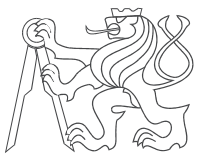
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

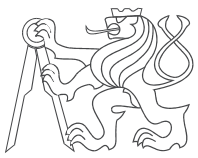


Implementation Methods of LD-RLS with Directional Forgetting for Embedded Systems on a Chip

DOCTORAL THESIS

August 2010 Ing. Roman Bartosiński

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering



Implementation Methods of LD-RLS with Directional Forgetting for Embedded Systems on a Chip

by

Ing. Roman Bartosiński

Supervisor: Ing. Jiří Kadlec, CSc

Dissertation submitted to the Faculty of Electrical Engineering of
Czech Technical University in Prague
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy (Ph.D.)

in the branch of study
Control Engineering and Robotics
of study program Electrical Engineering and Informatics

August 2010

PhD Programme:

Electrical Engineering and Information Technology

P 2612 Elektrotechnika a Informatika

Branch of Study:

Control Engineering and Robotics

2612V042 Řídicí Technika a Robotika

Supervisor:

Ing. Jiří Kadlec, CSc

Department of Signal Processing

Institute of Information Theory and Automation

Academy of Sciences of the Czech Republic

Copyright ©2010 Roman Bartosiński

Preface

This work was carried out in the joint research between the Department of Control Engineering, Czech Technical University in Prague and the Department of Signal Processing, Institute of Information Theory and Automation of the ASCR.

The university supported me during my postgraduate studies. As a researcher at the Department of Signal Processing I have had a great opportunity to work with the latest development tools and I also participate in several projects which have brought me knowledges useful for working on the thesis. Namely they were:

the European project **RECONF2** which provided me with a deeper insight to FPGAs and their dynamical reconfiguration.

the Czech project **SESAp** which enabled me to increase knowledges about Matlab/Simulink environment and automatic code generation from this tools.

the European project **ÆTHER** which brought me knowledges about multi-core embedded systems and their possible programming.

the European project **Apple-CORE** with knowledges about detailed internal structures of some microprocessors.

the European project **SMECY** focused on multi-core embedded systems.

The last project SMECY works with UTIA DSP platform, which is also used in this thesis. It allowed me to work on the thesis and to develop a software library and automatic code generator for the platform.

Acknowledgement

First and foremost, I wish to express my appreciation to my supervisor, Ing.Jiří Kadlec,CSc., for his shared experience, guidance, and patience. Additionally, I thank to my colleagues at the department of Signal Processing, Institute of Information Theory and Automation for providing stimulating environment, and special thanks go to Martin Daněk for his

proofreading services.
I am also grateful to Sir Terry Pratchett for his books and particularly his Discworld series. It is the one and only world where everything is possible (SQUEAK).

Finally, I am forever indebted to my wife Pavla and my parents for their understanding, endless patience and encouragement when it was most required.
Thank you.

Roman Bartosiński

Institute of Information Theory and Automation of the ASCR
Prague, August 2010

Implementation Methods of LD-RLS with Directional Forgetting for Embedded Systems on a Chip

Ing. Roman Bartosiński

Supervisor: Ing. Jiří Kadlec, CSc

The thesis deals with an implementation of the recursive least squares (RLS) based on the LDU decomposition (LD-RLS) with directional forgetting.

Today's implementations of adaptive algorithms for embedded systems use mainly the least mean square (LMS) algorithms for their simplicity and low computational complexity which result in high speed and throughput. RLS algorithms aren't so often used for their higher computational complexity.

The LD-RLS algorithms can be attractive for control applications to identify an unknown system or to track time-varying parameters. Solution of the LD-RLS algorithm directly contains the estimated parameters. It also offers the possibility to use a priori information about the identified system and its parameters.

Directional forgetting (DF) is an alternative to exponential forgetting (EF). DF was devised 25 years ago, but it is completely omitted in current implementations of RLS algorithms. It's meant mainly for more complicated computation and thus higher computational complexity. It is omitted despite it ensures more robust identification in systems with insufficiently excited inputs.

A possible implementations of the EF LD-RLS and DF LD-RLS algorithms were discussed in the thesis. Implementations for a systolic array were discussed to demonstrate the high data dependences in LD-RLS algorithms. The second architecture was used for the implementation of LD-RLS algorithms. It is based on the UTIA DSP platform which has been developed recently at the department of Signal Processing, ÚTIA AV ČR. The platform is designed to be a highly reconfigurable hardware accelerator for systems on a chip based on FPGAs. The thesis describes the extension of the platform with new features for implementation of DF LD-RLS. Another objective in the thesis is to improve the implementation methodology for the platform. Automatic code generation was developed and used to speed up the development process of implementation algorithms on the platform. Finally,the EF LD-RLS and DF LD-RLS algorithms were implemented in the hardware and compared with corresponding versions in software.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms and Symbols

Acronyms

ASIC	Application Specific Integrated Circuit
ClC	Clock Cycles
CPU	Central Processing Unit
DF	Directional Forgetting
DF LD-RLS	RLS based on the LDU decomposition with directional forgetting
DSP	Digital Signal Processing
EDK	Xilinx Embedded Development Kit
EF	Exponential Forgetting
EF LD-RLS	RLS based on the LDU decomposition with exponential forgetting
ES	Embedded System
FP	Floating Point Arithmetic
FPGA	Field-Programmable Gate Array
FxP	Fixed Point Arithmetic
HW	Hardware
LD-RLS	RLS based on the LDU decomposition
LMS	Least Mean Square
LNS	Logaritmic Arithmetic
MAC	Multiply-Accumulate function

MIMO	Multiple Input Multiple Output system
MISO	Multiple Input Single Output system
MSE	mean squared error
NoC	Network on a Chip
PE	Processing Element
PU	Processing Unit - structure which can be consist of PEs
RLS	Recursive Least Square
RNS	Residue Number System
SA	Systolic Array
SEN	system error norm
SIMO	Single Input Multiple Output system
SISO	Single Input Single Output system
SNR	signal noise ratio
SoC	System on a Chip
XCG	Xilinx CORE Generator
XSG	Xilinx System Generator

Symbols

$\mathbf{\Theta}$	Vector of parameters
$\mathbf{z}$	Data vector
$\mathbf{d}$	Extended data vector
$\mathbf{C}$	Covariance matrix
$\mathbf{V_z}, \mathbf{R}$	Information matrix $V_z = C^{-1}$
$\mathbf{V}$	Extended information matrix
λ	Scalar forgetting factor
$\hat{.}$	denotes estimation of the variable

Chapter 1

Introduction

Adaptive algorithms are used in many fields of human activities. Many adaptive algorithms have been devised, described and implemented in the last 40 years. They are implemented in hardware devices or software programs to adapt parameters of behaviour of the system in unknown or time-varying conditions in the application or its environment. More specifically, in control and *digital signal processing* (DSP) systems they are used for their ability to change the behaviour of the controller or filter according to the incoming signals and the environment of the system. The most frequent applications of adaptive algorithms in these domains are system identification, noise and echo cancellation and signal enhancement.

The linear systems or systems with linear models based on the *moving average* (MA), *autoregressive* (AR) or *autoregressive moving average* (ARMA) models mostly use adaptive algorithms based on the *least mean square* (LMS) or *recursive least squares* (RLS).

The standard or modified LMS algorithm is usually used in DSP applications where up to hundreds of parameters are adapted. The main advantage of the LMS algorithms is their simplicity, and so their implementation is computationally simple with the computational complexity $\mathcal{O}(n)$ (in other words they are fast). On the other hand, their main disadvantage is a slow convergence rate.

Therefore many applications use the RLS algorithm and its modifications. This algorithm is more complicated, and in general its computational complexity is $\mathcal{O}(n^2)$. There are

some modifications of the RLS algorithm known as the *fast RLS* with the computational complexity only $\mathcal{O}(n)$, but they are prone to numerical instability, which makes them unreliable.

Because the RLS algorithm is recursivelly computed in a finite-word length arithmetic, it can suffer from numerical instability of updated statistics due to round-off errors. To avoid this inconvenience the computed statistics aren't updated directly, but a decomposition of covariance matrix is updated. There are several decompositions used with the RLS algorithm: QR, LU, Cholesky, LDU and UDL decomposition are examples of such algorithms. The most frequently used decomposition is QR which is very suitable for parallel and pipelined processing. Therefore it is often used when implementing RLS in hardware parallel structures like *field programmable gate arrays* (FPGAs).

The LD-RLS algorithms are based on the LDU decomposition. The LDU decomposition has some advantages – it doesn't need to compute the roots, and it includes the estimated parameters directly in the solution of the decomposition. The second issue brings one more advantage – a priori estimation of parameters with their uncertainties can be used directly to initialize the algorithm. This property can be very useful in control applications.

The RLS has been used for a long time as a simple alternative to Kalman filtering for tracking time-varying parameters. The Kalman filter as an estimator is well known to be optimal under the state-space model assumption. One of the shortcomings of the Kalman filter is the requirement of complete prior knowledge of the state-space model and its parameters.

