sampling Rate	Angle	Distance	FPGA	DSP
capgun_60	capgun	96000	60	10 m	59.7135	60.8723
capgun_65	capgun	96000	65	10 m	65.2902	65.3307
capgun_70	capgun	96000	70	10 m	70.0727	69.8974
capgun_75	capgun	96000	75	10 m	74.8553	75.6528
capgun_77	capgun	96000	77	10 m	77.2465	77.0963
capgun_80	capgun	96000	80	10 m	80.4349	79.8943
capgun_85	capgun	96000	85	10 m	85.2175	85.3347
capgun_90	capgun	96000	90	10 m	89.2029	89.9956

BIBLIOGRAPHY

[1] Michael D. Ciletti, *Advanced Digital Design with the Verilog HDL*, New Jersey: Prentice Hall, 2003.

[2] Nasser Kehtarnanaz and Burc Simsek, *C6x-Based Digital Signal Processing*, New Jersey: Prentice Hall, 2000.

[3] Amy Malagamba, November 2005, "Assemble All Ye IP," *FPGA and Structured ASIC Journal*, [Online]. Available: http://www.fpgajournal.com/articles_2005/pdf/20051115_ip.pdf.

[4] *Yamaha Mixing Console MG10/2 Owner's Manual*, Yamaha Corporation, 2003, [Online]. Available: http://www2.yamaha.co.jp/manual/pdf/pa/english/mixers/MG10_2E.pdf.

[5] *Nady SCM Series Users Guide*, Nady Systems Inc., Application Note, November 15, 2000, [Online]. Available: http://www.nadywireless.com/pdf_files/manual_pdf/SCM1000UserG.pdf.

[6] *Firepod User's Manual*, Version 1.0, PreSonus Audio Electronics, 2004, [Online]. Available: http://www.presonus.com/pdf/fpmanual.pdf.

[7] *MT830R Miniature Omnidirectional Condenser Microphone*, Audio-Technica U.S., Inc., 2004, [Online]. Available: http://www.audio-technica.com/cms/resource_library/literature/86eb9a77d592ff8a/mt830r_english.pdf.

[8] *TMS320C6711, TMS320C6711B, TMS320C6711C Floating-Point Digital Signal Processors*, Texas Instruments Inc., 2005, [Online]. Available: http://focus.ti.com/lit/ds/symlink/tms320c6711.pdf.

[9] *XTremeDSP Development Kit Pro User Guide*, Nallatech Limited, 2005, [Online], Available: http://www.xilinx.com/products/boards/files/xtremedsp_dev_kit_user_guide.pdf.

[10] *Virtex-II Platform FPGAs: Complete Data Sheet*, Xilinx Inc., 2005, [Online]. Available: http://direct.xilinx.com/bvdocs/publications/ds031.pdf.