Florida State University Libraries

Electronic Theses, Treatises and Dissertations

The Graduate School

2009

Sparse FIR Filters and the Impact on FPGA Area Usage

Sean G. Patronis

FLORIDA STATE UNIVERSITY
FAMU-FSU COLLEGE OF ENGINEERING

SPARSE FIR FILTERS AND THE IMPACT ON FPGA AREA
USAGE

By

SEAN G. PATRONIS

A Thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2009

Copyright © 2009
Sean Patronis
All Rights Reserved

The members of this Committee approve the Thesis of Sean G. Patronis defended on December 3, 2008.

Linda DeBrunner
Professor Directing Thesis

Victor DeBrunner
Committee Member

Bruce A. Harvey
Committee Member

Approved:

Victor DeBrunner, Chair, Department of Electrical and Computer Engineering

Dean Ching-Jen Chen, Dean FAMU FSU College of Engineering

The Office of Graduate Studies has verified and approved the named committee members.

TABLE OF CONTENTS

LIST OF TABLES.....	v
LIST OF FIGURES.....	vi
ABSTRACT.....	viii
1. INTRODUCTION.....	1
1.1 Sparse FIR Filters.....	2
1.2 Sparse Filters in FPGAs.....	3
1.3 Xilinx Virtex-4 Design.....	3
1.4 ASICs vs. FPGAs.....	5
2. PREVIOUS RESEARCH.....	7
2.1 Delta-Sigma Converters.....	7
2.2 NRL Loop Filter Research.....	9
3. TOOLS USED.....	11
3.1 Mathworks Matlab®.....	11
3.1.1 Function Value Generator Functions: fv_sin.m and fv_rand.m.....	11
3.1.2 The Count Function.....	16
3.1.3 Sparseness Checker Function	17
3.1.4 Filter Mask Generator Function.....	19
3.1.4a Filter Type 1 – Evenly Spaced Zeros.....	19
3.1.4b Filter Type 2 – Linear Increasing Zeros Middle.....	22
3.1.4c Filter Type 3 – Linear Increasing Ones Middle.....	25
3.1.5 Coefficient File Generator Functions.....	28
3.1.5a Generate Sine Based Coefficients.....	28
3.1.4b Generate Random Coefficients.....	30
3.2 PHP Hypertext Preprocessor.....	31
3.2.1 VHDL Generator	31
3.3 Xilinx ISE.....	34
4. APPROACH.....	35
5. RESULTS.....	38
6. CONCLUSION AND FUTURE WORK.....	49
APPENDIX A.....	51
1. EXAMPLE VHDL FILE CREATED BY VHDLGEN.....	51
APPENDIX B.....	53

1. EXAMPLE SYNTHESIS REPORT.....	53
APPENDIX C.....	55
1. Data Tables	55
REFERENCES.....	60
BIOGRAPHICAL SKETCH.....	62

LIST OF TABLES

Table 1.1 – CLB contents.....	4
Table 2.1 – Previously designed sparse loop filter coefficients.....	8

LIST OF FIGURES

Figure 1.1 – CLB and Slice Layout [2].....	4
Figure 2.1 – Simple First Order Delta-Sigma Modulator.....	8
Figure 2.2 – Simple First Order Digital Delta-Sigma Modulator	8
Figure 2.3 – Frequency response of the NRL loop filter with 10 coefficients [9].....	10
Figure 3.1 – fv_sin.m code	12
Figure 3.2 – sin function used in fv_sin.....	13
Figure 3.3 – output of function fv_sin(ord).....	14
Figure 3.4 – fv_rand.m code.....	15
Figure 3.5 – output of function fv_rand(ord).....	16
Figure 3.6 – Count.m code.....	17
Figure 3.7 – sparseness_checker.m code.....	18
Figure 3.8 – filter_mask_gen	19
Figure 3.9 – Code snippet for filter mask type 1, evenly spaced zeros from left to right.....	20
Figure 3.10 – The output of Matlab function filter_mask_gen(1, 20, 0.45).....	21
Figure 3.11 – The output of Matlab function filter_mask_gen(1, 20, 0.75).....	22
Figure 3.12 – Code snippet for filter mask type 2, linear spaced zeros middle.....	23
Figure 3.13 – Sequence used for creating ft2 and ft3 masks.....	24
Figure 3.14 – The output of Matlab function filter_mask_gen(2, 20, 0.75).....	25
Figure 3.15 – Code snippet for filter mask type 2, linear spaced ones middle.....	25
Figure 3.16 – The output of Matlab function filter_mask_gen(3, 20, 0.75).....	27
Figure 3.17 – Code for Generating Sin Based Coefficient files.....	29
Figure 3.18 – Code For Generating Random Based Coefficient Files.....	30
Figure 3.19 – VHDLGen Run File Example.....	32
Figure 4.1 – Approach Work Flow	35
Figure 4.2 – Sparse Filter Design Filter Type Examples.....	36
Figure 5.1 – Filter Type 1 (evenly spaced zeros) Element Usage.....	38
Figure 5.2 – Filter Type 2 (Increasing zeros center) Element Usage.....	39

Figure 5.3 – Filter Type 3 (Increasing ones center) Element Usage.....	39
Figure 5.4 – LUT Usage For All Three Filter Types.....	40
Figure 5.5 – Slice Flip-flop Usage For All Three Filter Types.....	41
Figure 5.6 – Total Slices Used For All Three Filter Types.....	42
Figure 5.7 – DSP48 Slices Used For All Filter Types.....	43
Figure 5.8 – Slice vs DSP48 Utilization.....	44
Figure 5.9 – Filter Type 1 Random Coefficients.....	45
Figure 5.10 – Filter Type 2 Random Coefficients.....	45
Figure 5.11 – Filter Type 3 Random Coefficients.....	46
Figure 5.12– Slices Used For All Three Filter Types Random Coeffs.....	46
Figure 5.13– Slice Flip Flops Used For All Three Filter Types Random Coeffs.....	47
Figure 5.14– LUTs Used For All Three Filter Types Random Coeffs.....	47
Figure 5.15 – Virtex 4 LX Element Usage.....	48

ABSTRACT

In FIR filter design, a sparse filter is one that has a majority of zeros for coefficients. Generally, a sparse filter is designed in order to save area, and maybe speed up computations, but to implement the same sparse filter in an FPGA, the same area savings may not be realized. The research conducted in this work concentrated on creating sparse FIR filters where groups of non zero coefficients replaced by zeros were synthesized on Xilinx Virtex-4 FPGAs. Custom Matlab® functions were written in which different patterns of zero placements were populated on top of a base set of coefficients. These coefficient files were then used with custom PHP based code to generate VHDL files defining FIR filters. The FIR filters built in VHDL were synthesized in Xilinx ISE. After synthesis, data pertaining to area usage was then collected and analyzed. We conclude that sparsity in an FIR filter may not translate directly into FPGA space (area) savings on a Virtex-4 FPGA depending on order, sparsity and FPGA chip used.

CHAPTER 1

1. INTRODUCTION

High performance radar systems can be implemented using delta sigma converters. A core component of this delta sigma converter is the noise shaping loop filter. Inside this loop filter is a high order Finite Impulse Response (FIR) filter which requires special techniques to implement using a Field Programmable Gate Array (FPGA). Previous work at the Naval Research Laboratory (NRL) has focused on developing sparse filter designs.

Since the research being conducted will develop a production level product in small quantities, for extremely important and flexible tasks, Field Programmable Gate Arrays are a wise choice. The design that is created should also lend itself to re-programmability, since the limited important use of this design may be dynamic in the future. This work attempts use the strengths of the FPGA structure in this specific implementation.

In this research, the goal is to determine how sparseness in FIR filter design affects implementation and area usage in FPGAs while also investigating the potential for taking advantage of these implementation and area idiosyncrasies and leveraging them in the real world.

1.1 Sparse FIR Filters

What is sparseness and what does it mean to be sparse? Sparseness in this filter design means a filter with the majority of the coefficients being zero. In this particular case, greater than 94% of the filter coefficients are zero. Sparseness is important to this design for a few reasons. First, sparse filters constructed on an ASIC could take up less space when designed appropriately since zeros implemented in hardware equate to open circuits. When comparing two filters of similar order, with one being sparse and the other being non-sparse, the area used on a particular FPGA could be less when a sparse filter is implemented. Second, since there could be less area used in a particular FPGA, it can be reasonably said that if there was an ASIC/IC design of this filter, the layout of a sparse filter with very few non-zero coefficients would comprise a smaller area. Since zeros generally equate to a lack of computation and non-zero coefficients may at some point require computations, highly sparse filter designs could lead to smaller integrated circuit design with less computations. Third, multiplying in a FPGA or any design for that matter, is easier with a sparse filter. Since anything multiplied by zero is equal to zero, it takes care of itself naturally since the computation can be omitted. Since there are only a limited number of coefficients to multiply by, the number of steps involved for multiplying in a matrix form can be limited.

$$H(z) = \sum_{i=0}^N h_i z^{-i} \quad (1)$$

Many different DSP systems utilize FIR filters. Much of the time, the number of non-zero coefficients is directly proportional to the number of multiplications needed to realize the transfer function of the FIR filter [1]. Equation (1) shows the transfer function of a transposed FIR filter. The variable h represents the FIR filter coefficients where the number of coefficients is $N+1$. For every zero coefficient in the FIR filter, there is a multiplication by zero. When one is looking to save power, or computations, an FIR filter with more zeros and similar frequency

response characteristics to an FIR filter with less zeros would be warranted.

A sparse FIR filter is defined a filter in which there is a large number of zero coefficients. So, one advantage to using sparse FIR filters for filter design implementations, is reducing the number of multiplications. This can lower the power consumption of the filter. Another advantage is to minimize the area used with custom VLSI or Application Specific Integrated Circuit (ASIC) implementations, which leads to more efficient operation of higher order filters.

1.2 Sparse Filters in FPGAs

Many applications are well suited to using Field Programmable Gate Arrays (FPGAs) for algorithm implementation. FPGAs are chips that contain groups of re-programmable logic, memory, and processors. FPGAs are useful when a design requirement is limited in quantity since the overall costs allow them to be more economical to a project. Many FPGAs also allow designs to be reprogrammed on the chip which can be useful for circuit designs that may change over time. The interest for this research lies in how sparse FIR filters are implemented using FPGA hardware and how that specifically affects the area usage of the FPGA.

