1-1-2009

High Level FPGA Implementation Of Adaptive Signal Segmentation And Autoregressive Modeling Techniques

Beibei. Jiao
Ryerson University

HIGH LEVEL FPGA IMPLEMENTATION OF ADAPTIVE SIGNAL SEGMENTATION AND AUTOREGRESSIVE MODELING TECHNIQUES

by

Beibei Jiao

B.Eng., Wuhan University, P.R. China, 2004

A thesis
presented to Ryerson University
in partial fulfillment of the
requirement for the degree of
Master of Applied Science
in the Program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2009

Author's Declaration

I hereby declare that I am the sole author of this thesis.
I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.
Signature

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
Signature

Abstract

HIGH LEVEL FPGA IMPLEMENTATION OF ADAPTIVE SIGNAL SEGMENTATION AND AUTOREGRESSIVE MODELING TECHNIQUES

Beibei Jiao

Master of Applied Science

Department of Electrical and Computer Engineering

Ryerson University, 2009

This thesis contains new FPGA implementations of adaptive signal segmentation and autoregressive modeling techniques. Both designs use Simulink-to-FPGA methodology and have been successfully implemented onto Xilinx Virtex II Pro device. The implementation of adaptive signal segmentation is based on the conventional RLSL algorithm using double-precision floating point arithmetic for internal computation and is programmable for users providing data length and order selection functions. The implemented RLSL design provides very good performance of obtaining accurate conversion factor values with a mean correlation of 99.93% and accurate boundary positions for both synthesized and biomedical signals. The implementation of autoregressive (AR) modeling is based on the Burg-lattice algorithm using fixed point arithmetic. The implemented Burg design with order of 3 provides good performance of calculating AR coefficients of input biomedical signals.

Acknowledgments

I would like to express my deep gratitude to my supervisors Dr. Sridhar Krishnan and Dr. Adnan Kabbani at Ryerson University for their knowledgeable guidance and constant encouragement and support.

I also would like to greatly acknowledge those who have assisted me during my graduate study at Ryerson University. Particularly, I would like to thank all the members in SAR group and my friends for their kindness and encouragement.

Finally, I would like to especially thank my parents for their nonstop and warm support.

Contents

List of Tables

List of Acronyms

ACF - Autocorrelation function

AR - Autoregressive

ARMA - Autoregressive moving-average

ASIC - Application-specific integrated circuit

DSP - Digital signal processing

ECG - Electrocardiogram

EDA - Electronic design automation

EEG - Electroencephalogram

EGG - Electrogastrogram

EMG - Electromyogram

FFT - Fast Fourier transform

FPGA - Field programmable gate array

GLR - Generalized likelihood ratio

HDL - Hardware description language

IP - Intellectual property

LMS - Least-mean-square

LNS - Logarithmic numbers system

MA - Moving-average

MSE - Mean squared error

PCG - Phonocardiogram

QRD-LSL - QR-decomposition-based least-squares lattice

RAM - Random access memory

RLS - Recursive least-squares

RLSL - Recursive least-squares lattice

SEM - Spectral error measure

STFT - Short-time Fourier transform

TSE - Total squared error

VAG - Vibroarthrogram

VHDL - VHSIC hardware description language

VLSI - Very large scale integration

Chapter 1

Introduction

1.1 Motivation

It is well known that, in the real physical world, there exist abundant kinds of signals. All those signals carry lots of information that are of people's interests. People develop diverse techniques to analyze, interpret, manipulate, and process those signals, trying to obtain the information they are interested in.

Biomedical signals are one type of signals that have strong relationships with human bodies or human organisms, such as ECG (electrocardiogram) related to the heart, EMG (electromyogram) related to the skeletal muscle fibers, EEG (electroencephalogram) related to the brain, EGG (electrogastrogram) related to the stomach, PCG (phonocardiogram) related to the heart and blood, VAG (vibroarthrogram) related to the knee joint, speech signals and so forth [1]. These signals are all generated by human organisms and thus, they certainly carry significant information about these organisms. Having such information can help people understand humans better and then for further purpose, can help improve health care and the quality of life of individuals. Hence, biomedical signal analysis has attracted abundant researchers' attention and has huge clinical significance. For example, ECG is one of the simplest and fastest procedures used to evaluate the condition of heart. The electrical activity of patient's heart is measured, interpreted, and printed out for the physician's information and further interpretation. The results can provide important clues to the cardiologist about the need for further testing to assess for the possibility of either

structural or electrical abnormalities.

Biomedical signal processing covers many areas, including filtering; spectral analysis; modeling for feature representation and parameterization; and quantitative or objective analysis of physiological systems and phenomena [1]. It uses signal processing methods and algorithms implemented on computers or in electric hardware to objectively analyze biomedical signals. Although with the help of computers, some subjective errors caused by human observers can be easily avoided, i.e. errors caused by fatigue, there are still many difficulties in biomedical signal acquisition, processing and analysis, such as difficulties in accessibility to the targets, dynamic nature of biological systems, interactions and inter-relationships among physiological systems, physiological artifacts and interference, and energy limitations [1]. However, people can still use specific methods to solve those problems. For instance, the dynamic nature of biological systems is one main difficulty as it causes the signal to have stochastical and nonstationary behavior. To conquer this issue, the parametric modeling method can be used to solve the *stochastic* problem and segmentation method can be used to solve *nonstationary* problem. Many researches have demonstrated that parametric modeling is a useful method when dealing with random time series [2] [3] [4] and segmentation is an efficient approach to deal with nonstationary signals [1] [5] [6]. All of these researches are achieved by using modern computers, which rely on softwares, such as Matlab.

Since 1970s, VLSI (Very Large Scale Integration) technology has dramatically changed the world and human lives. Nowadays, one can not imagine a life without VLSI-chip dependent devices, like computers, digital cameras, cell phones, MP3 players, digital TV sets and so forth. All those digital products are relying on VLSL chips, including both ASIC (Application-Specific Integrated Circuit) and FPGA (Field Programmable Gate Array). In most recent years, the FPGA technology has been significantly developed and gained people's more and more preference due to its advantages of stronger functions of FPGA itself, shorter time to market, ability to reprogram and lower non-recurring engineering costs. Many applications have been achieved by using FPGA techniques in various areas, i.e. digital signal processing, aerospace, medical imaging, computer vision, speech recognition, ASIC prototyp-

ing, bioinformatics. More importantly, with the recent EDA (Electronic design automation) tools, more flexible and efficient high-level design methodology can be applied, such as C-to-FPGA [7], Stateflow diagram to VHDL (SF2VHD) [8], Matlab-to-FPGA (MATCH) [9] [10], Simulink-to-FPGA [11] [12]. In this research, the Simulink-to-FPGA design flow has been chosen to implement two biomedical algorithms for specific applications: adaptive segmentation and AR (autoregressive) modeling. The reason for choosing Simulink-to-FPGA design flow is that with the combination of Xilinx system generator and implementation tools, one can implement designs in a graphical and flexible way.

1.2 Research objectives

The objective of this research is to develop high level FPGA implementations of popular biomedical signal segmentation and modeling algorithms for further real-time processing purpose. The two algorithms are: RLSL (recursive least-squares lattice) algorithm for adaptive segmentation and Burg-lattice algorithm for AR modeling. The overview of this research is shown in Figure 1.1.

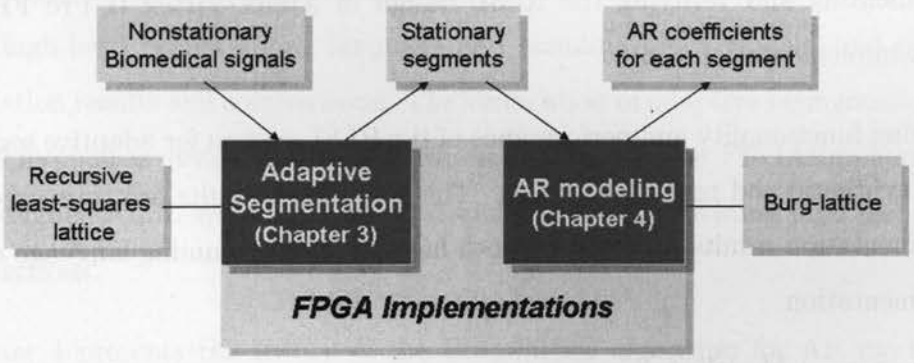

Figure 1.1: Overview of the research work

The RLSL hardware system should be able to calculate the conversion factor *gamma* values of each input sample for a pre-determined system order, which can be further used to be compared with a threshold value for adaptive segmentation. The Burg-lattice hardware system should be able to calculate the corresponding AR coefficients for the input data based

on a certain system order, which can be used for further AR modeling.

1.3 Original contributions

The main contributions of this research are described as follows:

Adaptive Segmentation

- Proposing a new system-level architecture for RLSL algorithm with a specific purpose of adaptive segmentation, which could be used for further real-time processing objective. The proposed design has an adjustable (programmable) order selection function and data length selection function, which are up to 16 stages and 5000 samples, respectively.
- Implementing design of RLSL algorithm using double precision floating point arithmetic for internal computation and data storage. The input data and output data are represented by using fixed point arithmetic type, which are easy to read and analyze.
- Implementing and verifying the RLSL design in Xilinx Virtex II Pro FPGA using Xilinx implementation tools.
- Verifying functionality and performance of the RLSL system for adaptive segmentation using synthetic and real knee signals. The simulation results are compared with the implementation results provided by both high-level programming language and FPGA implementation.

Autoregressive Modeling

- Designing the Burg-lattice algorithm on system-level with a specific purpose of calculating model parameters.
- Implementing and verifying the Burg-lattice design in Xilinx Virtex II Pro FPGA and comparing the results with ones obtained by using high-level programming languages.

1.4 Thesis organization

This thesis consists of five chapters:

- Chapter 1 introduced the significance of biomedical signal analysis and the reason why biomedical signal processing attracted researchers' interests. It also states the objectives of the project, the contribution of the author and the organization of this thesis.

- Chapter 2 starts with a review of adaptive segmentation methods of biomedical signals and presents the reasons why to choose the target algorithm: RLSL algorithm for hardware implementation. It provides an overview of parametric modeling, particularly AR modeling and the reason for choosing Burg-lattice algorithm for hardware implementation. It also presents a review of hardware implementation of the two applications and algorithms.

- Chapter 3 presents the theory of the RLSL algorithm for adaptive segmentation. The mathematical representation is described in details. It also presents the implementation with high-level programming language and simulink block-models, and provides the simulation results and comparisons. The verification of adaptive segmentation applying RLSL method is stated by using synthesized signals. The FPGA implementation of the designed RLSL system is presented and discussed, providing final test results and comparisons.

- Chapter 4 presents the theory of the Burg-lattice algorithm for AR modeling. The mathematical representation is provided in details as well as the advantages of the algorithm. Implementations with high-level programming language and simulink block models of Burg-lattice algorithm are followed. The simulation results and comparisons are displayed, followed by the FPGA implementation details and testing results.

- Chapter 5 concludes the thesis and presents discussion for future work of this research.

Chapter 2

Review

In this chapter, the reason why to choose adaptive segmentation rather than fixed segmentation and a review of adaptive segmentation techniques for biomedical signals will be presented first. Then a brief introduction on parametric modeling, particularly AR (autoregressive) modeling will be followed. At last, a review of VLSI implementations of the two selected algorithms will be stated.

2.1 Adaptive segmentation

Biomedical signals are typical *nonstationary* random signals. They are non-stationary as the statistical properties of such signals vary with time, i.e. the mean value varying with time, or having time-varying frequency spectrum. This causes challenges when one wants to use classical, well-developed spectrum analysis methods (i.e. Fourier transform) to analyze such signals. Take the EEG analysis as an example. The description of EEG in spectral domain can be used for feature extraction [13]. This requires the target signal to be stationary. Unfortunately, during its long time of observation, the EEG signal is non-stationary in nature. However, it can be considered to be locally stationary over short time intervals [13] [14]. Hence, using segmentation technique to partition the signal into stationary components is a straightforward and natural idea.

There are two categories of segmentation. One is fixed segmentation and the other is adaptive segmentation. Fixed segmentation uses fixed-size window for segmentation and it

is usually used in specific applications, such as STFT (short-time Fourier transform) [1]. Although fixed segmentation is simple but its performance is not efficient enough. The main problem for this method is the choice of the window size. Apparently, the window length should be long enough for meaningful analysis and also short enough to make sure that each segment is stationary. On one hand, selecting large fixed-size windows, the signal may still have dynamics during the window duration chosen, which actually requires more segments. On the other hand, selecting small fixed-size windows, the signal may still have stationary status much longer than that window duration, which actually hopes to use larger window size. In short, this method can not guarantee stationarity for each window and can not guarantee that the chosen window size is appropriate. Moreover, short-time analysis could be computationally expensive [1]. For example, the Fourier transform needs to be calculated for each segment of the signal in STFT method. Thus, if the window size is not chosen appropriately, it may have lots of segments in total, which would make the whole computation of using STFT method become very expensive and redundant. Therefore, it is much desirable and reasonable to find a more efficient segmentation method for non-stationary signals.

Adaptive segmentation is an alternative based on a more efficient idea, compared with fixed segmentation. It segments the signal when it is needed according to the dynamic characteristics of the signal itself. This means that the analysis window is kept as long as possible when the signal remains stationary, whereas set up a boundary and used a new window when the signal changes its properties abruptly and dramatically.

There are several approaches that have been used for adaptive segmentation of synthesized and/or real biomedical signals. These approaches are listed and described below.

SEM (spectral error measure)

Bodenstein and Praetorius [13] used SEM for adaptive segmentation of EEG signal and for further feature extraction. This method is based on AR modeling, using a fixed-length reference window for calculation of AR parameters in the reference window. And those AR parameters are used to model the samples in a test window and calculate the ACF

(autocorrelation function) of the test window, which is finally used to define a spectral error measure (SEM). If the error measure is larger than a specific threshold, then a segment boundary is set up and the procedure starts again. This SEM method was also applied by Tavathia et al. [3] for knee joint vibroarthrography's (VAG) adaptive segmentation.

ACF (autocorrelation function)

Michael and Houchin [15] used ACF method for adaptive segmentation, which directly estimated the values of short-time autocorrelation function of the signal to segment the boundaries. This method is the most general in that it does not make use of any explicit model, e.g. AR model. It uses a reference window with fixed-size at the beginning of each segment, and calculates the ACF for the reference window. Then it uses a sliding window (also called as test window) with equal length to the signal and calculates the ACF for the test window. If the difference between the ACF in the sliding test window and the reference window is significant, a segment boundary is set up and then repeats the procedure.

GLR (the generalized likelihood ratio)

Appel and Brandt [16] used GLR method for adaptive segmentation. The difference between this method and the previous two is that it uses a continuously growing reference window. The test window is a fixed-size sliding window like the previous two methods. The advantage of using the growing reference window is that it contains the maximum of information available from the beginning of each segment to the current instant, thus leading a minimum variance of the estimation of the reference parameters.

Appel and Briandt also compared these three methods in [17] using both synthesized signals and real EEG signals. They found that the GLR method provided the best performance among the three but had the highest computational complexity.

RLS (recursive least-squares)

Moussavi et al [6] used the RLS filter for adaptive segmentation of VAG signals. The advantage of this method, compared with SEM, ACF and GLR, is that it does not require any reference and test windows, but calculates the filter tap-weight vector for each sample of the input signal. It could provide good performance of adaptive segmentation of VAG

signals according to Moussavi's tests. However, this RLS method has high computational complexity and does not have the characteristic of modularity in nature for the convenience of hardware implementation.

RLSL (recursive least-squares lattice)

S.Krishnan et al [18] directly used an adaptive filter based on RLSL algorithm for adaptive segmentation of VAG signals. According to their research, this method could provide very good performance for biomedical signal segmentation. They made the input data run into the filter two times, wherein the first running to make the filter converge. For the second running, they compared the selected parameter to a predefined threshold value for each sample. Once the parameter value fell down less than the threshold value, a boundary was set up. S.Krishnan [19] also tested this method using synthesized signal and obtained good results. This method is also a recursive method on a sample-by-sample basis and does not use any short-time windows as the RLS method. More importantly, it is using the lattice structure that makes it more attractive and suitable for hardware implementation.

RLSL algorithm belongs to the fast RLS category. It is based on lattice structure that makes it work faster than the conventional RLS algorithm, since the computational complexity increased linearly with the order M. It involves both time updates and order updates, which makes it distinguishing from many other adaptive algorithms [20].

There are many advantages for RLSL algorithm [20] [21] [22] [23] [24].

- Fast rate of convergence, compared with those algorithms in LMS (least-mean-squares) family.
- Good computational complexity, linear with M, not as square of M like the conventional RLS algorithm or QRD-LSL algorithm.
- High modularity on stage-by-stage, each complete iteration sharing exactly the same structure; parallelism; concurrency.
- Good tracking capability.

- Order recursion, based on lattice structure, leading to computational efficiency and modularity.

RLSL algorithm converges fast as it is derived from the least-squares family. This is the fundamental advantage for consideration when compared with the stochastic gradient approach family, whose most popular algorithm is known as LMS (least-mean-square) algorithm. Fast rate of convergence can make the filter adapt rapidly according to the signals' statistical characteristics. RLSL has good computational complexity: its computational complexity is linear with order M, referred to $O(M)$ as the same as LMS algorithm but better than conventional RLS (recursive least-squares) algorithm. This advantage is due to the order-recursion property of the algorithm: the information gathered from the previous computations for order $M-1$ is carried over to the next order M. By contrast, the conventional RLS and square-root RLS algorithm (which is derived as to solve the issue of numerical instability in conventional RLS) have a computational complexity linearly increasing with square of M, referred to $O(M^2)$, which may become annoying from a hardware implementation point of view, particularly when M is large. RLSL has very high stage-by-stage modularity property. For each iteration, it is sharing exactly the same structure, which is convenient and suitable for FPGA implementation. RLSL also has very good tracking ability that can provide good performance on monitoring the statistical changes in the signal, which has been proved by previous researchers [6], [18]. It has not only time recursion but also order recursion that is based on using a lattice structure. It is evident that the lattice structure itself leads to good modularity and computational efficiency and possess better numerical properties than direct structures [23].

There are also some limitations for this algorithm.

- Possible numerical instability, after hundreds of thousands of literation [20], [22], [23], [25], [26].
- Dynamic range of parameters [21].

Any DSP algorithms implemented or applied in digital systems suffer from the finite word length effects or also called finite-precision effects. In practical, one can not use infinite precision to employ the design digitally. The finite-precision effects influence the performance of a digital implementation of the algorithm and make it deviate from its theoretical value. The two main factors that influence the nature of the deviation are the details of the algorithm itself and the form of numerical computation employed (fixed point or floating point) [20].

For algorithm itself, the RLSL algorithm mentioned before in this chapter is referred as conventional RLSL or indirect updating RLSL algorithm, which has been actually used in many researches [6], [18] and [19] for adaptive segmentation of real biomedical signals. Ling et al [23] proposed a RLSL algorithm with error-feedback, commonly known as the *direct* updating RLSL. They pointed out that the direct version of RLSL has better numerical accuracy and numerical stability based on their computer simulation results. Bunch et al [25] gave the same conclusion but they also mentioned that both the RLSL algorithms had the potential for large relative errors. Nonetheless, no explicit researches have been done using the indirect RLSL algorithm for real signal's adaptive segmentation. Moreover, the direct updating RLSL has more computational complexity than the indirect one.

Paleologu et al in [21] mentioned that the conventional RLSL algorithm had a drawback of large dynamics of parameters. However, they only focused on using a modified cost function to reduce the dynamic range of parameters, nothing related to the hardware implementation and any specific applications.

To sum up, the indirect RLSL algorithm has both advantages and disadvantages as mentioned above. However, it has been already applied for adaptive segmentation of real biomedical signals and achieved satisfactory performance with the use of softwares and programs based on double precision floating point type arithmetic on general PCs. Thereby, double precision floating point arithmetic is chosen to implement the indirect RLSL algorithm and the questions then would become: is this accessible and worthy to implement this algorithm onto specific FPGA and is its performance acceptable? The answers to these two

questions and more details will be provided and discussed in Chapter 3.

2.2 Parametric modeling

Parametric modeling is a typical method in dealing with random signals, as long as the signals are stationary. The basic idea for parametric modeling is that the present value of model output is assumed to be the linear combination of several past values of model output plus the linear combination of present and past values of model input, expressed in the following equation [1].

$$y(n) = -\sum_{k=1}^{P} a_k y(n-k) + G\sum_{l=0}^{Q} b_l x(n-l) \tag{2.1}$$

where $b_0 = 1$, $x(n)$ is the model input, $y(n)$ is the model output, and G is the gain factor.

Applying z-transform to the above equation, it is easy to obtain its transfer function as

$$H(z) = \frac{Y(z)}{X(z)} = G\frac{1 + \sum_{l=1}^{Q} b_l z^{-l}}{1 + \sum_{k=1}^{P} a_k z^{-k}} \tag{2.2}$$

In most cases, the gain factor is not important [1], and thus the system is fully characterized by a_k and b_l. a_k and b_l determine if the system is an all-pole system or an all-zero system or a pole-zero system.

There are three main modeling methods: AR(Autoregressive) modeling, MA(Moving-average) modeling and ARMA(Autoregressive moving-average) modeling. AR models correspond to the situation that b_l in Equation (2.2) is all equal to zero, whereas MA models correspond to the situation that a_k is all zero, and for ARMA models, a_k and b_l are both not all equal to zero. Among these three methods, AR modeling is a very popular one particularly in dealing with biomedical signals for several reasons: 1) some biomedical signals (i.e. speech signal) have an underlying autoregressive structure; 2) generally, any signal (even if it is not necessarily AR in nature) can be modeled as an AR process as long as an appropriate model order is selected; 3) estimation of model parameters is based on finding out the solution of a linear system of equations and many efficient algorithms are available

to compute the solution [27] [28]. The following section briefly introduces the AR modeling method.

2.2.1 AR modeling

AR modeling is such a widely used method in biomedical signal processing. For AR modeling, the parameters of AR model are of interests and investigated for use in signal analysis.

It has been demonstrated that in many cases, AR spectrum provides a better resolution than traditional Fourier spectrum [1], which can ease the signal analysis. To obtain the AR spectrum, one has to obtain the AR coefficients of the signal first [29]. Moreover, AR coefficients can be easily used in pattern classification of biomedical signal [30], [31] and data compression application [32].

