LPC and Sparse LPC Algorithms and RTL Architecture

Project Report

Group 974

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
LPC and Sparse LPC Algorithms and RTL Architecture

Theme:
Reconfigurable Computing

Project Period:
Master thises

Project Group:
14GR974

Participants:
Henrik Holbæk Pedersen

Supervisor:
Peter Koch
Tobias Lindstrøm Jensen

Copies: 3

Page Numbers: 156

Date of Completion:
August 26, 2015

Abstract:

This project covers the analysis of linear prediction coding, and sparse linear prediction algorithms, with the goal of creating a Register Transfer Level architecture.
Two algorithm is then suggested to find a set of, linear prediction coefficients, and sparse linear prediction coefficients, these algorithms are analysed finding the flow, and inherent parallelism. With the analysis of these algorithms a finite state machine with data path, is build consisting of a set of the control path, and data path. The control path is build using algorithmic state machine charts, and data path is build consisting a set of hardware blocks.
The sparse linear prediction algorithm analysed in this report is a novel idea recently presented by the supervisor Tobias Lindstrøm Jensen, therefore a larger part of the analysis is used to investigate it. The given algorithm consists of four iterations where a right hand side least squares problem is solved each iteration. Given this it is found that the Levinson algorithm is the most sufficient way of finding a solution, where other methods are also investigated. The report ends up with two architectures, which can be either implemented in VHDL or further improved employing different optimizations methods.

Contents

List of Acronyms

PG Precedence Graph

VoIP Voice Over IP

LPC Linear Predictive Coding

MOS Mean-Opinion score

CELP Code-Excided Linear Prediction

FPGA Field Programmable Gate Array

LPA Linear Predictive Analysis

SLPC Sparse Linear Predictive Coding

ADMM Alternating Direction Method of Multipliers

HD High-definition

AR Autoregressive

MA Moving Average

ARMA Autoregressive Moving Average

CDFG Control Data Flow Graph

MAM Multiply Addition Multiply

MSA Multiply Addition Subtraction

ASAP As Soon As Possible

ALAP As Late As Possible

FSM Finite State Machine

FSMD Finite State Machine with Data Path

PG Precedence Graph

DFG Data Flow Graph

VHDL VHSIC Hardware Description Language

IIR Infinite impulse response

MSD Mean Square Deviation

RTL Register-transfer level

ASM Algorithmic state machine

FFT Fast Fourier Transform

Preface

This report contains the work done by Henrik Holbæk Pedersen during his 9th and 10th semester of the master program, "Signal Processing and Computing", at Aalborg university. The project proposal was worked out by the student, and the supervising professors, Peter Koch and Tobias Lindstrøm Jensen. The proposed project concerns speech coding, where it is analysed if it is possible to derive a real time implementation of a speech encoding using linear predictive coding. Where the objective is finding a sparse residual, and compare this with a regular speech encoding. The first part of this report concerns describing and analysing these algorithms, into detail to understand the idea behind behind Linear Predictive coding. Where both proposed methods are simulated to get an insight into how the input and output of the algorithms is. The second part of the of the report concerns the architecture, where the algorithms are analysed with the goal of making a real time layer architecture. This part ends up with two Finite State Machines with Data path. last part of the report gives an introduction into how the Finite State Machine with Data Path (FSMD) can be turned into a VHSIC Hardware Description Language (VHDL) implementation, and lastly the report is concluded and discussed, where the resulting FSMD are considered looking at how these can be improved.
During this report a large part of the time was put into reading up on Linear Predictive Coding, as this was a new field of study. This also means that the Sparse Predictive Coding scheme was a new field, this is also due, that the proposed algorithm is novel. So the literature about this part is relative little. As the sparse coding scheme is a new novel idea a large part of time was used to analyse the given algorithm as no literature is available concerning the implementing. These unforeseen workloads means that only little time was put into implementing and improving the RTL analysis. Even though the size the project was larger then manageable, given the unforeseen workloads in analysing a new algorithm, it has been a very interesting and challenging experience, which gives a good insight into how long it takes to take a novel algorithm and turn into an application.

The Work done during the project period has been compiled into a report, and follows the the description below. The source code used to produce the given plots and figures are available on the attached CD.
As this report is done with an architecture in mind, a figure depicting the flow is depicted in figure 1.

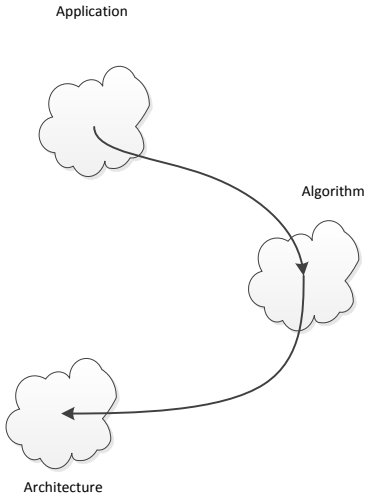

Figure 1: The a^3 figure representing the 3 different Levels of going from an application to having an architecture

Where the step from application into algorithm is described in Chapter 1-4, and the step from taking the algorithm into an architecture is described in Chapter 5-7.

Chapter 1 This chapter introduces the Linear predictive coding scheme, and the idea behind it. Ending up with a initial problem formulation and a set of requirements for the given architecture.

Chapter 2 This chapter analyses the Linear prediction ending up with the two given algorithms for solving the Linear Predictive Coding, and Sparse Linear Predictive Coding.

Chapter 3 This chapter analyses the novel Alternating Direction Method of Multipliers (ADMM) algorithm to identify solutions to solve it efficiently.

Chapter 4 This chapter simulates both the Linear Predictive Coding and sparse Linear Predictive Coding.

Chapter 5 This chapter introduces the method used to analyse the algorithm, going from the application into algorithm, which gives an architecture.

Chapter 6 This chapter analyses the Levinson-Durbin algorithm, used to find the Linear Prediction.

Chapter 7 This chapter analyses the ADMM algorithm, used to find the Sparse Linear Prediction.

Chapter 8 This chapter gives an insight into how the given architectures can be turned into an VHDL implementation.

Chapter 9 This chapter concludes the project, where the architecture is discussed and how it can be further improved.

Aalborg University, August 26, 2015

Henrik Holbæk Pedersen
<hhpe07@student.aau.dk>

Chapter 1

Introduction

In speech communication Voice Over IP (VoIP) are becoming more and more popular, and with that various protocols for VoIP exists. With each protocols there exists different voice encoding methods, called vocoders. These methods are evaluated on different parameters such as speech quality, communication delay, computational complexity of implementation, power consumption, robustness to noise and robustness to packet losses [Tokunbo Ogunfunmi and Madihally J. (Sim) Narasimha, 2012, p.37]
The idea of a vocoder is to compress the speech signal as much as possible and keep it to a point where the quality is still acceptable. Speech quality can be measured in different ways, one way of doing so is the Mean-Opinion score (MOS),which is a five point scale, giving scores from 1-5 where five is the best, and of 3.5 or higher implies high level of intelligibility, speaker recognition and naturalness [Tokunbo Ogunfunmi and Madihally J. (Sim) Narasimha, 2012, p.37]. One way of doing low bit rate speech coding is by using Linear Predictive Coding (LPC). LPC was developed for low bit-rate speech coding [Chu, p.263], where the early beginning of Linear prediction dates back to the 1940s [Vaidyanathan, p.1]. Where an early influential paper about LPC is the Makhoul [1975] according to [Vaidyanathan, p.1]. As LPC is a low bit-rate coding algorithm, providing a low quality sound, however the theories behind LPC gives foundation to more advanced speech coding schemes such as Code-Excided Linear Prediction (CELP) [Chu, p.281]. CELP is another coding method where a long and short term Linear prediction model is used [Chu, p.299]. Currently CELP and variations of it is being used for various applications, and therefore it is still interesting to investigate the fundamentals behind Linear prediction. As such this project will focus on two different ways to implement Linear prediction, Linear Predictive Coding and Sparse Linear Predictive Coding, in a real time implementation, on a Field Programmable Gate Array (FPGA). Both vocoders will be evaluated according to the computational complexity of implementation, power consumption, algorithmic delay and speech quality. The robustness to noise and robustness to packet loss is out of scope as it requires a more extensive setup and test.
In this chapter the model behind human speech is explained, and the link to a mathematical model of human speech.

1.1 Human Speech

Human speech is a combination of voiced and unvoiced speech john R Deller jr [1999], as shown in figure 1.1 When a person speaks air comes from the lungs and goes through the vocal cords and vocal tract. The vocal cords determines the pitch of the sound, this is done by having the vocal cords vibrate and the speed of the vibrations determines the pitch. For unvoiced the vocal cords remains open. The vocal tract determines the sound one makes, and it changes for each sound.
So the air volume in the lungs determines the gain, the vocal cords determines the frequencies of the excitation of the signal, and the rate of which the vocal cord changes frequencies. The vocal tract is where the sound is determined and the shape the tract forms decides which sound is made.

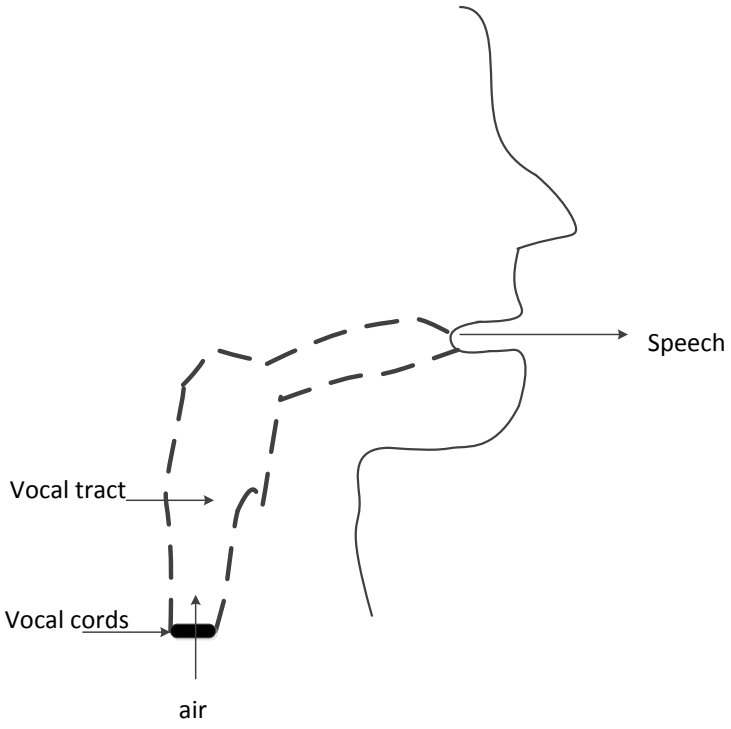

Figure 1.1: The figure depicts a human vocal cords and vocal tract

1.2 Speech Model

The human speech organ can be translated into a mathematical model, which can put each part of human speech into a mathematical expression. Figure 1.2 depicts the a model for speech, and the relationship is as follows:

Speech expression	Mathematical expression
Vocal tract	H(z)(Model filter)
Excitation signal	u(n)
Vocal cord vibration	V(voiced)
Vocal cord vibration period	T(pitch period)
No vocal cord vibration	UV(unvoiced)
Air Volume	G(gain)

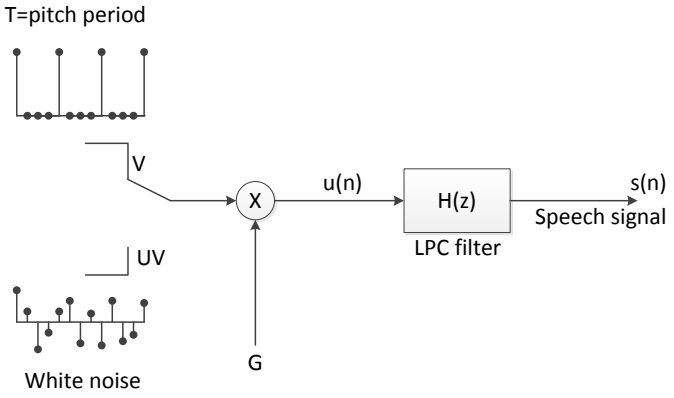

Figure 1.2: The figure depict the LPC model which models human speech

So to make a relationship with the human speech figure 1.1, when the human air moves through the vocal cords, if the vocal cords are vibrating then the pitch period T is determined by the frequency of which it vibrates. If the vocal cords are not vibrating then, the signal is instead white noise where there is no pitch period. The signal is then multiplied by a gain G, which describes the volume of air pressed through the vocal cords. The signal u(n) is then put through the Model filter, which models the vocal tract, This filter describes the impulse response of the vocal tract. The Model filter H(z) is an Infinite impulse response (IIR) filter given by:

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots a_p z^{-p}} \tag{1.1}$$

where p is the order of the filter.

1.3 Linear Predictive Coding

In Linear Predictive Coding the Model filter coefficients is extracted from the speech signal, this is done in an so called Linear Predictive Analysis (LPA). The LPA finds a set of linear predicted filter coefficients and an error between the predicted signal

and the real signal, mathematical background behind the Linear Predictive analysis is further described in section 2.1.1. The filter coefficients and the error is then packed and send through some protocol to another user, where the Linear Predictive Synthesis unpacks the coefficients and error, to recreate the speech signal, the scheme is depicted in figure 1.3.

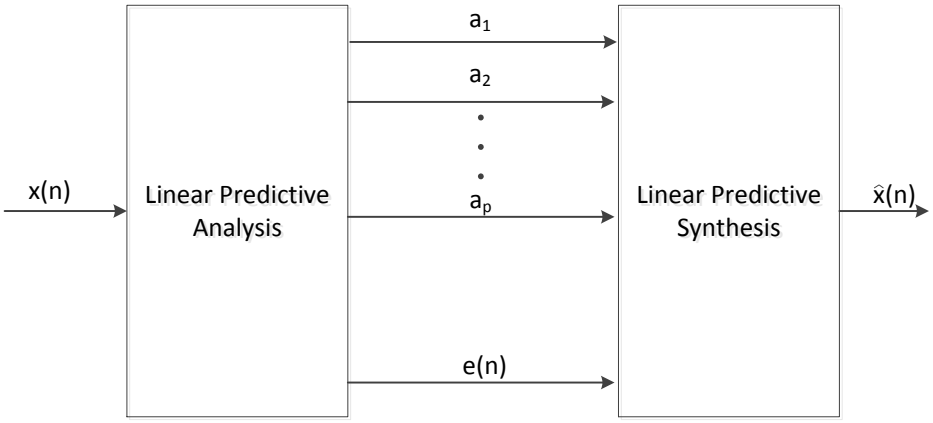

Figure 1.3: Linear predictive coding scheme

In the LPA the filter coefficients is commonly extracted using the Levinson-Durbin recursion, which is further described in section 2.1.2

Application

LPC is an old concept and is currently used for speech coding in various applications, with different expansion as:

- Low-Delay Code Excited Linear Prediction (LD-CELP)
- Conjugate-Structured Algebraic Code Excited Linear Prediction (CS-ACELP)
- Code Excited Linear Prediction (CELP)
- Algebraic Code Excited Linear Prediction (ACELP)

These implementations all use LPC, as a basis, where a set of filter coefficients and error is found.

1.4 Sparse Linear Predictive Coding

Lately a new Linear Predictive idea is being explored Daniele Giacobello [2010], this scheme is to make the coefficients and the error sparse. The idea behind this is to extract filter coefficients and error, where as much of the signal then is zero, and there for does not need to be represented. The concept of Sparse Linear Predictive Coding is further explained in section 4.1.

1.5 Initial Problem Formulation

In this section the setup for the LPC and Sparse Linear Predictive Coding (SLPC) is investigated, leading up to a initial requirement specification which is then used for the simulation.

1.5.1 LPC and sparse LPC encoding

There exists many different encodings using LPC, where the LPC coefficients and error is the basis. But each of these encodings can be implemented with different number of coefficients, LPC-10 standard consists of 10 LPC coefficients others use more. In [Hung Ngo, Mehrub Mehrubeoglu, 2010, p.15] the effect of increasing the number of coefficients effects signal to noise ratio. In this report it is decided to work with 12 LPC coefficients.
As described earlier voiced and unvoiced signals can be encoded different, this is referred to as excitation. But for this report the implementation an excitation model unnecessary since the main goal is to see the difference between the normal and sparse LPC implementation. so the voiced and unvoiced signals encoded as the same, this will lead to poorer sound, and poorer compression ratio. However with the main part being developing and implementing an architecture which can find LPC and sparse LPC coefficients, and compare, the poor sound does not effect the results.
It is decided to use a 16 kHz voiced signal as input, this maybe referred to as wideband audio or High-definition (HD) voice. Different encoders use different frequencies, the 16 kHz is used as a prof of concept for being able to use the best possible frequency, since normal voice is below 14 kHz. Narrowband voice encoding is normally in the 300-3400 Hz range. When taking in the signal it is split up into segments, in this report it is decided to keep each segment at 20 ms with a 10 ms overlap, see figure 1.4. With the overlapping it means that a new segment of 20 ms will be ready for processing for every 10 ms. This decision means that processing of a package should be processed within 10 ms, so no queuing arise. The 16 kHz signal and the 20 ms segments yields a package size of 320 samples.

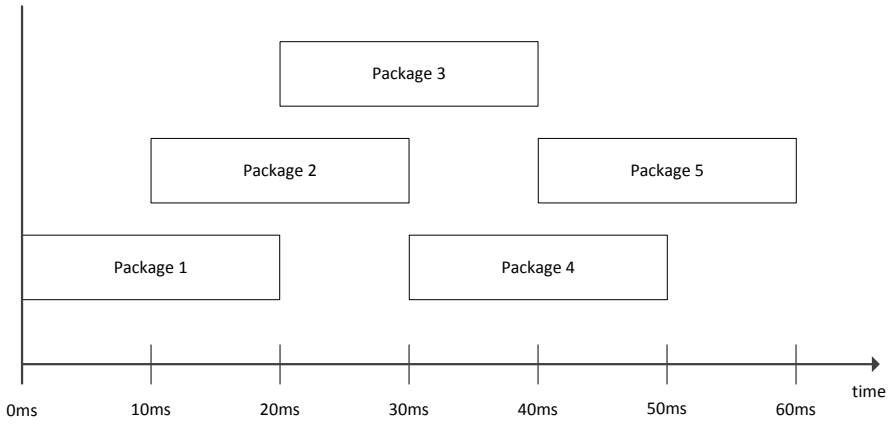

Figure 1.4: The figure shows how the packages containing 20 ms of 320 samples arrive every 10 ms

1.5.2 Preliminary Problem Description

In this section a preliminary problem description is made, which gives the basis for further analysis of the problem.

Preliminary problem statement

This Sparse Linear Predictive Coding idea is an interesting development, which can lead to a preliminary problem statement:

Is it possible to use Sparse Linear Predictive Coding to find the filter coefficients and error, can this be implemented on a FPGA and how is the complexity of the implementation compared to a normal Linear Predictive Coding implementation.

1.5.3 Preliminary Requirement specification

In the previous sections a set of specifications for the setup of the simulation and implementation is stated, this can be summed up to a Requirement specification. First the specifications for the simulation is listed, which then leads to a set of requirements.

General specifications

To make a simulation and analyse how the final specifications needs to be setup, a set of general specifications needs to be specified, which is as follows:

- no excitation model

- 16 kHz voiced signal
- 20 ms segments
- 10 ms overlap
- package size 320 samples
- 12 coefficients

With this set of specifications both LPC and SLPC can be analysed and simulated to find a final set of specifications.

Chapter 2

Linear Prediction Background

The basic of Linear prediction is to find a set of future values of a discrete time signal which is estimated as a linear function of past samples. Linear prediction can be done in different ways, one is to make an all-pole model, which is characterized by a set of a values given by $\{a_1, ..., a_p\}$. This is also known as an Autoregressive (AR) model. Another way is to use an all-zero model, which is characterized a set of b values given by $\{b_1, ..., b_l\}$. This is also referred to as the Moving Average (MA) model. Both models can be combined to a pole-zero model, called the Autoregressive Moving Average (ARMA) model. The most used linear predictive coding scheme uses the all-pole model. The all-pole model leads to the Yule-Walker equations which can be solved in different ways, for example steepest decent method, Newton's method, Cholesky decomposition, etc. A common used method for finding the coefficients, is using the Levinson-Durbin recursion. In this chapter the Linear Predictive Coding scheme will be explained, from taking the speech signal in as a discrete time signal, deriving the Yule-Walker equation, then the all-pole filter coefficients is found using the Levinson-Durbin recursion Makhoul [1975], Daniele Giacobello [2010]

2.1 Linear Predictive Coding

The prediction can then be used to setup a linear predictive coding scheme. It is to divide it into two parts, an analysis block and a synthesis block as shown in figure 2.1. Where the analysis block finds a set of coefficients, given by $\{a_1, ..., a_p\}$ to describe the signal $x(n)$. The block also finds an error signal, which is the difference between the predicted signal and the input signal, given by $e(n)$

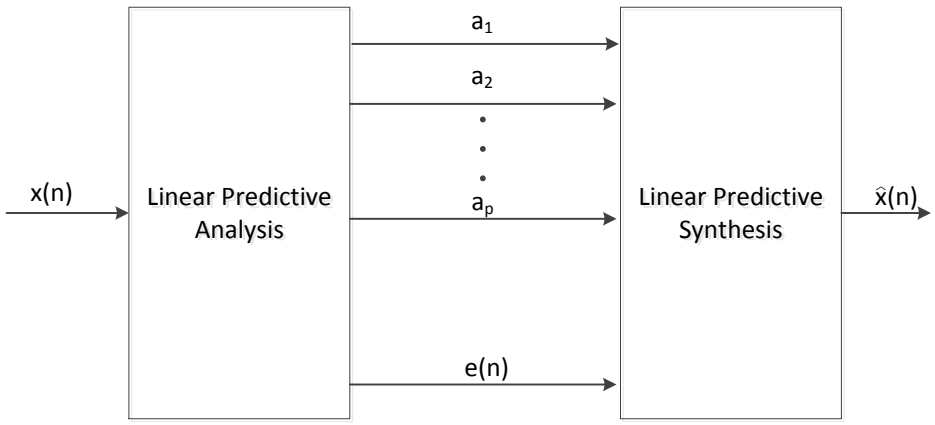

Figure 2.1: Linear predictive coding scheme

The synthesis block then takes in the a's, which are defined as a vector as in figure 2.1 and the error signal $e(n)$, and returns an $\hat{x}(n)$.

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_p \end{bmatrix} \tag{2.1}$$

2.1.1 Linear Prediction Speech Analysis

The idea of linear prediction analysis is that the analysis finds the coefficients of a IIR filter that predicts next values from the current and previous inputs. This is given by an all-pole filter in equation (1.1) which can be rewritten into:

$$\frac{Y(z)}{X(z)} = H(z) = \frac{1}{A(z)} = \frac{1}{1 + \sum_{k=1}^{p} a_k z^{-k}} \tag{2.2}$$

Assuming an unknown $u(n)$, the signal $s'(n)$ can be written as a prediction of a linear weighted summation of the past samples of $s(n)$

$$s'(n) = \sum_{k=1}^{p} a_k s(n-k) \tag{2.3}$$

With this time-domain prediction the error can be found by taking the difference between the prediction and the real signal as:

$$e(n) = s(n) - s'(n) \tag{2.4}$$

where e denominates the current error between the prediction and the input signal, this equation can be drawn as a figure 2.2, where the filter coefficients a_p is used to calculate the error.

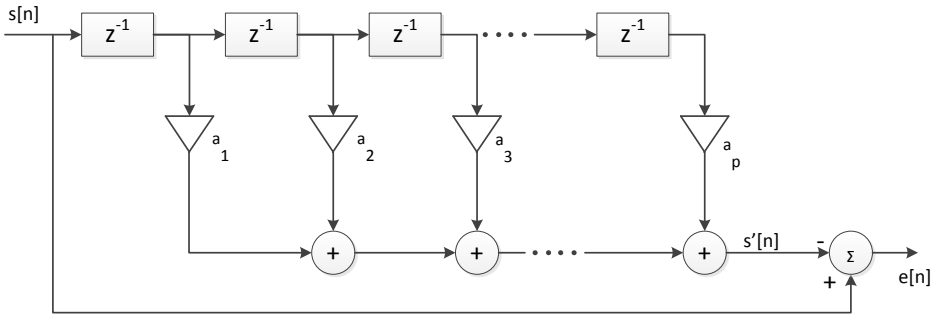

Figure 2.2: The figure depicts the relationship between the error signal $e(n)$ and the input signal $s(n)$ and $s'(n)$

With the error, a summed squared error E is found without specifying the range of summation. This gives an optimization problem, which can be turned into;

$$E = \sum_n e^2(n) = \sum_n \left(s(n) + \sum_{k=1}^{p} a_k s(n-k)\right)^2 \tag{2.5}$$

The error can be minimized by:

$$\frac{\partial E}{\partial a_i} = 0, 1 \leq i \leq p \tag{2.6}$$

where the error is differentiated by the coefficients a_i
About this optimization problem some things are known, it is a convex, linear least squares problem, which gives a solution to the problem. Figure 2.3 depicts a 2 dimensional drawing of the solution to the problem, where the solution is where the curve have minimum, and where the gradient is 0.

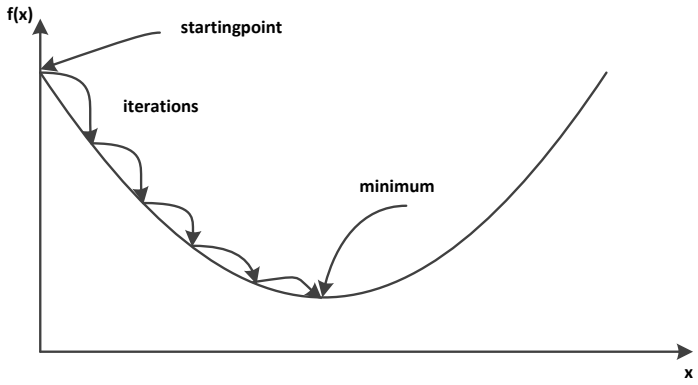

Figure 2.3: 2D drawing of the Least squares solution, solved by gradient decent

From Equation (2.5) and (2.6), Equation (2.7) is obtained, this equation is also known as the normal equation, in least squares terminology [Makhoul, 1975].

$$\sum_{k=1}^{p} a_k \sum_{n} s(n-k)s(n-i) = -\sum_{n} s(n)s(n-i), 1 \leq i \leq p \tag{2.7}$$

The minimum total least squared error from E given by E_p can be obtained by expanding Equation (2.5) and substituting with Equation(2.7) resulting in [Makhoul, 1975]

$$E_p = \sum_{n} s^2(n) + \sum_{k=1}^{p} a_k \sum_{n} s(n)s(n-k) \tag{2.8}$$

Now the range of the summation in Equation (2.5), (2.7) and (2.8) is specified as infinite as $-\infty < n < \infty$ and by speficying the autocorrelation function of the signal s(n) as:

$$R(i) = \sum_{n=-\infty}^{\infty} s(n)s(n+i) \tag{2.9}$$

This autocorrelation function is an even function so $R(-i) = R(i)$, and a toeplitz matrix, meaning that all elements along each diagonal are equal.
Equation (2.7) can be rewritten into:

$$\sum_{k=1}^{p} a_k R(i-k) = -R(i), 1 \leq i \leq p \tag{2.10}$$

and equation (2.8) can be rewritten into (2.11), which is shown in [Makhoul, 1975] to be equal to the gain squared.

$$G^2 = E_p = R(0) + \sum_{k=1}^{p} a_k R(k) \tag{2.11}$$

In reality the signal $s(n)$ is only known for a finite interval so $s'(n)$ is given by windowing the signal $s(n)$ with a windowing function $w(n)$ for a finite interval and otherwise zero as:

$$\hat{s}(n) = \begin{cases} s(n)w(n) & 0 \leq n \leq N-1 \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

the autocorrelation function is then given by

$$R(i) = \sum_{n=0}^{N-1-i} \hat{s}(n)\hat{s}(n+i) \tag{2.13}$$

To solve the linear prediction (2.10) can be expanded into matrix form and rewritten as:

$$Ra = -r \tag{2.14}$$

where R is given by a toeplitz matrix given by:

$$R = \begin{bmatrix} R(0) & R(1) & R(p-1) \\ R(1) & R(0) & R(p-2) \\ \vdots & \vdots & \vdots \\ R(p-1) & R(p-2) & R(0) \end{bmatrix} = X^T X \tag{2.15}$$

and -r is given by:

$$-r = -\begin{bmatrix} R(1) \\ R(2) \\ \vdots \\ R(p) \end{bmatrix} = X^T x \tag{2.16}$$

and to recall a is given by

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_p \end{bmatrix} \tag{2.17}$$

Equation (2.14) is called the Yule Walker equation.
Solving the equation (2.14) by minimizing using least squares leads to a conventional linear prediction problem on the form first by rewriting the equation into matrix form:

$$a = R^{-1}r = (X^T X)^{-1} X^T x \tag{2.18}$$

and then relating equation(2.18) and turning it into a minimization problem:

$$a = (X^T X)^{-1} X^T x = \underset{a}{\operatorname{argmin}} \|x - Xa\|_2 \tag{2.19}$$

The definition of x and X is given as:

$$x = \begin{bmatrix} s(N_1) \\ \vdots \\ s(N_2) \end{bmatrix} \tag{2.20}$$

$$X = \begin{bmatrix} x(N_1 - 1) & \dots & x(N_1 - K) \\ \vdots & & \vdots \\ x(N_2 - 1) & \dots & x(N_2 - K) \end{bmatrix} \tag{2.21}$$

Where the starting and ending points N_1 and N_2 can be chosen in various ways assuming $s(n) = 0$ for $n < 1$ and $n > N$, a common approach is $N_1 = 1$ and $N_2 = N + K$[Daniele Giacobello, 2010].
Equation(2.19) is a rewritten for expression of equation (2.5) and (2.6). This problem can be solved in various ways, a number of ways is, steepest decent, gradient decent, newton method, and other deviations there of Gauss-Jordan elimination etc. But since the minimization problem can be given by the Yule-Walker equation, a fast implementation can be the Levinson-Durbin recursion, which is low complexity and stable according to [GeorgeCybenko, 1980].

2.1.2 Levinson Durbin Recursion

The Levinson-Durbin Recursion is used to solve the Yule-Walker equation given by:

$$Ra = -r \tag{2.22}$$

Where R is a Toeplitz matrix and $a \in \mathbb{R}^p$ and $r \in \mathbb{R}^p$ are vectors as mentioned earlier.
The Yule-Walker equation can be solved by the Levinson-Durbin Recursion as in Algorithm 2.1 for this algorithm to run some initial values has to be given.

Algorithm 2.1 Levinson-Durbin

Require: $E_0 = R(0)$
  for $1 \leq i \leq p$ **do**
    $k_i = -\frac{R(i) + \sum_{j=1}^{i-1} a_j^{(i-1)} R(i-j)}{E_{(i-1)}}$
    $a_i^{(i)} = k_i$
    for $1 \leq j \leq i-1$ **do**
      $a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)}$
    end for
    $E_{(i)} = (1 - k_i^2) E_{(i-1)}$
  end for

A part of the Levinson-Durbin algorithm is the reflection coefficient which is also known as the partial correlation coefficient [Makhoul, 1975] and can be interpreted as the negative partial correlation between $s(n)$ and $s(n+1)$ while s is fixed.
The recursion can be divide into two loops, an outer and an inner loop. The inner loop updates the a coefficients taking $a_j^{(i-1)}$, minus the reflection coefficient k_i multiplied by $a_{i-j}^{(i-1)}$ upto the loop number, j stopping at i The outer loop calculates a new reflection coefficient for k_i by dividing the equation $R(i) + \sum_{j=1}^{i-1} a_j^{(i-1)} R(i-j)$ in the current loop number by the current calculated error. Then updating the reflection coefficient k_i, and then updating the current error $E_{(i)}$, by taking the $(1-k_i^2)E_{i-1}$
An implementation option for Levinson-Durbin is to normalize the autocorrelation coefficients by dividing all $R(i)$ by $R(0)$, this will scale the a coefficients to give a fix point solution, which can be beneficial while implementing the algorithm on hardware.

2.2 Sparse Linear Predictive Coding

When looking into sparse linear predictive coding, The need to look at how LPC is build up and what results a normal LPC implementation gives.
As an normally the Yule-Walker equation is solved given by:

$$Ra = -r \tag{2.23}$$

This equation can be solved recursive by using the Levinson-Durbin Recursion as described earlier, when solving the Yule-Walker equation like that it gives a set of coefficients a_p and an error signal e. The speech signal can be formulated as a continuous signal as:

$$s[t] = \sum_{p=1}^{P} a_p s[t-p] + e[t] \tag{2.24}$$

Where $s[t]$ is the speech signal and $e[t]$ is the error signal earlier mentioned as E which is minimized in the normal LPC scheme.
The normal problem is defined as 2-norm problem also known as a least squares problem, as in equation (2.19). So why is solving the problem as a 2-norm problem so popular: it gives a stable solution and the prediction error results in the Yule-Walker equation, and can be solved using the Levinson-Durbin recursion[Daniele Giacobello, 2010].
The idea of making the LPC implementation sparse is then to solve the same problem as equation (2.19), how ever the sparsity in a problem is measured as the cardinality, which correspond to the 0-norm ($||\dot{}||_0$). The problem with the 0-norm problem is that it is a combinatorial problem which can not be solved in polynomial time, so for implementation this is not variable. Instead the 1-norm is used, as it preforms well as a relaxation of 0-norm linear programming problem[Daniele Giacobello, 2010].

2.2.1 1-norm Sparse LPC

Into the basic problem a regularization criterion γ is introduced into the minimization problem, so that a weight on the error and the coefficients[Daniele Giacobello, 2010] regulate which element to focus the minimization on. This means that we seek to find a solution to the same problem, however instead of finding the normal solution, we seek a solution where the error residual and/or the coefficients of the solution is minimized, leading to lesser data to transmit to describing the signal. So taking the basic minimization problem in equation (2.19) and introducing this criterion yields equation (2.25) recalling the definations of X and x in equation (2.20) and (2.21).

$$\underset{a}{\text{minimize}} \|x - Xa\|_1 + \gamma\|a\|_1 \tag{2.25}$$

with $x \in \mathbb{R}^M$, $X \in \mathbb{R}^{M\times N}$ and $a \in \mathbb{R}^N$.
Where($\|\dot{}\|_1$) denotes the 1-norm and the γ is the regulation criterion which is a constant that decide which part of the problem is put most emphasis on minimizing, eg. the error or the coefficients. In [Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen , 2015] γ is calculated and set to $\gamma = 0.12$
There are different ways to solve this optimization problem in [Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen , 2015] two ways of solving this minimization problem is proposed, the Douglas-Rachford method and the ADMM. By using the ADMM the problem can be rewritten into a recursive implementation much like Levinson-Durbin recursion with small difference.

$$a^{(k+1)} = a_{\gamma,2} - \begin{bmatrix} -\gamma I \\ X \end{bmatrix}^+ (y^{(k)} - u^{(k)}) \tag{2.26}$$

$$e^{(k+1)} = x - Xa^{(k+1)} \tag{2.27}$$

$$y^{(k+1)} = \mathcal{S}_1/\rho\left(\begin{bmatrix} \gamma a^{(k+1)} \\ e^{(k+1)} \end{bmatrix} + u^{(k)} \right) \tag{2.28}$$

$$u^{(k+1)} = u^{(k)} + \begin{bmatrix} \gamma a^{(k+1)} \\ e^{(k+1)} \end{bmatrix} - y^{(k+1)}) \tag{2.29}$$

with $x \in \mathbb{R}^M$, $X \in \mathbb{R}^{M\times N}$ and $a \in \mathbb{R}^N$.
In equation (2.26) the $(.)^+$ denotes a Moore-Penrose pseudo-inverse[Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen , 2015]. The $\mathcal{S}$ in equation (2.28) is a soft-threshold function given by:

$$(S_t(v))_i = sign(v_i)max(|v_i| - t, 0) \tag{2.30}$$

For this algorithm the variable ρ is found empirically found to $\rho = 100$ according to [Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen , 2015].

2.2.2 Lower order

In subsection 2.2.1 sparse linear prediction is introduced, this in general works well for higher order LPC implementations, minimizing the coefficients and the error residual. However for encoding voiced signals, with a lower order implementation,(12 coefficients), the amount of sparsity in the coefficients little to none.
So instead another implementation is proposed, setting the regularization variable to $\gamma = 0$, so the problem becomes

$$\text{minimize} \quad \|x - Xa\|_1 \tag{2.31}$$

This minimization problem minimizes the error residual, with the 1-norm minimization.
Following the [Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen , 2015], this minimization problem can be solved, using the ADMM algorithm as follows:

$$a^{(k+1)} = a_2 - X^+(y^{(k)} - u^{(k)}) \tag{2.32}$$
$$e^{(k+1)} = x - Xa^{(k+1)} \tag{2.33}$$
$$y^{(k+1)} = S_{1/\rho}(e^{(k+1)} + u^{(k)}) \tag{2.34}$$
$$u^{(k+1)} = u^{(k)} + e^{(k+1)} - y^{(k+1)} \tag{2.35}$$

where

$$X^T X a_2 = X^T r, \quad a_2 = X^+ r, \quad X^+ = (X^T X)^{-1} X^T . \tag{2.36}$$

In the system the equation:

$$X^T X \alpha_2 = X^T r \tag{2.37}$$

can be solved with the Durbin algorithm [Gene H. Golub]. Let $z^{(k+1)} = X^+(y^{(k)} - u^{(k)}) = X^+(b^{(k)})$, then $z^{(k+1)}$ can be obtained by solving

$$X^T X z^{(k+1)} = X^T b^{(k)} \tag{2.38}$$

Notice again that $R = X^T X$ is symmetric and Toeplitz and the above system can be solved using efficiently using the Levinson algorithm [Gene H. Golub].