The standard RLS algorithms are not directly applicable for parameter estimation when the unknown parameters vary with time, but they can be used with discounting (forgetting) old data. There are many variants of discounting methods. One of the most simple methods is the *exponential forgetting* (EF) and it is based on using a time- and data-invariant scalar forgetting factor, then old information is discounted uniformly. This uniform discarding can lead to a numerical instability called "wind-up" when the system is not sufficiently excited.

Several methods to avoid this wind-up has been devised. One of them is the *restricted exponential forgetting* named also *directional forgetting* derived in (Kulhavý, 1983). This method forgets old information only in the direction in which new data bring new informa-

tion. The concept of the directional forgetting exists in several variants, but none of them is used widely in practical applications. It's meant mainly for more complicated computation and thus higher computational complexity. It also adds more data dependences in the computation, and then the RLS algorithm with the directional forgetting cannot be parallelized or pipelined in a simple way.

Many *embedded systems* need for their function some adaptive algorithm, therefore the implementation of an adaptive algorithm in embedded systems is often one of important tasks in the design of such systems.

An embedded system is a special-purpose system designed and embedded in a larger system, where it is used to perform dedicated functions like data or signal processing or system control. Today's embedded systems are based on more universal microcontrollers or microprocessors which are used as the cheapest solution. Another component used as the core of embedded systems is an *application specific integrated circuit* (ASIC). ASICs are used mainly in consumer electronic, because such systems are most profitable in large volumes. Systems with ASICs (and partially with microcontrollers) provide a fixed set of functions which cannot be changed or upgraded.

The embedded systems often use programmable logic devices like FPGAs for some specific functions, mainly for signal or data processing accelerators. The FPGAs have been often used for building the whole *system on a chip* (SoC) recently. Embedded systems with SoCs became very popular for their main advantages, which are flexibility, high speed, high throughput, dependability(fault-tolerant systems), parallel processing, dynamic reconfiguration and other. The SoCs based on FPGAs have another big advantage – they can be simply configured as a multiprocessor system or a *network on a chip* NoC. Systems with many *processing units* (PU) or *processing elements* (PE) allow to use the parallel implementation of algorithms. One of the possible parallel implementation is a *systolic array* (SA) which is a regular network of interconnected processing elements.

1.1 Objectives of the Dissertation

This work focuses on the implementation of the recursive least squares algorithm based on the LDU decomposition with directional forgetting.

Directional forgetting and the LDU decomposition of RLS are not commonly used in practice. It is due to their higher computational complexity which can decrease their advantages against classical RLS with exponential forgetting. The main disadvantage of the directional forgetting (DF) method is its higher data dependence which is added to the RLS algorithm, it means that an RLS algorithm with DF cannot be implemented in pipelined parallel while the same RLS algorithm with EF can. The hypothesis based on the analysis of the DF method is that parallel implementation of the DF LD-RLS algorithm can be only partially possible. To verify this hypothesis we will try to design DF LD-RLS on a systolic array as one of the possible parallel structures.

A set of independent hardware accelerators connected to a processor can be regarded as another parallel platform. Such a platform is the UTIA DSP platform which has been developed recently. It is a flexible, re-programmable and reconfigurable hardware accelerator. In the basic version it contains pipelined floating-point operations such as addition, multiplication and division. The platform has been developed as an accelerator for the soft-core processor MicroBlaze for SoCs in the Xilinx FPGAs. To use this platform, it must be extended for implementation of the DF LD-RLS algorithm. Because the UTIA DSP platform is relatively new, the methodology for implementating algorithms on this platform is not finished. Then the improvement of the present methodology can be identified as one of the next objectives.

The current development in computing architectures is aimed at using many small processing cores with basic mathematical operations in one chip and their connection by on-chip network (Silvano et al, 2010). The UTIA DSP platform can be used as such a processing core, therefore possibilities to implement LD-RLS in such systems should be discussed.

Summary of Objectives

This summary shows the objectives of this thesis

- To summarize theoretical background for implementation of recursive least squares based on the LDU decomposition with directional forgetting.
- To design a structure of LD-RLS with directional forgetting mapped to architecture based on systolic arrays.
- To extend the vector-like UTIA DSP platform with functions required for implementing LD-RLS with directional forgetting.
- To implement effectively LD-RLS with directional forgetting on the vector-like UTIA DSP platform.
- To improve the implementation methodology for the UTIA DSP platform.
- To develop tools for the improved methodology.
- To discuss the possibility to use the UTIA DSP platform in multi-core systems like a NoC.

1.2 Structure of the Dissertation

The next chapters treat the following topics:

Chapter 2 presents the survey of the developed methods for recursive least squares algorithms and their implementations. An emphasis is put on the directional forgetting algorithm which is an alternative to the exponential forgetting that prevents numerical instability known as estimator wind-up. The second part of the chapter contains a description of embedded systems and especially system on a chip implemented in FPGAs. Information about arithmetic used in embedded systems is presented at the end of the chapter.

Chapter 3 contains background information about adaptive algorithms used in linear models of unknown systems. One of them is described in detail, it is the *recursive least squares* (RLS) algorithm based on the LDU decomposition. An essential part of the chapter deals with forgetting methods used in adaptive algorithms. The chapter ends with a discussion of the correct comparison of different adaptive algorithms.

Chapter 4 contains a description of implementations of LD-RLS algorithms on two platforms. The first of the two architectures is the well known systolic array. The second architecture is the UTIA DSP platform which has been developed recently. The platform with pipelined floating-point operations such as addition, multiplication and division is used and extended in this work. In this chapter, an implementation of the LD-RLS algorithm for an embedded system is discussed. The chapter ends with a discussion of ways to implement systolic arrays in the network of UTIA DSP platforms used as the processing elements.

Chapter 5 summarizes the results reached in the work. Parameters of the modified UTIA DSP platform and implemented LD-RLS algorithms on the platform are presented. The chapter also contains several experiments with test cases - the RLS algorithm based on the LDU decomposition with directional forgetting used in system identification and tracking slowly time-varying parameters.

Chapter 6 contains the conclusions of the dissertation, where the results are summarized, dissertation objectives are evaluated.

Chapter 2

State of the Art

Many adaptive algorithms for control and DSP applications have been devised, described and implemented in the last 40 years. In this chapter, historical development of adaptive algorithms with directional forgetting, which is an alternative to exponential forgetting, is presented. Both forgetting algorithms used for time updates in RLS algorithms have advantages and drawbacks which are mentioned.

The algorithm will be implemented on a system-on-a-chip built in an FPGA and so information related to embedded systems and FPGAs are in this chapter. Automatic code generation will be used to speed up the development process of implementation algorithms on the selected UTIA DSP platform, therefore some references to this issue are provided at the end of the chapter.

2.1 Adaptive Algorithms

Adaptive identification and parameter tracking require the use of on-line identification techniques. Variations of the working conditions may change the model parameters and therefore an estimator designed for the time-varying case should be used. Such an estimator can be based on the standard recursive least squares method supplemented with an algorithm for discounting (forgetting) old data. The common and mostly used algorithm is *exponential forgetting* (EF) which is based on assumption that recent information is more

valuable than older data. The EF algorithm forgets old data uniformly in all directions of a data vector.

The *recursive least squares* algorithm based on the *LDU decomposition* of the covariance matrix with *exponential forgetting* (EF LD-RLS) suitable for systems based on microprocessors with small memory consumption has been published in (Peterka, 1982).

The main drawback of the EF method is called *wind-up*, and it comes when a data vector is not persistently exciting, i.e. when it does not carry sufficient information. The old data is discounted continuously, but only a part of the old data can be replaced by new data. As a consequence, some eigenvalues of the covariance matrix will tend to be zero and the Kalman gain will tend to be unbounded. In that case the algorithm is very sensitive to noise and thus the estimation may be completely unreliable.

One of the methods to avoid the EF windup is *directional forgetting* DF which is attractive for its potential performance and algorithmic simplicity. The disadvantage of the DF technique is its higher data dependence in algorithm than for EF. In DF algorithms, the data is considered to have directions, and the old data is exponentially forgotten only in the specific directions. How to implement the directional forgetting remains a question. The theory and basic algorithm for RLS with DF for traditional microprocessors has been described in the UTIA internal report (Kulhavý, 1983). (Kulhavý and Kárný, 1984) and (Kulhavý, 1987) propose a DF algorithm based on the Bayesian estimation approach.

The RLS with UDL factorization of the covariance matrix (UD-RLS) were used besides the QR factorization in several works (Kadlec, 1986), (Chisci and Mosca, 1987). In (Chisci and Mosca, 1987) the authors proposed two architectures for UD-RLS with DF based on systolic architectures with $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ processing elements. It is similar to RLS with LD factorization with some differences and therefore it is assumed that the structure of the systolic array for LD-RLS will be very close to the structure for UD-RLS.

This algorithm can prevent the estimator windup, but it may lose its tracking capability in some directions because eigenvalues of the information matrix may become unbounded in this algorithm as discussed in (Bittanti et al., 1990*a*).