1.3 Xilinx Virtex-4 Design

The Virtex 4 FPGA by Xilinx is a programmable logic device that is a powerful alternative to ASICs. There are 3 models of the Virtex 4: LX, SX, and FX. The LX model is optimized for logic applications, the SX is optimized for DSP applications, and the FX is optimized for embedded platform applications. The SX version of the Virtex 4 FPGA was chosen in this research for its DSP capabilities. All Virtex 4s share similar architectural similarities, Digital Clock Manager Blocks, high speed Ram Blocks, IO blocks, ExtremDSP slices, and Configurable Logic Blocks [3]. The main architectural focus points in this design are the overall use of slices (part of the CLBs), and the DSP48 slices. The Virtex-4 family is chosen for this research since it is the selected target architecture for the delta sigma converter hardware design of interest to the Office of Naval Research and the Naval Research Laboratory.

Configurable Logic Blocks or CLBs are the primary resources for implementing logic in this FPGA chip. As shown in Figure 1.1, each CLB contains four slices broken into two pairs. The SLICEL slices are logic only slices, while the SLICEM slices have a common shift chain and Distributed RAM (DRAM). Both SLICEM and SLICEL columns have independent carry chains [2].

Figure 1.1 – CLB and Slice Layout [2]

The contents of each CLB are given in Table 1.1 below. It can be seen that each CLB contains four slices. Each of those four slices contains two LUTs, two D-FlipFlops, and two Mult_ANDs. SLICEM slices exclusively contain 64 bits of Distributed RAM and 64 bits of Shift Registers.

Table 1.1 – CLB contents

Slices	LUTs	Storage Elements	Mult_ANDs	Arithmetic & Carry Chains
4	8	8	8	2

The LUTs in each slice are used for function generation and each have 4 inputs. These

LUTs can implement any arbitrarily defined 4 input Boolean function. The output of these LUTs can be redirected out of the slice through most slice elements. Each slice contains multiplexers that can be leveraged to combine up to 8 function generators which can provide any function of five to eight inputs in a CLB. For large inputs or when wide logic is needed, these LUTs can be spanned across multiple CLBs [2].

The storage elements available in each slice can be configured as edge triggered D-Flip-Flops or level sensitive latches. The input to these elements can be driven by the output of a LUT, multiplexer, or by the slice inputs.

The SLICEM slices are unique in that they contain a shift chain and RAM. The shift chain allows delays to be added in or bypassing the use of FlipFlops without consuming extra elements. These shift chains can utilize the LUTs to create larger shifts for a delay of up to 64 clock cycles [2].

Each DSP48 tile on the Virtex-4 chip contains two DSP48 slices. The DSP48 slices are optimized for efficient DSP operations and are highly interconnected. DSP48s are able to be used with little supporting logic in the surrounding FPGA. The DSP48 slice is primarily an 18x18-bit two's complement multiplier with a 48-bit sign extended accumulator[4]. What makes this component versatile is its interconnections and pipelining of its inputs, intermediate products, and accumulator outputs, which can be programmed and routed or aggregated to or with other DSP48 slices [4]. These DSP48 slices are well-suited for use in FIR filter design.

1.4 ASICs vs. FPGAs

Although the target criteria for implementation in this research is the Virtex 4 FPGA, it should be noted that in many aspects, a custom ASIC implementation of the target filter would be superior. In area usage alone, an FPGA has been shown to use a mean of 40 times more area when implementing just logic [14]. When comparing logic, memory and DSP functions used together, an FPGA can use as little as 9.8 times to as much as 36 times the area compared with the same implementation on an ASIC. Kuon and Rose have also shown that the Dynamic Power consumption of an FPGA is from 5.2 to 12 times more than that of an ASIC, as well speed of an ASIC can be from 2.1 to 4.5 times as fast as an FPGA [14]. Though where performance, power

consumption, and area utilization favor ASICs, cost effectiveness, and turn around time favor FPGAs. The cost just for generating a mask for an ASIC design on the 90nm process can be over \$1.2 million, without even considering a the per device charge [15] where as a single FPGA can be as little as \$30 each. The turn around time for a design based on as ASIC may be as much as 2-5 months where as an FPGA may be 1-4 weeks once a customer's RTL (Register Transfer Level) is firm [15]. Since the design criteria for this research is for less than 100 of these filters, on an economical level, FPGAs are the solution. Power utilization and speed deferences when compared to an ASIC will have to be worked around.

CHAPTER 2

2. PREVIOUS RESEARCH

2.1 Delta-Sigma Converters

The driving force for conducting this research was to improve upon a communications system that contains a high performance delta-sigma converter. Unlike more traditional delta-sigma converters, this particular delta-sigma converter happens to be a very high order (upwards of 200) [5]. Conventional converters are usually clocked at the Nyquist rate [6] of the input signal, but delta sigma converters must be oversampled (ie. clocked at a much higher rate). The resulting signal from the output of a delta-sigma converter is usually a high speed, low resolution signal just like other oversampled analog-to-digital converters. In order to have a high resolution signal, the output of a delta-sigma converter must be put through a decimation filter. Since the delta-sigma converter oversampled the input signal, any noise produced from the delta-sigma converter is at much higher frequencies than the input signal. This high frequency converter noise can be filtered, which is the basis of noise shaping. The oversampling and noise filtering does come at a cost, there is noise introduced during quantization, which is the same bandwidth as the input signal.

Delta-sigma converters contain a delta-sigma modulator which converts the input signal to a high speed, low resolution output. Figure 2.1 shows a very simple first order delta-sigma modulator. The integrator plays the action role in the modulator, while the noise produced by the modulator is represented by $e(n)$. Since there is negative feedback of the output $y(n)$ back to the input, $x(n)$, the output signal is usually around the same value as the input signal.

Figure 2.1 – Simple First Order Delta-Sigma Modulator

Figure 2.2 – Simple First Order Digital Delta-Sigma Modulator

Figure 2.2 shows a basic first order digital delta-sigma modulator, the core of which is just a simple integrator constructed with an adder, register and feedback. The output of the integrator is then put through a comparator which quantizes it into a single bit. The single bit output of the comparator is then run through a Digital-to-Digital Converter (DDC) which converts the single bit signal back to a multi-bit signal. This multi-bit signal is then fed back and subtracted from the multi-bit input signal $x(n)$. From this structure in Figure 2.2 it is clear how this delta (subtraction) sigma (addition) modulator was named since the input signal is first subtracted from the output then added. Other types of modulators usually have the active filter in the feedback loop [5].

2.2 NRL Loop Filter Research

Previous research performed at the Naval Research Laboratory has focused on creating a specific high order Delta-Sigma loop filter [7, 8]. This filter has been designed to be an extremely sparse [1] form. The resulting sparse filter has order 198 and contains just 10 coefficients listed in table 2.1 below. This particular filter gives the frequency response shown in Figure 2.3. With only 10 coefficients, this sparse loop filter can reach an attenuation just over 80dB.

Table 2.1 – Previously designed sparse loop filter coefficients

[illegible]

Figure 2.3 – Frequency response of the NRL loop filter with 10 coefficients [9]

The bulk of the research in this paper is devoted to understanding how a sparse filter can be implemented in an FPGA, and how zero placement in a particular FIR filter impacts overall area usage of FPGAs.

CHAPTER 3

3. TOOLS USED

A variety of tools were used during the completion of this research. Some tools were “off the shelf,” while others were more custom and “home grown.” This section will define the tools and custom programs used for conducting the research and some sample input and output to help explain how they were used for this thesis.

3.1 Mathworks Matlab®

Mathworks Matlab® is a high-level language and interactive environment used in computations, algorithm development, data visualization, and data analysis. It contains a powerful toolboxes customized for signal and image processing, communications, and modeling and analysis [10]. The primary use of Matlab during this research was to create functions that would facilitate generation of the filter masks, to place zeros in appropriate locations in a set of FIR filter coefficients. This was accomplished by set of functions and scripts described below. The flow for this particular set of Matlab functions was to generate representative set coefficient values, run those values through a mask to distribute zeros appropriately, then create representative coefficient files at the desired sparseness values, which looks like this:

Function Value Generators → apply filter masks → generate coefficient files

3.1.1 Function Value Generator Functions: *fv_sin.m* and *fv_rand.m*

To enable control of zero placement, we automated the generation of filters with specifically placed zeros. Since the exact non-zero coefficient values do not directly impact area usage when multiplications are implemented using embedded multipliers or LUTs, we did not

design specific filters but instead generated representative coefficient values. We used two sets of coefficient values: one set is based on values generated using a sine wave and the other set was generated randomly.

Figure 3.1 shows the Matlab code that generates the sine based values used in creating the sine based coefficients. This code creates a function named `fv_sin`. In order to use this code, one would type `fv_sin(ord)` where `ord` is equal to the filter order.

```
function [fv_sin_out] = fv_sin(ord)
% fv_sin(ord) : gives function values of a sin based filter
%   with number of values dependent on the order given

bitwidth=16
for k=1:ord
    negmask(k) = (-1)^k;
    negmask;
end

fv_sin_out = negmask.*(0.5*(sin(-pi/2:((2*pi)/(ord-1)):((3*pi)/2)))
+0.51);

end
```

Figure 3.1 – fv_sin.m code

The inner workings of `fv_sin(ord)` are easy to understand. First, the bit width is set for the filter coefficients. Sixteen is chosen here, but can be changed easily by editing the function. Next, a mask named “negmask” in Figure 3.1, of size `ord`, is created that has alternating positive and negative ones. This negmask is created in the “for” loop. The reason for creating negmask with alternating negative numbers was to include both positive and negative coefficient values.