Chapter 3

ADMM Algorithm Analysis

This chapter concerns the analysis of the algorithm ADMM. First Control Data Flow Graph (CDFG) for the algorithm are derived, which is then used to analyse the flow of the algorithm, and how it can be split up. Then the different parts of the algorithm is analysed to investigate different possible solutions for each step.
The ADMM algorithm which is given as follows:

$$a^{(k+1)} = a_2 - X^+(y^{(k)} - u^{(k)}) \tag{3.1}$$
$$e^{(k+1)} = x - Xa^{(k+1)} \tag{3.2}$$
$$y^{(k+1)} = S_{1/\rho}(e^{(k+1)} + u^{(k)}) \tag{3.3}$$
$$u^{(k+1)} = u^{(k)} + e^{(k+1)} - y^{(k+1)} \tag{3.4}$$

which is initialized with $a_2 = X^+x$ given as the normal least squares solution. The variables $y^{(k)}$ and $u^{(k)}$ are initialized as a zero vector, as stated in Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen [2015].
So to make a CDFG the algorithm is split into five parts, where the last four runs in a loop. First part is to find the least squares solution, then run the ADMM algorithm, first find new $a^{(k+1)}$ coefficients, then update the error $e^{(k+1)}$, find new $y^{(k+1)}$ and $u^{(k+1)}$ vectors, then loop back to find $a^{(k+2)}$. The loop runs for a predetermined number of iterations before the algorithm terminates, number of iterations are further described in section 4.
The CDFG for the ADMM algorithm is depicted in figure 3.1, where the five blocks as previous explained are used.

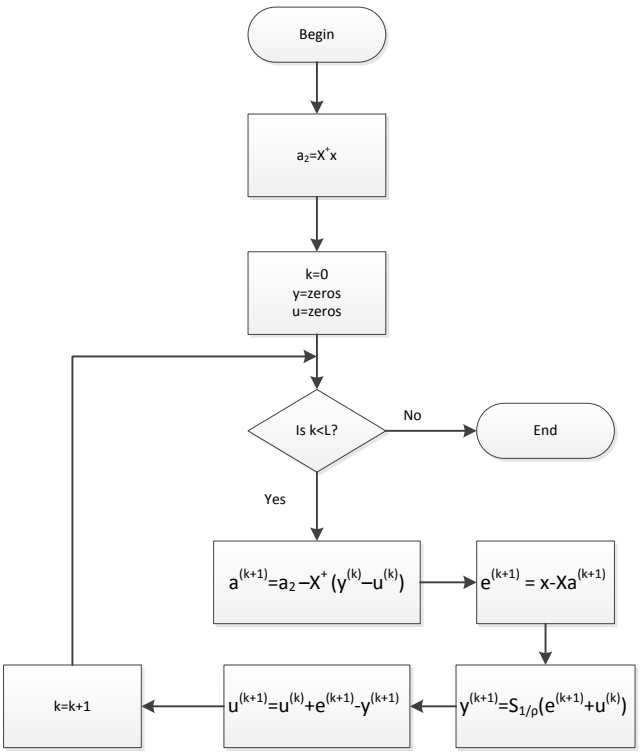

Figure 3.1: CDFG depicting the full ADMM Algorithm.

3.1 CDFG possibilities

Analysing the CDFG there is different possibilities, one possibility is, if it is possible to take out the Moore-Penrose pseudo-inverse matrix of X, out of the loop, as follows: The block with $a^{(k+1)} = a_2 - X^+(y^{(k)} - u^{(k)})$ contains the X^+ which is the Moore-Penrose pseudo-inverse matrix of X which is given by $X^+ = (X^T X)^{-1} X^T$ and is a constant variable, which does not change for every loop. So it can be taken out of the loop and calculated beforehand. In the first iteration of the algorithm where $y^{(0)}$ and $u^{(0)}$ is set to zero, means that the first $a^{(1)}$ is equal to a_2. The last three blocks is depended of the previous step, so they needs to be calculated inside the loop, and they are dependent of each other and can therefore not be calculated in parallel. With these updates a new CDFG can be derived as follows:

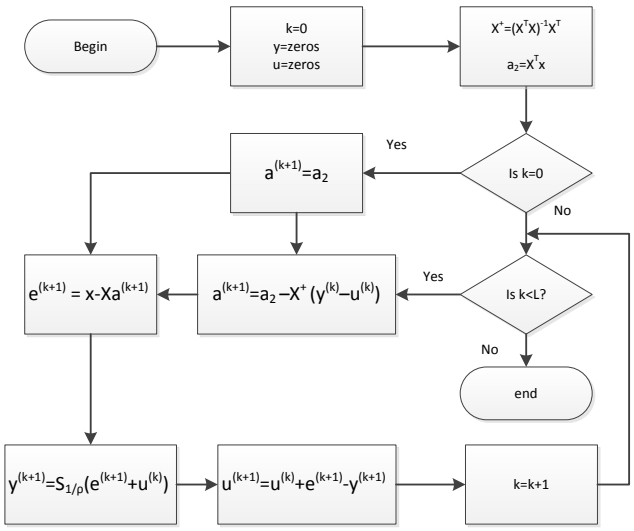

Figure 3.2: CDFG depicting the full ADMM Algorithm, where the calculations which can be taken out of the loop is moved.

The four main blocks in the CDFG are given names and referred to by these names in the rest of this section. The names are $a^{(k+1)}$ is "update coefficients", $e^{(k+1)}$ is "update error", $y^{(k+1)}$ is "update y" and $u^{(k+1)}$ is "update u". The Moore-Penrose pseudo-inverse calculation is given the name "Moore-Penrose", so then the new CDFG is depicted in figure 3.3

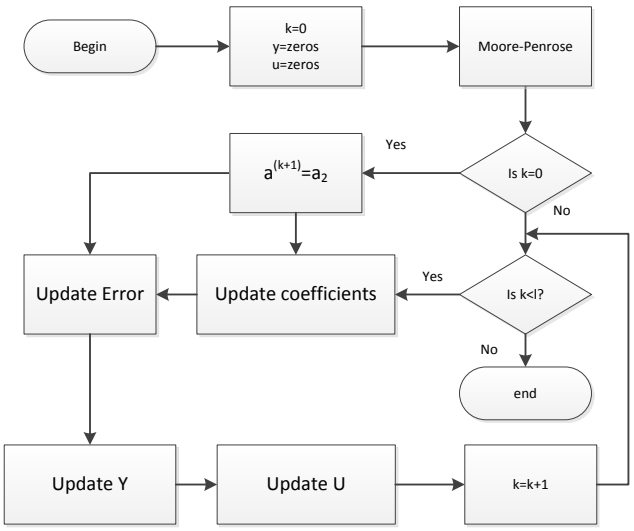

Figure 3.3: CDFG depicting the full ADMM Algorithm, where the blocks in the loop is given names.

Another possible CDFG is to keep the Moore-Penrose pseudo-inverse inside the loop, and calculate it using the Levinson algorithm, as described in 2.2.2, which is a prof that the equation $X^{+}(y^{(k)} - u^{(k)})$ can be solved efficiently using the Levinson algorithm, however this means running the Levinson algorithm in every loop. The CDFG with the Moore-Penrose pseudo-inverse inside the loop, and instead the "Update coefficients", is renamed "Update coefficients version 2"

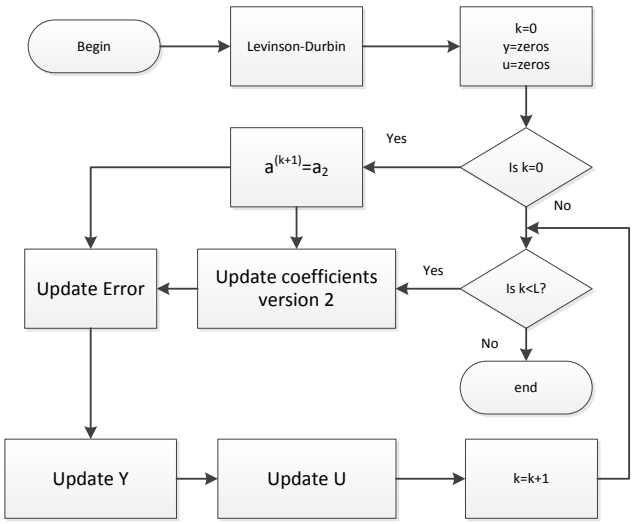

Figure 3.4: CDFG depicting the full ADMM Algorithm, where the blocks in the loop is given names, and the Moore-Penrose pseudo-inverse is inside the loop.

3.2 The different blocks

With the overall CDFG derived as depicted in figure 3.3, a CDFG can be build for the different blocks.

3.3 Moore-Penrose

The "Moore-Penrose" block takes in one input, the X matrix which is a 332 times 12. The "Moore-Penrose" block translated into a CDFG is as follows:

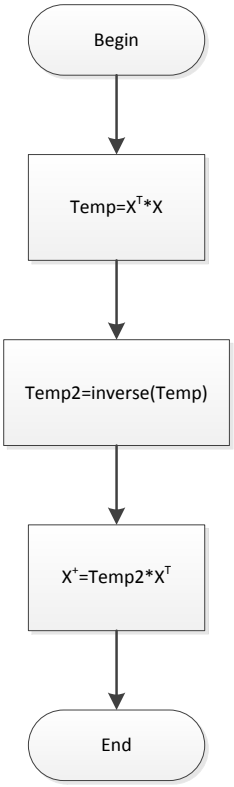

Figure 3.5: Calculating the Moore-Penrose pseudo-inverse matrix of X

The calculation looks strait forward, but it contains a multiplication of X(332x12) X^T(12x332), an inversion of the result which is (12x12), and a multiplication of the X^T(12x332) matrix, which then gives the X^+(12x332). This means that there is many calculations involved in finding the Moore-Penrose pseudo-inverse, and there is a large workload in the inversion of a 12x12 matrix. The main problem is inverting a 12x12 matrix which can be solved by finding the analytical solution, which include finding the determinant and cofactors of the matrix. Knowing that the matrix is toepliz does however give another way of inverting it, as decribed in P. G. MARTINSSON, V. ROKHLIN AND M. TYGERT [2005], which gives a complexity of $O(n \log^2 n)$.

3.4 Update coefficients

The "update coefficients" block is calculated by the following: The a_2 vector which is the least square solution, is subtracted by the result, of the $y^{(k)}$ vector is subtracted

by the $u^{(k)}$, then to be multiplied by the calculated Moore-Penrose pseudo-inverse X matrix. The CDFG for the update coefficients is strait forward when the Moore-Penrose pseudo-inversion is outside of the loop, as depicted in figure 3.6

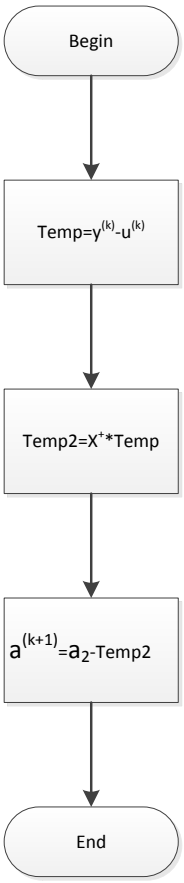

Figure 3.6: Calculating the new a^{k+1} coefficient

However as described earlier this solution finding the Moore-Penrose matrix outside is not a variable solution given the complexity of finding this matrix.

3.5 Update coefficients version 2

The "update coefficient version 2" is rather different then the other version, because instead of having the Moore-Penrose pseudo-inversion outside of the loop. There is two possibilities, either solving by QR-decomposition, or by applying the Levinson algorithm.

3.5.1 QR-decomposition

The calculation can be solved by using the QR-decompositon. From section 2.2.2 we have the following equation:

$$X^T X z^{(k+1)} = X^T b^{(k)} \tag{3.5}$$

For the Moore-Penrose $z^{(k+1)} = X^+(y^{(k)} - u^{(k)}) = X^+(b^{(k)})$ to simplify this $z^{(k+1)}$ is set to α, so by using QR-decomposition and solving for α, means that we have:

$$\alpha = (X^T X)^{-1} X^T b^{(k)} \tag{3.6}$$
$$= X^+ b^{(k)} \tag{3.7}$$

The X matrix can be substituted by a QR decomposition, which means that instead of the X matrix we find a upper triangular matrix R and a matrix Q so $X = QR$. this leads to the following:

$$\alpha = X^+ b^{(k)} \tag{3.8}$$
$$= (R^T Q^T Q R)^{-1} R^T Q^T b^{(k)} \tag{3.9}$$
$$= (R^T I R)^{-1} R^T Q^T b^{(k)} \tag{3.10}$$
$$= R^{-1} R^{-T} R^T Q^T b^{(k)} \tag{3.11}$$
$$= R^{-1} Q^T b \tag{3.12}$$
$$\tag{3.13}$$

if we then set $\overline{b} = Q^T b$ we get the following:

$$\alpha = R^{-1} \overline{b} \tag{3.14}$$

This can then be rewritten into:

$$R\alpha = \overline{b} \tag{3.15}$$

Now we can solve the α knowing that the $\overline{b} = Q^T b$ where b is the vector $y^{(k)} - u^{(k)}$ and the R is an upper-triangular matrix[Gene H. Golub, p. 49]. The equation in (3.15) is then solved by the "back substitution"[Gene H. Golub, p. 89].
This gives a set of equation from 1 to n where n is the number of coefficients:

$$\alpha_n = \frac{\overline{b}_n}{R_{n,n}} \tag{3.16}$$

$$\alpha_{n-1} = \frac{\overline{b}_{n-1} - R_{n-1,n}\alpha_n}{R_{n-1,n-1}} \tag{3.17}$$

and so on.
To start the "back substitution", the input matrix R matrix is needed, so to find R matrices the Gram-Schmidt or the modified Gram-Schmidt method as described in

Algorithm 3.1 Modified Gram-Schmidt

for $i = 1$ to n **do**
  $v_i = a_i$
end for
for $i = 1$ to n **do**
  $r_{ii} = \| v_i \|_2$
  $q_i = v_i / r_{ii}$
  for $j = i + 1$ to n **do**
    $r_{ij} = q_i v_j$
    $v_j = -v_j r_{ij} q_i$
  end for
end for

algorithm 3.1 taken from [Lloyd N. Trefethen, p.51-59] can be used to find the matrix R.
The two methods have a flop count of $2mn^2$ for a m x n matrix[Lloyd N. Trefethen, p.59], and the back substitution method have a cost of n^2
Combining both the Gram-Schmidt and the QR factorization, this means $2mn^2 + n^2$ flops. However, since the X matrix is a constant, so the Gram-Schmidt algorithm can be solved outside of the loop of the ADMM algorithm.

3.5.2 Levinson Algorithm

The calculation can be done by finding the least squares solution to $X^+(y^{(k)} - u^{(k)})$ which is explained in 2.2.2, using the Levinson algorithm. The Levinson algorithm is a little different from the Levinson-Durbin altorithm. Where the Levinson-Durbin algorithm solves the Yule-Walker equation, with a toeplitz matrix, The Levinson algorithm can solve same right hand side problem. The difference between the algorithms are explained in [Gene H. Golub, p. 210-211] where the Levinson-Durbin is mentioned as the Durbin algorithm. The Levinson algorithm used is showed in Algorithm 3.2, where the solution is given by x where the objectiv function solved is given by $Rx = b$, where $x \in \mathbb{R}^p$, $R \in \mathbb{R}^{p \times p}$,$b \in \mathbb{R}^p$ and R is a Toeplitz matrix.
This in then compared with the Levinson-Durbin algorithm in 3.3, where the solution is given by $Ry = -r$, where $y \in \mathbb{R}^p$, $R \in \mathbb{R}^{p \times p}$, $r \in \mathbb{R}^p$ and R is the Toeplitz matrix as earlier described in equation 2.15 given by the vector r.
In the Levinson algorithm, the Durbin algorithm is calculated implicit for every iteration, plus 3 lines of extra calculations, for the μ, v and the x calculation.

3.5.3 Operation count

The Levinson Algorithm operation count is $4n^2$ in [Gene H. Golub, p.211].
As described earlier the operation count using the QR decomposition is calculating $2mn^2$ outside of the loop and n^2 and some overhead of multiplying the $Q^T b$ which

Algorithm 3.2 Levinson

Require: $E_0 = R(0); y(1) = -r(1) = \alpha; x(1) = b(1)$
  for $i = 1 : p-1$ **do**
    $E_{(i)} = (1-\alpha^2)E_{(i-1)}$
    $\mu = \frac{(b(i+1)-r(1:i)^T x(i:-1:1))}{E_{(i)}}$
    $v(1:i) = x(1:i) + \mu y(i:-1:1)$
    $x(1:i+1) = \begin{bmatrix} v(1:i) \\ \mu \end{bmatrix}$
    if $i < p-1$ **then**
      $\alpha = \frac{-(r(i+1)+r(1:i)^T y(i:-1:1))}{E_{(i)}}$
      $z(1:i) = y(1:i) + \alpha y(i:-1:1)$
      $y(1:i+1) = \begin{bmatrix} z(1:i) \\ \alpha \end{bmatrix}$
    end if
  end for

Algorithm 3.3 Levinson-Durbin

Require: $E_0 = R(0)$; $\alpha = y(1) = -r(1)$
  for $i = 1 : p-1$ **do**
    $E_{(i)} = (1-\alpha^2)E_{(i-1)}$
    $\alpha = \frac{-(r(i+1)+r(1:i)^T y(i:-1:1))}{E_{(i)}}$
    $z(1:i) = y(1:i) + \alpha y(i:-1:1)$
    $y(1:i+1) = \begin{bmatrix} z(1:i) \\ \alpha \end{bmatrix}$
  end for

is $(2n-1)m$ inside the loop, the operation count becomes $(2n-1)m+n^2$. Where using the Levinson Algorithm is $4n^2$, this can be written into a table as shown in the following table:

Method	init OpS	OpS	init	1 iteration	40 iterations
Levinson Algorithm	0	$4n^2$	0	576	23040
QR-Decomposition	$2mn^2 + (2n-1)m$	n^2	72708	100	4000
Moore-Penrose	$mn^2 + nlog^2n$	mn^2	3.3233e+004	33200	1328000

It is decided to use the Levinson Algorithm to update the coefficients, since it uses fewer operations then the QR-Decomposition. If the algorithm had to run for a larger number of iterations, it is also seen that the QR-Decomposition uses lesser operations each iteration, so at some point it will become favourable to use that method instead.
A CDFG is drawn for this in figure 7.6 which takes the autocorrelation $r = X^T x$, where X is the matrix, and $x = (y - u)$ [Daniele Giacobello, 2010, p.7] as an input and gives the vector y as an output.
A CDFG is needed to calculate the input r which is depicted in figure 3.8.

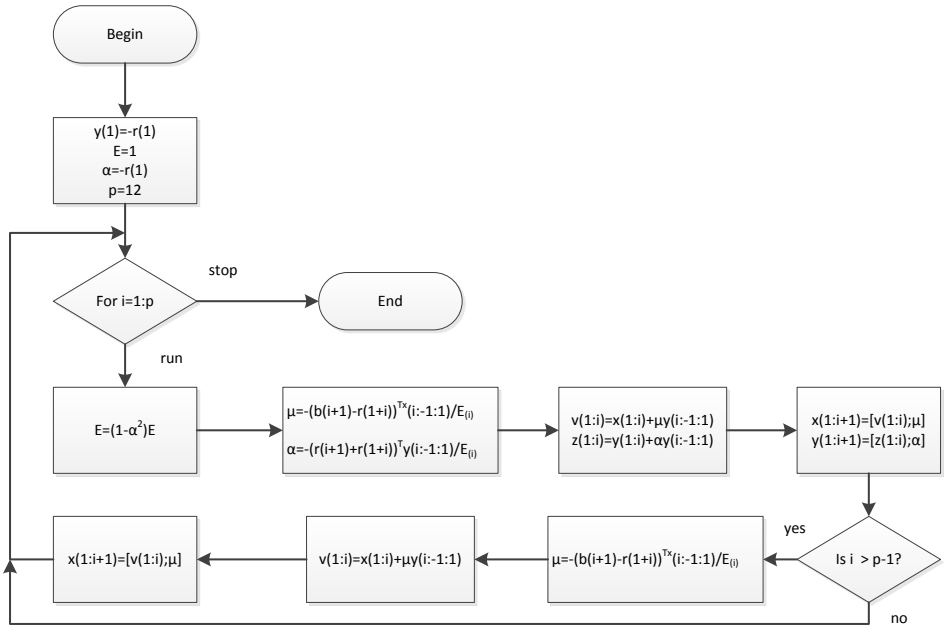

Figure 3.7: CDFG for the Levinson Algorithm

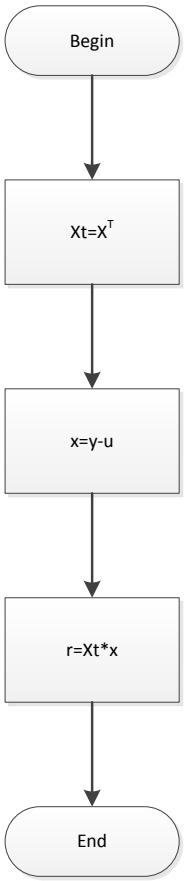

Figure 3.8: CDFG for calculating the input r to the Levinson Algorithm

3.6 Update error

The "update error" block is x(332) vector subtracted by the vector, which is a result of multiplying the X(332x12) matrix by the $a^{(k+1)}(12)$ vector, which then gives the $e^{(k+1)}$ which is 332 variables long, the CDFG is depicted in figure 3.9

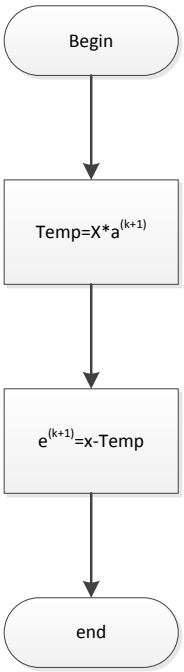

Figure 3.9: Updating the error vector

3.7 Update y

The "update y" block is includes the soft threshold function, which is split into it's own CDFG explained in 3.7.1. It is calculated by adding the error $e^{(k+1)}$ with $u^{(k)}$, and then taking the soft threshold of that, which returns a vector which is 332 long. The CDFG is depicted in figure 3.10.

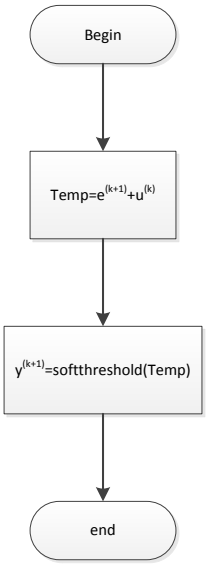

Figure 3.10: Updating the y vector

3.7.1 Soft threshold function

The soft threshold function is a function which looks at the variable, and if it is within the decided ρ value then the variable is set to 0, else the function takes the absolute value of the variable, minus the ρ value, and then multiply with the sign of the variable to begin with. To make a more visual of what the Soft threshold function does, it is depicted in figure 3.11, where the variable t is the value evaluated. In figure 3.12 the CDFG for the soft threshold is depicted, where the input is the vector "temp" in the block "Update Y", which is given as the input, and it returns an vector as an output, after every variable in the vector is evaluated using the soft-threshold function.

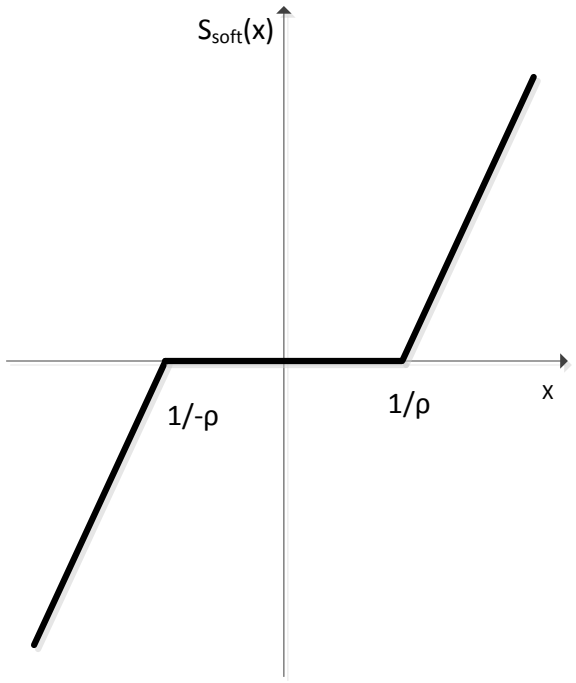

Figure 3.11: Figure depicting how the Soft threshold function works

The CDFG for the Soft thresholdfunction is depicted in figure 3.12.

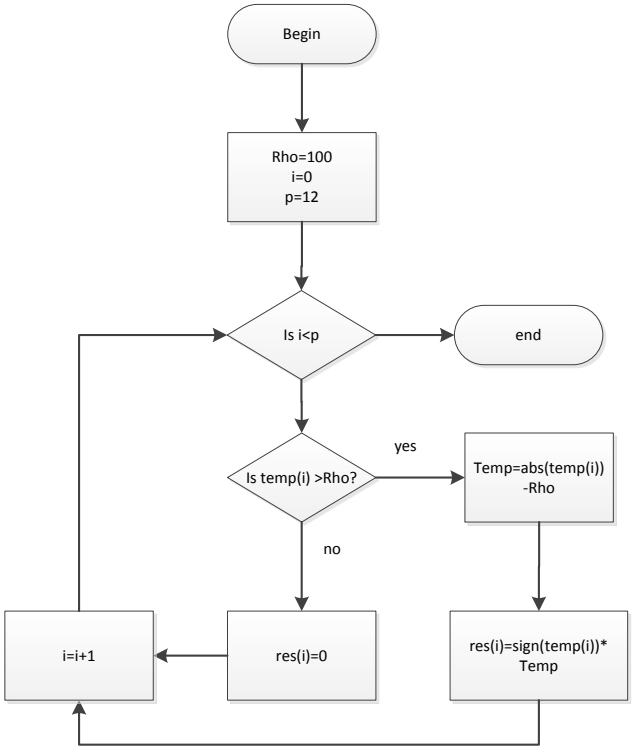

Figure 3.12: CDFG for the Soft threshold function

3.8 Update u

The "update u" block is a simpler block, which adds $u^{(k)}$ with $e^{(k+1)}$, and subtract $y^{(k+1)}$. Which means that it is two vector additions and a vector subtraction, where all vectors are 332 long. The CDFG is depicted in figure 3.13

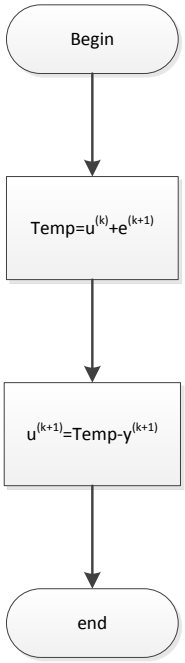

Figure 3.13: Updating the u vector

3.9 Conclusion

This chapter contains the analysis of the ADMM algorithm, finding the flow and possible ways of implementing the algorithm. It was found that while implementing the algorithm, that solving the *Update A*, takes fewest calculations using the Levinson algorithm, for lower number of n, and if the number of n goes to infinite ,using the QR-Decompesition is better. However at $n = 12$ the Levinson algorithm is best.

Chapter 4

Linear Prediction Simulation

In this chapter the linear prediction setup is simulated, using Levinson-Durbin algorithm to find the a_p coefficients, the implementation is simulated together with a implementation of ADMM algorithm, as explained earlier in section 4.1. This is done to check how the outputs are for the different solutions. First a set of test signals are specified, which are used through the simulation setup. Then the output of the respective signals are plotted and the difference between the normal LPC and SLPC coefficients can be examined.
Further studies of the convergence of the ADMM algorithm, and how the error signal changes over a number of iterations. Lastly the Matlab implementation is profiled to verify the bottleneck in the ADMM implementation.

4.1 Test signals

The implementation is tested on 3 different signals.

1. a with a 100 Hz sinus tone
2. a recorded voice with the letter A
3. a random generated signal with low pass filtered noise

All the following plots are done on a frame 20 ms at 16 kHz, 320 samples per frame. First the spectrum and autocorrelation is plotted for each of the 3 signals, where the 100 Hz signal is plotted in figure 4.1, Spoken A in figure 4.2, and last the 4.3.

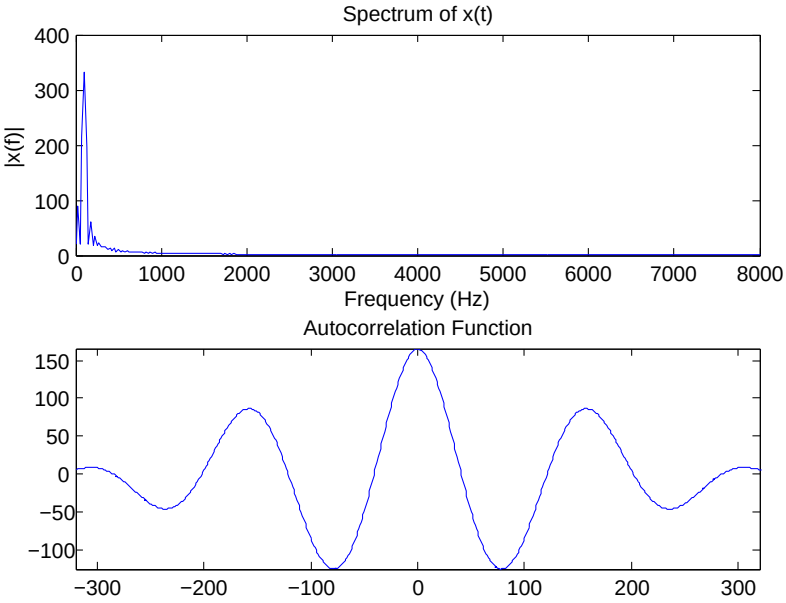

Figure 4.1: Spectrum of a 100 hz sinusoid signal and the autocorrelation function

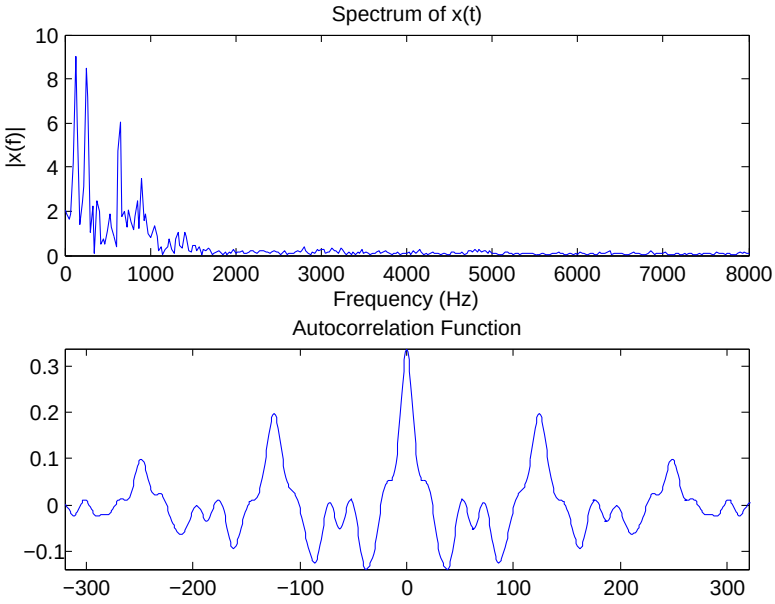

Figure 4.2: Spectrum of the spoken A signal and the autocorrelation function

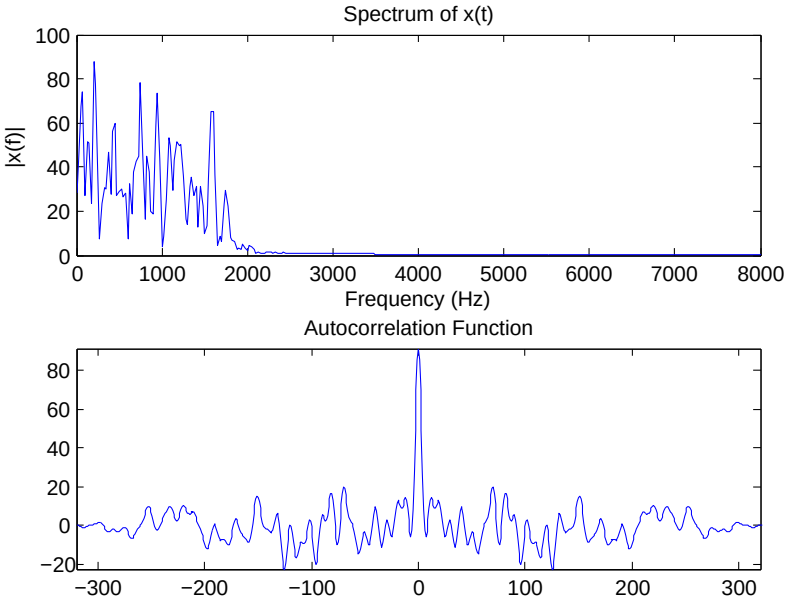

Figure 4.3: Spectrum of the voiced signal and the autocorrelation function

4.2 Output signals

The LPC and SLPC coefficients are then calculated using the LPC and the ADMM algorithm. This gives a set of coefficients, these coefficients are then plotted by taking a Fast Fourier Transform (FFT) of them, and plotting them using the decibel scale. The signals are plotted against the real signal also in the same scale. The plots for the same figures are depicted in figure 4.4, 4.5 , 4.6 with 100 Hz, Spoken A and generated signal respectably.

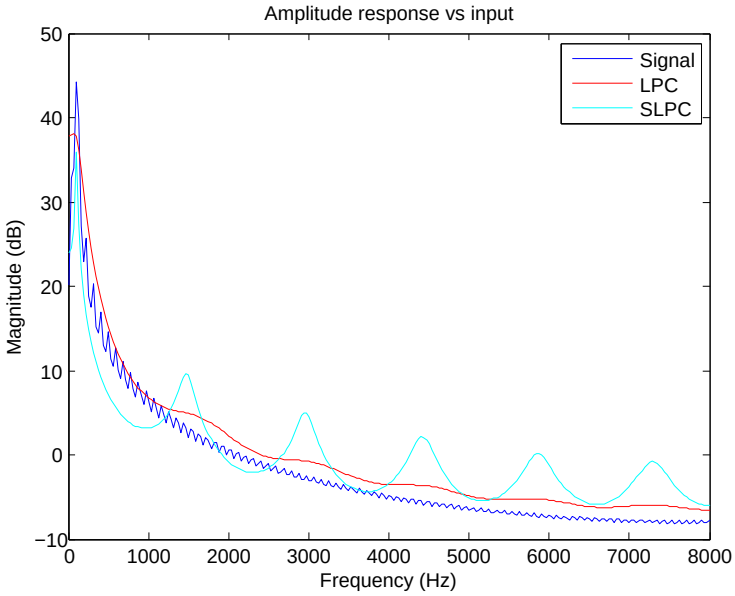

Figure 4.4: Amplitude response LPC coefficients, and the SLPC coefficients plotted against the 100 Hz signal

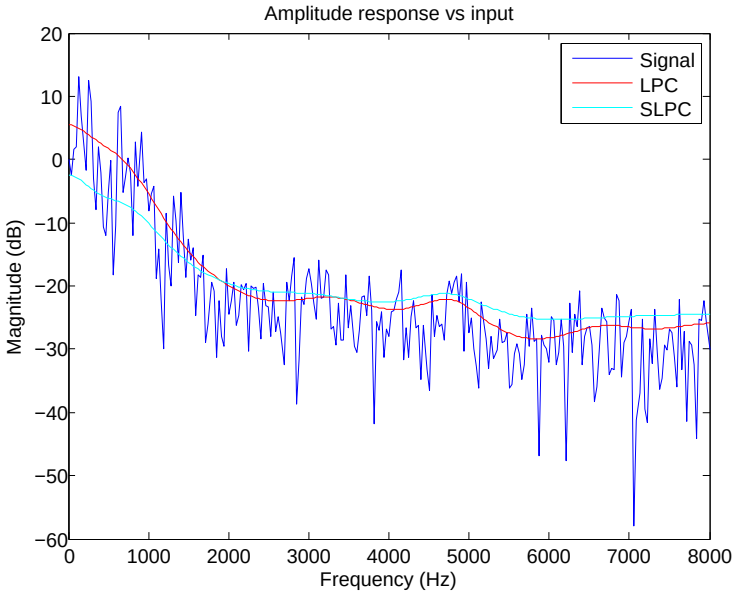

Figure 4.5: Amplitude response LPCcoefficients, and the LPC coefficients plotted against the spoken A

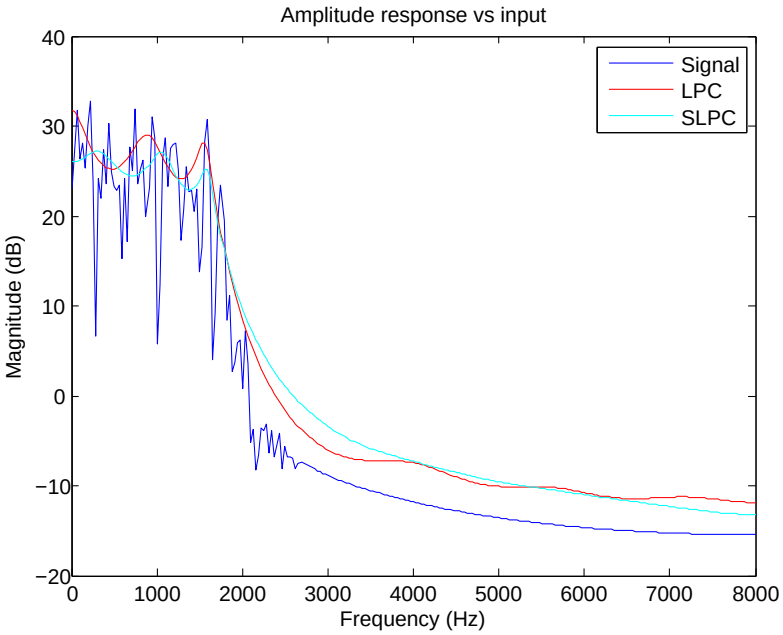

Figure 4.6: Amplitude response LPC coefficients, and the SLPC coefficients plotted against the generated low pass filtered signal

As seen in the figure 4.4, 4.5 and 4.6 the SLPC signal is almost the same as the LPC signal and follows almost, with the exceptions of the 100 Hz signal, some kind of ripple effect occurs. The plot of the spoken A is where both algorithms produces an a signal vector which are most alike, which is good as this is the normal kind of signals packed using speech coding. The signal with the low-pass filtered noise in figure 4.6 is almost also the same with little difference in peaks, seen by the red and cyan lines above the real signal. Also the amplitude seems to be above the real signal, which is possible a implementation error. So according to the simulations, the SLPC implementation with the ADMM algorithm, can find LPC coefficients which approximates the signal, and gives a similar amplitude response as the normal LPC implementation.
To plot these figures it is needed to know the gain of the signal, this can be calculated as shown in equation 4.2, the gain is used to multiply the signal with the given gain to get the correct magnitude.

$$G^2 = R(0) + \sum_{k=i}^{p} a_k R(k) \tag{4.1}$$

4.3 Analysing the ADMM implementation

Here the ADMM implementation of the algorithm is analysed. There is a few things which is of interest, first is to see the Mean Square Deviation (MSD) of the error and coefficients as the algorithm iterates, to see how many iterations it takes before the algorithm becomes converges. Second it is interesting to see how the coefficients and error changes, and if the error vector becomes more sparse by running the algorithm. Last it is interesting to see the runtime of the algorithm in Matlab to see if it gives an insight into which part of the algorithm is more computational demanding.
The analysis of the ADMM algorithm, is done with the low-pass filtered signal, which gives a new random signal for each time the algorithm is called. So that the way the algorithm changes for each iteration, this is done, this MSD runs over 1000 iterations of the algorithm to find the average convergence. The reason for using the low-pass filtered generated signal to test the algorithm, is that it is random generated, so while this generator can easily generate 1000 runs, of 20 ms random over and over, is easier than creating many spoken signals which also should behave random.

