12-31-2004

VLSI implementation of an efficient method for the computation of line spectral frequencies

David L. Reynolds
Rowan University

VLSI IMPLEMENTATION OF AN EFFICIENT METHOD FOR THE COMPUTATION OF LINE SPECTRAL FREQUENCIES

By
David L. Reynolds

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical and Computer Engineering
Major: Engineering (Electrical Engineering)

Approved:

Members of the Committee:

In Charge of Major Work

For the Major Department

For the College

Rowan University
Glassboro, NJ
2004

ABSTRACT

David L. Reynolds

VLSI IMPLEMENTATION OF AN EFFICIENT

METHOD FOR THE COMPUTATION OF

LINE SPECTRAL FREQUENCIES

2004/04

Dr. Ravi P. Ramachandran

Dr. Linda M. Head

Master of Science in Electrical Engineering

In speech coding applications using linear predictive techniques, the computation of line spectral frequencies (LSFs) from the predictor coefficients is an extremely computationally intensive task. The unique properties of the symmetric and antisymmetric polynomial roots limit the region which must be searched, however it is still necessary to perform a root-finding algorithm on a high-order polynomial.

Certain algorithms have been developed to reduce the complexity of the root finding exercise. One such algorithm, developed by Ramachandran and Kabal[1], takes advantage of certain properties of the symmetric and antisymmetric polynomials to map the upper portion of the unit circle onto the real interval [-1,1] by converting the polynomials into a Chebyshev polynomial series representation. Because Chebyshev polynomials may be evaluated efficiently using the Clenshaw recurrence formula, far fewer computations are necessary to search the linear region for zero crossings.

This work investigates the implementation of the Ramachandran-Kabal algorithm in a VLSI design suitable for integration into larger speech processing systems. An implementation exclusively in VHDL is developed. Simulation of the VHDL design is performed and the post-synthesis results evaluated.

TABLE OF CONTENTS

1. INTRODUCTION AND BACKGROUND 1

1.1. MOTIVATION 1

1.2. OBJECTIVES 2

1.3. SCOPE OF THESIS 3

1.3.1. DIGITAL SPEECH CODING 3

1.3.2. COMPUTATION OF LINE SPECTRAL FREQUENCIES 9

1.3.3. NOVEL ALGORITHMS FOR THE EFFICIENT COMPUTATION OF LSFS 9

1.3.4. THE METHOD OF RAMACHANDRAN AND KABAL 10

1.4. VLSI ARCHITECTURE FOR DIGITAL SIGNAL PROCESSING 15

1.4.1. DEVICE TYPES 15

1.4.2. VLSI DESIGN TECHNIQUES FOR DIGITAL SIGNAL PROCESSING 18

1.4.3. SPECIAL CONSIDERATIONS IN EMBEDDED SYSTEMS 19

2. ARCHITECTURAL DESCRIPTION 20

2.1. DESIGN APPROACH 20

2.2. GENERAL ARCHITECTURE FEATURES 21

2.3. VLSI ENTITIES 22

2.3.1. SYMMETRIC AND ANTISYMMETRIC POLYNOMIAL COMPUTATION 22

2.3.2. POLYNOMIAL DEFLATION 25

2.3.3. COMPUTATION OF CHEBYSHEV POLYNOMIAL SERIES COEFFICIENTS 27

2.3.4. CLENSHAW RECURRENCE COMPUTATION 29

2.3.5. DETERMINATION OF ROOT LOCATIONS 32

2.3.6. COMPUTATION OF THE LSFs 35

2.3.7. FLOATING POINT ENTITIES 37

3. DETAILED DESIGN AND SYNTHESIS 38

3.1. VHDL REPRESENTATION 38

3.1.1. VHDL IMPLEMENTATION OF *ATOPQ* 38

3.1.2. VHDL IMPLEMENTATION OF *POLYDIV* 42

3.1.3. VHDL IMPLEMENTATION OF *CHEBFORM* 43
3.1.4. VHDL IMPLEMENTATION OF *CLENSHAW* 44
3.1.5. VHDL IMPLEMENTATION OF *ROOTFINDER* 45
3.1.6. VHDL IMPLEMENTATION OF *ACOS* 46
4. VHDL SIMULATION RESULTS 47
4.1. SIMULATION APPROACH 47
4.2. PRE-SYNTHESIS SIMULATION RESULTS 47
4.2.1. SIMULATION RESULTS: *ATOPQ* 48
4.2.2. SIMULATION RESULTS: *POLYDIV* 49
4.2.3. SIMULATION RESULTS: *CHEBFORM* 51
4.2.4. SIMULATION RESULTS: *CLENSHAW* 53
4.2.5. SIMULATION RESULTS: *ROOTFINDER* 54
4.3. SIMULATION RESULTS OF 10^{TH} ORDER CASE 56
4.3.1. SIMULATION RESULTS: *ATOPQ* 10^{th} ORDER CASE 56
4.3.2. SIMULATION RESULTS: *POLYDIV* 10^{th} ORDER CASE 57
4.3.3. SIMULATION RESULTS: *CHEBFORM* 10^{th} ORDER CASE 57
4.3.4. SIMULATION RESULTS: *CLENSHAW* 10^{th} ORDER CASE 58
4.3.5. SIMULATION RESULTS: *ROOTFINDER* 10^{th} ORDER CASE 59
5. SYNTHESIS 61
5.1. SYNTHESIS PROCESS 61
5.1.1. SYNTHESIS RESULTS 61
5.1.2. POST SYNTHESIS SIMULATION 62
5.2. OVERALL SIMULATION RESULTS 63
6. LAYOUT 65
6.1. LAYOUT OF EACH ENTITY 65
6.1.1. LAYOUT OF *ACOS* ENTITY 65
6.1.2. LAYOUT OF *ATOPQ* ENTITY 66
6.1.3. LAYOUT OF *CHEBFORM* ENTITY 67

6.1.4. LAYOUT OF *POLYDIV* ENTITY 68
6.1.5. LAYOUT OF *CLENSHAW* ENTITY 69
6.1.6. LAYOUT OF *FPMULT* ENTITY 70
6.1.7. LAYOUT OF *FPADD* ENTITY 71
6.1.8. LAYOUT OF *FPDIV* ENTITY 72
6.1.9. LAYOUT OF *ROOTFINDER* ENTITY 73
6.2. OVERALL ASIC FLOORPLAN 74
7. SUMMARY AND DISCUSSION 76
7.1. SUMMARY 76
7.2. CONCLUSIONS 77
7.3. FURTHER WORK 78
8. VHDL LISTINGS 80
8.1. ATPQ VHDL LISTING 80
8.2. *POLYDIV* VHDL LISTING 82
8.3. *CHEBFORM* VHDL LISTING 85
8.4. *CLENSHAW* VHDL LISTING 86
8.5. *ROOTFINDER* VHDL LISTING 89
8.6. *ACOS* VHDL LISTING 92
9. References 96

LIST OF FIGURES

Figure 1: Root Locations of Polynomials *P(z)* and *Q(z)* 8

Figure 2: Plots of Chebyshev Polynomial Series $G_1(x)$ and $G_2(x)$ (12^{th} Order) 14

Figure 3: State Machine Architecture of the *atopq* Entity 24

Figure 4: Misalignment in 10^{th} Order Case 25

Figure 5: State Machine Architecture of the *polydiv* Entity 27

Figure 6: State Machine Architecture of the *chebform* Entity 29

Figure 7: State Machine Architecture of the *clenshaw* Entity 31

Figure 8: State Machine Architecture of the *rootfinder* Entity 34

Figure 9: State Machine Architecture of the *arcos* Entity 36

Figure 10: VHDL Declaration of the *atopq* Entity 39

Figure 11: VHDL State Machine Implementation of the *atopq* Entitiy 40

Figure 12: VHDL Interface to External Floating Point Entities 41

Figure 13: VHDL Declaration of the *polydiv* Entity 43

Figure 14: VHDL Declaration of the *chebform* Entity 44

Figure 15: VHDL Declaration of the *clenshaw* Entity 44

Figure 16: VHDL Declaration of the *rootfinder* Entity 45

Figure 17: VHDL Declaration of the *acos* Entity 46

Figure 18: *atopq* Simulation Timing Diagram 49

Figure 19: *polydiv* Simulation Timing Diagram 51

Figure 20: *chebform* Simulation Timing Diagram 52

Figure 21: *clenshaw* Simulation Timing Diagram 54

Figure 22: *rootfinder* Simulation Timing Diagram 55

Figure 23: *acos* Entity Layout 66

Figure 24: *atopq* Entity Layout 67

Figure 25: *chebform* Entity Layout 68

Figure 26: *polydiv* Entity Layout 69

Figure 27: *clenshaw* Entity Layout 70

Figure 28: *fpmult* Entity Layout 71

Figure 29: *fpadd* Entity Layout 72

Figure 30: *fpdiv* Entity Layout 73

Figure 31: *rootfinder* Entity Layout 74

Figure 32: Overall ASIC Layout 75

LIST OF TABLES

Table 1: *atopq* Simulation Results 48

Table 2: *polydiv* Simulation Results 50

Table 3: *chebform* Simulation Results 52

Table 4: *clenshaw* Simulation Results 53

Table 5: *rootfinder* Simulation Results 55

Table 6: *atopq* Simulation Results (10^{th} Order Case) 56

Table 7: *polydiv* Simulation Results (10^{th} Order Case) 57

Table 8: *chebform* Simulation Results (10^{th} Order Case) 58

Table 9: *clenshaw* Simulation Results (10^{th} Order Case) 59

Table 10: *rootfinder* Simulation Results (10^{th} Order Case) 60

Table 11: Gate Count by Entity 61

Table 12: Performance by Entity 62

Table 13: ASIC Results Versus Expected Output 64

Table 14: Size and Area by Entity 65

1. INTRODUCTION AND BACKGROUND

Vocoders are algorithmic constructs that encode speech for digital transmission over band limited communications channels. Vocoders based on linear predictive analysis are a popular implementation for modern communications systems, many of which are embedded systems such as cellular telephones, digital radios and cryptographic devices.

Linear predictive vocoders encode short segments of a sampled voice signal into a significantly smaller set of parameters based on a model of the human vocal tract. The receiving device can perform a reconstruction of the original speech signal that is a good representation of the original speech signal assuming an adequate bit-rate through the transmission channel.

1.1. MOTIVATION

Linear predictive vocoders are often implemented in embedded systems. Many such systems are very limited in terms of memory and power with cellular telephones being a typical example. Given limitations such as these, implementation of complex algorithms, such as linear prediction, is challenging on such platforms. In addition, the use of general-purpose processors or DSPs sometimes exceeds cost or power constraints. In these cases, ASICs or other customized logic devices may be employed in the implementation of algorithms.

In linear predictive speech coding, it is often desired to compute line spectral frequencies (LSFs) from the linear predictive coefficients for use in transmission[2]. This computation requires the isolation of the roots of high order polynomials. Novel

algorithms have been developed for computing LSFs efficiently[1,3]. In this thesis, the implementation of the algorithm of Ramachandran and Kabal[1] in a VLSI design suitable for use in larger voice coding systems is evaluated.

1.2. OBJECTIVES

The primary objective of this work is to investigate the implementation of a novel algorithm for the computation of LSFs in a VLSI design. This objective is further decomposed as follows:

1. Perform functional decomposition of the chosen algorithm.
2. Develop a VLSI architecture for the implementation of the algorithm.
3. Perform detailed design of the defined architecture in VHDL.
4. Simulate the design at the VHDL level.
5. Perform synthesis of the design using a 0.5um technology.
6. Resimulate the system using VHDL representations of the gate-level implementation generated by the synthesis tool.
7. Layout of the design.

An understanding of the effect of physical constraints of a given fabrication technology on algorithmic performance will be developed. The results will be evaluated for suitability for the target embedded environment.

1.3. SCOPE OF THESIS

The thesis will implement a VLSI design of a DSP algorithm. The design will be implemented entirely in VHDL and will proceed from a VHDL level simulation, through synthesis and post-synthesis simulation accounting for the physical properties of the VLSI technology selected.

1.3.1. DIGITAL SPEECH CODING

1.3.1.1.LINEAR PREDICTIVE SPEECH CODING

Linear predictive methods of voice coding form an important foundation element of many modern voice coding systems. For example, many digital communication systems use methods such as MELP (Mixed Exitation Linear Prediction) or CELP (Code Excited Linear Prediction)[4] to encode voice communication for efficient transmission over band limited channels. Common applications include cellular and digital telephony.

Linear prediction applied to a speech-coding task attempts to find a model of the spectral envelope of a brief segment, or frame, of speech. The form of the model is the all-pole digital filter given by

$$H(z) = \frac{S(z)}{U(z)} = \frac{1}{1 - \sum_{k=1}^{P} a_k z^{-k}}$$

Eq. 1

where the z-transform of the speech signal, $S(z)$, is the output function and $U(z)$ is the input excitation function which is chosen to be either an impulse train or random noise depending upon whether the segment is voiced or unvoiced speech. The a_k coefficients

are computed to give the model an estimate of the spectral envelope of the speech frame being analyzed. This computation is the basis of linear predictive analysis[5].

The sampled speech signals are related to the digital filter by the following difference equation:

$$s(n) = \sum_{k=1}^{P} a_k s(n-k) + u(n)$$

Eq. 2

The term

$$\tilde{s} = \sum_{k=1}^{P} a_k s(n-k)$$

Eq. 3

is the estimate of *s(n)* based on a weighted combination of p previous samples. The goal is to reduce the variance between the actual and estimated speech signal. The prediction error is given by:

$$e(n) = s(n) - \tilde{s} = s(n) - \tilde{s} = \sum_{k=1}^{P} a_k s(n-k)$$

Eq. 4

where *e(n)* is the error signal and $\tilde{s}$ is the estimated speech signal. This indicates that the prediction error sequence is the output of a system described by the following function (when *s(n)* is the input):

$$A(z) = 1 - \sum_{k=1}^{P} a_k z^{-k}$$

Eq. 5

Thus, if Eq. 2 is an exact representation of the system, then $e(n) = u(n)$ and the prediction error filter $A(z)$ is the inverse filter for the system $H(z)$.

$$H(z) = \frac{1}{A(z)}$$

Eq. 6

Passing a speech signal through the filter defined by $A(z)$ results in the removal of near-sample correlations and produces a residual signal[2]. It is the magnitude spectrum of $\frac{1}{A(z)}$ which is the estimate of the spectral envelope of the speech once the a_k coefficients were determined[2].

Given a speech signal, it is necessary to determine a set of predictor coefficients a_k such that a good estimate of the spectral envelope of the speech is obtained[6]. In linear prediction, this is often done through the mean-squared minimization of the prediction error. As speech is time varying, the estimates are based on short segments, or frames, of the sampled speech signal.

A popular method for the determination of the set of predictor coefficients through prediction error minimization is the autocorrelation method. Using this technique, the samples outside of the segment being analyzed are assumed to be zero. The predictor coefficients are computed as part of a system of linear equations, with the system matrix being Toeplitz, which can be efficiently solved using the Levinson-Durbin algorithm. When used in speech transmission applications, the autocorrelation method has the advantage of guaranteeing the stability of $\frac{1}{A(z)}$. Once the predictor coefficients of $A(z)$ have been determined, they must be quantized for transmission through a given

communication channel. Both the a_k coefficients and the linear predictive residual value must be quantized and transmitted such that a receiver can reconstruct the encoded speech frame. One popular quantization scheme is the conversion of the a_k coefficients into line spectral frequencies (LSFs). LSFs have certain properties which make them attractive for transmission. First, they are approximately related to the formant frequencies and bandwidths present in the speech[2]. A distortion measure can be obtained in terms of LSFs which is closely related to the spectral distortion, which should be minimized to ensure adequate fidelity in transmission[2].

The LSFs are determined by the roots of the symmetric and antisymmetric polynomials *P(z)* and *Q(z)* which are related to *A(z)* as follows:

$$P(z) = A(z) + z^{-(p+1)} A(z^{-1})$$

Eq. 7

$$Q(z) = A(z) - z^{-(p+1)} A(z^{-1})$$

Eq. 8

P(z) and *Q(z)* possess certain properties.

1. The roots of *P(z)* and *Q(z)* lie on the unit circle.
2. The roots of *P(z)* and *Q(z)* are simple (no repeated roots exist).
3. The roots of *P(z)* and *Q(z)* interlace on the unit circle.

The LSFs are defined as the angles of the roots of *P(z)* and *Q(z)* with respect to the positive real axis[7]. Because the roots are symmetrical across the real axis, *A(z)* can be completely described by only those LSFs in the upper unit semicircle. Conversion of the a_k coefficients into LSFs preserves the interlacing property which guarantees the stability of $\frac{1}{A(z)}$. Figure 1 shows the root locations of *P(z)* and *Q(z)* for a 12^{th} order system. The roots lie on the unit circle. This implies that a stable $\frac{1}{A(z)}$ can be guaranteed after quantizing the LSFs by ordering them to preserve the interlacing property. This underscores the advantage of using LSFs in practical systems.

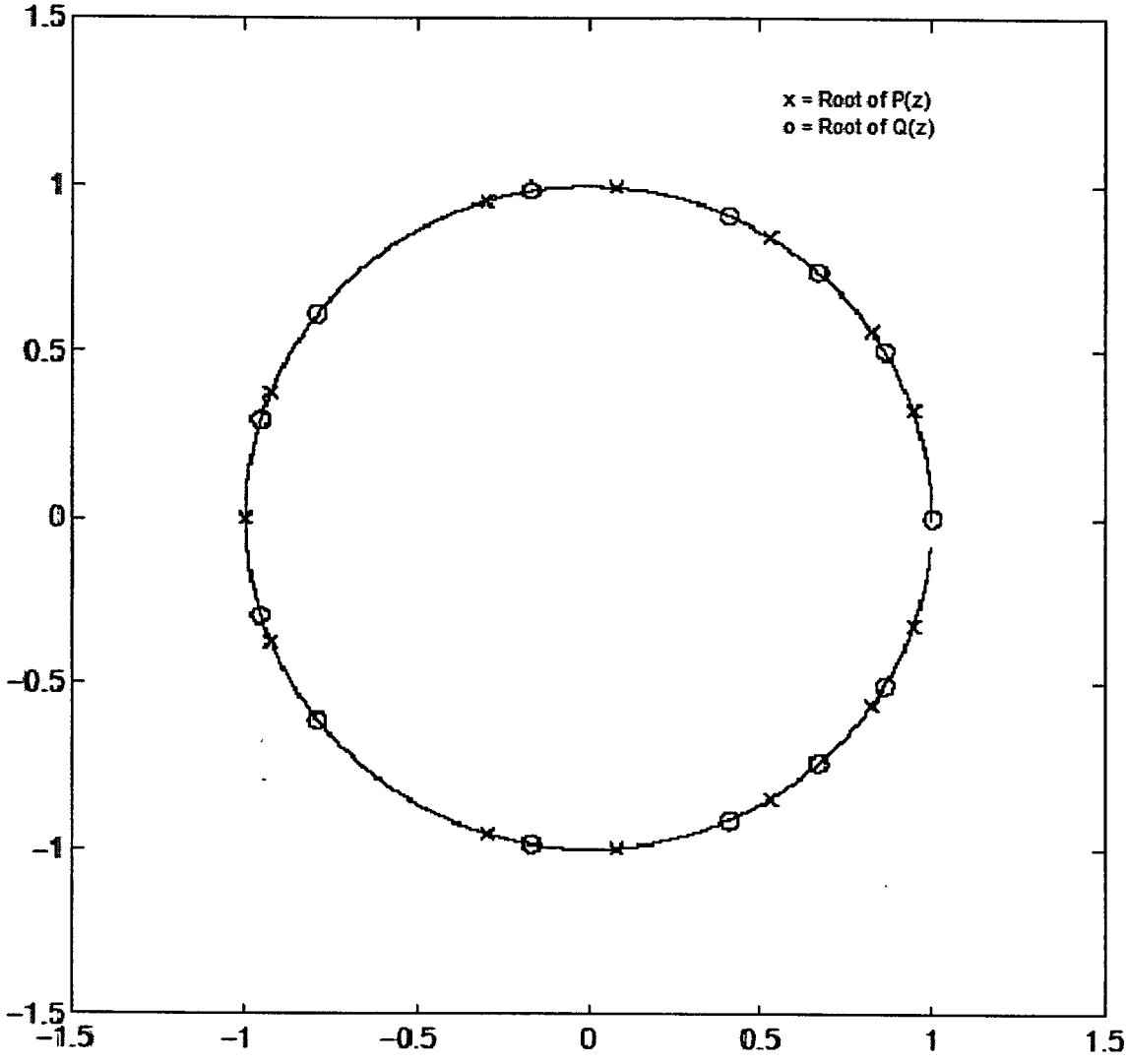

Figure 1: Root Locations of Polynomials *P(z)* and *Q(z)*

These properties are useful in speech transmission applications because in the event of errors, the receiver can easily take corrective action minimizing the impact of the errors on the reconstruction of a given frame of speech.

1.3.2. COMPUTATION OF LINE SPECTRAL FREQUENCIES

Practical linear predictive speech processing systems are generally of 10^{th} or 12^{th} order, depending upon the specific application. For example, speaker recognition may require a 12^{th} order system, however speech coding for communications would use a 10^{th} order system to reduce the amount of data that must be transmitted. Obtaining the LSFs for a given frame of speech entails the computation of the roots of the high order polynomials *P(z)* and *Q(z)*. Isolation of the roots of such high order polynomials consumes significant computational resources. Use of a generalized root finding algorithm in practical applications is not an optimal solution[8], although the special properties of the roots of *P(z)* and *Q(z)* limit the space which must be searched.

1.3.3. NOVEL ALGORITHMS FOR THE EFFICIENT COMPUTATION OF LSFS

Efficient methods have been proposed by Kabal and Ramachandran[1] as well as Wu and Chen[3] for the computation of LSFs. The method of Ramachandran and Kabal has been chosen for implementation in a VLSI design because of its inherent efficiency and the ease with which it can be functionally decomposed. This method constrains the maximum number of computations necessary to isolate the roots of *P(z)* and *Q(z)* making system behavior predictable. The algorithm by Wu and Chen[3] proposes a decimation-in-degree algorithm to obtain two polynomials in x followed by execution of a modified Newton-Raphson method to estimate the roots of the polynomials. The polynomials are deflated as each root location is isolated, and Newton-Raphson applied again as necessary to compensate for inaccuracies as deflation shifts the location of remaining roots. Wu and Chen have demonstrated this method to converge faster than the

Ramachandran and Kabal method in a software implementation, however the increased complexity of the algorithm in terms of both computational steps and conditional logic make it more difficult to implement in a VLSI design.