The sine function is sized so that all values generated will be greater than zero, yet less than one. The sin function is also shifted so that the greatest value for the coefficients are located towards the center of all the coefficients. Figure 3.2 depicts the sine function before it is multiplied by the negmask. Finally, the sin function gets multiplied by the negmask, which generates a set of coefficients displayed in Figure 3.3

Figure 3.2 – sin function used in fv_sin

Figure 3.3 – output of function `fv_sin(ord)`

The need for creating a random set of coefficients was realized when noticing that due to the periodic nature of the sin function, `fv_sin(ord)` generated coefficients where some of the absolute values were equal. Since we are attempting to determine how zeros in a FIR filter affects the area usage of an FPGA, we needed to create another set of coefficients that were diverse and may not have this same value equivalence as often. The value generator `fv_rand(ord)` is even easier to understand than its counterpart. Figure 3.4 shows the code used to define the `fv_rand` function.

```
function [fv_rand_out] = fv_rand(ord)
% fv_rand(ord) : gives function values of a random based filter
%   with number of values dependent on the order given

fv_rand_out = rand(1,ord).*sign(rand(1,ord) - 0.5)

end
```

Figure 3.4 – fv_rand.m code

Matlab® can create a random number from 0 to 1 by using the rand function. After a random number is generated, it is then multiplied by output of the sign function inside of which is another random number generated, that random number inside the sign function is then subtracted by 0.5, creating a negative number when the value of the random number is less than 0. The sign function is built into Matlab and returns a 1 if the element is greater than 0, returns a 0 if the element is 0, and returns a -1 if the element is less than 0. The final output of a particular run of fv_rand(ord) can be seen in Figure 3.5. The values of the random coefficients are random, with no weight going to any particular portion of the coefficients.

Figure 3.5 – output of function `fv_rand(ord)`

3.1.2 The Count Function

The function `Count(data,condition)` was created out of necessity to be able to count the number of times a particular element appears in a matrix. Figure 3.6 shows the code for `Count.m`.

```

function Result = Count(data,condition)
% COUNT (A,B)
% Counts the number of elements in A that match the criteria specified in
B.
% Example:
% Data = [1 2 3 4 3 2 7 6 9 1 1 2 5 9 9];
% Count(Data,'==9')
% ans = 3

nElements = length(data);
IndexIDs = 1:nElements;
Result = eval(['data' condition]);
Result = IndexIDs(Result);
Result = length(Result);

```

Figure 3.6 – Count.m code

The Count.m code is useful when attempting to count the number of zeros contained in a matrix. Count(data,condition) was written to be flexible and used for more than counting zeros in a matrix, but was just used for counting zeros in this research. First the length of the data is measured, then an index is created for each element in the data passed to the Count function. The condition is then run against all elements in the data passed to the Count function with the eval function, which executes a string. In the case of this research, the condition used was “==0” since we were interested in how many elements are equal to zero in a given set of values. Everywhere the condition is true, the index is saved in a result set. The result set is then counted with the length function, and a result meeting the condition is given.

3.1.3 Sparseness Checker Function

The sparseness checker is a function that we created in order to evaluate how sparse a particular matrix or set of filter coefficients is. It was created primarily as a way to verify that the filter masks that were generated were indeed creating the sparseness level intended. Figure 3.7 details the code used to create the sparseness checker.

```

function [sparseness_pct, min_sparseness] = sparseness_checker(A)
% sparseness_checker(A) returns a decimal percentage of how many
% zeros
% are in the given matrix (A),it also returns a minimum for the given
% filter
%
% sparseness_pct: from 0 to 1, 0=no zeros, 0.5=half zeros, 1=all zero

%find out the size of matrix A
[a b] = size(A);
%count the number of zeros in matrix A
num_zeros = Count(A, '==0');

sparseness_pct = num_zeros/b;
min_sparseness = (num_zeros - 1)/b;

```

Figure 3.7 – sparseness_checker.m code

The function `sparseness_checker(A)` takes in a row matrix `A` and outputs two values. The first value it outputs is the sparseness percentage given as a value from 0 to 1. The second value returned by the `sparseness_checker` function is the minimum sparseness as a value from 0 to 1. The minimum sparseness is a check value to make sure that a particular sparseness percentage can be attained. The first step in determining either of the two output values, is to determine the size of the matrix that is passed into it. The built-in Matlab® function `size(A)` outputs the number of rows and the number of columns into two variables `a` and `b`, respectively. The number of zeros in the row matrix is then calculated by using the `Count.m` function described earlier, which in this case, counts the number of zeros in a given matrix and stores them in the `num_zeros` variable. In order to calculate the sparseness percentage, the function divides the number of zeros by the size of the matrix, to get a value less than 1. This value is saved as the `sparseness_pct`. For ease of using this `sparseness_checker` function with other Matlab code written, the `min_sparseness` variable takes one zero off of the `sparseness_pct` calculation value, in order to give a range of sparseness values for other programs to compare with.

3.1.4 Filter Mask Generator Function

The filter mask generator function is a custom function that is written in Matlab. The main purpose of the filter mask generator is to generate a mask of 1's and 0's in the appropriate locations for the particular research being conducted. The filter mask generator function takes 3 variables as inputs, the filter type (ft), order (ord), and sparse percentage (sp), and then outputs a particular filter mask. Figure 3.8 details the usage for the filter mask generator function.

```
function [filt_mask] = filter_mask_gen(ft, ord, sp)
% filter_mask_gen(ft,ord,sp) Creates a mask filter from filter type, order, and
% sparse percentage (in decimal form)
% filter types:
%     1 -- Evenly spaced Zeros distributed from left to right
%     2 -- Linear Middle (increasing zeros as we get towards the center)
%     3 -- Linear Middle (increasing ones as we get towards the center)
% order: scalar positive integer (ie. 100, 230, 431 etc...)
% sparse percentage: from 0 to 1, 0=no zeros, 0.5=half zeros, 1=all zeros
```

Figure 3.8 – filter_mask_gen

There are three types of filter masks that are created for this research; evenly spaced zeros, linear middle with zeros, and linear middle with ones. Each of the filter masks distributes zeros in a different way. The last two filter masks are symmetric with respect to the center, whereas the first filter type is asymmetric. The detail of how each filter is created is detailed further in this chapter. It is important to note, that when using the `filter_mask_gen` function, that the order variable is a scalar positive integer, and the sparseness percentage is a positive decimal number between 0 and 1.

3.1.4a Filter Type 1 – Evenly Spaced Zeros

The first filter mask type, referred to as filter type 1 (ft1) is a set of evenly spaced zeros distributed from left to right. Figure 3.9 below displays the code snippet used to create the mask for filter type 1.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   if filter type is an evenly spaced filter from left to right
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if ft==1
    %% create matrix mask for coef distribution
    % find out how may zeroes due to sparsity and round
    num_zeroes=round(ord*sp);
    % find out how many non-zero coeffs are in filter and round it
    num_coeff=round(ord-num_zeroes);
    %% space the coeffs in the mask
    % if sparsity is less than .5
    if sp < .50
        %find the floor for the zero coef spacing
        zero_coeff_spacing=floor(ord/num_zeroes);
        %create a matrix of all 1s
        A=ones(1,ord);
        n=zero_coeff_spacing;
        %replace the appropriate 1s with 0s in the designated places
        for k=1:num_zeroes
            A(n)=0;
            n=n+zero_coeff_spacing;
        end
    A;
    end
    %%if sparsity is greater than .5
    if sp >= .50
        %find the floor of the nonzero coef spacing
        nonzero_coeff_spacing=floor(ord/num_coeff);
        %create a matrix of all zeros
        A=zeros(1,ord);
        n=nonzero_coeff_spacing;
        %replace the appropriate 0s with 1s in the designated places
        for k=1:num_coeff
            A(n)=1;
            n=n+nonzero_coeff_spacing;
        end
    A;
    end
    %ft_1 = fv_sin(ord);
    actual_sparseness = num_zeroes/ord
end

```

Figure 3.9 – Code snippet for filter mask type 1, evenly spaced zeros from left to right

The first step in creating ft1's mask is to determine how many zeros are needed to be distributed within the mask to achieve a desired sparseness percentage. This number is referred to as num_zeros in Figure 3.9 above and is calculated by multiplying the order by the sparseness of the mask percentage. From the num_zeros value it can then be determined the number of non zero components or ones that are located in the end result. We can obtain the number of ones by subtracting num_zeros from the order.

Once the number of zeros is determined, it can be decided to either place the 1's in the mask, or place the 0's in the mask depending on which ever quantity is less. A simple if statement takes care of this. If the sparseness percentage is less than 50 percent, then a matrix of all 1's is created and the zeros are placed appropriately. In order to determine the equal spacing for the zeros, the floor of the order divided by the number of zeros is used. A for loop is used to place the zeros in the appropriate places within the matrix of all ones. Figure 3.10 shows the output of filt_gen_mask where the filter type equals 1, order equals 20, and sparseness equals 45 percent.

```
>> filter_mask_gen( 1, 20, 0.45 )  
actual_sparseness =  
    0.4500  
  
filt_thesis =  
    1    0    1    0    1    0    1    0    1    0    1    0    1    0    1    0    1    0    1    1
```

Figure 3.10 – The output of Matlab function filter_mask_gen(1, 20, 0.45)

If the sparseness percentage is more than or equal to 50 percent, then a matrix of all 0's is created and the ones are placed appropriately. In order to determine the equal spacing for the ones, the floor of the order divided by the number of coefficients is used. A for loop is used to place the ones in the appropriate places within the matrix of all zeros. Figure 3.11 shows the output of filt_gen_mask where the filter type equals 1, order equals 20, and sparseness equals 75 percent.

```
>> filter_mask_gen( 1, 20, 0.75 )  
actual_sparseness =  
    0.7500  
  
filt_thesis =  
    0    0    0    1    0    0    0    1    0    0    0    1    0    0    0    1    0    0    0    1
```

Figure 3.11 – The output of Matlab function `filter_mask_gen( 1, 20, 0.75 )`

3.1.4b Filter Type 2 – Linear Increasing Zeros Middle

The second filter type that the filter mask generator function produces is a mask in which there are an increasing number of zeros towards the middle (or center) of the mask. This filter mask type is called linear middle increasing zeros, or filter mask type 2 (ft2). Figure 3.12 below shows a Matlab code snippet for the code used to generate this mask.