The authors in (Parkum et al., 1992) tried to formulate a general forgetting algorithm from algorithms which were known in that time. The resulting algorithm is based on

the assumption that the parameter covariance matrix is bounded from below and from above, and it implies that the forgetting is non-uniform in space. So the authors proposed another method for forgetting - *selective forgetting* presented in (Parkum et al., 1990).In this method the covariance matrix is written as

$$\mathbf{C}(t) = \sum_{i=1}^{p} \alpha_i(t)\mathbf{v}_i(t)\mathbf{v}_i^T(t), \tag{2.1}$$

where $\alpha_1(t), ..., \alpha_p(t)$ are eigenvalues of $\mathbf{C}(t)$ and $v_1, ..., v_p$ are the corresponding eigenvectors. The time update of the covariance matrix in the selective forgetting is computed as

$$\mathbf{C}(t+1 \mid t) = \sum_{i=1}^{p} \frac{\alpha_i(t \mid t)}{\lambda_i(t)}\mathbf{v}_i(t)\mathbf{v}_i^T(t) \quad 0 < \lambda_i(t) < 1, \tag{2.2}$$

i.e. each eigenvalue of the covariance matrix is not divided by one common forgetting factor λ, but each eigenvalue is divided by its own forgetting factor λ_i chosen as a function of the amount of information received in the direction v_i.

In (Moonen, 1993) the author proposed a systolic array for computation of RLS with directional forgetting based on SVD factorization. It allows to reach $\mathcal{O}(1)$ throughput rate unlike $\mathcal{O}(n)$ reached in (Chisci and Mosca, 1987). Unfortunately, the data dependences in RLS with DF are such that efficient parallel architectures for DF LD-RLS are not easily obtained.

Apart from the directional forgetting, there are many attempts to avoid wind-up by modifications of exponential forgetting - from various methods of variable forgetting factor to, for example, preventing the information matrix from becoming singular with adding a positive definite matrix to ensure that it is always invertible (Gunnarsson, 1996).

In (Cao and H., 2000) the authors proposed a novel algorithm for directional forgetting based on matrix decomposition. In the paper the authors replaced a uniform exponential forgetting factor with forgetting matrix $F(t)$.

$$\mathbf{V}_\mathbf{z}(k) = \mathbf{F}(k)\mathbf{V}_\mathbf{z}(k-1) + \mathbf{z}(k)\mathbf{z}^T(k), \tag{2.3}$$

where the forgetting matrix is given by

$$\mathbf{F}(k) = \mathbf{I} - \mathbf{M}(k) = \mathbf{I} - \{(1-\lambda)\alpha(k)\mathbf{V}_\mathbf{z}(k-1)\mathbf{z}(k)\mathbf{z}^T(k)\} \tag{2.4}$$

and

$$\alpha(k) = \frac{1}{\mathbf{z}^T(k)\mathbf{V}_\mathbf{z}(k-1)\mathbf{z}(k)} \tag{2.5}$$

Another proposal how to avoid wind-up in EF RLS is in (Stenlund and Gustafsson, 2002). The authors mention some of the methods described above and compared them with the authors' proposal. They consider the filter as a control system where the goal is to achieve a pre-specified covariance matrix.

$$\mathbf{C}(t) = \mathbf{C}(t-1) - \frac{\mathbf{C}(t-1)\mathbf{z}(t)\mathbf{z}^T(t)\mathbf{C}(t-1)}{\mathbf{R}(t) + \mathbf{z}^T(t)\mathbf{C}(t-1)\mathbf{z}(t)} + \mathbf{Q}(t) \tag{2.6}$$

$$\mathbf{Q}(t) = \frac{\mathbf{P}_d\mathbf{z}(t)\mathbf{z}^T(t)\mathbf{P}_d}{\mathbf{R}(t) + \mathbf{z}^T(t)\mathbf{P}_d\mathbf{z}(t)}, \tag{2.7}$$

where $\mathbf{P}_d$ is the desired convergence point for the matrix $\mathbf{P}$. They denote this algorithm as adaptive Kalman filter algorithm.

The question is where a compromise between the tracking capabilities and the misadjustment and stability is.

Adaptive identifiers have the advantage that the system is continuously modeled and controller parameters are evaluated on-line, thus resulting in superior performance. They can be realized in several ways according to requirements. Recently such algorithms have been often used in embedded systems.

2.2 Embedded Systems

The term *embedded system* (ES) represents every computer system with a *processing unit* (PU) which is dedicated to control specific device or to serve specific purpose. An embedded system can be defined as a specialized computer system that is part of a larger system or machine designed to perform a specific function (Barr, 1998). Each PU contains one or more *processing elements* (PE) which actually process data in the system.

Embedded systems are everywhere around us. They are, for example, in multi-media devices (MP3s, televisions), household devices (refrigerators, microwave ovens, air-conditioners), electronic devices (printers, mobile telephones), traffic (information and control systems) and so on.

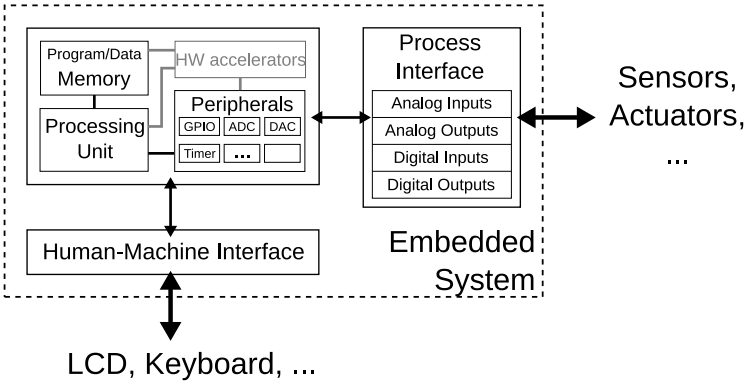


Figure 2.1: Generic embedded system

From the technical point of view the computer system in ES is mostly made as a *system-on-a-chip* (SoC), i.e. all parts in the upper left box in Figure 2.1 are implemented in one chip. Embedded systems for high production volumes are mostly designed as *application specific integrated circuits* (ASIC) which also often include other parts as shown in Figure 2.1. For smaller designs or lower production volumes, ESs are based on a universal *central processing unit* (CPU), *micro-controller unit* (MCU) or *field programmable gate array* (FPGA) which mostly contains the whole SoC.

Development in computer architectures is aimed at increasing performance and efficiency with decreasing power consumption. The parallelism is mainly used to reach these requirements. A lot of parallel architectures are proposed each year. There are several classifications used to compare such architectures. One of them is the Flynn's classification which categorizes systems into four major classes according to number of instruction and data streams in the system (Flynn, 1966) as described below.

- *Single-Instruction Single-Data Stream* (SISD) category contains all computers with one PE, i.e. sequential systems which cannot perform parallel operations.

- *Single-Instruction Multiple-Data Stream* (SIMD) category is very often mentioned in papers. Architectures in this group have more PEs with one common program (set of instructions) and each PE processes its own data stream. It implies that all PEs must process data equally and their program cannot contain branches. It is suitable mainly for DSP applications where a huge amount of data in independent channels is processed equally. Most vector and array architectures belong to this

group. Popularity of the SIMD concept is also due to its simple programming.

- *Multiple-Instruction Single-Data Stream* (MISD) category contains mainly architectures with pipelined PEs and PEs which process data stream simultaneously in independent parallel ways. A MISD system is suitable for applications aimed at image processing or classification (e.g. robot vision, neural networks).

- *Multiple-Instruction Multiple-Data Stream* (MIMD) is the most common and widely used form.

The thesis describes implementation on highly reconfigurable *UTIA DSP Platform* based on FPGAs which can contain more PEs with pipelined basic operations and so it can be directly used as a SISD or SIMD architecture or with more units as a MIMD or also as a MISD architecture.

2.2.1 FPGA Architecture

The used UTIA DSP Platform is a hardware accelerator for SoCs on FPGAs (Kadlec et al., 2007), (Daněk et al., 2008). FPGAs are reprogrammable chips with regular structure of configurable logic blocks and other hardwired blocks, such as memories, multipliers, circuits for clock distribution, DSP blocks, input/output blocks. These blocks are interconnected by programmable routing resources - switches and lines. And any custom function can be implemented in the hardware from these blocks. The chips are programmed by bitstreams which are a binary form of configuration of the blocks and interconnections prepared as a design in a hardware description language (HDL) (e.g. VHDL, Verilog); or in a high-level language (e.g. C, Handel-C) or a visual tool (e.g. Xiling System Generator).

FPGAs are truly parallel architectures where each independent function is assigned to a dedicated part of the chip and can work autonomously. It is a big advantage which allows to reach high performance of the implemented functions. Performance of many functions can be increased by parallel implementation of several instances of the required function.

FPGAs offer another advantage - arithmetic used in the implemented functions is not hardwired in the chip, but it can be selected and implemented as needed.

2.2.2 Arithmetics Used in FPGA

There are several alternatives which arithmetic could be used to implement algorithms on embedded systems. A selected arithmetic is dependent on considered or used hardware where the algorithm will be running. The most well known and used are floating- and fixed-point binary arithmetic and they are the ones supported by standard processors (common processors or digital signal processors).