1.3.4. THE METHOD OF RAMACHANDRAN AND KABAL

This method may be broken down into four basic steps:

1. Compute the symmetric and anti-symmetric polynomial coefficients given the coefficients of *A(z)*.
2. Deflate the resulting polynomials by their trivial roots at $z = +1$ and $z = -1$.
3. Convert the deflated polynomials to their Chebyshev representations.
4. Evaluate the Chebyshev polynomials over the interval [-1,1] and note where the zero crossings occur. These locations determine the LSFs of the system.

This method starts with the p coefficient linear predictive filter given by

$$A(z) = 1 - \sum_{k=1}^{p} a(k) z^{-k}$$

Eq. 9

where *a(k)* are the Linear Predictive (LP) coefficients. The first algorithmic step is to compute the symmetric polynomial *P(z)* and the antisymmetric polynomial *Q(z)* from *A(z)*. The corresponding equations are

$$P(z) = A(z) + z^{-(p+1)}A(z^{-1})$$

Eq. 10

$$Q(z) = A(z) - z^{-(p+1)}A(z^{-1})$$

Eq. 11

The roots of *P(z)* and *Q(z)* are on the unit circle, are simple and interlace. The LSFs are the angles of the roots whose imaginary part is positive. For practical applications, the order *p* is typically 10 or 12, making the isolation of the polynomial roots and arduous task given the high resource cost in most VLSI systems.

Two trivial roots at $z = \pm 1$ are first removed using a simple difference equation[1] that essentially accomplishes polynomial deflation. The remaining roots must be found explicitly. When p is even, we define the deflated polynomials as $G_1(z) = P(z)/1 + z^{-1})$ and $G_2(z) = Q(z)/1 - z^{-1})$. When p is odd, we define the deflated polynomials as $G_1(z) = P(z)$ and $G_2(z) = Q(z)/1 - z^{-2})$. Suppose the orders of $G_1(z)$ and $G_2(z)$ are $2M_1$ and $2M_2$ respectively. When *p* is even, $M_1 = M_2 = p/2$. When *p* is odd, $M_1 = (p+1)/2$ and $M_2 = (p-1)/2$. Note that $G_1(z)$ and $G_2(z)$ have an inherent coefficient symmetry[1]:

$$G_1(z) = 1 + g_1 z^{-1} + \ldots + g_1(M_1)z^{-M_1} + \ldots + g_1(1)z^{-(2M_1-1)} + z^{-2M_1}$$

Eq. 12

$$G_2(z) = 1 + g_2 z^{-1} + \ldots + g_2(M_2)z^{-M_2} + \ldots + g_2(1)z^{-(2M_2-1)} + z^{-2M_2}$$

Eq. 13

The second algorithmic step is to deflate the polynomials *P(z)* and *Q(z)* to get $G_1(z)$ and $G_2(z)$ respectively.

Since only the roots with positive imaginary parts are of interest, the total number of LSFs is $M_1 + M_2 = p$ whether p is odd or even. The LSF vector consists of an ordered set of angles between 0 and π. Using coefficient symmetry, substituting $z = e^{j\omega}$ in the expressions for $G_1(z)$ and $G_2(z)$ and removing the linear phase term results in the following cosine series expansions[1]:

$$G_1(\omega) = 2\cos(M_1\omega) + 2g_1(1)\cos(M_1 - 1)\omega + \ldots + 2g_1(M_1 - 1)\cos\omega + g_1(M_1)$$

Eq. 14

$$G_2(\omega) = 2\cos(M_2\omega) + 2g_2(1)\cos(M_2 - 1)\omega + \ldots + 2g_2(M_2 - 1)\cos\omega + g_2(M_2)$$

Eq. 15

These series may be expressed in the form of Chebyshev polynomials in x by applying the frequency mapping $\cos(m\omega) = T_m(x)$ where $T_m(x)$ is the mth order Chebyshev polynomial in x. The mapping when applied to $G_1(\omega)$ and $G_2(\omega)$ leads to

$$G_1(x) = 2T_{M_1}(x) + 2g_1(1)T_{M_1-1}(x) + \ldots + 2g_1(M_1 - 1)T_1(x) + g_1(M_1)$$

Eq. 16

$$G_2(x) = 2T_{M_2}(x) + 2g_2(1)T_{M_2-1}(x) + \ldots + 2g_2(M_2 - 1)T_2(x) + g_2(M_2)$$

Eq. 17

Any Chebyshev polynomial series in this form can be evaluated efficiently through the application of the Clenshaw Recurrence Formula[9]. Thus each evaluation of N terms may be achieved with N multiplications and $2N$ additions[1]. The third step is to transform $G_1(z)$ and $G_2(z)$ into their Chebyshev polynomial series form $G_1(x)$ and $G_2(x)$ respectively.

Transforming the polynomials into the Chebyshev domain effectively maps the upper part of the unit circle onto a linear region from $x = -1$ to $x = +1$. The roots are isolated through the evaluation of the Chebyshev polynomial series over this region and observing the zero crossings. Figure 2 shows plots of the G_1 and G_2 polynomials and their zero crossings. A zero crossing is detected by observing a sign change in the Chebyshev polynomial series. The root location is then evaluated at a higher resolution in the neighborhood of the sign change. It has been experimentally determined that a coarse resolution of 0.02 and a fine resolution of 0.0015 is adequate to isolate the root to an acceptable precision for 10^{th} and 12^{th} order systems[1]. These are the values used in this design. It should also be noted that the interlacing root peroperty on the unit circle carries over to the Chebyshev domain. Thus, the first root found by starting the search at $x = 1$ will be a root of $G_1(x)$. The second root will be a root of $G_2(x)$. This further increases the efficiency of the algorithm as one can alternate between evaluation of $G_1(x)$ and $G_2(x)$ as roots are found.

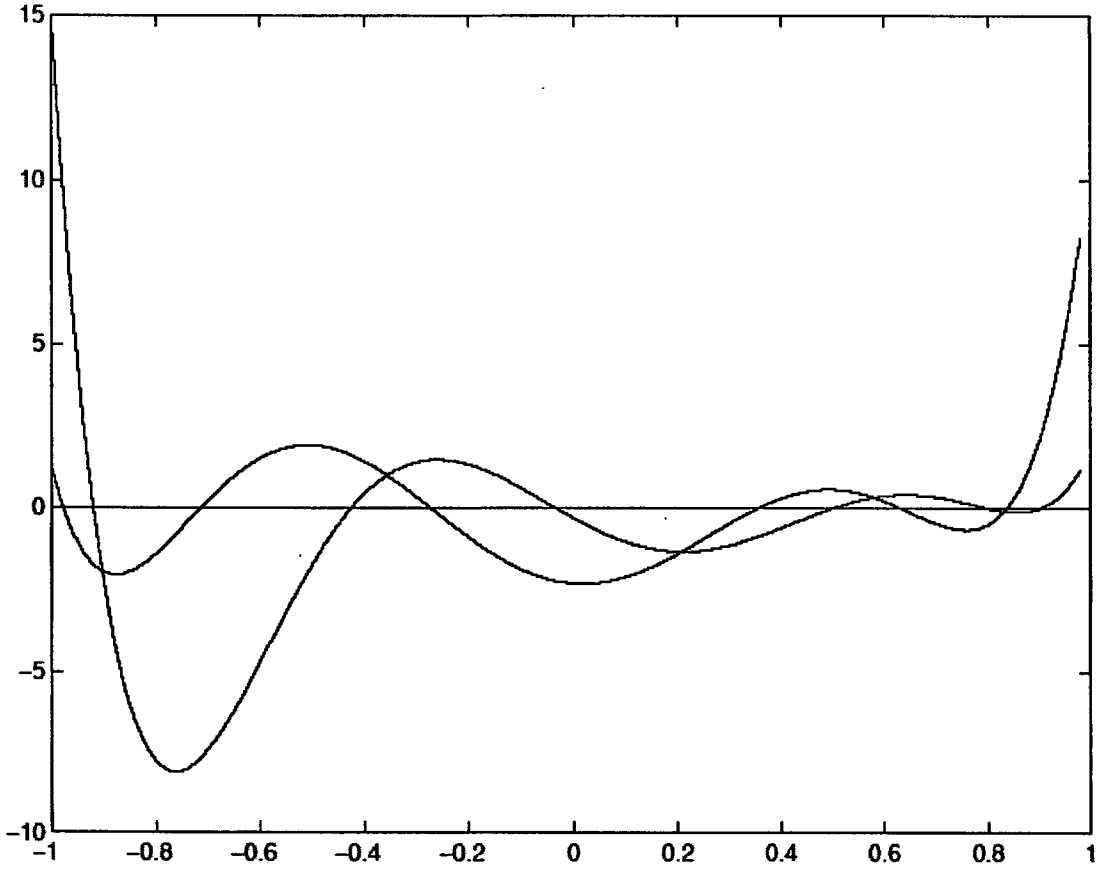

Figure 2: Plots of Chebyshev Polynomial Series $G_1(x)$ and $G_2(x)$ (12^{th} Order)

Finally, the fourth step is to isolate the roots of $G_1(x)$ and $G_2(x)$ as described above. When all the roots are found, the LSFs are computed as the inverse cosine of the roots of $G_1(x)$ and $G_2(x)$.

Note that for the purposes of applying this algorithm to LSF computation, a 10^{th} order system may be seen as a special case of a 12^{th} order system. The coarse and fine resolutions determined by Ramachandran and Kabal may be used for any system of 12^{th} order or lower, however care must be taken not to apply them to higher order systems in which the roots may be closer together and smaller increment values must be used. System orders higher than 12^{th} are rarely seen in practical speech processing applications.

1.4. VLSI ARCHITECTURE FOR DIGITAL SIGNAL PROCESSING

1.4.1. DEVICE TYPES

A variety of semiconductor devices are available for implementing signal processing algorithms in embedded systems. These range from general-purpose devices to application specific integrated circuits (ASICs). The choice of device type depends upon the requirements of the system in question.

1.4.1.1.GENERAL PURPOSE DSP CHIPS

General purpose DSP chips are specialized microprocessors or microcontrollers which are optimized for the types of computations encountered in digital signal processing applications. Generally, they are characterized by multiple functional units, multipliers, data RAM, a fast ALU and on-chip memory sufficient to hold significant portions of the data being operated on[10]. Digital signal processing algorithms benefit from the presence of fast multiplier-accumulators. These elements are important features of general purpose DSP chips and frequently complete a multiply-accumulate operation in one clock-cycle[10].

In contrast to conventional microprocessors, DSP processors make frequent use of multiple busses and memories. This technique allows delivery of multiple operands for single-cycle execution, which improves overall data throughput[10]. Throughput is further enhanced through the use of large on-chip memories which can be accessed with lower overhead than external memories.

1.4.1.2.FLOATING POINT DSP PROCESSORS

To address applications requiring high resolution and precision as well as wide dynamic range, floating point processor architectures offer advantages over integer architectures[10]. Many of the same architectural techniques are employed in floating point processors, such as multiple busses, multiple memory spaces, and fast arithmetic units[10]. Floating point math capabilities in such devices employ 16, 18, 22 or 24 bit mantissas and 4, 6 or 8 bit exponents, with 32 bit schemes with 24 bit mantissas and 8 bit exponents being the most popular implementation[11]. DMA controllers are sometimes used to ensure a constant supply of data to operate on[10].

1.4.1.3.PARALLEL DSP CHIPS

Parallelism and pipelining may be used to considerably enhance performance[11]. Pipelining is a technique where registers are inserted in a data path between stages of combinatorial logic[12]. While very useful in achieving increased data throughput, pipelined systems require more chip space due to the additional registers required[12]. Parallelism can often be used when an operation may be separated and will significantly speed execution time. An example is the separation of real and imaginary parts of a complex number operation in a system with two ALUs present[10].

1.4.1.4.RISC DSP

Reduced instruction set computer (RISC) processors are designed with fewer instructions and a simpler design. With the chip space gained through the elimination of infrequently used instructions, the remaining elements of the processor are highly

optimized for performance. Instructions which are not inherently present must be emulated in software. DSP processors are considered a specialized subset of RISC processors and there is less differentiation between conventional RISC chips and DSPs as this technology matures[10]. RISC DSPs generally use highly parallel and pipelined structures to maximize computational throughput[10].

1.4.1.5.DSP CORES

A DSP core provides a building block for a larger DSP processor system. DSP cores are configurable DSP processors designed for inclusion in a larger VLSI system[11]. Such cores provide the designer with the ability to produce systems which are more efficient than general purpose processors, but do not require the complete design of a DSP processor.

1.4.1.6.APPLICATION SPECIFIC INTEGRATED CIRCUITS (ASICs)

The maximum flexibility and performance in meeting design goals is provided by ASICs. An ASIC provides designers complete freedom to optimize a design for a given application. ASICs are constructed of standardized cells or gate arrays[11] and as their name implies, are designed around specific applications. While ASICs provide designs optimized for a given application, the penalty is increased effort necessary to implement and validate the design[12]. This restricts their use to high volume, low cost applications or in situations where overall performance is of paramount importance.

1.4.2. VLSI DESIGN TECHNIQUES FOR DIGITAL SIGNAL PROCESSING

Certain design practices have been established and are commonly used in VLSI architectures intended for digital signal processing. As previously mentioned, parallelism and pipelining are well established methods to maximize the efficiency of VLSI DSP systems[11]. Simply put, pipelining inserts registers in the data path between combinatorial elements[12]. Latency is introduced in a pipelined system as the pipeline is filled, however subsequent operations may be performed at every clock cycle. The addition of pipeline registers comes at the cost of increased area requirements. Recently, wave pipelining has been used to increase performance beyond what is possible using conventional pipelining techniques. Wave pipelining relies on the interconnection capacitance between elements as intermediate storage[13]. Significant performance increases have been demonstrated using wave pipelining[13], however extreme care must be taken during the design process to ensure proper operation. Signal propagation and interconnection capacitance are critical considerations in the design of wave-pipelined systems[13]. Because registers are not used, significant area is not required as in conventional pipelining.

Parallelism is another example whereby increased performance may be achieved at the cost of increased area requirements. In parallel architectures, operations that would ordinarily be performed sequentially are executed simultaneously by distributing data to multiple operational units that operate in tandem[11]. It should be noted however, that not all computations lend themselves to parallel decomposition[11].

Processors and coprocessors intended for DSP often are equipped with Direct Memory Access (DMA) controllers. DMA controllers allow the DSP core to directly access system memory which is frequently shared with a general purpose host processor.

Data processing can be significantly enhanced because the DSP subsystem is capable of fetching data independent of the host processor.

1.4.3. SPECIAL CONSIDERATIONS IN EMBEDDED SYSTEMS

Embedded systems impose unique considerations in systems design. Common examples of embedded systems in which DSP applications are prevalent are cellular telephony, industrial control and sensor applications, medical device design and cryptographic applications. In each case, it can be assumed that resources, primarily power, memory and processor speed, are limited. Design efforts must observe these inherent limitations. Many of these platforms are portable, battery powered devices and much work has been done in low power design techniques.

2. ARCHITECTURAL DESCRIPTION

2.1. DESIGN APPROACH

The objective of this effort is the design and implementation of the Ramachandran-Kabal algorithm in a VLSI design. The design goals have been established as follows:

1. The design is to be implemented entirely in VHDL.
2. The design is to be optimized for speed and minimal size.
3. A structure is to be chosen suitable for integration into larger systems requiring computation of LSFs.

The initial step in the design approach is the functional decomposition of the Ramachandran-Kabal algorithm. The algorithm lends itself well to decomposition into the following sequential steps:

1. Given the coefficients of $A(z)$, compute the symmetric and anti-symmetric polynomials $P(z)$ and $Q(z)$.
2. Deflate the polynomials by the trivial roots at z = +1 and z = -1.
3. Rearrange the deflated polynomial coefficients into their corresponding Chebyshev polynomial series form $G_1(x)$ and $G_2(x)$.
4. Evaluate the Chebyshev polynomial series over the interval [-1,+1] and identify the location of each zero crossing. Evaluation begins at $x = +1$ and

proceeds to $x = -1$ using an increment of x small enough to detect the zero crossings. The root nearest $x = +1$ will be a root of $G_1(x)$[1].

5. When a zero crossing is found, the local region is re-evaluated to isolate the root location with more precision. Ramachandran and Kabal have determined that a coarse increment of 0.02 is sufficiently small to avoid missing a zero crossing, and a fine resolution of 0.0015 is sufficient to adequately determine the root location[1].
6. Because the interlacing property of the roots is preserved in the Chebyshev domain, when a root of $G_1(x)$ is located, the next root along the x axis will be a root of $G_2(x)$ and vice versa.
7. When all root locations have been found, the LSFs may be computed by evaluating the *arcos(x)*.

Each of these algorithmic steps is isolated and a VLSI entity designed to perform the operation in question. This approach was chosen because it decomposes the design into manageable units and it makes debugging the design significantly easier because each portion of the algorithm may be tested in isolation before moving to the next algorithmic block.

2.2. GENERAL ARCHITECTURE FEATURES

To support the design goal of having a VLSI entity suitable for use in larger systems, it was decided to accommodate input and output data in 32-bit IEEE 754 floating point format, with a 23-bit mantissa, 8-bit exponent and a sign bit. Practical linear predictive

systems are generally of 10^{th} or 12^{th} order[14]. The architecture is designed for a 12^{th} order system. The input to the system will be twelve 32-bit vectors representing the A(z) coefficients in the 32-bit IEEE 754 floating point format. Because 10^{th} order systems may be represented as a special case of 12^{th} order data, the design presented here may be used for 10^{th} order data by simply setting two of the input vectors to zero and shifting the coefficient vectors. In order to simplify hardware implementation for the 10^{th} order case, the coefficients are shifted during computation of the symmetric and antisymmetric polynomials. This procedure is explained in more detail below.

The sequential nature of the algorithm is carried over into the architecture. Each major VHDL entity corresponds to an algorithmic step.

2.3. VLSI ENTITIES

Each of the major algorithmic steps is decomposed into a VLSI entity. Data is presented to each entity in 32-bit IEEE 754 floating-point format. In general, each algorithmic step can be implemented as a series of register transfers, multiplications and additions. For this reason, a similar finite state machine architecture is chosen for each stage. The following sections provide a brief description of the general architecture of each entity.

2.3.1. SYMMETRIC AND ANTISYMMETRIC POLYNOMIAL COMPUTATION

The inputs to the design are the twelve 32-bit floating point coefficients of *A(z)*. The first step in the algorithm is the computation of the symmetric and antisymmetric

polynomials *P(z)* and *Q(z)*. The entity performing this function, *atopq*, accepts as its input the twelve 32-bit vectors representing the input coefficients. The entity performs the following computation to obtain the *P(z)* and *Q(z)* polynomials:

$$P(z) = A(z) + z^{-(p+1)}A(z^{-1})$$

Eq. 18

$$Q(z) = A(z) - z^{-(p+1)}A(z^{-1})$$

Eq. 19

Numerically, this is simply a combination of additions and multiplications of coefficients and the constants +1 and –1. The *atopq* entity implements a state machine which transitions on each clock edge (rising and falling). The state machine shifts the data as required, presents operators to the external floating-point entities, and offloads the results of floating point computations. Figure 3 illustrates the state machine architecture of the *atopq* entity.

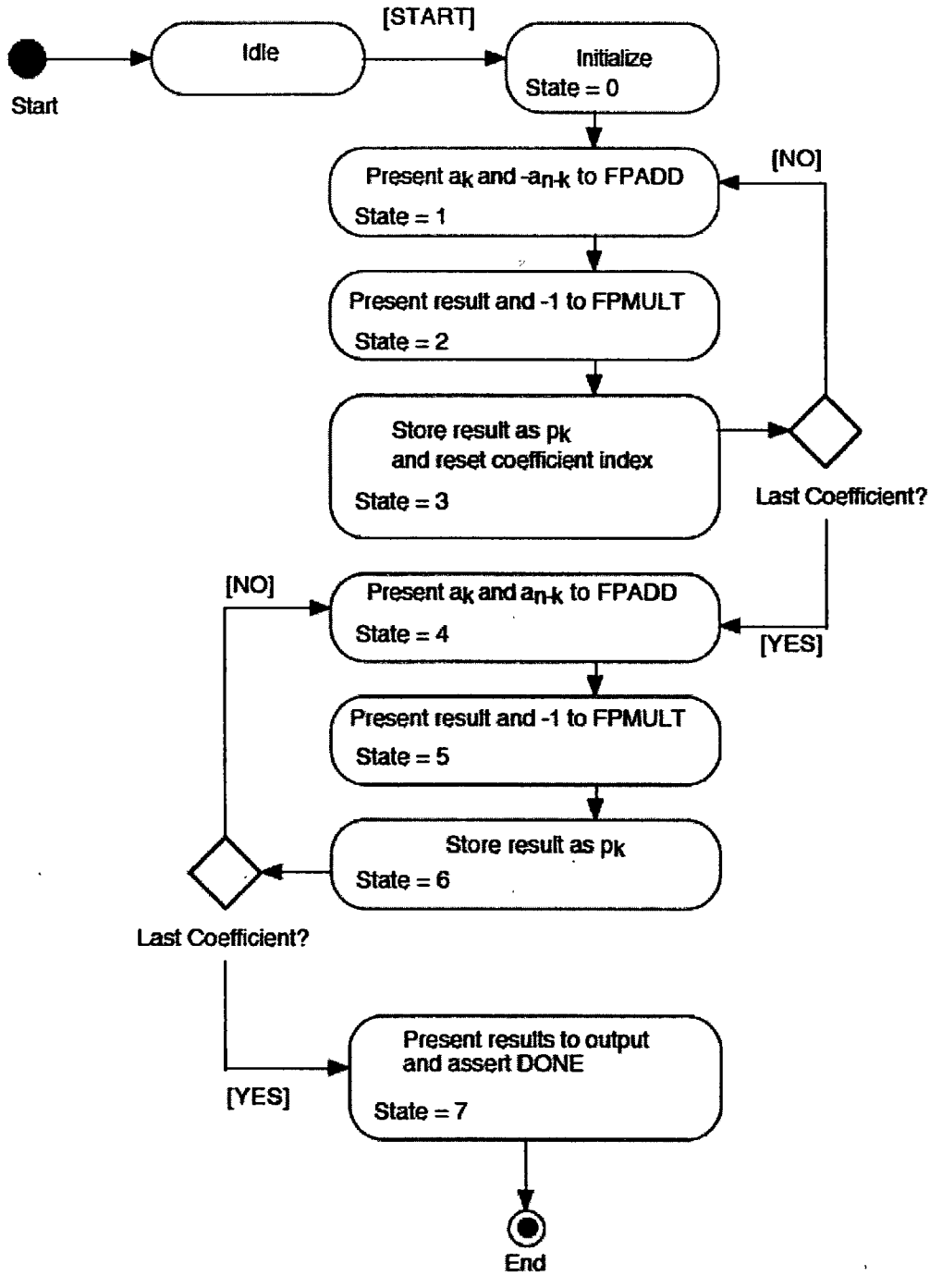

Figure 3: State Machine Architecture of the *atopq* Entity

For handling the case of 10^{th} order systems, the unused high-order coefficient is set to zero. When this condition exists, the resulting *P(z)* and *Q(z)* coefficients are misaligned as shown in Figure 4.