4.3.1 Convergence

First the convergence of the algorithm is analysed, this is done by taking the MSD of the error signal. The MSD is calculated by taking the difference between two iterations, squared, and since it is two vectors they are summed up, and the mean is then found by dividing by the number of elements I as stated in equation 4.2.

$$MSD(k) = \frac{1}{I}\sum_{i=1}^{I}(e(i,k) - e(i+1,k))^2 \tag{4.2}$$

$$MSDA(k) = \frac{1}{I}\sum_{i=1}^{I}(a(i,k) - a(i+1,k))^2 \tag{4.3}$$

Then to plot the change over time, a MSD is calculated for every iteration of the algorithm.
Figure 4.7 and 4.8 depicts the MSD for the coefficients and the error, over 60 iterations, it is here seen that both the error and the coefficients stop changing around 20 iterations, but little change still occurs until around 30 for the error and 60 for the coefficients. however this changes for each new input there is, so instead the same set up is implemented, but the same code is run for 1000 times, with a new random input generated every time. Then the mean of the 1000 generated MSD vectors is found which gives a better insight into the average number of iterations needed before the algorithm settles, these figures are depicted in 4.9 and 4.10. It is here noticeable that neither of them never settles completely but around 30-40 iterations they are both fairly low.

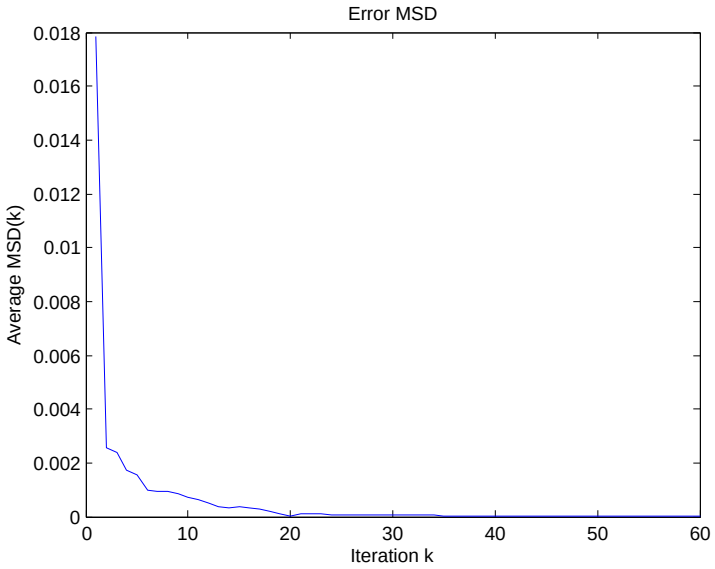

Figure 4.7: MSD of the Error over 60 iterations of the ADMM algorithm

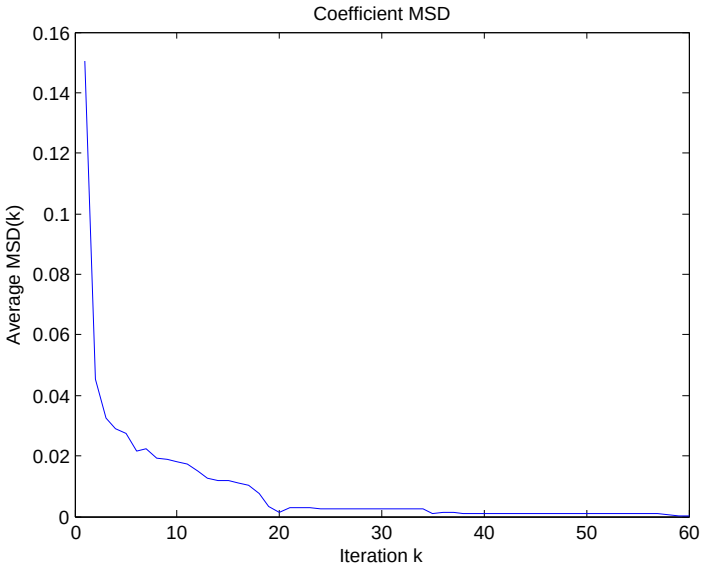

Figure 4.8: MSD of the coefficient over 60 iterations of the ADMM algorithm

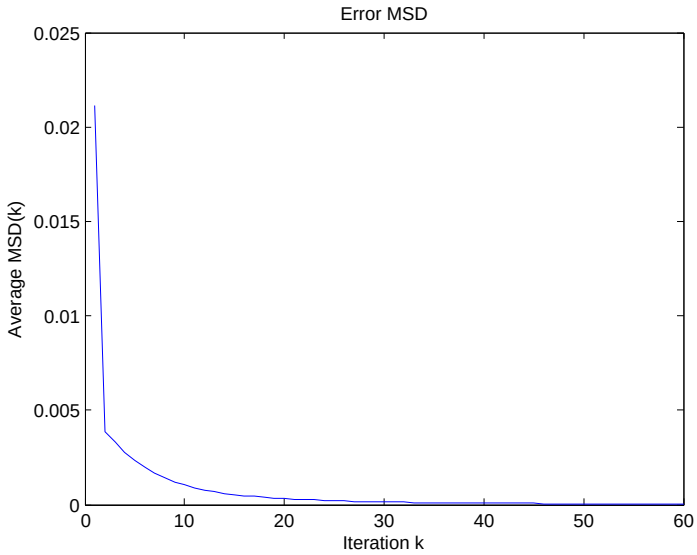

Figure 4.9: Average of the MSD Error over 60 iterations of the ADMM algorithm averaged over 1000 times

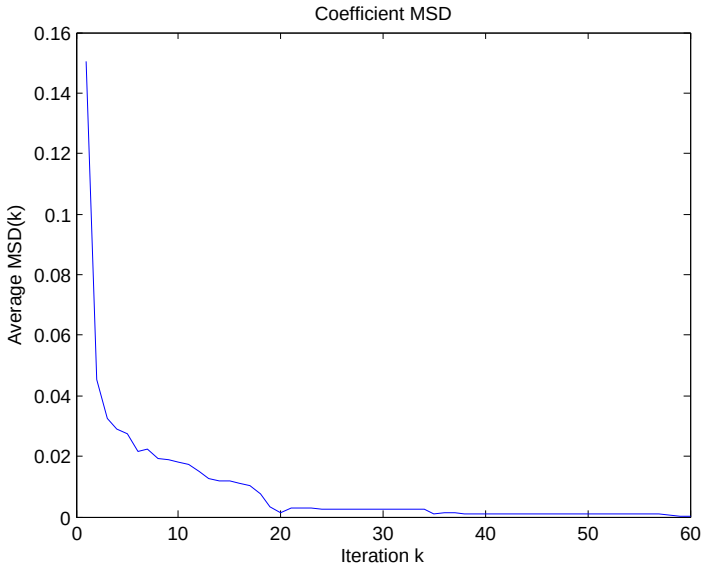

Figure 4.10: Average MSD of the coefficient over 60 iterations of the ADMM algorithm averaged over 1000 times

4.3.2 Changes in signals

The error signal and coefficient vector changes as the ADMM algorithm runs, this is plotted for the low pass filtered noise in figure 4.11, where the same signal is plotted, first in 4.11 where it is visible to see how the 332 error samples change from 1 iteration to the 10 iteration, where the 1 iteration is the normal LPC solution. In figure 4.12 where the 332 error samples change from 50 iteration to the 100 iteration, and becomes more and more sparse.

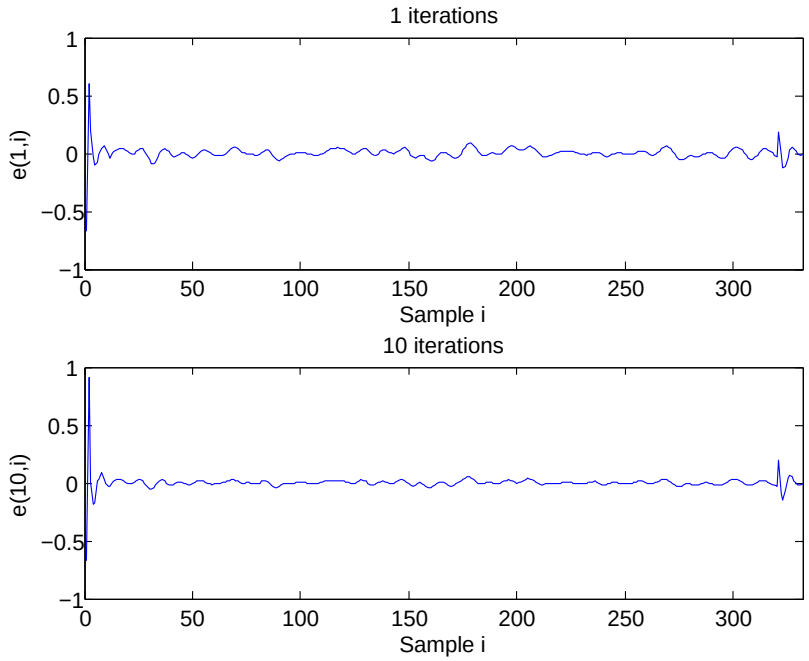

Figure 4.11: Plot showing the error for the 1 and 10 iteration.

Listing 4.1: Sparse LPC matlab implementation

```
a2=X^+x
for k=0:K
    ak = a2 - X\(y - u);
    e = x-X*ak;
    y = S(e+u, 1/rho);
    u = u + e -y;
end
```

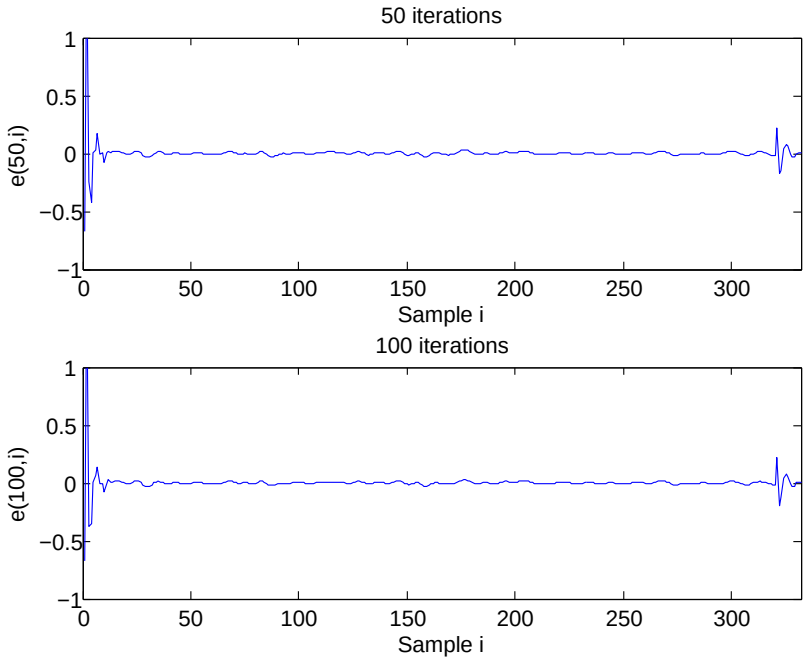

Figure 4.12: Plot showing the error for the 50 and 100 iteration.

4.3.3 Profiling of Matlab implementation

The implemented code in Matlab is implemented as 3, where each line of the four lines are directly translated into Matlab code as shown in Listing 4.1.
This piece of code is then evaluated using the build in Matlab profiler, which can give an insight into the bottlenecks of this implementation. This is done with it in mind that the implementation might be suboptimal and other implementations might yield better results. The profiler results for 1000 iterations of the implementation is as following:

Operation	Time	Times called
ak	0.344s	1000
e	0.02s	1000
y	0.016s	1000
u	0.009s	1000
initial lpc	0.006s	1

It is seen in the table that the calculation of ak, which takes the longest time, which is expected, as this line in this implementation solves a least-squares problem. In the real-time implementation in hardware, the least-squares problem will be solved using the Levinson recursion as described in 3.5.3 as it is the implementation with the lowest complexity.

4.4 Conclusion

In this chapter the ADMM algorithm is simulated and evaluated, and it is shown that it can find a set of LPC coefficients, which yields a sparse error signal. The algorithm convergence is also investigated to asses the number of iterations for sufficient convergence. The convergence of the algorithm shows that at around 30-40 iterations the algorithm hardly alters the coefficients, and the error. So it is decided that the algorithm should stop after 40 iterations as the improvement from running further is too small. The error signal is plotted at iteration 1, 10 and 50 to analyse changes, and there is visible change to sparsity of the error signal. The plotted coefficients is visible also changing, however further knowledges from that is significant small, as the objective function is not focusing on making this signal more sparse. The profiling of the Matlab implementation proves the conclusion from earlier that the least-squares problem in the algorithm is the most time consuming.

Chapter 5

Architecture

5.1 Application

As described in the introduction, this report focus on finding a Register-transfer level (RTL) architecture of a LPC and a SLPC implementation. This can be done in different ways. To begin with a paradigm is set up, that we have an application, which is finding the Linear prediction coefficients. This application leads to a set of algorithms, in this case there is the LPC and SLPC algorithms. These algorithms can then be translated into an architecture. The three different levels can be represented by a figure depicted in 5.1.

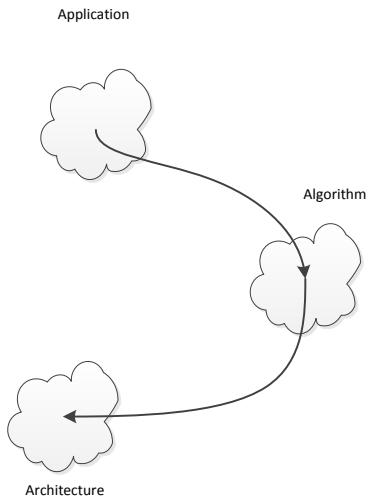

Figure 5.1: The a^3 figure representing the 3 different levels

5.2 Algorithm

In this report the application is decided as being finding the Linear Prediction Coefficients, and the algorithms is the Levinson-Durbin algorithm described in 2, and

the novel ADMM algorithm described in 2.2.1. where both these algorithms can be translated into many different architectures.
This report starts by explaining the application in 1 where the application of finding the Linear Prediction Coefficients to represent the human speech. Then we go into the algorithms in chapter 2, where the two different algorithms are explained. As in the figure 5.2 each level can be further investigated, and iterated back to, which is done in Chapter 3, where the ADMM algorithm is further investigated. This is done because the ADMM algorithm is a newly proposed algorithm for finding the Sparse Linear Prediction Coefficients.

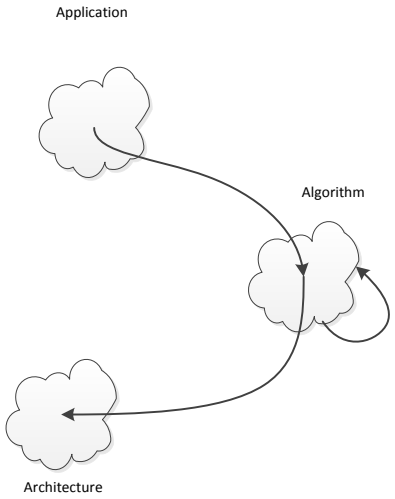

Figure 5.2: The a^3 figure representing the 3 different levels where the algorithm is investigated, by iterating in the algorithm level

When going from the algorithm to making an architecture, there is different ways. In this report it is decided to start with making Control Data Flow graphs, which gives an insight into the flow of the algorithm. With the CDFG in hand, it is possible to translate these into a Precedence graph. However as these Algorithms are complex sets of Precedence graphs has been designed. These Precedence graphs can then be translated into an architecture, this flow is depicted in figure 5.3.

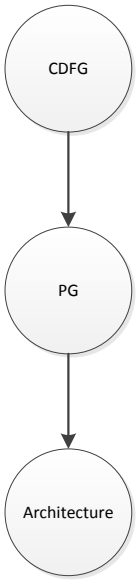

Figure 5.3: Mapping from CDFG into the Precedence graphs, then into an architecture

5.3 Architecture

The mapping of the Precedence Graph (PG) into architecture can be done in many ways. First we need to set a set of paradigm, which is used when making the architecture these are:

- Allocation Select the number of functional units, communication and storage
- Scheduling Introduces time into the precedence graphs, clock cycles for the operation
- Assignment Decides which functional to execute for each operation
- Controller design Control signals, their sequence, and how they are generated

5.3.1 Cost function

To be able to go further with the architecture, some choices needs to be taken, as how to make the design is dependent on the constraint in the architecture.
To identify the constraint in this architecture, the cost function is introduced. Where the design metrics are usually physical area (A), execution time (T), power consumption (P) and noise (N). These metrics are general opposite to each other meaning

that if one is increased others are decreased and vice versa. The cost function is shown in 5.1.

$$C = f(A, T, P, N) \tag{5.1}$$

With the cost function, given that there is a time constraint, no constraint on the physical area. Where the Power consumption given in equation 5.2. Where C_L is the load capacitance, which is given by the hardware, V_{DD} is the voltage of which the hardware runs, and the f_{clk} is the clock frequency.

$$P = C_L V_{DD}^2 f_{clk} \tag{5.2}$$

With the constraint on the time and no areal constraint, and the cost function, the strategy of lowering the cost of running the algorithm is therefore to; lower the supply voltage, as this is squared in 5.2. When lowering the voltage, the f_{clk} is lowered to as a consequence. This means fewer clocks to execute the algorithm. To compensate for the lowered number of clocks, increased utilization of the inherent parallelism is needed.

5.3.2 Design Space

When going from the precedence graphs there is the four different metrics as explained in 5.3.

- Allocation
- Scheduling
- Assignment
- Controller design

These four metrics is then iterated over according to which metric is most important. As described in 5.3.1 the important metric in this report is the time. This means that when going from the PG into an architecture, the first thing found is a scheduling, then allocation. This concept is depicted in figure 5.4 where it is each of the 4 iterations going back to a previous layer depicts how the design can go up a layer to make changes.

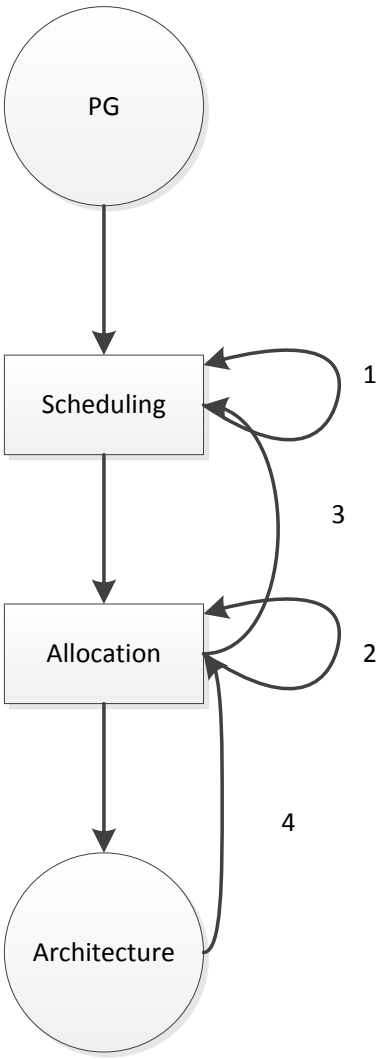

Figure 5.4: How the design can iterate for each improvements in the architecture as the design is updated.

This design iteration can be drawn in the Y-chart [Daniel D. Gajski, p.43] depicted in figure 5.5, which is the chart of how to design a FPGA architecture. We start with a given process structure of the algorithm. With the given structure the RTL behaviour can be described which is given by CDFG, and PG. This can be translated into a RTL structure, given by the FSMD. The then given RTL structure gives the Logic behaviour, with is build up by Algorithmic state machine (ASM) charts [Pappas, 1994, p.348-355], which then gives the logic layout.

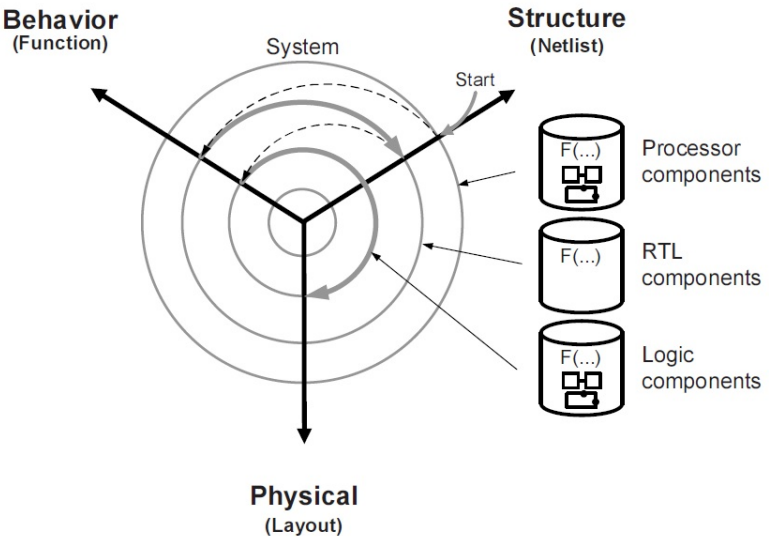

Figure 5.5: The Gajski Y-chart [Daniel D. Gajski, p.43] for system design of an FPGA

5.3.3 Scheduling

As given by the cost function, explained in 5.3.1 the goal in this implementation is to find a scheduling which is as fast as possible with the time constraint and given that there is no area constraint.
This means that there is one rather simple scheduling method which follows these constraints, the method is As Soon As Possible. As Soon As Possible (ASAP) scheduling schedules each operation as soon as they are available, and does not consider any hardware constraint, therefore it is decided to make all the scheduling using ASAP.

5.3.4 Finite State Machine With Data path

With the architecture done, a Finite State Machine with Data path is constructed. This means that we have a Finite state Machine in the Control path as depicted in figure 5.6, Finite State Machine (FSM) then controls the signals to and from the Data Path. The data path contains the hardware logic which calculates the in and outputs.

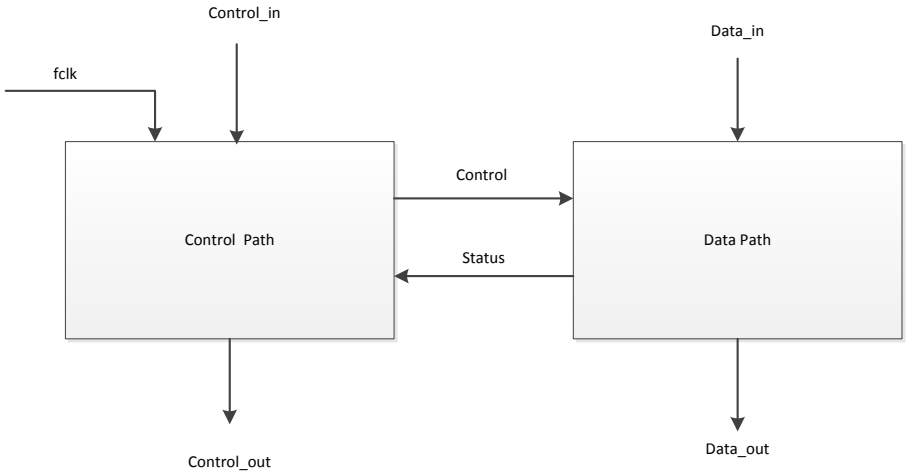

Figure 5.6: Finite State Machine with Data path

Chapter 6

LPC RTL Analysis

In this Chapter a RTL architecture of the Levinson-Durbin recursion will be derived. First CDFG for the algorithm are derived, using that PG are given, and with that a scheduling can be extrapolated, by assuming unlimited hardware. With the precedence graphs and the schedule, a FSMD can be derived.

Algorithm 6.1 Levinson-Durbin

Require: $E_0 = R(0)$
  for $1 \leq i \leq p$ **do**
    $k_i = -\frac{R(i)+\sum_{j=1}^{i-1} a_j^{(i-1)} R(i-j)}{E_{(i-1)}}$
    $a_i^{(i)} = k_i$
    for $1 \leq j \leq i-1$ **do**
      $a_j^{(i)} = a_j^{(i-1)} - k_i a_{i-j}^{(i-1)}$
    end for
    $E_{(i)} = (1 - k_i^2) E_{(i-1)}$
  end for

From Chapter 2.1.2 the Levinson-Durbin is given as an algorithm in 6.1. This algorithm is analysed and divided up into separate parts which can be translated into the CDFG in figure 6.1. The different blocks are different parts of the algorithm, so each block takes an input from last block and gives one output, the loop runs for as many a_p coefficients is wanted.

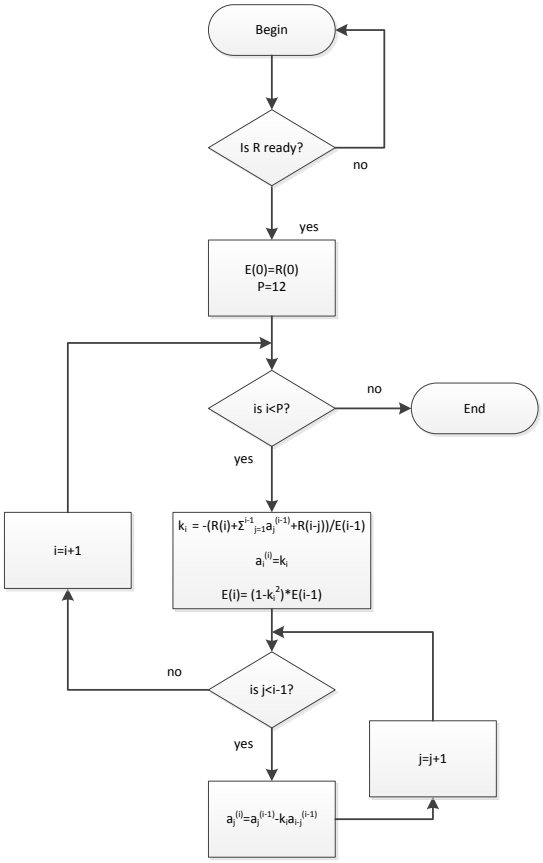

Figure 6.1: CDFG depicting the full Levinson-Durbin Algorithm

This CDFG is the basic version where multiplication operations are done in each block, this can be broken down, so each block contains one mathematical operations. This is done in figure 6.2 where some of the blocks are represented by another CDFG.

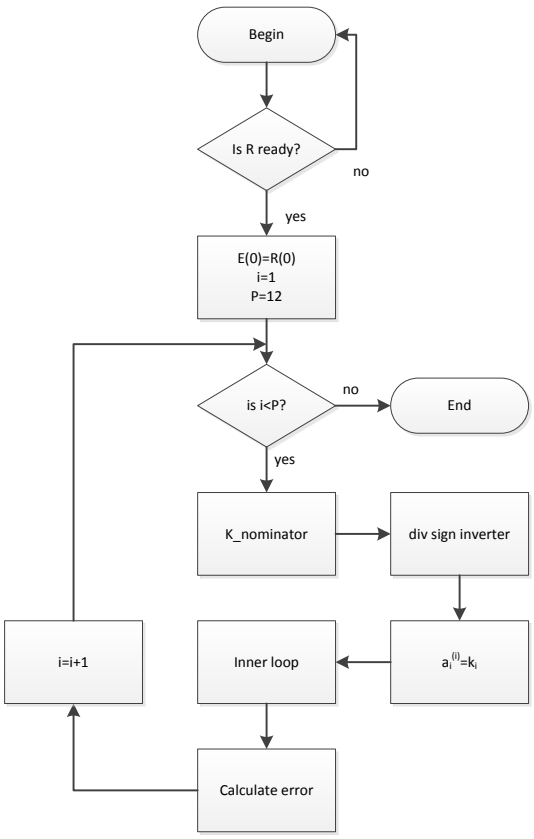

Figure 6.2: CDFG depicting the full Levinson-Durbin Algorithm where the the algorithm is split into different parts, given separate names

The "K_nominator" block is depicted in figure 6.3 which calculates the product of the sum in the algorithm and the addition of R(i): $R(i) + \sum_{j=1}^{i-1} a_j^{(i-1)} R(i-j)$

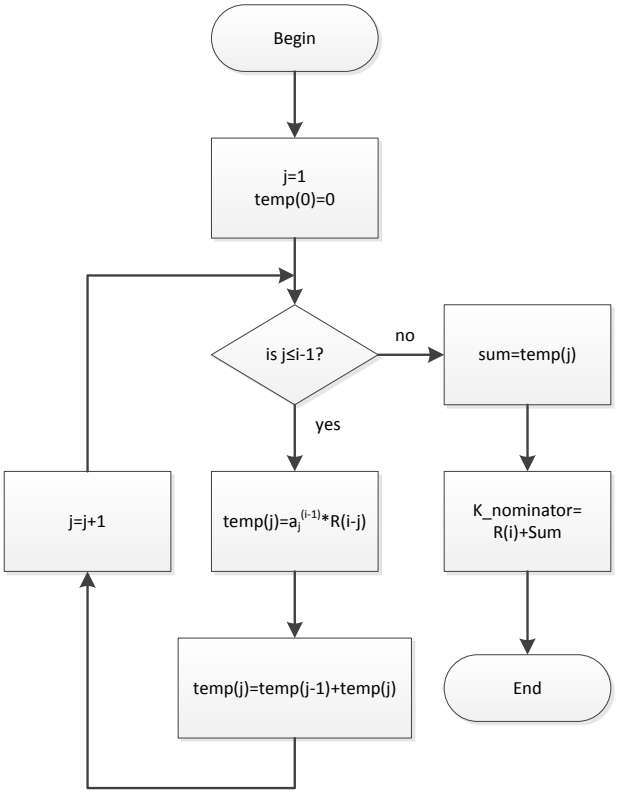

Figure 6.3: CDFG depicting K_nominator

The block "div sign inverter", divides the K_nominator by the error $E(i-1)$, and invert the sign of the result, this gives the result of k_i in the algorithm and is depicted in figure 6.4.

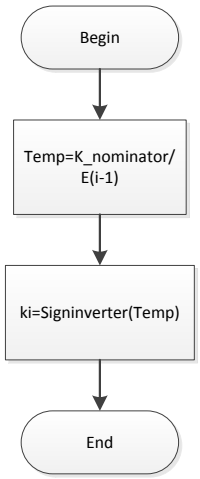

Figure 6.4: CDFG depicting the division of the error E(i-1) and the sign inversion

The block "Inner Loop", is depicted in figure 6.5, and this is where the calculation: $a_j^{(i)} = a_j^{(i-1)} - k_i a_{i-j}^{(i-1)}$ is done.

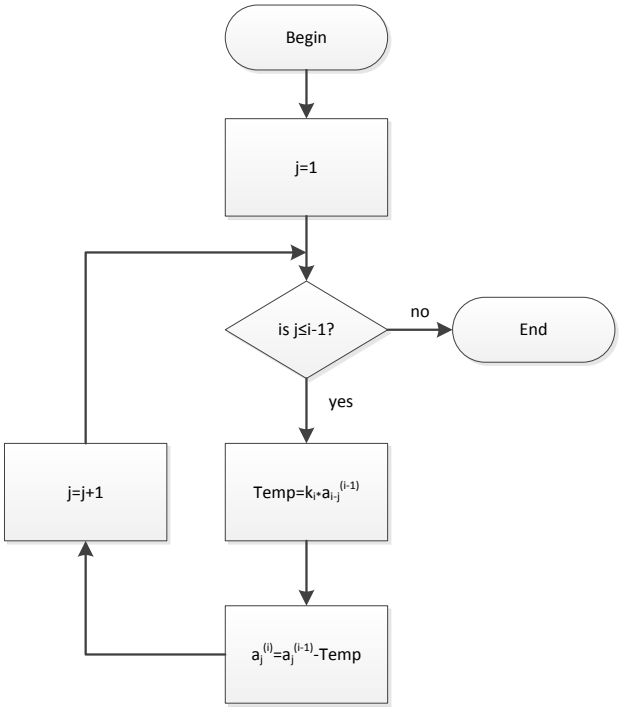

Figure 6.5: CDFG depicting the inner loop of the algorithm

The last block "Calculate Error", is where the new error is calculated, which is depicted in figure 6.6, showing the calculation: $E_{(i)} = (1 - k_i^2)E_{(i-1)}$.

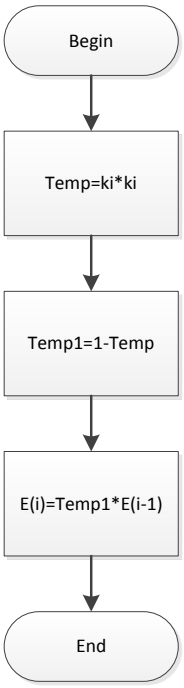

Figure 6.6: CDFG depicting the error calculation

6.1 Data Flow Graphs

Analysing the CDFG in figure 6.3, it is obvious that there is just one overall loop, and that the inner loop possible can be unravelled. The calculate error and div sign inverter block is simple, and independent of the inner loop. This leads to the Data Flow Graph (DFG) depicted in figure 6.7 and 6.8.

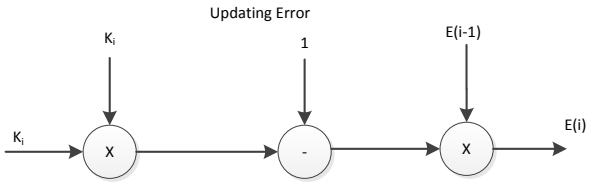

Figure 6.7: Updating the error using the current k_i and the last error $E(i-1)$

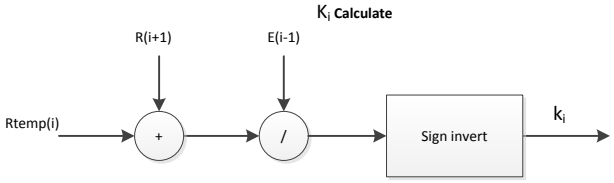

Figure 6.8: Dividing by the Error and inverting the sign k_i value

The first calculation of "K_nominator" is independent of the "inner loop", because the summation in the nominator goes to-1. In the next iteration the a_j^{i-1} which is equal to the last calculated k_i, this is multiplied by $R(1)$ and added with $R(i+1)$ as depicted in figure 6.9.