For an AR model, the output is modeled as the linear combination of P past values of the model output and the present model input (no past values of the model input are used) as [1]

$$y(n) = -\sum_{k=1}^{P} a_k y(n-k) + Gx(n) \tag{2.3}$$

Again, applying the z-transform to the above equation, then the AR transfer function is

$$H(z) = \frac{G}{1 + \sum_{k=1}^{P} a_k z^{-k}} \tag{2.4}$$

Factorizing the denominator polynomials in Equation (2.4), the transfer function can be expressed as

$$H(z) = \frac{G}{\prod_{k=1}^{P}(1 - p_k z^{-1})} \tag{2.5}$$

The parameters p_k, $(k = 1, 2, \ldots, P)$, are the poles of $H(z)$ or the system. The AR model only has system poles, no zeros and therefore, it is also called the all-pole model. The purpose here is to obtain those AR parameters (also known as AR coefficients).

In many cases of biomedical signals, e.g. the EEG or the PCG, the input $x(n)$ is totally unknown. Hence, the output $y(n)$ can be only approximately predicted as the linear combination of past values of the output

$$\widetilde{y}(n) = -\sum_{k=1}^{P} a_k y(n-k) \tag{2.6}$$

Obviously, there exists an error as

$$e(n) = y(n) - \widetilde{y}(n) = y(n) + \sum_{k=1}^{P} a_k y(n-k) \tag{2.7}$$

In the method of least-squares, which is derived in the time domain, the parameters a_k are obtained by minimizing the MSE (mean squared error) or TSE (total squared error) with respect to each of the parameters [1], [33]. There are several techniques that can do the job of computing the model coefficients, directly or iteratively. Generally, iterative methods cost more computation to achieve a desired degree of convergence than the direct methods [33]. There are some commonly used approaches for directly estimating prediction parameters: the autocorrelation method, the covariance method, the square-root or Cholesky decomposition method, and the Burg method. All these methods are trying to solve the *normal equations*, a set of p equations for the predictor coefficients $a_k, 1 \leq k \leq p$. Autocorrelation or covariance method requires large computational operations and storage locations. Square-root or Cholesky decomposition method has less computations compared with the previous two methods. Further reduction in computation and storage room can be achieved by using Levinson-Durbin algorithm, which provides a recursive method to solve the set of normal equations. This method has big savings in operations and storage locations from the previous methods [33]. The Burg algorithm is another most popular algorithm for non-adaptive AR models such as the Levinson-Durbin algorithm, where non-adaptive means that the model parameters are chosen to give the best fit of a *sequence* of data samples, not like adaptive models that the values of parameters are updated on the arrival of each new data sample [14]. Figure 2.1 shows the methods of parametric modeling [14].

One main advantage for Burg algorithm compared with Levinson-Durbin algorithm is that in obtaining the solution of AR parameters of order M, one just simply add one more stage without affecting the earlier computations for the lower stages, which is more suitable for VLSI consideration, while for Levinson-Durbin algorithm, one actually computes the

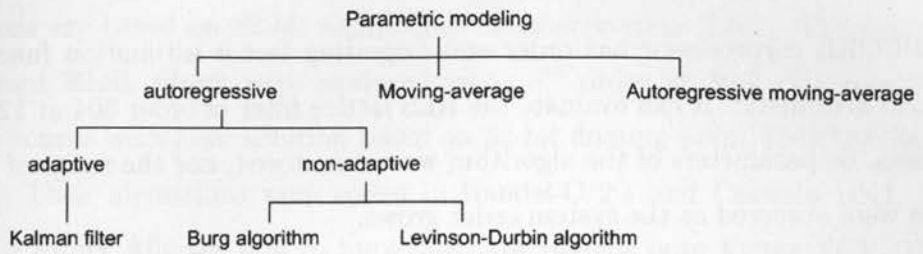

Figure 2.1: Methods for parametric modeling and algorithms for AR modeling

model parameters for all orders up to the desired order M. In addition, Levinson-Durbin algorithm could be numerically relatively unstable [33]. Hence, it is reasonable to conclude that using Burg algorithm to compute AR coefficients is a satisfactory choice and is more attractive for hardware implementation. More details of Burg algorithm will be provided in Chapter 4.

2.3 Review of hardware implementation of the two applications

To the best of the author's knowledge, there is no reported literature about hardware implementation of direct adaptive segmentation, but there are few papers discuss the implementation of RLSL algorithms (direct or indirect RLSL) for other applications.

Zdenek Pohl et al. [34] proposed an error-feedback RLSL filter (the direct RLSL) with the estimation of an unknown order and forgetting factor of identified system as a PCORE coprocessor for Xilinx EDK. Their design used the LNS (logarithmic numbers system) arithmetic. The proposed FPGA coprocessor implementation was able to evaluate the direct RLSL filter of order 504 at 12kHz input data sampling rate. The advantage and deficiency of their design are listed below:

- A hardware coprocessor rather than a standalone IP core. The PCORE coprocessor contains a RLS lattice core, which is based on RLSL with error-feedback algorithm (the direct RLSL algorithm as mention before). However, the coprocessor needs the standard C programming and debugging to get accessed, which is still software-based.

- The PCORE corprocessor has order and forgetting factor estimation function using the LNS arithmetic. It can evaluate the RLS lattice filter of order 504 at 12kHz. Nevertheless, no parameters of the algorithm were monitored, nor the range of parameter values were observed as the system order grows.

- The aim of the work was to provide a versatile highly configurable hardware RLSL (direct RLSL) core for DSP applications [34]. Hence, they did not have a specific application purpose of implementing RLSL algorithm, such as biomedical signal adaptive segmentation, which uses the chosen parameter to monitor the significant changes in the target biomedical signal and adaptively segment it.

Antonin Hermanek et al. [35] presented a FPGA implementation of a noise canceler with an adaptive RLS-lattice filter in Xilinx devices. They used LNS for internal computation and demonstrated that the noise canceler could run at the XSV800 prototyping board in real-time with 16 kHz sampling rate for the filter order of 160. They also provided the comparison of the performance of FPGA echo canceler and their implementation on TI TMS320C6711 (IEEE 32-bit floating point device) of 100^{th} order. The advantage and deficiency of their design are listed below:

- Used 19-bit LNS arithmetic for computations, decreasing the computational complexity of multiplications, divisions and square-root operations, but increasing the computational complexity of additions and subtractions. The detailed information of accuracy of such LNS arithmetic system was not provided yet.

- Used four pipelined macros running in parallel and each macro used lattice structure in sequential, hence providing up to $4n^{th}$ order computation. But the information about the usage of device area of their lattice noise cancellation design that could run 160 stages for real-time application was not provided.

F. Albu et al. [36] implemented the conventional RLSL and its normalized version on Virtex XCV2000E-6 operating with 24-bit fixed point input/output signals. Their internal

computations are based on 32-bit logarithmic number system (LNS). The normalized and un-normalized RLSL filters were analyzed using 8^{th} order at 36.7 kHz compared to the clock cycle counts with DSP solution based on 32-bit floating point TMS320C3x/4x 50Mhz processors. Their algorithms were coded in Handel-C 2.1 and Celoxica DK1. They used Synplify 5.3 Xilinx Alliance 3.3i to implement the designs onto Virtex XCV2000E-6. The advantage and deficiency of their design are listed below:

- Used 32-bit LNS arithmetic for internal computations, decreasing the computational complexity of multiplications, divisions and square-root operations, but increasing the computational complexity of additions and subtractions.
- Analyzed designs on FPGA under 8^{th} order at 36.7 kHz but they did not provided information about their design performance for higher system order.
- Implemented the RLSL algorithm without monitoring any parameters or taking advantage of using those parameters for specific applications, i.e. adaptive segmentation.

For AR modeling, one implementation can be found in [37]. They implemented the Burg algorithm onto the AMD29500 microprogrammable byte slice DSP and NECμPD77230 single-chip DSP. The AMD DSP system can have a sixteenth-order modeling rate at 17kHz while the NEC DSP system can have a sixteen-order model at 8kHz. The advantage and disadvantage of their design are listed below:

- Implemented the Burg algorithm onto a series of DSP microprocessor systems, not implementing on single VLSI chip, either FPGA or ASIC.
- Required DSP board and softwares for real application, e.g. processing medical images.

Summary

Adaptive segmentation using conventional RLSL method and AR modeling using Burg-lattice method are good choices for biomedical signal segmentation and modeling. Moreover, both methods have attractive properties to implement onto hardware. The next two chapters will describe these two methods in more details and their specific FPGA implementations.

Chapter 3

Adaptive segmentation with RLSL algorithm and hardware implementation

3.1 Theory of RLSL algorithm

The conventional RLSL (recursive least-squares lattice, sometimes regarding as RLS-lattice or recursive-LSL) algorithm involves both order and time recursions as mentioned in Chapter 2. It is first a time-recursive algorithm extended from the method of least-squares, and then involves order-recursions using lattice structure to reduce the high computational complexity for RLS algorithm.

The mathematical representations of the RLSL algorithm are expressed as follows [1].

Initialization:

- at $n = 0$, and for each order $m = 1, 2, \ldots, M$, set the cross-correlation $\Delta_{m-1}(0) = 0$; forward/backward prediction error power $F_{m-1}(0) = B_{m-1}(0) = \delta$; the conversion factor $\gamma_{0,c}(0) = 1$, where the index n represents the number of samples, M is the order of the system and δ is a small positive constant, e.g. $\delta = 0.001$.

- at $n \geq 1$,$m = 0$, set the forward/backward prediction error $f_0(n) = b_0(n) = u(n)$; $F_0(n) = B_0(n) = \lambda F_0(n-1) + \|u(n)\|^2$; $\gamma_{0,c}(n) = 1$, where $u(n)$ is the input data and λ is the forgetting factor, e.g.$\lambda = 1$.

- for joint-process estimation, at $n = 0$, $m = 0, 1, \ldots, M$, set the scalar $\rho_m(0) = 0$; at $n \geq 1$, $m = 0$, set a *priori* estimation error $e_0(n) = d(n)$, where $d(n)$ is the desired response.

Prediction

For $n = 1, 2, \ldots, N_s$, $m = 1, 2, \ldots, M$, the parameters are computed as follows:

$$\Delta_{m-1}(n) = \lambda \Delta_{m-1}(n-1) + \frac{b_{m-1}(n-1) f_{m-1}(n)}{\gamma_{m-1,c}(n-1)} \tag{3.1}$$

$\Delta_{m-1}(n)$ is the cross-correlation between $f_{m-1}(n)$ and $b_{m-1}(n-1)$.

$$\gamma_{m,f}(n) = -\frac{\Delta_{m-1}(n)}{B_{m-1}(n)} \tag{3.2}$$

$$\gamma_{m,b}(n) = -\frac{\Delta_{m-1}(n)}{F_{m-1}(n-1)} \tag{3.3}$$

$\gamma_{m,f}(n)$ and $\gamma_{m,b}(n)$ are the forward and backward reflection coefficients. Generally, $\gamma_{m,f}(n) \neq \gamma_{m,b}(n)$ as $B_{m-1}(n)$ and $F_{m-1}(n-1)$ are unequal. The reason to call this algorithm as *indirect* RLSL is that it needs to calculate the cross-correlation first, and then to update the forward/backward reflection coefficients, not directly updating the forward/backward reflection coefficients.

$$f_m(n) = f_{m-1}(n) + \gamma_{m,f}(n) b_{m-1}(n-1) \tag{3.4}$$

$$b_m(n) = b_{m-1}(n-1) + \gamma_{m,b}(n) f_{m-1}(n) \tag{3.5}$$

$f_m(n)$ and $b_m(n)$ are forward and backward prediction errors. They are updated based on a lattice structure as Figure 3.1 shows.

$$F_m(n) = F_{m-1}(n) + \gamma_{m,f}(n) \Delta_{m-1}(n) \tag{3.6}$$

$$B_m(n) = B_{m-1}(n-1) + \gamma_{m,b}(n) \Delta_{m-1}(n) \tag{3.7}$$

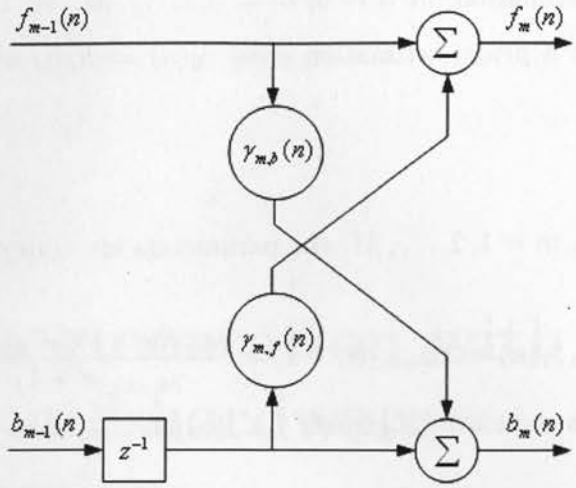

Figure 3.1: Lattice structure for one stage

$F_m(n)$ and $B_m(n)$ are the forward and backward prediction error powers.

The conversion factor $\gamma_{m,c}(n)$ is updated as:

$$\gamma_{m,c}(n) = \gamma_{m-1,c}(n) - \frac{b_{m-1}^2(n)}{B_{m-1}(n)} \tag{3.8}$$

Filtering

For $n = 1, 2, \ldots, N_s, m = 0, 1, \ldots, M$:

$$\rho_m(n) = \lambda \rho_m(n-1) + \frac{b_m(n)}{\gamma_{m,c}(n)} e_{m-1}(n) \tag{3.9}$$

$$\kappa_m(n) = \frac{\rho_m(n)}{B_m(n)} \tag{3.10}$$

$$e_m(n) = e_{m-1}(n) - \kappa_m(n) b_m(n) \tag{3.11}$$

where $\rho_m(n)$ is the scalar, $\kappa_m(n)$ is the regression coefficient and $e_m(n)$ is the a *posteriori* estimation error.

According to [18] [19], the conversion factor γ_c is a good choice for monitoring the statistical changes in the target non-stationary signals. The input data is required to pass

the filter twice. The first pass is to make the filter converge, and the second pass is to obtain the conversion factor γ_c for each sample (the second round γ_c), which is needed for the further purpose: compare each γ_c with a predefined threshold value to detect the segment boundaries. After a few iterations at the beginning, γ_c gets close to unity. If there is a significant change in the target signal, there will be a dramatical drop of the γ_c value. Whether the drop of the γ_c value can be considered as a *dramatical* drop is determined by the threshold. If the γ_c value falls down below the threshold, it is believed to have a dramatical drop and should set up a boundary at that instant. Hence, the determination of the threshold value is very important as well. However, this value is user-defined depending on different applications and different kinds of signals. In [19], the threshold value is chosen as 0.9985 to segment VAG signals. In [1], the RLSL algorithm is used to segment PCG signals using a threshold of 0.995. It is reasonable to conclude that to obtain good performance of adaptive segmentation, high precision is certainly required to represent the conversion factor values. For example, if using 0.9985 as the threshold, the RLSL system should be able to distinguish the difference between 0.9985 and 0.9986. Undoubtedly, double precision floating point arithmetic can meet this requirement.

3.2 Implementation and verification of adaptive segmentation based on high-level language

To test and verify the selected method of using RLSL algorithm for adaptive segmentation, a C program has been created to implement the RLSL algorithm based on double precision floating point on a general PC. Meanwhile, a synthesized nonstationary signal has been generated for verifying the developed C program.

Figure 3.2 shows the flow chart of the RLSL algorithm. According to Figure 3.2, after initialization, the parameters of the first sample, including cross-correlation, forward/backward reflection coefficients, forward/backward prediction errors, forward/backward prediction error powers and conversion factor, will be updated for each stage until reaching the last stage (the system order). After the first sample finishes its update and obtains its conversion

factor value of the last stage, the second sample starts to update with the same procedure until the last sample finishes its updates. Furthermore, the 2^{nd} round conversion factor γ_c of each sample is compared with a threshold to detect boundaries.

Figure 3.3 shows the plot of a synthesized nonstationary signal. It consists of three segments: s1, s2 and s3. Each of them is a sequence of 200 samples, modeled and generated by a 3^{rd} AR model based on Equation (3.12). The adjacent segments are modeled differently by means of changing one AR coefficient. Thus, the whole synthesized signal is obviously non-stationary. Figure 3.4 shows the FFT-spectrum plots of the three segments in Figure 3.3. It is evident that the adjacent segments have different spectra, which means that they have different characteristics, leading the whole signal non-stationary. The reason of using such a synthesized signal is that we know exactly where the boundaries are and it will be appropriate to verify the segmentation algorithm with this synthetic signal before applying it to real-world biomedical signals. In this case, the boundaries are at 201 and 401 time samples.

Each segment of the whole synthetic signal can be mathematically represented as follow:

$$s(n) = w(n) + a_1 s(n-1) + a_2 s(n-2) + a_3 s(n-3) \tag{3.12}$$

where $w(n)$ is the random white noise, a_1 to a_3 are the AR coefficients and the initial values of $s(n)$ are equal to zeros, which means $s(-2)$, $s(-1)$ and $s(0)$ are equal to zero.

Figure 3.5 shows the second round conversion factor $\gamma_{m,c}$ values of each sample obtained by using order of 5 ($m = 5$) for RLSL. Apparently, there are two dramatical drops as shown in the figure, which are at 203 and 402, quite close to the actual time positions (201 and 401). If an appropriate threshold value has been chosen, e.g. 0.95, then, the two dramatical drops, whose values are both below 0.95, will definitely be picked up, indicating two boundaries, and then the segmentation into stationary segments would be done satisfactorily.

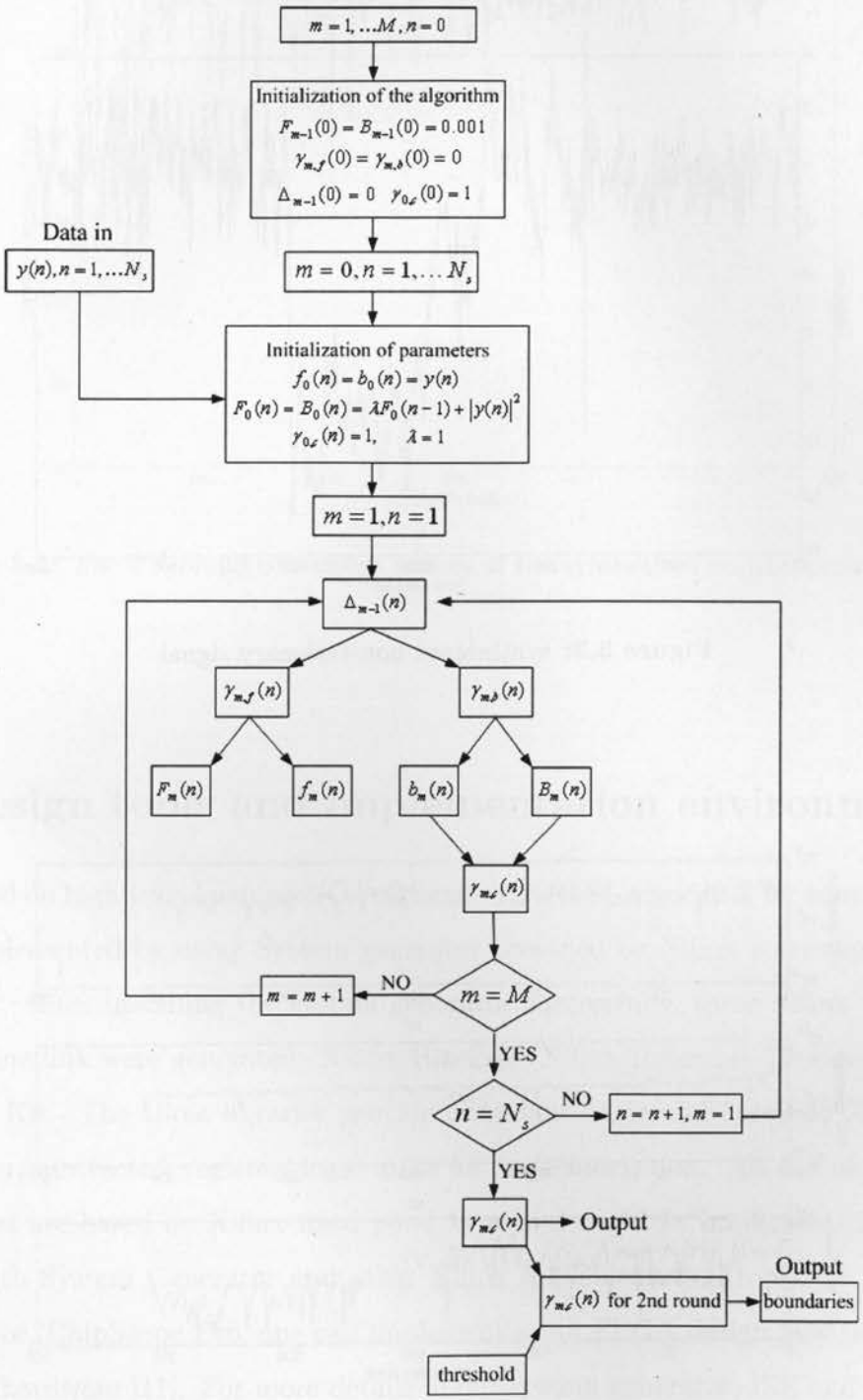

Figure 3.2: Flow chart of RLSL algorithm

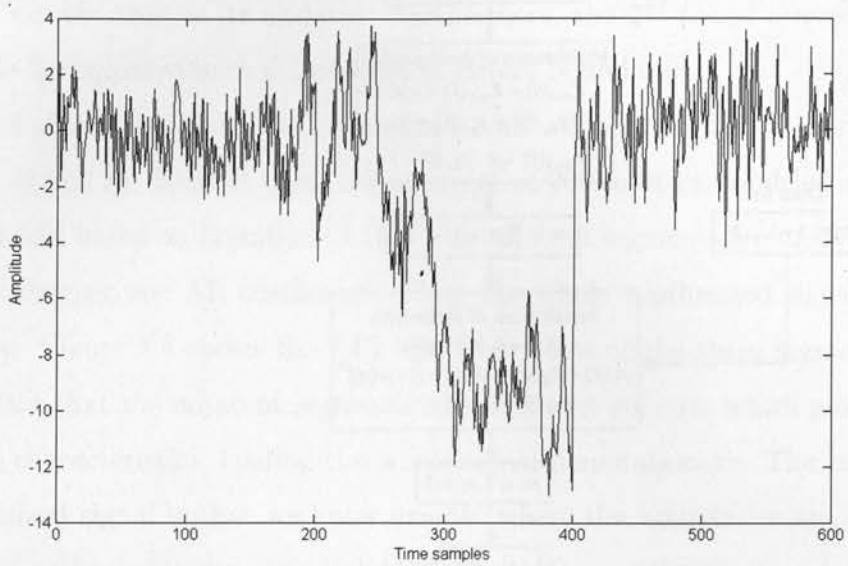

Figure 3.3: synthesized nonstationary signal

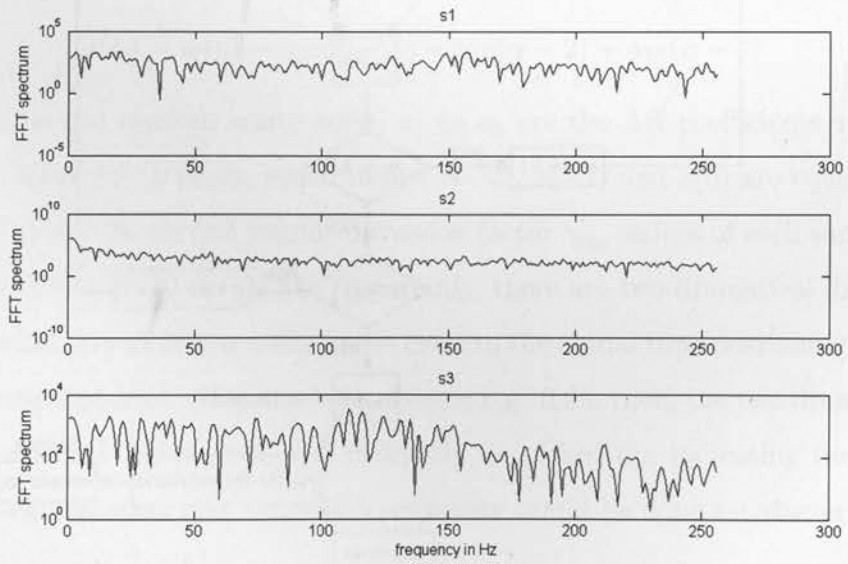

Figure 3.4: Spectra of the three segments consisting of the synthesized non-stationary signal