```

if ft==2
%start with the beginning of the right (a) and left (c) sequence
%sequence should look something like this:
% ...1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0...
a=[1 0 0];
c=[0 1 1];
%loop 3 -> ord number of times to create the pattern like above
for k=3:ord
    a=[a 1 zeros(1,k)];
    c=[c 0 ones(1,k)];
end
%flip the mostly 1 pattern
c=fliplr(c);
%combine the two matrices
M=[c a];
%find middle point in sequence
[y z] = size(M)
middle = z/2;
count=0;
offset = 0;
found = 0;
while found ~= 1
    %create 1/2 window for finding sparseness in sequence (will flip and combine for full
window)
    start_window = 1 + offset;
    end_window = start_window + (ord/2-1);
    half_window_size = end_window-start_window;
    %flip window around and add to it's original self, to create a new full window
    left_half_window = M(start_window:end_window);
    right_half_window = fliplr(left_half_window);
    new_full_window = [left_half_window right_half_window];
    size_new_full_window = size(new_full_window);
    %find where the window should go... start at the left and work right
    A = new_full_window;
    [window_sparse_actual window_sparse_min] = sparseness_checker(A);
    %if equal or within the range of sparseness.... we are done...
    if (sp == window_sparse_actual || sp < window_sparse_actual)
        window_sparse_min;
        actual_sparseness = window_sparse_actual
        found = 1;
    %if the window sparseness is less than required sparseness shift window to the right
    elseif window_sparse_actual < sp
        offset = offset + 1;
    %make sure we do not get stuck in a loop... error out.
    elseif offset == middle*2
        fprintf error_loop
        found = 1;
    end
end
end
end

```

Figure 3.12 – Code snippet for filter mask type 2, linear spaced zeros middle

The first step in creating ft2's mask is to build a sequence of 1's and 0's that fit into a pattern that serves as a basis. The repeating pattern that is generated can be seen in Figure 3.13 and contains a sequence of numbers approximately 2 times the order of the mask that needs to be generated. The reason for generating this pattern with increased order is make it easy to find a particular sparseness percentage window. The for loop at the beginning of the code snippet in Figure 3.12 details the beginning of creating the sequence. It can be seen that as the sequence moves left, the frequency of 1's increases linearly. Likewise, as the sequence moves right, the frequency of 0's increases linearly. This sequence is the basis for both filter mask types 2 and 3.

...1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0...

Figure 3.13 – Sequence used for creating ft2 and ft3 masks

Once the sequence is realized, a window is created starting from the left of the sequence. This window is created for ft2 by first creating two smaller windows of half the filter order called the “right half window” and the “left half window”. For ft2, a copy of the “left half window” is flipped from left to right and called the “right half window”. These two windows are “glued” together to create a full window.

After the full window is created, the sparseness of that window is checked using the sparseness checker function mentioned in Section 3.1.3, which verifies if the particular window falls within a range of sparseness values returned by the checker. If the sparseness criteria is not met, new left and right half windows are created from a shifted right portion of the sequence, and then retested for sparseness compliance. When the sparseness criteria is finally met, the proper filter mask 2 is returned. Figure 3.14 shows the output of filter_gen_mask where the filter type equals 2, order equals 20, and the sparseness percentage equals 75 percent.

```
>> filter_mask_gen( 2, 20, 0.75 )
actual_sparseness =
    0.8000

filt_thesis =

    0    0    1    0    0    0    1    0    0    0    0    0    0    1    0    0    0    1    0    0
```

Figure 3.14 – The output of Matlab function `filter_mask_gen( 2, 20, 0.75 )`

From Figure 3.14, it can be seen that it was requested that the mask be 75 percent sparse, but the actual sparseness percentage was 80. For this mask type, with only 20 coefficients, a 75 percent sparse mask cannot be created. In this case, the next closest sparse filter is created. Luckily, the higher order the mask filter is, the less likely a sparse mask cannot be created for a given sparse percentage or the closer the sparseness percentage will be to what was requested. For example, if the order for Figure 3.14 is changed to 50, the actual sparseness percentage is equal to 76 percent, much closer to the requested 75 percent. When the order is changed to 80, the actual sparseness is equal to 75 percent.

3.1.4c Filter Type 3 – Linear Increasing Ones Middle

The third filter type that the filter mask generator function produces is a mask in which there are an increasing number of ones the closer towards the middle (or center) of the mask one gets. This filter mask type is called linear middle increasing increasing, or filter mask type 3 (ft3) for short. Figure 3.15 below shows a Matlab code snippet for the code used to generate this mask.

```

if ft==3
%start with the beginning of the right (a) and left (c) sequence
%sequence should look something like this:
% ...1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0...
a=[1 0 0];
c=[0 1 1];
%loop 3 -> ord number of times to create the pattern like above
for k=3:ord
    a=[a 1 zeros(1,k)];
    c=[c 0 ones(1,k)];
end
%flip the mostly 1 pattern
c=fliplr(c);
%combine the two matrices
M=[c a];
%find middle point in sequence
[y z] = size(M)
middle = z/2;
count=0;
offset = 0;
found = 0;
while found ~= 1
    %create 1/2 window for finding sparseness in sequence (will flip and combine for full
window)
    start_window = 1 + offset;
    end_window = start_window + (ord/2-1);
    half_window_size = end_window-start_window;
    %flip window around and add to it's original self, to create a new full window
    right_half_window = M(start_window:end_window);
    left_half_window = fliplr(right_half_window);
    new_full_window = [left_half_window right_half_window];
    size_new_full_window = size(new_full_window);
    %find where the window should go... start at the left and work right
    A = new_full_window;
    [window_sparse_actual window_sparse_min] = sparseness_checker(A);
    %if equal or within the range of sparseness.... we are done...
    if (sp == window_sparse_actual || sp < window_sparse_actual)
        window_sparse_min;
        actual_sparseness = window_sparse_actual
        found = 1;
    %if the window sparseness is less than required sparseness shift window to the right
    elseif window_sparse_actual < sp
        offset = offset + 1;
    %make sure we do not get stuck in a loop... error out.
    elseif offset == middle*2
        fprintf error_loop
        found = 1;
    end
end
end
end

```

Figure 3.15 – Code snippet for filter mask type 2, linear spaced ones middle

The first step in creating ft3's mask is the same as ft2's mask, that is, build a sequence of 1's and 0's that fit into a pattern that can be worked with. The repeating pattern that is generated can be seen in Figure 3.13 and contains a sequence of numbers on the approximately 2 times larger than the order of the mask that needs to be generated. The reason for generating this pattern in such a large order is to make it easy to find a particular sparseness percentage window. The for loop at the beginning of the code snippet in Figure 3.12 details the beginning of creating the sequence. It can be seen that as the sequence moves left, the frequency of 1's increases linearly. Likewise, as the sequence moves right, the frequency of 0's increases linearly. This sequence is the basis for both filter mask types 2 and 3.

Once the sequence is realized, a window is created starting from the left of the sequence. This window is created for ft3 by first creating two smaller windows similar to that of ft2 called the “right half window” and the “left half window”. Though, for ft3 The right half window is created first from the beginning of the sequence on the left, then is flipped from left to right and called the “the left half window”. These two windows are “glued” together to create a full window with the correct dispersion of 1's and 0's.

After the full window is created, the sparseness of that window is checked using the sparseness checker function mentioned in Section 3.1.3, which verifies if the particular window falls within a range of sparseness values returned by the checker. If the sparseness criteria is not met, new left and right half windows are created from a shifted right portion of the sequence, and then retested for sparseness compliance. When the sparseness criteria is finally met, the proper filter mask 3 is returned. Figure 3.16 shows the output of filter_gen_mask where the filter type equals 3, order equals 20, and the sparseness percentage equals 75 percent.

```
>> filter_mask_gen( 3, 20, 0.75 )
actual_sparseness =
    0.8000

filt_thesis =
    0    0    0    1    0    0    0    1    0    0    0    0    1    0    0    0    1    0    0    0
```

Figure 3.16 – The output of Matlab function filter_mask_gen(3, 20, 0.75)

Just like for filter type 2, from Figure 3.16, it can be seen that it was requested that the mask be 75 percent sparse, but the actual sparseness percentage was 80. For this mask type, with only 20 coefficients, a 75 percent sparse mask cannot be created. In this case, the next closest sparse filter is created. Luckily, the higher order the mask filter is, the less likely a sparse mask cannot be created for a given sparse percentage or, the closer the sparseness percentage will be to what was requested. For example, if the order for Figure 3.16 is changed to 50, the actual sparseness percentage is equal to 76 percent, much closer to the requested 75 percent. When the order is changed to 80, the actual sparseness is equal to 75 percent.

3.1.5 Coefficient File Generator Functions

The final custom Matlab functions are two functions that generate the final coefficient files that will be used for the rest of the research. The two functions used are named `generate_coeffs()` and `generate_rand_coeffs()`. These functions take two variables as input: filter type (a number 1 - 3) and order (a positive scalar integer). The output is a file with a list of numbers (coefficients) separated by carriage returns. These two functions utilize all the previously mentioned Matlab code mentioned in this research. They are the only functions that need be run in order to have an output of coefficient files for varying sparsities.

3.1.5a Generate Sine Based Coefficients

The root of `generate_coeffs(ft,ord)` is a for loop that names and creates a filter coefficient file for sparseness levels defined in the loop itself. Figure 3.17 shows the details of the code used for generating the sine based coefficient files.

```

function [generate_coeffs_out] = generate_coeffs(ft,ord)
%% generate_coeffs(ft,ord) generate coefficients for thesis based off the filter type and
order.

for sp=0:0.05:1
    b = filter_mask_gen(ft,ord,sp).*(fv_sin(ord))
    %create name for file
    var1 = 'coeff_';
    var2 = 'filt_';
    undsc = '_';
    var3 = 'order_';
    var4 = '_sparse_';
    var5 = '.coeff';
    name = [var1 var2 num2str(ft) undsc var3 num2str(ord) var4 num2str(sp) var5]
    fid = fopen( name , 'w');
        fprintf(fid, '%-2.20f\n' , b);
    fclose(fid);
end

```

Figure 3.17 – Code for Generating Sin Based Coefficient files

The function `generate_coeffs()` calls two other custom Matlab® functions, `filter_mask_gen()` and `fv_sin()` and multiplies them together element by element. This function essentially applies the mask to the function value generator. Once this mask is applied, the coefficient file is named and saved outside of Matlab. First, a name variable is created based upon the details of the coefficient file to be generated. Details such as the order, sparseness percentage and filter type are saved in the name variable. Then the Matlab function `fopen` is used to open a file named for the details in which it was created for writing onto the file system. The Matlab function `fprintf` is then used to write the coefficients to the named file, and the Matlab function `fclose` saves the file onto the file system.

3.1.4b Generate Random Coefficients

The root of `generate_rand_coeffs(ft,ord)` is much the same of its counter part, a for loop that names and creates a filter coefficient file for sparseness levels defined in the loop itself. Figure 3.18 shows the details of the code used for generating the random based coefficient files.