Fixed-Point Arithmetic

The *fixed-point* FxP arithmetic is mainly used because of the high-speed which can be achieved with relatively simple arithmetic units (with small amount of used resources). On the other hand ,the dynamic data range is small and therefore it can be difficult or impossible to adapt some algorithms for fixed-point arithmetic.

Generally, every number in fixed-point arithmetic can be written as

$$r = R^{-B}\left[-b_{l-1}R^{l-1} + \sum_{i=0}^{l-2} b_i R^i\right], \quad b_{i\in 0,\ldots,l-1} \in 0,\ldots,R-1 \tag{2.8}$$

where R is a radix. For $R = 2$ it is a binary FxP arithmetic.

The usual fixed-point data formats in digital signal processing make use of the binary two's complement representation. Then, the value of a number is

$$r = 2^{-B}\left[-b_{l-1}2^{l-1} + \sum_{i=0}^{l-2} b_i 2^i\right], \quad b_i \in 0,1, \tag{2.9}$$

where l is the total word length, B is the location of the binary point, b_{l-1} is the sign of the number. There are two special cases, r is an integer when $B = 0$, and r is a fractional number when $B = l-1$. The second case is commonly used in DSP. In this case positive numbers are represented by $sign = 0$ and the value $|r|$ in all other bits whereas the negative numbers have $sign = 1$ and all other bits represent number $2 - |r|$. And therefore $-(-r) = r$ $(2 - (2 - |r|) = 2 - |r| - 2)$.

The main advantage of two binary two's complement representation compared to other representations is in the simplicity of hardware for adding and subtracting with no distinctions between the sign and binary digits (Hanselmann, 1987). Subtraction is implemented

by addition of the complemented number ($a-b = a+(-b)$). The sum of two l-bit numbers requires $l+1$ bits.

The FxP multiplication is more complicated. The operation can be written as

$$\begin{aligned} y_I &= a_I * b_I, \\ y_F &= a_F * b_F, \end{aligned} \tag{2.10}$$

where parts with I are the integer parts of the numbers and F are the fractional parts. The product of two l-bit numbers is a $(2l-1)$-bit number. This is because there is a sign-bit in each factor but the product needs only one.

The division is obviously the most complex among the basic operations. It has to solve

$$y = \frac{a}{b}, \tag{2.11}$$

which can be computed sequentially as

$$a = Q \cdot b + R, \quad 0 \leqq R < b$$

where Q is an integer quotient and R is an integer remainder. The result y is a combination of quotients obtained from a recursively divided remainder from a previous recursion.

Floating-Point Arithmetic

The *floating-point* (FP) arithmetic has usually a higher dynamic range and accuracy than fixed-point (if a standard word length of the FP arithmetic is used). The 32-bit single precision format (standard IEEE 754) consists of the mantissa's sign bit s, an 8-bit exponent e and 23 bits of mantissa for the fraction f.

Figure 2.2: Format of the FP numbers

The decimal value is given by

$$r = (-1)^s \cdot 2^{e-127} \cdot (1+f). \tag{2.12}$$

The IEEE standard also defines the following special values

IEEE-754 format	**sign** [bits]	**exponent** [bits]	**mantissa** [bits]
single (32bits)	1	8	23
double (64bits)	1	11	52

Table 2.1: Commonly used IEEE 754 formats of FP numbers

- **NaN** - Not a Number - result of an invalid operation such as $\frac{0}{0}$, $sqrt(-1)$ or $\infty \cdot 0$.
- $+\infty/-\infty$ - positive/negative infinity
- $\mathbf{-0}$ - negative zero

A fundamental difference from the fixed-point quantization is that there the error is an absolute one, i.e. the additional noise is independent of the signal, but with the floating-point arithmetic the error is a relative one - dependent on the signal amplitude. The dynamic range spans $2^{-126} \approx 10^{-38}$ up to $2^{+128} \approx 3 \cdot 10^{38}$ and the accuracy according to 2^{-23} as the value of the least significant bit in f.

Another difference between FP and FxP is their precision, which is worse for FP. An integer number can be represented exactly in the FP format only if its absolute value fits the mantissa, i.e. it is less than 2^{23} and 2^{53} for *single* and *double* formats respectively.

FP operations are more complicated than in the FxP arithmetic, they use data formats with more bits in most cases and FP numbers must be checked and corrected after operations with rounding and normalizing if necessary. For adding and subtracting the operands must have common fractional order and therefore their mantissas and exponents have to be corrected. The multiplication and division is simpler than addition, mantissas are processed similarly to the FxP arithmetics and then exponent must be added/subtracted.

Other arithmetic systems

There are two other arithmetic systems - logarithmic and residual, but they are not so usual in commonly used computers or embedded systems.

The *logarithmic number system* (LNS) is an alternative representation of floating point numbers (Matousek et al., 2002), (Tichý, 2006). It brings very simple multiplication and

division when the arguments are only added or subtracted, respectively. But addition and subtraction is more complicated, and they lead to evaluation of non-linear functions. Another advantage of LNS is an undemanding computation of power and root when a value of the exponent e is multiplied or divided by a constant.

The format of a logarithmic number is

$$r = (-1)^s 2^e, \tag{2.13}$$

where s is the sign bit and e is the fixed-point number.

The theoretical basis for the *residue number system* (RNS) has its ground roots in Fermat and Gauss (Svoboda, 1957). The RNS allows the decomposition of a given dynamic range (bit-width) in slices of smaller range on which the computation can be implemented in parallel at higher speed (Omondi and Premkumar, 2007), (Chokshi et al., 2009). The advantages of RNS are simplified and fast addition and multiplication.

In a residue number system (RNS) an integer r is represented as an ordered set of n residues $\{r_1, r_2, ..., r_n\}$, where $r_i = r \mod m_i$. The system is defined by a set of different prime numbers $\{m_1, m_2, ..., m_n\}$ called the moduli. Any integer in the range $\langle 0, M)$ can be uniquely represented where $M = \prod_{i=1}^{n} m_i$ is called the dynamic range of the moduli set.

2.2.3 Automatic Generation of Code

HW/SW co-design and partitioning of the problem between HW and SW are related to implementations on embedded systems and especially ESs based on DSPs or FPGAs. It always strongly depends on the target architecture, and therefore many specific projects and solutions are available for automatic partitioning and code generation.

For the *UTIA DSP Platform*, which is a specific vector architecture, there aren't any tools for automatic partitioning and code generation. Therefore a part of this work deals with the development of a generator of firmware code from algorithms in the Matlab environment.

Some works aimed to automatic code generation for ES are (Niemann, 1998), (Zhao and Malik, 1999), (Ramanujam et al., 2001), (Glesner et al., 2002), (Baleani et al., 2002), (Labrecque et al., 2007).

2.3 Summary

Historical background of recursive least square algorithms with directional forgetting and relation with some other RLS algorithms has been presented.

The need for higher sampling rates mainly in signal processing applications encourages development of algorithms for parallel architectures with higher throughput. In the work, such development is described for the UTIA DSP platform which is used as a vector architecture for implementing RLS based on the LDU decomposition with directional forgetting. The platform is built in an FPGA with floating-point arithmetic, therefore some of arithmetics employed in algorithms implemented in FPGAs have been also described.

Chapter 3

Adaptive Algorithms

This chapter contains background information about adaptive algorithms used in linear models of unknown systems, namely the *recursive least squares* (RLS) algorithm based on the LDU decomposition. An essential part of this chapter deals with forgetting methods used in adaptive algorithms. The chapter ends with a discussion of the correct comparison of different adaptive algorithms.

3.1 Introduction

Generally, an adaptive algorithm changes its behaviour or parameters based on its current state and inputs. Adaptive algorithms used in control and signal processing are mainly used for automatic adjustment of filter coefficients. A filter with an adaptive algorithm is called *adaptive filter*. Adaptive algorithms are classified according to the following features:

- the **rate of convergence** is speed of approaching the optimal solution,
- **tracking** is the ability to follow changes in time-varying systems,
- **numerical robustness** depends on specific implementations when numerical inaccuracies can cause instability,
- **computational complexity** is a complexity of the algorithm expressed as the number of operations or the spent time.

There are several basic types of applications of the adaptive filter:

System Identification

The purpose of the adaptive filter is to approximate parameters of an unknown system from system inputs and outputs. If the unknown system is dynamical with time-varying parameters, the used adaptive algorithm must be able to track such changes.

In this case both the system and the filter have common inputs, and the filter is adjusted by the difference between their outputs. Then the filter represents a model of the unknown system.