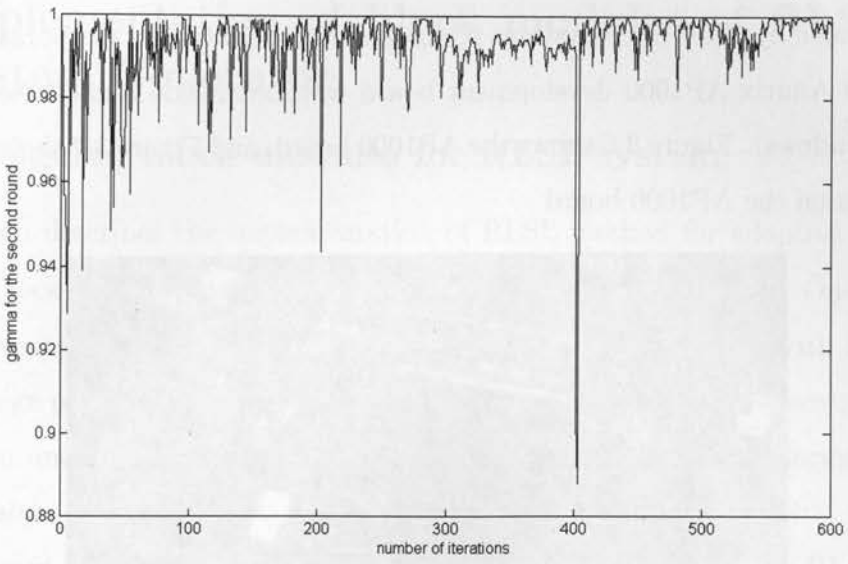

Figure 3.5: The 2^{nd} round conversion factors of the synthesized nonstationary signal

3.3 Design tools and implementation environment

After tested on high-level language (C language), the RLSL algorithm for adaptive segmentation is implemented by using System generator provided by Xilinx company in Simulink environment. After installing the system generator successfully, three Xilinx libraries embedded in Simulink were generated: Xilinx Blockset, Xilinx Reference Blockset and Xilinx XtremeDSP Kit. The three libraries provide abundant basic and specified Xilinx blocks, such as adder/subtracter, register, logic units for implementation. All the blocks in these three libraries are based on Xilinx fixed point type and could be implemented onto Xilinx FPGAs. With System Generator and other Xilinx implementation tools, i.e. Xilinx ISE, CoreGenerator, ChipScope Pro, one can implement a full FPGA design flow from Simulink modeling to hardware [11]. For more details about system generator, ISE and other Xilinx implementing tools, one can refer to [38] [39] [40].

In this research, the target FPGA chip is Xilinx Virtex II Pro xc2vp100-6ff1704. The

platform provided by CMC Microsystems (Canadian Microelectronics Corporation Microsystems) has the Amirix AP1000 development board installed inside a x86-based PC (32-bit editions of Windows). Figure 3.6 shows the AP1000 board, and Figure 3.7 shows the location of components on the AP1000 board.

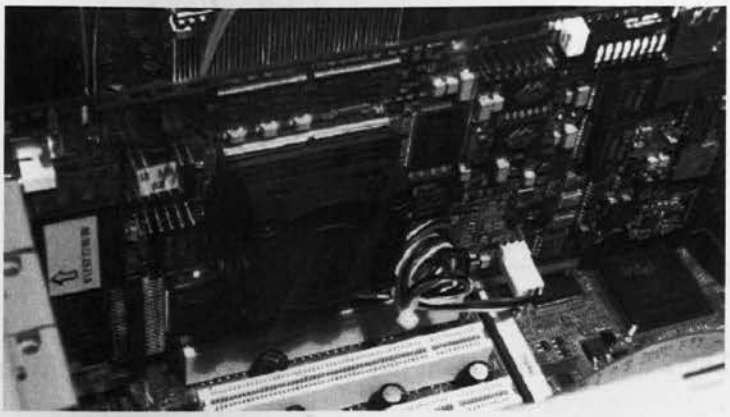

Figure 3.6: The Amirix AP1000 development board with Virtex-II Pro FPGA installed in the PC

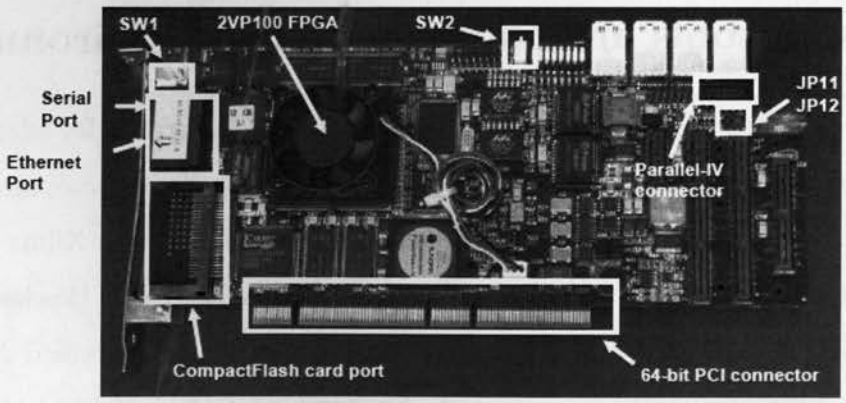

Figure 3.7: Location of components on the AP1000 board

3.4 Implementation of block modules of RLSL using System Generator

3.4.1 Designing block modules for RLSL system

This section describes the implementation of RLSL method for adaptive segmentation on simulink block level, using System generator from Xilinx company. One of the most important advantage of using System Generator is that it is combined with Simulink that makes the design procedure more friendly, straightforward and efficient, since one can always run simulation and see the results in various ways, i.e. observing waveforms in scope, or seeing data values in a text file.

As mentioned before, for adaptive segmentation that uses conventional RLSL algorithm, all the input data are necessary to run into the system for two times, hence requiring control subsystem to control the data in and out. Furthermore, all the samples are processed one by one and according to the algorithm (mathematical equations), adjacent samples are inter-related with each other and adjacent stages for one sample are inter-related as well, thereby requiring memory subsystem to store useful values of different parameters. Additionally, it is necessary to have a processing subsystem for calculating the parameters according to Equations (3.1) to (3.8).

To acquire high accuracy as C implementation does, double precision floating point type arithmetic has been used to implement the internal computation of RLSL algorithm. All the data for computing and storing are in 64 bits in total, with 1 sign bit, 11 bits for exponent and 52 bits for fraction. The remaining of this section depicts the models of RLSL system with more details.

Figure 3.8 shows the conceptual diagram of the whole RLSL system. It consists of four subsystems: *data control* subsystem, *memory* subsystem, *processing* subsystem and *final gammas* subsystem. The *data control* subsystem has data communication with all other subsystems. It collects the input data and controls the data flow into other subsystems sample by sample. It also generates several logic control signals for the rest of subsystems

to make the whole system working correctly. *memory* subsystem has data communication with the *processing* subsystem as for each sample, the values of parameters for each stage are required to store in memories for further samples or further stages. The *final gammas* subsystem stores the conversion factor values of the last stage for each sample. This *final gammas* subsystem could be omitted when used for real-time processing. When used for real-time adaptive segmentation, the threshold is supposed to be known to the user, hence, each new updated final γ_c value can be compared with the threshold immediately without storing.

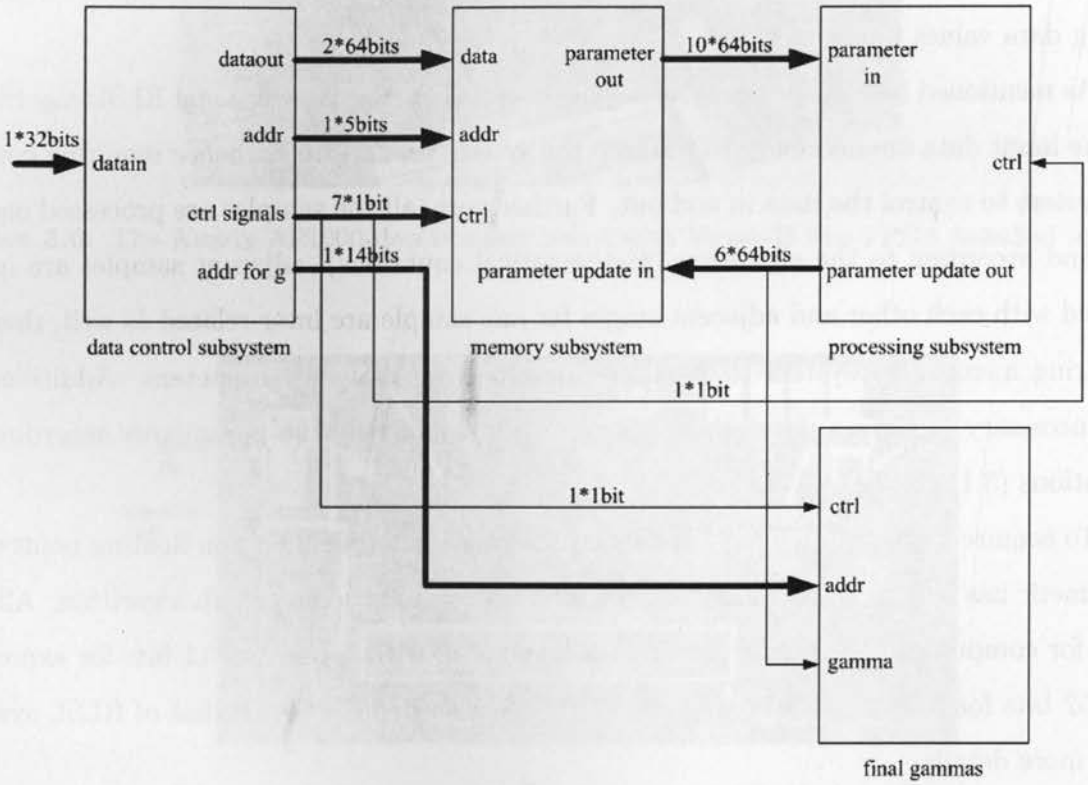

Figure 3.8: Block diagram of RLSL system

Figure 3.9 shows the real top-level models of RLSL system built in Simulink block level using System generator. Similar to the conceptual diagram of RLSL system, the real RLSL system built in Simulink block level also has four subsystems: *data control*, *memory*, *processing* and *for final gammas*.

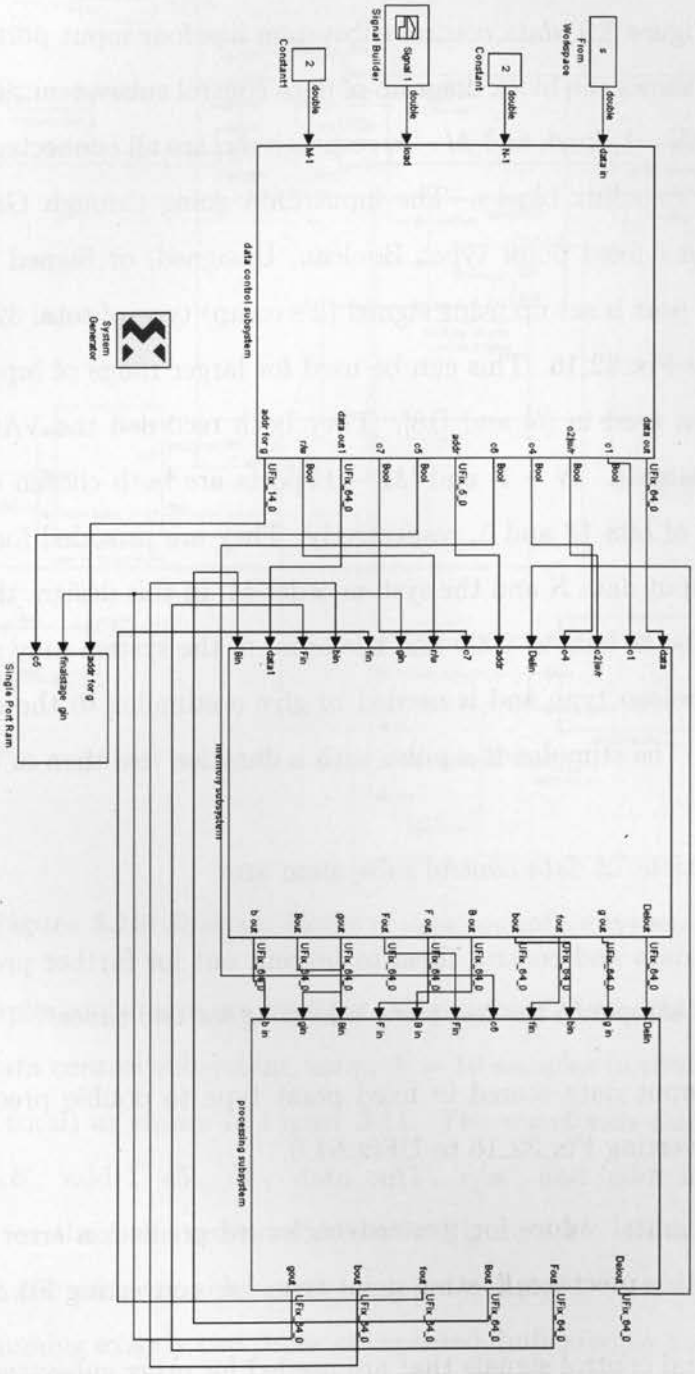

Figure 3.9: Top-level model of RLSL system

As shown in Figure 3.9, *data control* subsystem has four input ports and eleven output ports. Figure 3.10 shows the block diagram of data control subsystem. The four input ports, named as *data in*, $N-1$, *load*, and $M-1$, respectively, are all connected with outside world by using *Gateway In* xilinx blocks. The input data going through *Gateway In* has to be converted into Xilinx fixed point type: Boolean, Unsigned, or Signed (2's comp). In this design, the *data in* port is set up using signed (2's comp) type of total 32 bits with 16 bits in fraction, marked as Fix_32_16. This can be used for larger range of input signals compared with what has been used in [6] and [18]. They both recorded the VAG signal as integers using 12 bits per sample. '$N-1$' and '$M-1$' ports are both chosen using unsigned type with total number of bits 13 and 5, respectively. They are provided for users to determine the length of the input data N and the system order M. In this design, the input data length for processing can be as long as 5000 and the order of the system may reach up to 32. The 'load' port is of Boolean type and is needed to give a stimulus to the counters inside *data control subsystem*. The stimulus is a pulse with a duration less than or equal to one system clock cycle.

The main functions of *data control* subsystem are:

1. collect input data and control them to be sent out for further processing one by one from the first sample to the last sample looping for two times.

2. convert the input data stored in fixed point type to double precision floating point type, i.e. converting Fix_32_16 to UFix_64_0.

3. calculate the initial values for forward/backward prediction error power and convert them into double precision floating point type, i.e. converting Fix_63_32 to UFix_64_0.

4. generate several control signals that are needed for other subsystems.

Instead of describing each sub modules within *data control* subsystem one by one, it is more efficient to show the simulation waveforms of the output ports of *data control subsystem* and see their time-sequence relationships. To make it more clear and understandable, only

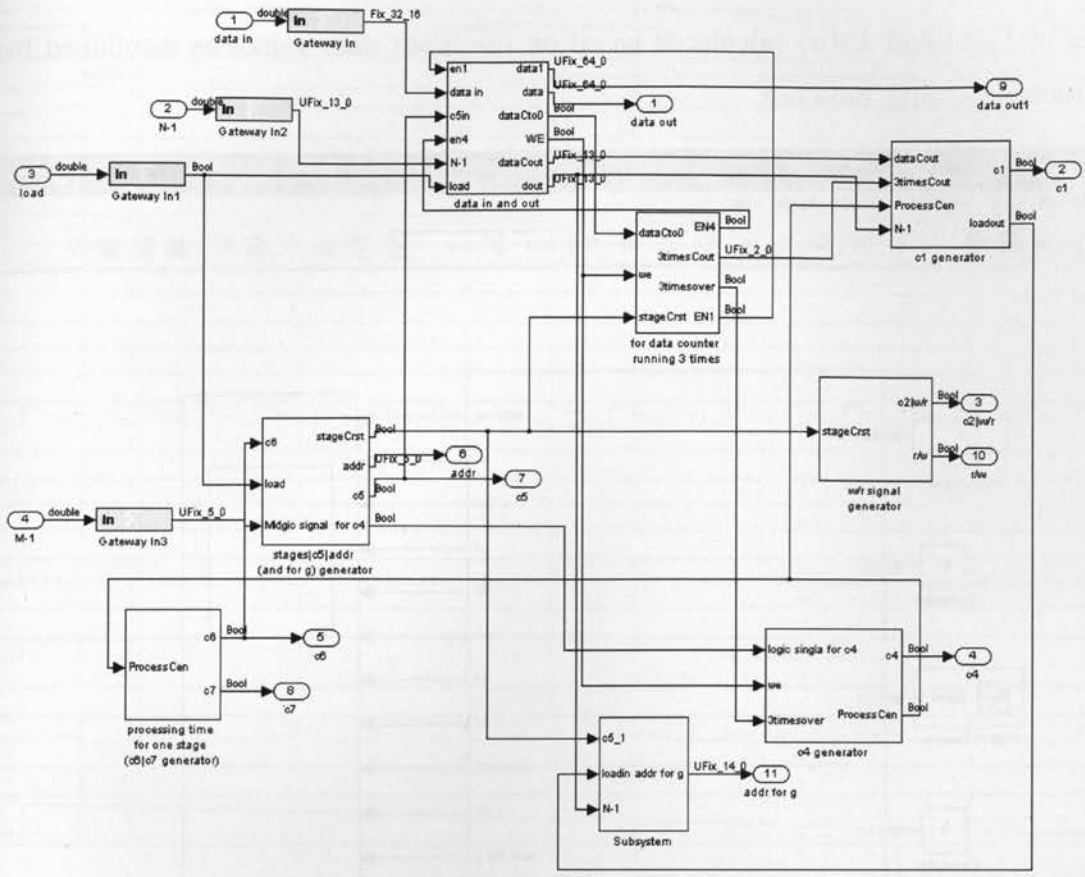

Figure 3.10: Diagram blocks of data control subsystem

small number of samples and stages are used for this purpose. Figure 3.12 shows the output port waveforms of *data control* subsystem, using $N = 10$ samples in total and system order $M = 5$ (5 stages in total) as shown in Figure 3.11. The waveforms came from 'data out', 'c1', 'c2|w/r', 'c4', 'c6', 'addr', 'c5', 'c7', 'data out1', 'r/w', and 'addr for g', from bottom to top, respectively, as the same order as shown in Figure 3.11. *Data out* sends out the stored input data to the *memory* subsystem sample by sample. It is clear that the input sample sequence is running exactly two times as expected, indicated by the third dash line in Figure 3.12. *c1, c2|w/r, c4, c7* are the control signals provided for *memory* subsystem. *c6* is generated for the *processing* subsystem. *c2|w/r, r/w, addr* generate the read/write control signals and addresses for the RAMs in *memory* subsystem. *data out1* sends out the initial

values of $F_0(n)$ and $B_0(n)$ calculated based on the input data values as mentioned before, simultaneously with *data out*.