```
function [generate_rand_coeffs_out] = generate_rand_coeffs(ft,ord)
%% generate_rand_coeffs(ft,ord) generate coefficients for based off the filter type and
order.

for sp=0:0.05:1
    b = filter_mask_gen(ft,ord,sp).*(fv_rand(ord))
    %create name for file
    var1 = 'coeff_';
    var2 = 'filt_';
    undsc = '_';
    var3 = 'order_';
    var4 = '_sparse_';
    var5 = '.coeff';
    var6 = '_rand_';
    name = [var1 var2 num2str(ft) undsc var3 num2str(ord) var4 num2str(sp) var6 var5]
    fid = fopen( name , 'w');
        fprintf(fid, '%-2.20f\n' , b);
    fclose(fid);
end
```

Figure 3.18 – Code For Generating Random Based Coefficient Files

The function `generate_rand_coeffs()` calls two other custom Matlab® functions, `filter_mask_gen()` and `fv_rand()` and multiplies them together element by element. This function essentially applies the mask to the function value generator. Once this mask is applied, the coefficient file is named and saved outside of Matlab. First, a name variable is created based upon the details of the coefficient file to be generated. Details such as the order, sparseness percentage and filter type are saved in the name variable. Then the Matlab function `fopen` is used to open a file named for the details in which it was created for writing onto the file system. The Matlab function `fprintf` is then used to write the coefficients to the named file, and the Matlab function `fclose` saves the file in the file system.

3.2 PHP Hypertext Preprocessor

PHP Hypertext Preprocessor or PHP for short, is a widely used Open Source general purpose scripting language that is primarily used in web based programming since it is easily embedded into HTML [11]. The use of PHP in this research was primarily as a scripting language which is similar to that of Perl and Visual Basic Script. One advantage to using PHP is that it is Open Source and readily available free of charge. PHP is mostly platform independent and can be installed on most operating systems such as Linux, Solaris, Mac, BSD, and Windows. This is especially handy when research is conducted cross platforms such as this research was. PHP was essential to the process of generating VHDL code in this research.

3.2.1 VHDL Generator

For this research, a custom PHP based Application Programming Interface called VHDLGen was utilized [9]. The VHDLGen API was written by Charles D. Howard for research in our group in order to create VHDL circuit files based upon a given coefficient file or files. The reasoning for creating this code was to help the group understand how to implement FIR filters in varying formats utilizing various features of FPGAs. This custom PHP code was essential in creating the VHDL code used for this research. Without it, creating the VHDL code by hand would have been a tedious, time consuming and error prone exercise.

To start using the VHDLGen API a run file must be created that details how VHDLGen is utilized. Basic understanding of PHP code is somewhat of a prerequisite in order to create a run file for VHDLGen. Figure 3.19 below shows an example run file creating two VHDL circuit files, one with order 200, filter type 1, and 15 percent sparse, the other with order 200, filter type 2, and 25 percent sparse.