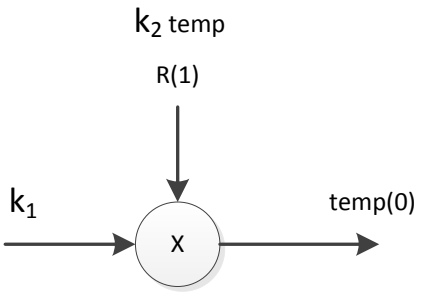

Figure 6.9: k'th temporary value used to calculate k_2

Now the inner loop is calculated and the result of the inner loop is used in the next "K_nominator" block. So these blocks become dependent of each other and therefore the following DFG shows the combination of the "K_nominator" and the "inner loop" as they expands for each iteration of the algorithm.
So the calculation depicted in figure 6.9 is always done, combined with a set of DFG, these DFG are depicted in figure 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.17, 6.18, 6.19, where each figure is named k_i $temp(2-11)$

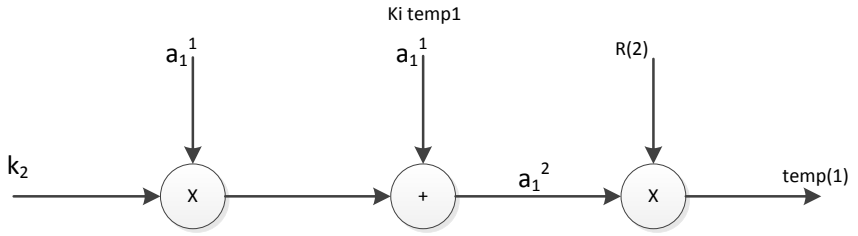

Figure 6.10: k'th temporary value used to calculate k_3

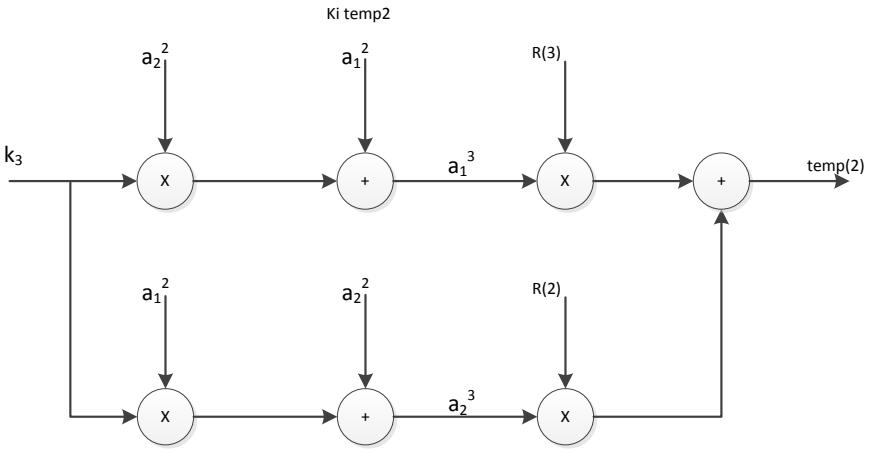

Figure 6.11: k'th temporary value used to calculate k_4

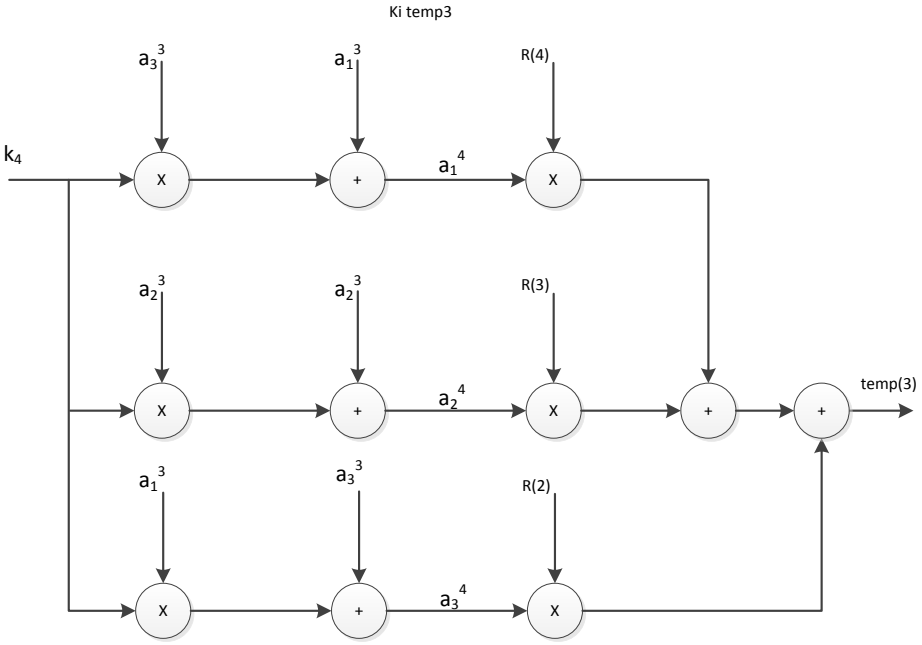

Figure 6.12: k'th temporary value used to calculate k_5

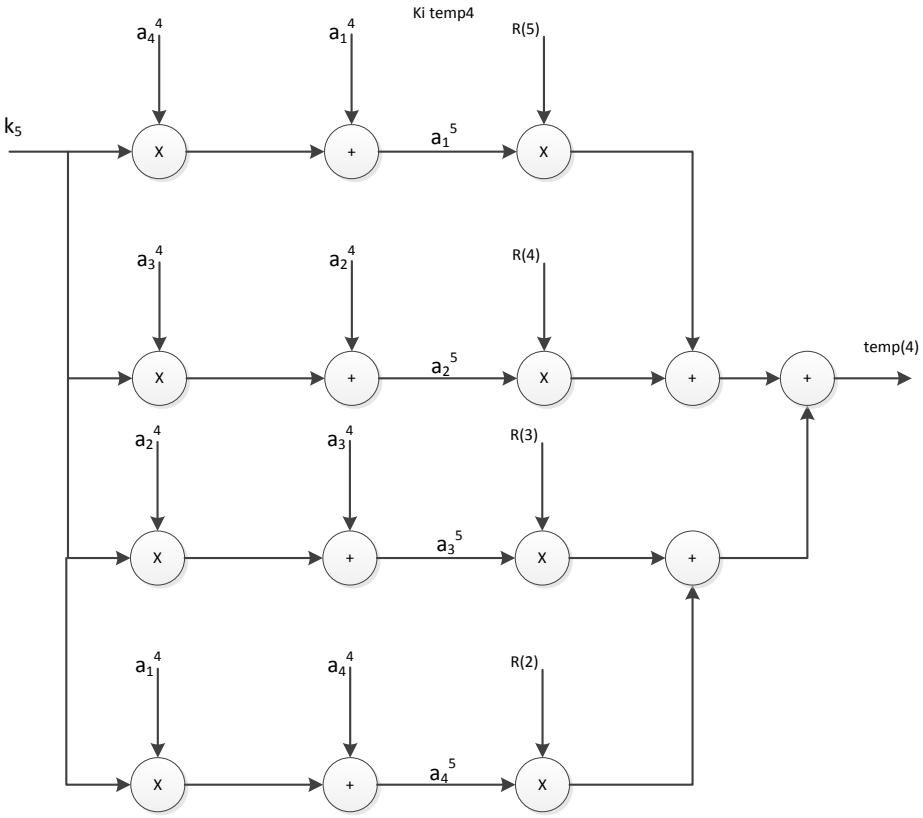

Figure 6.13: k'th temporary value used to calculate k_6

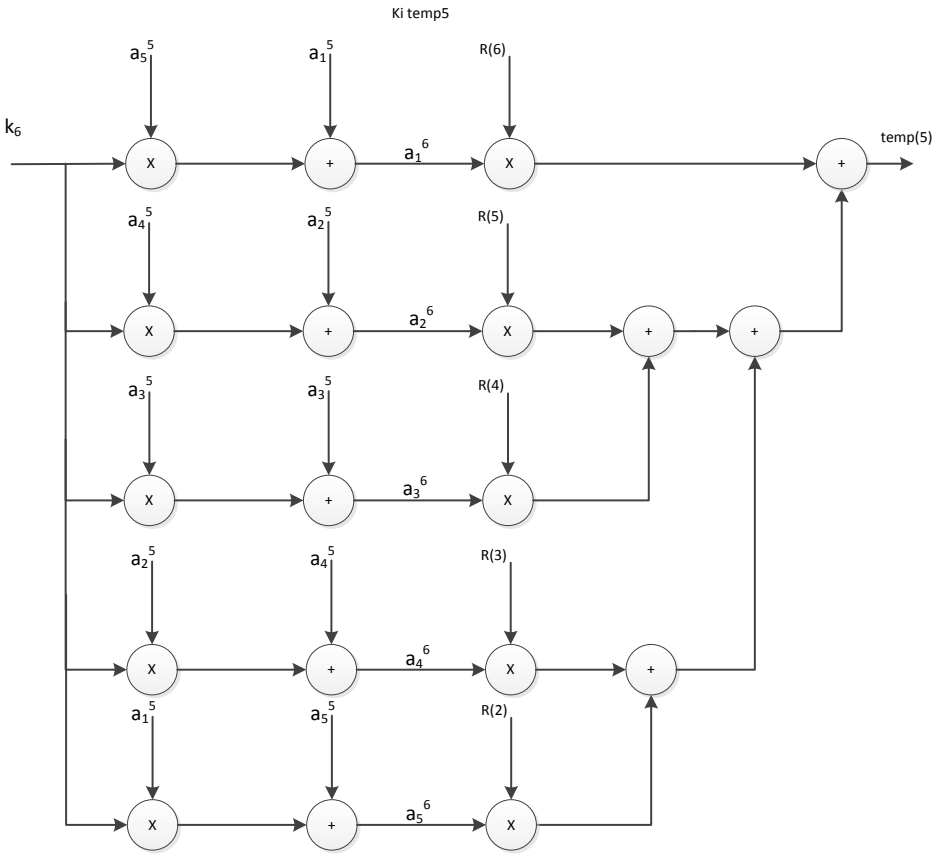

Figure 6.14: k'th temporary value used to calculate k_7

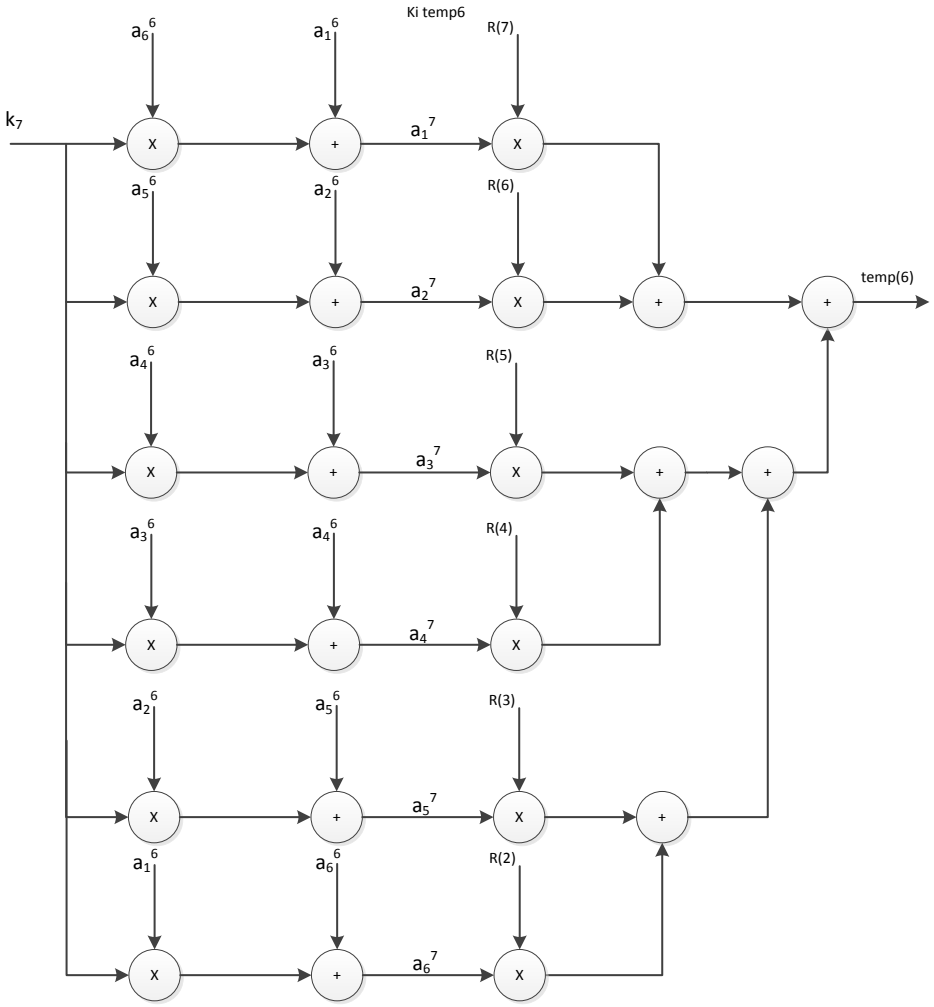

Figure 6.15: k'th temporary value used to calculate k_8

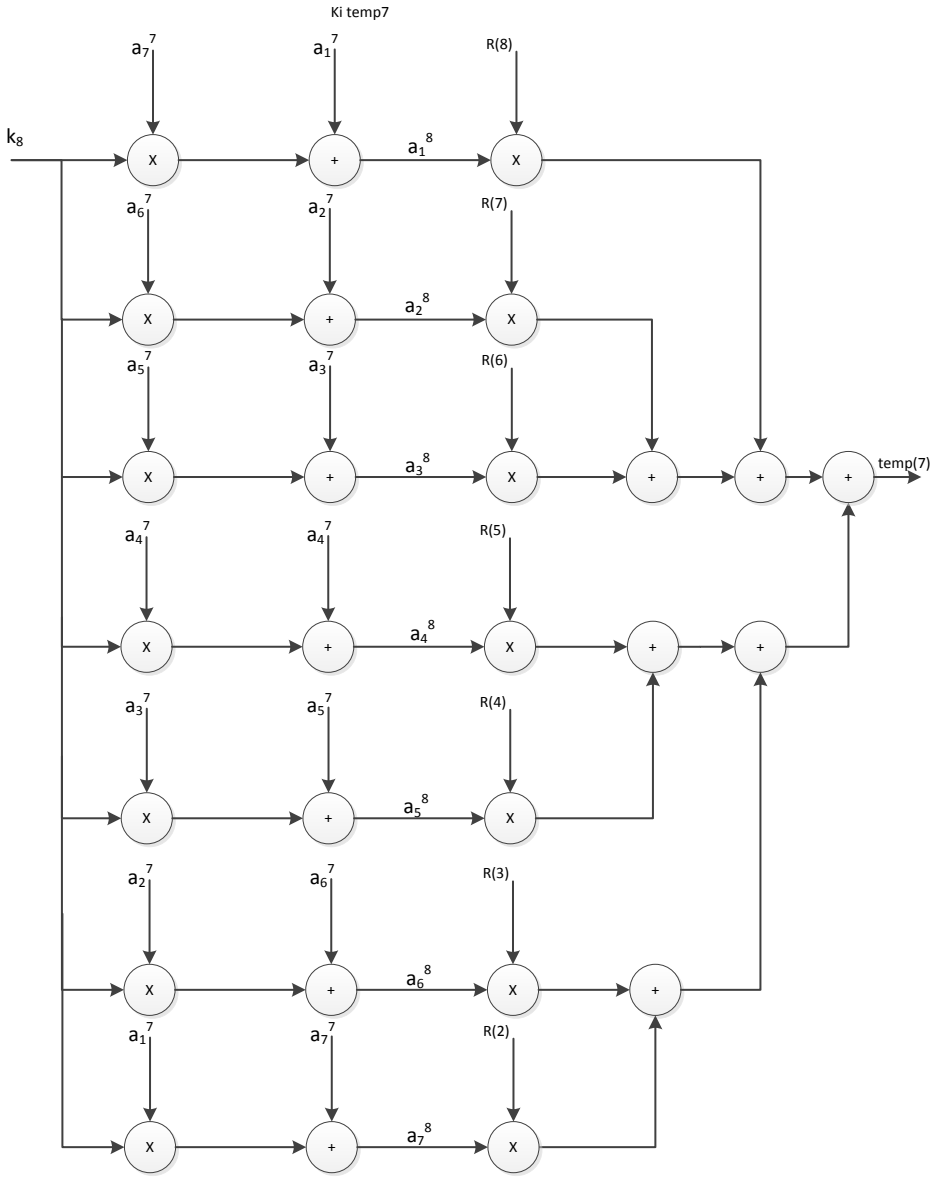

Figure 6.16: k'th temporary value used to calculate k_9

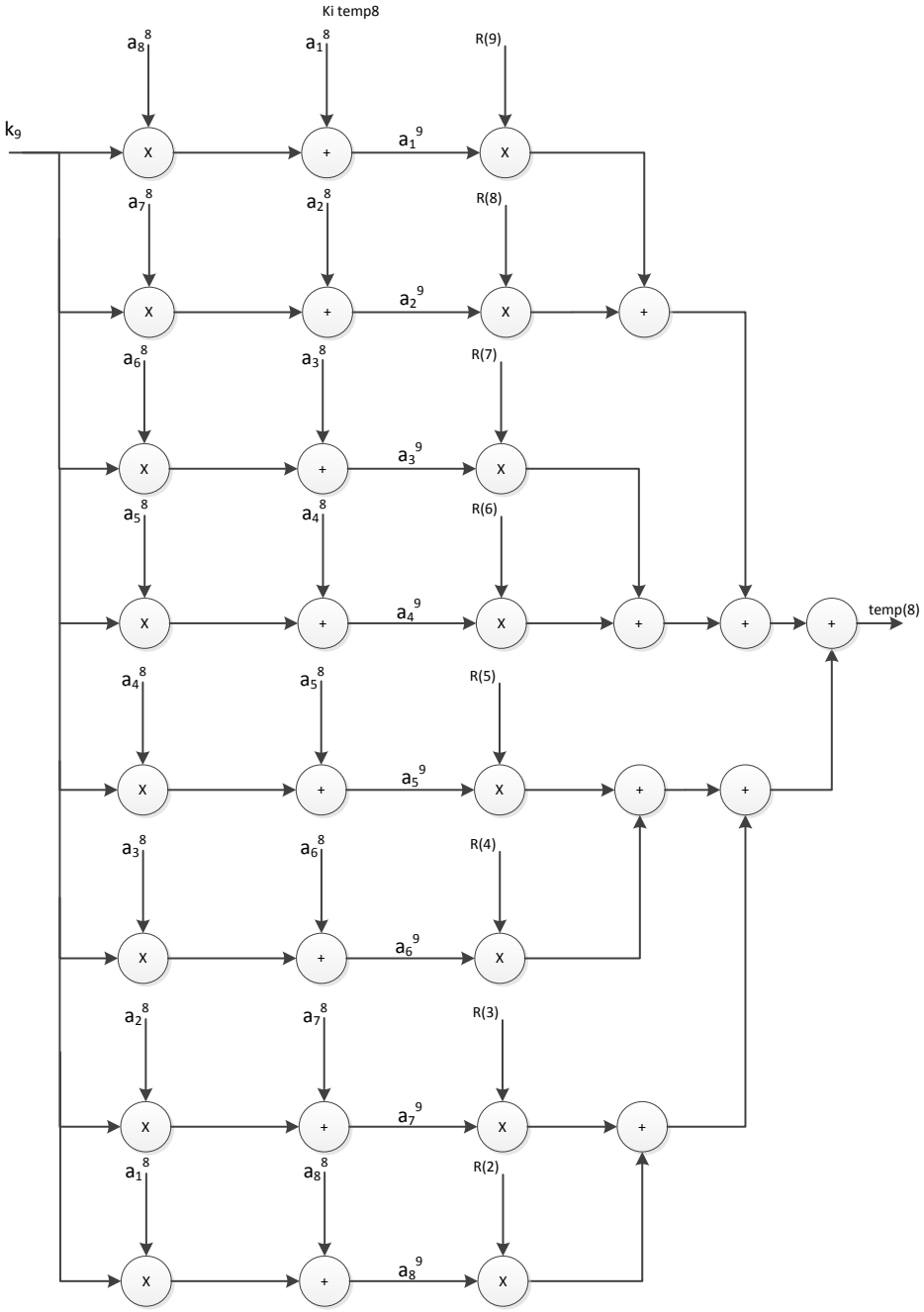

Figure 6.17: k'th temporary value used to calculate k_{10}

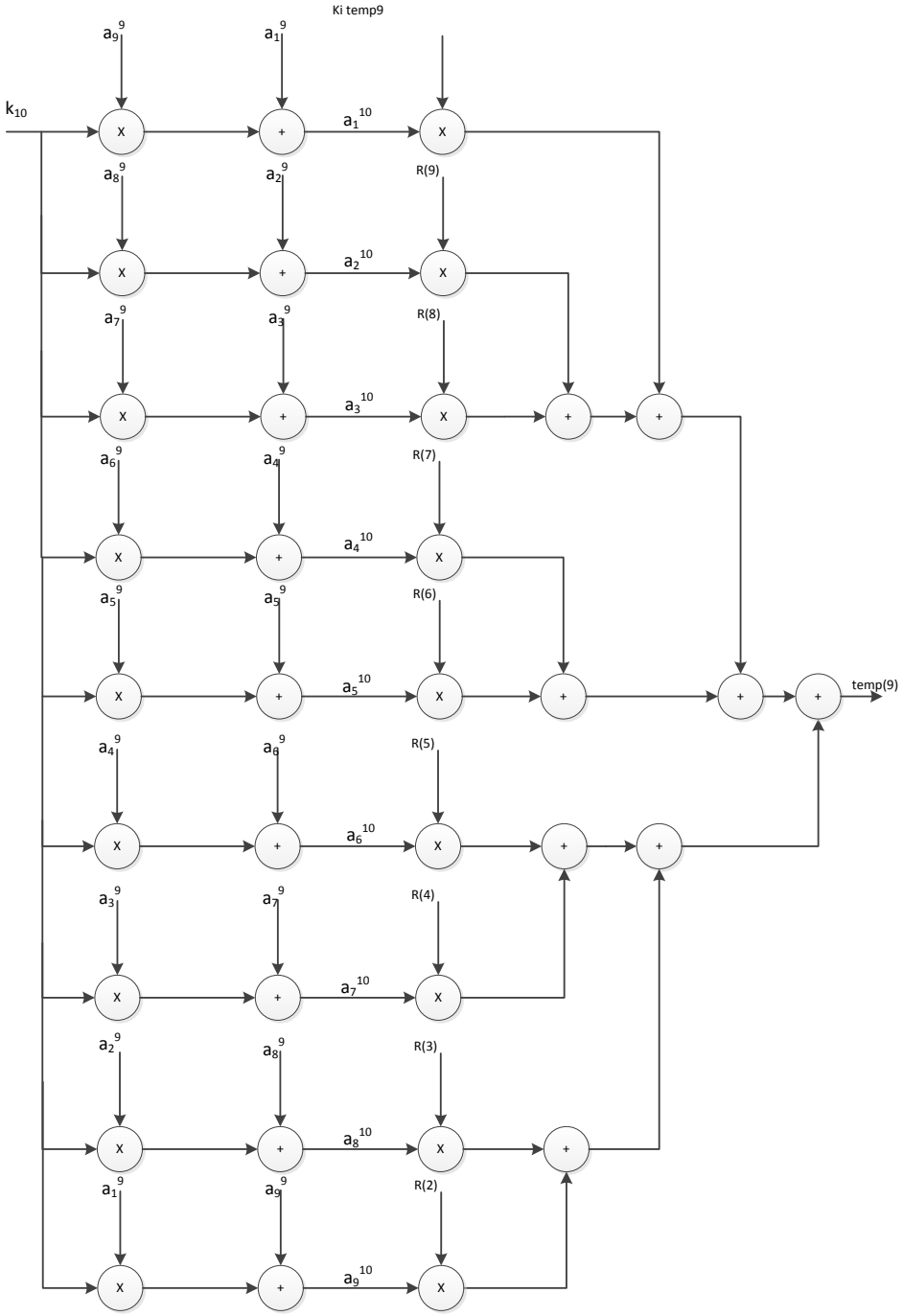

Figure 6.18: k'th temporary value used to calculate k_{11}

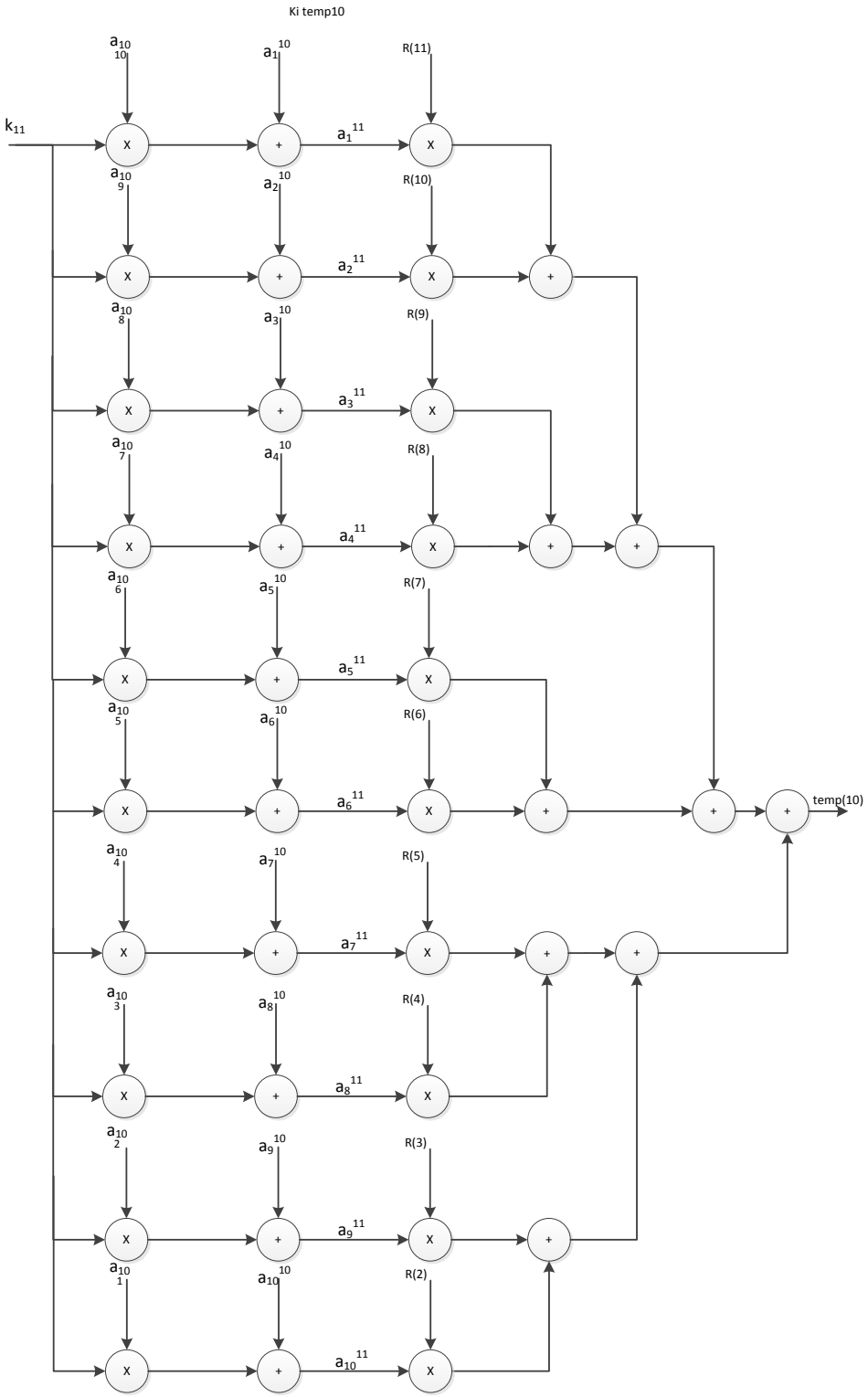

Figure 6.19: k'th temporary value used to calculate k_{12}

The result from the $k_i temp$ block is added with the result the $k_i temp(i)$, as depicted in figure 6.20 which then can be send to the k_i calculate.

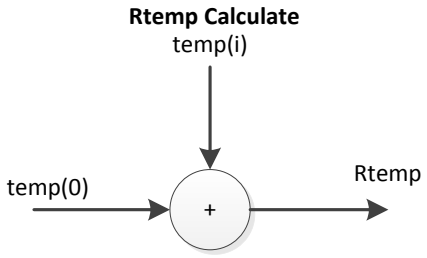

Figure 6.20: R'th temporary value used to calculate k_2

6.2 Precedence Graphs

In this subsection the PG for each part of the Levinson-Durbin algorithm are presented. For the Precedence graphs there is no hardware limits.
As all the DFG presented earlier are homogeneous DFG, where there is one input and one output to each operation, and there is no feed back, the precedence relations, and parallelism, can be seen immediately. So in this subsection the DFG's for the "K_nominator" and "inner loop", named k_i $temp(2-11)$ are analysed to find out how the precedence are, as they are the only DFG with inherent parallelism, and the degree of parallelism changes over time.
The first figure is 6.21 where there is no parallelism. The next figure 6.22 is the next iteration, where there is some parallelism, plus there is an added summation. Jumping ahead to the last k_i $temp11$ in figure 6.23 shows that for each iteration a new set of multiplication subtraction and multiplication is added, and additions are added to sum up the result.

Figure 6.21: Precedence graph for calculating k_i temp2

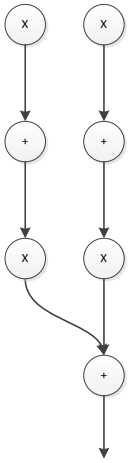

Figure 6.22: Precedence graph for calculating k_i temp3

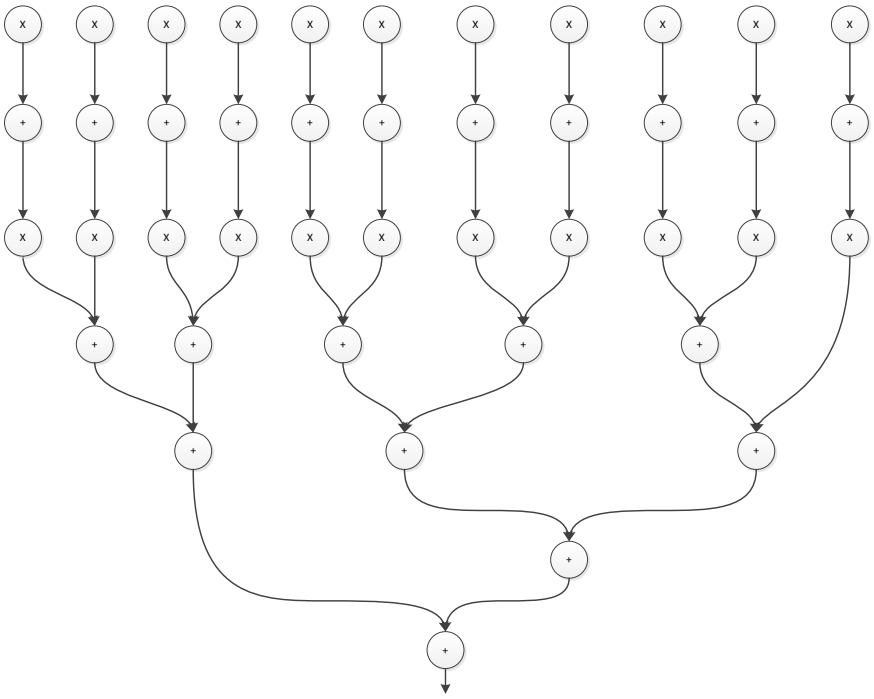

Figure 6.23: Precedence graph for calculating k_i temp11

So as seen with the precedence graphs for k_i temp(2-11) that the summation of these calculations means that the further the algorithm runs the more time and hardware, it takes in each iteration. Where the 3 first calculations is the same for just adding another column for each iteration. So to simplify this a block can be defined, containing a multiplication, a Addition and a multiplication, this block is then called MAM, and depicted in figure 6.24.

Figure 6.24: PG for the MAM block

6.3 Scheduling

In this section time is added to precedences graphs, as C-steps, so a preliminary estimation of how many C-steps it takes to run the algorithm, with unlimited hardware, and each operation taking 1 C-step. The scheduling method used for this section is ASAP scheduling, as the primary goal for the scheduling, is to execute the calculations as fast as possible since the execution is time sensitive. With the scheduling the clock cycle of the processor can then reduced so the calculation is done within the time limit.
ASAP scheduling for k_i calculate is depicted in figure 6.25, which contains 3 c-steps.

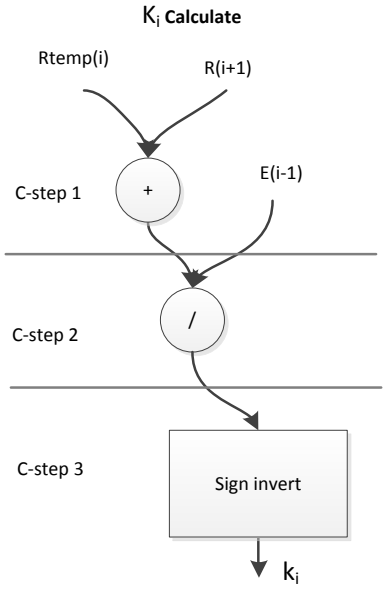

Figure 6.25: Scheduling for the calculation of the k_i value.

In figure 6.26 the Scheduling for the error update is depicted, which contains 3 c-steps.

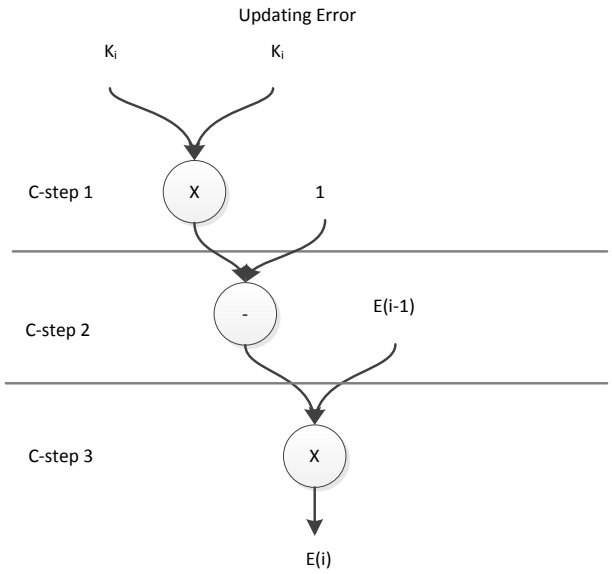

Figure 6.26: Scheduling for the error update.

Figure 6.27 depicts the k_i temp2, which contains 3 c-steps.

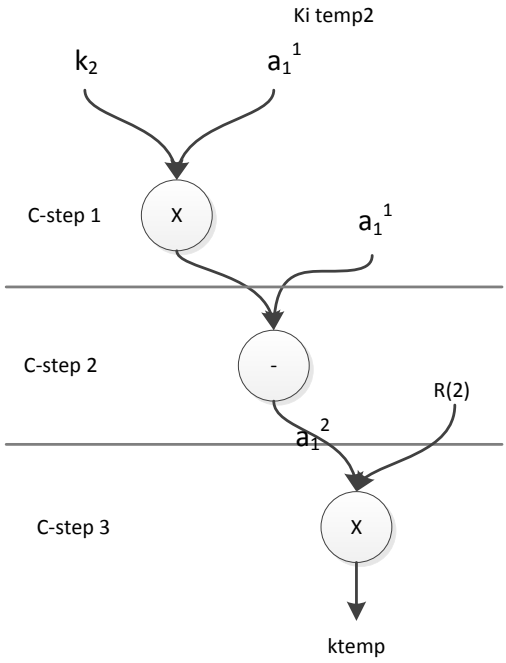

Figure 6.27: Scheduling for the k_i temp1.

Figure 6.28 depicts the scheduling for the k_i temp3 where the MAM block takes up 3 c-steps.

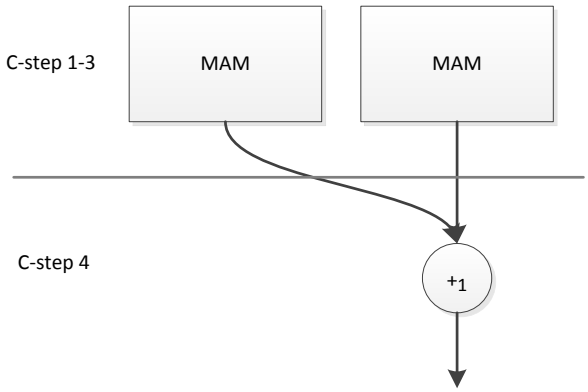

Figure 6.28: Scheduling for the k_i temp2.

Figure 6.29 depicts the scheduling for the k_i temp11 where the MAM block takes up 3 C-steps, and combined it takes up 7 C-steps.

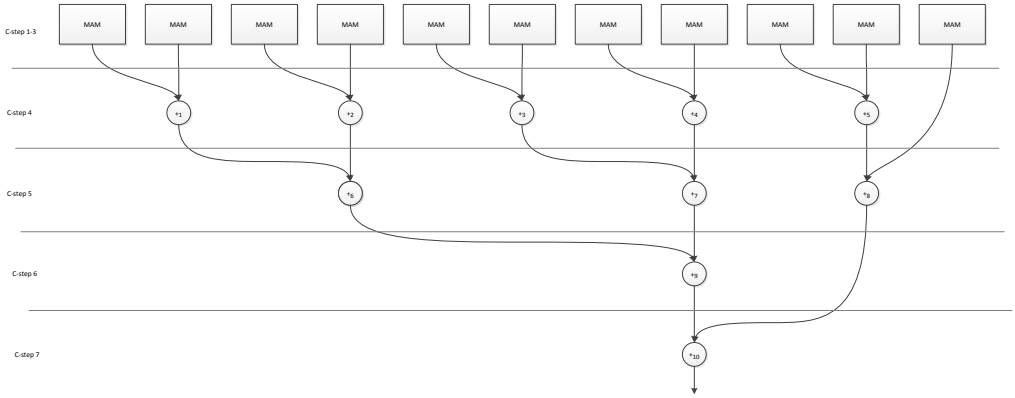

Figure 6.29: Scheduling for the k_i temp11.