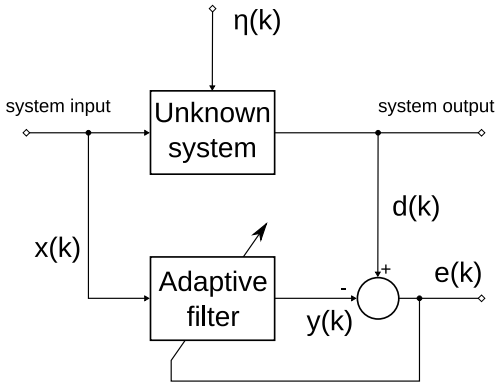


Figure 3.1: Adaptive Filter for System Identification

Noise Cancellation and Echo Cancellation

The input signal for the adaptive filter is a noise which is correlated with a noise in the desired signal. The filter can eliminate the noise from the desired signal if the desired signal contains an uncorrelated noise. One of possible applications of noise cancellation is echo cancellation when the unknown system represents the echo path, and the filter eliminates the echo from the required signal.

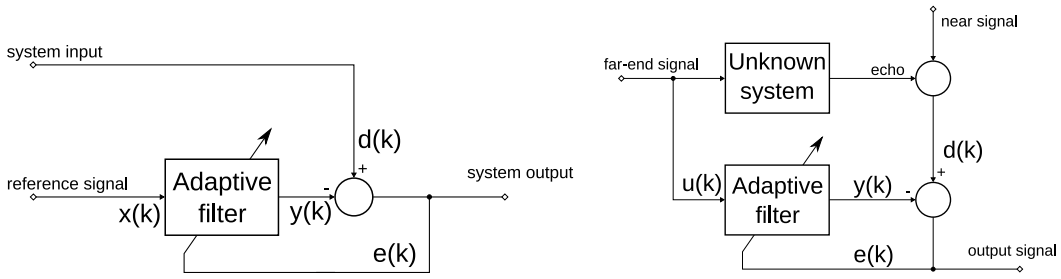


Figure 3.2: Adaptive Filter in task of the Noise Cancellation and the Echo Cancellation

Channel Equalization

The adaptive filter provides an inverse model of an unknown system. One of typical applications is mobile communication when the unknown system is the transmission channel, and the filter eliminates channel distortion.

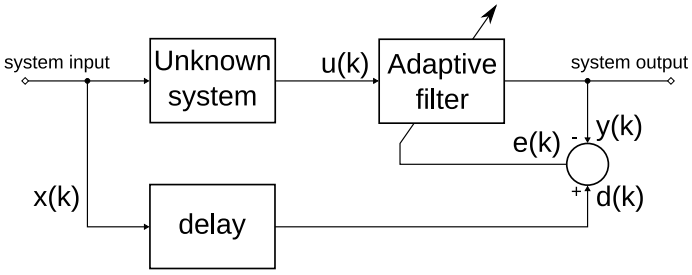


Figure 3.3: Adaptive Filter for Channel Equalization

Many adaptive algorithms have been devised for both linear and non-linear filtering. Each of them has some advantages and some disadvantages. Among the classic algorithms the *least mean square* LMS and *recursive least squares* RLS are the most frequently used for linear filtering. Many modifications of these algorithms have been developed.

3.1.1 Models of the Unknown System

The selection of an adaptive algorithm depends on the used model of the unknown system. Figure 3.4 shows notations used for the following models of systems. In the model, $\boldsymbol{\Theta}$ is a vector of parameters of the unknown system, it can contain parameters a, b, c according to the selected model; $\hat{\boldsymbol{\Theta}}$ is a vector of parameters of the adaptive filter; y is an output of

the system; and $\hat{y}$ is an output of the filter. Outputs of the system and the filter are given by the functions

$$y(k) = f(\boldsymbol{\Theta}(k), \mathbf{u}(k), \mathbf{y}(k-1), k) \tag{3.1}$$

and

$$\hat{y}(k) = f(\hat{\boldsymbol{\Theta}}(k), \mathbf{u}(k), \hat{\mathbf{y}}(k-1), k) \tag{3.2}$$

respectively.

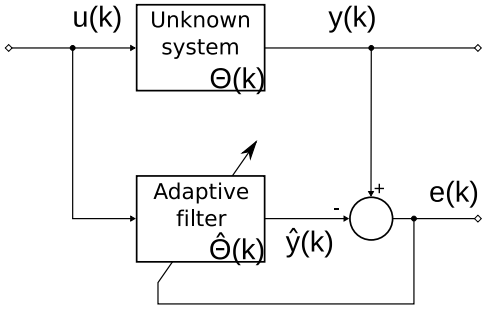


Figure 3.4: A model for System Identification

The RLS algorithm can be used with the following linear models:

- *Moving average* (MA) model. In this model the system output depends only on the current and past system inputs

$$y(k) = \sum_{i=0}^{nb} b_i u(k-i), \tag{3.3}$$

  where $y(k)$ is the system output at time k, $u(k)$ is the system input at time k and b_i is the i-th parameter of the system. The MA model is essentially a *finite impulse response* (FIR) filter where nb is its order.

- *Autoregressive* (AR) model. This model contains a dependence on past outputs

$$y(k) = -\sum_{i=1}^{na} a_i y(k-i). \tag{3.4}$$

  It is a *infinite impulse response* (IIR) filter where na is its order.

- *Autoregressive moving average* (ARMA) model is the combination of the two models mentioned above

$$y(k) = -\sum_{i=1}^{na} a_i y(k-i) + \sum_{i=0}^{nb} b_i u(k-i). \tag{3.5}$$

- *Autoregressive moving average model with exogenous inputs* (ARMAX) is an ARMA model with an extra input mainly for measurable noise.

$$y(k) = -\sum_{i=1}^{na} a_i y(k-i) + \sum_{i=0}^{nb} b_i u(k-i) + \sum_{i=0}^{nc} c_i \eta(k-i). \tag{3.6}$$

In the following text we will suppose that a system is modeled by ARMA with its output defined as a function of its inputs, past outputs and unmeasurable white error. The output is then

$$y(k) = \mathbf{\Theta}^T(k)\mathbf{z}(k) + e_s(k), \tag{3.7}$$

where $\Theta(k)$ is a vector of model parameters

$$\mathbf{\Theta}^T(k) = [a_1, a_2, ..., a_{na}, b_0, b_1, ..., b_{nb}], \tag{3.8}$$

and $\mathbf{z}(k)$ is a *data regressor*, i.e. a vector of the current input data and past input and output data

$$\mathbf{z}^T(k) = [-y(k-1), -y(k-2), ..., -y(k-na), u(k), u(k-1), ..., u(k-nb)]. \tag{3.9}$$

The adaptive algorithms tries to minimize the error between the output of an unknown system y and the output of the filter $\hat{y}$ as shown in Figure 3.4,

$$e(k) = y(k) - \hat{y}(k). \tag{3.10}$$

There are more ways to minimize the error, two of them are described in the following text.

3.2 Adaptive Algorithms

3.2.1 Least Mean Square

The basic *least mean square* (LMS) algorithm uses the *mean square error* (MSE) as the optimization cost function

$$J_{MSE}(\mathbf{\Theta})) = E(\mathbf{e}^2) = E(|\mathbf{y} - \hat{\mathbf{y}}|^2), \tag{3.11}$$

The LMS algorithm is very popular among other adaptive algorithms for its properties. The key property is its linear computational complexity $\mathcal{O}(n)$. The main disadvantage of the LMS algorithm is its slow convergence.

3.2.2 Least Squares

The technique of the *least squares* (LS) was proposed by Karl Gauss around 1795 to predict the motion of planets and comets using telescopic measurements. The LS and its many variants have been extensively applied to solving estimation problems in many application fields.

The cost function of LS is the time-averaged squared error

$$J_{LS}(\mathbf{\Theta}) = \sum_{i=1}^{k} e^2(i) = \sum_{i=1}^{k} [y(i) - \hat{y}(i)]^2 = \sum_{i=1}^{k} \left[y(i) - \hat{\mathbf{\Theta}}^T(i)\mathbf{z}(i)\right]^2. \tag{3.12}$$

The cost function can be written in the matrix form (Bobál et al., 1999)

$$J_{LS}(\mathbf{\Theta}) = \mathbf{e}^T\mathbf{e} = (\mathbf{y} - \mathbf{U\Theta})^T(\mathbf{y} - \mathbf{U\Theta}), \tag{3.13}$$

where y is a vector of system outputs

$$\mathbf{y}^T = [y(1), y(2), y(3), ..., y(k)], \tag{3.14}$$

e is a vector of errors

$$\mathbf{e}^T = [e(1), e(2), e(3), ..., e(k)], \tag{3.15}$$

and $\mathbf{U}$ is a matrix of data regressors $\mathbf{z}(k)$

$$\mathbf{U} = \begin{bmatrix} \mathbf{z}^T(0) \\ \mathbf{z}^T(1) \\ \mathbf{z}^T(2) \\ \vdots \\ \mathbf{z}^T(k-2) \\ \mathbf{z}^T(k-1) \end{bmatrix}. \tag{3.16}$$

The estimated parameters to minimize the cost function (3.13) can be evaluated from

$$\left.\frac{\partial J_{LS}}{\partial \mathbf{\Theta}}\right|_{\mathbf{\Theta}=\hat{\mathbf{\Theta}}} = 0. \tag{3.17}$$

The solution of Equation (3.17) is then

$$\hat{\mathbf{\Theta}} = (\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T\mathbf{y} = \mathbf{C}.\mathbf{c}_{zy}, \tag{3.18}$$

where $\mathbf{C} = \mathbf{U}^T\mathbf{U}$ is the covariance matrix of inputs, and $\mathbf{c}_{zy} = \mathbf{U}^T\mathbf{y}$ is the cross-covariance vector.