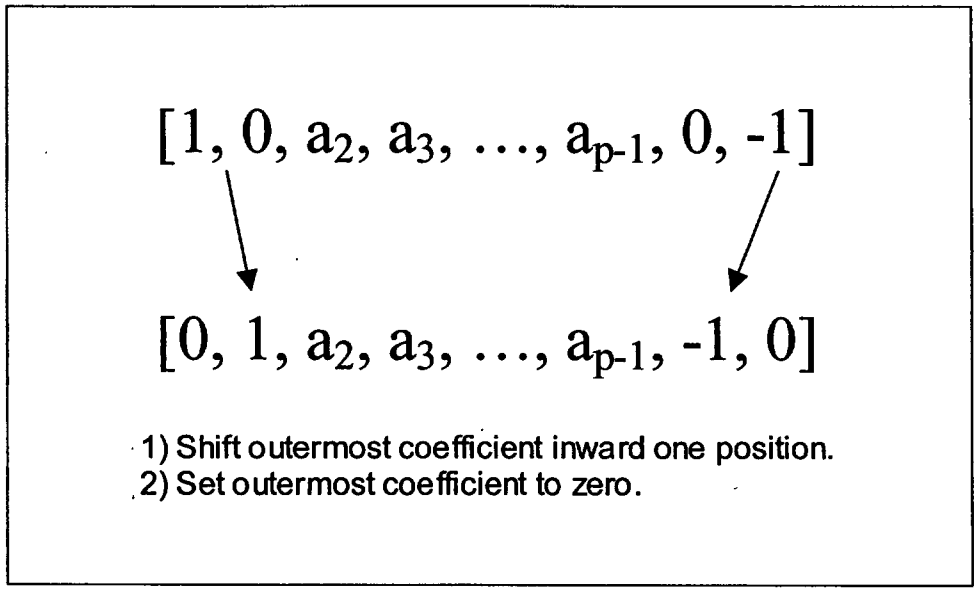

Figure 4: Misalignment in 10^{th} Order Case

The misalignment is simply corrected prior to further processing. In this way, both 10^{th} and 12^{th} order systems may be handled without significantly increasing the complexity of the system. The 10^{th} order data may be further processed by the 12^{th} order architecture following these manipulations.

2.3.2. POLYNOMIAL DEFLATION

The output of the *atopq* entity are the coefficients of the *P(z)* and *Q(z)* polynomials. Each polynomial is 12^{th} order and contains a trivial root which may be removed to simplify root isolation in later stages of the algorithm. The *P(z)* polynomial is deflated by

the trivial root at $z = +1$ while the $Q(z)$ polynomial is deflated by the trivial root at $z = -1$ for the 12^{th} order case. The polynomial deflation may be achieved through synthetic division or coefficient manipulation in the form of a difference equation. Either technique results in a combination of addition and subtraction of coefficients since synthetic division involves multiplication by unity, thus there is little difference in efficiency. The design presented here uses the synthetic division approach.

Because the algorithm is identical regardless of the trivial root to be removed, the entity *polydiv*, which performs the polynomial deflation, accepts the fourteen 32-bit coefficient vectors of one polynomial as it's input. A single bit indicates whether the polynomial is a symmetric or antisymmetric polynomial which determines the trivial root to be removed. The output is the remaining thirteen coefficients following the polynomial deflation. Two *polydiv* entities are used in the design to process the $P(z)$ and $Q(z)$ coefficients in parallel.

Again, a state machine architecture is employed in the *polydiv* entity. The input coefficient vector is iterated through as necessary to perform the synthetic division. Again, the floating point operators are presented to the external floating point adder and multiplier as required and the results offloaded. Figure 5 illustrates the state machine architecture of the *polydiv* entity.

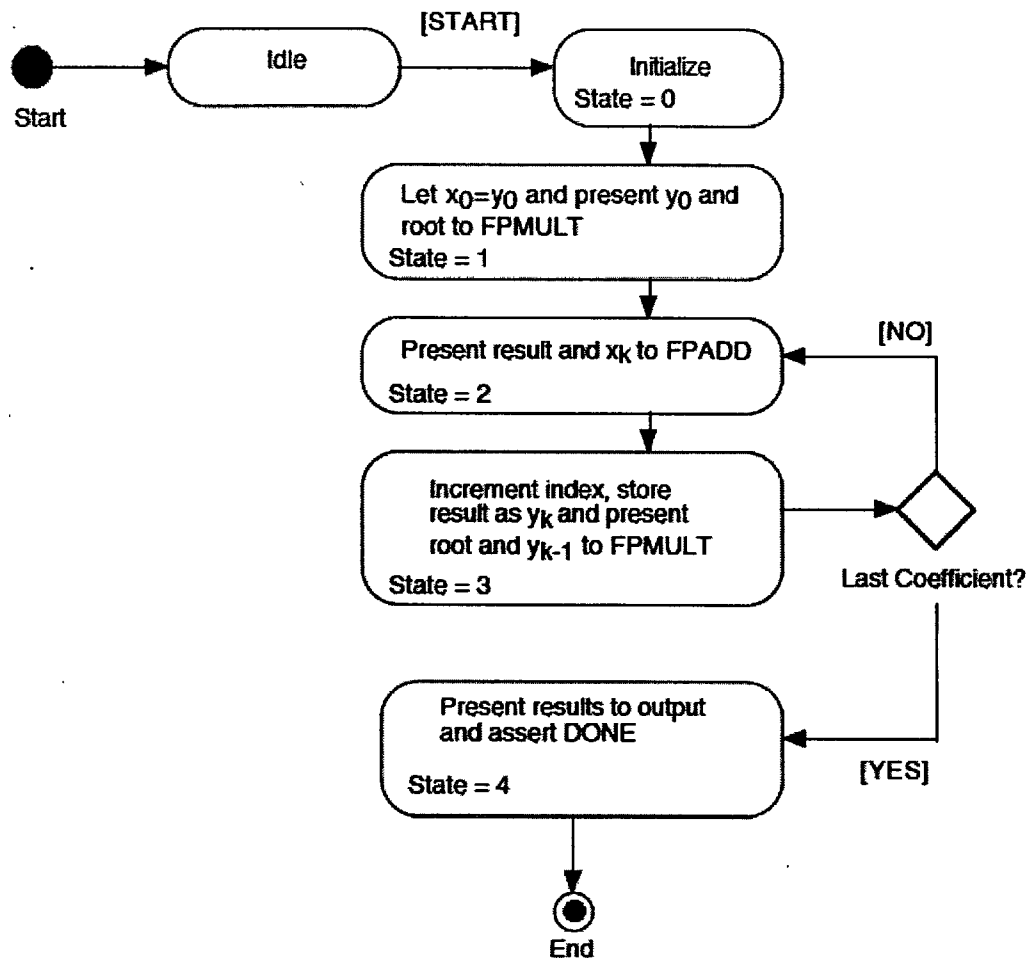

Figure 5: State Machine Architecture of the *polydiv* Entity

2.3.3. COMPUTATION OF CHEBYSHEV POLYNOMIAL SERIES COEFFICIENTS

Given the deflated *P(z)* and *Q(z)* polynomials, it is necessary to next rearrange the coefficients into Chebyshev polynomial series representations. This is a simple matter of arranging the *P(z)* and *Q(z)* polynomial coefficients and multiplying them by two. The Chebyshev polynomial series form of the symmetric and antisymmetric polynomials possess only seven coefficients versus thirteen required by the *z*-domain representation.

The input to the entity which performs the computation of Chebyshev coefficients, *chebform*, accepts as its inputs the first six coefficients of the *P(z)* and *Q(z)* polynomials.

Because these two polynomials are symmetric and antisymmetric, half of the coefficients are redundant information in the conversion to their Chebyshev form. The output is the resulting Chebyshev polynomial series coefficients for both resulting polynomials, a total of two sets of seven coefficients.

The state machine used in the *chebform* entity is very similar in structure to those used in other algorithmic entities. This state machine is illustrated in Figure 6.

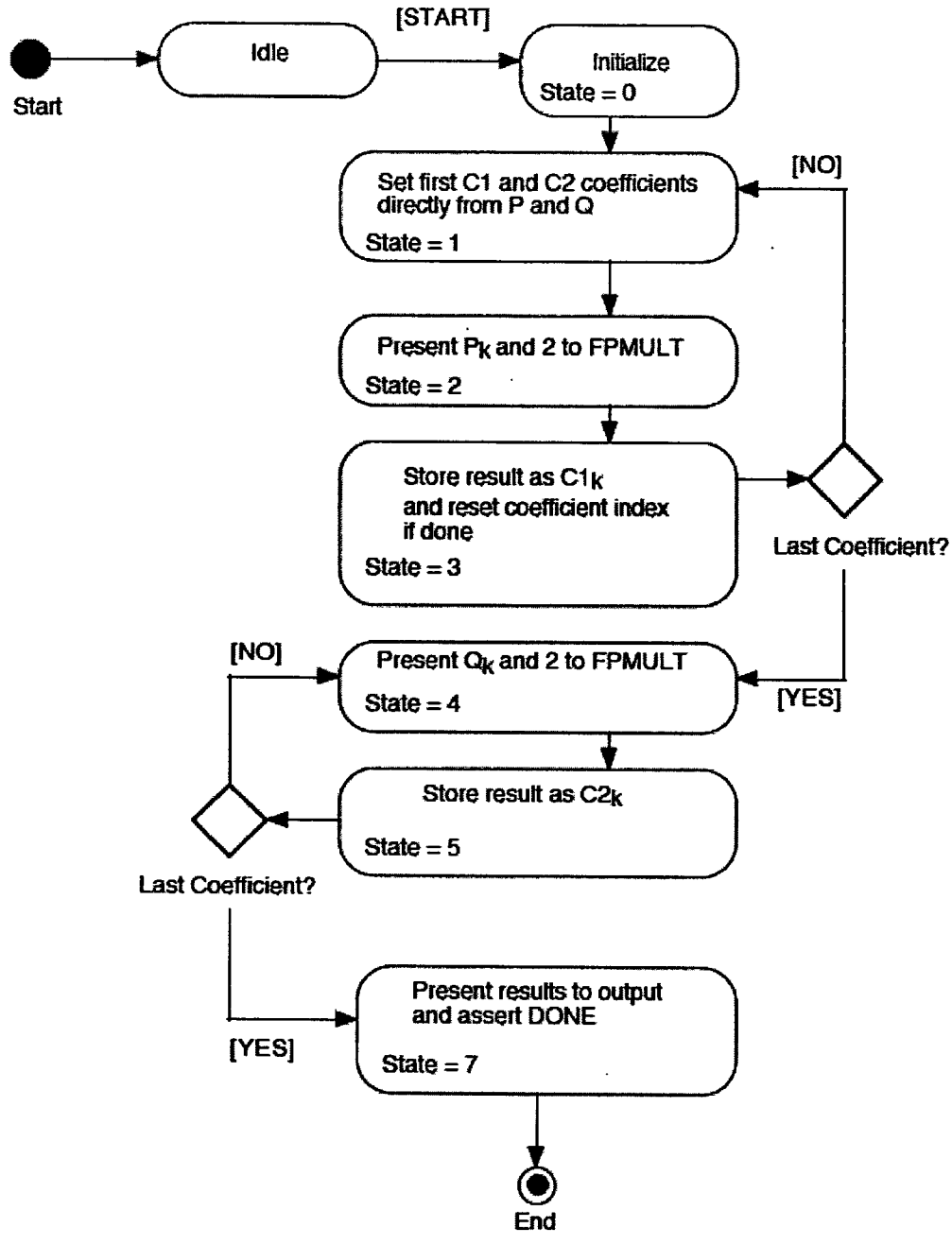

Figure 6: State Machine Architecture of the *chebform* Entity

2.3.4. CLENSHAW RECURRENCE COMPUTATION

With the polynomials in their Chebyshev form, it is now necessary to evaluate the polynomials over the interval [-1,+1]. Evaluation of polynomials in the form of Chebyshev polynomial series may be efficiently performed using the Clenshaw

Recurrence Formula[9]. This recurrence formula may be expressed as follows for a Chebyshev polynomial series expressed in the form $C(x) = \sum_{k=0}^{N-1} c_k T_k(x)$.

$$b(x) = 2xb_{k+1}(x) - b_{k+2}(x) + c_k$$

Eq. 20

With initial conditions $b_N(x) = b_{N+1}(x) = 0$, the recursion is used to calculate $b_0(x)$ and $b_2(x)$. The Chebyshev polynomials may then be evaluated by:

$$C(x) = \sum_{k=0}^{N-1}[b_k(x) - 2xb_{k+1}(x) + b_{k+2}(x)]T_k(x) = \frac{b_0(x) + b_2(x) + c_0}{2}$$

Eq. 21

The *clenshaw* entity performs this evaluation. Its inputs are the c_k coefficients of the Chebyshev polynomial series and a location on the x-axis where the polynomial series is to be evaluated. The recurrence is applied and an output $y = C(x)$ is computed. Figure 7 shows the state machine structure for the *clenshaw* entity.

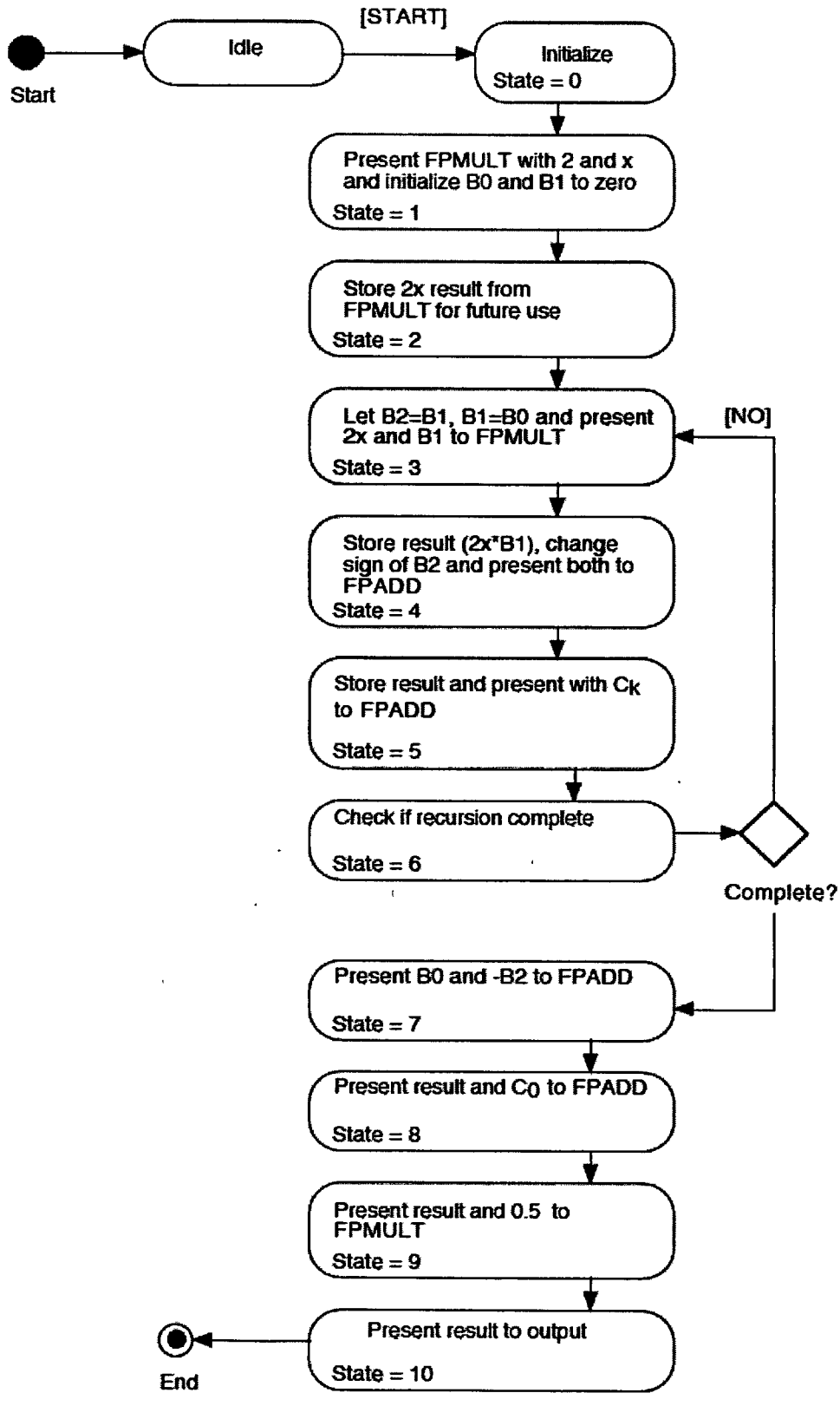

Figure 7: State Machine Architecture of the *clenshaw* Entity

2.3.5. DETERMINATION OF ROOT LOCATIONS

The core of the algorithm is the evaluation of the Chebyshev polynomial series over the interval [-1,+1]. The points at which the polynomial evaluation curves cross are the roots which determine the LSFs of the system. The Ramachandran and Kabal algorithm evaluates the polynomials over this interval with a coarse increment. When a zero crossing is detected, the local region is re-evaluated using a fine increment to isolate the root with more precision. Following the precise determination of the root location, a simple linear interpolation is employed to further isolate the root.

The *rootfinder* entity performs this evaluation using the *clenshaw* entity to efficiently perform each individual evaluation. Zero crossings are identified by observing the sign bit of the 32-bit floating point representation of the result, $y = C(x)$, computed at each point along the x-axis. Given a coarse increment of $\delta = 0.02$, this evaluation must be performed 100 times for the coarse scan and an additional 160 times worst case assuming four bisections per root in a 12^{th} order system. In practice, this will be slightly less depending upon the precise root location in relation to the points at which the polynomial series is evaluated. Given a potential of 260 evaluations, observing the sign changes directly in the floating-point representations is preferable to a full 32-bit floating-point magnitude comparison.

Again a state machine architecture is employed. The *rootfinder* entity accepts the coefficients of the Chebyshev representations of the symmetric and antisymmetric polynomials as its inputs. The state machine evaluates the polynomials starting at $x = +1$. The evaluation continues along the x-axis with the appropriate coefficients and x location

being presented to the *clenshaw* entity. The result returned from *clenshaw* is evaluated for sign changes. In the event a sign change is detected, the x location and increment is changed, but the state machine continues to transition sequentially. Taking advantage of the interlacing property of the roots, the coefficient set is swapped as each root is found.

Following evaluation over the entire [-1,+1] interval, the results are presented at the outputs in their 32-bit floating point format. Figure 8 illustrates the state machine architecture for the *rootfinder* entity. This is the most complex state machine implementation in the design.

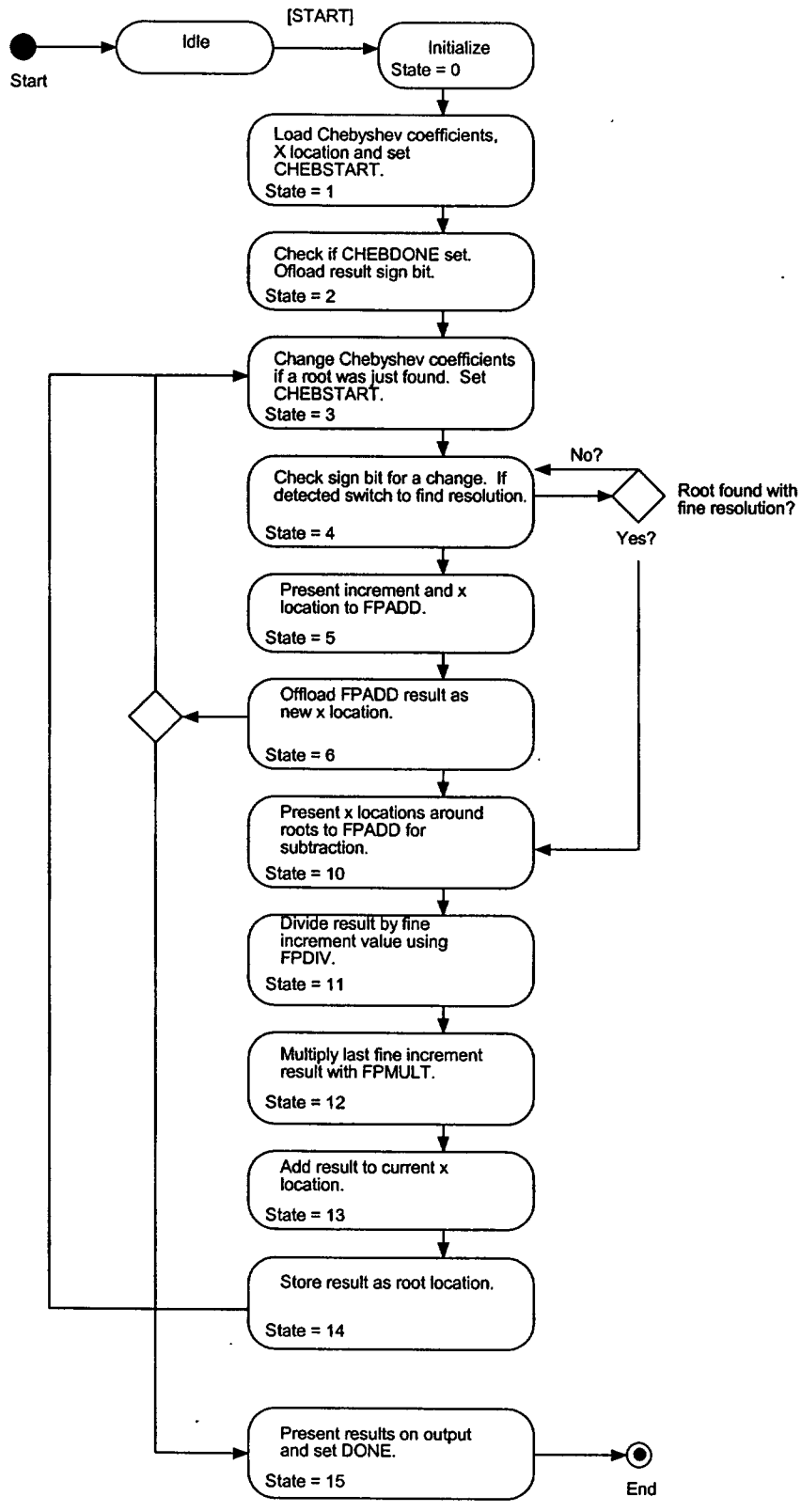

Figure 8: State Machine Architecture of the *rootfinder* Entity

2.3.6. COMPUTATION OF THE LSFs

After the locations of each root has been found on the x-axis, the arccosine must be computed to obtain the line spectral frequencies. To perform this computation, a four term Taylor series expansion is used.