The scheduling for the k_i temp3-4 takes up 5 C-steps, k_i temp5-8 takes up 6 C-steps, k_i temp9-10 takes up 7 C-steps. But they are not depicted as their scheduling is trivial solutions which looks like the scheduling for k_i temp2 and k_i temp1 in figure 6.28 and 6.29 respectively.
Now knowing the C-steps each part of the algorithm, and knowing the precedence relation a scheduling can be made where every calculation is done as soon as possible. This scheduling is depicted in figure 6.30

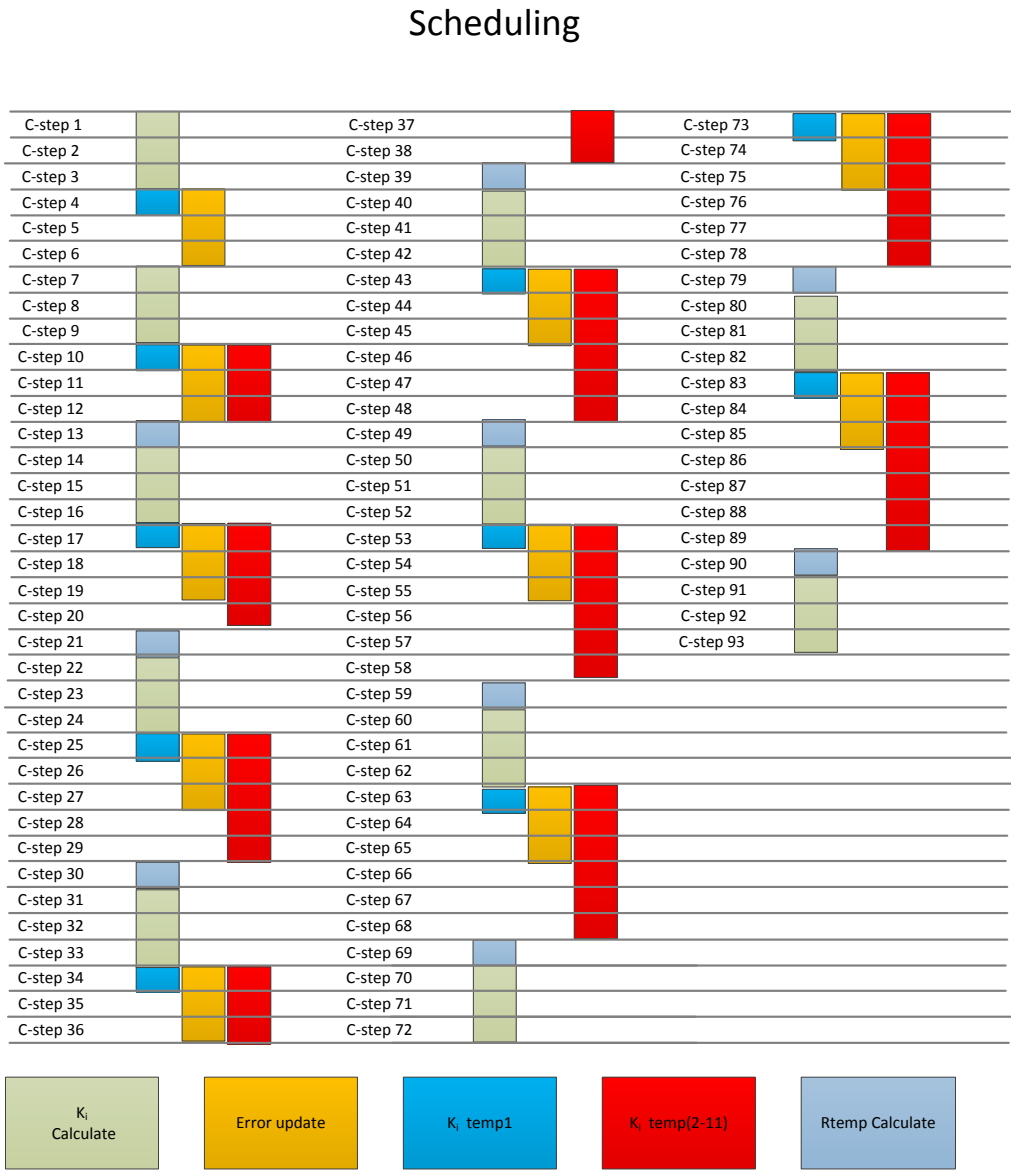

Figure 6.30: The figure depicts a scheduling of the Levinson-Durbin Algorithm

6.3.1 Hardware

In this subsection the hardware is described, using two's complement sign number representation. By analysing the algorithm, and how each block is build up in the DFG, hardware blocks can be build which can solve different parts. This leads to three different hardware blocks, one which can do the sign invert, another block

which can do the division, and a last block which can do the multiplication, division and substitution.

None restoring division

Division in hardware can be done in different ways, two common ways is restoring division and none restoring division [Mostafa Abd-El-Barr, 2005, p.73].

Restoring division is where the number is subtracted by the divisor, if the result is negative, then the number is restored by adding the divisor, if positive the number is then past on as the remainder. The remainder is then bit left shifted, then the remainder is subtracted by the divisor. Then if the number is negative the number is restored by adding the the divisor, and put as the new remainder, if it is positive it is put as the new remainder, This continues for as many bit there is, this potential gives 2(S+A)-A calculations, where S is subtraction, and A is additions, for 16 bit calculations, this gives potential 31 iterations.

None restoring division can be put as an algorithm, repeat following by n bit times: if the sign is positive left shift number and dividend, subtract the number by divisor, and set dividend LSB to "1". Else if the sign is negative, shift the number and dividend, subtract the number by the divisor and set the dividend LSB to "0". If the sign is 1, add then number to the divisor. This algorithm gives 2(n)+1, where n is the number of bits, so this algorithm takes 17 iterations to complete.
Using the previous calculations of how many calculations is needed for restoring and none restoring division, it is decided to use none restoring division. To give another prof of concept for the calculation using none restoring division using the number 8=(1000) divided by 3=(0011), the calculation is depicted in figure 6.31, as poven in [Mostafa Abd-El-Barr, 2005, p.74]. as seen in the picture the result is 2=(0010) and a remainder of 2=(0010), which makes sense as 8 divided by 3 gives 2,666. The examples used is an integer calculation, and therefore will it give a result with a quotient and a remainder which can not be divided by the divisor.

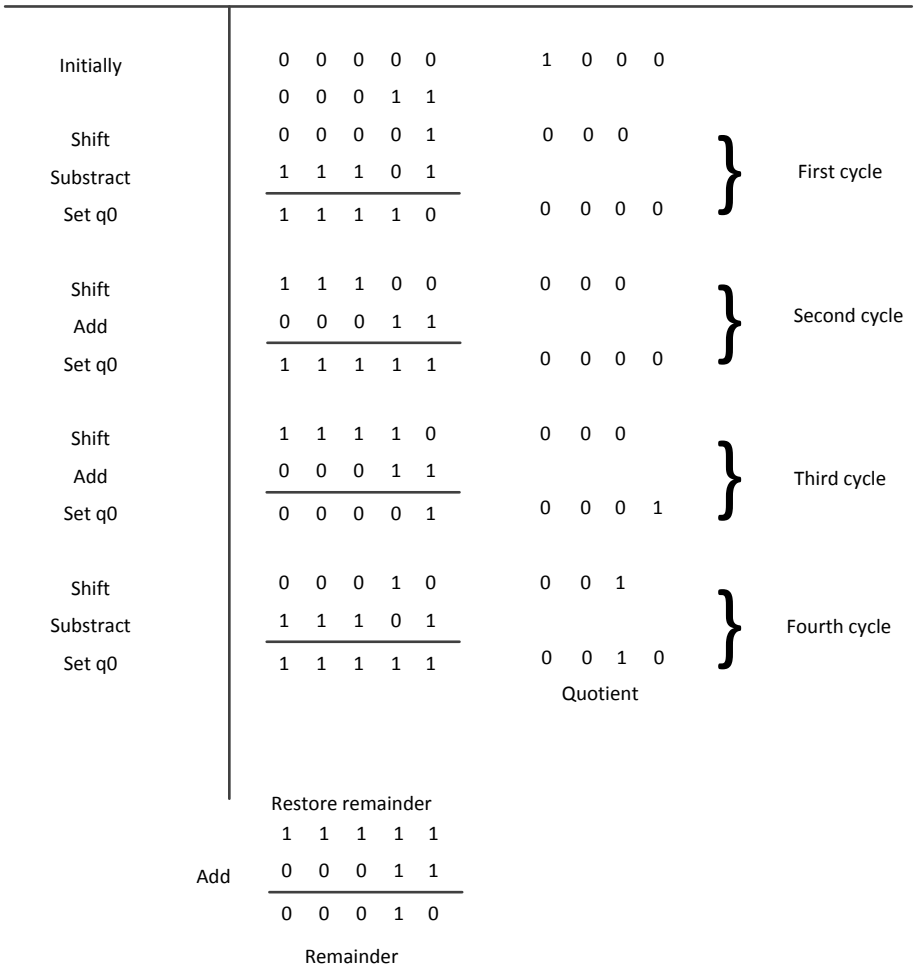

Figure 6.31: None restoring division where 8 is divided by 3

The hardware for none restoring division requires some control through the calculations, as the sign of the remainder decided if the next iteration is an addition or a subtraction. The hardware setup is depicted in figure 6.32 where it is seen that the hardware takes an input for the divisor, and the input number , clock, a clear signal and the MUX_sel to put in the number to begin with. Then there is the control part which sends the sign bit back to control if the calculation should be set to addition or subtraction, there is the control of the dividend and the result. The hardware block is a RTL drawing made from the hardware setup depicted in [Mostafa Abd-El-Barr, 2005, p.74].

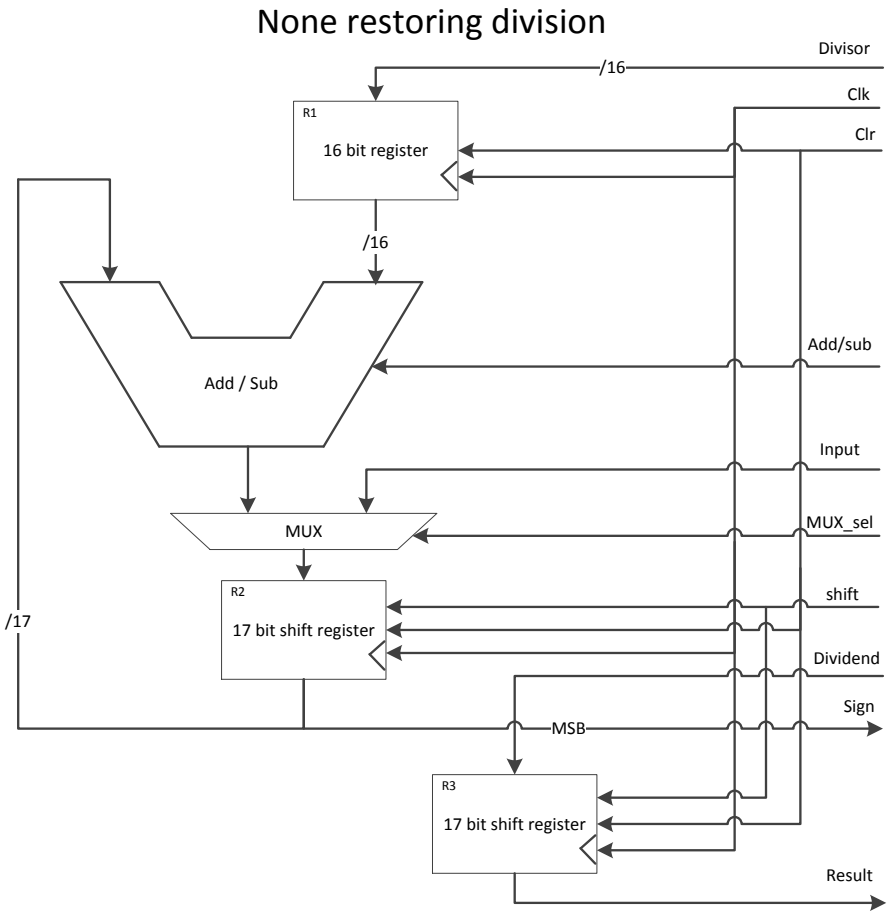

Figure 6.32: None restoring division hardware

Sign invert

To invert the sign using two's complement number representation, the most significant bit is inverted, and a least significant bit is added going from plus to minus sign, the same is done going from minus to plus sign. Figure 6.33 depicts the sign invert hardware, it takes an input, and the input with one LSB set to "1", a clock signal, and a clear signal to clear the registers, and it gives a result.

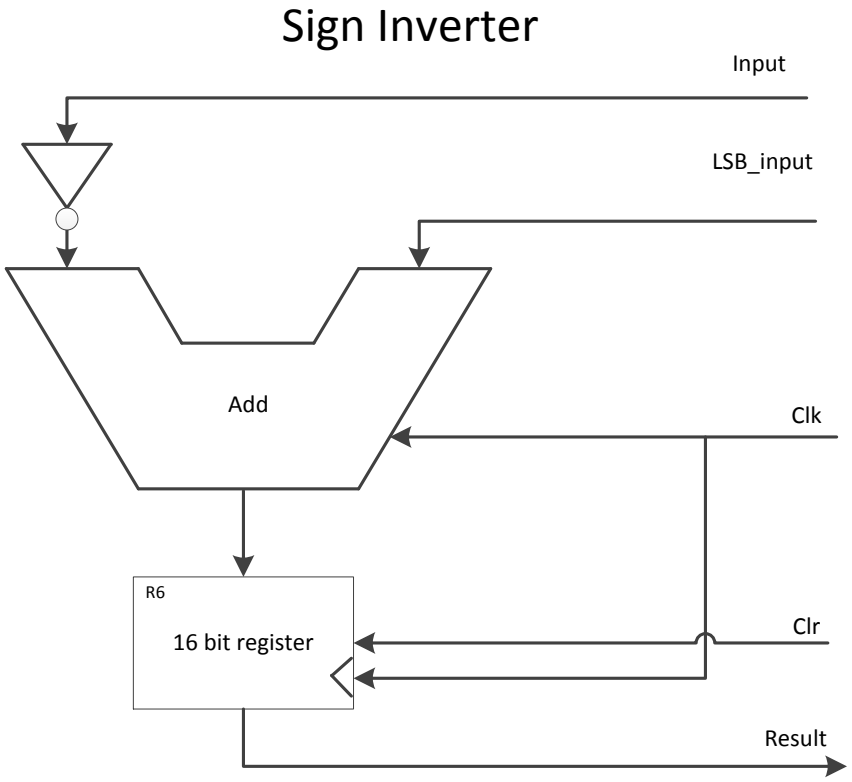

Figure 6.33: Sign invert hardware set up

Multiplication, Division and Substitution

The last hardware block can do the remaining calculations. The hardware is build so that it can do independent multiplications, and independent additions or subtractions, or it can do multiplication addition and multiplication following each other, as the MAM block described earlier in 6.3. In the hardware setup there is the inputs and outputs as follows:

Input_ADD/SUB1	1st Input for the addition or subtraction
Input_ADD/SUB2	2nd Input for the addition or subtraction
Input_MUL1	1st Input for the multiplication
Input_MUL2	2nd Input for the multiplication
MUX_Sel1	MUX selects if the input number for the multiplication is a new input from input_MUL1 or a number from the addition or division
MUX_Sel2	MUX selects if the input number for the addition/subtraction is a new input from input_ADD/SUB2 or a number from the multiplication
Mul_res	Gives the result from the multiplication register
Add_res	Gives the result from the Add/sub register
Clr	Clears the registers
Clk	Clock
Add/sub_sel	Selects addition or subtraction

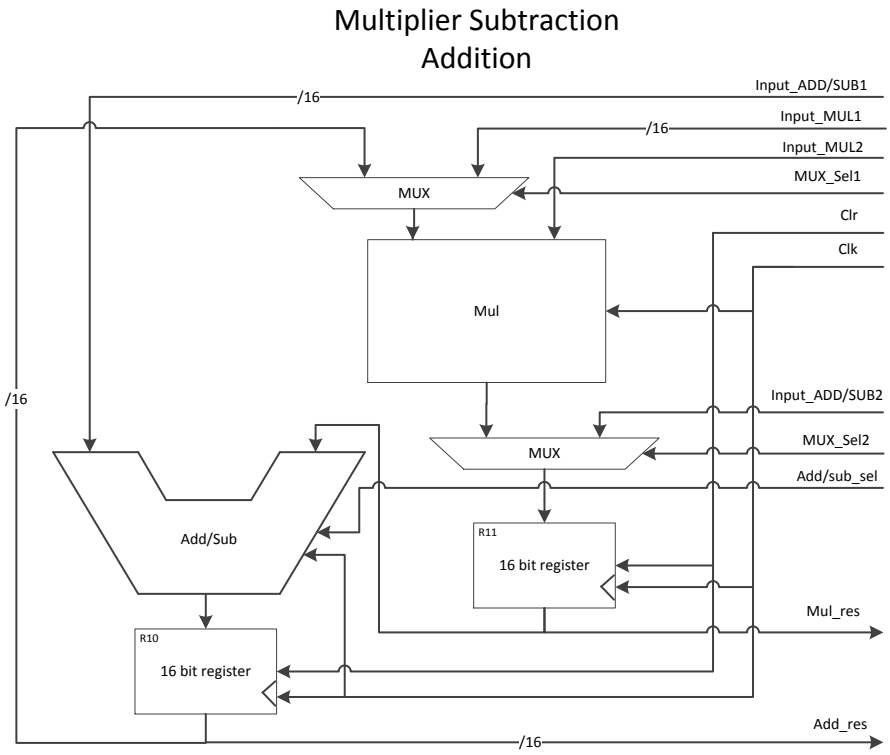

Figure 6.34: Multiply Subtraction Addition hardware block used to solve the MAM part of the code.

6.4 Finite State Machine

In this section a FSM is build up containing a master ASM chart which is the driving state machine, and several slave ASM charts which is in charge of driving the hardware and doing the separate calculations, and controlling the registers, and the intermediate register logic, input and output.

6.4.1 Master ASM

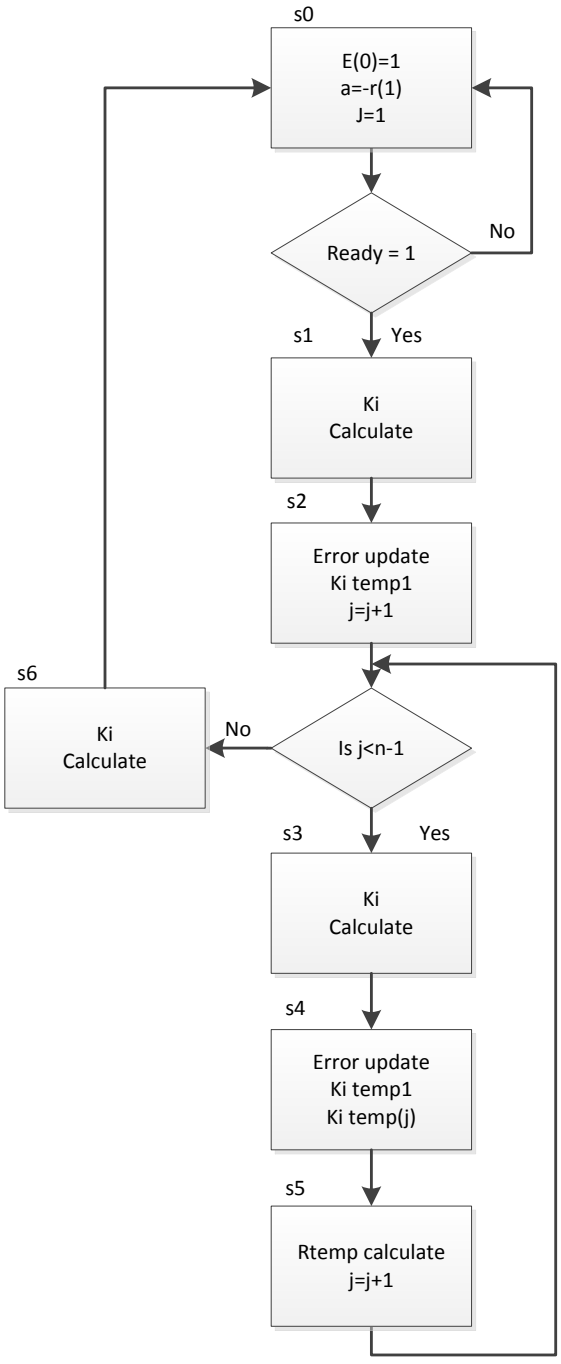

Figure 6.35: ASM chart which runs the other state mashines

6.4.2 ASM Ki Calculate

The ASM chart driving the Ki Calculate block, is depicted in 6.36.

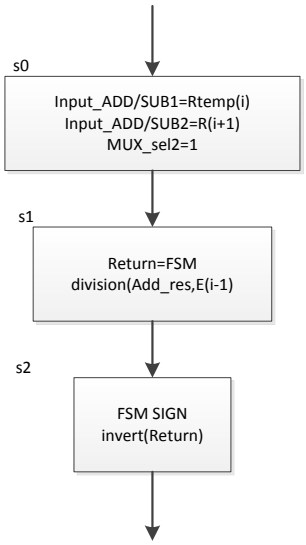

Figure 6.36: ASM chart driving the Ki calculate block

6.4.3 ASM Ki temp1

The driving ASM chart for the Ki temp1 block is depicted in figure 6.37 , it is a simple block which does a simple calculation.

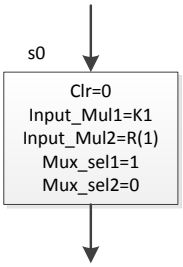

Figure 6.37: The ASM chart which runs the Ki temp1 calculation

6.4.4 ASM Rtemp Calculate

This block which sums the Ki temp1 and Ki temp(j) variables, the ASM chart is depicted in figure 6.38 which drives a Multiply Addition Subtraction (MSA) block.

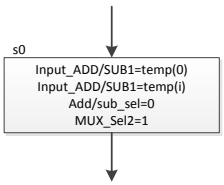

Figure 6.38: The ASM chart which runs the Rtemp calculate block calculation

6.4.5 ASM Ki temp()

The driving part of the ASM chart for sending the right set of variables to the Ki temp() block is depicted in figure 6.39.

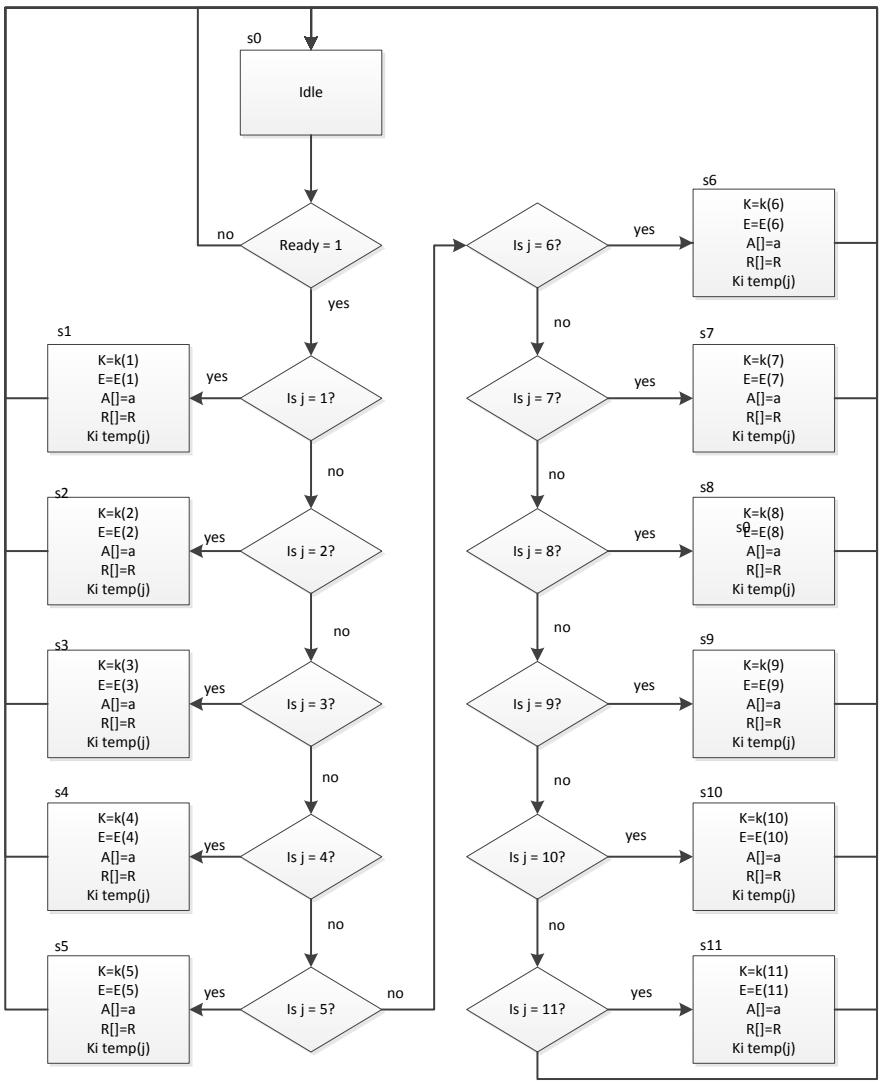

Figure 6.39: The ASM chart which runs the MAM part of the Levinson-Durbin algorithm

The slave ASM chart for the Ki temp() algorithm, sends the signals to drive the MSA hardware block, and since there is parallelism in some of them the calls from the algorithm goes to different MSA blocks, as seen in figure 6.40,6.41,6.42,6.43 and6.44. Only the slaves for run 1-4 and the last number 11 is depicted as the rest can be derived from the once depicted.

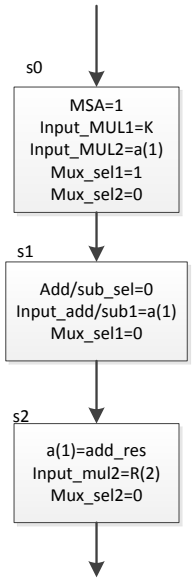

Figure 6.40: The ASM chart for the Ki temp(2) part of the Levinson-durbin algorithm

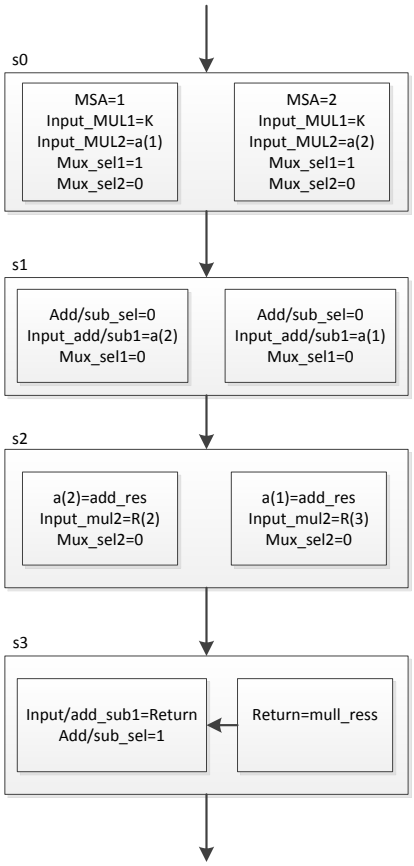

Figure 6.41: The ASM chart for the Ki temp(3) part of the Levinson-durbin algorithm

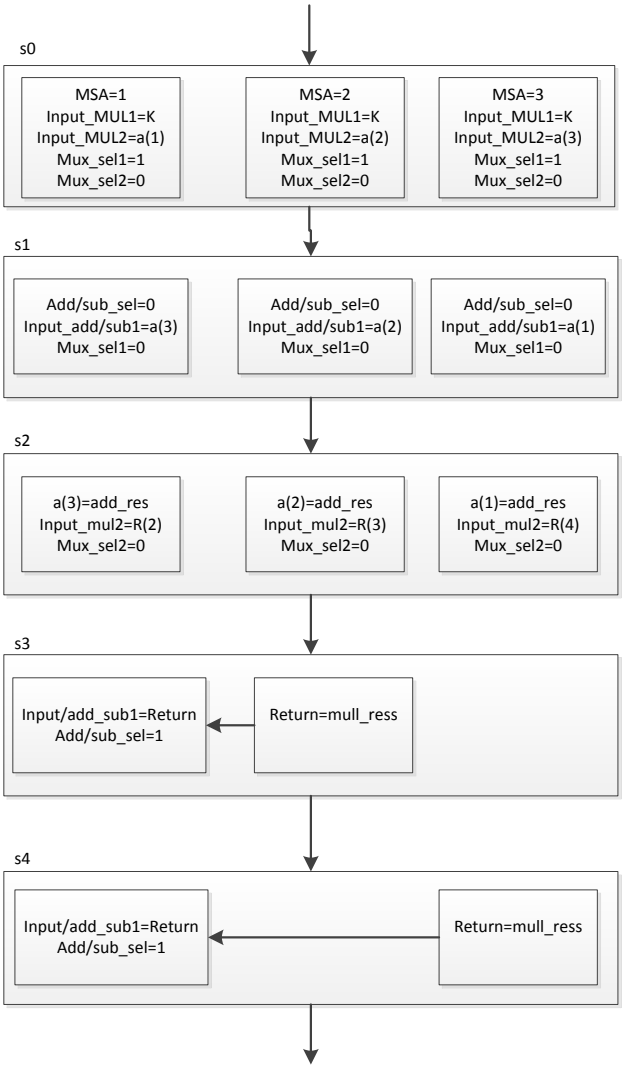

Figure 6.42: The ASM chart for the Ki temp(4) part of the Levinson-durbin algorithm

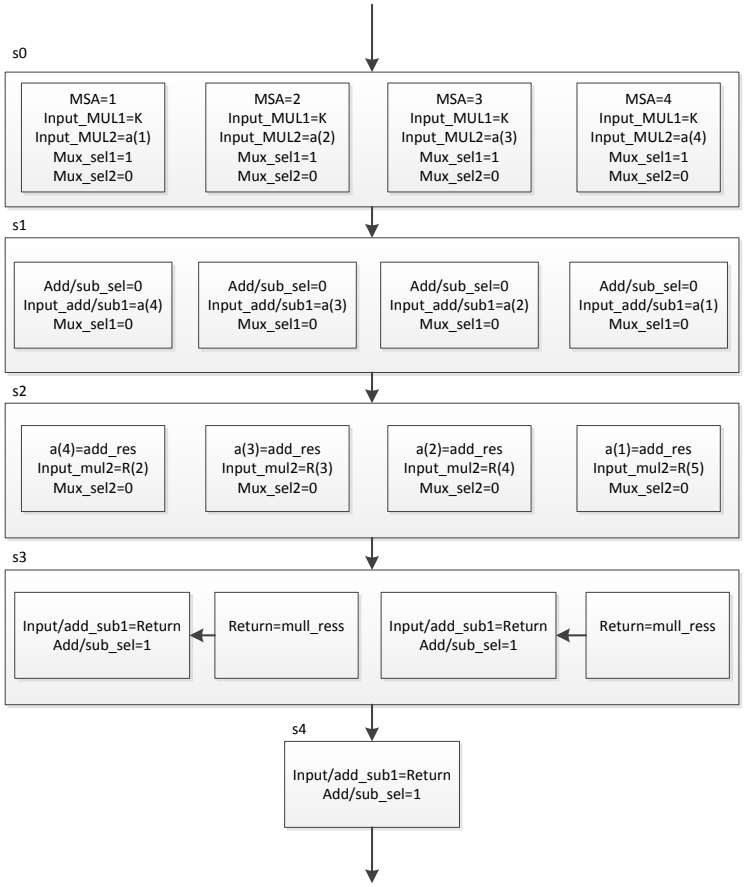

Figure 6.43: The ASM chart for the Ki temp(5) part of the Levinson-durbin algorithm

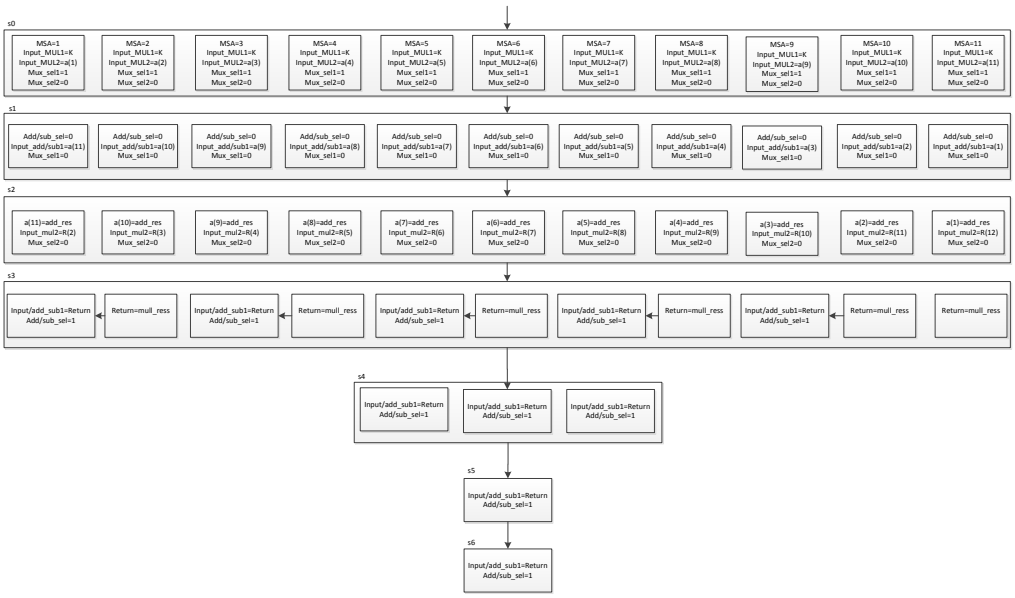

Figure 6.44: The ASM chart for the Ki temp(11) part of the Levinson-durbin algorithm

6.4.6 ASM sign invert

To drive the sign inverter a ASM is presented in figure 6.45 where the input is inverted and added one LSB "0001".

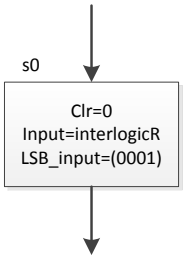

Figure 6.45: the ASM chart for inverting the sign

6.4.7 ASM division

As earlier ASM chart, as depicted in figure 6.46 runs 16 iterations of the loop, then to restore the remainder if the sign of the remainder is negative, by adding the divisor.

So the result is the result output, plus the output og the "sign" where the remainder is restored, as depicted in figure 6.46

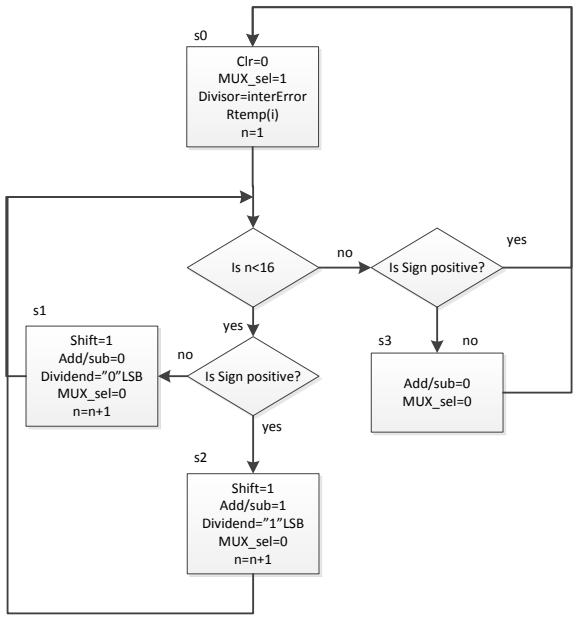

Figure 6.46: The ASM chart driving the division

6.4.8 ASM error

When the data is ready the inputs is set and first the multiplication is done, then 1 is subtracted by the result of the multiplication. The result is then multiplied by the old Error, when done the calculation done flag is set, and the result is sent to the intermediated logic registers, as depicted in figure 6.47.

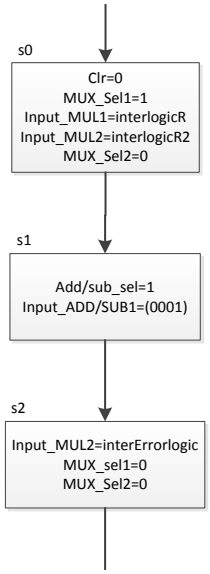

Figure 6.47: ASM describing the signals to drive the Error update

6.4.9 Final Schedule

With the earlier schedule made considering control-steps, where each operation takes 1 C-step, we can now make a final schedule. The difference is that the division takes 17 clocks each C-steps, unlike the other operations.
This changes the scheduling radically since the division takes up 17 clocks each time, and the rest of the functions have to wait for the division to be done. The new schedule is depicted in 6.48, the change is that before it was the k_i temp(2-11) that took the longest now it is the division which takes the longest, changing it from 93 C-steps into 189 clocks.