3.2.3 Recursive Least Squares

The LS algorithm is used for system identification. When we have already measured all data, we can compute the estimation in one batch. If we consider that we need the estimation during measurement when we have only past data, the *recursive least squares* (RLS) algorithm can be used.

It computes estimations by the LS method from the past data at time k. The main idea of RLS is to compute a solution at time k from the results of the previous computation at time $(k-1)$. This approach to the problem increases the efficiency of the LS algorithm, and it allows "on-line" identification of systems with time-varying parameters.

Although the RLS algorithm is theoretically equivalent to the block LS algorithm, it suffers from the following shortcomings:

- numerical instability due to round-off errors caused by its recursive operations in a finite-word length environment,

- slow tracking capability for time-varying parameters,
- big sensitivity to the initial conditions of the algorithm.

These shortcomings can be suppressed by extra forgetting methods as will be described in the next section.

The derivation of the RLS algorithm has been described in many papers. It is for example in (Peterka, 1982). In the following text we provide a short summary of its derivation. The cost function of the LS algorithm (3.12) for the ARMA model at time k is

$$J_{LS}(\mathbf{\Theta}(k)) = \left[\begin{array}{cc} 1 & -\mathbf{\Theta}^T \end{array} \right] \sum_{i=1}^{k} \mathbf{d}(i)\mathbf{d}^T(i) \left[\begin{array}{c} 1 \\ -\mathbf{\Theta} \end{array} \right] = \left[\begin{array}{cc} 1 & -\mathbf{\Theta}^T \end{array} \right] \mathbf{V}(k) \left[\begin{array}{c} 1 \\ -\mathbf{\Theta} \end{array} \right], \quad (3.19)$$

where we denote a new vector composed of y and z as an extended data regressor d

$$\mathbf{d}(k) = \left[\begin{array}{c} \mathbf{y}(k) \\ \mathbf{z}(k) \end{array} \right]. \quad (3.20)$$

We denote the sum of data dyads as an *extended information matrix* V. The extended information matrix can be evaluated recursively

$$\mathbf{V}(k) = \sum_{i=1}^{k} \mathbf{d}(i)\mathbf{d}^T(i) = \mathbf{V}(k-1) + \mathbf{d}(k)\mathbf{d}^T(k). \quad (3.21)$$

It consists of block matrices

$$\mathbf{V}(k) = \left[\begin{array}{cc} \mathbf{V}_y(k) & \mathbf{V}_{zy}^T(k) \\ \mathbf{V}_{zy}(k) & \mathbf{V}_z(k) \end{array} \right], \quad (3.22)$$

where the information matrix $\mathbf{V}_\mathbf{z}(k) = \mathbf{C}^{-1}(k)$ is the inverse of the covariance matrix $\mathbf{C}(k)$. From Equations 3.21 and 3.22 one can see that the information matrix is updated as

$$\mathbf{V}_\mathbf{z}(k) = \mathbf{V}_\mathbf{z}(k-1) + \mathbf{z}(k)\mathbf{z}^T(k). \quad (3.23)$$

The direct update of the covariance matrix can be obtained by applying the *matrix inversion lemma* (Golub and Van Loan, 1996) to Equation 3.23.

$$\mathbf{C}(k) = \mathbf{C}(k-1) - \frac{\mathbf{C}(k-1)\mathbf{z}(k)\mathbf{z}^T(k)\mathbf{C}(k-1)}{1 + \mathbf{z}^T(k)\mathbf{C}(k-1)\mathbf{z}(k)} \quad (3.24)$$

The update of the estimated parameters is

$$\hat{\mathbf{\Theta}}(k) = \hat{\mathbf{\Theta}}(k-1) + \frac{\mathbf{C}(k-1)\mathbf{z}(k)}{1 + \mathbf{z}^T(k)\mathbf{C}(k-1)\mathbf{z}(k)}\hat{e}(k), \tag{3.25}$$

where $\hat{e}(k)$ is the priori estimation error defined as

$$\hat{e}(k) = y(k) - \hat{\mathbf{\Theta}}^T(k-1)\mathbf{z}(k), \tag{3.26}$$

and $e(k)$ is the posteriori estimation error

$$e(k) = y(k) - \mathbf{\Theta}^T(k)\mathbf{z}(k). \tag{3.27}$$

3.3 Forgetting Used with RLS

The conventional RLS algorithm is suitable for systems with constant parameters and without a measurable noise. Such an RLS does not provide enough adaptivity to the estimator. Several methods were devised to improve the RLS algorithm for systems with a noise or time-varying parameters and to avoid numerical instability of the RLS algorithm. The most widespread and essentially single method is called *weighting* or *forgetting*. The common idea of all forgetting methods is based on an assumption that a recent information is more valuable than older data.

There are many papers where forgetting algorithms or their modifications are suggested. All of them try to improve the stability and robustness of adaptive algorithms with forgetting. The main forgetting method is *exponential forgetting* (EF). Most papers that describe it are from between 1980 and 2000, when many classical forgetting methods were devised, and many improvements of the exponential forgetting have been published. In that time many papers were published about other groups of forgetting algorithms which are based on a concept of *restricted exponential forgetting* known also as *directional forgetting* (DF). In these algorithms, the forgetting factor is set according to information in the input data.

In this section we will describe basic forgetting methods used in adaptive algorithms based on RLS.

3.3.1 Exponential Forgetting

Exponential forgetting (EF) is the basic forgetting method used as the minimal extension of conventional RLS. The principle of EF is to reduce impacts of older data on the cost function in RLS by multiplicating the last extended information matrix $\mathbf{V}(k-1)$ by a *forgetting factor* (FF) $\lambda \in (0,1)$ in each step. It brings exponential reduction of the impact in time.

The evolution of the extended information matrix with exponential forgetting is

$$\mathbf{V}(k) = \lambda \mathbf{V}(k-1) + \mathbf{d}(k)\mathbf{d}^T(k). \tag{3.28}$$

Of course, the evolution of the information matrix $\mathbf{V}_{\mathbf{z}}(k-1)$ is the same, and it affects the covariance matrix $\mathbf{C}(k-1)$. The evolution of the covariance matrix is then

$$\mathbf{C}(k) = \frac{1}{\lambda(k)}\left(\mathbf{C}(k-1) - \frac{\mathbf{C}(k-1)\mathbf{z}(k)\mathbf{z}^T(k)\mathbf{C}(k-1)}{\lambda(k) + \mathbf{z}^T(k)\mathbf{C}(k-1)\mathbf{z}(k)}\right). \tag{3.29}$$

The evolution of the parameter estimation is the same as for pure RLS

$$\hat{\mathbf{\Theta}}(k) = \hat{\mathbf{\Theta}}(k-1) + \frac{\mathbf{C}(k-1)\mathbf{z}(k-1)}{1 + \mathbf{z}^T(k-1)\mathbf{C}(k-1)\mathbf{z}(k-1)}\hat{e}(k), \tag{3.30}$$

In (Peterka, 1981), (Kulhavý, 1983), (Kadlec, 1986) the update step is divided into two steps - the data update and the time update, in that case the denotation of the time indices is as follows

$a(k\|k-1)$	a in the previous time step after the time update
$a(k\|k)$	a in the current time step after the data update and before the time update
$a(k+1\|k)$	a in the current time step after the time update

The data update of $\mathbf{V}, \mathbf{C}$ and $\hat{\mathbf{\Theta}}$ is

$$\begin{aligned}
\mathbf{V}(k|k) &= \mathbf{V}(k|k-1) + \mathbf{d}(k)\mathbf{d}^T(k) \\
\mathbf{C}(k|k) &= \mathbf{C}(k|k-1) - \frac{\mathbf{C}(k|k-1)\mathbf{z}(k)\mathbf{z}^T(k)\mathbf{C}(k|k-1)}{1 + \mathbf{z}^T(k)\mathbf{C}(k|k-1)\mathbf{z}(k)} \\
\hat{\mathbf{\Theta}}(k|k) &= \hat{\mathbf{\Theta}}(k|k-1) + \frac{\mathbf{C}(k|k-1)\mathbf{z}(k)}{1 + \mathbf{z}^T(k)\mathbf{C}(k|k-1)\mathbf{z}(k)}\hat{e}(k),
\end{aligned} \tag{3.31}$$

and the time update of $\mathbf{V}, \mathbf{C}$ and $\hat{\mathbf{\Theta}}$ is

$$\begin{aligned}
\mathbf{V}(k+1|k) &= \lambda \mathbf{V}(k|k) \\
\mathbf{C}(k+1|k) &= \frac{1}{\lambda}\mathbf{C}(k|k) \\
\hat{\mathbf{\Theta}}(k+1|k) &= \hat{\mathbf{\Theta}}(k|k).
\end{aligned} \tag{3.32}$$

For the forgetting factor $\lambda = 1$ it is an RLS algorithm without exponential weighting (without forgetting). In this case all data in history have the same influence on the cost function. This means that such an RLS algorithm can produce a correct estimation of parameters only for the problem of system identification with static parameters and without noise. If there is noise or time-varying parameters in the model, then we can obviously suppose that new data describe the model better than older data. The simplest way to model this behaviour is to use exponential forgetting with the forgetting factor $\lambda(k) \in (0, 1)$. The forgetting factor controls the rate of forgetting, i.e. newer data have smaller impact on the information matrix with a lower value of $\lambda(k)$. But with lower $\lambda(k)$ the estimator has a worse ability to predict the evolution of the system. Finding the optimal[1] value of the forgetting factor $\lambda(k)$ is crucial for the correct behaviour of the system. A commonly used interval of the forgetting factor values is $\lambda(k) \in < 0.95, 0.999 >$.