The entity *arccos* performs this computation and is very similar in structure to the *clenshaw* entity in that it iterates through several states presenting intermediate results to the external floating point adder and multiplier and the sums and products offloaded at the next state transition. Figure 9 illustrates the structure of the *arccos* entity.

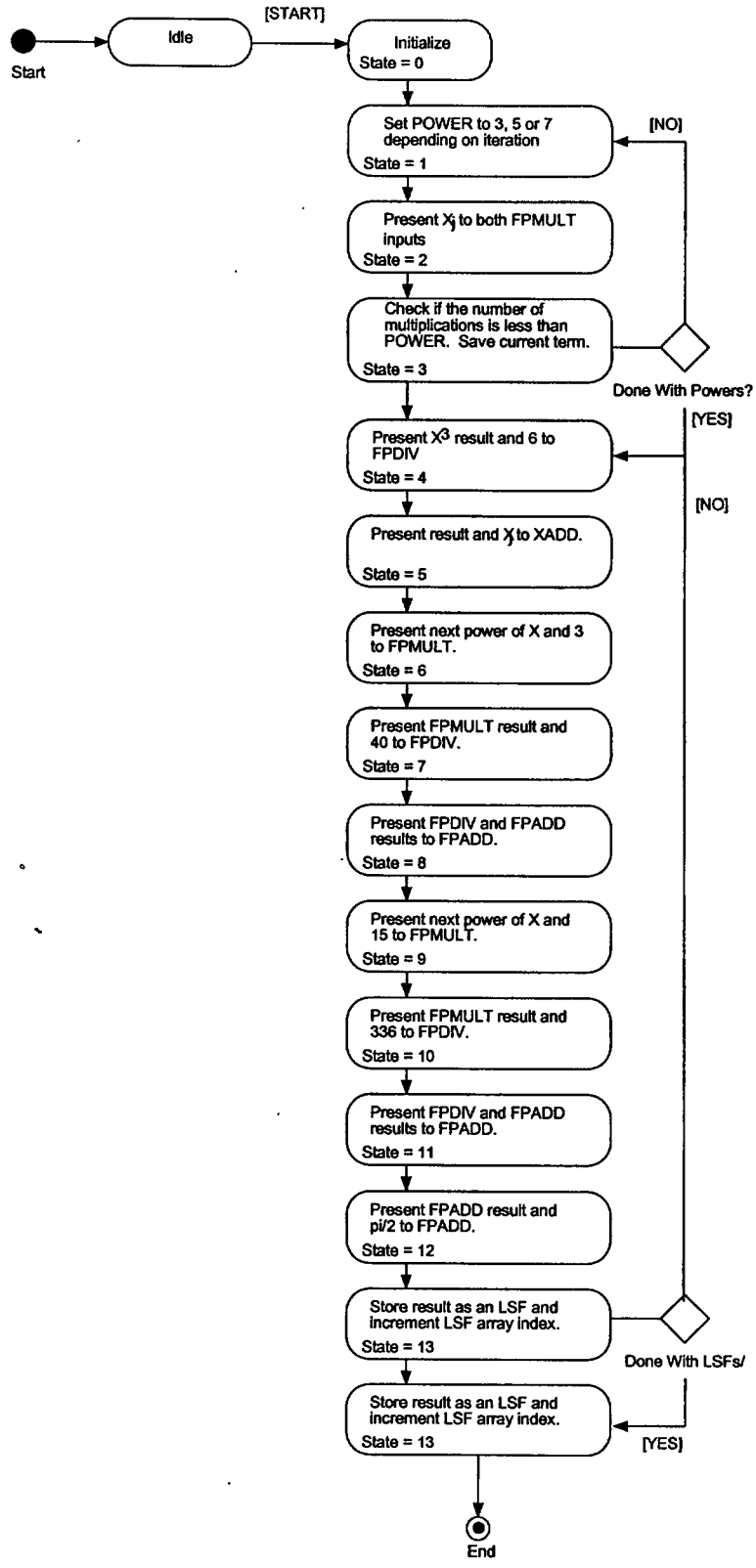

Figure 9: State Machine Architecture of the *arcos* Entity

2.3.7. FLOATING POINT ENTITIES

Because this design is intended for implementation into larger systems, it was assumed that floating-point units would be available as a common resource. The floating-point entities used are simple combinatorial implementations of a 32-bit adder, multiplier and divider. Two operands are presented to the entity and the result is presented to the output. These entities are not clocked, so when used in a design of this nature, care must be taken that timing constraints associated with performing these operations are not exceeded.

3. DETAILED DESIGN AND SYNTHESIS

3.1. VHDL REPRESENTATION

A fundamental goal of this design effort is the expression of the complete design in VHDL. This allows implementation in various ASIC technologies as well as FPGAs if size requirements can be met. Given the architecture previously described, the implementation in VHDL is a straightforward process. The state machine structure of each entity results in similarity in the VHDL implementations.

3.1.1. VHDL IMPLEMENTATION OF *ATOPQ*

The *atopq* entity is typical of the state machine architecture used repeatedly throughout the design and will be discussed thoroughly here. Other entities sharing similar structures will refer back to the *atopq* implementation when design details are discussed.

As described in the architectural sections, the input to the system is a set of twelve 32-bit vectors which contain the IEEE 754 floating point representations of the linear predictor coefficients of a system. The *atopq* entity is the first to process these inputs and produce the coefficients of the symmetric and antisymmetric polynomials $P(z)$ and $Q(z)$. Thus, the input to the *atopq* entity is twelve 32-bit vectors. In addition, clock and control signals are also inputs to the entity. Figure 10 shows the VHDL declaration of the *atopq* entity.

```
entity atopq is
        port(

        -- The A,P and Q ports are for the A(z) coefficient inputs and the P(z) adn
        -- Q(z) coefficient outputs
        A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11          :in std_logic_vector(31 downto 0);
        P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13:out std_logic_vector(31 downto 0);
        Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13:out std_logic_vector(31 downto 0);
        -- The X and Y outputs are for floating point variables destined for the adder
        -- or multiplier.
        XADD,YADD                                      :out std_logic_vector(31 downto 0);
        ZADD                                           :in std_logic_vector(31 downto 0);
        XMULT,YMULT                                    :out std_logic_vector(31 downto 0);
        ZMULT                                          :in std_logic_vector(31 downto 0);
        -- The START bit is asserted when the A(z) coefficients have been presented and
        -- will start execution of the state machine.  The CLK input is the system clock.
        -- The DONE output will be asserted when the final data is placed on the P and Q
        -- output ports.
        START                                          :in std_logic;
        CLK                                            :in std_logic;
        DONE                                           :out std_logic);
end atopq;
```

Figure 10: VHDL Declaration of the *atopq* Entity

The outputs of this entity are two sets of 32-bit floating point representations of the *P(z)* and *Q(z)* polynomials as well as a discrete *DONE* signal used to indicate to the next entity that data presented on *atopq*'s output is stable and ready for use in further processing.

The state machine implementation is quite straightforward. Once data is presented on the input of *atopq*, the *START* bit is asserted. This triggers the internal state machine to proceed through it's processing. Figure 11 shows a partial state machine implementation of *atopq* in VHDL. The state machine remains in state 0 until the *START* bit becomes logical 1. At that time, all internal variables are initialized to their initial conditions and the state variable transitions from 0 to 1. At each consecutive state, a subset of the operations necessary for the entity to accomplish its function is performed.

```
case STATE is

        when 0 =>
        -- The first thing we do is copy the inputs into the local variable for
        -- processing
        A(0) := A0;
        A(1) := A1;
        A(2) := A2;
        A(3) := A3;
        A(4) := A4;
        A(5) := A5;
        A(6) := A6;
        A(7) := A7;
        A(8) := A8;
        A(9) := A9;
        A(10) := A10;
        A(11) := A11;
        ...
        ...
        STATE := 1;
        INDEX := 0;

        when 1 =>
        ...
        ...
        STATE := 2;

        when 2 =>
        ...
        ...
        STATE := 3;

        when 3 =>
        P(index + 1) := ZMULT; -- added + 1
        if INDEX = 11 then -- changed from 10
                INDEX := 0;
                STATE := 4;
        else
                INDEX := INDEX + 1;
                STATE := 1;
        end if;

        ...
        ...
        ...

        when 7 =>
        P0 <= P(0);
        P1 <= P(1);
        P2 <= P(2);

        ...
        ...
        ...

        Q12 <= Q(12);
        Q13 <= Q(13);
        XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        STATE := 8;

        when others =>
        --DONE <= '1';

end case;
```

Figure 11: VHDL State Machine Implementation of the *atopq* Entitiy

If floating point operations must be performed, the external entities *fpmult*, *fpadd*, and *fpdiv* are used. For an algorithmic entity such as *atopq* to make use of these floating point entities, a set of ports are provided. Referring to Figure 10, it can be seen that *XADD*, *YADD*, *ZADD*, *XMULT*, *YMULT*, and *ZMULT* ports are provided. The X and Y ports are outputs through which *atopq* presents operands to the external floating point entities. The results are presented on the Z ports by the entities when the operation is complete. In this design, the floating-point entities are strictly combinatorial and are not clocked. Figure 12 shows several states in the state machine of *atopq* where operands are presented to external floating point entities.

```
...
...
when 1 =>
XADD <= A(index);
TEMPF := A(11 - index);
TEMPF(31) := not TEMPF(31);
YADD <= TEMPF;
STATE := 2;

when 2 =>
XMULT <= ZADD;
YMULT <= neg_one_float;
STATE := 3;

when 3 =>
P(index + 1) := ZMULT; -- added + 1
...
...
when 4 =>
XADD <= A(INDEX);
YADD <= A(11 - INDEX);
STATE := 5;

when 5 =>
XMULT <= ZADD;
YMULT <= neg_one_float;
STATE := 6;

when 6 =>
Q(index + 1) := ZMULT;
...
```

Figure 12: VHDL Interface to External Floating Point Entities

In this instance, *atopq* presents two floating point operands to the external *fpadd* entity in state 1. The state transitions to state 2 on the next clock cycle and the result of the floating point addition is read from the *ZADD* port and immediately loaded as an operand to the *XMULT* port together with another operand. The output of the external *fpmult* entity is then presented on the *ZMULT* port which is read and stored in state 4. All interfaces to external floating-point entities follow this type of process.

Note that because several entities share the floating-point resources, the output ports are placed in a high impedance state when the floating point entities are not in use. This design technique precludes the use of floating point entities by more than one algorithmic block of the design at any given time.

3.1.2. VHDL IMPLEMENTATION OF *POLYDIV*

The assertion of the *DONE* output of the *atopq* entity causes the *polydiv* entity to begin processing. The outputs of *atopq* are presented to the inputs of the two *polydiv* entities that remove the trivial roots from the polynomials. The VHDL declaration of *polydiv* is shown in Figure 13.

```
entity polydiv is
        port(

        -- X0 to X13 is the input to the polynomial divider
        X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13:in std_logic_vector(31 downto 0);
        -- Y0 to Y12 is the output of the polynomial division
        Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9,Y10,Y11,Y12     :out std_logic_vector(31 downto 0);
        -- XADD,YADD and ZADD are interfaces to an external floating point adder
        XADD,YADD                                      :out std_logic_vector(31 downto 0);
        ZADD                                           :in std_logic_vector(31 downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT                                    :out std_logic_vector(31 downto 0);
        ZMULT                                          :in std_logic_vector(31 downto 0);
        -- PQSEL selects if we're working with a P or Q polynomial and removes the
appropriate root
        PQSEL                                          :in std_logic;

        CLK                                            :in std_logic;
        START                                          :in std_logic;
        DONE                                           :out std_logic);
end polydiv;
```

Figure 13: VHDL Declaration of the *polydiv* Entity

The *PQSEL* input is used by *polydiv* to determine which root is to be removed.

The structure of *polydiv* is similar to *atopq*. Each state defined in the architecture is implemented using VHDL case statements exactly as implemented in other state machine based entities as shown in Figure 12. Again, when floating point operations are required, operands are presented to output ports connected to the *fpmult*, *fpadd*, and *fpdiv* entities as required.

3.1.3. VHDL IMPLEMENTATION OF *CHEBFORM*

When the *polydiv* entity completes processing, the coefficients of the *P(z)* and *Q(z)* polynomials are presented to the output and the *DONE* bit is asserted. The *DONE* signal from *atopq* is connected to the *START* signal and the coefficient outputs are connected to the inputs of the *chebform* entity. Figure 14 shows the declaration of the *chebform* entity.

```
entity chebform is
        port (
        -- P0-P10 and Q0-Q10 are the P(z) and Q(z) polynomial coefficients
        P0,P1,P2,P3,P4,P5,P6                        :in std_logic_vector(31 downto 0);
        Q0,Q1,Q2,Q3,Q4,Q5,Q6                        :in std_logic_vector(31 downto 0);
        -- C1_0-C1_6 and C2_0-C2_6 are the resulting Chebyshev coefficients
        C1_0,C1_1,C1_2,C1_3,C1_4,C1_5,C1_6          :out std_logic_vector(31 downto 0);
        C2_0,C2_1,C2_2,C2_3,C2_4,C2_5,C2_6          :out std_logic_vector(31 downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT                                 :out std_logic_vector(31 downto 0);
        ZMULT                                       :in std_logic_vector(31 downto 0);
        CLK                                         :in std_logic;
        START                                       :in std_logic;
        DONE                                        :out std_logic);
end chebform;
```

Figure 14: VHDL Declaration of the *chebform* Entity

The state machine implementation of *chebform* is identical in structure to Figure 11.

3.1.4. VHDL IMPLEMENTATION OF *CLENSHAW*

The *clenshaw* entity isused by *rootfinder* as it scans the region [-1,+1] in search of zero crossings. The *clenshaw* entity performs the efficient computation of the Chebyshev polynomial at a given point. The input to *clenshaw* are the coefficients of the Chebyshev polynomial in question, and the x point at which it is to be evaluated. Like the other entities, a *START* and *DONE* signal are provided to start the internal state machine of *clenshaw* and to signal to external systems when the computation is complete. Figure 15 shows the declaration of the *clenshaw* entity.

```
entity clenshaw is
        port (

        -- C0-C6 are the Chebyshev polynomial coefficients.
        C0,C1,C2,C3,C4,C5,C6    :in std_logic_vector(31 downto 0);
        -- X is the input to the polynomial evaluation.
        X                       :in std_logic_vector(31 downto 0);
        -- Y is the output of the polynomial evaluation.
        Y                       :out std_logic_vector(31 downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT             :out std_logic_vector(31 downto 0);
        ZMULT                   :in std_logic_vector(31 downto 0);
        XADD,YADD               :out std_logic_vector(31 downto 0);
        ZADD                    :in std_logic_vector(31 downto 0);
        CLK                     :in std_logic;
        START                   :in std_logic;
        DONE                    :out std_logic);
end clenshaw;
```

Figure 15: VHDL Declaration of the *clenshaw* Entity

3.1.5. VHDL IMPLEMENTATION OF *ROOTFINDER*

The main entity in the algorithm is *rootfinder*. This entity uses *clenshaw* to evaluate the Chebyshev polynomials over the interval [-1,+1]. Following each evaluation, the sign bit of the result from the polynomial evaluation is checked for a sign change. When a sign change is detected, the increment is adjusted to the fine resolution scan and the state machine is executed again over the region where the root was located.

Following subsequent detection of the zero crossing using the fine resolution scan, the two evaluations on either side of the actual root location are used in a linear interpolation computation to further refine the precise root location. The linear interpolation is performed in another portion of the state machine for *rootfinder*. Figure 16 shows the VHDL declaration for *rootfinder*.

```
entity rootfinder is
        port (
        -- C1 and C2 are the Chebyshev coefficients
        C1_0,C1_1,C1_2,C1_3,C1_4,C1_5,C1_6    :in std_logic_vector(31 downto 0);
        C2_0,C2_1,C2_2,C2_3,C2_4,C2_5,C2_6    :in std_logic_vector(31 downto 0);
        -- X0-X10 are the zero crossing locations in the Chebyshev domain
        X0,X1,X2,X3,X4,X5,X6,X7,X8,X9         :out std_logic_vector(31 downto 0);
        -- CL0-CL5 and CX are interfaces to the Clenshaw evaluator.
        CL0,CL1,CL2,CL3,CL4,CL5,CL6,CX              :out std_logic_vector(31 downto 0);
        CY                                    :in std_logic_vector(31 downto 0);
        -- the following are interfaces to an external floating point units
        XMULT,YMULT                           :out std_logic_vector(31 downto 0);
        ZMULT                                 :in std_logic_vector(31 downto 0);
        XADD,YADD                             :out std_logic_vector(31 downto 0);
        ZADD                                  :in std_logic_vector(31 downto 0);
        XDIV,YDIV                             :out std_logic_vector(31 downto 0);
        ZDIV                                  :in std_logic_vector(31 downto 0);
        CLK                                   :in std_logic;
        START                                 :in std_logic;
        CHEBDONE                               :in std_logic;
        CHEBSTART                              :out std_logic;
        DONE                                  :out std_logic;
end rootfinder;
```

Figure 16: VHDL Declaration of the *rootfinder* Entity

3.1.6. VHDL IMPLEMENTATION OF *ACOS*

The output of *rootfinder* is a set of zero crossing locations in the Chebyshev domain. To obtain LSFs, the arccosine must be computed on each of these. The *acos* performs a simple Taylor series expansion approximation of the arccosine on each of the locations on the *x*-axis where a zero crossing was detected.

The *acos* entity accepts the root location as its input and produces the corresponding LSFs on its outputs.

```
entity acos is
        port (
        -- X0-X9 are the zero crossing locations in the Chebyshev domain
        X0,X1,X2,X3,X4,X5,X6,X7,X8,X9                          :in std_logic_vector(31
downto 0);
        -- LSF0-9 are the line spectral frequencies.
        LSF0,LSF1,LSF2,LSF3,LSF4,LSF5,LSF6,LSF7,LSF8,LSF9     :out std_logic_vector(31
downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT             :out std_logic_vector(31 downto 0);
        ZMULT                   :in std_logic_vector(31 downto 0);
        XADD,YADD               :out std_logic_vector(31 downto 0);
        ZADD                    :in std_logic_vector(31 downto 0);
        XDIV,YDIV               :out std_logic_vector(31 downto 0);
        ZDIV                    :in std_logic_vector(31 downto 0);
        CLK                     :in std_logic;
        START                   :in std_logic;
        DONE                    :out std_logic);
end acos;
```

Figure 17: VHDL Declaration of the *acos* Entity

4. VHDL SIMULATION RESULTS

4.1. SIMULATION APPROACH

The initial step in simulation is to confirm that the VHDL implementation of each entity produces expected results. These implementations will be used as the input to the synthesis tool. Test benches have been developed to exercise each entity with known inputs and verify that outputs are as expected.

Following synthesis, the VHDL representations provided by the synthesis tool are used to provide a more accurate model in terms of the actual implementation arrived at by the synthesis tool. The gate-level models used in the VHDL entities generated by the synthesis tools are those associated with the synthesis library used. In this case, the AMI C5 gate models from the Mentor Graphics ASIC Design Kit are used. Ideally, the output of the original VHDL models will be identical to those generated by the synthesis tool when exercised by the same test bench. It is important to note that these models do not account for parasitic effects that are present in the completely placed and routed design. All simulations were conducted using Modelsim from Mentor Graphics.

4.2. PRE-SYNTHESIS SIMULATION RESULTS

Each entity was exercised using the test benches to verify correct performance. The results are presented in the following sections for each entity with $A(z)$ coefficients obtained from a reference software implementation of the algorithm for a single frame of speech.

4.2.1. SIMULATION RESULTS: *ATOPQ*

The *atopq* entity is presented with the test *A(z)* coefficients. Following assertion of the *START* input, the outputs are presented after 38 clock cycles. Table 1 shows the outputs of *atopq* versus expected results. Results are shown in floating point format to highlight numerical differences due to precision issues.

Table 1: *atopq* Simulation Results

Input	Expected P(z) Coefficients	Computed P(z) Coefficients	Expected Q(z) Coefficients	Computed Q(z) Coefficients
0.929777	1	1	1	1
-0.403188	-0.9664054	-0.966405	-0.8931486	-0.893149
-0.266029	0.4418179	0.441818	0.3645581	0.364558
0.0103105	0.2182245	0.218224	0.3138335	0.313834
-0.264474	0.2265575	0.226557	-0.2471785	-0.247179
0.117257	-0.248127	-0.248127	0.777075	0.777075
0.0645595	-0.0526975	-0.052697	-0.1818165	-0.181817
-0.512601	0.0526975	0.052697	-0.1818165	-0.181817
0.236868	0.248127	0.248127	0.777075	0.777075
-0.0478045	-0.2265575	-0.226557	-0.2471785	-0.247179
0.0386299	-0.2182245	-0.218224	0.3138335	0.313834
-0.0366284	-0.4418179	-0.441818	0.3645581	0.364558
	0.9664054	0.966405	-0.8931486	-0.893149
	-1	-1	1	1

The outputs of *atopq* as exercised by the test bench matched expected results. The outputs were resolved in 38 clock cycles as can be seen in Figure 18.