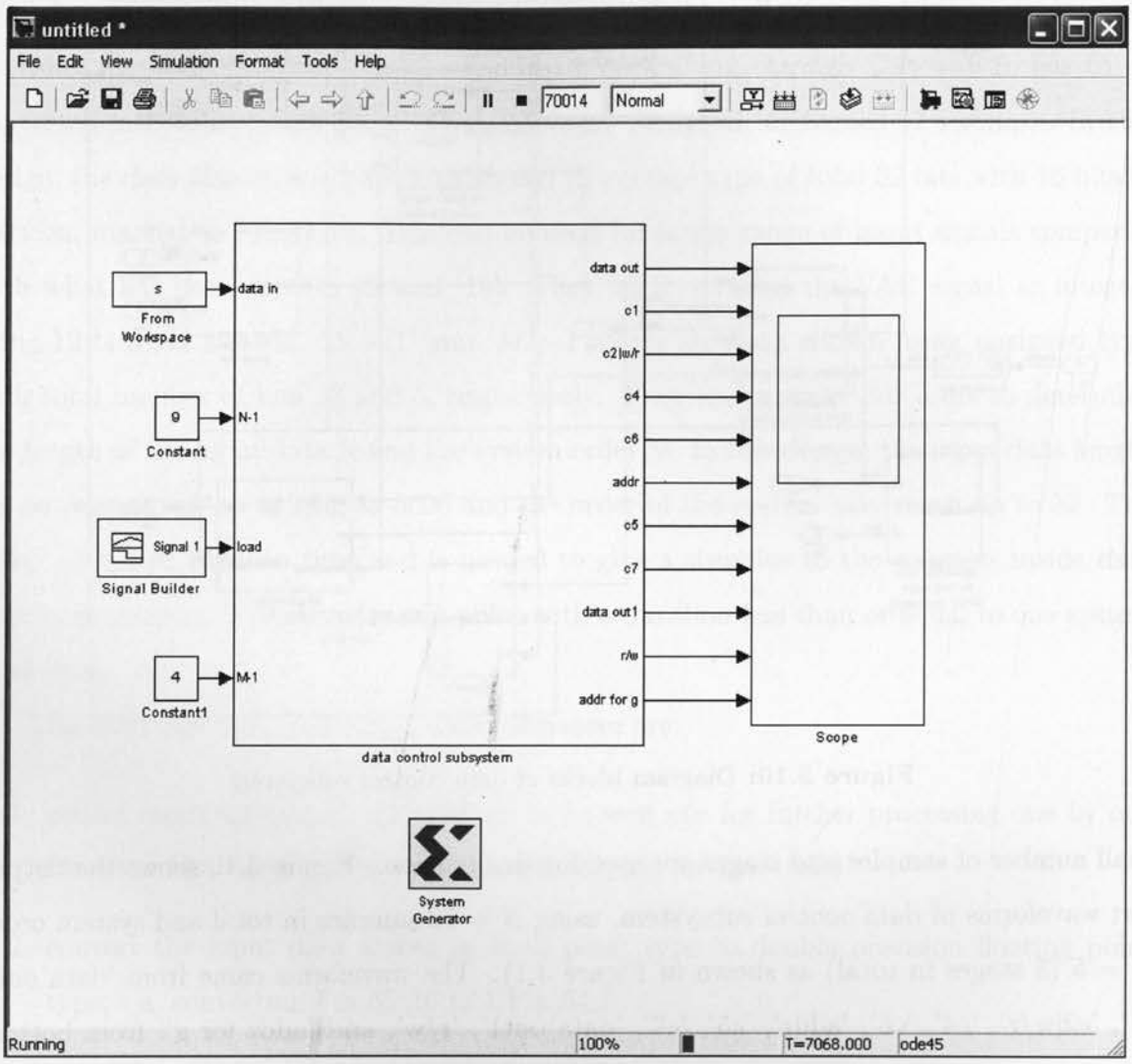

Figure 3.11: Simulation of data control subsystem

Figure 3.13 shows the block diagram of the *memory* subsystem. Since RLSL algorithm involves both time and order updates, different parameters need different memory unit structures to store their corresponding values. This could be explained as follows. By observing the right-hand sides of Equation (3.1)-(3.8), it is easy to find out that all the parameters can be divided into three categories as listed below. Those parameters that have the same

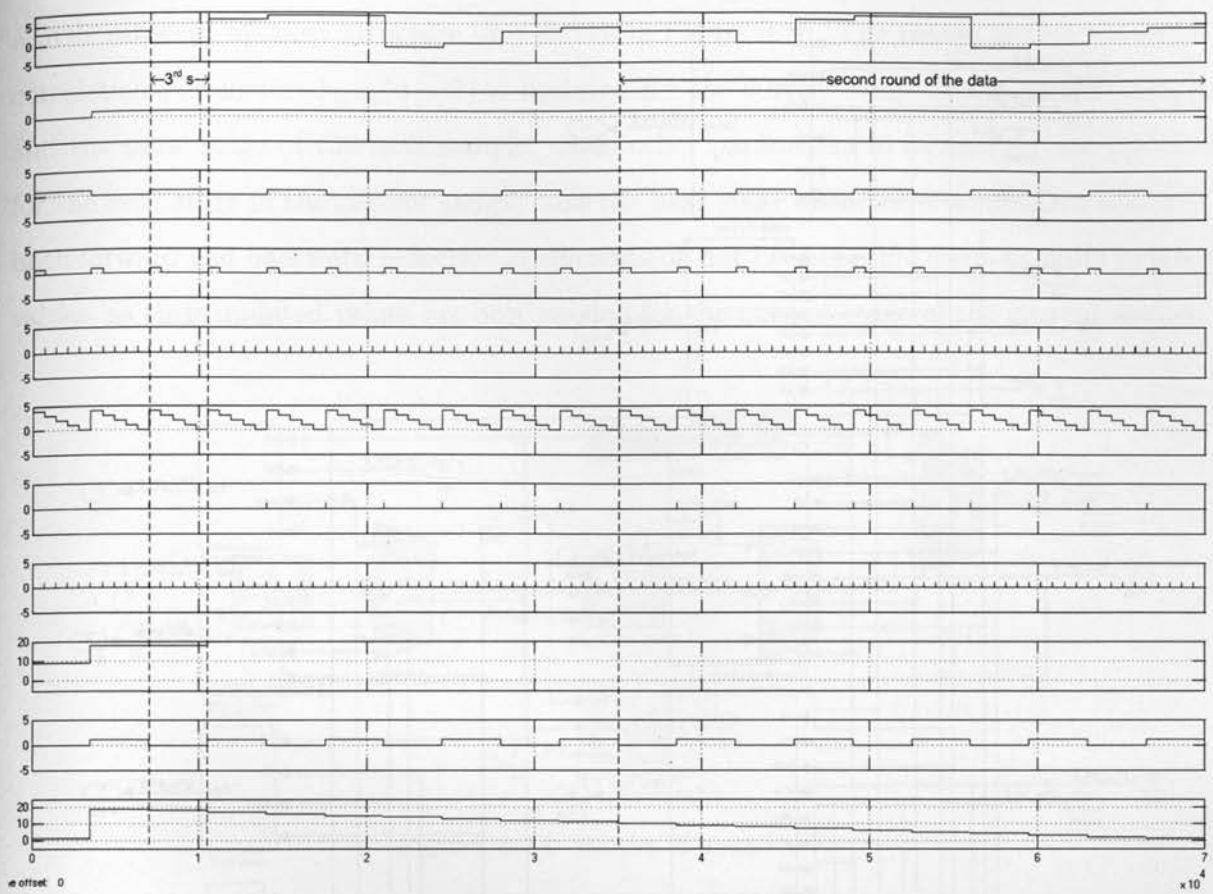

Figure 3.12: Waveforms of data control subsystem outputs

subscript and index are in the same category.

1. $b_{m-1}(n-1)$, $\gamma_{m-1,c}(n-1)$ in Equation (3.1); $F_{m-1}(n-1)$ in (3.3); $b_{m-1}(n-1)$ in (3.4) and (3.5); $B_{m-1}(n-1)$ in (3.7)

2. $f_{m-1}(n)$ in (3.1); $B_{m-1}(n)$ in (3.2); $f_{m-1}(n)$ in (3.4) and (3.5); $F_{m-1}(n)$ in (3.6); $\gamma_{m-1,c}(n)$, $b_{m-1}(n)$, $B_{m-1}(n)$ in (3.8)

3. $\Delta_{m-1}(n-1)$ in (3.1)

Parameters in the same category share the same block structures in *memory* subsystem. Figure 3.14 shows the memory unit for category 1, whereas Figure 3.15 shows the memory

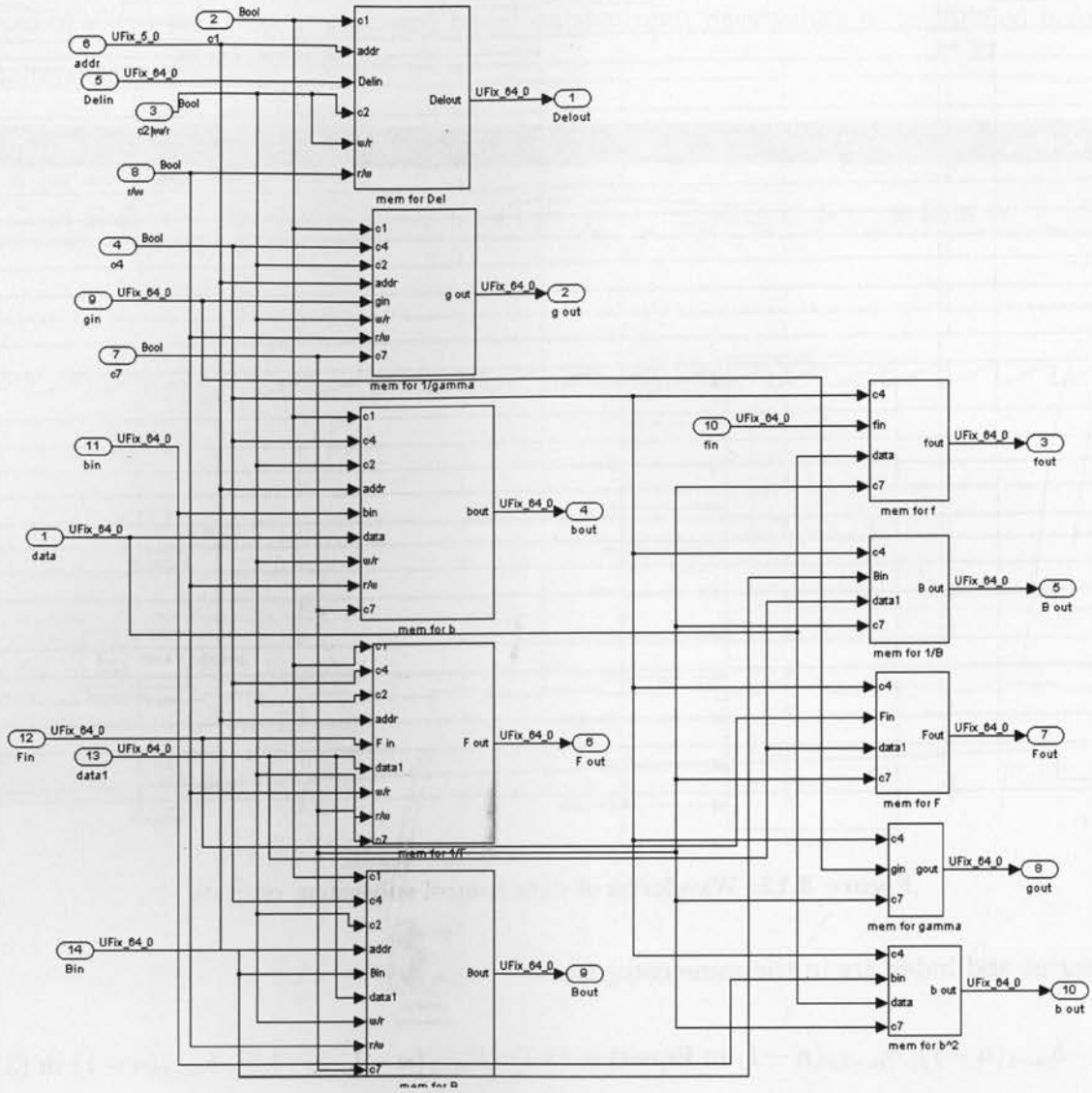

Figure 3.13: Block diagram of memory subsystem

unit for category 2 and Figure 3.16 shows the memory unit for category 3. It is obvious that the memory unit for category 2 is simpler than that of category 1. This is because for the current sample, those parameters in category 2 just require the values of previous stage of the same sample, not requiring the values of previous stage of the previous sample as what is required in category 1. One should also notice that, although $\Delta_{m-1}(n-1)$ in Equation (3.1) has the same subscript and index as those parameters in category 1, it has

a little different memory structure unit shown in Figure 3.16. The reason is that the cross-correlation parameter $\Delta_{m-1}(n-1)$ is updated for the current stage of the current sample and the same stage of the next sample, while other parameters in category 1 are updated for the next stage of the current sample and the next stage of the next sample. In addition, both forward and backward reflection coefficients do not need specific memory units to store values, as their updated values are only needed for the current stage of the current sample.

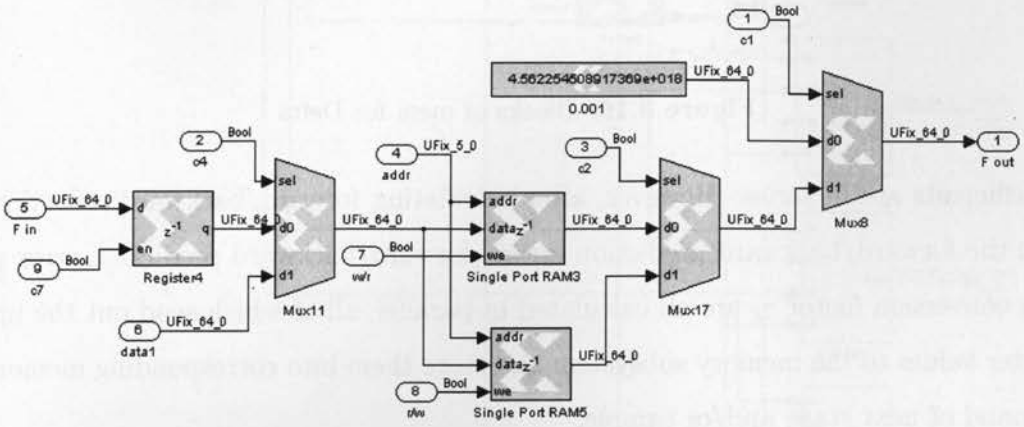

Figure 3.14: Blocks of mem for F

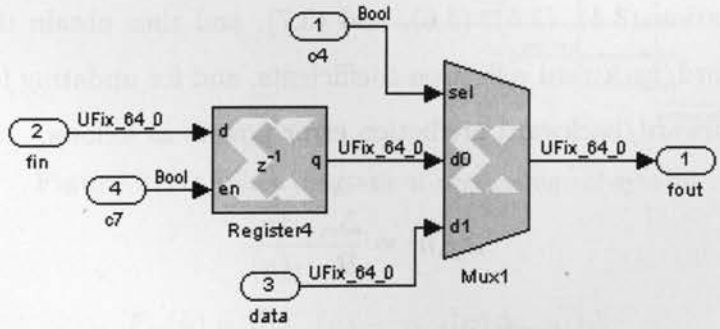

Figure 3.15: Blocks of mem for f

Figure 3.17 shows the block diagram of processing subsystem. It consists of eight subsystems calculating and updating the eight parameters as shown in the figure. The implemented system achieves indirectly RLSL algorithm, since it calculates cross-correlation Δ_{m-1} first and then updates reflection coefficients. Hence, the modules of calculating Δ_{m-1} and reflec-

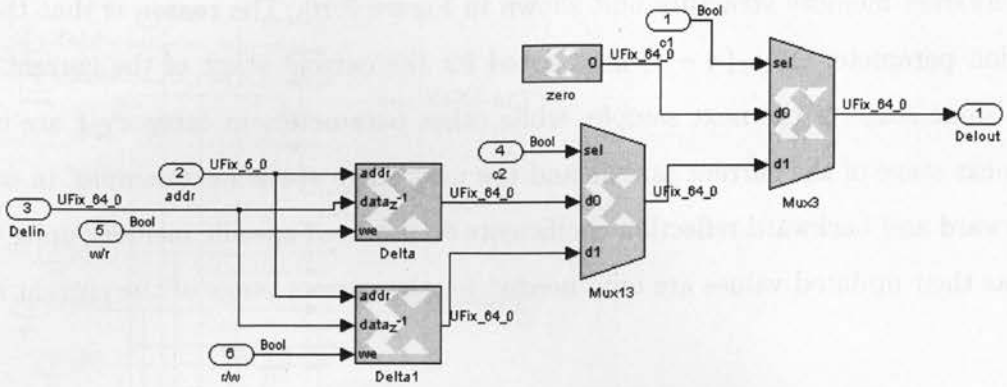

Figure 3.16: Blocks of mem for Delta

tion coefficients are in series. However, after calculating forward/backward reflection coefficients, the forward/backward prediction errors, forward/backward prediction error powers and the conversion factor γ_c are all calculated in parallel, all of which send out the updated parameter values to the memory subsystem and store them into corresponding memories for the demand of next stage and/or sample.

Recalling the mathematical equations of RLSL algorithm described earlier in this chapter, to save floating point operation blocks, substitute the sign of minus in Equation (3.2) and (3.3) into Equation (3.4), (3.5), (3.6), and (3.7), and thus obtain the new equations for calculating forward/backward reflection coefficients, and for updating forward/backward prediction errors, forward/backward prediction error powers as follows.

$$\gamma_{m,f}(n) = \frac{\Delta_{m-1}(n)}{B_{m-1}(n)} \tag{3.13}$$

$$\gamma_{m,b}(n) = \frac{\Delta_{m-1}(n)}{F_{m-1}(n-1)} \tag{3.14}$$

$$f_m(n) = f_{m-1}(n) - \gamma_{m,f}(n) b_{m-1}(n-1) \tag{3.15}$$

$$b_m(n) = b_{m-1}(n-1) - \gamma_{m,b}(n) f_{m-1}(n) \tag{3.16}$$

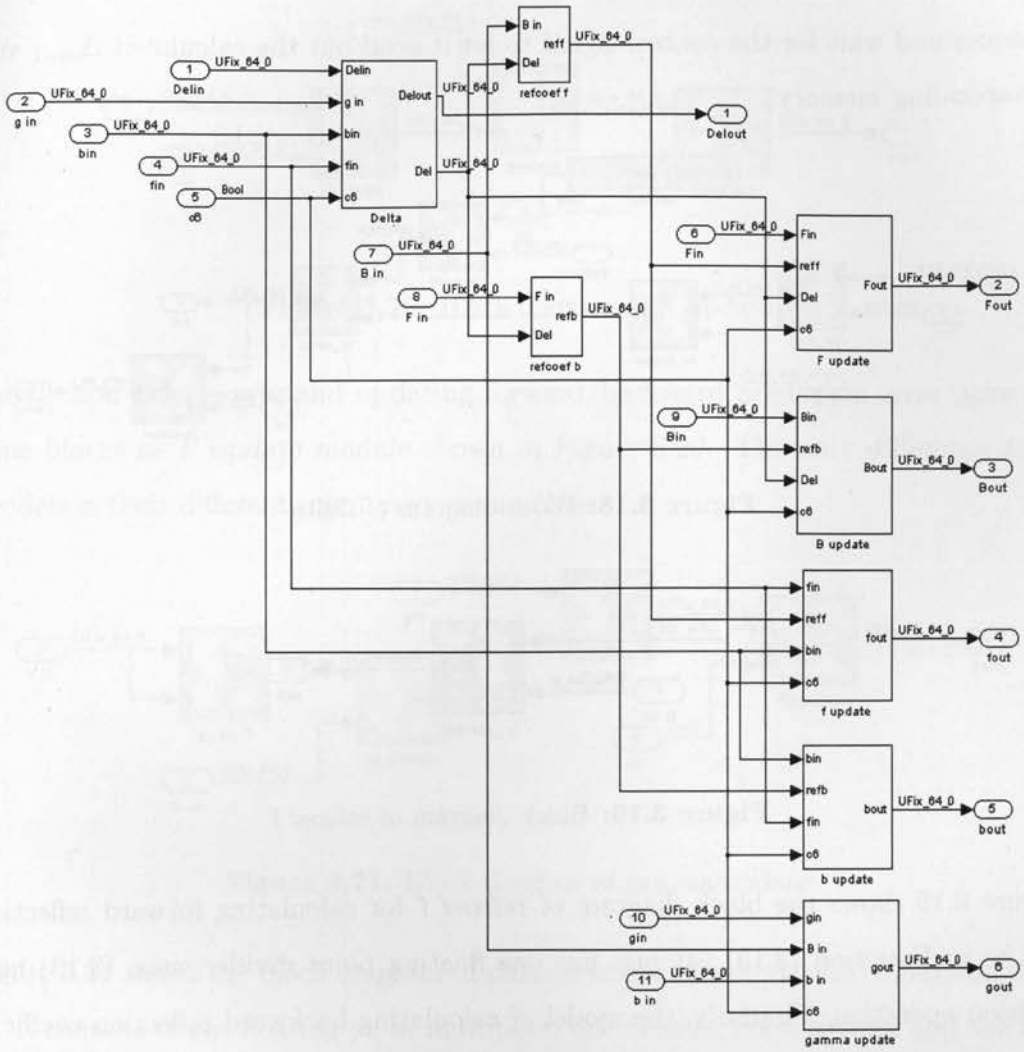

Figure 3.17: Block diagram of processing subsystem

$$F_m(n) = F_{m-1}(n) - \gamma_{m,f}(n)\Delta_{m-1}(n) \tag{3.17}$$

$$B_m(n) = B_{m-1}(n-1) - \gamma_{m,b}(n)\Delta_{m-1}(n) \tag{3.18}$$

Figure 3.18 shows the block diagram of *Delta* for calculating Δ_{m-1} values. It has two floating point multipliers, one floating point divider and one floating point adder as the same number processing units in Equation 3.1. Another register is used to store the calculated

Δ_{m-1} values and wait for the control signal to let it send out the calculated Δ_{m-1} value to its corresponding memory.