```

<?php
function __autoload($class_name) {
    require_once "lib/$class_name.class.php";
}
$filters = array(
    array(
        "bitwidth" => 18,
        "decimalplaces" => 16,
        "coefffile" => $ouputDir.'coeff_filt_1_order_200_sparse_0.15.coeff',
        "entityname" => "coeff_filt_1_order_200_sparse_015_coeff",
    ),
    array(
        "bitwidth" => 18,
        "decimalplaces" => 16,
        "coefffile" => $ouputDir.'coeff_filt_2_order_200_sparse_0.25.coeff',
        "entityname" => "coeff_filt_2_order_200_sparse_025_coeff",
    ),
);
foreach($filters as $filter){
    echo ($filter['bitwidth'] . "\n");
    echo ($filter['decimalplaces'] . "\n");
    echo ($filter['entityname'] . "\n");
    echo ($filter['coefffile'] . "\n");
    $fir = new FIRFilterData();
    $fir->setBitWidth($filter['bitwidth']);
    $fir->setDecimalPlaces($filter['decimalplaces']);
    $fir->loadCoeffsFromFile($filter['coefffile']);
    $VHDLGenerator = new VHDLcircuitFIR($fir);
    $VHDLGenerator->entityName = $filter['entityname'].'_b'.$filter['bitwidth'].'_fb'.
$filter['decimalplaces'];
    $VHDLGenerator->libraries = array('ieee');
    $VHDLGenerator->libUse[] = 'ieee.std_logic_1164.ALL';
    $VHDLGenerator->libUse[] = 'ieee.std_logic_signed.ALL';
    $VHDLGenerator->libUse[] = 'ieee.numeric_std.ALL';
    $VHDLGenerator->libUse[] = 'ieee.numeric_bit.ALL';
    $VHDLGenerator->ports['X'] = array('direction' => 'IN','type' =>
'STD_LOGIC_VECTOR(W-1 DOWNT0 0)');
    $VHDLGenerator->ports['CLK'] = array('direction' => 'IN','type' => 'STD_LOGIC');
    $VHDLGenerator->ports['RST'] = array('direction' => 'IN','type' => 'STD_LOGIC');
    $VHDLGenerator->ports['Y'] = array('direction' => 'OUT','type' =>
'STD_LOGIC_VECTOR(W-1 DOWNT0 0)');
    $VHDLGenerator->outputDirectory = '/home/output';
    $VHDLGenerator->save();
}

```

Figure 3.19 – VHDLGen Run File Example

At the beginning of the run file, the “__autoload” function is a feature in PHP 5 that allows VHDLGen API classes to be loaded as needed without the need to define them all in a

particular PHP file. As the VHDLGen API is expanded, keeping track of the required classes needed to correctly instantiate VHDLGen is taken care of by the “__autoload” function. Since VHDLGen has many class files, this function definition limits the amount of code that needs to be written at the top of each run file in order for the run file to function properly [9].

After the classes for the VHDLGen API are loaded, the FIRFilterData object is defined for each VHDL circuit file. The FIRFilterData object implements a transposed FIR filter in VHDL using the standard VHDL multiplication operator “*” [9]. The FIRFilterData object requires certain variables to be defined: the bit width, the number of decimal places to use, and coefficient variables. These three variables are used to correctly quantize the coefficients before creating the VHDL circuit file. The recommended way of passing coefficients into the VHDLGen run file is to call them from a coefficient file on the local file system. VHDLGen will parse through the file looking for any string that can be formatted as a number while ignoring all other values, and load those numbers for processing.

Once the FIRFilterData object is loaded with the necessary values, the VHDLcircuitFIR object can be used to build the VHDL file. The VHDLcircuitFIR object is responsible for formatting the FIR filter into a form that a VHDL interpreter can understand. It expects that there has already been some setup for the VHDL circuit port names and libraries with the use of the “\$VHDLGenerator->” statements listed in Figure 3.19.

The last line of Figure 3.19 which calls the “save()” function, creates VHDL circuit file based on all the data contained in the run file. The VHDL circuit file is saved on the local file system, the case of Figure 3.19, that file system location is “/home/output.” A summarized example of the VHDL circuit file created by VHDLGen can be seen in Appendix A.

3.3 Xilinx ISE

Xilinx ISE WebPACK Software is a fully featured front-to-back FPGA design solution for Linux and Windows operating systems. Xilinx ISE WebPACK offers HDL synthesis and simulation, device fitting, and JTAG programming [12]. The software supports most of the device families manufactured by Xilinx. The reason for using this software for research purposes was due to its price, availability, and feature set.

The main feature used in the Xilinx ISE WebPACK software was its ability to synthesize a VHDL circuit file and place it on a selected chip. Each of the VHDL circuit files like the one listed in Appendix A, were synthesized and the summary statistics were collected after each synthesis. An example synthesis report can be found in appendix B.

CHAPTER 4

4. APPROACH

The primary approach for analysis was to create a simple structure on the FPGA in which to implement sparse and non sparse FIR filters. The structure we chose was a simple Transposed form FIR filter in which the primary structure was the FIR filter itself. We chose a large 200 order filter in which to implement our design in order to have very large, very sparse filters to compare with one another. Figure 4.1 below shows the work flow used in the approach to this research.

Figure 4.1 – Approach Work Flow

Before implementing the FIR filter, we created the filter coefficient files. The first set of FIR filter coefficients were created using a modified sine function in which no coefficients were greater than 1 or less than -1. The sine function was weighted towards the center where the magnitude of the filter coefficients were larger towards the middle of the filter. Every other filter coefficient was inverted so we would have both positive and negative coefficient values. The second set of FIR filter coefficients created used random numbers between -1 and 1 with no particular weighting or distribution of the coefficients.

$$h_1 0 h_3 0 h_5 0 h_7 0 h_9 0 h_{11} 0 h_{13} 0 h_{15} 0 h_{17} 0 h_{19} 0$$

a) 50% sparse, evenly spaced zeros, 20 order

$$h_1 0 0 h_4 0 0 h_7 0 0 h_{10} 0 0 h_{13} 0 0 h_{16} 0 0 h_{19} 0$$

b) 65% sparse, evenly spaced zeros, 20 order

$$0 h_2 h_3 0 h_5 0 0 h_8 0 0 0 h_{12} 0 0 h_{15} 0 h_{18} h_{19} 0$$

c) 55% sparse, linear spaced zeros center, 20 order

$$h_1 0 0 h_4 0 0 0 h_8 0 0 0 0 h_{13} 0 0 0 h_{17} 0 0 h_{20}$$

d) 75% sparse, linear spaced zeros center, 20 order

$$h_1 0 0 0 h_5 h_6 0 0 h_9 h_{10} h_{11} h_{12} 0 0 h_{15} h_{16} 0 0 0 h_{20}$$

e) 50% sparse, linear spaced zeros edges, 20 order

$$h_1 0 0 0 0 h_6 0 0 0 h_{10} h_{11} 0 0 0 h_{14} 0 0 0 0 h_{20}$$

f) 70% sparse, linear spaced zeros edges, 20 order

Figure 4.2 – Sparse Filter Design Filter Type Examples

After the first set of FIR filter coefficients was created, we created a set of filter masks that places zeros throughout the filter depending on how sparse the desired FIR filter should be. The Figure 4.2 (a - f) depict an FIR filter of order twenty with different sparseness masks.

The first filter mask we created was an evenly (as possible) spaced zeros mask. This mask would space zeros evenly throughout the 200 order filter depending on sparseness required from left to right as figures 4.2 a) and 4.2 b) show. Some sparseness percentages do not lend themselves well to evenly spacing when it comes to the beginning or end of a filter, but zero or coefficient placement was consistent throughout all iterations of all the generated coefficient files.

The second filter mask we created was linear spaced zeros mask in which there were more zeros towards the center of the FIR filter with all coefficients or zeros symmetric around the center. Figures 4.2 c) and 4.2 d) show two different 20 order filters with this pattern.

The third filter mask that we created was another linear spaced mask in which there were more zeros towards the outside edges of the filter with all coefficients or zeros symmetric around the center. Figures 4.2 e) and 4.2 f) illustrate this zero spacing pattern.

The idea behind creating these masks was to investigate if the placement or groupings of zeros has an impact on area usage on the FPGA. The masks and the coefficients were then merged together for the appropriate sparseness levels using custom Matlab functions described earlier in this research. The output of this process produced sparse and non sparse FIR filter coefficient files.

Once the sparse and non sparse FIR filter coefficient files were created, we generated VHDL code to define the simple FIR filter using a custom PHP API named VHDLGen [9]. VHDLGen requires that a run file be created with FIR filter details defined. The output of the VHDLGen run files is a set of VHDL circuit files defining a simple FIR filter structure in transposed form.

After the VHDL circuit files are generated from the VHDLGen run files, a separate VHDL file is generated for each of the different filter masks for each of the sparse and non sparse filters. Each of these filters are then synthesized in Xilinx ISE 10.1 with a balanced design goal. After synthesis, the summary statistics are collected for analysis of area usage. These results are discussed in the results chapter.

CHAPTER 5

5. RESULTS

In each of the graphs below we investigate how the placement of the zeros in the FIR filter impacts the utilization of the FPGA. As we can see from our first run with the sine based FIR filters in Figures 5.1, 5.2, and 5.3, there is a steep drop in Slice and LUT usage from 0% sparse through 35% sparse. But between 35% sparse through 95% sparse, there is not as dramatic of a decline in LUTs or Slice usage. Throughout all the first run figures below (5.1-5.3) we have very consistent Slice flip-flop usage, just slightly declining as the FIR filter sparseness increases.

Figure 5.1 – Filter Type 1 (evenly spaced zeros) Element Usage

There seems to be some oddity in the evenly spaced zeros (ft1) in Figure 4.1 when the sparseness gets between 70% and 80%. Instead of the continual decrease like as in the

utilization of slice flip-flops, we see an increase. At these points in the linear spaced zeros FIR filter, we see some exaggerated asymmetry in the coefficients where there are a grouping of zeros at the end of the FIR filter itself. This was true regardless of what the coefficients were, which was discovered through random coefficients, and depended solely on the placement and grouping of zeros.

Figure 5.2 – Filter Type 2 (Increasing zeros center) Element Usage

Figure 5.3 – Filter Type 3 (Increasing non-zeros center) Element Usage

The three filter types (ft1 ft2 ft3) listed in Figures 5.4, 5.5, and 5.6 correspond directly to the filter types (evenly spaced zeros, linear increasing zeros (more zeros towards center), linear increasing zeros(more zeros towards edges)) listed the Figures 5.1-5.3. As shown in Figure 5.4, the LUTs used were fairly consistent for filter types 2 and 3 with little deviation in the quantity used. Filter type 1 seems to have a little more deviation from filter types 2 and 3, and there exists some points of interest at 30% sparseness, where filter type 1 uses 12% more LUTs than filter type2, and at 75% sparseness where filter type 1 uses almost 44% less LUTs than filter type 2.

Figure 5.4 – LUT Usage For All Three Filter Types

Figure 5.5 – Slice Flip-flop Usage For All Three Filter Types

When comparing the Slice flip-flops as seen in Figure 5.5, filter type 1 seems to consistently use more flip-flops through all sparseness percentages. At 75% sparseness, filter type 1 uses 40% more flip-flops than filter type 2. Filter type 1 uses 12% less slice flip-flops at the 90% range than filter type 2.

Figure 5.6 – Total Slices Used For All Three Filter Types

Slices are the elements on the Xilinx FPGAs that encompass Look Up Tables, flip-flops, and other logic cells that interconnect throughout the FPGA itself. In Figure 5.6 we can see that filter type 1 almost always uses more Slices when compared to the other filter types. This is especially evident at the 75% sparseness data point where filter type 1 uses around 40% more slices than filter type 2. When the sparsity of the filter reaches 90%, it seems that filter type 1 uses around 12% less slices than filter type 3.

As mentioned at the beginning of this research, DSP48 blocks are a great resource for DSP implementations. The DSP48 slices are optimized for efficient DSP operations, and are highly interconnected. They are able to be used with little supporting logic in the surrounding FPGA. Figure 5.7 below details how the DSP48 slices were used for each of the three filter types.

Figure 5.7 – DSP48 Slices Used For All Filter Types

There is a fairly straightforward pattern that can be seen in the DSP48 slice usage. All three filter types seem to use approximately the same number of DSP48 slices which correspond directly with the number of nonzero coefficients. The maximum number of DSP48 slices on the chip used in this research is 128. All 128 DSP48 slices were used up until the sparseness level for this 200 order filter reached somewhere between 35 and 40 percent (130 and 120 nonzero coefficients respectively). The slight discrepancies between the DSP48 slice usage between filter type 1 when compared with filter types 2 and 3 can be explained by the filter mask generator. For the values that are not equal, the actual sparseness values were as much as 2% off at the maximum. The first time this difference is noticed in Figure 5.7 is at 60% where filter

type 1 uses 80 DSP48 slices, where as filter types 2 and 3 use 78 DSP48 slices. After the filter masks were applied, filter types 2 and 3 had 2 less nonzero coefficients than filter type1.

Figure 5.8 – Slice vs DSP48 Utilization

Figure 5.8 compares the utilization percentage of slices and DSP48s on the Xilinx Virtex4-sx25 used in this research. We have chosen to use the results from ft2 to display since all filter types regardless of which coefficient sets used, show approximately the same results. There is a transition point around 35% sparseness that impacts the total utilization of the FPGA. As the the FIR filter design increases in sparseness after 35% there is a steeper decline in the number of DSP48s used than in the slices utilized. Overall it seems that the number of DSP48 slices used and the number of nonzero coefficients are identical when there are less than 128 nonzero coefficients. The decline in total slices used seems to only decrease slightly after the 35 percent sparseness level.

To be complete in our analysis, we had to generate some random coefficients that would hopefully demonstrate that the value of the particular element did not have a tremendous impact on the area utilization of an FPGA. The following figures detail the results from synthesizing the generated VHDL files with random coefficients. There are not quite as many data points collected, but the overall trends can be noticed.

Figure 5.9 – Filter Type 1 Random Coefficients

Figure 5.10 – Filter Type 2 Random Coefficients

Figure 5.11 – Filter Type 3 Random Coefficients

As can be seen from Figures 5.9 – 5.11, the general trend in element usage from the random coefficients makes a steep decline until about the 40% mark in sparseness. Slice flip flop usage stays in small general downward decline the more sparse a particular implementation gets. Just as with the sine based coefficients, in filter type 1 we see a small increase in elemental usage in the higher sparseness synthesis runs.

Figure 5.12– Slices Used For All Three Filter Types Random Coeffs

Figure 5.13– Slice Flip Flops Used For All Three Filter Types Random Coeffs

Figure 5.14– LUTs Used For All Three Filter Types Random Coeffs

Figures 5.12 – 5.14 compare the elemental usage side by side for each of the three filter types while using randomly generated filter coefficients. The general trends are similar to those in Figures 5.4 – 5.6 when using the sine based coefficients. As with the sine based filter type 1, the random coefficient based filter type 1 seems to use more FPGA elements than filter types 2

and 3. Overall it seems that filter type 3 uses the least number of elements on average when compared to the other two filter types while using random coefficients. With that being said, the differences in elemental usage when comparing filter types 1 and 2 are minimal.

Figure 5.15 – Virtex 4 LX Element Usage

We attempted to use a different FPGA chip from the Virtex 4 family for analysis and completed one initial run with the Virtex 4 LX25 using filter type 1. This particular chip does not have a high enough capacity to handle a higher order filter that was non sparse. As you can see from Figure 5.15, the LUT and slices utilization do not show until the filter is at least 30% sparse. The Virtex 4 LX FPGA only contains 48 DSP slices where as the SX chip contains 128 DSP48 slices. When LUTs, slices, and slice flip-flops are compared between the two chip types, the LX chip actually has a little more of these elements than the SX chip. It is convenient to note that when this LX chip implements our filter at 80% sparse, it uses exactly the same number of elements as the same filter type 1 on the SX chip.

CHAPTER 6

6. CONCLUSION AND FUTURE WORK

This work investigated how sparseness and zero placement impacts area utilization of a Virtex-4 FPGA. Overall it has been shown that regardless of distribution of zeros throughout the FIR filter, there is a steep drop in FPGA area utilization from 0% through 35% sparsity with our 200 order filter. With the Virtex-4 sx25 chip, the actual point at which all the DSP48 slices are fully utilized is at 36% sparsity. The difference in area utilization from a 35% sparse FIR filter to a 90% sparse FIR filter when compared to a FIR filter with no zeros is less than 10%. It would seem that there can be great area savings by making a high order, such as our 200 order example FIR filter, at least 35% sparse, but attempting to take an already sparse filter that may be 35% sparse or more and attempting to make it more sparse in order to save area, has diminishing returns. Instead, reducing the order is likely the best way to reduce area requirements. When order reduction is not applicable, it would then seem that the next best approach to limiting area utilization with FIR filters on FPGAs would be to make sure that all DSP48 slices are utilized. Utilizing DSP48 slices is automatic in Xilinx ISE when synthesized on Virtex FPGAs, and generally one DSP48 slice is used per non-zero filter coefficient. Area utilization seems to have diminishing returns after the point at which less than 100% of the DSP utilization takes place. It would also seem that the symmetry of the zeros in the FIR filter has an impact on the area utilization as evident in our linear spaced filter type 1.

In general, newer FPGAs still utilize more area, power, and are slower than ASICs. When DSP slices are involved, the utilization, power, and area gap is lessened. There seems to be a trade off in filter design that may become apparent to maximize the utilization of DSP slices in FPGAs. On one hand, designers may already have a sparse FIR filter with few non-zero coefficients that may benefit from redesign into a less sparse filter on a particular FPGA chip in order to maximize DSP slice utilization. On the other hand, a designer may have a large order filter that is currently using all of the DSP slices and may benefit from adding some sparseness to their filter designs. All these design criteria rely on the particular FPGA chip used and the

number of coefficients in their FIR filter design. This seems to be an optimization problem that can best be addressed by investigating all lines of FPGA technologies and partial redesigns of certain FIR filters.

As seen with filter type 1, zero groupings can have an impact on FPGA area utilization at different sparsity levels. When FIR filters are not symmetric with respect to the center and have large groupings of non-zero coefficients, they seem to use a little more area than when compared to a FIR filter that is symmetric with respect to the center and have non-zero coefficients spaced out appropriately. More investigation needs to be conducted with other zero groupings to see if there is a greater impact of designing FIR filters that have other arrangements of zeros that may or may not be symmetric.

APPENDIX A

1. EXAMPLE VHDL FILE CREATED BY VHDLGEN

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.numeric_std.ALL;
USE ieee.numeric_bit.ALL;

ENTITY equiripple_200 IS
    GENERIC(
        W : integer := 16;
        F_BITS : integer := 14;
        H_COUNT : integer := 16;
        ORDER : integer := 20
    );
    PORT(
        X : IN STD_LOGIC_VECTOR(W-1 DOWNTO 0);
        CLK : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        Y : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0)
    );
END equiripple_200;

ARCHITECTURE behavioral OF equiripple_200 is
    type reg_array is array (0 to ORDER-1) of signed(W-1 downto 0);

    signal REG : reg_array;

    function to_stdlogicvector(b : signed) return std_logic_vector is
        variable result : std_logic_vector( b'LENGTH-1 DOWNTO 0 );
    begin
        FOR i IN result'RANGE LOOP
            CASE b(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
                WHEN others => result(i) := '0';
            END CASE;
        END LOOP;
        RETURN result;
    end function;

    function mult_and_quant(X, h : signed) return signed is
        variable result : signed(2*W-1 downto 0);
    begin
        result := X * h;
        return result(W+F_BITS-1 downto F_BITS);
    end function;

BEGIN
    process(CLK,RST)
        type coeffs_array is array (0 to H_COUNT) of signed(W-1 downto 0);
        constant h : coeffs_array := (
```

```

"00000000101001100",
"0000000010011100",
"00000000110001000",
"00000000000010101",
"1111101111010101",
"111110100101011",
"0001001001011011",
"0001001101001100",
"0010001101111011",
"0001001101001100",
"00000001101100010",
"111110100101011",
"1111101111010101",
"0000100000011111",
"00000000110001000",
"0000000010011100",
"00000000101001100");
begin
  if (RST = '1') then
    for i in 0 to ORDER-1 loop
      REG(i) <= (others => '0');
    end LOOP;
    Y <= (others => '0');
  elsif (rising_edge(CLK)) then
    -- FIR REGISTER CHAIN
    Y <= to_stdlogicvector(mult_and_quant(signed(X), h(16)) +
REG(19));
    REG(19) <= mult_and_quant(signed(X), h(15)) + REG(18);
    REG(18) <= mult_and_quant(signed(X), h(14)) + REG(17);
    REG(17) <= mult_and_quant(signed(X), h(13)) + REG(16);
    REG(16) <= mult_and_quant(signed(X), h(12)) + REG(15);
    REG(15) <= REG(14); --Zero Coeff
    REG(14) <= mult_and_quant(signed(X), h(11)) + REG(13);
    REG(13) <= mult_and_quant(signed(X), h(10)) + REG(12);
    REG(12) <= mult_and_quant(signed(X), h(9)) + REG(11);
    REG(11) <= REG(10); --Zero Coeff
    REG(10) <= mult_and_quant(signed(X), h(8)) + REG(9);
    REG(9) <= REG(8); --Zero Coeff
    REG(8) <= mult_and_quant(signed(X), h(7)) + REG(7);
    REG(7) <= mult_and_quant(signed(X), h(6)) + REG(6);
    REG(6) <= mult_and_quant(signed(X), h(5)) + REG(5);
    REG(5) <= REG(4); --Zero Coeff
    REG(4) <= mult_and_quant(signed(X), h(4)) + REG(3);
    REG(3) <= mult_and_quant(signed(X), h(3)) + REG(2);
    REG(2) <= mult_and_quant(signed(X), h(2)) + REG(1);
    REG(1) <= mult_and_quant(signed(X), h(1)) + REG(0);
    REG(0) <= mult_and_quant(signed(X), h(0));
  end if;
end process;
END behavioral;