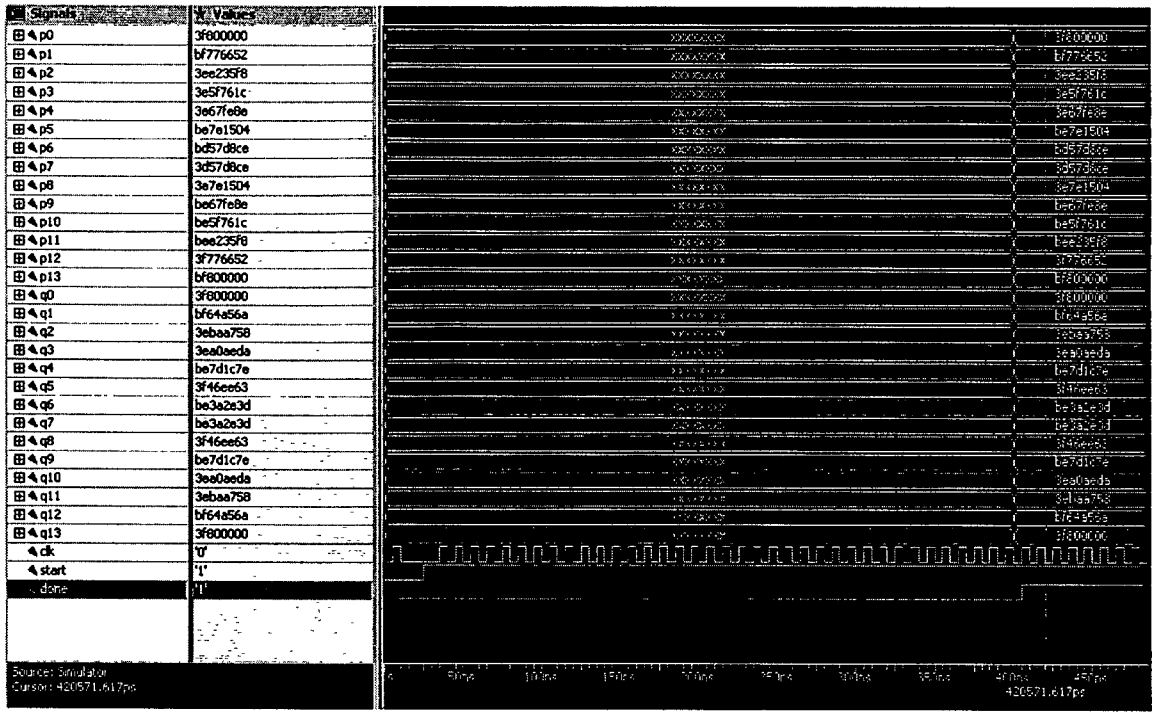

Figure 18: *atopq* Simulation Timing Diagram

4.2.2. SIMULATION RESULTS: *POLYDIV*

The *polydiv* entity is presented with the polynomial coefficients. Generally these are the outputs of *atopq*, however below the entity is presented with theoretical results from a reference software implementation to eliminate precision problems resulting from the previous stage from affecting the results from the *polydiv* test bench. The result is obtained after 13 clock cycles. Test results are presented in Table 2.

Table 2: *polydiv* Simulation Results

Input	Expected $P_{def}(z)$ Coefficients	Computed $P_{def}(z)$ Coefficients
1	1	1
-0.9664054	0.0335946	0.0335901
0.4418179	0.4754125	0.475408
0.2182245	0.693637	0.693633
0.2265575	0.9201945	0.920191
-0.248127	0.6720675	0.672061
-0.0526975	0.61937	0.619361
0.0526975	0.6720675	0.672059
0.248127	0.9201945	0.920186
-0.2265575	0.693637	0.693626
-0.2182245	0.4754125	0.475406
-0.4418179	0.0335946	0.0335861
0.9664054	1	0.999991
-1		

The outputs of *polydiv* as exercised by the test bench match expected results. The output is presented after 13 clock cycles. Figure 19 is the simulation timing diagram for the entity.

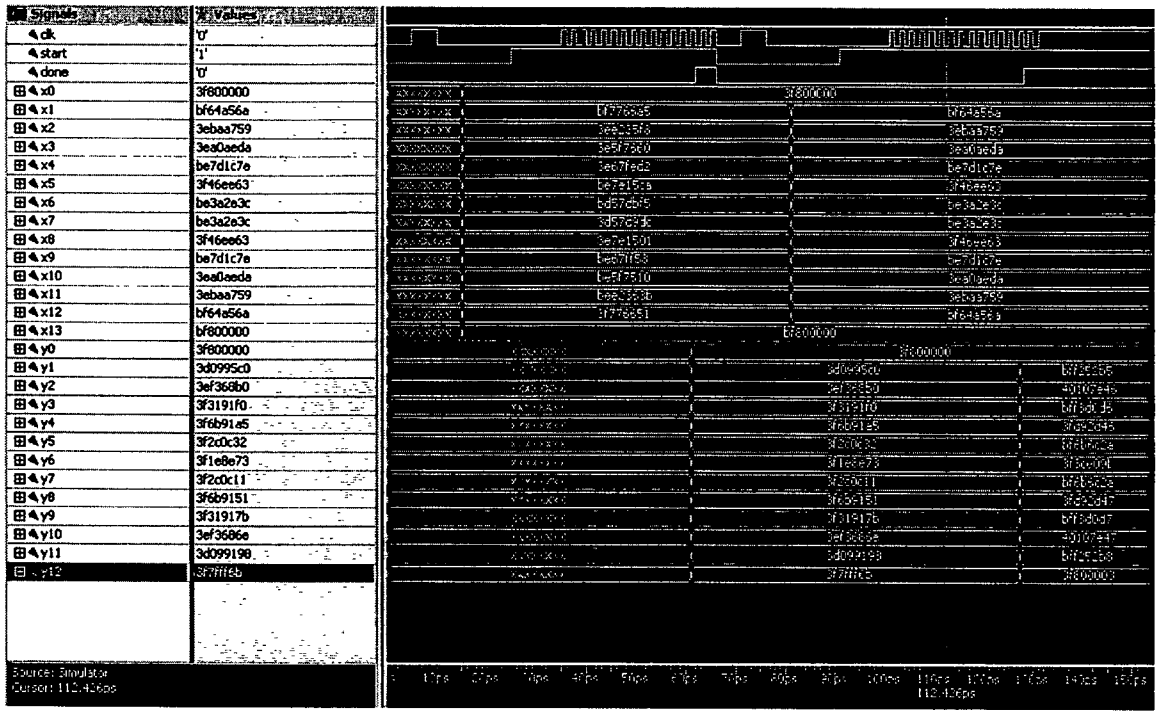

Figure 19: *polydiv* Simulation Timing Diagram

4.2.3. SIMULATION RESULTS: *CHEBFORM*

The *chebform* entity converts the *P(z)* and *Q(z)* polynomials to their corresponding Chebyshev form. The entity is presented with the deflated polynomial coefficients. The outputs of *chebform* are the coefficients of the corresponding Chebyshev polynomials. The output is resolved in 13 clock cycles. Table 3 presents the results obtained from the *chebform* entity.

Table 3: *chebform* Simulation Results

Input	Expected Coefficients	Computed Coefficients
1	0.61937	0.61937
0.0335946	1.344135	1.34414
0.4754125	1.840389	1.84039
0.693637	1.387274	1.38727
0.9201945	0.950825	0.950826
0.6720675	0.0671892	0.06719
0.61937	2	2
0.6720675		
0.9201945		
0.693637		
0.4754125		
0.0335946		
1		

The outputs of *chebform* as exercised by the test bench matched expected results. The timing diagram for the *chebform* simulation is shown in Figure 20.

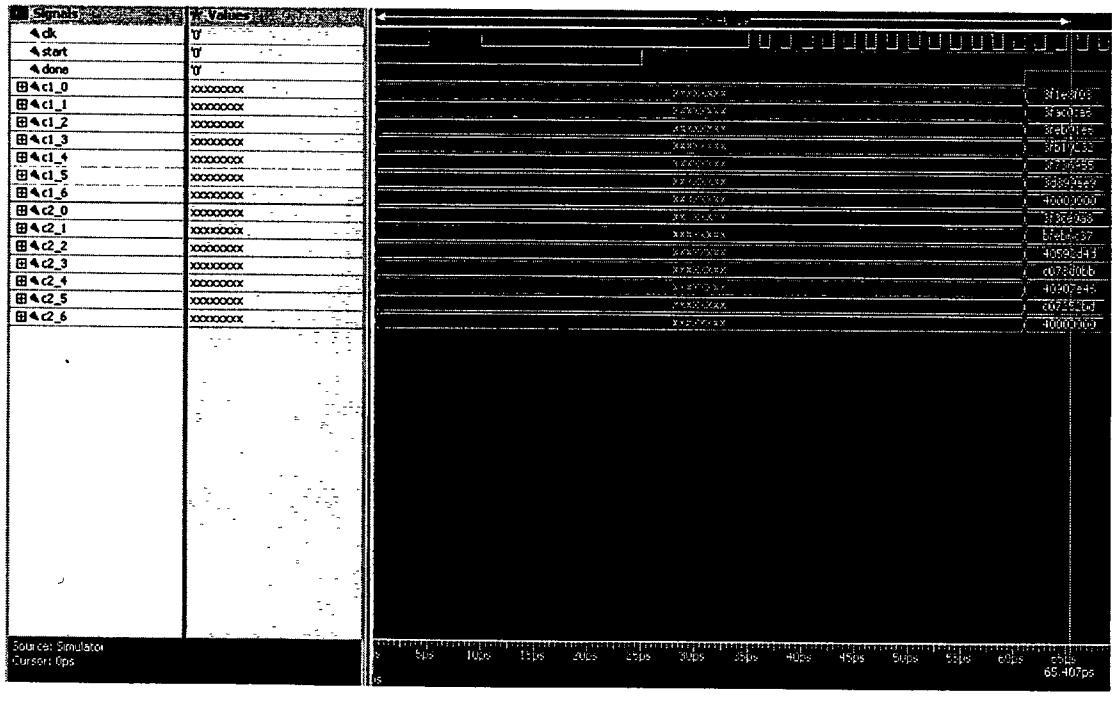

Figure 20: *chebform* Simulation Timing Diagram

4.2.4. SIMULATION RESULTS: *CLENSHAW*

The *clenshaw* entity is used to evaluate the Chebyshev polynomials at specific points on the x-axis using the Clenshaw Recurrence Formula. The entity is provided with the Chebyshev coefficients and a value for x. Upon completion of the computation, the result is presented on the y output. Table 4 shows the *clenshaw* simulation results for an x value of 0.98.

Table 4: *clenshaw* Simulation Results

Input Chebyshev Coefficients	**Expected Output**	**Computed Output**
0.7378032	0.675631	0.675626
-1.8392394		
3.3933894		
-3.8877464		
4.5154134		
-3.7862972		
2		

The outputs of *clenshaw* as exercised by the test bench matched expected results. Figure 21 shows the simulation results for *clenshaw*.

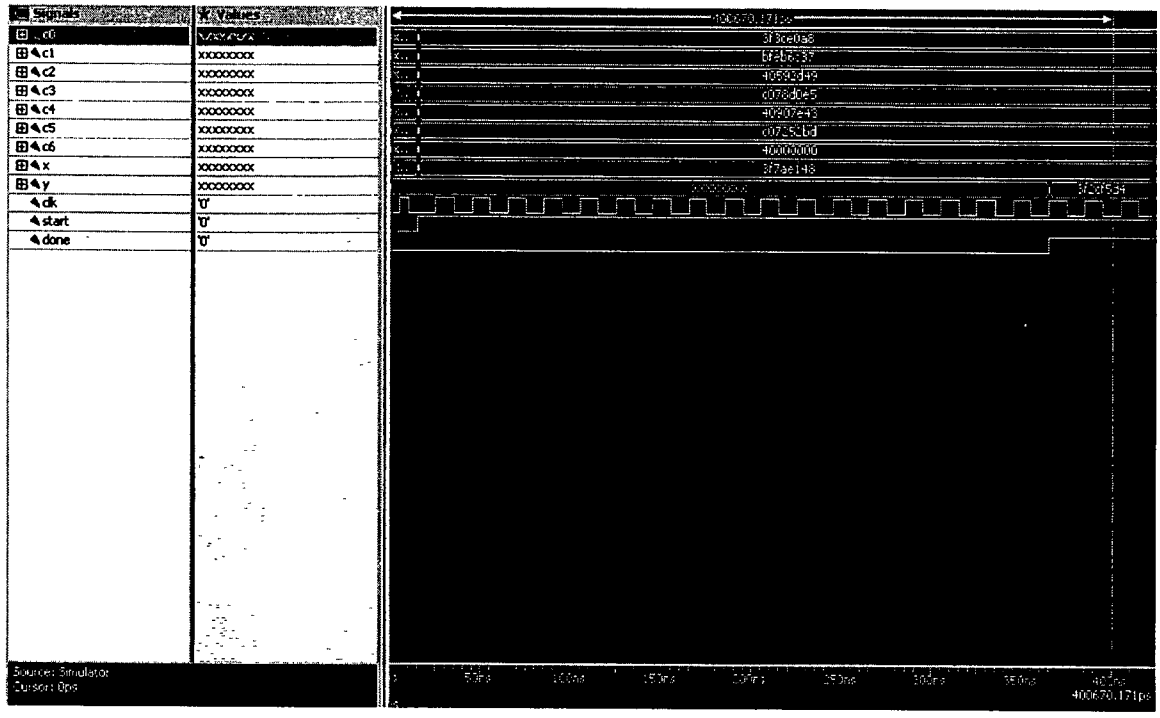

Figure 21: *clenshaw* Simulation Timing Diagram

4.2.5. SIMULATION RESULTS: *ROOTFINDER*

The primary entity in the execution of the algorithm is the *rootfinder* entity. This entity is provided with the coefficients of the Chebyshev polynomials and performs the scan of the region [-1,+1] in search of the zero crossings. The *rootfinder* entity is presented a known set of Chebyshev coefficients and the resulting zero crossing locations are computed and presented at the outputs. Table 5 shows the simulation results for *rootfinder*.

Table 5: *rootfinder* Simulation Results

Input A(z) Coefficients	Expected Outputs	Computed Outputs
0.929777	0.92621144666714	0.926211
-0.403188	0.85810873075617	0.858109
-0.266029	0.81315778984863	0.813158
0.0103105	0.64938386851508	0.649384
-0.264474	0.52544554098200	0.525446
0.117257	0.37905174475715	0.379052
0.0645595	-0.015244580367536	-0.01524
-0.512601	-0.25330834458677	-0.25331
0.236868	-0.40368236893924	-0.40368
-0.0478045	-0.69202503202314	-0.69203
0.0386299	-0.89931369114020	-0.89931
-0.0366284	-0.95800704025428	-0.95801

The outputs of *rootfinder* as exercised by the test bench match expected results.

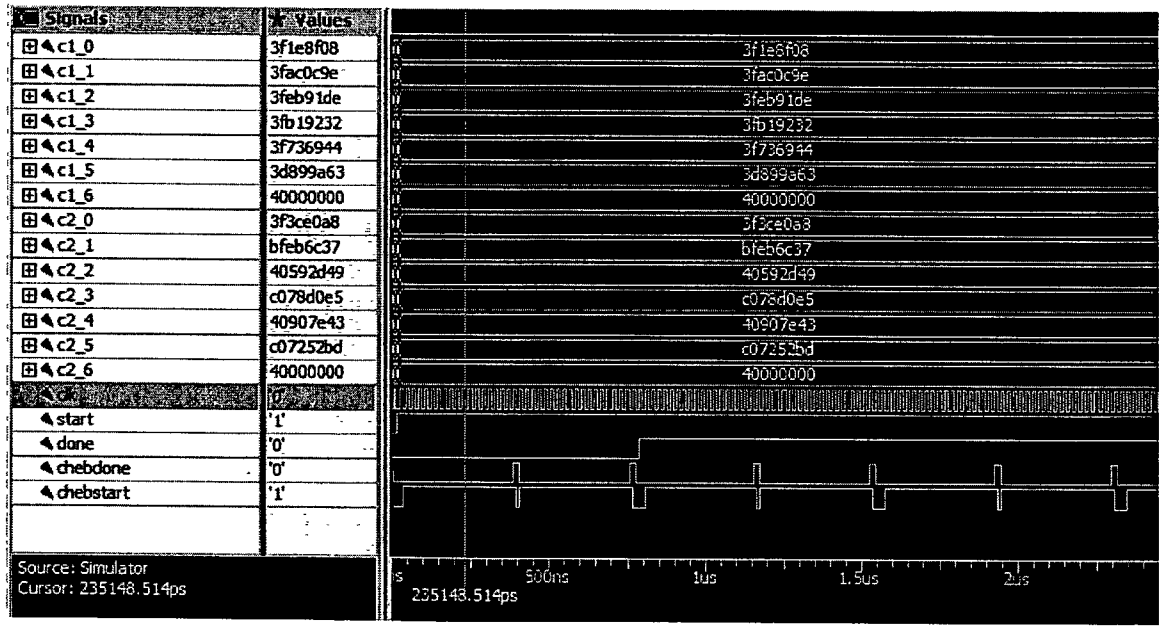

Figure 22: *rootfinder* Simulation Timing Diagram

4.3. SIMULATION RESULTS OF 10^{TH} ORDER CASE

All results presented previously were based on a 12^{th} order system. To verify that the design adequately processes data for the 10^{th} order case, the identical test benches were used with 10^{th} order data as the input vector. The following sections present the results of the 10^{th} order case for each entity.

4.3.1. SIMULATION RESULTS: *ATOPQ* 10^{th} ORDER CASE

Table 6 shows the outputs of *atopq* versus expected results. Results are shown in floating point format to highlight numerical differences due to precision issues.

Table 6: *atopq* Simulation Results (10^{th} Order Case)

Input	Expected P(z) Coefficients	Computed P(z) Coefficients
0	0	0
0.850473	1	1
-0.330070	-0.809264	-0.809264
0.524596	0.545613	0.545613
-0.447751	-0.723928	-0.723928
0.389482	0.79225	0.79225
-0.434903	-0.824385	-0.824385
0.344499	0.824385	0.824385
-0.199332	-0.79225	-0.79225
0.215543	0.723928	0.723928
0.041209	-0.545613	-0.545613
0.0	0.809264	0.809264
	-1	-1
	0	0

The outputs of *atopq* as exercised by the test bench matched expected results.

4.3.2. SIMULATION RESULTS: *POLYDIV* 10th ORDER CASE

Test results are presented in Table 7.

Table 7: *polydiv* Simulation Results (10th Order Case)

Input	Expected Coefficients	Computed Coefficients
0	0	0
1	1	1
0.190736	0.190736	0.190736
0.736349	0.736349	0.736349
0.012421	0.012421	0.0124211
0.804671	0.804671	0.804671
-0.019714	-0.019714	-0.0197139
0.804671	0.804671	0.804671
0.012421	0.012421	0.0124211
0.736349	0.736349	0.736349
0.19073	0.190736	0.190736
1	1	1
0	0	0

The outputs of *polydiv* as exercised by the test bench match expected results.

4.3.3. SIMULATION RESULTS: *CHEBFORM* 10th ORDER CASE

Table 8 presents the results obtained from the *chebform* entity.

Table 8: *chebform* Simulation Results (10^{th} Order Case)

Input	Expected Coefficients	Computed Coefficients
0	-0.019714	-0.19714
1	1.609342	1.60934
0.190736	0.024842	0.024842
0.736349	1.472698	1.4727
0.012421	0.381472	0.381472
0.804671	2	2
-0.019714	0	0
0.804671		
0.012421		
0.736349		
0.190736		
1		
0		

The outputs of *chebform* as exercised by the test bench matched expected results.

4.3.4. SIMULATION RESULTS: *CLENSHAW* 10^{th} ORDER CASE

Table 9 shows the *clenshaw* simulation results.

Table 9: *clenshaw* Simulation Results (10^{th} Order Case)

Input Chebyshev Coefficients	**Expected Output (@ 0.98)**	**Computed Output (@ 0.98)**
-0.019714	-0.15604185569792	-0.156042
1.609342		
0.024842		
1.472698		
0.381472		
2		
0		

The outputs of *clenshaw* as exercised by the test bench matched expected results.

4.3.5. SIMULATION RESULTS: *ROOTFINDER* 10^{th} ORDER CASE

Table 10 shows the simulation results for *rootfinder*.

Table 10: *rootfinder* Simulation Results (10^{th} Order Case)

Input A(z) Coefficients	Expected Outputs	Computed Outputs
0.850473	0.95611004596066	0.956110
-0.330070	0.64783128270799	0.647831
0.524596	0.63035961400801	0.630360
-0.447751	0.030747646531559	0.0307476
0.389482	-0.48245715781960	-0.482457
-0.434903	-0.59600213515672	-0.596002
0.344499	-0.84344620450057	-0.843446
-0.199332		
0.215543		
0.041209		

The outputs of *rootfinder* as exercised by the test bench match expected results.

5. SYNTHESIS

5.1. SYNTHESIS PROCESS

Once a VHDL model of the entities had been successfully simulated and the results verified, the next step is synthesis. Leonardo-Spectrum was used to process the VHDL descriptions of each entity into actual gate-level implementations using the AMI C5 technology library. The output of the synthesis process is an EDF netlist and a VHDL description of the entity, which is based on the gate level implementation, generated by the synthesis tool.

5.1.1. SYNTHESIS RESULTS

The gate count for each entity has been determined during synthesis. Table 11 shows the results.

Table 11: Gate Count by Entity

Entity	Gate Count
acos	7410
atopq	7699
chebform	5204
polydiv	5517
clenshaw	3139
fpmult	6548
fpadd	2285
fpdiv	7419
rootfinder	6570
Total:	**51791**

In an actual system implementation, the *fpadd*, *fpmult*, and *fpdiv* entities would most likely be replaced with the floating-point arithmetic units available in the larger speech processing system.

5.1.2. POST SYNTHESIS SIMULATION

The VHDL models generated by the synthesis tool reflects the gate-level implementation determined during synthesis. This allows simulation of actual gate delays associated with the implementation technology. Simulation using these models was performed using the exact test bench used for the original VHDL model of each entity.

It was determined that each entity performed identically in terms of actual output resolved for the given test input vectors using the test benches designed for original simulation efforts. In addition, all entities performed correctly with clock periods as short as 10ns, which implies 100MHz operation is possible. It is important to note, however, that parasitic effects are not considered in the synthesized VHDL models. Table 12 shows a summary of performance for each entity.

Table 12: Performance by Entity

Entity	10ns Clock	Total Cycles
acos	Yes	102
atopq	Yes	38
chebform	Yes	13
polydiv	Yes	13
clenshaw	Yes	18
fpmult	Yes	1
fpadd	Yes	1
fpdiv	Yes	1
rootfinder	Yes	4658 (Worst Case)
Total:		**4842 (Worst Case)**

As mentioned, each entity met the 10ns clock period performance goal. The total clock cycles shown do not include the single clock performance of the floating-point entities because they are accounted for in the testing of each major entity. The *rootfinder* entity exhibits somewhat nondeterministic performance because the total number of clock cycles necessary for *rootfinder* to resolve an output will depend upon the exact root locations in relation to the point at which a zero crossing is detected. The worst case assumes that all bisections of the region near the root location must be evaluated. The *rootfinder* evaluates a Chebyshev polynomial 100 times for the coarse scan and a maximum of 156 times for the fine resolution scan.