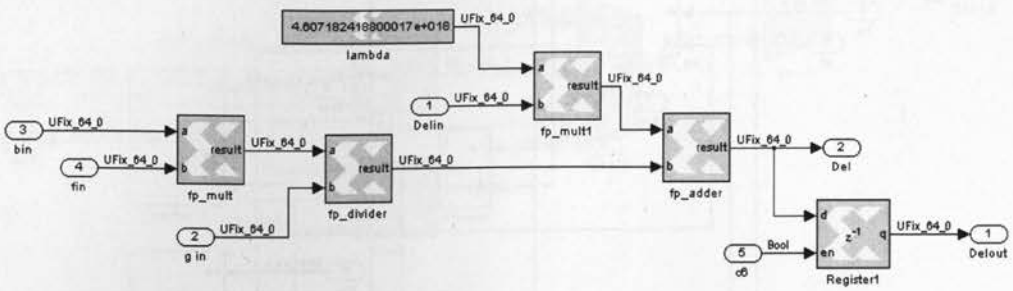

Figure 3.18: Block diagram of Delta

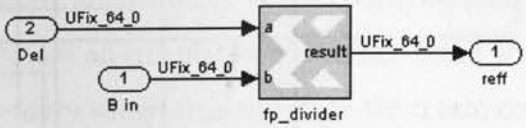

Figure 3.19: Block diagram of refcoef f

Figure 3.19 shows the block diagram of *refcoef f* for calculating forward reflection coefficient as in Equation (3.13). It just has one floating point divider since (3.13) has just one division operation. Similarly, the model of calculating backward reflection coefficient is exactly the same as computing forward reflection coefficient: it also has only one floating point divider. The only difference for these two models is their different input port connections. For forward reflection model, the denominator of the divider is connected with *B in*, while for backward reflection model, it is connected with *F in*. Apparently, the output ports are different: one is *reff* and the other is *refb*, representing forward reflection coefficient and backward reflection coefficient respectively.

Figure 3.20 shows the block diagram *F update* for updating forward prediction error power. It consists of one floating point multiplier and one floating point subtracter. Similarly, one extra register is used to store the updated values. It holds the same function as Equation (3.17) does. According to Equation (3.15)-(3.18), the modules of updating back-

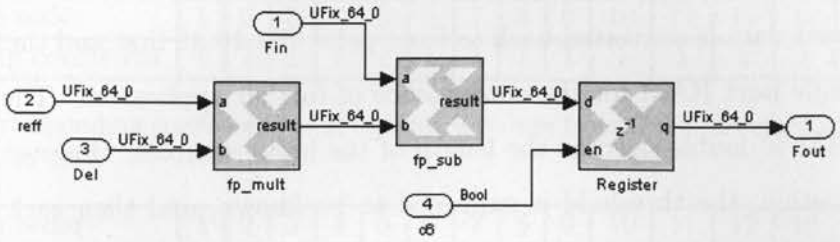

Figure 3.20: Block diagram of F update

ward prediction error power and updating forward/backward prediction error have exactly the same blocks as *F update* module shown in Figure 3.20. The only difference for these four models is their different input port connections.

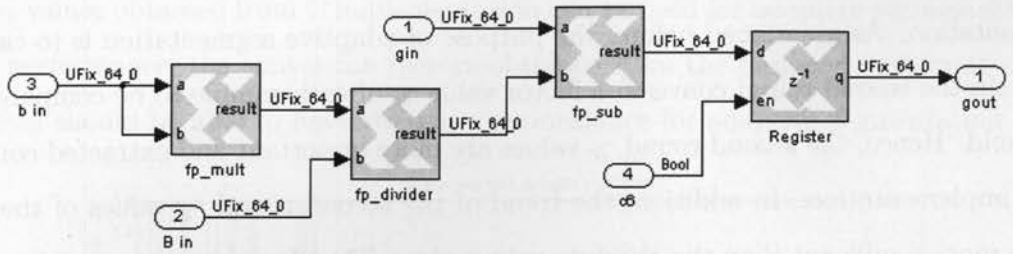

Figure 3.21: Block diagram of gamma update

Figure 3.21 shows the block diagram of *gamma update* for updating conversion factor γ_c values. It consists of one floating point multiplier, one floating point divider and one floating point subtracter as the same operations in Equation (3.8). Also, another register is used to store the updated γ_c values and wait for being sent out to its corresponding memory.

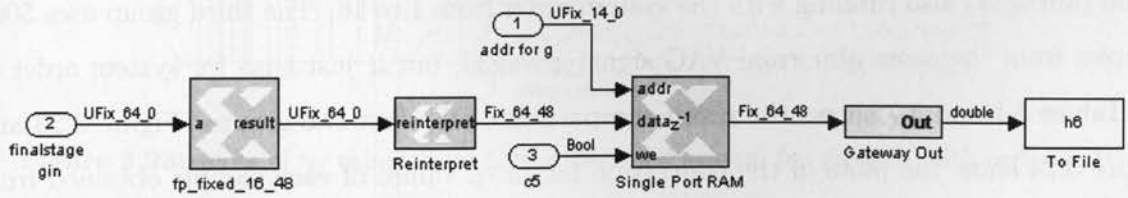

Figure 3.22: Block diagram of final gammas subsystem

Figure 3.22 shows the block diagram of *for final gammas* subsystem. In this subsystem,

the floating point data is converted back to fixed point Fix_16_48 first and then sent out and stored in a single port RAM for the convenience of further application. The depth of this single port RAM is double-sized as the length of the input samples. However, for real-time purpose application, the threshold is supposed to be known, and then each final γ_c value could be compared with the threshold instantaneously, and there is no need to have a RAM for storing all the γ_c values for all input data.

3.4.2 Simulation results and Comparison with high-level language

To test the functionality of the RLSL system built in the previous section, we compare the simulation results obtained from simulink block level with the results obtained from the C implementation. As mentioned before, the purpose of adaptive segmentation is to calculate and obtain the second round conversion factor value γ_c of each sample to be compared with a threshold. Hence, the second round γ_c values are more important and extracted compared with C implementation. In addition, the trend of the second round γ_c values of the target signal is more significant than the absolute values of γ_c. Therefore, the corresponding coefficient of the second round conversion factor values obtained from the C implementation and the simulink block implementation have been calculated to evaluate the performance of the designed system at simulink block level. The input data is the same for both implementations, taken from a real VAG signal. The comparison is based on three groups of simulations. The first group uses 84 samples from a normal VAG signal (novag27) running with the system order from 1 to 16 and the second one uses another 84 samples from an abnormal VAG signal (abvag34) also running with the system order from 1 to 16. The third group uses 5000 samples from the same abnormal VAG signal abvag34, but it just runs for system order of 16. Tables 3.1 and 3.2 shows the result comparison of the first two groups. Figure 3.23 and Figure 3.24 show the plots of the conversion factor γ_c values of each sample obtained from C implementation and the simulink block implementation running with system order of 16 in the first two groups. Figure 3.25 shows the γ_c values of each sample in group 3.

In conclusion, according to Table 3.1 and 3.2, the corresponding coefficients for the second

system order	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
corresponding coefficient	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3.1: Corresponding coefficients of 2^{nd} round γ_c values from C and designed system for group 1; order from 1 to 16

system order	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
corresponding coefficient	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3.2: Corresponding coefficients of 2^{nd} round γ_c values from C and designed system for group 2; order from 1 to 16

round γ_c values obtained from both C implementation and simulink level implementation are 100% correlated when running different system orders: from 1 to 16. Since the second round γ_c values obtained from C implementation can be used for adaptive segmentation with a good performance, the conversion factors obtained from the designed system at simulink block level should be able to have the same performance for adaptive segmentation as well.

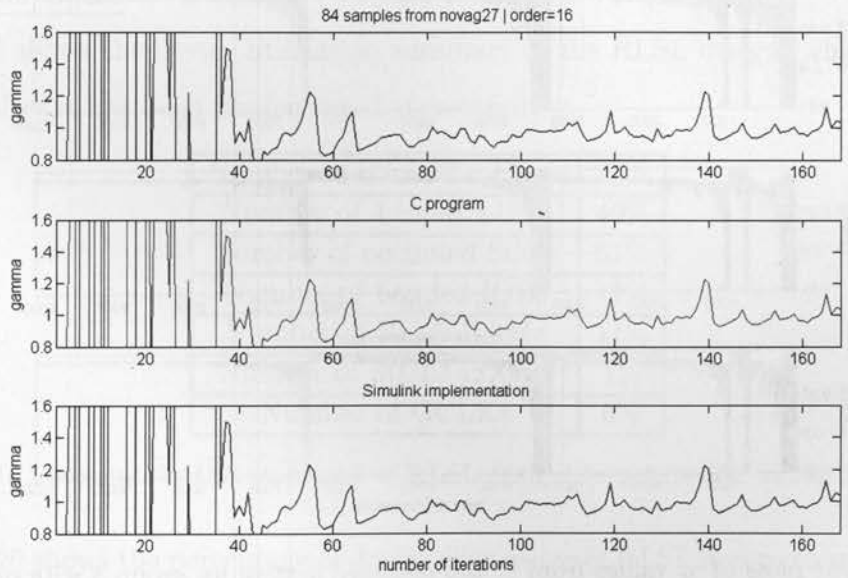

Figure 3.23: plots of γ_c values from C and designed system for group 1 with order of 16

Moreover, according to Figures 3.23, 3.24, and 3.25, the 2^{nd} round γ_c values (from 85 to 168 in x axis of the first two and from 5001 to 10000 in x axis of the third one) obtained from C implementation and the designed system at simulink block level match very well. Thus,

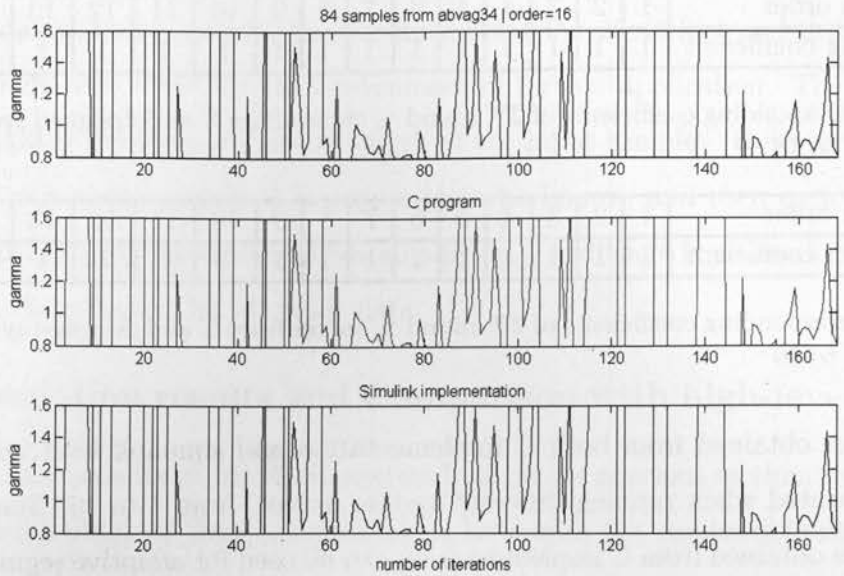

Figure 3.24: plots of γ_c values from C and designed system for group 2 with order of 16

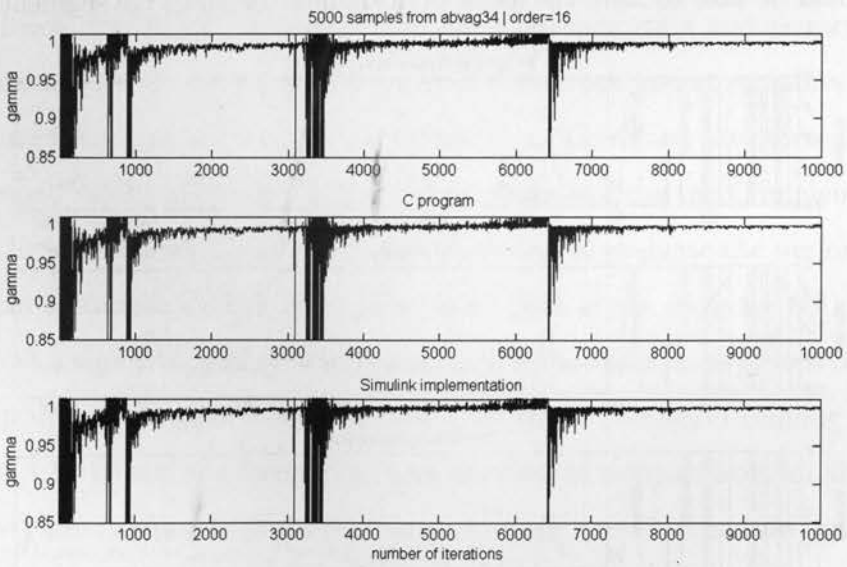

Figure 3.25: plots of γ_c values from C and designed system for group 3 with order of 16

it is concluded that the RLSL system implemented on simulink block level works correctly and satisfactory as C implementation does.

3.5 Implementation of RLSL on FPGA and result comparison

After building up the models in Simulink block level, the System generator tool can generate the synthesizable VHDL or verilog code for the design, which can be imported into Xilinx ISE for further design implementation. Xilinx ISE tool can do synthesis, implementation (translate, map, place & route) and device programming for the target design. Behavioral simulation is supported by Mentor Graphics ModelSim. After downloading the design to FPGA successfully, the verification can be achieved by using Xilinx ChipScope Pro tool.

For the design of RLSL, VHDL code is chosen for system generator. The clock frequency is selected as 20MHz. The maximum data length for processing according to the physical design is 5000 samples. The maximum system order for processing according to the physical design is 32. The target FPGA device is xc2vp100-6-ff1704.

Area Requirements

Table 3.3 shows the device utilization summary of the RLSL design, which is provided by ISE tool after it has been implemented successfully.

Number of Slice Flip Flops	54%
Number of 4 input LUTs	40%
Number of occupied Slices	61%
Number of bonded IOBs	11%
Number of Block RAMs	11%
Number of MULT18X18s	1%
Number of GCLKs	6%

Table 3.3: Device utilization summary of RLSL design; target device: xc2vp100-6-ff1704

Figure 3.26 shows the percentage of device slice usage of RLSL design running under different system orders, including testbenches for each implementation. The system is running under the order of 16, 20, 24, 28, 32, respectively, with a total of 4096 input data samples.

According to Table 3.3 and Figure 3.26, it is clear that the RLSL FPGA implementation occupies around 61% source of the target device and more importantly, when running different orders (from 16 to 32), the area of the occupied slices does not change much (in

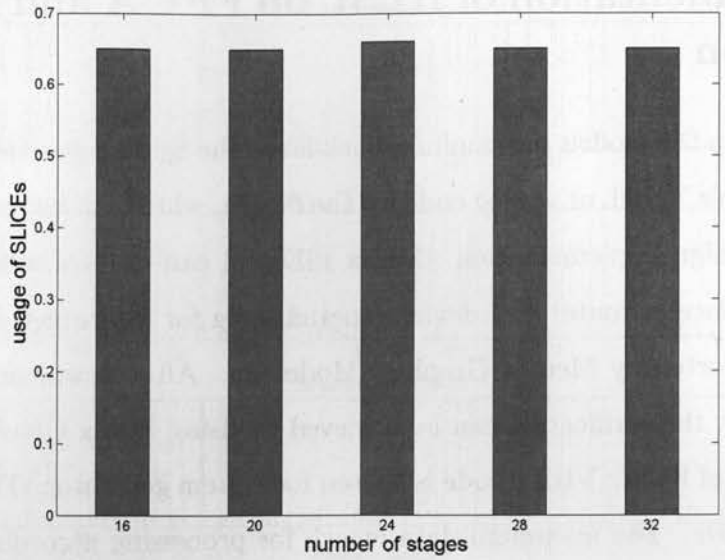

Figure 3.26: Comparison of the device slice usage of RLSL algorithm with different orders, including testbench for each implementation

that the testbench also occupies some source, hence it certainly has some difference, but not much). This is what it is expected, as the whole RLSL design holds the same architecture for each stage of each sample.

Clock and Timing

The system is a single rate system, one *clk* for all. The clock frequency is chosen as 20 MHz (period = 50 ns). For one sample running for one stage, the processing time is setup as 200 clock cycles. The reason is that in the critical path for calculating those parameters, there are two division in series, and each floating point divider has maximum latency of 56. All other floating point adder/subtracter and multipliers have their maximum latencies. Hence, to make the circuits have enough time to obtain stable and correct results, 200 latency is determined for the processing time of one stage, controlled by a counter in *data control* subsystem. Then for processing 4096 samples of 16 stages to get the second round γ values for each sample, the whole processing time is:

$$clock\ cycles = 200 \times 16 \times 4096 \times 2 = 2.62144 \times 10^7 \tag{3.19}$$

The FPGA clk period is 50 ns, then the real time for processing 4096 samples with 16 stages is:

$$Time = 50 \times 2.62144 \times 10^7 ns = 1.31072 \times 10^9 ns = 1.31072 s \tag{3.20}$$

Limitations

As the RLSL algorithm is related with both time and order, the performance is dependent upon these two factors. The length of the data and the chosen order influence the real processing time. Take the implementation of running 4096 samples and choosing order of 16 as an example. According to the static timing report, the minimum period is 34.981 ns (maximum frequency 28.587 MHz). Hence, for the whole 4096 samples for only one stage, the processing time is

$$Time = 34.981 \times 200 \times 4096 \times 2(ns) = 0.0573128704 \times 10^9 ns = 0.0573128704(s) \tag{3.21}$$

Generally, the sampling rate of 2 kHz is typical for many biomedical signals' acquisition [1] [6] [18]. Hence, to collect 4096 samples, it will take 2.048 seconds. Therefore, according to Equation (3.21), the maximum order (the limitation) for processing 4096 samples is

$$order = 2.048 \div 0.0573128704 = 35.733681 \approx 36 \tag{3.22}$$

Performance

As mentioned in previous chapters, the main purpose of this RLSL design is to calculate the 2^{nd} round γ_c values for each sample. Hence, the performance can be evaluated by comparing the γ_c values with those obtained from high-level language, i.e. C implementation and/or simulink block level implementation, to see whether the system works correctly or not and whether the γ_c values are accurate enough or not.

After downloading the design to FPGA successfully, using chipscope one can see the output port value captured by chipscope. Figure 3.27 shows one test example of processing 4096 samples running for 16 stages. The last γ_c value of the second round is captured and represented by the signed fixed point version Fix_16_48 in Hexadecimal, which is equal to

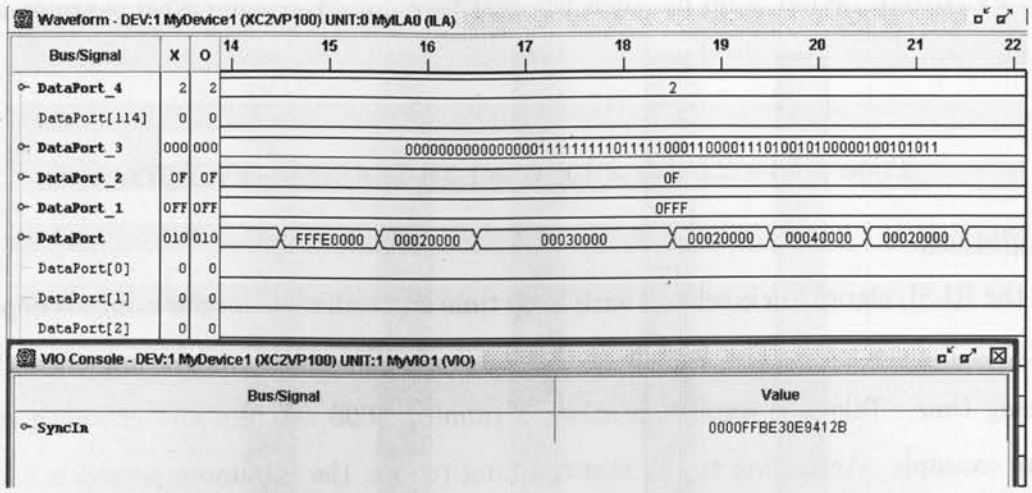

Figure 3.27: Monitoring signal using Chipscope

0.9985 in decimal (calculated by Matlab). Taking the same input data to the C implementation running 16 stages, the last γ_c value obtained is 0.9985. Two additional examples have also been implemented to test the RLSL design. One takes 4095 samples running 16 stages to compare the last γ_c value of the second round and the other takes 4094 samples running 16 stages to compare the last γ_c value of the second round. The result comparison of the three examples mentioned above is shown in Table 3.4. It is clear that the γ_c value obtained from the FPGA implementation has very high accuracy with a mean correlation of 99.93%, which will certainly meet the requirement in real-time adaptive segmentation of biomedical signals.

Example No.	1	2	3
Parameters	N=4096; M=16	N=4095; M=16	N=4094; M=16
$\gamma_c(2N)$ from C	0.9985	1.001	0.9977
$\gamma_c(2N)$ from FPGA	0.9985	1.003	0.9977
correlation	100%	99.8%	100%

Table 3.4: Comparison of the last γ_c value of the 2^{nd} round obtained from C and FPGA implementations

In real applications, the second round γ_c values for all the input data will be compared with a threshold instantaneously. The procedure should be like this: whenever the first 2^{nd}

round γ_c value is available, it is then compared with the threshold value immediately. If the γ_c value is below the threshold, that means a boundary should be set up there. Meanwhile, due to [19] [20], one segment for VAG signal should contain at least 120 samples. The number of 120 is defined as *the minimum desired segment length* suitable for further modeling. If the distance between two boundaries is less than 120 samples, then, the second boundary of the two should be ignored. Based on this purpose of obtaining positions of boundaries, some modifications have been done to the design in simulink block level as explained below.

Figure 3.28 shows the renewed RLSL system. It has one more subsystem: *boundaries subsystem*, which gives out the positions of the boundaries among the input data. Figure 3.29 shows the block diagram of the *boundaries subsystem*. It sends out the numbers that represent the positions of boundaries, as long as the distance between adjacent boundaries is equal or greater than 120. Figure 3.30 shows the block diagram that compares the 2^{nd} round γ_c values with a user-defined threshold value and sends out the numbers representing the boundary positions. Since γ_c values are available as double precision floating point, a fixed point to floating point converter is used to convert the threshold value from Fix_64_32 type to double precision floating point type.

After building up the new RLSL system in simulink block level, re-generating the VHDL code, synthesizing, translating, mapping, placing & routing, and downloading design to FPGA, the chipscope is still used to see the output of the system and compare the result with the simulation results obtained from C implementation and simulink block level implementation. The new RLSL system has been tested for three groups: one is taking 500 samples from the real knee signal novag27 with order of 5 and threshold value of 0.9985; the second one is taking the same 500 samples and the same threshold value 0.9985 but with order of 16; the third one is taking the synthesized signal of 600 samples (used before in this chapter for verification of adaptive segmentation) and threshold value of 0.95. The minimum desired segment length is using 120 for all of the three. The results are shown in Tables 3.5, 3.6 and 3.7.