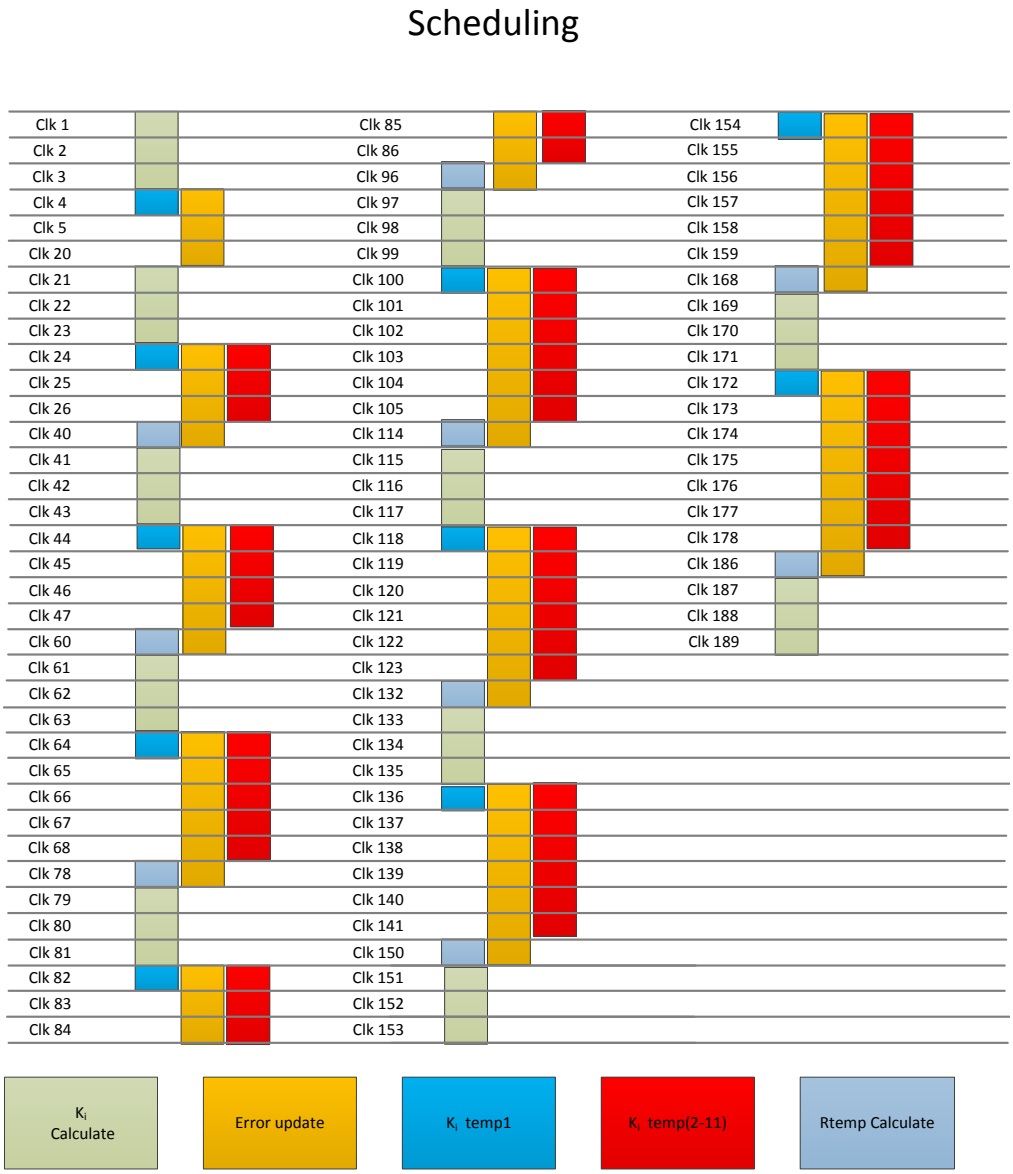

Figure 6.48: Final schedule where the clock cycles are depicted

6.4.10 Registers

As seen in figure 6.49 the number of registers needed are incrementing as the vector a grows for each iteration. This means that in the beginning most of the registers are unused.

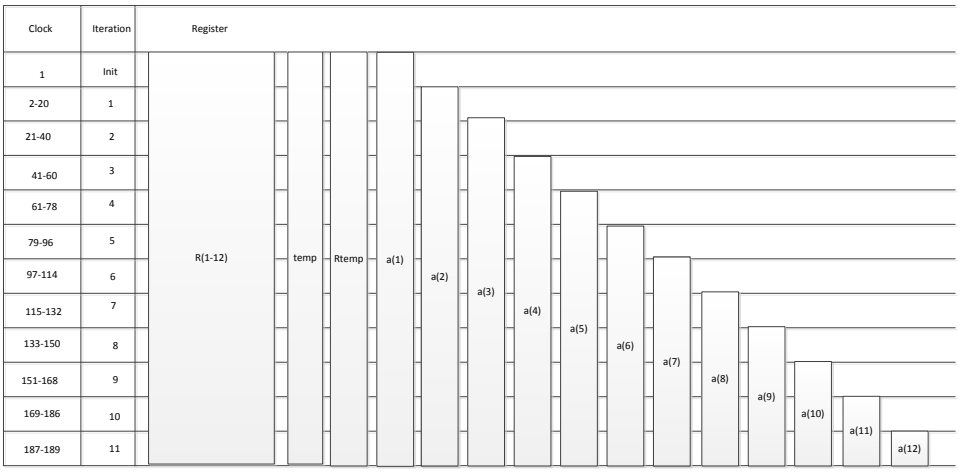

Figure 6.49: Register Lifetime analysis

6.4.11 State vector

The controlling part of the Finite State Machine is the state vector, which iterates in the states to send the signals to control the Data Path. The states for driving the control path is depicted in figure 6.50, where the description for each step in the State vector is described in the following tabular:

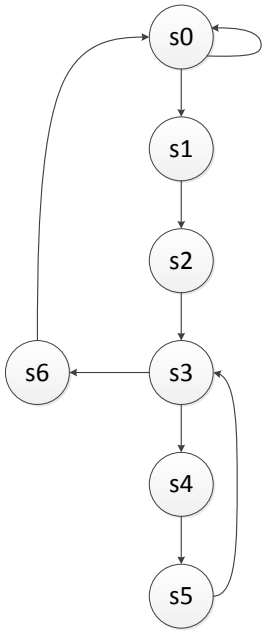

Figure 6.50: State vector for the Levinson-Durbin algorithm

State	Call
s0	Idle iteration, waiting for ready signal
s1	k_i calculate
s2	update error, k_i temp1
s3	k_i calculate
s4	update error, k_i temp1, k_i temp(j)
s5	Rtemp calulate
s6	k_i calculate

The different iterations consists of a set of calls, which is dependent on each other where each of the calls for each iteration is presented in the following table:

iteration	calls
1	k_i Calculate, k_i temp1, Error update
2	k_i Calculate, k_i temp1, k_i temp2, Error update, Rtemp Calculate
3	k_i Calculate, k_i temp1, k_i temp3, Error update, Rtemp Calculate
4	k_i Calculate, k_i temp1, k_i temp4, Error update, Rtemp Calculate
5	k_i Calculate, k_i temp1, k_i temp5, Error update, Rtemp Calculate
6	k_i Calculate, k_i temp1, k_i temp6, Error update, Rtemp Calculate
7	k_i Calculate, k_i temp1, k_i temp7, Error update, Rtemp Calculate
8	k_i Calculate, k_i temp1, k_i temp8, Error update, Rtemp Calculate
9	k_i Calculate, k_i temp1, k_i temp9, Error update, Rtemp Calculate
10	k_i Calculate, k_i temp1, k_i temp10, Error update, Rtemp Calculate
11	k_i Calculate, k_i temp1, k_i temp11, Error update, Rtemp Calculate

6.5 Conclusion

With these ASM and the hardware defined a full FSMD as depicted in figure 6.51 can be build, where each of the ASM charts makes up the Control path and a set of signal is send to the Data Path to do the calculations.

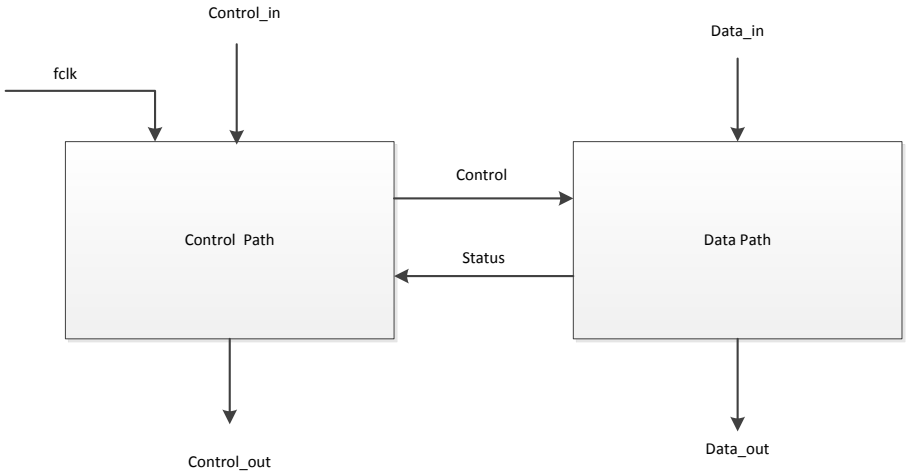

Figure 6.51: Finite State Machine with Data path

With this a RTL architecture is derived, for the Levinson-Durbin algorithm. All this gives a complete FSMD which can then be implemented in a FPGA.

Chapter 7

ADMM RTL Analysis

In this chapter a RTL analysis of the ADMM is found. First the flow of the algorithm is analysed by making a set of CDFG. With these CDFG precedence of each operation is know, which then can be worked into precedence graphs, which can give an insight into how much parallelism there is in the algorithm. Knowing the operations needed for the algorithm and the precedence, a FSMD can be designed.
A Finite State Machine with Datapath consists of two different parts, the Data path, and the the control path.
Where the DataPath includes the hardware arithmetic used to do the calculations, and the control path is where the flow of the algorithm is controlled.

7.1 Control Data Flow Graphs

As described earlier in section 3 a set of CDFG can be drawn to describe the flow of the ADMM algorithm, in this section these CDFG are further described to go into detail of how the algorithm flows.
First a CDFG for the Algorithm, where as earlier there are four main parts/lines in the algorithm which the flow graphs describe later in detail. Figure 7.1 describes one iteration of the algorithm, where the initial variables for the algorithm is found.

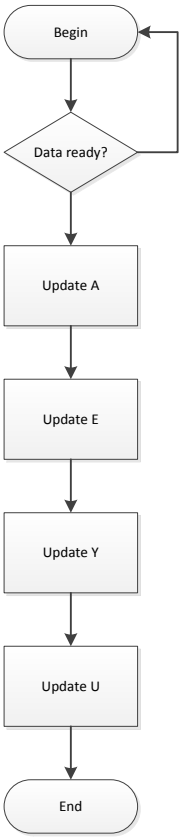

Figure 7.1: Initial CDFG of the ADMM algorithm

Figure 7.1 describes four blocks which is $UpdateA$, $UpdateE$, $UpdateY$ and $UpdateU$, these are the 4 building blocks of the algorithm

7.1.1 Update A

The $UpdateA$ block is the most complicated block as described earlier in section 3 because of the least-squares problem, which is decided to be solved by using the Levinson algorithm, described in section 3, the block also consists of a vector subtraction, with a 12 variable long vector. The CDFG for the algorithm is depicted in figure 7.2, where the levinson block is further described later in 7.1.5

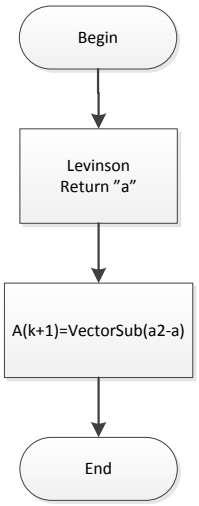

Figure 7.2: DFG describing the Update A block

7.1.2 Update E

The *UpdateE* depicted in figure 7.36 is where the error vector is updated, this is done by multiplying the a coefficients found in *updateA*, with the X matrix defined in equation2.21, the result is subtracted from x, which is the objective function for the equation. This gives the residual also called error for the signal.

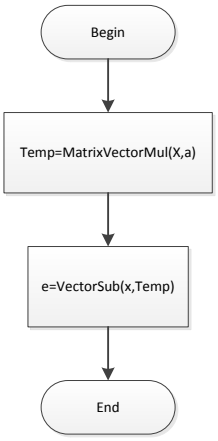

Figure 7.3: DFG describing the Update E block

7.1.3 Update Y

The $UpdateY$ is where the threshold function is used on the sum of the error$e$ and the u vectors. The threshold function is described earlier in section 3.7.1. The threshold function is the function which makes the error signal more and more sparse each iteration, as it sets the value below the threshold to 0 and the values above +/- the threshold to the value minus the threshold value. The $UpdateY$ CDFG is depicted in figure 7.4

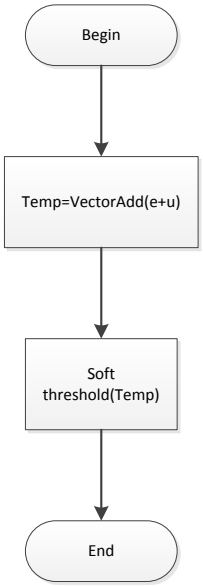

Figure 7.4: DFG describing the Update Y block

7.1.4 Update U

The $UpdateU$ CDFG is depicted in figure 7.5, this is the simple block where two vectors(u and e) are added and subtracted by the y.

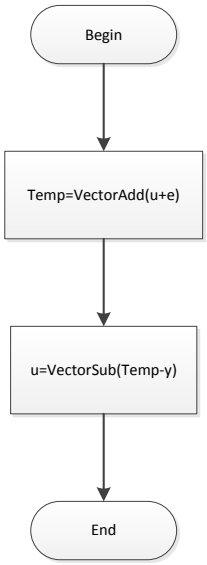

Figure 7.5: DFG describing the Update U block

7.1.5 CDFG Levinson

As described earlier in section 3 where the Levinson Algorithm is described in Algorithm 3.2 it consists of an outer loop and an inner loop, which is also visible in figure 7.6. The thing about the inner loop is that it is identical with the $Levinson-Durbin$ algorithm, this is also described in [Gene H. Golub, p.211]. The thing about the outer and inner loop ,is that the first iteration of the inner loop is independent of the outer loop in the first iteration. The second loop of the inner loop is dependent of the first iteration of the outer loop. This means that the algorithm can be run in parallel, so that the outer and inner loops runs at the same time. This is also described in third edition of the Gene H. Golub book.

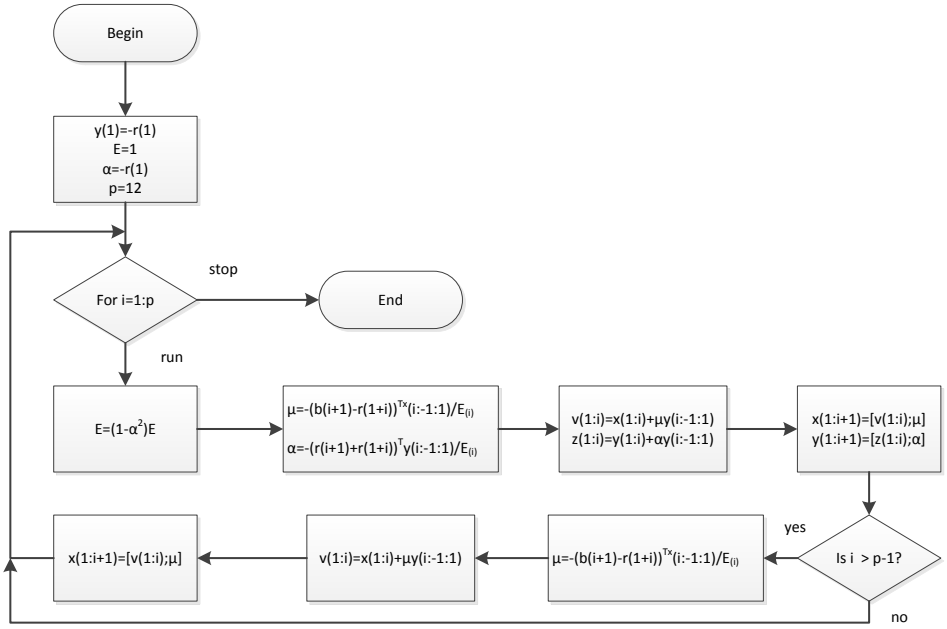

Figure 7.6: CDFG for the Levinson Algorithm

Analysing the outer and the inner loop of the Levinson shows that the difference is sign of α is inverted, where the sign of μ is the same, and in the calculation of the respective lines, there is a subtraction in the μ line, and a addition in the α line. This means that the outer loop is a little faster because there is no sign inversion. Given these conclusion, it means that the same hardware and flow as used for the Levinson-Durbin algorithm can be used for the Levinson.

7.2 Precedence graphs

The CDFG previously described in chapter are all doing vector/ matrix calculations, and when there is one input there is one output for each block, and partial calculations are dependent on each other. So if block diagrams were to be made of these CDFG it would result in homogeneous block diagrams, so the same information can be extrapolated using precedence graphs.

7.2.1 Update A

As described earlier in 2.2.2 the $X^+(y-u)$ can be solved by solving the *Levinson* algorithm where objective function is $Rx = b$ where R is X^TX as described earlier, and in this case b is $X^T(y-u)$ so to solve this for each iteration, first a vector

subtraction is needed for $y - u$ as these are both vectors which are 332 long, a simplified precedence graph is used to describe the parallelism, as shown in figure 7.7. As seen from the precedence graph it is possible to do all the calculations in 1 step, with all operations run in parallel.

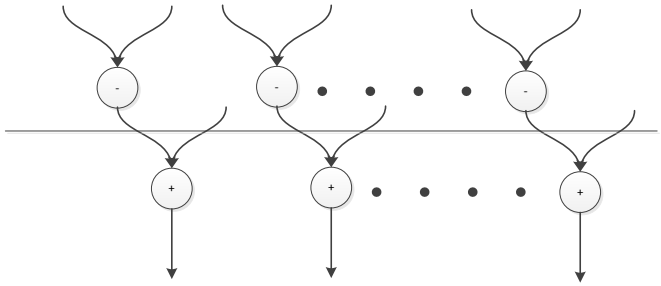

Figure 7.7: Precedence graph for the vector subtraction

This vector is then multiplied by the matrix X^T, this means that the 332 variable long vector is multiplied $332x12$ matrix the precedence graph for this is depicted in figure 7.8 in a simplified version, the real version is 332 multiplications, and then in second "step" 166 addition, then 83 additions, then 41 additions then 21, then 10 then 5 then 3 then 1 and 1. This means that in the real precedence graph, there is 1 "C-step" with multiplications and 9 "C-steps" with additions.

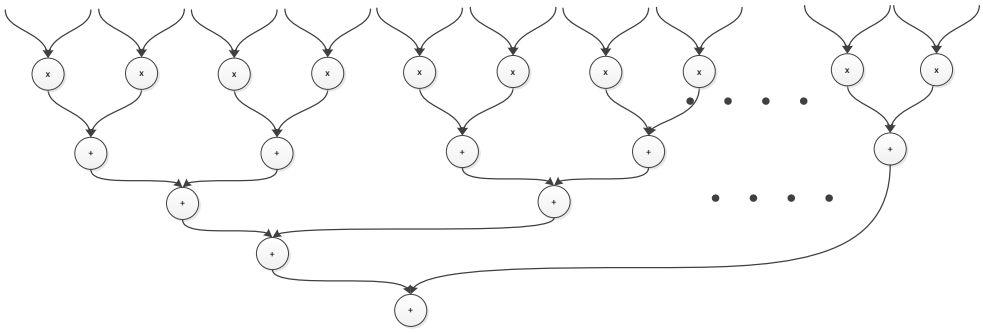

Figure 7.8: Precedence graph for the vector multiplied by the matrix X

Levinson

With the needed variables for the algorithm, the Levinson algorithm can now be run, where the r variables precedence graph was shown earlier.

The following precedence graphs are half of the *Levinson* algorithm, as the *Durbin* part was depicted earlier in section 6, and as earlier stated they are almost the same, and can run in parallel.
First the $\mu temp$ calculate where for first iteration gives the first Rtemp, as shown in figure 7.9, this is then sent to $\mu_i calculate$ where the first μ_i value is found in 7.10.

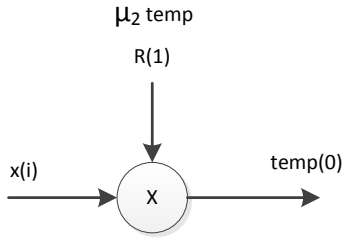

Figure 7.9: Multiplying the $R(1)$ coefficient with the last $x(i) coefficient$

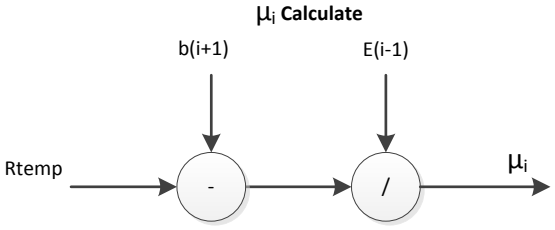

Figure 7.10: the temp value minus the $b(i+1)$ value, then divided by the error, giving μ

With the first μ_i which is μ_1, it is send to μ_i $temp1$ which gives $temp(1)$, as shown in figure 7.11. This result is then send to *Rtemp calculate* together with the result from$\mu_i temp$.

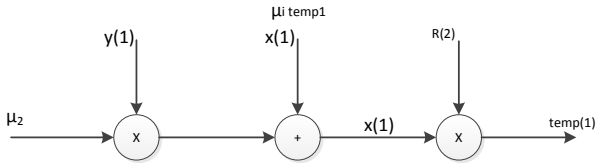

Figure 7.11: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value.

From here on the mu_1temp $mu_itemp(i)$ is called for each iteration, then summed up by the $RtempCalculate$ which is then subtracted by the $b$value, and divided by the error E this keeps on going from figure 7.11 to fig:mytemp11