Given this performance, the design can resolve an output in a worst case of 4842 clock cycles at a clock rate of 100MHz. This allows evaluation of a 12^{th} order system in approximately 50us, which is significantly shorter than a typical frame of digitized speech.

5.2. OVERALL SIMULATION RESULTS

Because each of the previous simulation steps exercises each entity with known good input data, the effects of precision issues introduced by the design are isolated to each individual entity. In actual use, the output of many of the design entities provide the inputs to the next, introducing the possibility of compounding numerical errors due to precision problems.

The entire design was exercised using an input vector and the resulting outputs produced. Table 13 shows the output of the entire design to a given input, compared with the output produced by a known-good software implementation of the algorithm.

Table 13: ASIC Results Versus Expected Output

Input A(z) Coefficients	Expected Outputs	Computed Outputs
0.929777	0.0615229	0.061523
-0.403188	0.0858198	0.085820
-0.266029	0.0988733	0.098873
0.0103105	0.137513	0.137513
-0.264474	0.161949	0.161949
0.117257	0.188125	0.188125
0.0645595	0.252426	0.252426
-0.512601	0.290759	0.290759
0.236868	0.316135	0.316135
-0.0478045	0.371641	0.371641
0.0386299	0.427967	0.427967
-0.0366284	0.453713	0.453713

These results are not yet correlated to real-world performance. Variations in computed LSFs could introduce distortion at the receiving side. Methods to compute this distortion given these results have not been determined.

Barring errors in the implementation of the algorithmic entities, the factors that affect the numerical performance of the design are:

1. The performance of the floating-point entities.
2. The number of terms in the *acos* entity for converting the zero crossing locations into LSFs.

In a real system, the floating point entities would be replaced with a globally available floating point resource. The number of terms in the *acos* entity can easily be adjusted as needed, however the more terms used the longer resolution of the result will take. Also, a size penalty must be paid.

6. LAYOUT

6.1. LAYOUT OF EACH ENTITY

Layout was performed using Mentor Graphic's IC Station. The EDF file generated during synthesis was used as an input to IC Station and defines the interconnects between the individual devices from the AMI-05 device library. Each entity was placed and routed by the automated place and route tools provided by IC Station.

Following layout, the area requirements for each entity were determined. These are summarized in Table 14.

Table 14: Size and Area by Entity

Entity	Size (λ)	Area (mm^2)
Acos	11408 x 7999	11.178
Atopq	12303 x 8954	13.495
Chebform	7268 x 6706	5.971
Polydiv	8322 x 6815	6.948 (x2 = 13.896)
Clenshaw	5461 x 5252	3.513
Fpmult	7106 x 7046	6.133
Fpadd	5086 x 4303	2.681
Fpdiv	8464 x 7985	8.279
Rootfinder	8246 x 7985	7.716
Total:		**79.81**

Again, because the floating-point entities would most likely not be used when integrating with a larger system, the total area requirement for the design is approximately $61 mm^2$.

6.1.1. LAYOUT OF *ACOS* ENTITY

The following figure is the *acos* entity following layout and routing.

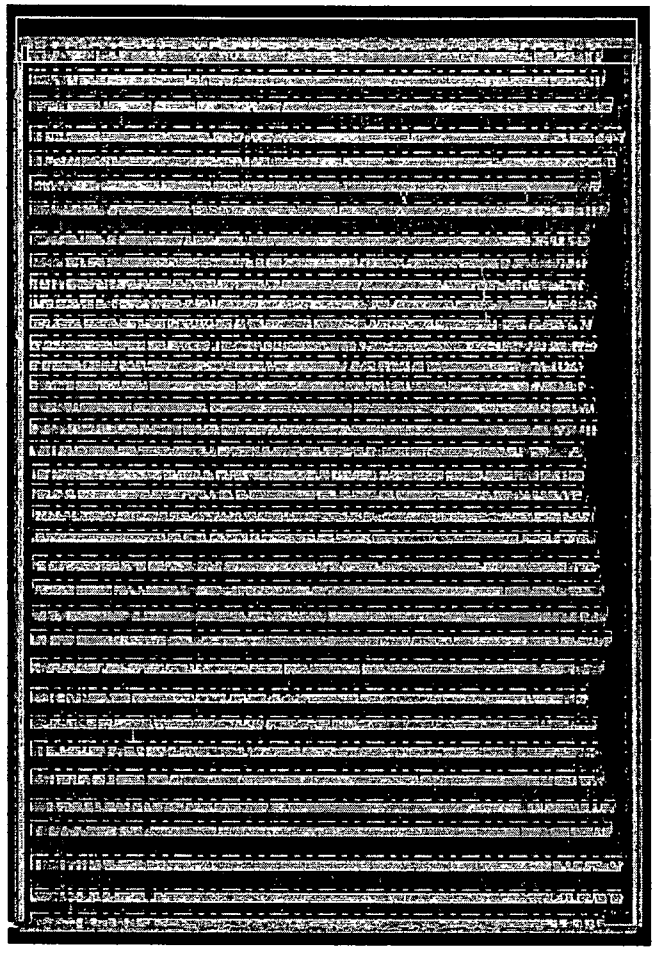

Figure 23: *acos* Entity Layout

6.1.2. LAYOUT OF *ATOPQ* ENTITY

The following figure is the *atopq* entity following layout and routing.

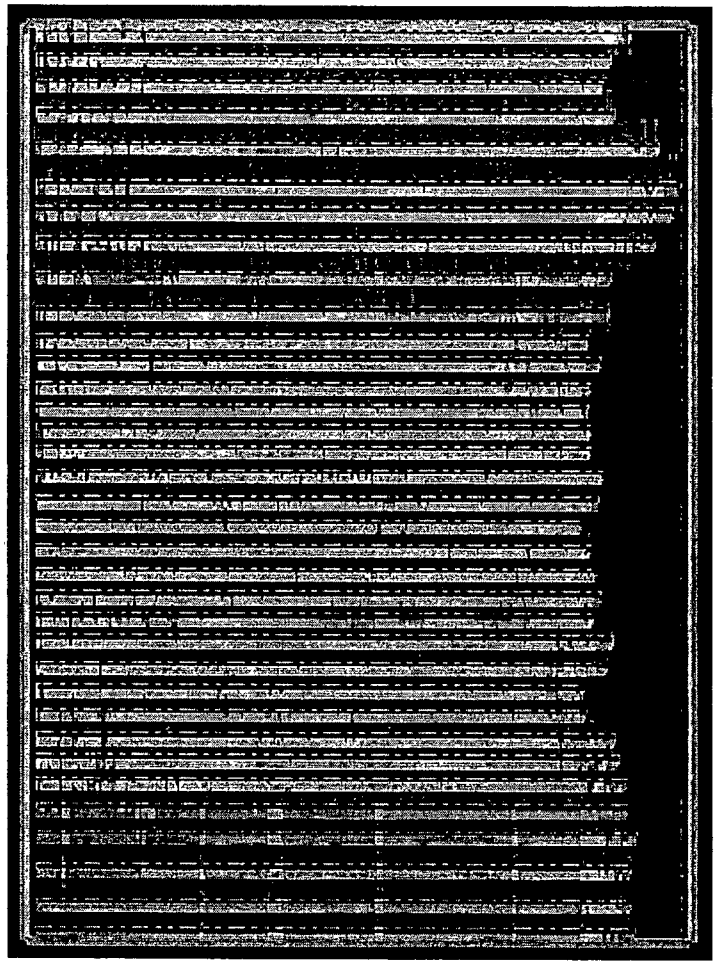

Figure 24: *atopq* Entity Layout

6.1.3. LAYOUT OF *CHEBFORM* ENTITY

The following figure is the *atopq* entity following layout and routing.

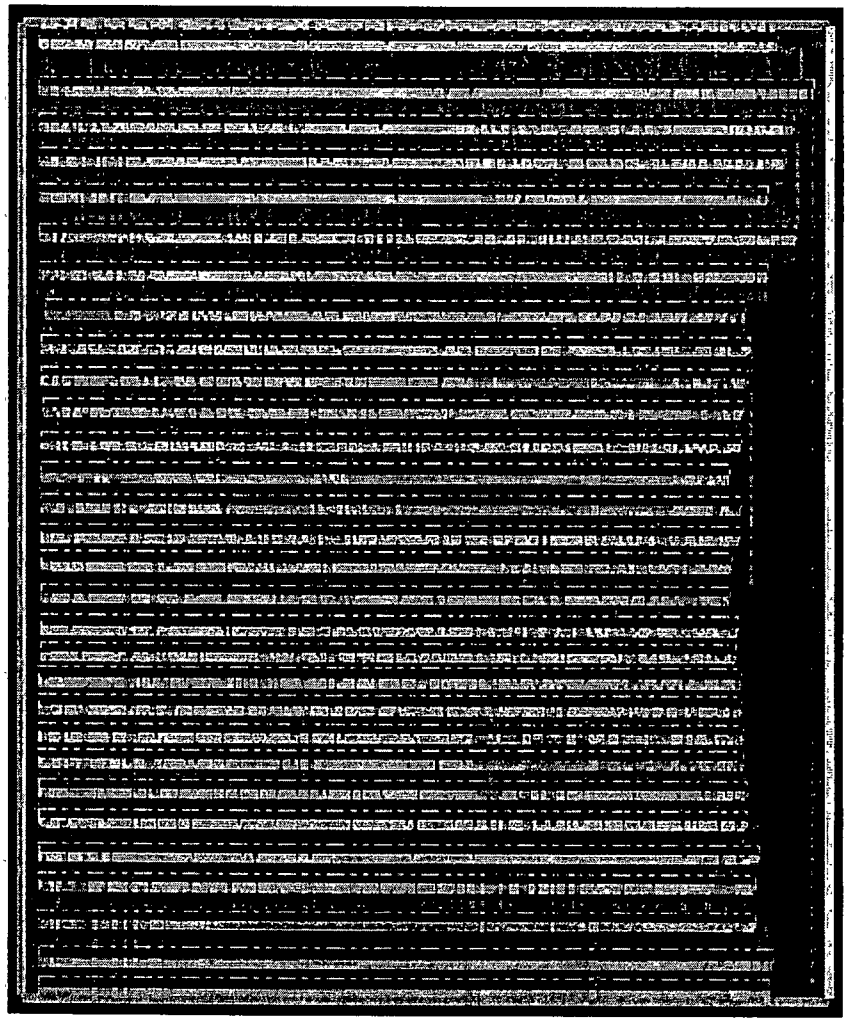

Figure 25: *chebform* Entity Layout

6.1.4. LAYOUT OF *POLYDIV* ENTITY

The following figure is the *atopq* entity following layout and routing.

Figure 26: *polydiv* Entity Layout

6.1.5. LAYOUT OF *CLENSHAW* ENTITY

The following figure is the *atopq* entity following layout and routing.

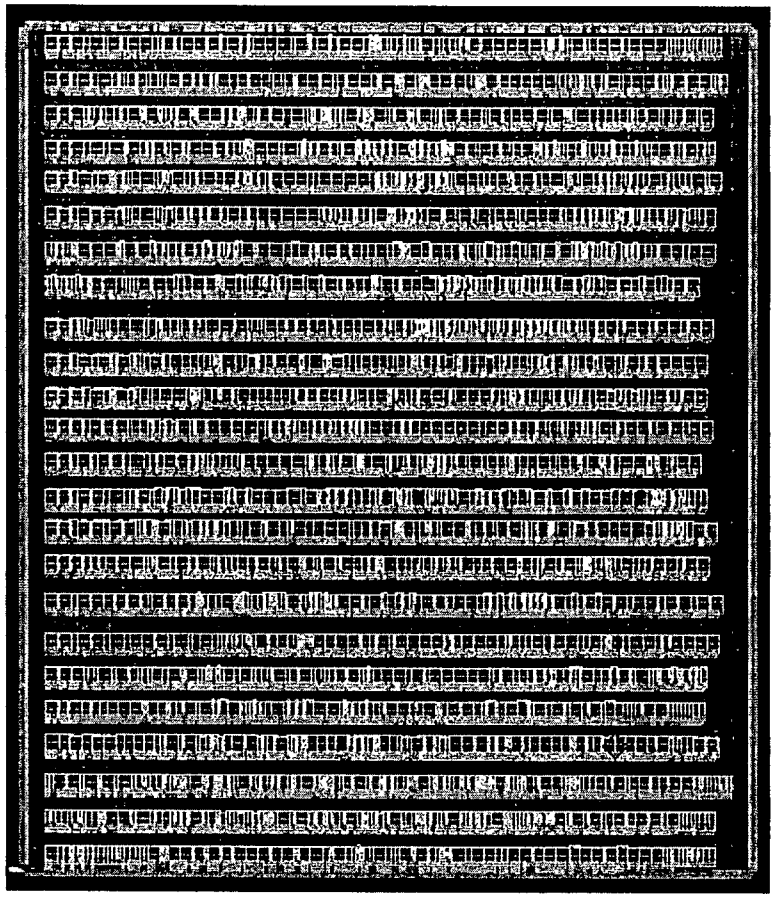

Figure 27: *clenshaw* Entity Layout

6.1.6. LAYOUT OF *FPMULT* ENTITY

The following figure is the *fpmult* entity following layout and routing.

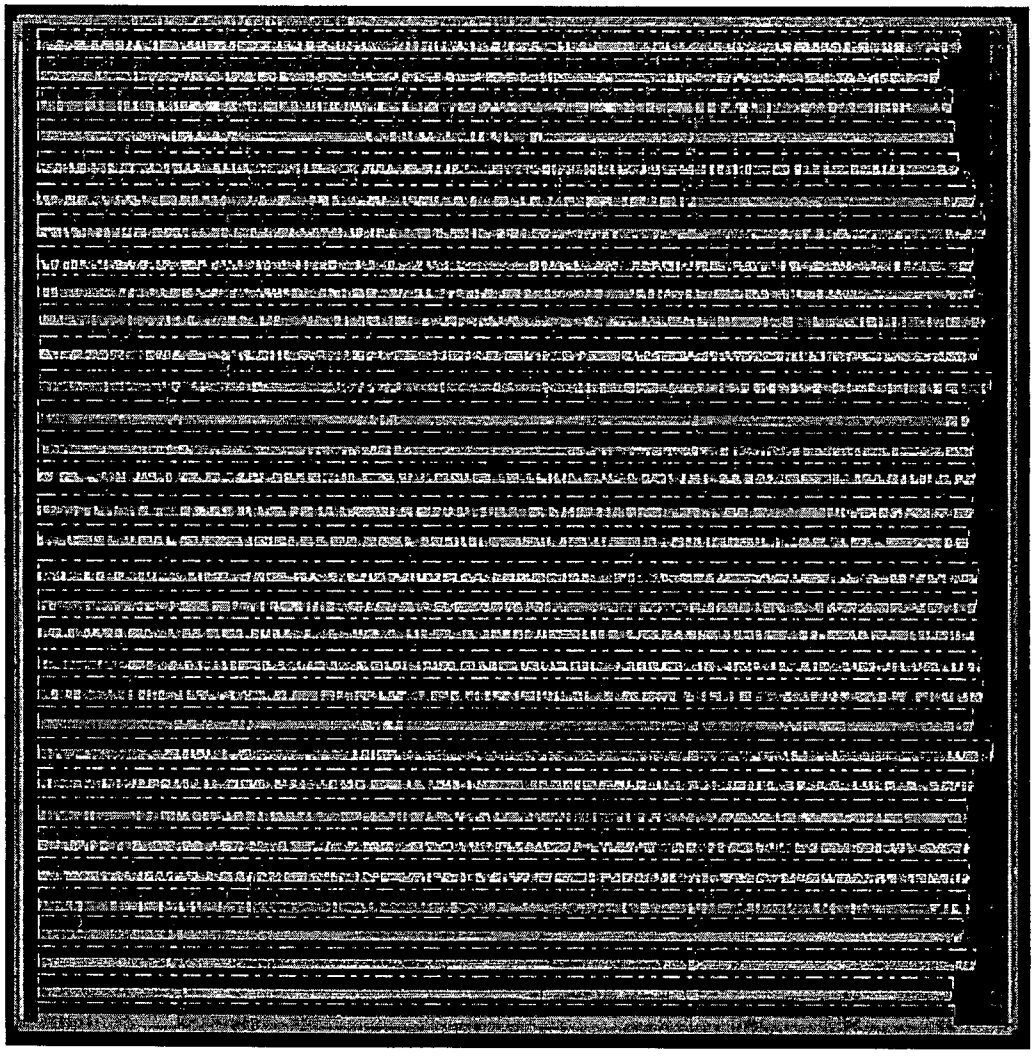

Figure 28: *fpmult* Entity Layout

6.1.7. LAYOUT OF *FPADD* ENTITY

The following figure is the *fpadd* entity following layout and routing.

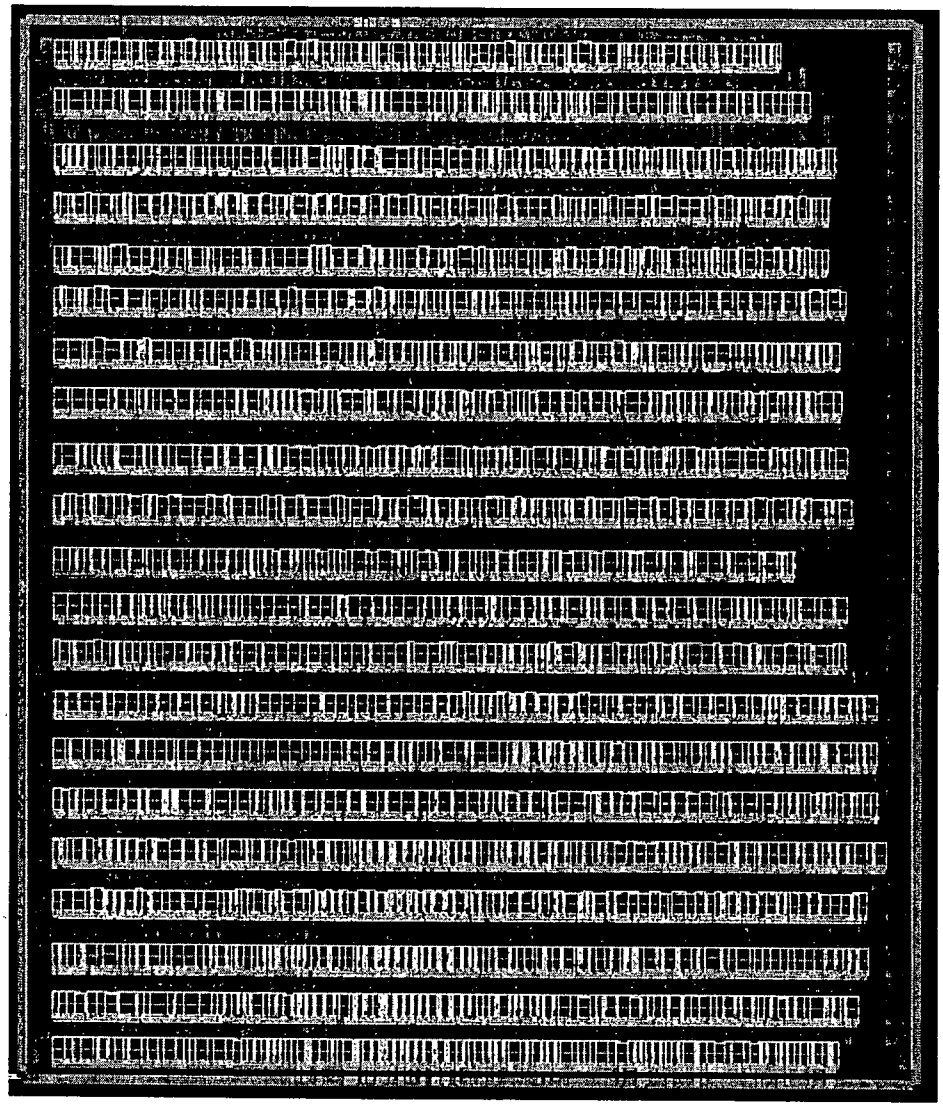

Figure 29: *fpadd* Entity Layout

6.1.8. LAYOUT OF *FPDIV* ENTITY

The following figure is the *fpdiv* entity following layout and routing.

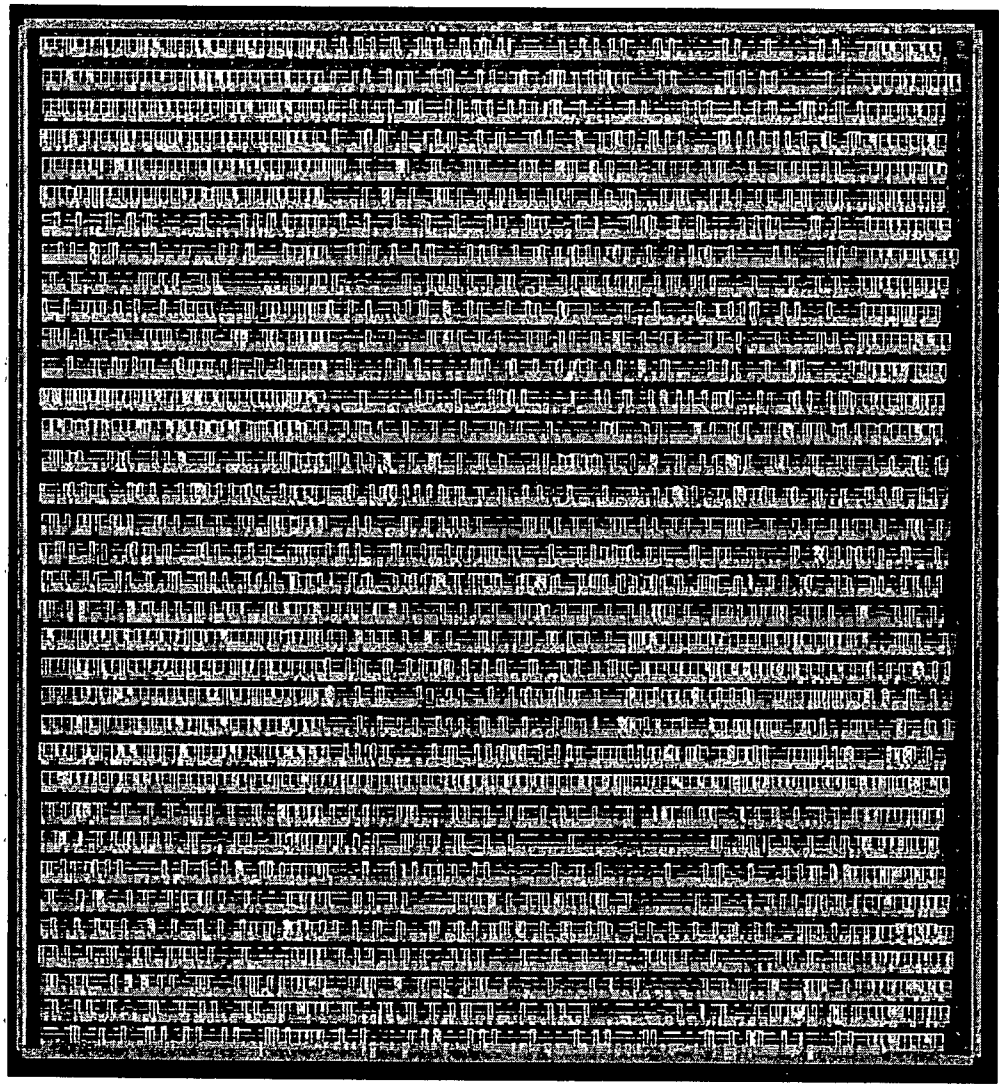

Figure 30: *fpdiv* Entity Layout

6.1.9. LAYOUT OF *ROOTFINDER* ENTITY

The following figure is the *rootfinder* entity following layout and routing.