For FPGA implementation, only the last boundary can be captured and displayed by

C	122	244	364	**484**
Simulink	120	240	363	**483**
FPGA	—	—	—	**482**

Table 3.5: Group 1: Boundaries obtained from C implementation, simulink block level and FPGA implementation; $N = 500$, $M = 5$, $threshold = 0.9985$, $min = 120$

C	120	240	360	**480**
Simulink	120	240	360	**480**
FPGA	—	—	—	**480**

Table 3.6: Group 2: Boundaries obtained from C implementation, simulink block level and FPGA implementation; $N = 500$, $M = 16$, $threshold = 0.9985$, $min = 120$

C	203	**402**
Simulink	202	**401**
FPGA	—	**401**

Table 3.7: Group 3: Boundaries obtained from C implementation, simulink block level and FPGA implementation; $N = 600$, $M = 5$, $threshold = 0.95$, $min = 120$

chipscope. However, according to Tables 3.5, 3.6 and 3.7, no matter using VAG signal or synthesized signal, no matter using low order or high order, the last boundary obtained by FPGA implementation is very similar to the C implementation and simulink block level implementation, which means the FPGA implementation of RLSL method does work correctly and provide very good performance of adaptive signal segmentation. Table 3.8 shows the device utilization summary of the new RLSL design with boundaries subsystem.

Number of Slice Flip Flops	55%
Number of 4 input LUTs	41%
Number of occupied Slices	71%
Number of bonded IOBs	11%
Number of Block RAMs	12%
Number of MULT18X18s	1%
Number of GCLKs	6%

Table 3.8: Device utilization summary of RLSL design with boundaries subsystem; target device: xc2vp100-6-ff1704

Summary

The RLSL method for adaptive segmentation has been implemented for the first time onto FPGA with programmable functions and based on double precision floating point type that can provide high accuracy. After the target nonstationary biomedical signal has been adaptively segmented, AR modeling method can be used for each stationary segment for further application, i.e. using Burg-lattice algorithm to calculate AR parameters of the segment. In next Chapter, the Burg-lattice algorithm and its hardware implementation will be provided in details.

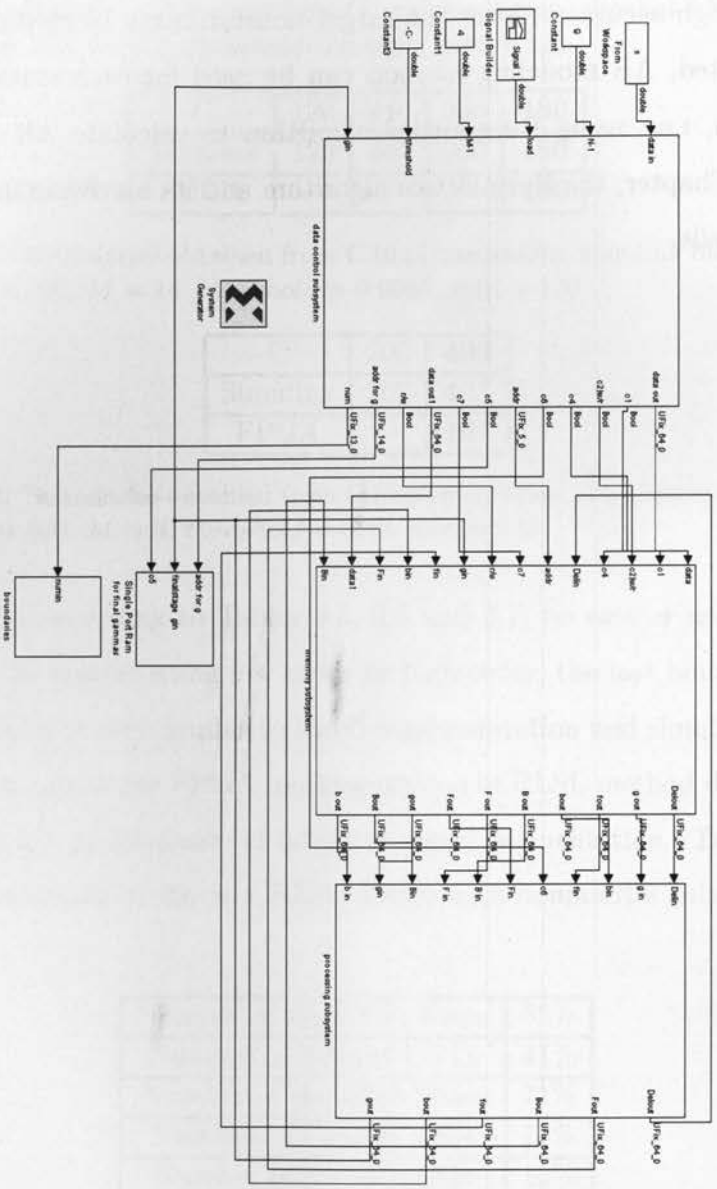

Figure 3.28: Diagram of RLSL with boundaries subsystem

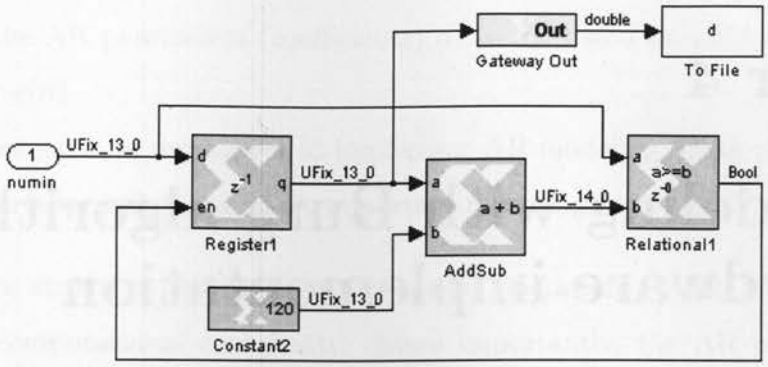

Figure 3.29: Block diagram of boundaries subsystem

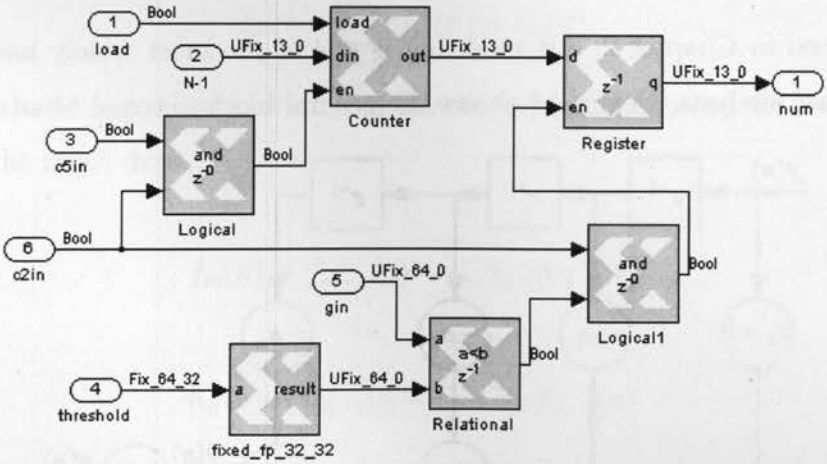

Figure 3.30: Block diagram of comparing 2^{nd} γ_c values with threshold value

Chapter 4

AR modeling with Burg algorithm and hardware implementation

In this chapter, the theory of the Burg algorithm will be presented first mathematically, and then designing the algorithm in simulink block level will be described. The simulation and result comparisons with high-level languages will be followed. The FPGA implementation of Burg algorithm and conclusions will be presented at last.

4.1 The theory of Burg algorithm

As mentioned in Chapter 2, AR modeling is one of the most widely used methods in biomedical signal analysis. Figure 4.1 shows the general block diagram of an AR model.

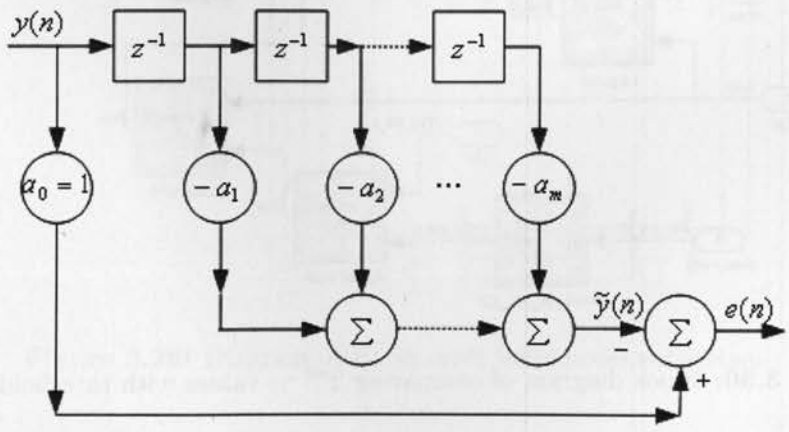

Figure 4.1: Signal-flow diagram of AR model

$y(n)$ is the current sample of a stationary input signal with a certain length. $\widetilde{y}(n)$ is the approximate predicted value of the current sample and $e(n)$ is the forward prediction error. a_1 to a_m are the parameters of the AR model. Generally, the purpose of AR modeling is to compute the AR parameters (coefficients) a_1 to a_m based on minimizing the forward prediction error $e(n)$.

Among those existing techniques to implement AR modeling, Burg algorithm is one of the most popular approaches due to its important advantages:

- uses lattice structure performing the recursive operations, which leads to modularity and less computational complexity. More importantly, the AR coefficients can be computed for any model order by simply adding one or more lattice stages without affecting the earlier computations for lower orders.
- guarantees a minimum-phase design for the lattice predictor.

The Burg algorithm is based on minimizing the sum of the squared forward and backward prediction errors. The cost function is given as [1]

$$\xi_m = \sum_{n=m+1}^{N} f_m^2(n) + b_m^2(n) \tag{4.1}$$

where $f_m(n)$ and $b_m(n)$ are forward and backward prediction error for order of m. N is the length of the input data.

$$f_m(n) = f_{m-1}(n) - \gamma_m b_{m-1}(n-1) \tag{4.2}$$

$$b_m(n) = b_{m-1}(n-1) - \gamma_m f_{m-1}(n) \tag{4.3}$$

Equation (4.2) and (4.3) are the recursion equations for forward and backward prediction error updates. They use the lattice structure for computing forward/backward prediction errors as shown in Figure 4.2, similarly to the RLSL algorithm described in Chapter 3. The only difference is that for Burg algorithm, there is only one reflection coefficient ($\gamma_m(n)$)

for forward/backward prediction error updates. However, for RLSL algorithm, there are two different reflection coefficients (forward/backward reflection coefficients: $\gamma_{m,f}(n)$ and $\gamma_{m,b}(n)$) for forward/backward prediction error updates, respectively.

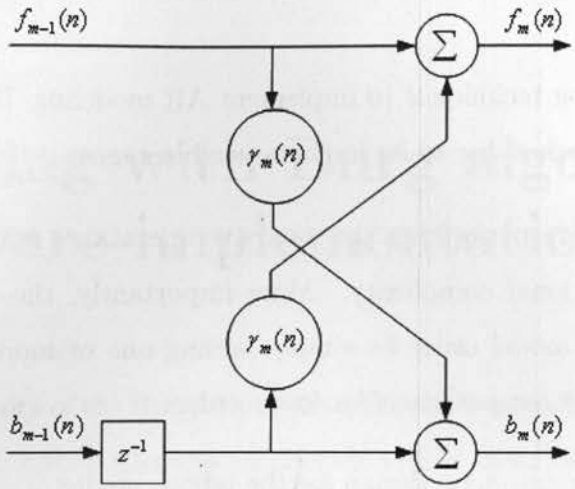

Figure 4.2: lattice structure that performs the recursion equations for one stage of Burg algorithm

The reflection coefficient of Burg algorithm is calculated as

$$\gamma_m = 2\frac{\sum_{n=m}^{N-1} f_{m-1}(n)b_{m-1}(n-1)}{\sum_{n=m}^{N-1}\left[f_{m-1}^2(n) + b_{m-1}^2(n-1)\right]} \tag{4.4}$$

It is easy to realize that the absolute value of reflection coefficient $|\gamma_m|$ is always lesser than unity. This property can guarantee minimum-phase for the lattice predictor [1].

The AR model parameters can be computed from the reflection coefficient by using the relationship in Equation (4.5)

$$a_{m,k} = a_{m-1,k} - \gamma_m a_{m-1,m-k} \tag{4.5}$$

In addition, it is notified here that in many literatures (e.g. [1]), the reflection coefficient is represented using the negative value of what is obtained in Equation (4.4), which means that the reflection coefficient value in Equation (4.4) will be multiplied by -1. Thereby, correspondingly, in Equations (4.2), (4.3) and (4.5), the *subtraction* will become *addition*.

In this thesis, the design of Burg algorithm is all based on Equations (4.2), (4.3), (4.4) and (4.5).

The procedure of calculating AR coefficients $a_{m,k}$ can be listed below.

Initialization

$f_0(n) = b_0(n) = x(n), n = 0, 1, \ldots, N-1$

$a_0 = 1$

1^{st} *iteration:*

$$\gamma_1 = 2\frac{\sum_{n=1}^{N-1} x(n)x(n-1)}{\sum_{n=1}^{N-1}[x^2(n) + x^2(n-1)]} \tag{4.6}$$

$$f_1(n) = x(n) - \gamma_1 x(n-1), n = 1, 2, \ldots, N-1 \tag{4.7}$$

$$b_1(n) = x(n-1) - \gamma_1 x(n), n = 1, 2, \ldots, N-1 \tag{4.8}$$

$$\begin{bmatrix} 1 \\ a_{1,1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \gamma_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow a_{1,1} = -\gamma_1 \tag{4.9}$$

r^{th} *iteration:*

$$\gamma_r = 2\frac{\sum_{n=r}^{N-1} f_{r-1}(n)br - 1(n-1)}{\sum_{n=r}^{N-1}[f_{r-1}^2(n) + b_{r-1}^2(n-1)]} \tag{4.10}$$

$$f_r(n) = f_{r-1}(n) - \gamma_r b_{r-1}(n-1), n = r, \ldots, N-1 \tag{4.11}$$

$$b_r(n) = b_{r-1}(n-1) - \gamma_r f_{r-1}(n), n = r, \ldots, N-1 \tag{4.12}$$

$$\begin{bmatrix} 1 \\ a_{r,1} \\ a_{r,2} \\ \vdots \\ a_{r,r-1} \\ a_{r,r} \end{bmatrix} = \begin{bmatrix} 1 \\ a_{r-1,1} \\ a_{r-1,2} \\ \vdots \\ a_{r-1,r-1} \\ 0 \end{bmatrix} - \gamma_r \begin{bmatrix} 0 \\ a_{r-1,r-1} \\ a_{r-1,r-2} \\ \vdots \\ a_{r-1,1} \\ 1 \end{bmatrix} \rightarrow a_{1,1}, \ldots, a_{r,r} \tag{4.13}$$

where for each stage, $a_{r,r} = -\gamma_r$.

Compared with the conventional RLSL algorithm described in Chapter 3, the following conclusions can be made:

- Burg algorithm is not as *adaptive* as RLSL, since it does not update parameters based on sample-by-sample manner but on block-by-block manner. According to (4.4), with the increase of the system order, the length of data required for calculating reflection coefficient decreases. Furthermore, with the increase of system order, the number of calculated AR coefficients increases as well. Hence for adjacent stages, their architectures are not exactly the same. Thus, the Burg system is hard to implement using the same architecture for all the stages.

- The reflection coefficient of Burg algorithm, which is used to update parameters (forward/backward prediction errors), is always less than or equal to unity. Thus, if normalization is employed to the input data and the total length of the input data is known, then for each stage, the range of values of the parameters can be determined. This means that how many number of bits used to represent these parameters can be calculated, hence fixed point arithmetic could be a solution. This is discussed in details below.

Normalization

Normalization is applied to the input data, where the input data is given as $x(n), n = 0, \ldots, N-1$.

Generally, normalization is to do a division: $x'(n) = \frac{1}{M}x(n)$, where M is the maximum absolute value of $x(n)$ and thus, $|x'(n)| \leq 1$. For the first stage, the first step is to calculate the reflection coefficient. Then, according to Equation (4.6), the range of the values of numerator and denominator in Equation (4.6) for the first stage would become

$$\left|\sum_{n=1}^{N-1} x'(n)x'(n-1)\right| \leq \sum_{n=1}^{N-1} \left|x'(n)\right| \left|x'(n-1)\right| \tag{4.14}$$

$$\leq \sum_{n=1}^{N-1} 1 \cdot 1 = N-1 \tag{4.15}$$

$$\left|\sum_{n=1}^{N-1}\left[x'^2(n)+x'^2(n-1)\right]\right| = \sum_{n=1}^{N-1}\left|\left[x'^2(n)+x'^2(n-1)\right]\right| \tag{4.16}$$

$$= \sum_{n=1}^{N-1}\left[\left|x'^2(n)\right|+\left|x'^2(n-1)\right|\right] \tag{4.17}$$

$$\leq \sum_{n=1}^{N-1}[1+1] = 2(N-1) \tag{4.18}$$

Although normalization is used to the input data, the reflection coefficient updated in the first stage does not change its value as

$$\gamma_1' = 2\frac{\sum_{n=1}^{N-1} x'(n)x'(n-1)}{\sum_{n=1}^{N-1}[x'^2(n)+x'^2(n-1)]} \tag{4.19}$$

$$= 2\frac{\sum_{n=1}^{N-1}\frac{1}{M}x(n)\cdot\frac{1}{M}x(n-1)}{\sum_{n=1}^{N-1}\left[\left(\frac{1}{M}x(n)\right)^2+\left(\frac{1}{M}x(n-1)\right)^2\right]} \tag{4.20}$$

$$= 2\frac{\frac{1}{M^2}\cdot\sum_{n=1}^{N-1}x(n)x(n-1)}{\frac{1}{M^2}\cdot\sum_{n=1}^{N-1}[x^2(n)+x^2(n-1)]} \tag{4.21}$$

$$= 2\frac{\sum_{n=1}^{N-1}x(n)x(n-1)}{\sum_{n=1}^{N-1}[x^2(n)+x^2(n-1)]} = \gamma_1 \tag{4.22}$$

Then it comes to update forward and backward prediction errors. According to Equations (4.7) and (4.8) and the fact that the absolute value of reflection coefficient is always less than or equal to unity, it is easy to obtain

$$\left|f_1'(n)\right| = \left|x'(n)-\gamma_1 x'(n-1)\right| \tag{4.23}$$

$$\leq \left|x'(n)\right|+\left|-\gamma_1 x'(n-1)\right| \tag{4.24}$$

$$\leq \left|x'(n)\right|+|-\gamma_1|\left|x'(n-1)\right| \tag{4.25}$$

$$\leq (1+1\cdot 1) = 2 \tag{4.26}$$

$$\begin{aligned}
\left|b_1^{'}(n)\right| &= \left|x^{'}(n-1) - \gamma_1 x^{'}(n)\right| && (4.27)\\
&\leq \left|x^{'}(n-1)\right| + \left|-\gamma_1 x^{'}(n)\right| && (4.28)\\
&\leq \left|x^{'}(n-1)\right| + |-\gamma_1| \left|x^{'}(n)\right| && (4.29)\\
&\leq (1 + 1 \cdot 1) = 2 && (4.30)
\end{aligned}$$

Accordingly, the relationships between new forward/backward prediction errors (with use of normalization) and original forward/backward prediction errors (without use of normalization) become as

$$\begin{aligned}
f_1^{'}(n) &= x^{'}(n) - \gamma_1^{'} x^{'}(n-1) && (4.31)\\
&= \frac{1}{M} x(n) - \gamma_1 \frac{1}{M} x(n-1) && (4.32)\\
&= \frac{1}{M} \cdot f_1(n), n = 1, 2, \ldots, N-1 && (4.33)
\end{aligned}$$

$$\begin{aligned}
b_1^{'}(n) &= x^{'}(n-1) - \gamma_1^{'} x^{'}(n) && (4.34)\\
&= \frac{1}{M} x(n-1) - \gamma_1 \frac{1}{M} x(n) && (4.35)\\
&= \frac{1}{M} \cdot b_1(n), n = 1, 2, \ldots, N-1 && (4.36)
\end{aligned}$$

Finally, for AR coefficients updates, according to Equation (4.5), calculating AR coefficients just requires the values of the reflection coefficient of the current stage and the AR coefficients from the previous stages. Additionally, the reflection coefficient does not change its value if use normalization to the input data as presented above. Hence, it is easy to conclude that the values of updated AR coefficients will not change with utilizing normalization to the input data: the roots of AR coefficients are always inside and or on the unit circle.

The second stage and the successive stages are similar to the first stage and then one can easily obtain the relationships shown in Table 4.1, where N is the length of the input data and r is the order (the number of stages).

Based on this relationship, once the data length N and number of stages (system order) r have been determined, the range of the values of parameters are also specified. Therefore, the

Numerator of γ_r	$\leq 4^{r-1}(N-r)$
Denominator of γ_r	$\leq 4^{r-1}(N-r)$
γ_r	≤ 1
$f_r(n)$	$\leq 2^{r-1}$
$b_r(n)$	$\leq 2^{r-1}$
a_k	≤ 1

Table 4.1: the relationship between the range of values of the parameters, length of input data and order for Burg algorithm

total number of bits used to represent these parameters can be calculated correspondingly. For example, if the input data length is 8000 and system order is 3, Fix_40_20 fixed point (signed 2'comp) can be used for all internal computations. Using 20 bits in fraction could obtain good accuracy, which will be shown later.

4.2 Simulink implementation and verification of Burg algorithm

4.2.1 Simulink module design

The implementation of Burg algorithm follows the same procedure as the implementation of RLSL algorithm in Chapter 3. The first step is to implement Burg algorithm on simulink block level using System Generator. The second step is to generate the HDL code for the design and then download it onto FPGA. Similarly, before downloading the design onto FPGA, it is still necessary to test the implementation at simulink block level by comparing the simulation result with the one provided by C implementation and with a Matlab command `arburg` that directly calculates AR coefficients of an input data based on Burg algorithm. The implementation of Burg algorithm at simulink block level is described below.