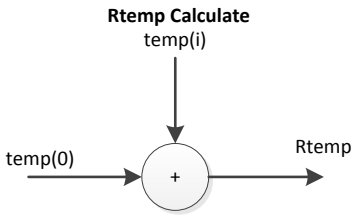

Figure 7.12: adding the first temp value with the last calculated

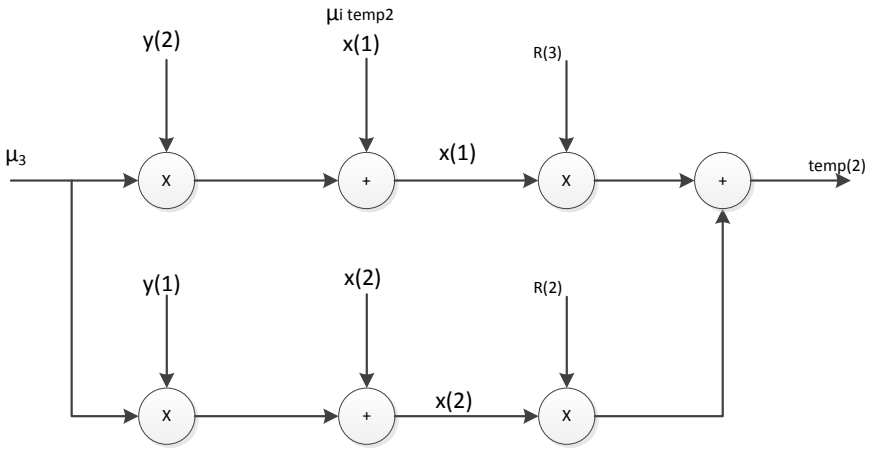

Figure 7.13: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

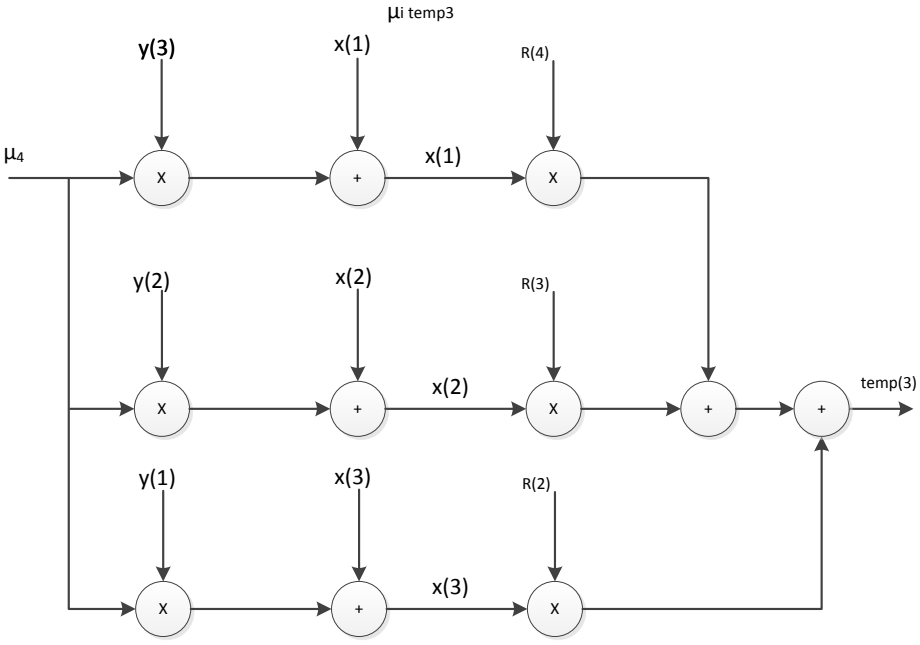

Figure 7.14: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

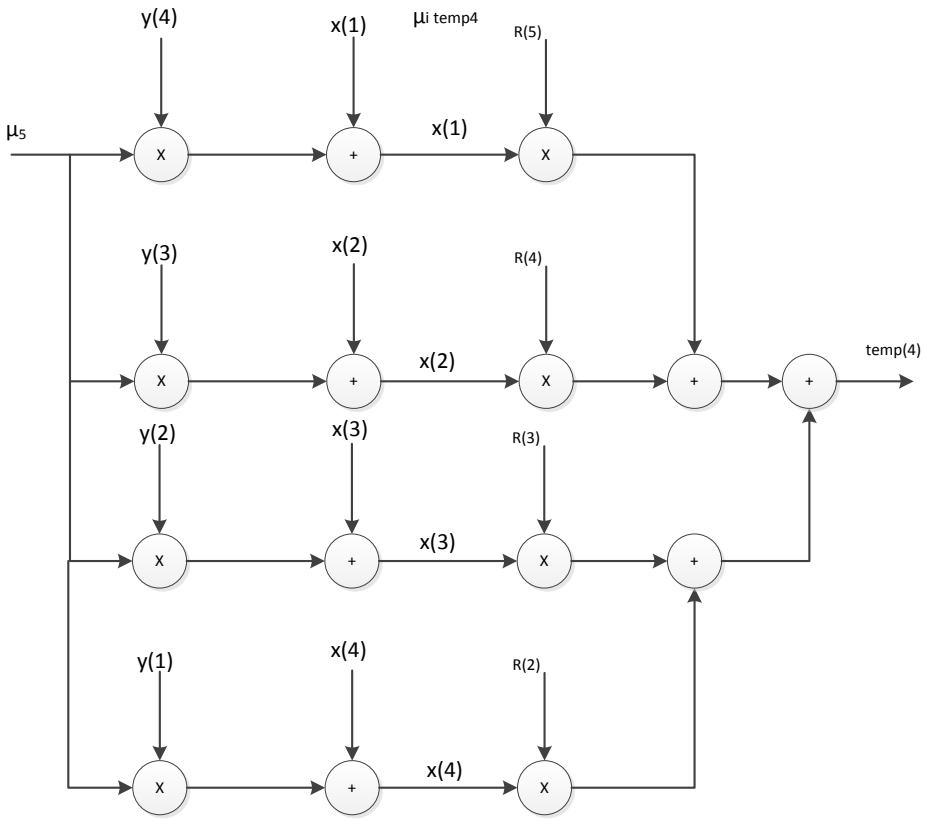

Figure 7.15: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

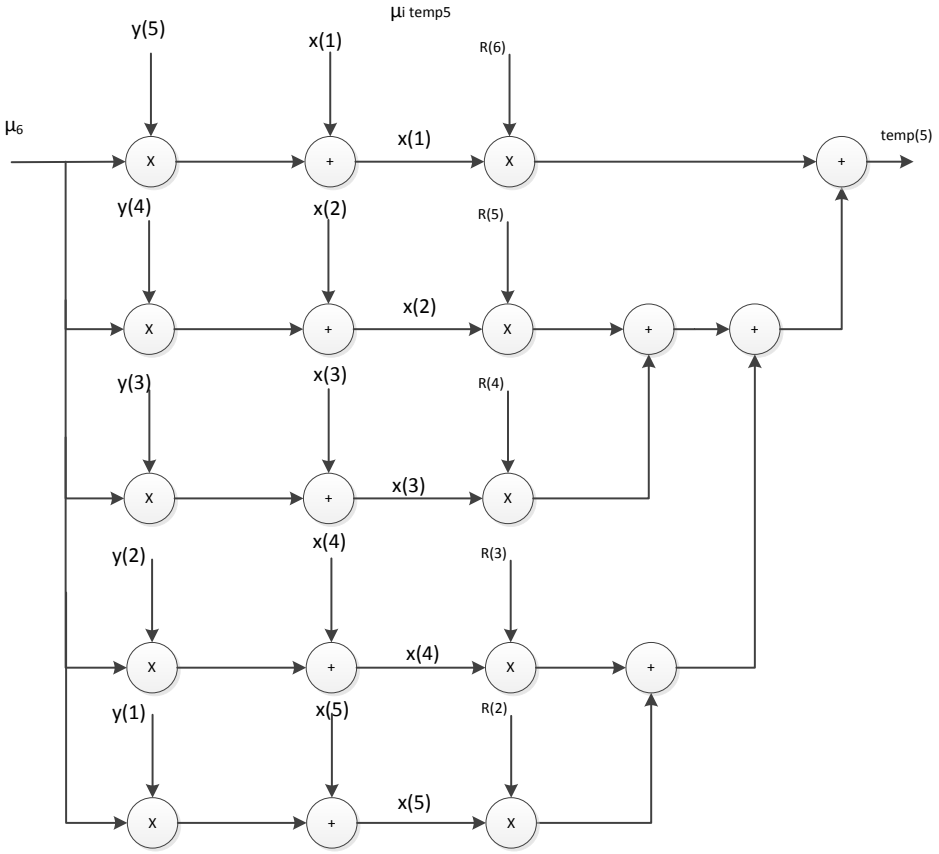

Figure 7.16: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

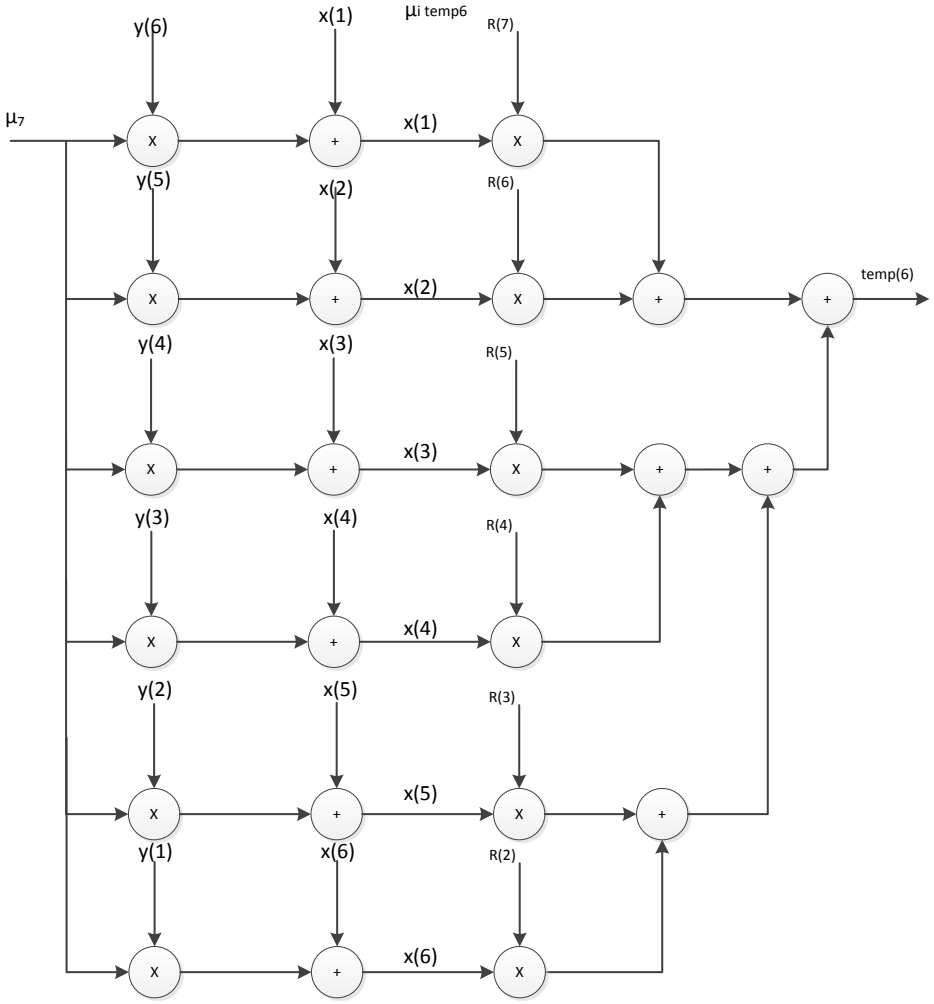

Figure 7.17: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

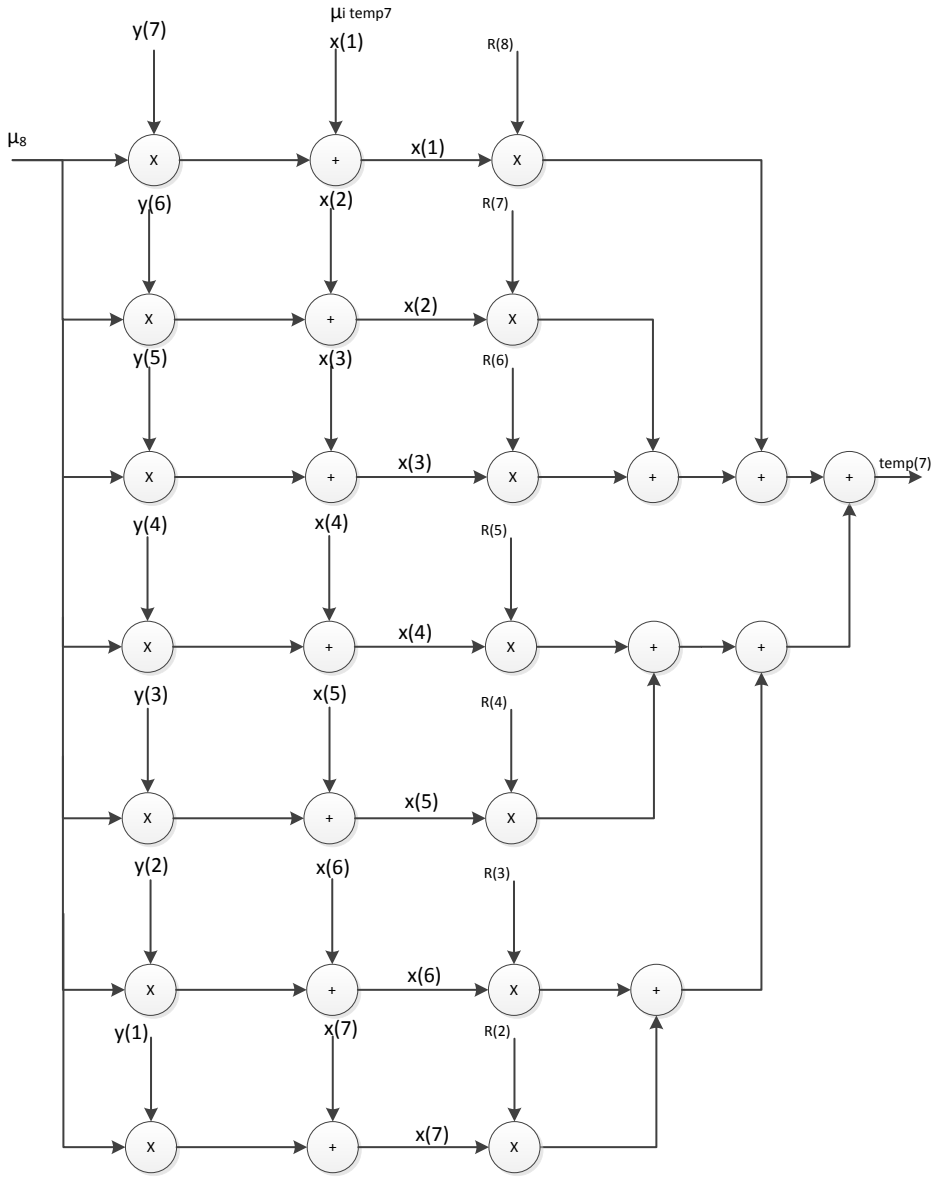

Figure 7.18: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

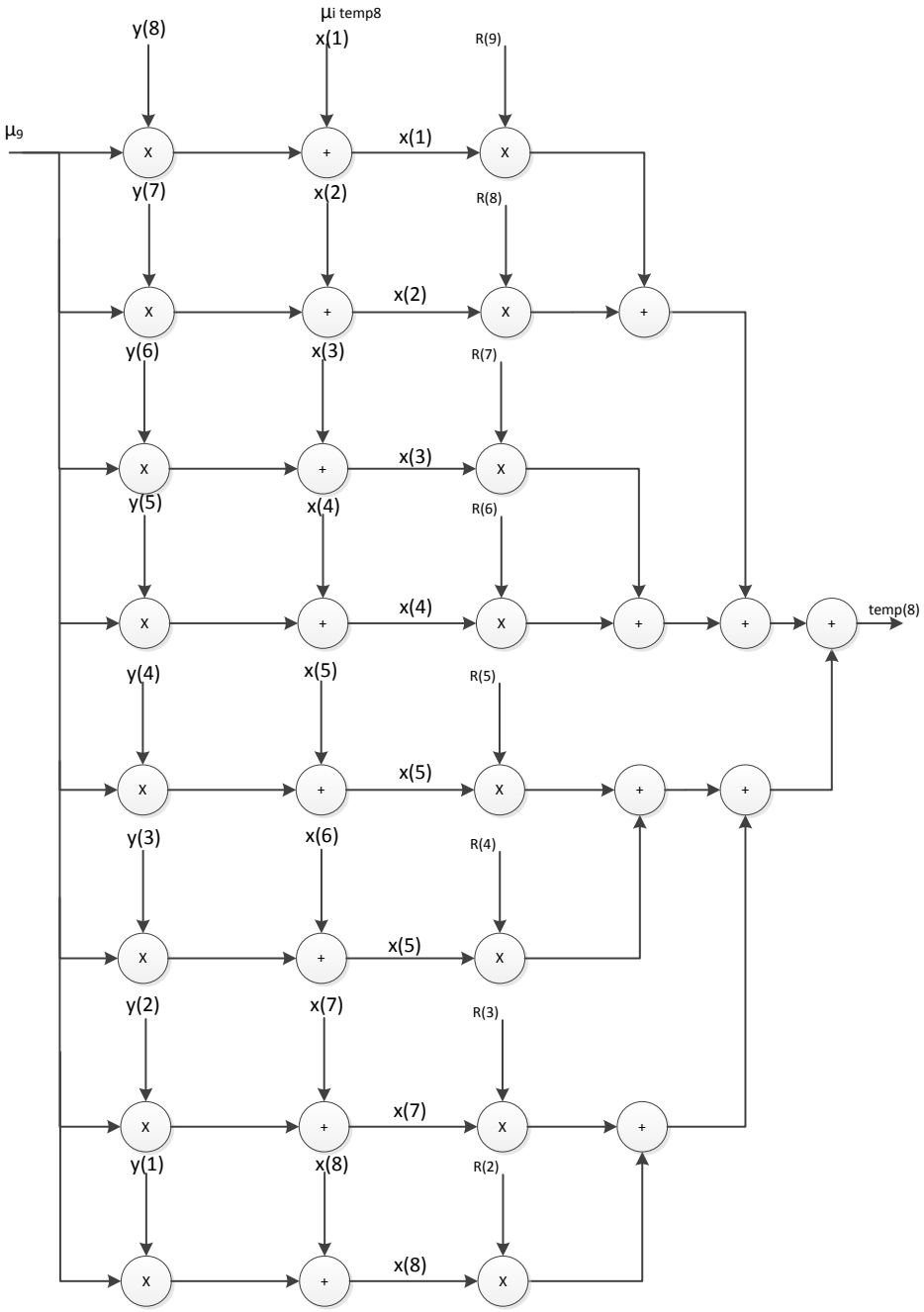

Figure 7.19: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

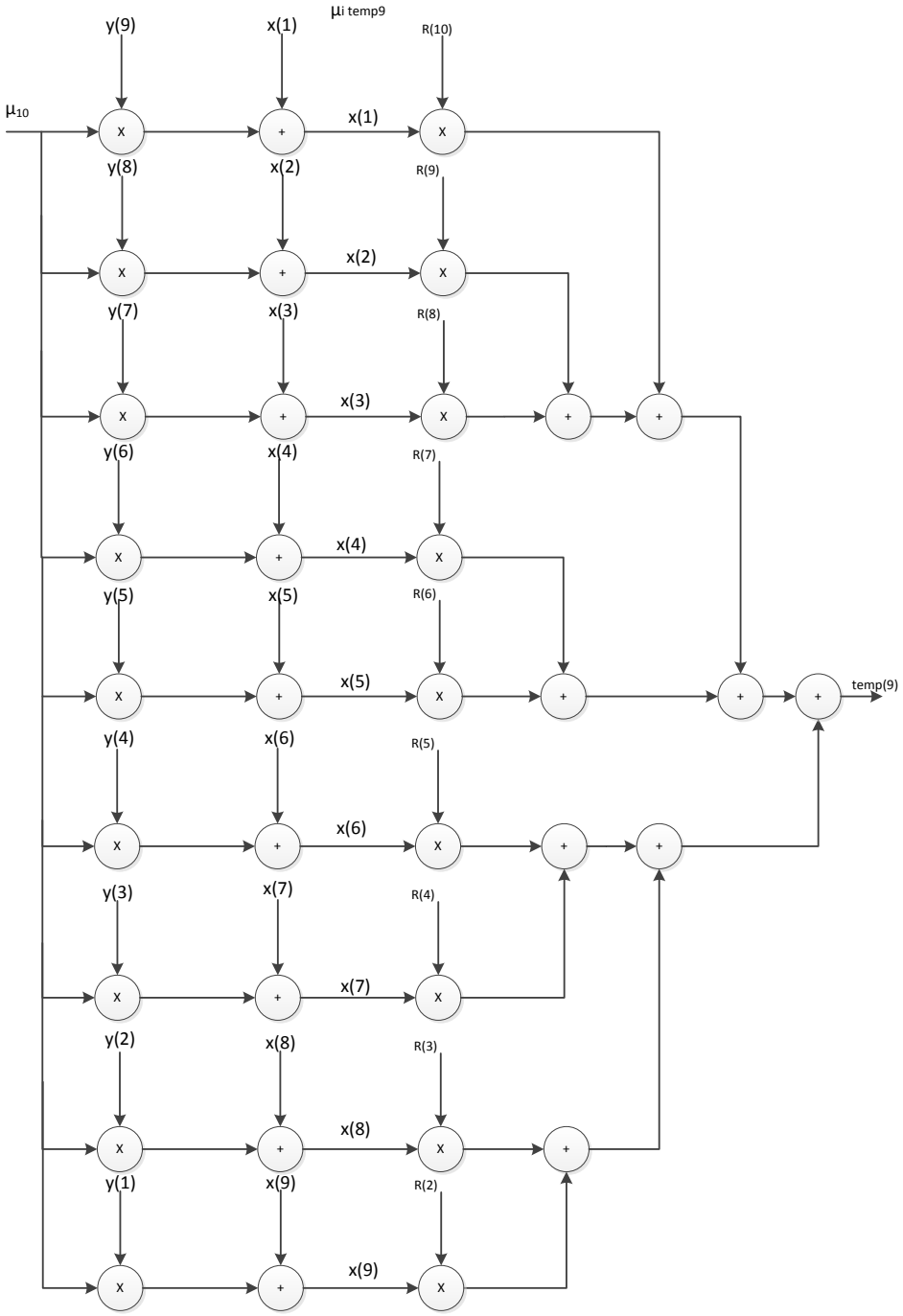

Figure 7.20: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

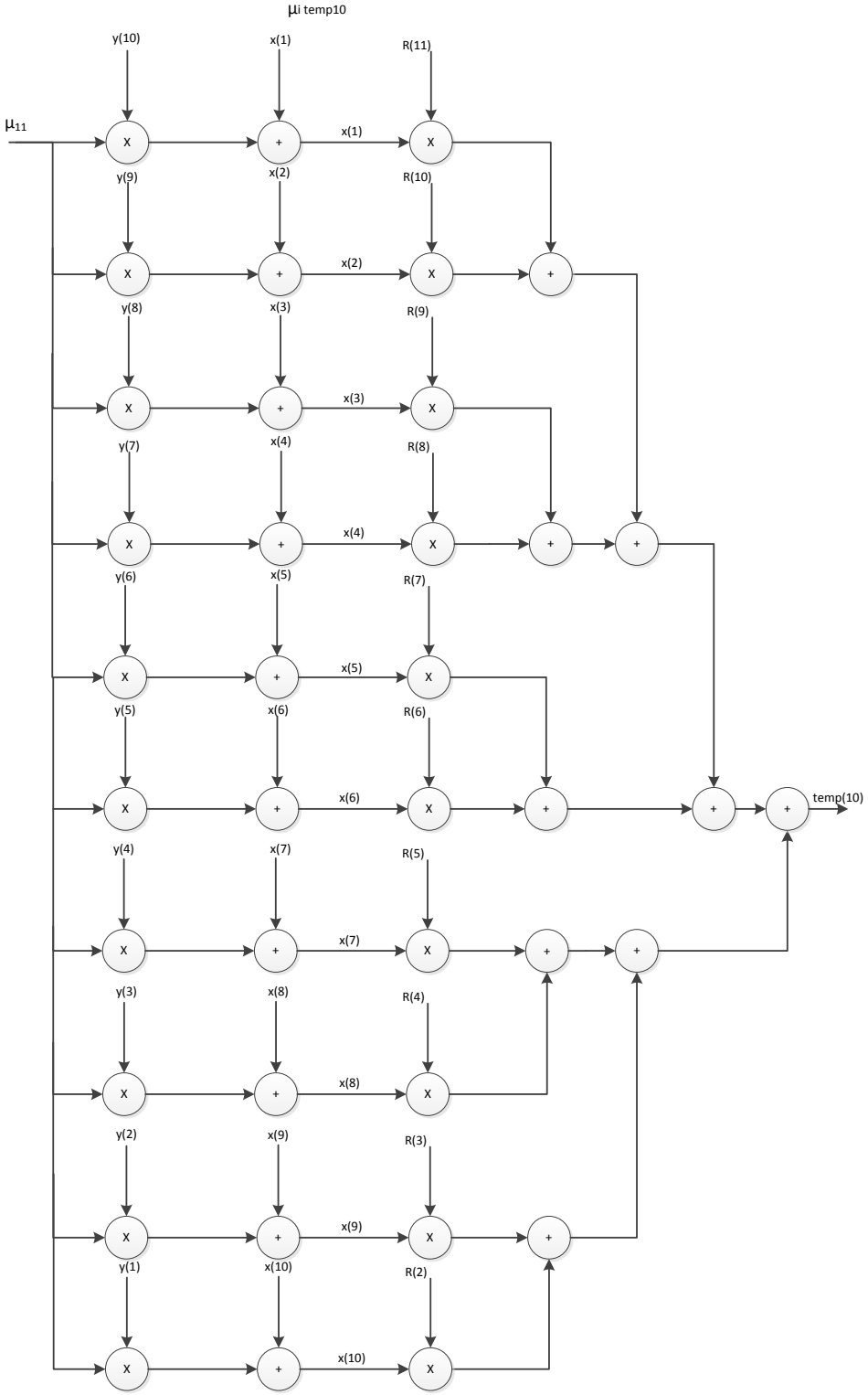

Figure 7.21: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

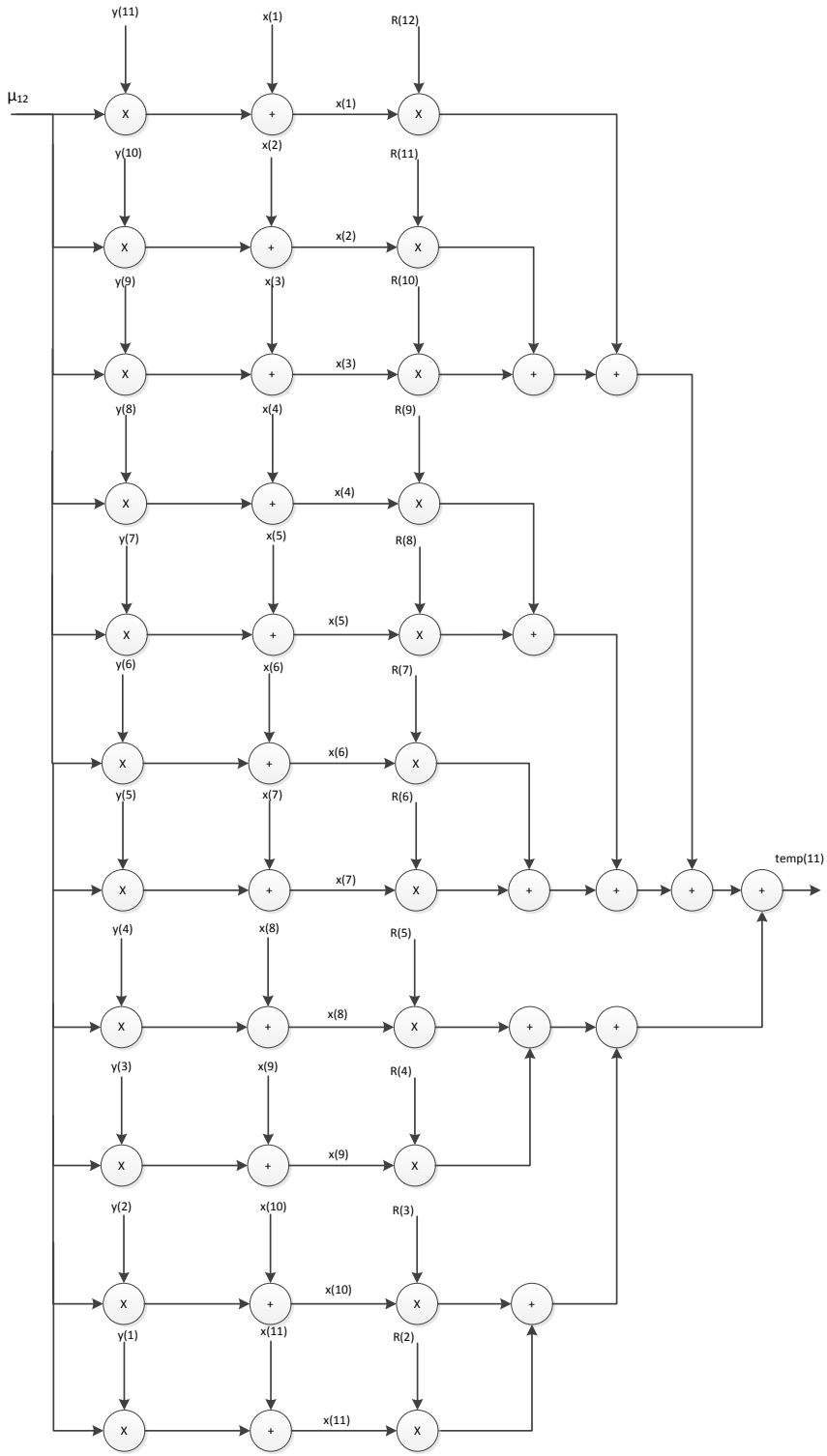

Figure 7.22: Multiplying the μ_i with the vector y, adding each variable with the x vector, which gives the new x vector, this is multiplied by the R vector, and summed up to temp value

When the Levinson algorithm is done, the a_2 vector is subtracted by the vector given by the algorithm, the precedence graph is the same as shown figure 7.7

Update error

The Error calculation is done with a vector matrix multiplication with a $332x12$ matrix and a 12 variable long vector, this gives a 332 variable long vector, this can be done with 12 iterations of a 332 long vector multiplication as shown in figure 7.23, where each iteration is multiplied and accumulated up with the previous calculated vector.

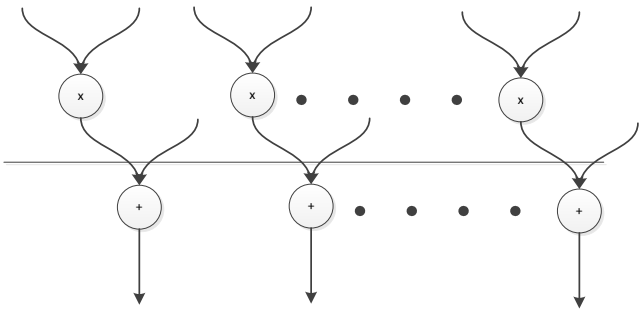

Figure 7.23: Vector matrix multiplication where the result is accumulated from the previous vector multiplication

The result is then subtracted from the vector x as in figure 7.7, which gives the error e

Update y

The *Update Y* is then calculated by adding the error e with the vector u as shown in figure 7.24 and the threshold function is then put on this value.

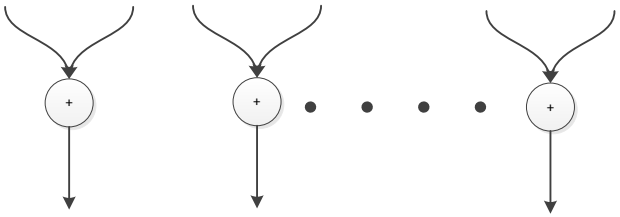

Figure 7.24: Vector addition

The threshold function is not of a specific length, as each value in the vector is checked, if it is larger or smaller then the threshold, and if it is larger then the absolute value, a subtraction is needed. If the value is smaller, the value is set to 0 however since there is 332 values, one of them is most likely larger, so with the setup, a subtraction and an addition is always needed, giving the precedence graph as shown in figure 7.25

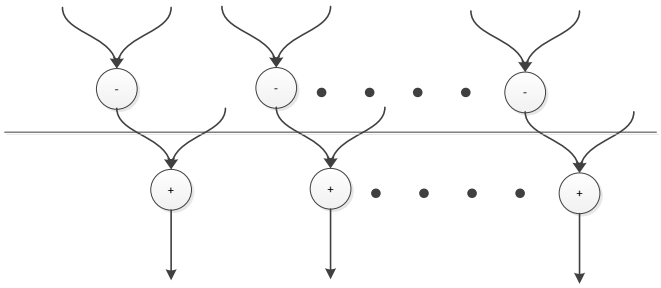

Figure 7.25: Precedense graph for the Threshold function

Update u

The Update U is a rather simple step, where the vector u is added to the error e and subtracted by the vector y where the vector multiplication precedence graphs was shown earlier in figure 7.24. The vector subtraction gives the came precedence as the vector multiplication where it just subtract instead.

7.3 Hardware

The hardware used for these calculations can be made in different ways, where i have decided to use the same hardware if possible from the Levinson-Durbin algorithm.

The most important hardware part is the MSA as this block can do most normal calculations, and is build so it can calculate the multiply substitution multiply, as used in the Levinson-Durbin, and the Multiply addition, multiply, as used for the Levinson algorithm.
This block can also do the operations separate if needed, as shown in figure 7.26.

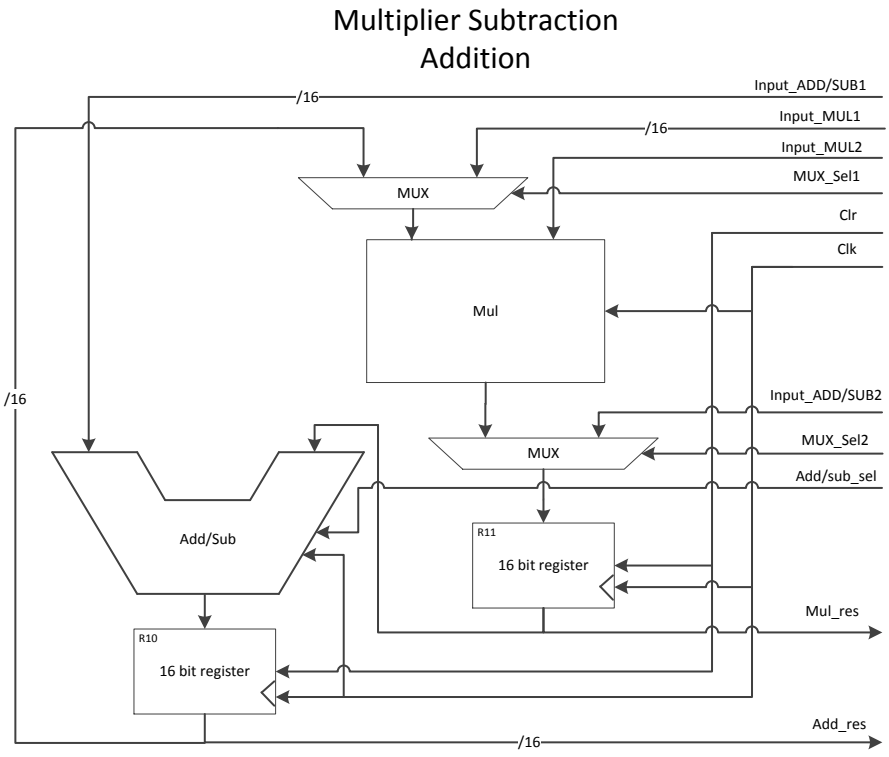

Figure 7.26: The hardware block made for the Multiplier Subtraction Addition.

The second needed part is the None restoring division, depicted in figure 7.27, which is used once for each iteration of the Levinson-Durbin, and twice for each iteration of the Levinson algorithm.

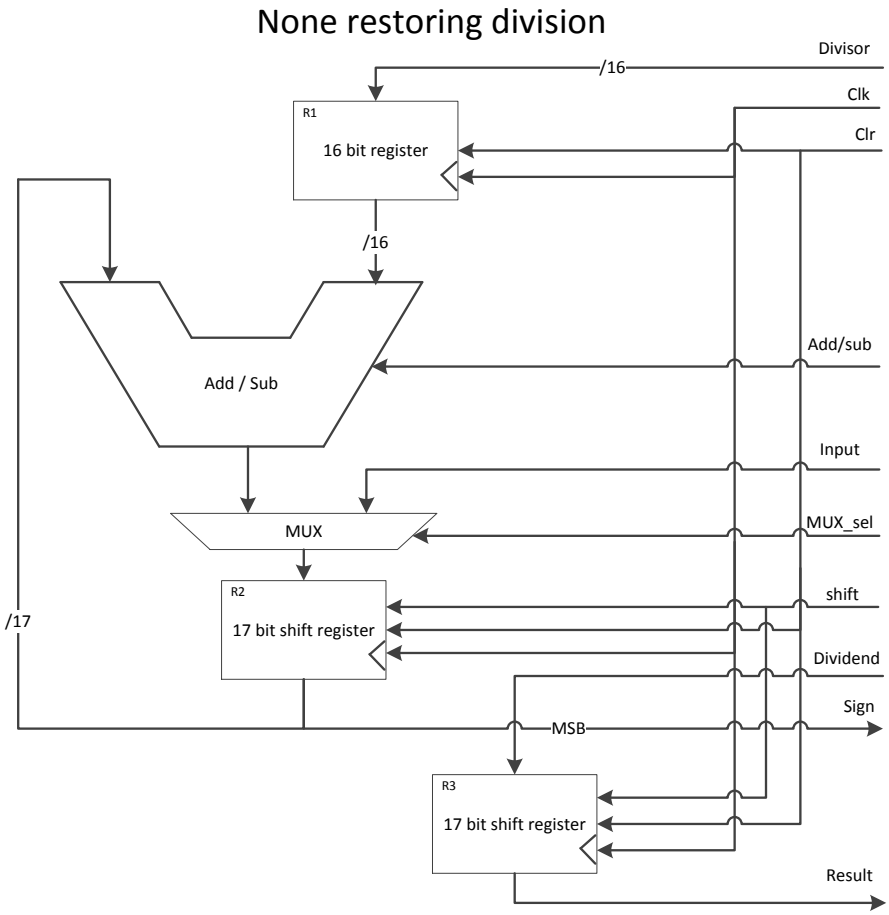

Figure 7.27: None restoring division hardware

The third needed block from earlier is the sign inverter, however since the sign inverter, can be used for the threshold function, as depicted in figure 7.28. The idea with the threshold function is that the sign of the variable is removed and the threshold value is subtracted, of the sign then changes again, the value was to low, and the value is set to 0, of the sign remains the same, then the first sign decides if it is positive or negative, and the sign is then changed accordingly.

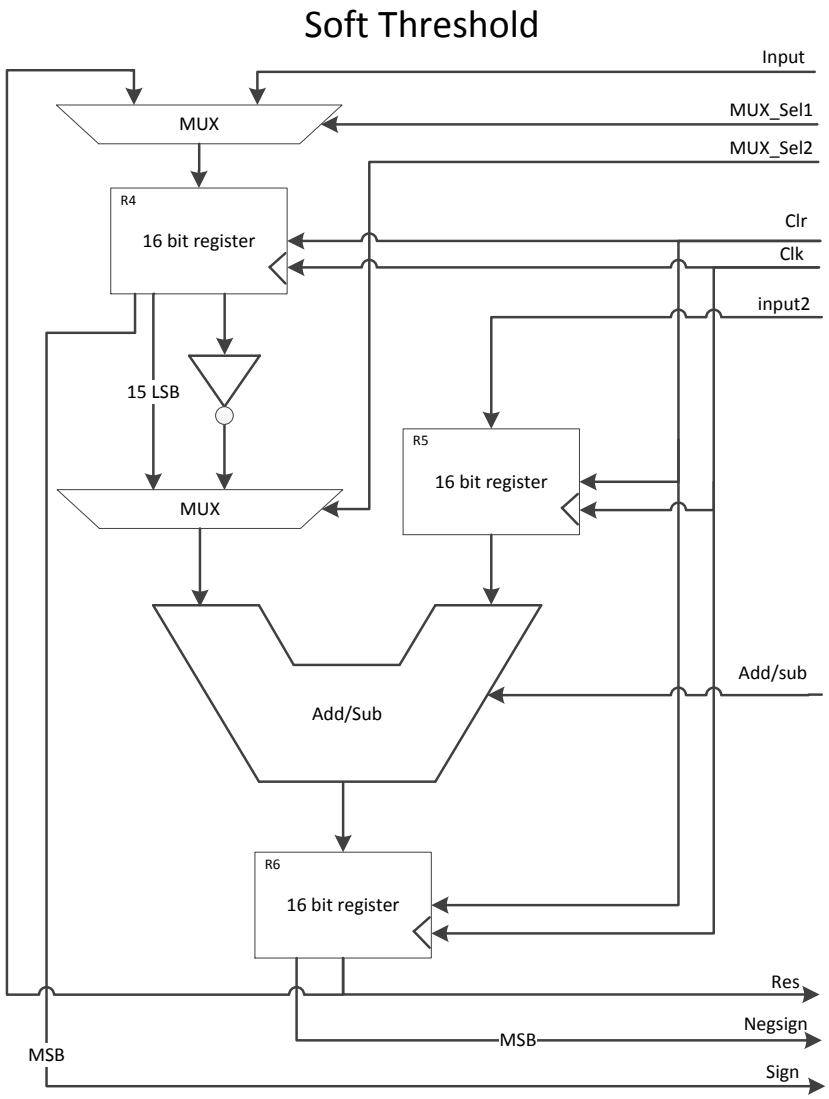

Figure 7.28: Hardware block for calculating the soft threshold function

As the threshold block and MSA are designed, with the purpose of making several of them so that it can run in parallel, such a way that there is 332 blocks of them at the same time.

7.4 Scheduling

The Scheduling for this ADMM algorithm is complex as it is so large that is has to be divided up into blocks.

7.4.1 Update A

The first part is the Levinson-Durbin algorithm as scheduled earlier and depicted in figure 6.30 where a run of the algorithm takes up 93 C-step, this algorithm is used to calculate the initial a coefficients, since it is initial calculations, means that the calculation can be done outside the loop. Then as shown earlier with 332 parallel MSA blocks, it is possible to calculate the $y - u$ in 1 C-step. The next part is the Levinson algorithm as depicted in 7.29 where the Levinson and Levinson-durbin scheduling is combined, and as seen they take up the same time plus 8 because of the extra calculation for the outer loop in the Levinson algorithm, giving 101 C-steps. This gives the conclusion that it takes 101 C-steps to each iteration to calculate the new a coefficient.

7.4.2 Update E

The next block is where the error is calculated, where as mentioned earlier in the precedence graphs section7.2 the updating step consists of a vector matrix calculating which can be done in 12 C-steps, and a vector subtracted by the result of the vector matrix calculation, which can be done in 1 C-step.

This leads to the conclusion that the Update E takes 13 C-steps.

7.4.3 Update y

The update y block is the threshold function, and as shown in my hardware design of the threshold, it can be done rather simple. So the $e + u$ takes 1 C-step, and the threshold takes 2 C-Steps, however the initial threshold value also needs to be calculated, which can be done before the ADMM algorithm. So this gives a 3 C-steps and 1 initial C-Step.

7.4.4 Update U

The block for updating the u vector takes one vector addition and a vector subtraction. So combined it takes 2 C-steps each iteration.

7.4.5 Conclusion

With the four different blocks of the algorithm, the number of C-steps can be calculated. This gives the following table:

Block	Initilization	1 iteration	40 iterations
Update A	93	101	4133
Update E	0	13	520
Update Y	1	3	121
Update U	0	2	80
Combined	94	119	4854

This also table gives the same indications as the simulation see section 4.3.3, where the *UpdateA* also took significant longer then the other parts.
Knowing the dependencies of each of the parts of the algorithm can lead to an chart showing timing of the update A block runs from start to end, this is depicted in figure7.29. Where it is seen that many of the different blocks does not run continuously, and the dependencies of input for another block keeps the functional units from being used coherently.

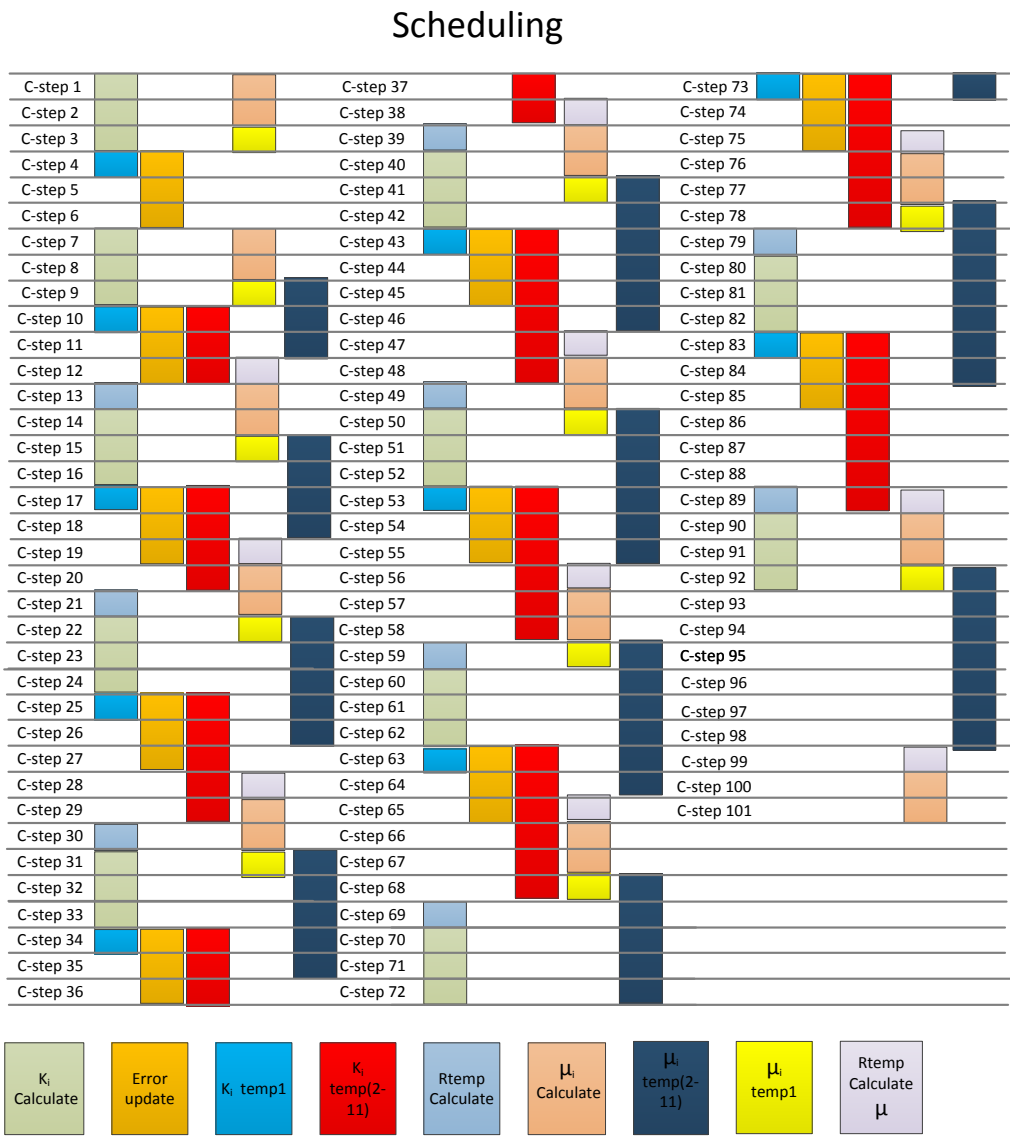

Figure 7.29: Scheduling of the Levinson Algorithm

However good results this is, it is calculations done by summing up the C-steps. In this algorithm the only part takes more clocks then C-steps is the division. As the addition, multiplication and subtraction can be calculated in 1 clock cycle.
The division takes up number of bits-1 operations to calculate the result, this means that the scheduling can be updated to go by operations instead of C-steps. From the earlier schedule of the Levinson-Durbin in figure 6.48 we know it takes 189 clocks,

when running the Levinson algorithm, it takes 12 more clocks, because of the extra calculation giving 201 clocks for the Levinson algorithm.

Block	Initilization	1 iteration	40 iterations
Update A	189	201	8229
Update E	0	13	520
Update Y	1	3	121
Update U	0	2	40
Combined	190	219	8950

As earlier the table shows the same bottleneck with the Update A, since the only part which included a division was the Update A.

7.5 Finite State Machine

While making the Finite state machine for the ADMM algorithm, it is possible to reuse many of the ASM charts from the Levinson-Durbin algorithm, as the inner loop of the Levinson Algorithm, is the Levinson-Durbin algorithm.
Therefore a ASM chart is made from the old ASM of the Levinson-Durbin algorithm. First a ASM chart is made, for the full ADMM, as depicted in 7.30. It starts with calling the old Levinson-Durbin algorithm to find the initial a_2 coefficients. After finding the coefficients, a set of loops are initialized, where the first loop decides how many times the ADMM algorithm runs. The second loop decides how many coefficients the algorithm finds. inside the loops contains two calls one calling the first iteration of the ADMM algorithm, and the other is parallel and calls the Levinson algorithm. As explained earlier both these algorithms can run in parallel.
After the two algorithms finds the first set of a coefficients the algorithm breaks out and update the error e, the vector y and u, then to reset set the counter inside the inner loop to update the coefficients again.

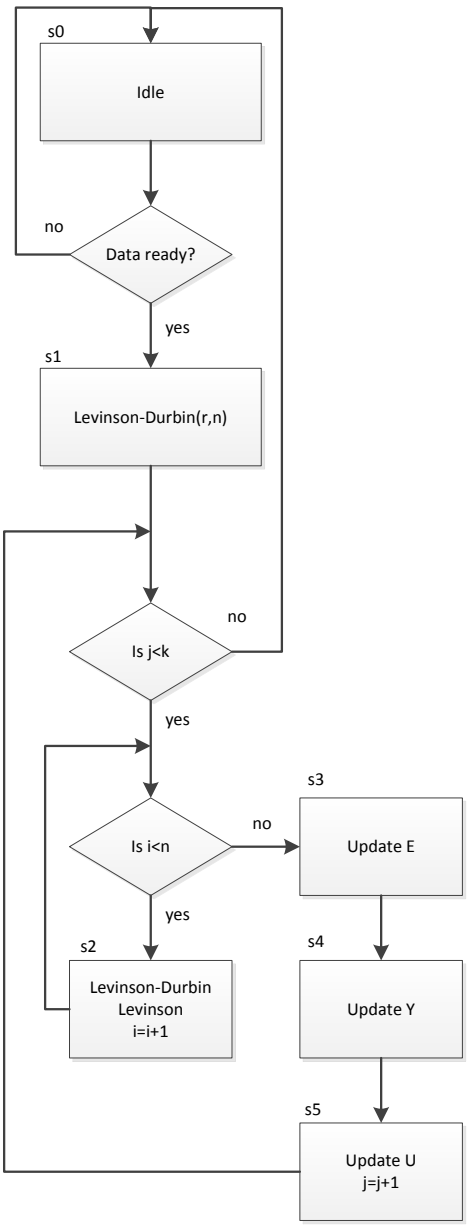

Figure 7.30: ASM chart controlling other ASM charts

The depicted ASM in figure 7.32 depicts the Master ASM chart for the Levinson algorithm, which handles the input to the Levinson algorithm, while the figure 6.35 depicts the Master ASM for running the Levinson-Durbin algorithm. Both these

algorithms calls other ASM charts which then sends the right input to the architect units. Where two of them for the Levinson algorithm are depicted in figure 7.34 and 7.35, and for the Levinson-Durbin the same figures are depicted earlier in the rapport in figure 6.40 and 6.44.