In some approaches *variable exponential forgetting* is used to increase the quality of EF. An example is in (Navrátil and Bobál, 2005). It doesn't use one static value of $\lambda(k) = \lambda_0$, but the value evolves in time, for example as

$$\lambda(k) = \lambda_0 \lambda(k-1) + 1 - \lambda_0, \tag{3.33}$$

where is $\lambda(0) = \lambda_0 \in < 0.95, 0.99 >$. The identification forgets more at the beginning and then the forgetting is slower (see Figure 3.5) with this function of the forgetting factor. The function asymptotically goes to one, which means the latest data isn't weighted for long running filtering/identification. λ_0 controls the speed of forgetting deceleration at the beginning of filtering. It tries to compensate the wrong initial behaviour of the algorithm when the estimator starts up from unknown state.

The standard exponential forgetting has proved to be a simple method of discarding obsolete information. It suppresses all accumulated information regardless of the character of the measured data, and it does not admit the incorporation of available information about parameter variations other than through the choice of the forgetting factor. Therefore more complicated models of forgetting have been proposed.

Figure 3.6 shows an example of the evolution of estimated parameters and the covariance matrix and their relation in the parameter space. The step of the data update with

[1] It means optimal for the required purpose.

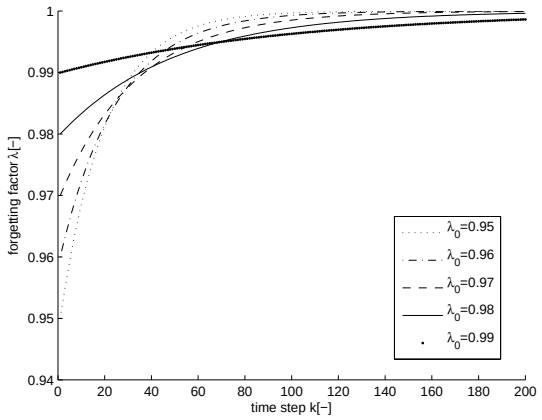


Figure 3.5: Evolution of a variable exponential forgetting factor

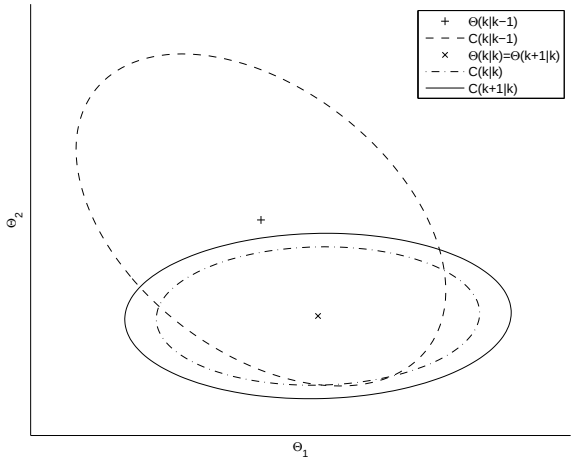


Figure 3.6: An example of evolution of the ellipse of concentration for exponential forgetting

newly incoming data shifts time indices from $(k|k-1) \equiv (k-1)$ to indices $(k|k)$. It shifts the estimated parameters to new positions and reduces the covariance matrix (and thus the area of the ellipse of concentration). The time update, which shifts the time indices from $(k|k)$ to $(k+1|k) \equiv (k)$), contains forgetting which extends the area of the ellipse.

3.3.2 Data-Dependent Forgetting

The aim of these methods is to eliminate the main problem with the evolution of the covariance matrix (and, of course, the evolution of the ellipse of concentration) in exponential forgetting, when the information is discarded without respecting the distribution of an information in the incoming data (demonstrated in Figure 3.6). This uniform discarding can cause wind-up of the covariance matrix (flattening of the ellipse of concentration). There are several different methods which implement some of the data-dependent forgetting as mentioned in Chapter 2, only directional forgetting is shown in the following text.

Directional Forgetting

It has been devised as one possible way to avoid the wind-up in exponential forgetting. The described version is from (Kulhavý, 1983), and this forgetting method is used in the implementation in the following chapter. The information matrix evolves according to the incoming information

$$\begin{aligned} &\|\zeta(k-1)\| > 0 \\ &\qquad \mathbf{V}_{\mathbf{z}}(k) = \mathbf{V}_{\mathbf{z}}(k-1) + \left(\lambda(k) - \frac{1-\lambda(k)}{\zeta(k-1)}\right)\mathbf{z}(k)\mathbf{z}^T(k), \\ &\|\zeta(k-1)\| = 0 \\ &\qquad \mathbf{V}_{\mathbf{z}}(k) = \mathbf{V}_{\mathbf{z}}(k-1). \end{aligned} \tag{3.34}$$

The evolution of the covariance matrix is then

$$\begin{aligned} &\|\zeta(k-1)\| > 0 \\ &\qquad \mathbf{C}(k) = \mathbf{C}(k-1) - \frac{\mathbf{C}(k-1)\mathbf{z}(k)\mathbf{z}^T(k)\mathbf{C}(k-1)}{\varepsilon^{-1}(k-1) + \zeta(k-1)}, \\ &\|\zeta(k-1)\| = 0 \\ &\qquad \mathbf{C}(k) = \mathbf{C}(k-1), \end{aligned} \tag{3.35}$$

and the evolution of the parameter estimation is

$$\begin{aligned} &\|\zeta(k-1)\| > 0 \\ &\qquad \hat{\mathbf{\Theta}}(k) = \hat{\mathbf{\Theta}}(k-1) + \frac{\mathbf{C}(k-1)\mathbf{z}(k)}{1+\zeta(k-1)}\hat{e}(k), \\ &\|\zeta(k-1)\| = 0 \\ &\qquad \hat{\mathbf{\Theta}}(k) = \hat{\mathbf{\Theta}}(k-1), \end{aligned} \tag{3.36}$$

where

$$\begin{aligned} \zeta(k-1) &= \mathbf{z}^T(k)\mathbf{C}(k-1)\mathbf{z}(k), \\ \varepsilon(k-1) &= \lambda(k) - \frac{1-\lambda(k)}{\zeta(k-1)}. \end{aligned} \tag{3.37}$$

The update can also be written in two steps as the data update and the time update. The data update of $\mathbf{V}, \mathbf{C}$ and $\hat{\mathbf{\Theta}}$ is

$$\begin{aligned} \mathbf{V}_{\mathbf{z}}(k|k) &= \mathbf{V}_{\mathbf{z}}(k|k-1) + \mathbf{z}(k)\mathbf{z}^T(k) \\ \mathbf{C}(k|k) &= \mathbf{C}(k|k-1) - \frac{\mathbf{C}(k|k-1)\mathbf{z}(k)\mathbf{z}^T(k)\mathbf{C}(k|k-1)}{1+\zeta(k|k-1)} \\ \hat{\mathbf{\Theta}}(k|k) &= \hat{\mathbf{\Theta}}(k|k-1) + \frac{\mathbf{C}(k|k-1)\mathbf{z}(k)}{1+\zeta(k|k-1)}\hat{e}(k), \\ \zeta(k|k-1) &= \mathbf{z}^T(k)\mathbf{C}(k|k-1)\mathbf{z}(k), \end{aligned} \tag{3.38}$$

and the time update of $\mathbf{V}, \mathbf{C}$ and $\hat{\mathbf{\Theta}}$ is

$$\begin{aligned} &\hat{\mathbf{\Theta}}(k+1|k) = \hat{\mathbf{\Theta}}(k|k) \\ &\|\zeta(k|k-1)\| > 0 \\ &\qquad \mathbf{V}_{\mathbf{z}}(k+1|k) = \mathbf{V}_{\mathbf{z}}(k|k) - \frac{1-\lambda(k)}{\zeta(k|k-1)}\mathbf{z}(k)\mathbf{z}^T(k) \\ &\qquad \mathbf{C}(k+1|k) = \mathbf{C}(k|k) + \frac{1-\lambda(k)}{\lambda(k)}\frac{\mathbf{C}(k|k)\mathbf{z}(k)\mathbf{z}^T(k)\mathbf{C}(k|k)}{\mathbf{z}^T(k)\mathbf{C}(k|k)\mathbf{z}(k)}, \\ &\|\zeta(k|k-1)\| = 0 \\ &\qquad \mathbf{V}(k+1|k) = \lambda(k)\mathbf{V}(k|k) \\ &\qquad \mathbf{C}(k+1|k) = \mathbf{C}(k|k) \end{aligned} \tag{3.39}$$

Figure 3.7 illustrates evolution of the estimated parameters and the covariance matrix depicted by the ellipse of concentration. The data update contains the same operations as it was described for the exponential forgetting. But in the time update the ellipse is blown up only in the direction of the new incoming information, which is parallel with the line between the old and new positions of the estimated parameters.