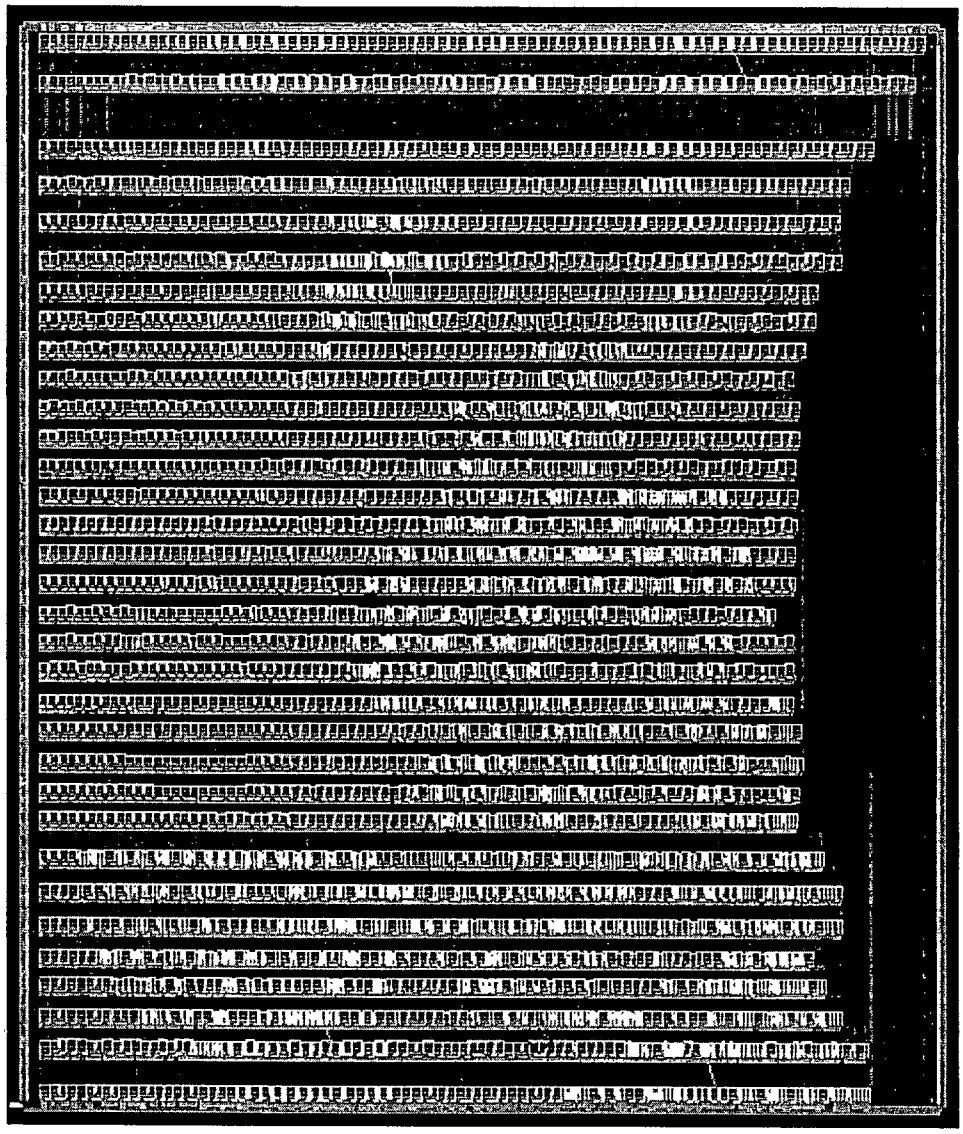

Figure 31: *rootfinder* Entity Layout

6.2. OVERALL ASIC FLOORPLAN

With all entities routed, floorplanning of the complete subsystem has been done. Placement of each entity has been chosen to reduce interconnections to the extent possible. Floating point elements have not been placed, however should this design be

fabricated for testing purposes, both a bus interface and the floating point entities would be required. Figure 32 shows the ASIC floorplan.

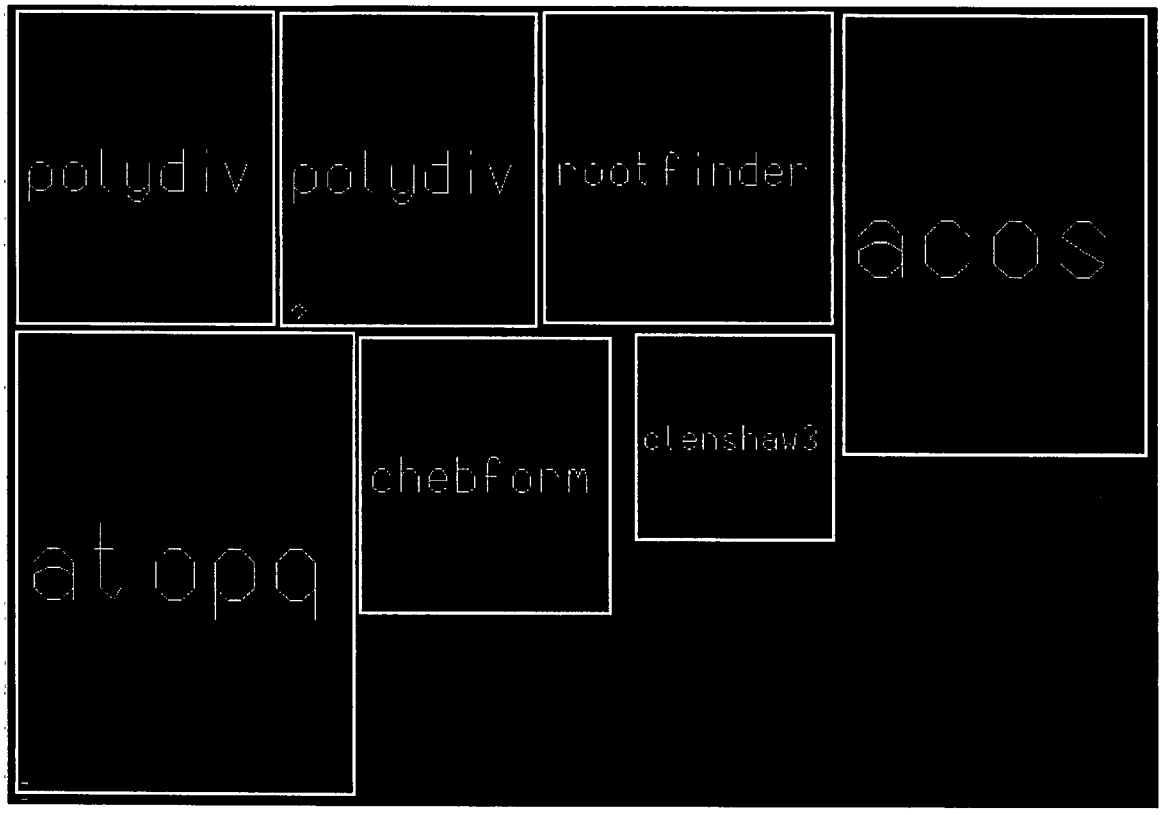

Figure 32: Overall ASIC Layout

7. SUMMARY AND DISCUSSION

7.1. SUMMARY

A VLSI design implementing an efficient method for the computation of LSFs has been developed and presented. The design implements the method proposed by Ramachandran and Kabal[1] for the efficient computation of Line Spectral Frequencies (LSFs) using Chebyshev polynomials. This method was first decomposed into the following steps:

1. Compute the coefficients of the symmetric and antisymmetric polynomials from the *A(z)* coefficients.
2. Deflate the resulting polynomials by their trivial roots.
3. Put the deflated polynomials into their corresponding Chebyshev form.
4. Evaluate the Chebyshev polynomials over the interval [-1,+1] and identify the zero crossing locations.
5. Compute the arccosine function of the zero crossing locations to obtain the LSFs.

For each major algorithmic step, a VHDL entity was designed to perform the required computations. 32-bit floating-point numerics are used throughout the design.

Following the design, simulation of the VHDL was performed and performance verified. The output of the entities under test corresponded with the outputs generated by a known-good software implementation of the algorithm. For the case of 10^{th} order

systems, the coefficients of the symmetric and antisymmetric polynomials are shifted in response to setting the outermost *A(z)* coefficients to zero when loading 10^{th} order data into the system. Using this method, the 10^{th} order polynomials may be processed as a 12^{th} order system with the unused coefficients set to zero. This makes the architecture uniformly 12^{th} order, but allows processing of 10^{th} order data.

Synthesis was performed for each entity using the AMI C5 process library and Mentor Graphics Leonardo-Spectrum. The resulting gate counts are listed in Table 11 for each entity. The synthesis tool produced a VHDL model of each entity using the actual gate-level models from the technology model used. These models were exercised with the test benches used to test the original entities. Actual gate delays are introduced in these models, however testing revealed that the design goal of 100MHz clocking could be achieved using the synthesis library. At that rate, the design is capable of resolving an output in 50us or less.

Layout has been performed using Mentor Graphic's IC Station. Each entity was placed and routed and size estimates determined in Table 14. The overall area consumed is approximately $66mm^2$. Note that this includes the floating point entities, however should this design be integrated into a larger speech coding system global floating point resources would be available and need not be duplicated.

7.2. CONCLUSIONS

The design presented here is capable of resolving the roots of the symmetric and antisymmetric polynomials resulting from linear predictive analysis and computing the corresponding LSFs in less than 50us with precision comparable to known-good software

implementations. Numerical differences observed are believed to be the result of the floating point implementation used as well as the number of terms used in the Taylor series expansion for computation of the arccosine.

Because of the importance of 10^{th} and 12^{th} order systems in speech coding and speaker recognition systems, the design has been made such that 10^{th} and 12^{th} order data can be processed with the same architecture. Systems of other orders cannot be processed, however modification for any even ordered system is a straightforward design change.

7.3. FURTHER WORK

The performance of the design should be further investigated. Numerical differences between the software implementations and VHDL models should be correlated to real-world performance metrics such as distortion on the receiving side. To achieve this, modified test benches must be created capable of processing large amounts of floating point data encoded as ASCII strings in text files. The specific performance metrics must also be selected.

Following a more thorough confirmation of adequate performance, the design should be simulated using post-layout parasitic effects. Parasitic capacitance is likely to further reduce the maximum rate at which the design can operate. The exact limits of performance should be determined.

To actually test real-world performance, the design may be fabricated. To do so, a bus interface design must be implemented to allow loading of the twelve 32-bit input vectors as a series of memory loads 8- or 16-bits in width. In this way an external system

can exercise the design. The same data used in VHDL test bench simulation could be used for comparison and real throughput measured.

8. VHDL LISTINGS

8.1. ATPQ VHDL LISTING

```
library IEEE;
use IEEE.std_logic_1164.all;

entity atopq is
        port(

        -- The A,P and Q ports are for the A(z) coefficient inputs and the P(z) adn
        -- Q(z) coefficient outputs
        A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11          :in std_logic_vector(31 downto 0);
        P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13        :out std_logic_vector(31
                                                              downto 0);
        Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13        :out std_logic_vector(31
                                                              downto 0);
        -- The X and Y outputs are for floating point variables destined for the adder
        -- or multiplier.
        XADD,YADD                                          :out std_logic_vector(31 downto 0);
        ZADD                                               :in std_logic_vector(31 downto 0);
        XMULT,YMULT                                        :out std_logic_vector(31 downto 0);
        ZMULT                                              :in std_logic_vector(31 downto 0);
        -- The START bit is asserted when the A(z) coefficients have been presented and
        -- will start execution of the state machine.  The CLK input is the system clock.
        -- The DONE output will be asserted when the final data is placed on the P and Q
        -- output ports.
        START                                              :in std_logic;
        CLK                                                :in std_logic;
        DONE                                               :out std_logic);
end atopq;

architecture beh of atopq is
subtype float32 is std_logic_vector(31 downto 0); -- subtype to create arrays of floats
type floatvect12 is array (11 downto 0) of float32; -- an array of floats
type floatvect14 is array (13 downto 0) of float32; -- an array of floats
constant one_float            :std_logic_vector(31 downto 0) :=
"00111111100000000000000000000000";
constant neg_one_float :std_logic_vector(31 downto 0) :=
"10111111100000000000000000000000";

begin
mainproc:process(CLK)
variable A                    :floatvect12;
variable P                    :floatvect14;
variable Q                    :floatvect14;
variable INDEX          :integer range 0 to 11;
variable STATE          :integer range 0 to 99;       -- when finished, limit range to
actual requirement
variable TEMPF          :std_logic_vector(31 downto 0);
variable TENTHFLAG      :std_logic;
begin
if START = '0' then
        DONE <= '0';
        STATE := 0;
        XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

else
        -- Here is the state machine that executes if the START bit is asserted.
```

```
case STATE is

        when 0 =>
        -- The first thing we do is copy the inputs into the local variable for
        -- processing
        A(0) := A0;
        A(1) := A1;
        A(2) := A2;
        A(3) := A3;
        A(4) := A4;
        A(5) := A5;
        A(6) := A6;
        A(7) := A7;
        A(8) := A8;
        A(9) := A9;
        A(10) := A10;
        A(11) := A11;
        -- We know that P(0) and Q(0) are always 1, so we assign them here rather
        than going
        -- through the evaluation.  The same goes for P(11) and Q(11) which are -1
        and 1
        -- respectively.
        P(0) := one_float;
        Q(0) := one_float;
        P(13):= neg_one_float;
        Q(13):= one_float;
        -- Here we check if this is a tenth order system.  If so, the input vector
        has zeros
        -- at eather end.
        if ((A(0) = "00000000000000000000000000000000")) then
                TENTHFLAG := '1';
                DONE <= '1';
        else
                TENTHFLAG := '0';
        end if;
        STATE := 1;
        INDEX := 0;

        when 1 =>
        XADD <= A(index);
        TEMPF := A(11 - index);
        TEMPF(31) := not TEMPF(31);
        YADD <= TEMPF;
        STATE := 2;

        when 2 =>
        XMULT <= ZADD;
        YMULT <= neg_one_float;
        STATE := 3;

        when 3 =>
        P(index + 1) := ZMULT; -- added + 1
        if INDEX = 11 then -- changed from 10
                INDEX := 0;
                STATE := 4;
        else
                INDEX := INDEX + 1;
                STATE := 1;
        end if;

        when 4 =>
        XADD <= A(INDEX);
        YADD <= A(11 - INDEX);
        STATE := 5;

        when 5 =>
        XMULT <= ZADD;
        YMULT <= neg_one_float;
        STATE := 6;

        when 6 =>
```

```
                Q(index + 1) := ZMULT;
                if INDEX = 11 then
                        STATE := 7;
                        -- If this is a 10th order system, we shift the coefficients.
                        if TENTHFLAG = '1' then
                                P(1) := P(0);
                                P(0) := "00000000000000000000000000000000";
                                P(12) := P(13);
                                P(13) := "00000000000000000000000000000000";
                                Q(1) := Q(0);
                                Q(0) := "00000000000000000000000000000000";
                                Q(12) := Q(13);
                                Q(13) := "00000000000000000000000000000000";
                        end if;
                else
                        INDEX := index + 1;
                        STATE := 4;
                end if;

                when 7 =>
                P0 <= P(0);
                P1 <= P(1);
                P2 <= P(2);
                P3 <= P(3);
                P4 <= P(4);
                P5 <= P(5);
                P6 <= P(6);
                P7 <= P(7);
                P8 <= P(8);
                P9 <= P(9);
                P10 <= P(10);
                P11 <= P(11);
                P12 <= P(12);
                P13 <= P(13);
                Q0 <= Q(0);
                Q1 <= Q(1);
                Q2 <= Q(2);
                Q3 <= Q(3);
                Q4 <= Q(4);
                Q5 <= Q(5);
                Q6 <= Q(6);
                Q7 <= Q(7);
                Q8 <= Q(8);
                Q9 <= Q(9);
                Q10 <= Q(10);
                Q11 <= Q(11);
                Q12 <= Q(12);
                Q13 <= Q(13);
                XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                STATE := 8;

                when others =>
                --DONE <= '1';

        end case;
end if;

end process;

end beh;
```

8.2. *POLYDIV* VHDL LISTING

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity polydiv is
        port(

        -- X0 to X13 is the input to the polynomial divider
        X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13        :in std_logic_vector(31
                                                              downto 0);
        -- Y0 to Y12 is the output of the polynomial division
        Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9,Y10,Y11,Y12     :out std_logic_vector(31 downto 0);
        -- XADD,YADD and ZADD are interfaces to an external floating point adder
        XADD,YADD                                      :out std_logic_vector(31 downto 0);
        ZADD                                           :in std_logic_vector(31 downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT                                    :out std_logic_vector(31 downto 0);
        ZMULT                                          :in std_logic_vector(31 downto 0);
        -- PQSEL selects if we're working with a P or Q polynomial and removes the
        appropriate root
        PQSEL                                          :in std_logic;

        CLK                                            :in std_logic;
        START                                          :in std_logic;
        DONE                                           :out std_logic);
end polydiv;

architecture beh of polydiv is
subtype float32 is std_logic_vector(31 downto 0);
type floatvect is array(13 downto 0) of float32;
constant one_float     :std_logic_vector(31 downto 0) :=
"00111111100000000000000000000000";
constant neg_one_float :std_logic_vector(31 downto 0) :=
"10111111100000000000000000000000";

begin
        mainproc:process(CLK)
        variable X             :floatvect;
        variable Y             :floatvect;
        variable ROOT  :std_logic_vector(31 downto 0);
        variable TEMPF :std_logic_vector(31 downto 0);
        variable INDEX :integer range 0 to 13;
        variable STATE :integer range 0 to 99;
        begin
        if START = '0' then
                DONE <= '0';
                STATE := 0;
                XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

        else

                -- Here's the case statement that implements the state machine
                -- of the polynomial divider.
                case STATE is

                        when 0 =>
                        X(0) := X0;
                        X(1) := X1;
                        X(2) := X2;
                        X(3) := X3;
                        X(4) := X4;
                        X(5) := X5;
                        X(6) := X6;
                        X(7) := X7;
                        X(8) := X8;
                        X(9) := X9;
                        X(10) := X10;
                        X(11) := X11;
                        X(12) := X12;
                        X(13) := X13;
                        if PQSEL = '1' then
```

```
                ROOT := one_float;
            else
                ROOT := neg_one_float;
            end if;
            STATE := 1;

            when 1 =>
            Y(0) := X(0);
            XMULT <= ROOT;
            YMULT <= Y(0);
            INDEX := 1;
            STATE := STATE + 1;

            -- Now we start iterating through the remaining coefficients
            when 2 =>

            -- Offload the multiplier and load the adder with the result and
            the next
            -- X coefficient.
            XADD <= ZMULT;
            YADD <= X(INDEX);
            STATE := STATE + 1;

            when 3 =>
            -- First we offload the adder output, which is our next coefficient
            in the
            -- result.
            Y(INDEX) := ZADD;
            INDEX := INDEX + 1;
            -- Next, the multiplier is loaded with the next coefficient and the
            root.
            XMULT <= ROOT;
            YMULT <= Y(INDEX-1);
            if INDEX = 13 then
                STATE := STATE + 1;
            else
                STATE := 2;
            end if;

            when 4 =>
            -- Here we load the output signals and set the done flag.
            Y0 <= Y(0);
            Y1 <= Y(1);
            Y2 <= Y(2);
            Y3 <= Y(3);
            Y4 <= Y(4);
            Y5 <= Y(5);
            Y6 <= Y(6);
            Y7 <= Y(7);
            Y8 <= Y(8);
            Y9 <= Y(9);
            Y10 <= Y(10);
            Y11 <= Y(11);
            Y12 <= Y(12);
            XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
            YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
            XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
            YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        STATE := STATE + 1;

            when others =>
            DONE <= '1';

        end case;

end if;
end process;
end beh;
```

8.3. *CHEBFORM* VHDL LISTING

```
library IEEE;
use IEEE.std_logic_1164.all;

entity chebform is
        port (
        -- P0-P10 and Q0-Q10 are the P(z) and Q(z) polynomial coefficients
        P0,P1,P2,P3,P4,P5,P6                            :in std_logic_vector(31 downto 0);
        Q0,Q1,Q2,Q3,Q4,Q5,Q6                            :in std_logic_vector(31 downto 0);
        -- C1_0-C1_6 and C2_0-C2_6 are the resulting Chebyshev coefficients
        C1_0,C1_1,C1_2,C1_3,C1_4,C1_5,C1_6              :out std_logic_vector(31 downto 0);
        C2_0,C2_1,C2_2,C2_3,C2_4,C2_5,C2_6              :out std_logic_vector(31 downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT                                     :out std_logic_vector(31 downto 0);
        ZMULT                                           :in std_logic_vector(31 downto 0);
        CLK                                             :in std_logic;
        START                                           :in std_logic;
        DONE                                            :out std_logic);
end chebform;

architecture beh of chebform is
subtype float32 is std_logic_vector(31 downto 0);
type floatvect7 is array (6 downto 0) of float32;
constant two_float :std_logic_vector(31 downto 0) := "01000000000000000000000000000000";

begin
mainproc:process(CLK)
variable P              :floatvect7;
variable Q              :floatvect7;
variable C1             :floatvect7;
variable C2             :floatvect7;
variable INDEX :integer range 0 to 10;
variable STATE :integer range 0 to 99;        -- note - fix this range when you know more
begin
        if START = '0' then
                DONE <= '0';
                STATE := 0;
                XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

        else
                -- Here is the state machine that executes when START is asserted
                case STATE is