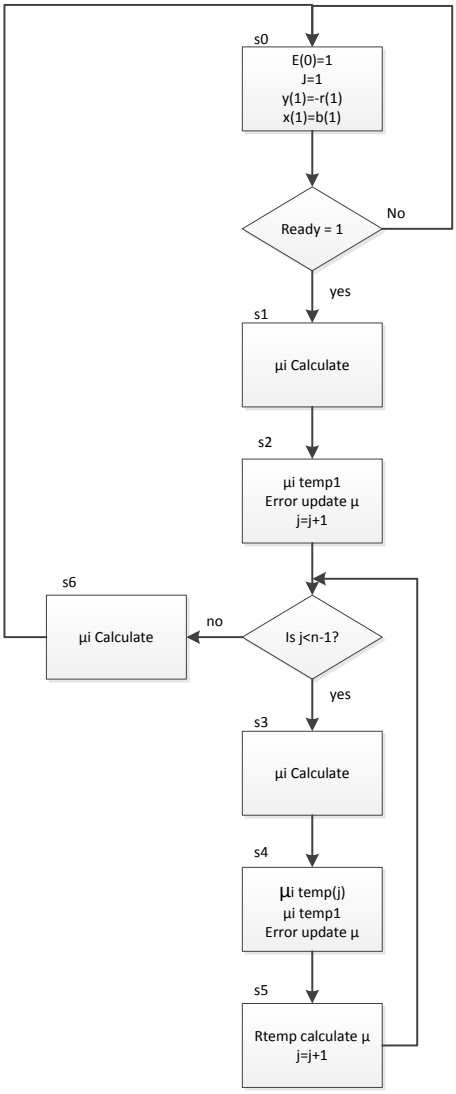

Figure 7.31: Levinson ASM chart for controlling how the Levinson algorithm runs in the outer loop

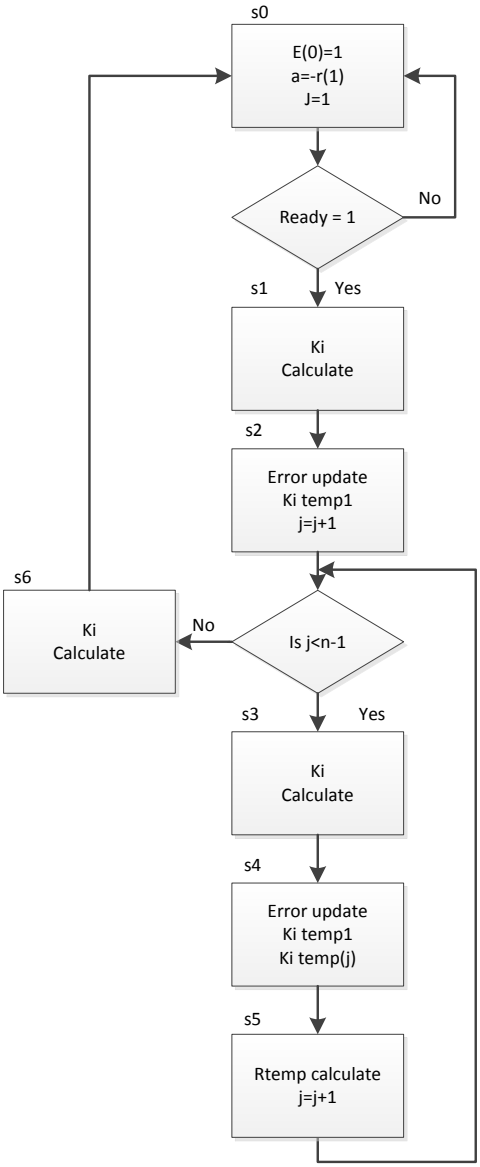

Figure 7.32: Levinson-Durbin ASM chart for controlling how the algorithm runs in the inner loop

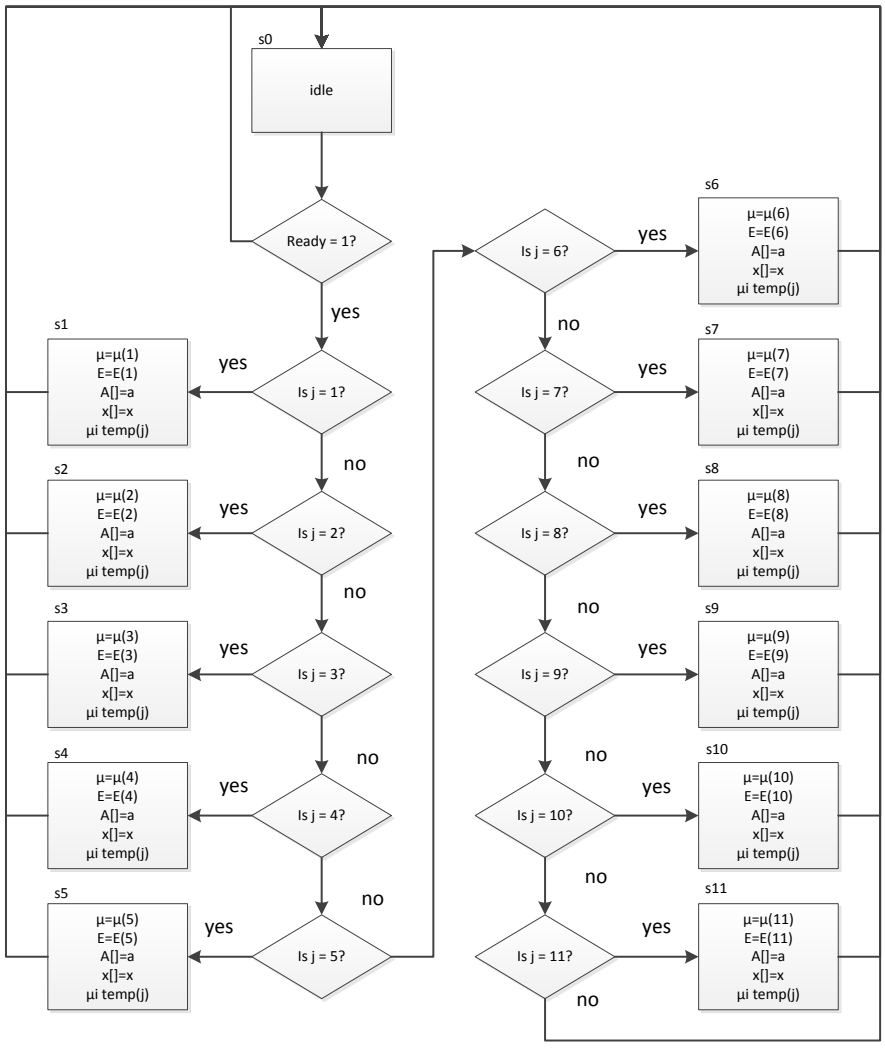

Figure 7.33: The ASM chart for controlling the input to the multiply add multiply operation in the Levinson algorithm

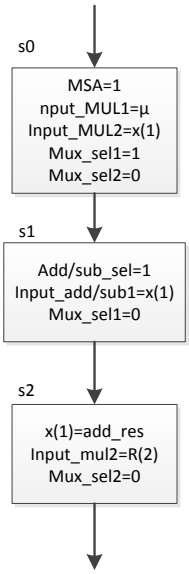

Figure 7.34: The ASM chart for controlling the input to the first multiply add multiply

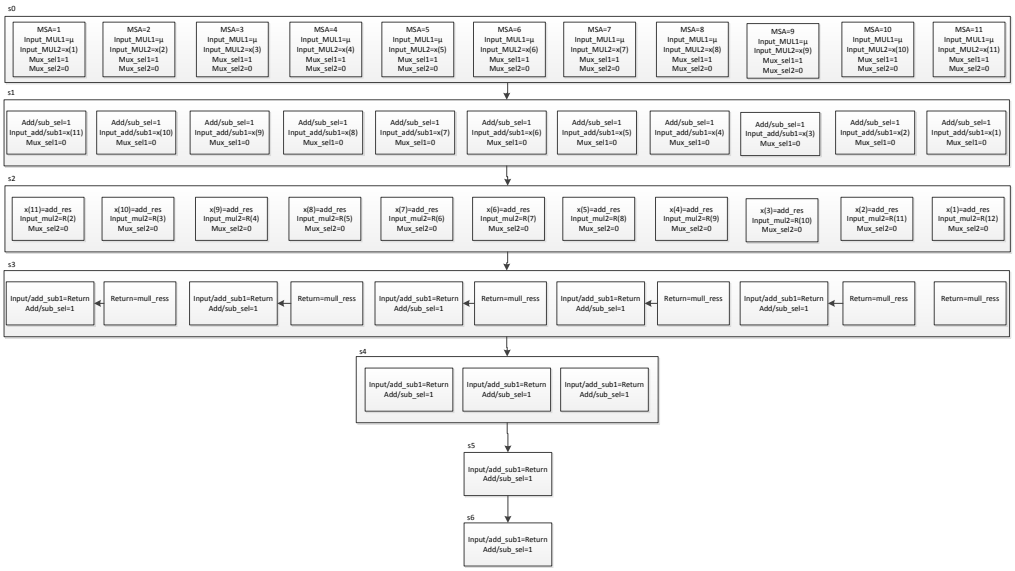

Figure 7.35: The ASM chart for controlling the input to the 11th iteration multiply add multiply

7.5.1 ASM update E

As earlier described the, then the update E calculates the error, by finding the residual of the objective function, and putting numbers into it. This is done by multiplying the X matrix with the vector a. The result is then subtracted from the vector x.
To get an overview of what happens, the CDFG from earlier is depicted in figure 7.36, where the afore mentioned operations are separated into two blocks. The first finds the temp value, which is the matrix vector multiplication. as depicted in figure 7.37, This ASM chart is a simplification of how it can be done, where a counter keeps multiplying each column of the matrix with the vector, and summed up, as described in section 7.2.1.

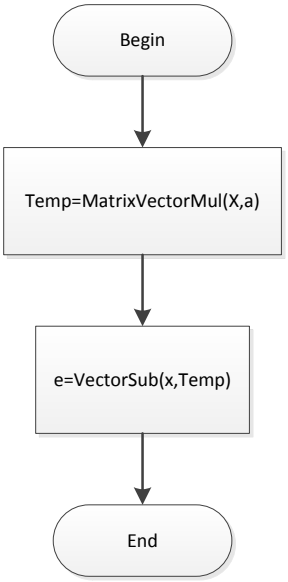

Figure 7.36: DFG describing the Update E block

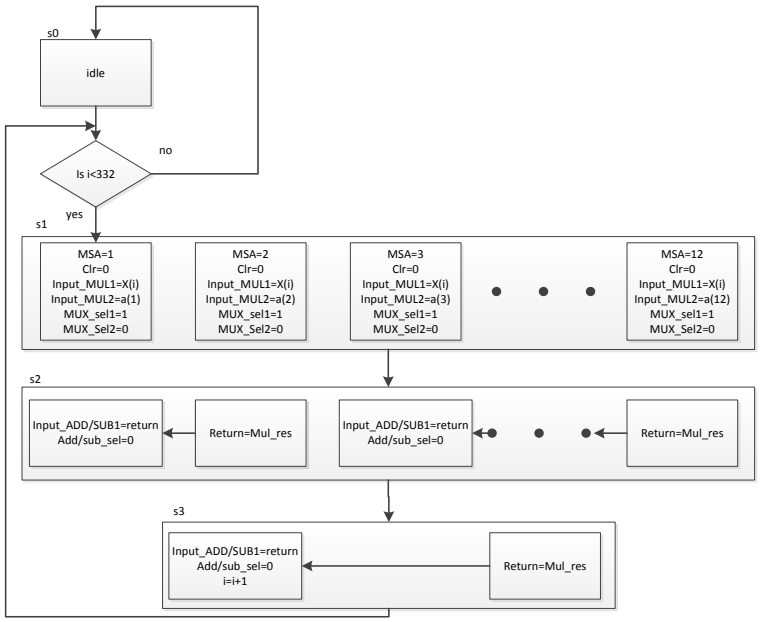

Figure 7.37: The ASM chart calculating the temporary result from multiplying the X matrix with the a vector

With the new vector in a temporary vector temp(), it is then multiplied with the vector $x()$, as depicted in figure 7.38, which then returns the new updated error vector e

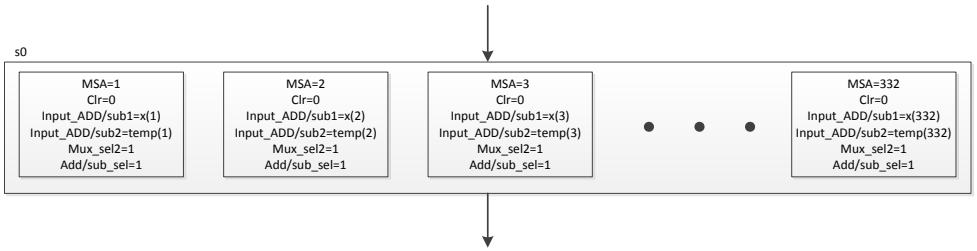

Figure 7.38: The ASM chart giving the result of calculating the error vector e

7.5.2 ASM update y

The ASM chart for update y, starts with adding the e and u vectors, depicted 7.39, where the result of this is put through a threshold function as earlier described, depicted in figure 7.40, this ASM chart controls the other ASM chart such that it can make multiply calls to parallel threshold functions. The ASM chart which calculates each element of the vector is depicted in figure 7.41.

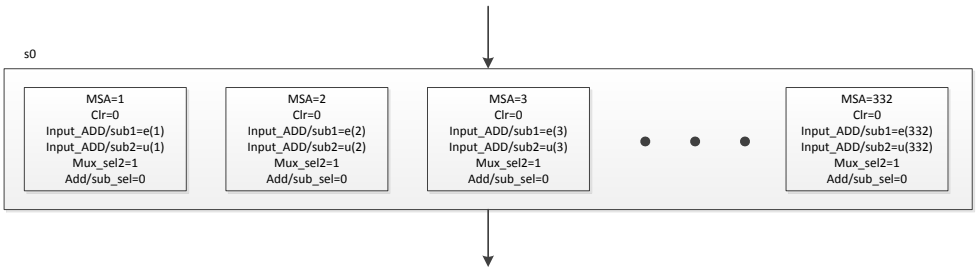

Figure 7.39: The ASM chart adding the e and the u vectors returning the temp() vector

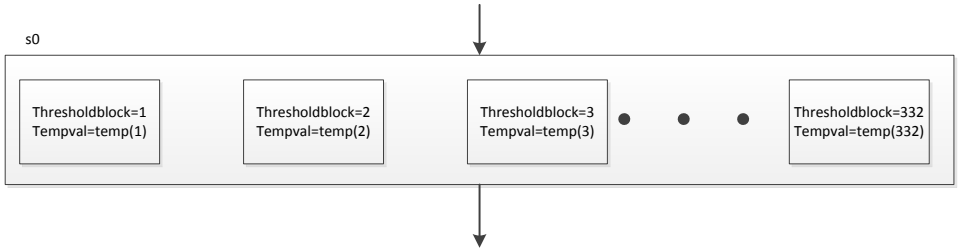

Figure 7.40: The ASM chart controlling the threshold hardware input

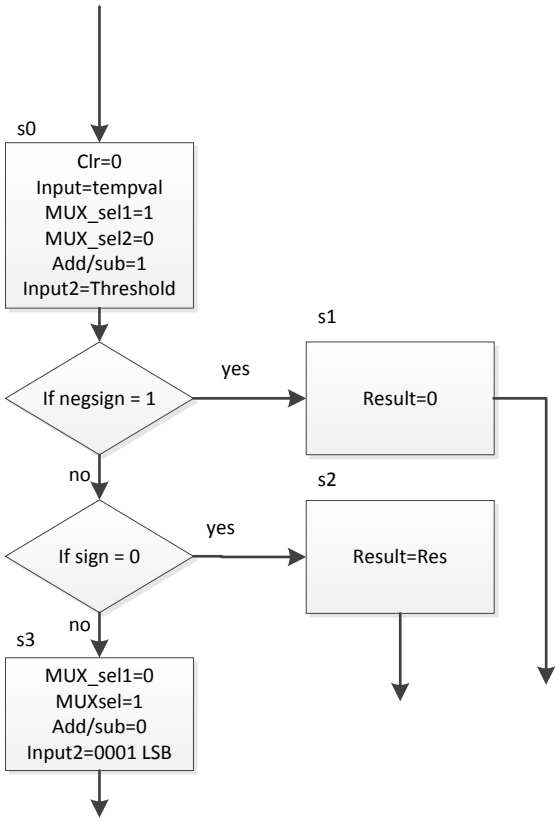

Figure 7.41: The ASM chart Sending the numbers to a specific hardware unit.

7.5.3 ASM update u

The update U ASM chart consists of two parts a vector addition and a subtraction, however as the same multiplication, with u and e have been done before as an input to the threshold function, the same variable can be reused, and since the temp() vector still contains the result from last. The second part is where the vector y is subtracted from the temp() vector, as depicted in figure 7.42.

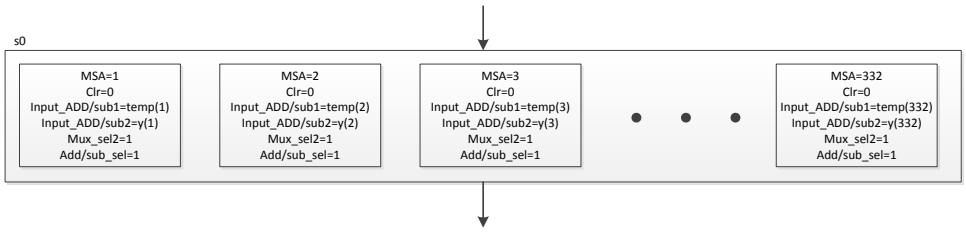

Figure 7.42: The ASM chart subtracting y from the temp() vector

7.5.4 Registers

As seen in figure 7.43 from earlier, number of registers needed are incrementing as the vector a grows for each iteration for the Levinson-Durbin algorithm. For the Levinson algorithm we have almost the same registers as with the Levinson, the difference is that the algorithm runs beside each other, where the outer loop uses the inner loop's result to calculate a new coefficient, as depicted in figure

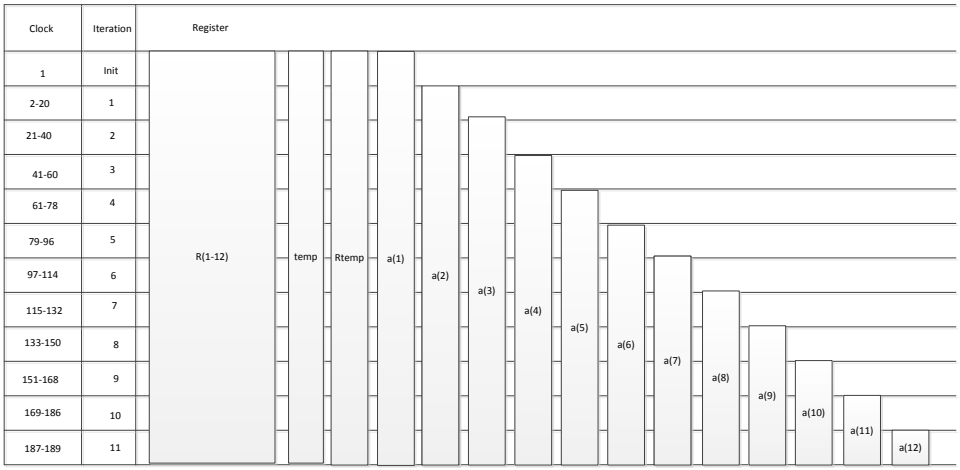

Figure 7.43: Register Lifetime analysis of the Levinson-Durbin

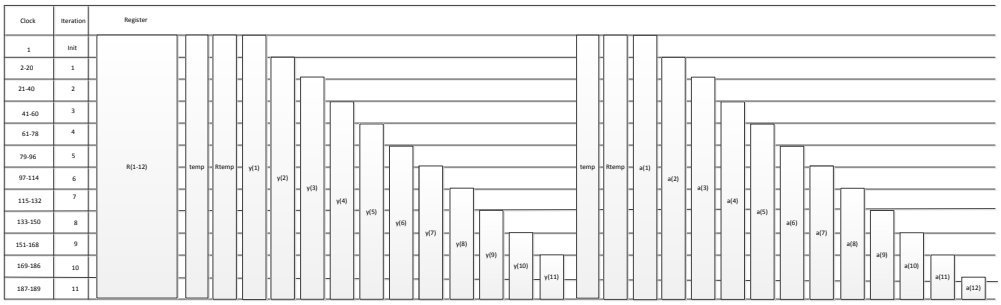

Figure 7.44: Register Lifetime analysis of the Levinson algorithm

For the ADMM algorithm the number of registers remains constant as the $x(k)$, $a(k)$, $y(k)$, $u(k)$ and $e(k)$ has the same length and is updated in each iteration into the same vector, so 5 registers of the length k is needed plus the registers for the Levinson algorithm.

7.5.5 State vector

The controlling part of the Finite State Machine is the state vector, which iterates in the states to send the signals to control the Data Path. The states for driving the control path is depicted in figure 7.45. Where there are 5 states, which is described in table:

State	Description
s0	Idle iteration, waiting for ready signal, and counter for iterations.
s1	Update A
s2	Update E
s3	Update Y
s4	Update U

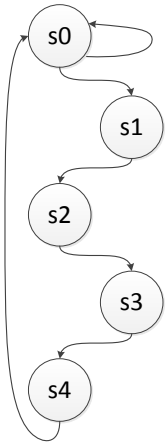

Figure 7.45: State vector for the Levinson-Durbin algorithm

These States are described earlier in section 7.5 where a set of different ASM charts is depicted for each of the states.
The following state vector depicts the flow of the Levinson algorithm, as it is a bit different from the Levinson-Durbin state vector described in section 6.4.11.

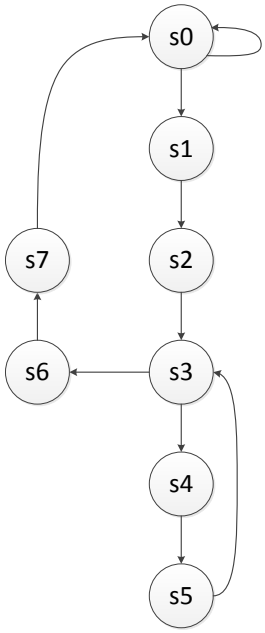

Figure 7.46: State vector for the Levinson algorithm

State	Call
s0	Idle iteration, waiting for ready signal
s1	k_i calculate, μ_i calculate
s2	update error, k_i temp1, update error μ, μ_i temp1
s3	k_i calculate, μ_i calculate
s4	update error, k_i temp1, k_i temp(j), update error μ , μ_i temp1, μ_i temp(j)
s5	Rtemp calulate, Rtemp calulate μ
s6	k_i calculate ,update error μ , μ_i temp1, μ_i temp(j)
s7	μ_i calculate

The different iterations consists of a set of calls, which is dependent on each other where each of the calls for each iteration is presented in the following table:

iteration	calls
1	k_i Calculate, k_i temp1, Error update, μ_i Calculate, μ_i temp1
2	k_i Calculate, k_i temp1, k_i temp2, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp2
3	k_i Calculate, k_i temp1, k_i temp3, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp3
4	k_i Calculate, k_i temp1, k_i temp4, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp4
5	k_i Calculate, k_i temp1, k_i temp5, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp5
6	k_i Calculate, k_i temp1, k_i temp6, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp6
7	k_i Calculate, k_i temp1, k_i temp7, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp7
8	k_i Calculate, k_i temp1, k_i temp8, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp8
9	k_i Calculate, k_i temp1, k_i temp9, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp9
10	k_i Calculate, k_i temp1, k_i temp10, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp10
11	k_i Calculate, k_i temp1, k_i temp11, Error update, Rtemp Calculate,μ_i Calculate, μ_i temp1 ,Rtemp Calculate μ, μ_i temp11
12	μ_i Calculate,Rtemp Calculate μ,

7.6 Conclusion

In this chapter a possible FSMD is found with a set of ASM and a data path comprised of a set of hardware. The FSMD gives a Scheduling for the algorithm, which takes 6759 clocks, which has to be calculated every 20 ms, which can be done without problem at 50 mHz clock, given $50.000.000 * 0.02 = 1.000.000 clocks$ for every 20 ms. As described 5 it is therefore possible to lower the clock in such a way that the calculation is done just in time for the new input. Thereby saving power.

Chapter 8

Implementation

In this chapter implementation of a FSMD as the two derived earlier in 6 and 7 is described.
As describing the full implementation is out of scope, and there is an infinite number of different ways to implement a FSMD in VHDL, therefore only a few aspects of the implementation is described.
First the top level of the design, depicted in figure 8.1, where an input is generated in the block Input, this block then sends the coefficients to a state machine inside the Durbin_cal block, finally when this state machine is done, the output is send to the store_ram, which then returns the results into the ram, and a signal is send to the input block that it is ready for a new input.

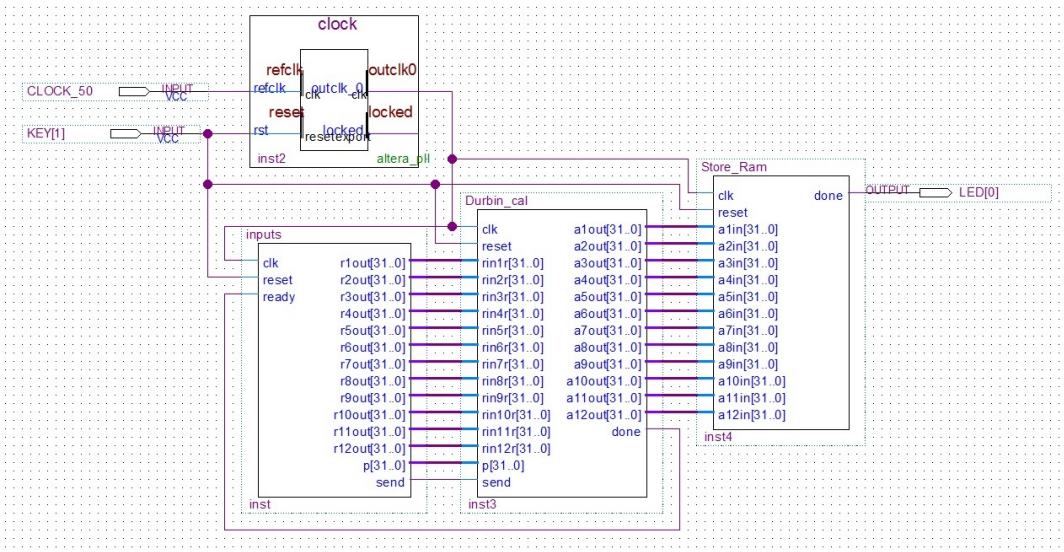

Figure 8.1: Top Level blockdiagram in vhdl.

Inside the Durbin_cal block is a state machine which iterates for 12 iterations as described in 6.4.11, a little part of a simplified code is depicted in listings 8.1 where the steps are shown with the math instead of the calls to calculate each part.
In VHDL there is many predefined blocks to do different calculations, therefore it is unnecessary to implement a specific multiplier, adder, sign inverter or divider as

Listing 8.1: Sparse LPC matlab implementation

```
case current_s is
    when s0 =>
    if (ready='1')then
         next_s <= s1;
         else
         next_s <= s0;
         end if
    when s1 =>
       temp:=a1*rin1r;
       a2:= -((temp+rin2r)/E);
       E:=(1-a2*a2)*E;
       next_s <= s0;

       when s2 =>
       a1:=(a2*a1+a1);
       temp:=a2*rin1r;
       temp2:=a1*rin2r;
       a3:= -((temp+temp2+rin3r)/E);
       E:=(1-a3*a3)*E;
       next_s <= s0;

       when s3 =>
       a1:=(a3*a2+a1);
       a2:=(a3*a1+a2);
       temp2:=a1*rin3r+a2*rin2r;
       temp:=a3*rin1r;
       a4:= -((temp+temp2+rin4r)/E);
       E:=(1-a4*a4)*E;
       next_s <= s0;
       end case;
```

described in the hardware section of the RTL analysis. Instead blocks can be dragged into a block diagram making up the different calculations. However the defined MSA hardware block depicted in figure 6.34, is specific to solve a part of both the Levinson-Durbin and Levinson Algorithms, therefore a specific block is designed for this, as depicted in figure 8.2 As seen the inputs are the same as in the figure 6.34. This block diagram turned into a symbol block, depicted in figure 8.3, which can then be used for other block diagrams where the MSA block is needed

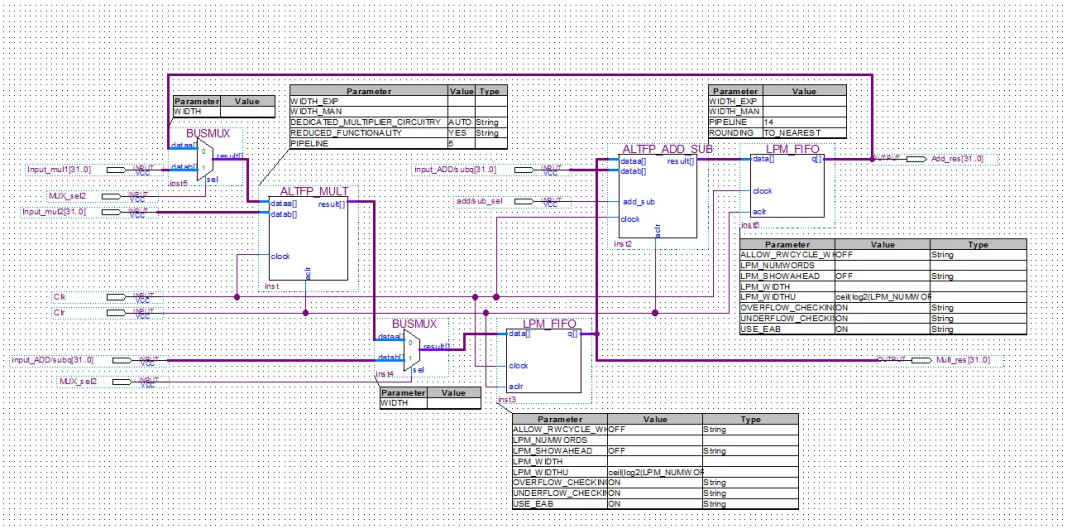

Figure 8.2: Block diagram showing the MSA block.

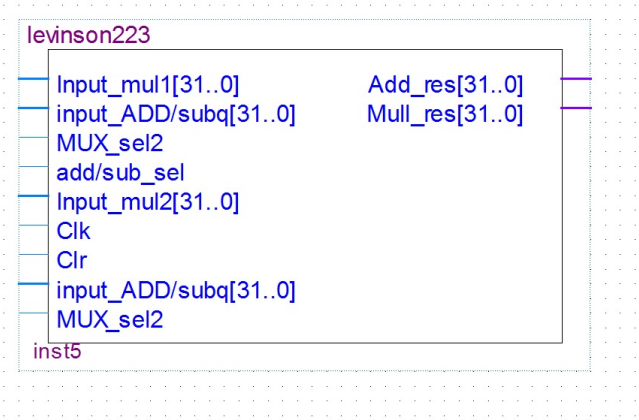

Figure 8.3: The MSA block when it is turned into a block

8.0.1 Conclusion

In this chapter a introduction into how the FSMD can be implemented in VHDL, where most of the designed hardware parts used for these algorithms are already

implemented in VHDL.

Chapter 9

Conclusion

The purpose of this project was stated by the problem statement:
Is it possible to use Sparse Linear Predictive Coding to find the filter coefficients and error, can this be implemented on a FPGA and how is the complexity of the implementation compared to a normal Linear Predictive Coding implementation.

This can be translated into being, analyse the LPC and SLPC implementations, using the Levinson-Durbin and ADMM algorithms respectively. Which can be turned into the following three Levels:

- Application
- Algorithm
- Architecture

9.1 Application

From the introduction the Application is given as finding a set of Linear Predictive Coefficients and a residual which can represent a given input signal. There exists various ways to find these coefficients, in this project it was decided to use the Levinson-Durbin algorithm. When making a sparse application there is some possibilities, the object of making the application sparse can be put on, either the Coefficients, the residual or both. In the report the formula for finding a sparse residual and coefficients were stated. However it were decided to only make the Residual sparse, as it simplifies the implementation.
The difference is the implementations of making both coefficients and residual sparse, and sparse residual is the vectors and matrices in and out puts. This means that the same set of operations are used for the proposed implementation ADMM algorithm and where the object is to make the whole signal sparse.
With the given two algorithms a Matlab simulation setup were made, to give an insight into the works of the algorithms.

9.2 Algorithm

The as the algorithms was decided as being the Levinson-Durbin and ADMM, these two are analysed. Where the Levinson-Durbin is strait forward with a set of operations for a given number of iterations, which in turn was turned into a set of precedence graphs. The ADMM algorithm were more complicated as it takes the input from a normal Linear Prediction. Inside the Algorithm the difference right hand side least squares problem is solved, Which were decided to be solved using the Levinson Algorithm, in the other steps of the ADMM algorithm a set of vector operations are used, and a soft threshold function. Given these operations to make up one iteration of the ADMM algorithm, a set of precedence graphs and flow diagrams were made.
Solving the least squares problem for the toepliz matrix inside the ADMM, a set of solutions were proposed. Where the Levinson Algorithm proved to take the fewest operations, and given the Levinson-Durbin architecture already given it proved simpler to implement.

9.3 Architecture

The Architectures implemented in this report for the Levinson-Durbin and ADMM were build upon the given Precedence graphs and flow diagrams. The architecture of the Levinson-Durbin algorithm is reused for implementing the ADMM algorithm as the input to the algorithm is a set of normal Linear Predictive Coefficients. Another part where some of the Levinson-Durbin algorithm is used, is while implementing the Levinson algorithm. This is done because the Levinson-Durbin is calculated implicit by the Levinson algorithm in the inner loop for every iteration. While further analysing the Levinson algorithm it is shown that there is little difference between the inner and outer loop of the Levinson Algorithm, and that they can run in parallel, therefore it is possible to use most of states for the algorithm. The architecture for the Levinson-Durbin algorithm and ADMM ends up giving a Datapath in which all calculations are done, and a FSM, given by a cluster of ASM charts which controls the signals to and from the Data path.

9.4 Future

While the given architectures can be implemented in a given FPGA, the architectures can be further optimized which is explained in the following two subsections 9.4.1 and 9.4.2.

9.4.1 Possible Optimization

When implementing a RTL architecture it is possible to run some optimizations. One of the optimizations is to see of it is possible to share registersDaniel D. Gajski. This

is possible as already described in 6.4.10 where the initial registers are shown and how register sharing is possible. Another possibility is to use register merging, this is where it is possible to use a register as for two variables, this is possible if the variables has none-overlapping access times. however as described in [Daniel D. Gajski, p.227], however this lowers the access time to the register, since it needs a longer address to allocate which variable is requested. More interesting is the possibility of doing chaining and multi-cycling [Daniel D. Gajski, p.229]. chaining means that two or more functional units or, arithmetic units are used in the same cycle, without storing the intermediate result. The other possible way was to do multi-cycling, this is where slower units are used to calculate none critical computations, meaning that an operation which normally takes 1 clock cycle, is slowed down, so a result is first calculated after tow or more cycles. This is interesting for this project, as there are many none critical paths in the calculation of both the Levinson-Durbin algorithm, and when calculating the ADMM algorithm. It is seen in the scheduling in 7.29 and 6.30 that many of the operations stands still while the critical operations runs.
While optimizing the number of functional units, it is possible to look into Functional Unit Sharing, this is where the possible number of functional units are decreased by sharing the units between the different calculations. It is also possible to implement connection sharing, this is done so that the number of connections between registers and functional units are shared and used more efficiently, and if possible the number if connections are decreased.
Going further into the possibilities of optimizing the RTL, then it is possible to do functional unit pipe-lining, this is where the functional takes a variable, do a partial calculation, before taking in a new number, while it finishes the calculation. By doing this, it is possible to decrease the delay between the output variables, however the throughput through the functional unit remains the same. While it also acts as a form of register being able to hold variables in clue while calculating the current result.

9.4.2 Possible Scheduling

When implementing a RTL architecture it is important to consider different ways to schedule the architecture.
For this project it was decided from the beginning to use ASAP scheduling, as the goal of the implementation is to implement a real-time implementation of voice encoding. The idea behind ASAP scheduling is to put unlimited hardware into the solution, and the run the code as fast as possible. The good thing with this solution is that it is then possible to decide at the end to lower the clock frequency of the FPGA, in such a way that the calculations are done just in time.
Instead of using ASAP, some parts of the architecture could be implemented with another scheduling form of scheduling to make use of the hardware in a better way, a few of the possible ways of schedule is, Force directed scheduling, As Late As Possible (ALAP), list scheduling.
The Force directed scheduling is useful when there is constrain on hardware and

time, then the scheduling can be run to see if it is possible to schedule given the time. The algorithm uses ASAP and ALAP to find out possible scheduling then to spread out the calculations on the arithmetic in such a way that it takes the least time given the time constraint.
The ALAP is a good way to schedule when there is a time constraint, without looking at hardware.
List scheduling is a way to schedule where a given priority list for each operation, then the algorithm keeps running to find a possible scheduling list where the calculations given the highest priority is scheduled first.
Given that the scheduling for the implementation is decided by the synthesis tool in the given program, it was also decided to use a simple scheduling, given that there is no hardware constraint, and that the clock frequency can then be lowered.

Bibliography

Wai C. Chu. *Speech Coding Algorithms: Foundation and Evolution of Standardized Coders*. WILEY, first edition edition. ISBN 0-471-37312-5.

Andreas Gerstlauer Gunar Schirner Daniel D. Gajski, Samar Abdi. *Embedded System Design*. Springer, first edition edition. ISBN 978-1-4419-0504-8.

Daniele Giacobello. Sparsity in linear predictive coding of speech. http://vbn.aau.dk/en/publications/sparsity-in-linear-predictive-coding-of-speech(384fab52-6717-49ab-babe-ef9b13d9ef3f).html, 2010.

Charles F. Van Loan Gene H. Golub. *Matrix Computation*. The Johns Hopkins University Press, third edition edition. ISBN 978-0801854149.

GeorgeCybenko. The numerical stability of the levinson-durbin algorithm for toeplitz systems of equations. SIAM Journal on Scientific and Statistical Computing 1, 303 (1980), 1980.

Hung Ngo, Mehrub Mehrubeoglu. Effect of the number of lpc coefficients on the quality of synthesized sounds. Academic Journal, 2010.

John G proakis john R Deller jr, John H L Hansen. *Discrete-Time Processing of Speech Signals*. New York : Institute of Electrical and Electronics Engineers, 1999. ISBN 0780353862.

David Bau III Lloyd N. Trefethen. *Numerical Linear Algebra*. SIAM, first edition edition. ISBN 978-0.898713-61-9.

John Makhoul. linear prediction: A tutorial review. PROCEEDINGS OF THE IEEE, VOL. 63, NO. 4, APRIL 1975, 1975.

Hesham El-Rewini Mostafa Abd-El-Barr. *fundamental of computer organization and architecture*. Wiley, 2005. ISBN 978-0-471-46741-0.

P. G. MARTINSSON, V. ROKHLIN AND M. TYGERT . A fast algorithm for the inversion of general toeplitz matrices. http://www.sciencedirect.com/, 2005.

Nicholas L. Pappas. *Digital Design*. WestPublishing Company, 1994. ISBN 0314012303.

Tobias Lindstrøm Jensen, Daniele Giacobello, Toon Van Waterschoot and Mads Græsbøll Christensen . Fast algorithms for high-order sparse linear prediction with applications to speech processing, 2015.

Tokunbo Ogunfunmi and Madihally J. (Sim) Narasimha. Speech over voip networks. IEEE CIRCUITS AND SYSTEMS MAGAZINE, 2012.

P. P. Vaidyanathan. *The Theory of Linear Prediction*. Morgan and Claypool Publishers, first edition edition. ISBN 1598295756.