```

APPENDIX B

1. EXAMPLE SYNTHESIS REPORT

sparse_200_rand Project Status (11/23/2008 - 12:28:13)			
Project File:	sparse_200_rand.isc	Current State:	Synthesized
Module Name:	coeff_filt_1_order_200_sparse_08_coeff_rand_b18_fb16	• Errors:	No Errors
Target Device:	xc4vsx25-12ff668	• Warnings:	2 Warnings
Product Version:	ISE 10.1.02 - WebPACK	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

sparse_200_rand Partition Summary		[-]
No partition information was found.		

Device Utilization Summary (estimated values)			[-]
Logic Utilization	Used	Available	Utilization
Number of Slices	1319	10240	12%
Number of Slice Flip Flops	2388	20480	11%
Number of 4 input LUTs	1866	20480	9%
Number of bonded IOBs	38	320	11%
Number of GCLKs	1	32	3%
Number of DSP48s	40	128	31%

Detailed Reports					[-]
Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Sun Nov 23 12:28:11 2008	0	2 Warnings	61 Infos
Translation Report					
Map Report					

Place and Route Report					
Static Timing Report					
Bitgen Report					

Date Generated: 11/23/2008 - 12:28:13

APPENDIX C

1. DATA TABLES USED FOR ANALYSIS

Filter Type 1 Sine Based Function Chip Utilization (Virtex 4 SX)

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s
Filter type 1	0.00%	8705	3595	16596	38	1	128
	5.00%	7714	3595	14521	38	1	128
	10.00%	6812	3595	12616	38	1	128
	15.00%	5850	3595	10585	38	1	128
	20.00%	4986	3593	8769	38	1	128
	25.00%	4040	3595	6771	38	1	128
	30.00%	3118	3595	4828	32	1	128
	35.00%	2084	3590	2688	38	1	128
	40.00%	1904	3590	2142	38	1	120
	45.00%	1918	3590	1962	38	1	110
	50.00%	1921	3572	1782	38	1	100
	55.00%	1789	3320	1674	38	1	90
	60.00%	1635	3015	1548	38	1	80
	65.00%	1481	2710	1422	38	1	70
	70.00%	1810	3296	1134	38	1	60
	75.00%	1950	3526	882	38	1	50
	80.00%	1311	2375	1854	38	1	40
	85.00%	1079	1935	1472	38	1	30
	90.00%	743	1338	1082	38	1	20
	95.00%	490	835	678	38	1	10
max available		10240	20480	20480	320	32	128

Filter Type 1 Random Based Function Chip Utilization (Virtex 4 SX)

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s
Filter Type 1	0.00%	8705	3595	16596	38	1	128
	10.00%	6497	3599	11986	38	1	128
	20.00%	4800	3600	8396	38	1	128
	30.00%	3016	3600	4636	38	1	128
	40.00%	1911	3600	2142	38	1	120
	50.00%	1926	3582	1782	38	1	100
	60.00%	1638	3021	1548	38	1	80
	70.00%	1818	3311	1134	38	1	60
	80.00%	1319	2388	1866	38	1	40
	90.00%	759	1364	1098	38	1	20
max available		10240	20480	20480	320	32	128

Filter Type 2 Sine Based Function Chip Utilization (Virtex 4 SX)

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s
Filter type 2	0.00%	8705	3595	16596	38	1	128
	5.00%	7758	3590	14613	38	1	128
	10.00%	6719	3577	12439	38	1	128
	15.00%	5636	3595	10128	38	1	128
	20.00%	4902	3577	8602	38	1	128
	25.00%	3896	3423	6810	38	1	128
	30.00%	2536	3281	4230	38	1	128
	35.00%	1661	3186	2528	38	1	128
	40.00%	1620	3097	2466	38	1	120
	45.00%	1562	2970	2308	38	1	110
	50.00%	1475	2798	2214	38	1	100
	55.00%	1441	2717	1998	38	1	90
	60.00%	1293	2428	1890	38	1	78
	65.00%	1283	2394	1842	38	1	70
	70.00%	1187	2200	1582	38	1	58
	75.00%	1160	2134	1564	38	1	50
	80.00%	1011	1851	1348	38	1	40
	85.00%	967	1745	1134	38	1	28
	90.00%	884	1514	1062	38	1	20
	95.00%	441	784	658	38	1	8
max available		10240	20480	20480	320	32	128

Filter Type 2 Random Based Function Chip Utilization (Virtex 4 SX)

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s
Filter Type 2	0.00%	8705	3595	16596	38	1	128
	10.00%	6519	3582	12039	38	1	128
	20.00%	4823	3578	8464	38	1	128
	30.00%	2640	3286	4438	38	1	128
	40.00%	1620	3098	2466	38	1	120
	50.00%	1477	2803	2214	38	1	100
	60.00%	1297	2436	1890	38	1	78
	70.00%	1187	2201	1590	38	1	58
	80.00%	1023	1873	1340	38	1	40
	90.00%	809	1452	990	38	1	20
max available		10240	20480	20480	320	32	128