                        when 0 =>
                        -- First thing, we copy all inputs into internal variables
                        P(0) := P0;
                        P(1) := P1;
                        P(2) := P2;
                        P(3) := P3;
                        P(4) := P4;
                        P(5) := P5;
                        P(6) := P6;
                        Q(0) := Q0;
                        Q(1) := Q1;
                        Q(2) := Q2;
                        Q(3) := Q3;
                        Q(4) := Q4;
                        Q(5) := Q5;
                        Q(6) := Q6;
                        STATE := STATE + 1;

                        when 1 =>
                        -- The first two are easy.
                        C1(6) := P(6);
                        C2(6) := Q(6);
                        INDEX := 0;
                        STATE := STATE + 1;
```

```
                -- Now we iterate through them, but we have to multiply by two for
                the remainder.
                when 2 =>
                XMULT <= two_float;
                YMULT <= P(INDEX);
                STATE := STATE + 1;

                when 3 =>
                C1(INDEX) := ZMULT;
                if(INDEX = 5) then
                    STATE := STATE + 1;
                    INDEX := 0;
                else
                    INDEX := INDEX + 1;
                    STATE := 2;
                end if;

            -- And we do the same for the C2 polynomial.
                when 4 =>
                XMULT <= two_float;
                YMULT <= Q(INDEX);
                STATE := STATE + 1;

                when 5 =>
                C2(INDEX) := ZMULT;
                if(INDEX = 5) then
                    STATE := STATE + 1;
                    INDEX := 0;
                else
                    INDEX := INDEX + 1;
                    STATE := 4;
                end if;

                when others =>
                -- Offload the results to the outputs.  The index order is
                -- reversed because that made the indexing simpler above.
                C1_0 <= C1(6);
                C1_1 <= C1(5);
                C1_2 <= C1(4);
                C1_3 <= C1(3);
                C1_4 <= C1(2);
                C1_5 <= C1(1);
                C1_6 <= C1(0);
                C2_0 <= C2(6);
                C2_1 <= C2(5);
                C2_2 <= C2(4);
                C2_3 <= C2(3);
                C2_4 <= C2(2);
                C2_5 <= C2(1);
                C2_6 <= C2(0);
                XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                DONE <= '1';

            end case;
        end if;
end process;
end beh;
```

8.4. *CLENSHAW* VHDL LISTING

```
library IEEE;
use IEEE.std_logic_1164.all;

entity clenshaw is
        port (

        -- C0-C6 are the Chebyshev polynomial coefficients.
        C0,C1,C2,C3,C4,C5,C6   :in std_logic_vector(31 downto 0);
```

```
        -- X is the input to the polynomial evaluation.
        X                       :in std_logic_vector(31 downto 0);
        -- Y is the output of the polynomial evaluation.
        Y                       :out std_logic_vector(31 downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT             :out std_logic_vector(31 downto 0);
        ZMULT                   :in std_logic_vector(31 downto 0);
        XADD,YADD               :out std_logic_vector(31 downto 0);
        ZADD                    :in std_logic_vector(31 downto 0);
        CLK                     :in std_logic;
        START                   :in std_logic;
        DONE                    :out std_logic);
end clenshaw;

architecture beh of clenshaw is
subtype float32 is std_logic_vector(31 downto 0);
type floatvect7 is array (6 downto 0) of float32;
constant zero_float :std_logic_vector(31 downto 0) := "00000000000000000000000000000000";
constant one_half_float :std_logic_vector(31 downto 0) :=
"00111111000000000000000000000000";
constant two_float:std_logic_vector(31 downto 0) := "01000000000000000000000000000000";

begin
mainproc:process(CLK)
variable C                      :floatvect7;
variable INDEX                  :integer range 0 to 10;
variable COUNT                  :integer range 0 to 6;
variable STATE                  :integer range 0 to 99;       -- note - fix this range when
you know more
variable B0,B1,B2               :std_logic_vector(31 downto 0);
variable TEMP1,TEMP2,TEMP3      :std_logic_vector(31 downto 0);
variable TWO_X                  :std_logic_vector(31 downto 0);
variable X_VAR                  :std_logic_vector(31 downto 0);
begin
        if START = '0' then
                DONE <= '0';
                STATE := 0;
                XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

        else
                -- Here is the state machine that executes when START is asserted
                case STATE is

                        when 0 =>
                        -- First thing, we copy all inputs into internal variables
                        C(0) := C0;
                        C(1) := C1;
                        C(2) := C2;
                        C(3) := C3;
                        C(4) := C4;
                        C(5) := C5;
                        C(6) := C6;
                        X_VAR := X;
                        COUNT := 6;
                        INDEX := 0;
                        STATE := STATE + 1;

                        when 1 =>
                        -- We will need 2X repeatedly, so we calculate it once here.
                        XMULT <= two_float;
                        YMULT <= X_VAR;
                        -- We also initialize our intermediate variables.
                        B0 := zero_float;
                        B1 := zero_float;
                        STATE := STATE + 1;

                        when 2 =>
                        TWO_X := ZMULT; -- and store it for future use
```

```
STATE := STATE + 1;

-- Here we begin to loop through to evaluate the recurrence.
when 3 =>
B2 := B1;
B1 := B0;
-- B0 = (2*X*B1)-B2+C(COUNT) so we load the multipler first to get
2X*B1
XMULT <= TWO_X;
YMULT <= B1;
STATE := STATE + 1;

when 4 =>
TEMP1 := ZMULT; -- off load the result of 2X*B1
-- next we subtract B2
TEMP2 := B2;
-- we change it's sign to subtract
TEMP2(31) := not B2(31);
XADD <= TEMP1;
YADD <= TEMP2;
STATE := STATE + 1;

when 5 =>
TEMP3 := ZADD; -- offload the previous result
-- Now we must add C(COUNT)
XADD <= TEMP3;
YADD <= C(COUNT);
STATE := STATE + 1;

when 6 =>
B0 := ZADD; -- offload the previous result
-- Now check if our loop is done.
if COUNT = 0 then
       STATE := STATE + 1;
else
       STATE := 3;
       COUNT := COUNT - 1;
end if;

when 7 =>
XADD <= B0;
TEMP1 := B2;
TEMP1(31) := not B2(31);
YADD <= TEMP1;
STATE := STATE + 1;

when 8 =>
TEMP3 := ZADD;
XADD <= TEMP3;
YADD <= C(0);
STATE := STATE + 1;

when 9 =>
TEMP1 := ZADD;
XMULT <= TEMP1;
YMULT <= one_half_float;
STATE := STATE + 1;

when 10 =>
TEMP3 := ZMULT;
Y <= TEMP3;
DONE <= '1';
XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
STATE := 0;

when others =>
STATE := 0;
```

```
            end case;
        end if;
end process;
end beh;
```

8.5. *ROOTFINDER* VHDL LISTING

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rootfinder is
        port (
        -- C1 and C2 are the Chebyshev coefficients
        C1_0,C1_1,C1_2,C1_3,C1_4,C1_5,C1_6    :in std_logic_vector(31 downto 0);
        C2_0,C2_1,C2_2,C2_3,C2_4,C2_5,C2_6    :in std_logic_vector(31 downto 0);
        -- X0-X10 are the zero crossing locations in the Chebyshev domain
        X0,X1,X2,X3,X4,X5,X6,X7,X8,X9         :out std_logic_vector(31 downto 0);
        -- CL0-CL5 and CX are interfaces to the Clenshaw evaluator.
        CL0,CL1,CL2,CL3,CL4,CL5,CL6,CX                :out std_logic_vector(31 downto 0);
        CY                                    :in std_logic_vector(31 downto 0);
        -- the following are interfaces to an external floating point units
        XMULT,YMULT                           :out std_logic_vector(31 downto 0);
        ZMULT                                 :in std_logic_vector(31 downto 0);
        XADD,YADD                             :out std_logic_vector(31 downto 0);
        ZADD                                  :in std_logic_vector(31 downto 0);
        XDIV,YDIV                             :out std_logic_vector(31 downto 0);
        ZDIV                                  :in std_logic_vector(31 downto 0);
        CLK                                   :in std_logic;
        START                                 :in std_logic;
        CHEBDONE                               :in std_logic;
        CHEBSTART                              :out std_logic;
        DONE                                  :out std_logic;
        DEBUG_FLOAT                           :out std_logic_vector(31 downto 0);
        DEBUG_BIT                             :out std_logic);
end rootfinder;

architecture beh of rootfinder is
subtype float32 is std_logic_vector(31 downto 0);
type floatvect11 is array (10 downto 0) of float32;
type floatvect7 is array (6 downto 0) of float32;
constant neg_one_float  :std_logic_vector(31 downto 0) :=
"10111111100000000000000000000000";
constant one_float      :std_logic_vector(31 downto 0) :=
"00111111100000000000000000000000";
-- DEBUG - these increments should be negative if we're adding them
constant coarse_inc     :std_logic_vector(31 downto 0) :=
"10111100101000111101011100001010";
constant fine_inc       :std_logic_vector(31 downto 0) :=
"10111010100000110001001101101111";


begin
mainproc:process(CLK)
variable C1             :floatvect7;
variable C2             :floatvect7;
variable XEST           :floatvect11;
variable XLOC            :std_logic_vector(31 downto 0);
variable OLD_XLOC        :std_logic_vector(31 downto 0);
variable YTEMP           :std_logic_vector(31 downto 0);
variable OLD_FRES       :std_logic_vector(31 downto 0);
variable INDEX          :integer range 0 to 11;
variable STATE          :integer range 0 to 99;       -- note - fix this range when you
know more
variable C1NOT2          :std_logic;
variable SIGN            :std_logic;
variable COARSE_SCAN     :std_logic;
variable FIRST_TIME     :std_logic;
variable SKIP_FLAG      :std_logic;
begin
```

```
if START = '0' then
        DONE <= '0';
        STATE := 0;
        XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        XDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        CHEBSTART <= '0';
        DEBUG_BIT <= '0';

else
        -- Here is the state machine that executes when START is asserted
        case STATE is

                when 0 =>
                -- First thing, we copy all inputs into internal variables
                C1(0) := C1_0;
                C1(1) := C1_1;
                C1(2) := C1_2;
                C1(3) := C1_3;
                C1(4) := C1_4;
                C1(5) := C1_5;
                C1(6) := C1_6;
                C2(0) := C2_0;
                C2(1) := C2_1;
                C2(2) := C2_2;
                C2(3) := C2_3;
                C2(4) := C2_4;
                C2(5) := C2_5;
                C2(6) := C2_6;
                FIRST_TIME := '1';
                COARSE_SCAN := '1';
                XLOC := one_float;
                OLD_XLOC := one_float;
                C1NOT2 := '0'; -- start with C2
                INDEX := 0;
                STATE := STATE + 1;

                when 1 =>
                -- We evaluate the Chebyshev polynomial at this point
                CL0 <= C2(0);
                CL1 <= C2(1);
                CL2 <= C2(2);
                CL3 <= C2(3);
                CL4 <= C2(4);
                CL5 <= C2(5);
                CL6 <= C2(6);
                CX <= XLOC;
                CHEBSTART <= '1';
                STATE := STATE + 1;

                when 2 =>
                if CHEBDONE = '1' then
                    CHEBSTART <= '0';
                    STATE := STATE + 1;
                    SIGN := CY(31);
                end if;

                when 3 =>
                if C1NOT2 = '1' then
                CL0 <= C1(0);
                CL1 <= C1(1);
                CL2 <= C1(2);
                CL3 <= C1(3);
                CL4 <= C1(4);
                CL5 <= C1(5);
                CL6 <= C1(6);
                else
                CL0 <= C2(0);
```

```
CL1 <= C2(1);
CL2 <= C2(2);
CL3 <= C2(3);
CL4 <= C2(4);
CL5 <= C2(5);
CL6 <= C2(6);
end if;
CX <= XLOC;
CHEBSTART <= '1';
STATE := STATE + 1;

when 4 =>
if CHEBDONE = '1' then
        CHEBSTART <= '0';
        SKIP_FLAG := '0';
        STATE := STATE + 1;
        -- DEBUG - double check this logic
        if ((CY(31) /= SIGN)) then
                DEBUG_BIT <= '1';
                if COARSE_SCAN = '1' then
                        -- a zero crossing has been detected.  Go to
                        fine resolution scan.
                        COARSE_SCAN := '0';
                        XLOC := OLD_XLOC;
                        STATE := 3;
                else
                                STATE := 10;
                                SKIP_FLAG := '1';

                end if;
        end if;
        if SKIP_FLAG = '0' then
                -- otherwise update XLOC.
                DEBUG_FLOAT <= XLOC;
                OLD_XLOC := XLOC;
                XADD <= XLOC;
                if COARSE_SCAN = '1' then
                        YADD <= coarse_inc;
                        OLD_FRES :=
                        "00000000000000000000000000000000";
                else
                        OLD_FRES := CY;
                        YADD <= fine_inc;
                end if;
        end if;
end if;

when 5 =>
XLOC := ZADD;
-- Here we check if XLOC is greater than 1
XADD <= XLOC;
XADD(31) <= not XLOC(31);
YADD <= one_float;
STATE := STATE + 1;

when 6 =>
if ZADD(31) = '0' then
        -- we're done since XLOC must be greater than +1
        DONE <= '1';
        STATE := 0;
else
         -- otherwise we keep evaluating.
        STATE := 3;
end if;

when 10 =>
-- Here we do the linear interpolation
-- First we subtract the current result from the last fine result.
XADD <= OLD_FRES;
YADD <= CY;
YADD(31) <= not CY(31);        -- invert for subtraction.
```

```
                        STATE := STATE + 1;

                        when 11 =>
                        -- next we divide this result into the fine increment
                        XDIV <= fine_inc;
                        YDIV <= ZADD;
                        STATE := STATE + 1;

                        when 12 =>
                        -- then we multiply by the last fine increment result
                        XMULT <= ZDIV;
                        YMULT <= CY;
                        STATE := STATE + 1;

                        when 13 =>
                        -- lastly we add this to our current xguess
                        XADD <= OLD_XLOC;
                        YADD <= ZMULT;
                        STATE := STATE + 1;

                        when 14 =>
                        -- we have found the approximate root location
                         XEST(INDEX) := ZADD;
                         INDEX := INDEX + 1;
                         -- switch to the other polynomial and begin scanning
                         C1NOT2 := not C1NOT2;
                         STATE := 3;

                        when others =>
                        -- Offload the results to the outputs
                        X0 <= XEST(0);
                        X1 <= XEST(1);
                        X2 <= XEST(2);
                        X3 <= XEST(3);
                        X4 <= XEST(4);
                        X5 <= XEST(5);
                        X6 <= XEST(6);
                        X7 <= XEST(7);
                        X8 <= XEST(8);
                        X9 <= XEST(9);
                        XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        XDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        YDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                        DONE <= '1';

                end case;
        end if;
end process;
end beh;
```

8.6. *ACOS* VHDL LISTING

```
library IEEE;
use IEEE.std_logic_1164.all;

entity acos is
        port (
        -- X0-X9 are the zero crossing locations in the Chebyshev domain
        X0,X1,X2,X3,X4,X5,X6,X7,X8,X9          :in std_logic_vector(31 downto 0);
        -- LSF0-9 are the line spectral frequencies.
        LSF0,LSF1,LSF2,LSF3,LSF4,LSF5,LSF6,LSF7,LSF8,LSF9     :out std_logic_vector(31
                                                               downto 0);
        -- XMULT,YMULT and ZMULT are interfaces to an external floating point multiplier
        XMULT,YMULT             :out std_logic_vector(31 downto 0);
        ZMULT                   :in std_logic_vector(31 downto 0);
        XADD,YADD               :out std_logic_vector(31 downto 0);
```

```
        ZADD                    :in std_logic_vector(31 downto 0);
        XDIV,YDIV               :out std_logic_vector(31 downto 0);
        ZDIV                    :in std_logic_vector(31 downto 0);
        CLK                     :in std_logic;
        START                   :in std_logic;
        DONE                    :out std_logic);
end acos;

architecture beh of acos is
subtype float32 is std_logic_vector(31 downto 0);
type floatvect11 is array (10 downto 0) of float32;
type floatvect3 is array (2 downto 0) of float32;
constant three_float:std_logic_vector(31 downto 0) := "01000000010000000000000000000000";
constant six_float:std_logic_vector(31 downto 0) := "01000000110000000000000000000000";
constant fifteen_float:std_logic_vector(31 downto 0) :=
"01000001011100000000000000000000";
constant forty_float:std_logic_vector(31 downto 0) := "01000010001000000000000000000000";
constant three_three_six_float:std_logic_vector(31 downto 0) :=
"01000011101010000000000000000000";
constant pi_over_two_float:std_logic_vector(31 downto 0) :=
"00111111110010010000111111010000";

begin
mainproc:process(CLK)
variable X              :floatvect11;
variable LSF            :floatvect11;
variable POWX           :floatvect3;
variable INDEX :integer range 0 to 10;
variable POWIDX         :integer range 0 to 2;
variable STATE :integer range 0 to 99;      -- note - fix this range when you know more
variable POWER :integer range 0 to 7;
variable TEMP  :std_logic_vector(31 downto 0);
begin
if START = '0' then
        DONE <= '0';
        STATE := 0;
        XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        XDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        YDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

else
        -- Here is the state machine that executes when START is asserted
        case STATE is

                when 0 =>
                -- First thing, we copy all inputs into internal variables
                INDEX := 0;
                X(0) := X0;
                X(1) := X1;
                X(2) := X2;
                X(3) := X3;
                X(4) := X4;
                X(5) := X5;
                X(6) := X6;
                X(7) := X7;
                X(8) := X8;
                X(9) := X9;
                POWIDX := 0;
                STATE := STATE + 1;

                when 1 =>
                -- now we start the Taylor series expansion for each value by taking
                -- the powers of x we need (3, 5, 7, etc.)
                case POWIDX is
                        when 0 =>
                        POWER := 3;
                        when 1 =>
                        POWER := 5;
```

```
        when 2 =>
        POWER := 7;
end case;
STATE := STATE + 1;

when 2 =>
XMULT <= X(INDEX);
YMULT <= X(INDEX);
STATE := STATE + 1;

when 3 =>
POWER := POWER - 1;
if POWER = 0 then
        -- We're done with this one!
        POWX(POWIDX) := ZMULT;
        if POWIDX = 2 then
                STATE := STATE + 1;
        else
                STATE := 1;
                POWIDX := POWIDX + 1;
        end if;
else
        TEMP := ZMULT;
        XMULT <= TEMP;
        YMULT <= X(INDEX);
end if;

when 4 =>
-- X + X^3/6
XDIV <= POWX(0);
YDIV <= six_float;
STATE := STATE + 1;

when 5 =>
XADD <= X(INDEX);
TEMP := ZDIV;
YADD <= TEMP;
STATE := STATE + 1;

when 6 =>
XMULT <= three_float;
YMULT <= POWX(1);
STATE := STATE + 1;

when 7 =>
TEMP := ZMULT;
XDIV <= TEMP;
YDIV <= forty_float;
STATE := STATE + 1;

when 8 =>
TEMP := ZADD;
XADD <= TEMP;
TEMP := ZDIV;
YADD <= TEMP;
STATE := STATE + 1;

when 9 =>
XMULT <= fifteen_float;
YMULT <= POWX(2);
STATE := STATE + 1;

when 10 =>
TEMP := ZMULT;
XDIV <= TEMP;
YDIV <= three_three_six_float;
STATE := STATE + 1;

when 11 =>
TEMP := ZADD;
XADD <= TEMP;
```

```
                TEMP := ZDIV;
                YADD <= TEMP;
                STATE := STATE + 1;

                when 12 =>
                XADD <= pi_over_two_float;
                TEMP := ZADD;
                YADD <= TEMP;
                YADD(31) <= not ZADD(31);
                STATE := STATE + 1;

                when 13 =>
                LSF(INDEX) := ZADD;
                INDEX := INDEX + 1;
                if INDEX = 9 then
                        -- we're finished
                        STATE := STATE + 1;
                else
                        STATE := 4;
                end if;

                when others =>
                -- Offload the results to the outputs
                LSF0 <= LSF(0);
                LSF1 <= LSF(1);
                LSF2 <= LSF(2);
                LSF3 <= LSF(3);
                LSF4 <= LSF(4);
                LSF5 <= LSF(5);
                LSF6 <= LSF(6);
                LSF7 <= LSF(7);
                LSF8 <= LSF(8);
                LSF9 <= LSF(9);
                XMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YMULT <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                XADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YADD <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                XDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                YDIV <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                DONE <= '1';

        end case;
end if;
end process;
end beh;
```

9. References

[1] P. Kabal and R.P. Ramachandran. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34, No. 6, pp. 1419-1425, December 1986.

[2] R.P. Ramachandran, M.M. Sondhi, N. Seshadri, B.S. Atal, "A Two Codebook Format for Robust Quantization of Line Spectral Frequencies", *IEEE Transactions of Speech and Audio Processing*, Vol. 3, No. 3, pp. 157-168, May 1995.

[3] C.H. Wu and J.H. Chen, "A Novel Two-Level Method for the Computation of the LSP Frequencies Using a Decimation in Degree Algorithm", *IEEE Transactions on Speech and Audio Processing*, Vol. 5, No. 2, pp. 106-115, March 1997.

[4] Faruque Saleh, *Cellular Mobile Systems Engineering*, Artech House, 1996.

[5] L.H. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice Hall, 1978.

[6] Insert old reference 9.??????

[7] F. Itakura, "Line Spectrum Representation of Linear Predictive Coefficients", *Journal of the Acoustical Society of America*, Vol. 57 Suppl. No. 1, p. S35, 1975.

[8] J. Rothweiler, "A Rootfinding Algorithm for Line Spectral Frequencies", *Proceedings of the 1999 International Conference on Acoustics, Speech and Signal Processing*, pp. II-661 – II-664, 1999.

[9] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1992.

[10] M.A. Bayoumi, "VLSI Architectures for DSP Applications: Current Trends", *IEEE Midwest Symposium on Circuits and Systems*, pp. 150-153, August 1992.

[11] G.A. Jullien and M.A. Bayoumi, "A Review of VLSI Technologies in Digital Signal Processing", *IEEE International Symposium on Circuits and Systems*, pp. 2347-2350, June 1991.

[12] M.J.S. Smith, *Application Specific Integrated Circuits*, Addison-Wesley, 1997.

[13] W.P. Burleson, M. Ciesielski, F. Klass and W. Liu, "Wave Pipelining: A Tutorial and Research Survey", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 6, Iss. 3, pp. 464-474, 1998.

[14] D.L. Reynolds, R.P. Ramachandran, L.M. Head, "VLSI Architecture for the Efficient Computation of Line Spectral Frequencies", *IEEE International Conference on Circuits and Systems*, Vol. 3, pp. 718-721, May 2003.