Figure 4.3 shows the flow diagram of calculating the reflection coefficient and AR coefficients for each stage. Two memories are needed for storing updated forward and backward prediction errors. The depth of these two memories is different for different stages, gradually decreasing one for adjacent stages. Multiplications, additions, summations and division are required to calculate the reflection coefficient. After obtaining the reflection coefficient,

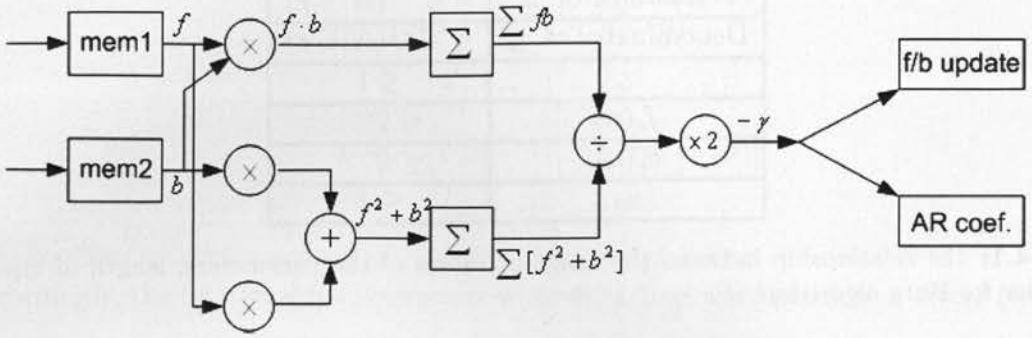

Figure 4.3: Diagram of Burg algorithm

one path is to update forward/backward prediction errors for the next stage and the other is to calculate AR coefficients for the current stage. For each stage, the architecture and modules of calculating the reflection coefficient and updating forward/backward prediction errors are the same. However, for calculating AR coefficients, with the increase of the number of stages, the number of desired AR coefficients also increases. Hence, compared with the current stage, the next stage will have one more module of computing AR coefficients, as all of the AR coefficients are calculated simultaneously in parallel. Thereby, the design of Burg algorithm, unlike the RLSL design, could just simply have one architecture suitable for all the stages. For adjacent stages, they are different. However, some components can share the same architecture and modules, i.e. the components of calculating the reflection coefficient and components for updating forward/backward prediction errors.

Figure 4.4 shows the simulink block diagram of Burg algorithm implementing 3 stages. *fid* is the input data from Workspace of Matlab. The data is the normalized data from the same VAG signal novag27 used in Chapter 3 with length of 8000 samples. *stage1* is the first stage of Burg algorithm. It has one input port *d* that is connected with *fid*. It also has four output ports, named as *fout*, *bout*, *ctrl*, and *ao1*. *fout* and *bout* send out updated forward and backward prediction errors to stage2. *ctrl* provides a control signal for stage2. *ao1* sends out the only one AR coefficient $a_{1,1}$ of stage1 (which is also the reflection coefficient of stage1) to the next stage for the calculation of AR coefficients of stage2.

stage2 is the second stage of Burg algorithm. It has four input ports, named as *en*, *fin*,

bin, and *ain1*, respectively. It also has five output ports, named as *fout*, *bout*, *ctrl*, *ao2* and *ao1*.

stage3 is the third stage of Burg algorithm (the last stage in this implementation). It has five input ports, *en*, *fin*, *bin*, *ain1*, and *ain2*; and three output ports, *ao3*, *ao2*, and *ao1*. Because it is the last stage, there are no *fout*, *bout*, *ctrl* for further stage.

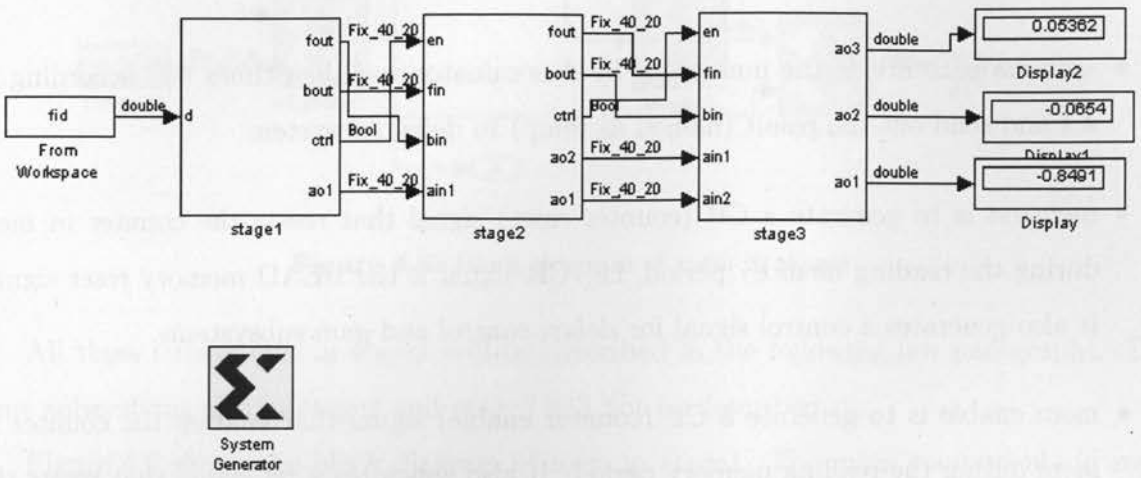

Figure 4.4: Block diagram of Burg for 3 stages

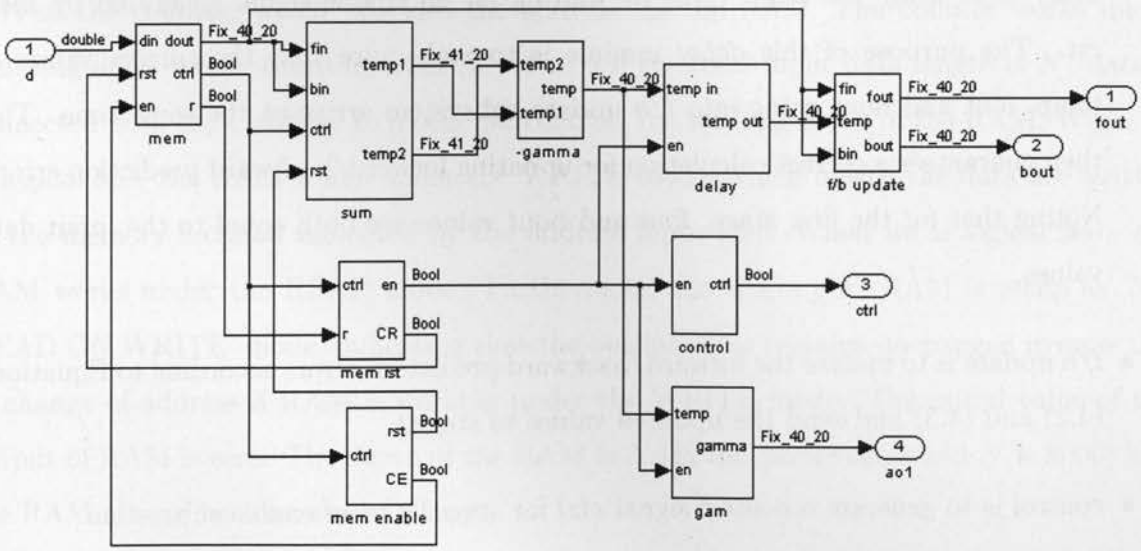

Figure 4.5: Block diagram of stage1 for Burg algorithm

Figure 4.5 shows the block diagram of *stage1* that consists of 9 subsystems.

- *mem* is to store input data and send out those data to *sum* for processing.
- *sum* is to calculate the two summations of numerator and denominator in Equation (4.4).
- *-gamma* is to divide the numerator by denominator and then times two according to 4.4 and send out the result (named as temp) to *delay* subsystem.
- *mem rst* is to generate a CR (counter reset) signal that resets the counter in *mem* during the reading memory period, i.e. CR signal is the READ memory reset signal. It also generates a control signal for *delay*, *control* and *gam* subsystems.
- *mem enable* is to generate a CE (counter enable) signal that enables the counter in *mem* during the reading memory period. It also generates a *rst* signal that resets the accumulators in *sum*.
- *delay* is to delay the temp value by waiting for an enable signal generated by *mem rst*. The purpose of this *delay* module is to make sure that the correct values of temp, fout and bout going into *f/b update* subsystem arrive at the same time. This then guarantees a correct calculation for updating forward/backward prediction errors. Noting that for the first stage, *fout* and *bout* values are both equal to the input data values.
- *f/b update* is to update the forward/backward prediction errors according to Equations (4.2) and (4.3) and send the updated values to stage2.
- *control* is to generate a control signal *ctrl* for stage2's *mem enable* subsystem.
- *gam* is to calculate $a_{1,1}$ value of stage1 and send it out for stage2.

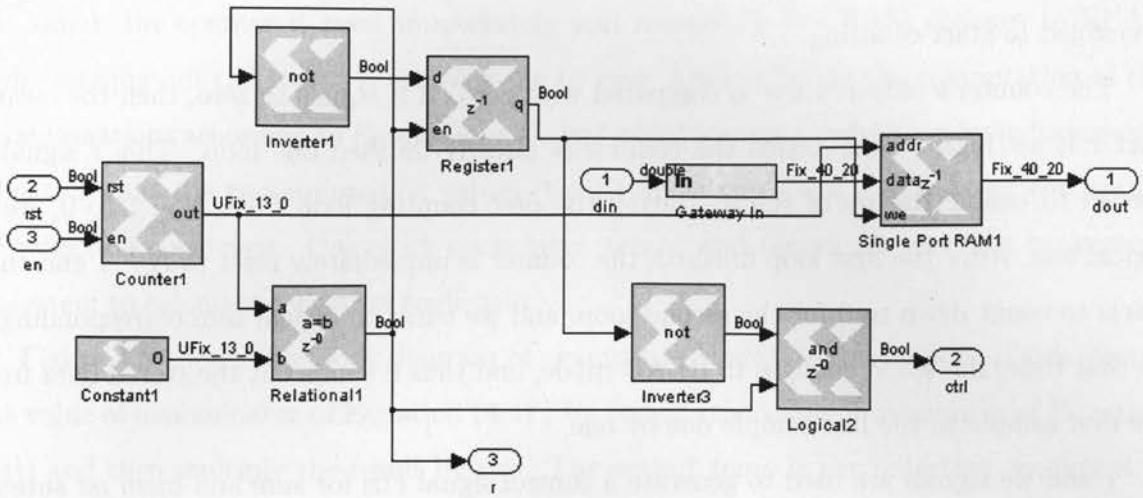

Figure 4.6: Block diagram of *mem* in stage1

All these subsystems in *stage1* will be described in the following few paragraphs. The same subsystems also in *stage2* and *stage3* will not be described.

Figure 4.6 shows the block diagram of *mem* in *stage1*. The main components in *mem* are the counter and the single port RAM.

The RAM has three input ports: *addr*, *data* and *we*. *addr* is connected with the output port of the counter, which provides the address for the RAM. The counter works under counting-down mode, counting from $N-1$ to 0 if the total input data length is N. *data* is connected with the *Gateway In* block. *we* controls the working mode of the RAM. When *we* is logical one, the RAM works under the WRITE mode, which means the data are written to the memory location indicated by the address input [38]. When *we* is logical zero, the RAM works under the READ mode. Furthermore, the single port RAM is setup as 'NO READ ON WRITE' mode, indicating that the output value remains unchanged irrespective of change of address if RAM is working under the WRITE mode. The initial value of the output of RAM is zero. The depth of the RAM is N (in this implementation $N = 8000$) and the RAM always has one clock latency.

The counter counts from $N-1$ down to 0. It has two control input ports: *rst* and *en*. When *rst* is logical one, the counter is reset to $N-1$. When *en* is logical one, the counter

is enabled to start counting.

The counter's output value is compared with zero. If it equals to zero, then the output port r is set high. That means the counter is already finished one loop. This r signal is needed to enable the *we* of RAM. During the first counting loop from $N-1$ to 0, *we* is logical one. After the first loop finished, the counter is immediately reset to $N-1$ and then starts to count down to 0 for the second loop, and *we* turns to logical zero correspondingly. At that time, the RAM changes to READ mode, and thus it sends out the stored data from the first sample to the last sample one by one.

r and *we* signals are used to generate a control signal *ctrl* for *sum* and *mem rst* subsystems.

Figure 4.7 shows the block diagram of *sum* in stage1. *sum* is to calculate the two summations in (4.4). The results will not be sent out to the following parts until *ctrl* port receives logical one.

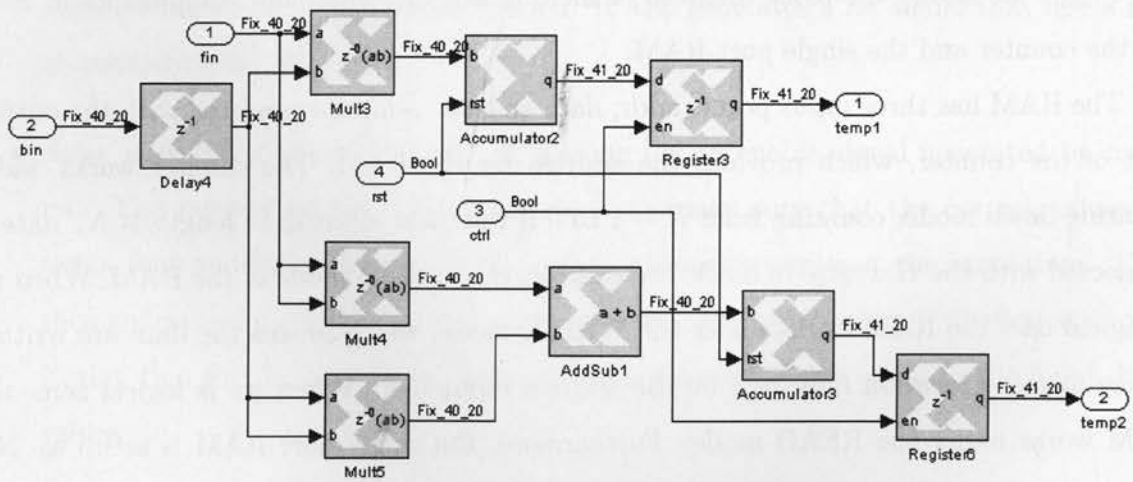

Figure 4.7: Block diagram of sum in stage1

The control signal *ctrl* is connected with the output port *ctrl* of *mem*. Combine the two subsystems *mem* and *sum* together to describe their behaviors: when the input data available, the counter starts to count, and the RAM is in WRITE mode. The N input samples are written to RAM one by one. After all the data is written to RAM (the RAM is full at

that time), the counter is reset immediately and meanwhile the RAM changes to READ mode, sending out the stored N samples one by one. After finishing the computation of the two summations according to Equation (4.4), *ctrl* signal becomes logical one instantaneously, indicating that the two summation values (temp1 and temp2) are available for computing the reflection coefficient. One clock cycle later, temp1 and temp2 are sent out to *-gamma* subsystem to calculate reflection coefficient.

Figure 4.8 shows the block diagram of *-gamma* in stage1. *-gamma* is to divide *temp1* (the value of denominator of Equation (4.4)) by *temp2* (the value of numerator of Equation (4.4)) and then multiply the result by two. The output *temp* is the reflection coefficient in Equation (4.4).

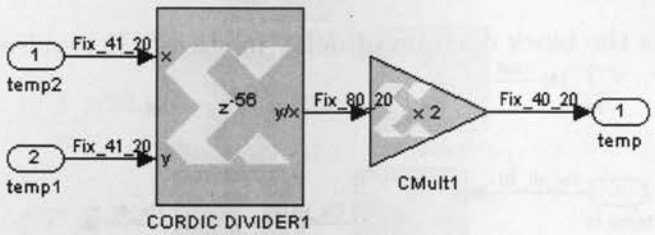

Figure 4.8: Block diagram of -gamma of stage1

Figure 4.9 shows the block diagram of *mem rst* in stage1. *mem rst* generates two control signals: *en* signal for *delay*, *control* and *gamma* subsystems, and *CR* signal for *mem* subsystem. *CR* is the signal that resets the counter in *mem*.

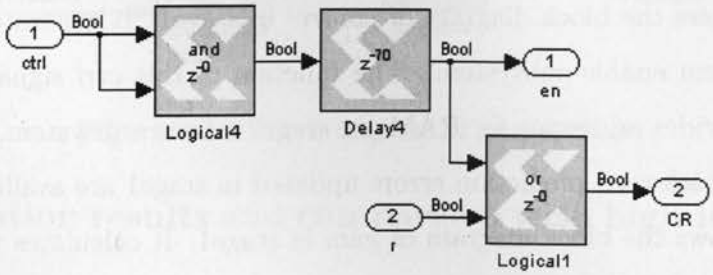

Figure 4.9: Block diagram of *mem rst* of stage1

Figure 4.10 shows the block diagram of *mem enable* in stage1. *mem enable* generates two control signals: *CE* for *mem* subsystem and *rst* for *sum* subsystem. It guarantees that *CE* holds logical one while the data is written to RAM.

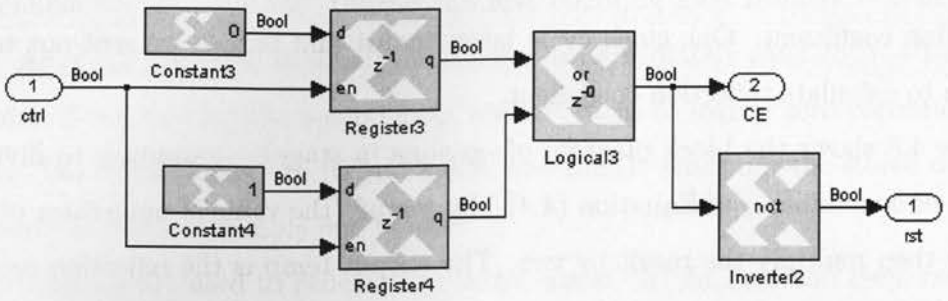

Figure 4.10: Block diagram of *mem enable* of stage1

Figure 4.11 shows the block diagram of *delay* in stage1. It requires an enable signal by *mem rst*.

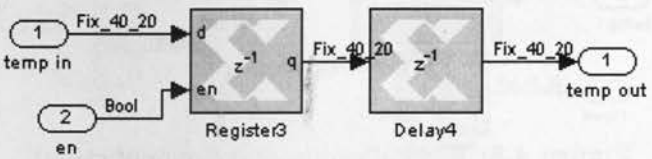

Figure 4.11: Block diagram of *delay* of stage1

Figure 4.12 shows the block diagram of *f/b update* in stage1. It does exactly the same function as Equation (4.2) and (4.3) do.

Figure 4.13 shows the block diagram of *control* in stage1. It generates a control signal *ctrl* for stage2's *mem enable* subsystem. The function of this *ctrl* signal is to enable the counter, which provides addresses for RAMs in stage2's *mem* subsystem, to start to count when the forward/backward prediction errors updated in stage1 are available.

Figure 4.14 shows the block diagram of *gam* in stage1. It calculates the AR coefficient of stage1, keeps the value unchanged and sends it out for the next stage.

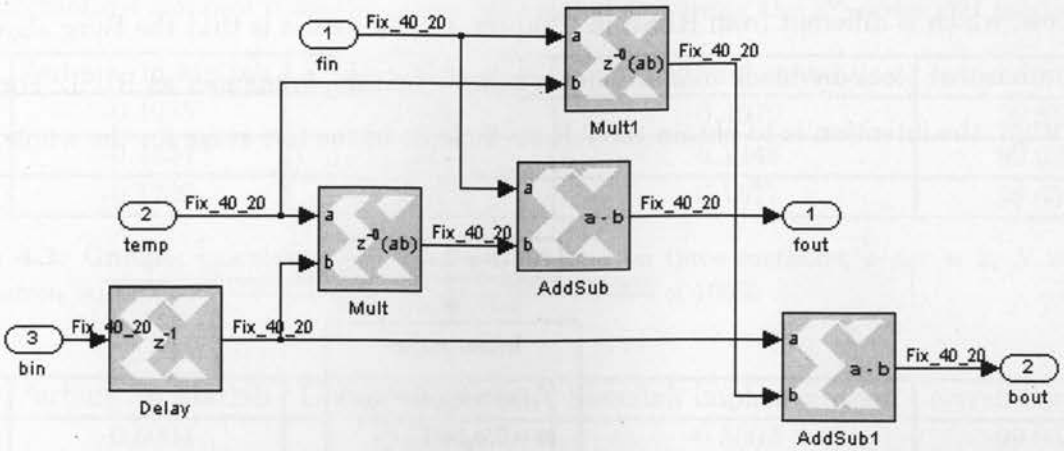

Figure 4.12: Block diagram of *f/b update* in stage1

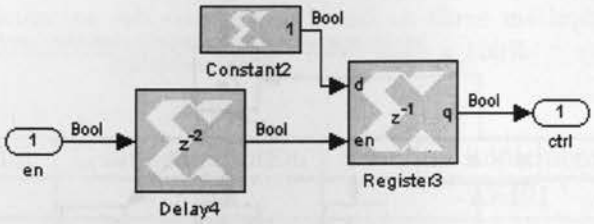

Figure 4.13: Block diagram of *control* in stage1

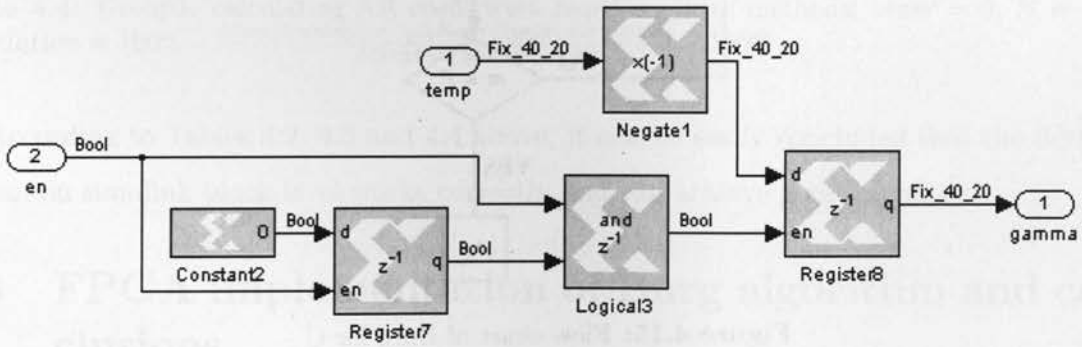

Figure 4.14: Block diagram of gam of stage1

4.2.2 Simulation results and comparison with high-level languages

The Burg algorithm has been implemented on C language first. Figure 4.15 shows the flow chart of the C implementation. Obviously, there is no time-update (no loop for N) in

the flow, which is different from RLSL in Chapter 3. The reason is that the Burg algorithm is on a basis of block-by-block manner, not a sample-by-sample manner as RLSL. For Burg algorithm, the intention is to obtain the AR coefficients of the last stage for the whole input data.