Filter Type 3 Sine Based Function Chip Utilization (Virtex 4 SX)

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s
Filter type 3	0.00%	8705	3595	16596	38	1	128
	5.00%	7699	3577	14510	38	1	128
	10.00%	6801	3577	12611	38	1	128
	15.00%	5734	3549	10403	38	1	128
	20.00%	4861	3526	8612	38	1	128
	25.00%	3957	3412	6887	38	1	128
	30.00%	2682	3411	4282	38	1	128
	35.00%	1700	3253	2430	38	1	128
	40.00%	1705	3246	2306	38	1	120
	45.00%	1629	3088	2196	38	1	110
	50.00%	1486	2819	2142	38	1	100
	55.00%	1461	2748	1996	38	1	90
	60.00%	1356	2540	1808	38	1	78
	65.00%	1316	2453	1774	38	1	70
	70.00%	1306	2407	1548	38	1	58
	75.00%	1179	2167	1530	38	1	50
	80.00%	1070	1955	1286	38	1	40
	85.00%	1025	1850	1172	38	1	28
	90.00%	847	1518	1146	38	1	20
	95.00%	442	786	657	38	1	8
max available		10240	20480	20480	320	32	128

Filter Type 3 Random Based Function Chip Utilization (Virtex 4 SX)

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s
Filter Type 3	0.00%	8705	3595	16596	38	1	128
	10.00%	6363	3582	11736	38	1	128
	20.00%	4678	3542	8205	38	1	128
	30.00%	2771	3441	4410	38	1	128
	40.00%	1694	3228	2358	38	1	120
	50.00%	1505	2849	2142	38	1	100
	60.00%	1365	2558	1804	38	1	78
	70.00%	1330	2450	1558	38	1	58
	80.00%	1088	1986	1284	38	1	40
	90.00%	868	1556	1098	38	1	20
max available		10240	20480	20480	320	32	128

**Filter Type 3 Random Based Function Chip Utilization (Virtex 4 SX) Power optimized
design goal**

Power optimized goal							
	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s	
Filter type 3	0.00%	8701	3595	16596	38	1	128
	5.00%	7695	3577	14510	38	1	128
	10.00%	6798	3577	12611	38	1	128
	15.00%	5731	3549	10403	38	1	128
	20.00%	4847	3503	8638	38	1	128
	25.00%	3921	3362	6915	38	1	128
	30.00%	2580	3236	4380	38	1	128
	35.00%	1597	3076	2530	38	1	128
	40.00%	1545	2967	2456	38	1	120
	45.00%	1487	2842	2312	38	1	110
	50.00%	1387	2646	2170	38	1	100
	55.00%	1348	2555	2024	38	1	90
	60.00%	1212	2289	1848	38	1	78
	65.00%	1160	2180	1774	38	1	70
	70.00%	1062	1981	1592	38	1	58
	75.00%	1001	1857	1486	38	1	50
	80.00%	892	1643	1308	38	1	40
	85.00%	783	1426	1162	38	1	28
	90.00%	588	1067	1054	38	1	20

Filter Type 1 Sine Based Function Chip Utilization (Virtex 4 LX)

xc4vlx25-12ff668

	sparseness	Slices	Slice Ffs	4 input LUT	IOBs	GCLKs	DSP48s	
Filter type 1	0.00%	x15096	3595	x28490	38	1	48	Over utilized
	10.00%	x13246	3595	x24588	38	1	48	Over utilized
	20.00%	x11443	3595	x20786	38	1	48	Over utilized
	30.00%	9691	3595	17102	38	1	48	
	40.00%	7860	3590	13229	38	1	48	
	50.00%	6570	3572	10544	38	1	48	
	60.00%	4685	3015	7330	38	1	48	
	70.00%	2892	3296	3154	38	1	48	
	80.00%	1311	2375	1854	38	1	40	
	90.00%	743	1338	1082	38	1	20	
max available		10752	21504	21504	448	32	48	

Filter type 3 Power design goal

virtex 4 sx25

Sine Based Filters Component Comparison Chip Utilization (Virtex 4 SX)

Sparseness	Slices			Slice FF			LUTs		
	ft1	ft2	ft3	ft1	ft2	ft3	ft1	ft2	ft3
0.00%	8705	8705	8705	3595	3595	3595	16596	16596	16596
5.00%	7714	7758	7699	3595	3590	3577	14521	14613	14510
10.00%	6812	6719	6801	3595	3577	3577	12616	12439	12611
15.00%	5850	5636	5734	3595	3595	3549	10585	10128	10403
20.00%	4986	4902	4861	3593	3577	3526	8769	8602	8612
25.00%	4040	3896	3957	3595	3423	3412	6771	6810	6887
30.00%	3118	2536	2682	3595	3281	3411	4828	4230	4282
35.00%	2084	1661	1700	3590	3186	3253	2688	2528	2430
40.00%	1904	1620	1705	3590	3097	3246	2142	2466	2306
45.00%	1918	1562	1629	3590	2970	3088	1962	2308	2196
50.00%	1921	1475	1486	3572	2798	2819	1782	2214	2142
55.00%	1789	1441	1461	3320	2717	2748	1674	1998	1996
60.00%	1635	1293	1356	3015	2428	2540	1548	1890	1808
65.00%	1481	1283	1316	2710	2394	2453	1422	1842	1774
70.00%	1810	1187	1306	3296	2200	2407	1134	1582	1548
75.00%	1950	1160	1179	3526	2134	2167	882	1564	1530
80.00%	1311	1011	1070	2375	1851	1955	1854	1348	1286
85.00%	1079	967	1025	1935	1745	1850	1472	1134	1172
90.00%	743	884	847	1338	1514	1518	1082	1062	1146
95.00%	490	441	442	835	784	786	678	658	657

Random Based Filter Component Comparison Chip Utilization (Virtex 4 SX)

sparseness	Slices			Slice FF			LUTs		
	ft1	ft2	ft3	ft1	ft2	ft3	ft1	ft2	ft3
0.00%	8705	8705	8705	3595	3595	3595	16596	16596	16596
10.00%	6497	6519	6363	3599	3582	3582	11986	12039	11736
20.00%	4800	4823	4678	3600	3578	3542	8396	8464	8205
30.00%	3016	2640	2771	3600	3286	3441	4636	4438	4410
40.00%	1911	1620	1694	3600	3098	3228	2142	2466	2358
50.00%	1926	1477	1505	3582	2803	2849	1782	2214	2142
60.00%	1638	1297	1365	3021	2436	2558	1548	1890	1804
70.00%	1818	1187	1330	3311	2201	2450	1134	1590	1558
80.00%	1319	1023	1088	2388	1873	1986	1866	1340	1284
90.00%	759	809	868	1364	1452	1556	1098	990	1098

REFERENCES

- [1] O. Gufstafsson, L. DeBrunner, V. DeBrunner, H. Johansson, "On the Design of Sparse Half- Band Like FIR Filters," in Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on, Pacific Grove, CA, 4-7 Nov. 2007.
- [2] Xilinx, "Virtex 4 User Guide," http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, June 17, 2008.
- [3] Xilinx, "Virtex 4 Family Overview Data Sheet," http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf, September 28, 2007.
- [4] Xilinx, "ExtremeDSP for Virtex-4 FPGAs User Guide," http://www.xilinx.com/support/documentation/user_guides/ug073.pdf, May 15 2008.
- [5] G. C. Schreier, R. Temes, Understanding Delta-Sigma Converters, John Wiley & Sons, 2005.
- [6] Don Johnson, "The Nyquist Frequency," Connexions Website, 2008, [Online; <http://cnx.org/content/m0090/2.7/>; accessed October 24th 2008].
- [7] D.P. Scholnik, "A Parallel Digital Architecture for Delta-Sigma Modulation," in *Proc. MWSCAS-2002 Circuits and Systems The 2002 45th Midwest Symposium on*, 2002, vol. 1, pp I-352-5 vol.1.
- [8] D.P. Scholnik and J.O. Coleman, "Space-Time vector Delta-Sigma Modulation," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2002*, 2002, vol. 3, pp II-719-722 vol.3.
- [9] C. D. Howard, "Minimizing FIR Filter Designs Implemented in FPGAs Utilizing Minimized Adder Graph Techniques," Thesis submitted Fall 2008.
- [10] The Mathworks "Matlab 7.7 product description" Website, 2008, [Online; <http://www.mathworks.com/products/matlab/description1.html> s_cid=ML_b1008_desintro; accessed October 27th 2008].

- [11] PHP Hypertext Preprocessor, “PHP manual” Website, 2008, [Online; <http://www.php.net/manual/en/>; accessed October 28th 2008].
- [12] Xilinx Products and Services, “ISE WebPACK Software” Website, 2008, [Online; http://www.xilinx.com/ise/logic_design_prod/webpack.htm; accessed October 29th 2008].
- [13] S. Patronis, L. DeBrunner, “Sparse FIR Filters and its Impact on FPGA Area Usage,” in Signals, Systems and Computers, 2008. ACSSC 2008. Conference Record of the Forty-Second Asilomar Conference on, Pacific Grove, CA, 4-7 Oct. 2008
- [14] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” in Proc. Of the 2006 ACM/SIGDA 14th *International Symposium on Field Programmable Gate Arrays*, 2006, pp 21-30
- [15] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, B. Troxel, “A Hybrid ASIC and FPGA Architecture,” in Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on 10-14 Nov. 2002 Pages(s): 187-194

BIOGRAPHICAL SKETCH

Sean Gregory Patronis was born September 8th 1980 in Fort Walton Beach, Florida. He received the Bachelor of Science in Electrical Engineering from Florida State University in 2003 where he graduated Magna Cum Laude. After taking some time away from school to peruse a career as a Systems Engineer and Systems Administrator, he came back as a part-time graduate student to earn a Masters of Science degree in Electrical Engineering from Florida State University in 2008. While at Florida State University, he worked as a Systems Engineer for the Office of Technology Integration, served as a Research Assistant for the Department of Electrical and Computer Engineering, and taught a few sections of PEN1136, Underwater Compressed Gas Diving. Sean Patronis has had some of his work published in the Asilomar Conference on Signals Systems and Computers in 2008.