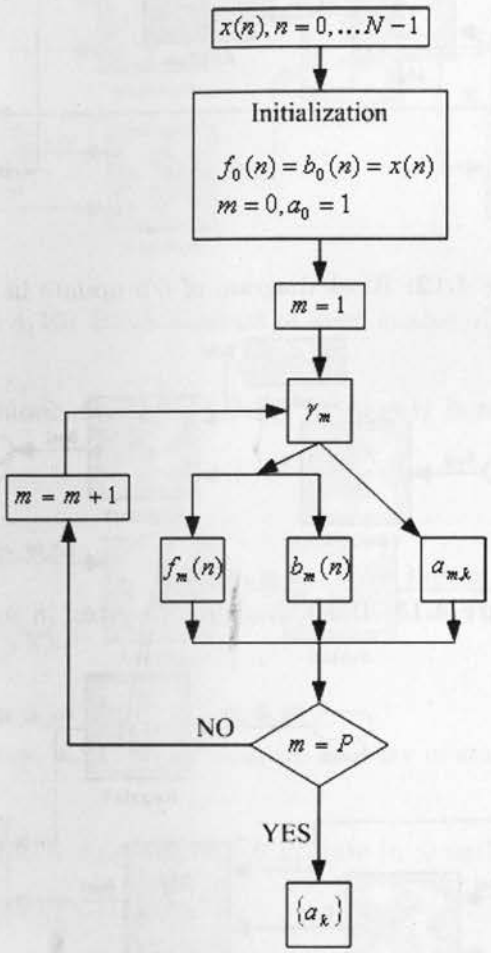

Figure 4.15: Flow chart of Burg for C

The comparison of simulation results is based on three groups between Matlab command, C implementation and simulink block level implementation as shown in Tables 4.2, 4.3 and 4.4. The first group is taking 120 normalized samples from the VAG signal novag27 used in Chapter 3 and using the 3^{rd} order AR model. The second one is taking 1000 normalized samples from the same VAG signal and using the 3^{rd} order AR model. The third one is taking

8000 normalized samples from the same VAG signal and using the 3^{th} order AR model.

$a_{m,k}$	'arburg' in Matlab	C implementation	Simulink implementation	correlation (%)
$a_{3,1}$	-0.4085	-0.4085	-0.4079	99.85
$a_{3,2}$	-0.1434	-0.1434	-0.1448	99.02
$a_{3,3}$	-0.1700	-0.1700	-0.1677	98.65

Table 4.2: Group1: calculating AR coefficients based on three methods; $order = 3$, $N = 120$; $correlation = 100\% - \frac{|Simulinkimplementation - Cimplementation|}{|Cimplementation|} \times 100\%$

$a_{m,k}$	'arburg' in Matlab	C implementation	Simulink implementation	correlation (%)
$a_{3,1}$	-0.5004	-0.5004	-0.5002	99.96
$a_{3,2}$	-0.1114	-0.1114	-0.1115	99.91
$a_{3,3}$	-0.0746	-0.0746	-0.07407	99.29

Table 4.3: Group2: calculating AR coefficients based on three methods; $order = 3$, $N = 1000$; $correlation = 100\% - \frac{|Simulinkimplementation - Cimplementation|}{|Cimplementation|} \times 100\%$

$a_{m,k}$	'arburg' in Matlab	C implementation	Simulink implementation	correlation (%)
$a_{3,1}$	-0.8486	-0.8486	-0.8491	99.94
$a_{3,2}$	-0.0655	-0.0655	-0.0654	99.85
$a_{3,3}$	0.0530	0.0530	0.05362	98.83

Table 4.4: Group3: calculating AR coefficients based on three methods; $order = 3$, $N = 8000$; $correlation = 100\% - \frac{|Simulinkimplementation - Cimplementation|}{|Cimplementation|} \times 100\%$

According to Tables 4.2, 4.3 and 4.4 above, it can be easily concluded that the designed system on simulink block level works correctly and can achieve good accuracy.

4.3 FPGA implementation of Burg algorithm and conclusions

To implement Burg algorithm onto FPGA, we follow the same design methodology of implementing RLSL as explained in Chapter 3.

Table 4.5 shows the device usage of Burg algorithm with order of 3. It does not occupy much resources of the target device, however, to implement more stages, more resources will be consumed.

Number of Slice Flip Flops	18%
Number of 4 input LUTs	20%
Number of occupied Slices	25%
Number of bonded IOBs	15%
Number of Block RAMs	20%
Number of MULT18X18s	32%
Number of GCLKs	6%

Table 4.5: Utilization summary of Burg algorithm implementing 3 stages; target device: xc2vp100-6-ff1704

To evaluate the performance of the FPGA implementation of Burg algorithm, a testbench has been built to test the designed system. After re-implementing and downloading the Burg system and testbench onto FPGA successfully, chipscope is used to capture and display the output values. The order of Burg algorithm is 3 and the input data is using the same normalized 8000 samples from the knee signal novag27 used in group3 for simulink block implementation testing earlier in this chapter. Figure 4.16 shows output values captured by chipscope. The three output values are shown in binary (and hexadecimal). After converting binary values to decimal (calculated by Matlab), the values are compared with the results obtained from C implementation and Matlab command **arburg** as shown in Table 4.6.

$a_{m,k}$	'arburg' in Matlab	C implementation	FPGA implementation	correlation (%)
$a_{3,1}$	-0.8486	-0.8486	-0.8493	99.92
$a_{3,2}$	-0.0655	-0.0655	-0.0652	99.54
$a_{3,3}$	0.0530	0.0530	0.0536	98.68

Table 4.6: AR coefficients comparison for 3 stages; $order = 3$, $N = 8000$; $correlation = 100\% - \frac{|FPGAimplementation - Cimplementation|}{|Cimplementation|} \times 100\%$

Conclusions

Burg algorithm has been implemented onto FPGA successfully with 3 stages processing input block data of 8000 samples. It uses fixed point arithmetic for internal computations. Compared with C implementation based on double precision floating point arithmetic, the implemented Burg design works correctly and has good accuracy, which demonstrates that using 20 bits in fractional part is satisfactory. To implement more stages (i.e. 6 stages) for

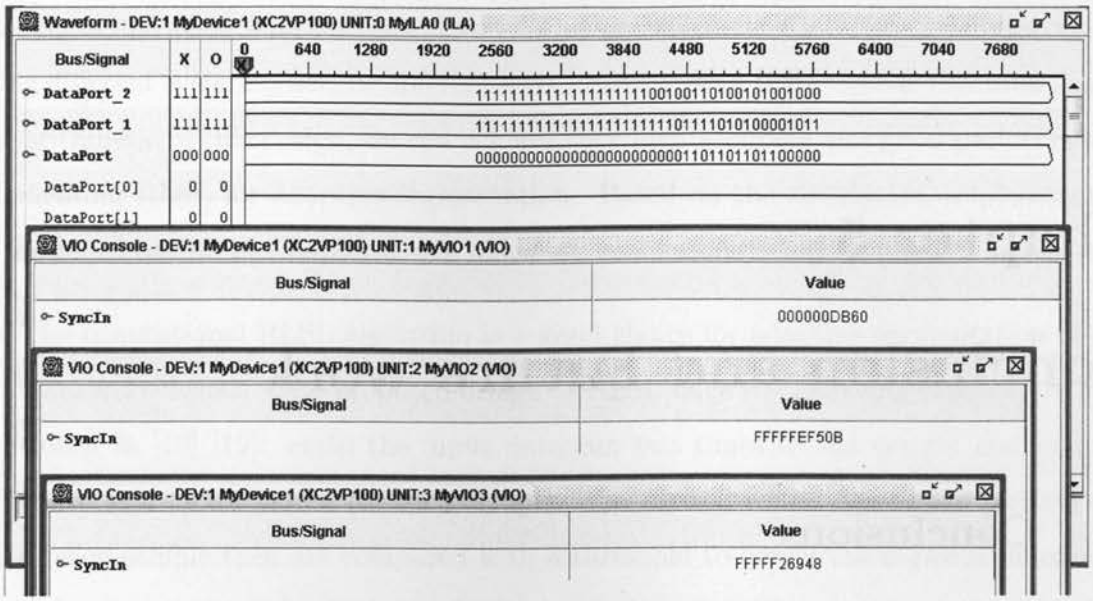

Figure 4.16: Output values of AR coefficients obtained from FPGA implementation; *order* = 3,$N = 8000$

processing the same data of 8000 samples, more number of bits need to be used in integral part and more area and resources will be occupied, as one needs to add models to the design to implement stage4 to stage6. However, one can find a solution to make the Burg system share exactly the same architecture for all the stages to save area and use more number of bits in fractional part, or consider using floating point arithmetic for internal calculations to achieve higher accuracy.

Chapter 5

Conclusion and Future work

5.1 Conclusions

In this thesis, high-level module designs of RLSL algorithm for adaptive segmentation and Burg algorithm for AR modeling have been proposed and implemented onto Xilinx FPGA Virtex II Pro by using Simulink-to-FPGA design flow. The introduction explained the motivation of this project: why we need the adaptive segmentation and AR modeling techniques and the reasons why we are interested in implementing the selected algorithms onto hardware. Chapter 2 gave a review of the existing methods for adaptive segmentation, the techniques for AR modeling and VLSI implementation survey of these two applications and algorithms. Chapter 3 described the RLSL method for adaptive segmentation and proposed its FPGA implementation. Chapter 4 described the Burg algorithm for AR modeling and proposed its FPGA implementation. The summary of the research and the discussion of the future work are presented in this chapter.

RLSL for adaptive segmentation

A new system-level module design based on the conventional recursive least-squares lattice algorithm, which can calculate the conversion factor values of each input sample for adaptive segmentation and further provide segment boundaries, has been proposed and implemented onto FPGA in this thesis. The design uses an architecture with high flexibility that provides user-defined order selection port and data length selection port, which are up

to 32 stages and 5000 samples respectively. This is convenient for different applications requiring different system orders or different data length. Using IEEE double precision floating point arithmetic in this design, we can achieve very high accuracy and good performance of implementing RLSL for adaptive segmentation. Based on the simulation, implementation and produced results in Chapter 3, the following conclusions could be made:

- The conventional RLSL algorithm is a good choice for adaptive segmentation of non-stationary signals. The proposed design of RLSL have the following functions as mentioned in [18] [19]: make the input data run two times to the system and calculate conversion factor values for each sample. The second round conversion factor values of each sample then are compared with a threshold to detect the segment boundaries. It has been tested for both synthesized signal and real biomedical signal, i.e. VAG signal. The performance of the implemented system is satisfactory in that it is flexible, programmable and accurate with a mean correlation of 99.93%.

- Double precision floating point arithmetic can be used for implementing RLSL method, which achieved very high accuracy performance, not only providing accurate conversion factor values of each input sample, but also providing accurate boundary positions.

- The proposed implementation is programmable for both data length and system order selection. It provides data length selection port and order selection port for users, which can be used for different applications and/or different input data.

- The proposed implementation is both area-efficient and stable. It shares an architecture used for all the stages. No matter how many stages set up to run for the system, the area of the system does not change. Additionally, the total area of the implemented RLSL design does not occupy much of the target device.

- The proposed implementation is speed-accessible for real-time processing purpose. It worked at 20MHz, which can be used for real-time processing, e.g. sampling rate at 2kHz as used in [18].

Burg algorithm for AR modeling

A new FPGA design based on Burg algorithm that can directly calculate the AR coefficients is proposed in this thesis. The design takes advantage of Xilinx System generator implementing tool. Based on the simulation and result comparison, the following conclusions could be made:

- Fixed-point type arithmetic representation is suitable for implementing Burg algorithm, since the system could not use an architecture for all the stages and if normalization is applied to the input data. The implemented Burg algorithm works correctly and can have good accuracy.
- The more stages it is implemented, the more area it consumes, and the area requirements would increase linearly according to system order based on the current design method: the adjacent stages do not share exactly the same architecture, but most of the models are the same.

Simulink-to-FPGA design flow

The two algorithms with specific application purposes are both implemented based on a simulink-to-FPGA design flow, which has salient features:

- Friendly graphics interface. Using simulink, it is easy to organize input data and also observe the output in different ways.
- Flexible modeling and ease of simulation. The design can be well organized into modules in hierarchy manners and so it is convenient to run simulation and debug.

5.2 Future work

The future work of the research could be:

- Find a solution to make the Burg algorithm share the same structure for each stage like RLSL that can save area.

- Use more number of bits in fractional part for Burg design or consider using floating point arithmetic for its internal calculations to achieve higher accuracy.

Bibliography

[1] R. M. Rangayyan, *Biomedical signal analysis: a case-study approach.* New York, N.Y.: Wiley-Interscience, 2002.

[2] R. M. Rangayyan, S. Krishnan, G. D. Bell, C. B. Frank, and K. O. Ladly, "Parametric representation and Screening of knee joint vibroarthrographic signals," *IEEE Transactions on biomedical engineering*, vol. 44, pp. 1068–1074, November 1997.

[3] S. Tavathia, R. Rangayyan, C. Frank, G. Bell, K. Ladly, and Y. Zhang, "Analysis of knee vibration signals using linear prediction," *IEEE transactions on biomedical engineering*, vol. 39, no. 9, pp. 959–970, 1992.

[4] M. Akay, J. L. Semmlow, W. Welkowitz, M. D. Bauer, and J. B. Kostis, "Detection of Coronary occlusions using autoregressive modeling of diastolic heart sounds," *IEEE transactions on biomedical engineering*, vol. 37, pp. 366–373, April 1990.

[5] B. Ahmadi, R. Aimrfattahi, E. Negahbani, M. Mansouri, and M. Taheri, "Comparison of adaptive and fixed segmentation in different calculation methods of electroencephalogram time-series entropy of estimating depth of anesthesia," 6^{th} *International special topic conference on ITAB, 2007, Tokyo*, pp. 265–268, 2008.

[6] Z. M. K. Moussavi, R. M. Rangayyan, G. D. Bell, C. B. Frank, K. O. Ladly, and Y.-T. Zhang, "Screening of vibroathrographic signals via adaptive segmentation and linear prediction modeling," *IEEE transactions on biomedical engineering*, vol. 43, pp. 15–23, January 1996.

[7] M. Diaby, M. Tuna, J. Desbarbieux, and F. Wajsburt, "High level synthesis methodology from C to FPGA used for a network protocol communication," *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping, 2004.*, pp. 103–108, June 2004.

[8] K. Camera, "SF2VHD: A stateflow to VHDL translator," *Master thesis, UC Berkeley*, 2001.

[9] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A system for synthesizing optimized FPGA hardware from Matlab," *IEEE/ACM International Conference on Computer Aided Design, 2001.*, pp. 314–319, 2001.

[10] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Automated synthesis of pipelined designs on FPGAs for signal and image processing applications described in MATLAB," *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference, 2001.*, pp. 645–648, Feb. 2001.

[11] X. Li, F. Sun, and E. Wu, "A Simulink-to-FPGA Co-Design of encryption module," *IEEE Asia Pacific Conference on Circuits and Systems, 2006*, pp. 2008–2011, Dec. 2006.

[12] M. A. Shanblatt and B. Fould, "A Simulink-to-FPGA implementation tool for enhanced design flow," *Proceedings of the 2005 IEEE international conference on microelectronic systems education (MSE'05)*, pp. 89–90, 2005.

[13] G. Bodenstein and H. Praetorius, "Feature extraction from the electroencephalogram by adaptive segmentation," *Proceedings of the IEEE*, vol. 65, pp. 642–652, May 1977.

[14] J. Pardey, S. Roberts, and L. Tarassenko, "A review of parametric modelling techniques for EEG analysis," *Med. Eng. Phys.*, vol. 18, pp. 2–11, January 1996.

[15] D. Michael and J. Houchin, "Automatic EEG analysis: A segmentation procedure based

on the autocorrelation function," *Electroencephalography and Clinical Neurophysiology 46*, pp. 232–235, 1979.

[16] U. Appel and A. V. Brandt, "Adaptive sequential segmentation of piecewise stationary time series," *Information Sciences*, vol. 29, pp. 27–56, 1983.

[17] U. Appel and A. V. Brandt, "A comparative study of three sequental time series segmentation algorithms," *Signal processing 6*, pp. 45–60, 1984.

[18] S. Krishnan, R. M. Rangayyan, G. D. Bell, C. B. Frank, and K. O. Ladly, "Recursive least-squares lattice-based adaptive segmentation and autoregressive modeling of knee joint vibroarthrographic signals," *Canadian Conference on Electrical and Computer Engineering, 1996*, vol. 1, pp. 339–342, May 1996.

[19] S. Krishnan, "Adaptive filtering, modeling, and classfication of knee joint vibroarthrographic signals," *Master's thesis, Department of Electrical and Computer Engineering, University of Calgary*, April 1996.

[20] S. Haykin, *Adaptive filter theory.* Upper Saddle River, N.J.: Prentice Hall, 4th ed., 2002.

[21] C. Paleologu, S. Ciochina, and A. A. Enescu, "Low dynamics RLSL adaptive algorithm using a priori estimation errors," *International Multi-Conference on Computing in the Global Information Technology, 2006*, pp. 47–47, Aug. 2006.

[22] R. C. North, R. Zeidler, W. H. Ku, and T. R. Albert, "A floating-point arithmetic error analysis of direct and indirect coeffcient updating techniques for adaptive lattice filters," *IEEE transactions on signal processing*, vol. 41, pp. 1809–1823, May 1993.

[23] F. Ling, D. Manolakis, and J. G. Proakis, "Numerically robust least-squares lattice-ladder algorithms with direct updating of the reflection coeffcients," *IEEE Transactions on acoustics, speech, and signal processing*, vol. 34, pp. 837–845, August 1986.

[24] B. Friedlander, "Lattice filters for adaptive processing," *Proceedings of the IEEE*, vol. 70, pp. 829–867, August 1982.

[25] J. R. Bunch, R. C. L. Borne, and I. K. Proudler, "Analysis of the direct and indirect *a posteriori* rlsl algorithm," *Numerical linear algebra with applications*, pp. 453–466, 2001.

[26] J. R. Bunch and R. C. LeBorne, "Error accumulation effects for the *a posteriori* RLSL prediction filter," *IEEE transctions on signal processing*, vol. 43, pp. 150–159, January 1995.

[27] M. Aboy, O. W. Marques, J. McNames, R. Hornero, T. Trong, and B. Goldstein, "Adaptive modeling and spectral estimation of nonstationary biomedical signals based on Kalman filtering," *IEEE Transactions on Biomedical Engineering*, vol. 52, pp. 1485–1489, Aug. 2005.

[28] D. Ge, N. Srinivasan, and S. M. Krishnan, "Cardiac arrhythmia classification using autoregressive modeling," *BioMedical Engineering Online (http://biomedical-engineering-online.com/content/1/1/5)*, Nov. 2002.

[29] M. Akay, M. Bauer, J. L. Semmlow, W. Welkowitz, and J. Kostis, "Autoregressive modeling of diastolic heart sounds," *IEEE engineering in medicine & biology society* 10^{th} *annual international conference*, 1988.

[30] S. H. Kim, H. B. Han, K. R. Hong, M. H. Lee, and S. H. Park, "Pattern classification of AR model parameters of EMG signal for the diagnosis of TMJ dysfunction syndrome," *Proceedings - TENCON 87: 1987 IEEE Region 10 Conference, 'Computers and Communications Technology Toward 2000'*, pp. 1317–1321, 1987.

[31] Z. Luo, F. Wang, and W. Ma, "Pattern classification of surface electromyography based on AR model and high-order neural network," *2006 IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications*, pp. 1–6, Aug. 2006.

[32] A. Angelidou, M. Strintzis, S. Panas, and G. Anogianakis, "On AR modelling for MEG spectral estimation, data compression and classification," *IEEE Transactions on Computers in Biology and Medicine*, pp. 379–87, Nov. 1992.

[33] J. Makhoul, "Linear prediction: a tutorial review," *Proceedings of the IEEE*, vol. 63, pp. 561–580, April 1975.

[34] Z. Pohl, M. Tichy, and J. Kadlec, "Implementation of the least-squares lattice with order and forgetting factor estimation for FPGA," *EURASIP Journal on advances in signal processing*, vol. 2008, June 2008.

[35] A. Hermanek, Z. Pohl, and J. Kadlec, "FPGA implementation of the adaptvie lattcie filter," *Springer-Verlag Berlin Heidelberg 2003*, pp. 1095–1098, 2003.

[36] F. Albu, L. Kadlec, and e. a. C. Softley, "Implementation of (normalised) RLSL lattice on Virtex," *in proceedings of the* 11^{th} *international conference on filed programmable logic and applications (FPL 01), Springer, Northern Ireland, UK*, pp. 91–100, August 2001.

[37] M. R. Smith, T. J. Smith, S. W. Nichols, S. T. Nichols, H. Orbay, and K. Campbell, "A hardware implementation of an autoregressive algorithm," *Meas. Sci Technol, IOP Publishing Ltd*, vol. 1, pp. 1000–1006, 1990.

[38] "Xilinx System Generator for DSP version 9.1.01 User's Guide," *(http://www.xilinx.com/support/sw_manuals/sysgen_ug.pdf)*, March 2007.

[39] "ISE In-Depth tutorial," *(http://download.xilinx.com/direct/ise9_tutorials/ise9tut.pdf)*, July 2007.

[40] "ChipScope Pro software and cores user guide (ChipScope Pro software 9.2i)," May 2007.