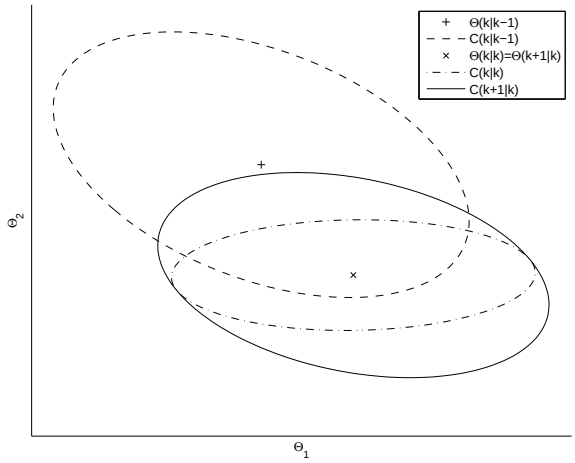


Figure 3.7: An example of evolution of the ellipse of concentration for directional forgetting

Directional Adaptive Forgetting

This method is a case of directional forgetting with the maximal restriction based on the assumption that the time updated density of unknown parameters is not purely Gaussian (Kulhavý, 1987). Then the value of the forgetting factor evolves according to the following equations

$$\begin{aligned}
\varepsilon(k-1) &= \varphi(k) - \frac{1-\varphi(k)}{\zeta(k-1)}, \\
\varphi(k) &= \left\{1 + (1+\rho)\left[ln\left(1+\zeta(k-1)\right)\right] + \left[\frac{\left(\nu(k-1)+1\right)\eta(k-1)}{1+\zeta(k-1)+\eta(k-1)} - 1\right]\frac{\zeta(k-1)}{1+\zeta(k-1)}\right\}^{-1}, \\
\eta(k) &= \frac{\hat{e}^2(k)}{\lambda(k)}, \\
\nu(k) &= \varphi(k)(\nu(k-1)+1), \\
\lambda(k) &= \varphi(k)\left[\lambda(k-1) + \frac{\hat{e}^2(k-1)}{1+\zeta(k-1)}\right],
\end{aligned} \tag{3.40}$$

where $\varphi(k)$ is a variable forgetting factor, and ρ is a heuristic parameter.

This section presented information about forgetting methods used in the RLS algorithm to improve its properties, such as stability and robustness.

3.4 LD-RLS Algorithms

There are many ways how the RLS algorithm can be implemented. In cases when the extended information matrix $\mathbf{V}_k$ is numerically ill-conditioned (nearly singular), we must use a computation which numerically ensures positive semi-definiteness of $\mathbf{V}(k)$ for all k. Otherwise the entire identification can numerically collapse. This problem has been solved with radical algorithms which use suitable decompositions of the inversion of the matrix $\mathbf{V}(k)$. Algorithms with QR, Cholesky, LDU and UDL decompositions are examples of such algorithms.

In this section we summarize the derivation of the LD-RLS algorithm[2] based on the LDU decomposition. The derivation is described for example in (Peterka, 1982). The computation of roots is not necessary in this method. Another advantage of the LD-RLS algorithm is that the estimated parameters $\hat{\mathbf{\Theta}}$ are directly included in the solution of the decomposition.

In the following derivation we omit the time indices for clarity. The inversion of the extended information matrix $\mathbf{V}$ can be decomposed to a lower triangular matrix $\mathbf{L}$ with ones on the diagonal, and a diagonal matrix $\mathbf{D}$ as follows

$$\mathbf{V}^{-1} = \mathbf{L}\mathbf{D}\mathbf{L}^T \tag{3.41}$$

Matrices $\mathbf{D}$ and $\mathbf{L}$ can be decomposed into blocks

$$\mathbf{D} = \begin{bmatrix} D_y & 0 \\ 0 & \mathbf{D}_z \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 1 & 0 \\ \mathbf{L}_{zy} & \mathbf{L}_z \end{bmatrix} \tag{3.42}$$

The cost function (3.19) is then

$$J_{RLS}(\mathbf{\Theta}) = \begin{bmatrix} 1 & -\mathbf{\Theta}^T \end{bmatrix} \mathbf{V} \begin{bmatrix} 1 \\ -\mathbf{\Theta} \end{bmatrix} = \begin{bmatrix} 1 & -\mathbf{\Theta}^T \end{bmatrix} (\mathbf{L}^{-1})^T \mathbf{D}^{-1} \mathbf{L}^{-1} \begin{bmatrix} 1 \\ -\mathbf{\Theta} \end{bmatrix}. \tag{3.43}$$

[2]The LD-RLS algorithm with exponential forgetting is called *LDFIL* in (Peterka, 1982), and the LD-RLS with directional forgetting is called *LDDIC* in (Kulhavý, 1983).

The inversion of the triangular matrix $\mathbf{L}$ and the diagonal matrix $\mathbf{D}$ (3.42) is

$$\begin{aligned}\mathbf{L}^{-1} &= \begin{bmatrix} 1 & 0 \\ -\mathbf{L}_z^{-1}\mathbf{L}_{zy} & \mathbf{L}_z^{-1} \end{bmatrix}, \\ \mathbf{D}^{-1} &= \begin{bmatrix} D_y^{-1} & 0 \\ 0 & \mathbf{D}_z^{-1} \end{bmatrix},\end{aligned} \tag{3.44}$$

then the cost function (3.43) is

$$J_{RLS}(\mathbf{\Theta}) = \begin{bmatrix} 1 & -\mathbf{\Theta}^T \end{bmatrix} \begin{bmatrix} 1 & -\mathbf{L}_{zy}^T(\mathbf{L}_z^{-1})^T \\ 0 & (\mathbf{L}_z^{-1})^T \end{bmatrix} \begin{bmatrix} D_y^{-1} & 0 \\ 0 & \mathbf{D}_z^{-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\mathbf{L}_z^{-1}\mathbf{L}_{zy} & \mathbf{L}_z^{-1} \end{bmatrix} \begin{bmatrix} 1 \\ -\mathbf{\Theta} \end{bmatrix} =$$

$$= D_y^{-1} + (-\mathbf{L}_{zy} - \mathbf{\Theta})^T(\mathbf{L}_z^{-1})^T\mathbf{D}_z^{-1}\mathbf{L}_z^{-1}(-\mathbf{L}_{zy} - \mathbf{\Theta}) \tag{3.45}$$

The second part of the cost function (3.45) depends only on parameters $\mathbf{\Theta}$. Therefore the absolute minimum for $\hat{\mathbf{\Theta}}$ is

$$\hat{\mathbf{\Theta}} = -\mathbf{L}_{zy}, \tag{3.46}$$

and the minimum of the cost function (3.45) is for $\mathbf{\Theta} = \hat{\mathbf{\Theta}}$ and its value is

$$J_{RLS}(\hat{\mathbf{\Theta}}) = \min_{\mathbf{\Theta}}(J_{RLS}(\mathbf{\Theta})) = D_y^{-1}. \tag{3.47}$$

By comparing Equations (3.42) with (3.46) and (3.47) we can see that the solution is directly contained in matrices $\mathbf{L}, \mathbf{D}$. The matrix $\mathbf{L}$ contains negative values of the estimated parameters, and the matrix $\mathbf{D}$ contains an inverted value of the cost function for the estimation of parameters

$$\mathbf{L} = \begin{bmatrix} 1 & 0 \\ -\hat{\mathbf{\Theta}} & \mathbf{L}_z \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} (J_{RLS}(\hat{\mathbf{\Theta}}))^{-1} & 0 \\ 0 & \mathbf{D}_z \end{bmatrix}. \tag{3.48}$$

The LD-RLS algorithm also has an advantage in the possibility to use directly a priori information about the system when the RLS algorithm is initialized. The matrix $\mathbf{L}$ can contain an initial estimation of parameters $\mathbf{\Theta}(0)$, and matrix $\mathbf{D}$ contains values which

represent the uncertainty of the initial estimation of parameters $\mathbf{\Theta}(0)$

$$\begin{aligned}\mathbf{L}(0) &= \begin{bmatrix} 1 & 0 & 0 \\ & 1 & 0 \\ -\hat{\mathbf{\Theta}}(0) & \ddots & \\ & 0 & 1 \end{bmatrix}, \\ \mathbf{D}_{(0)} &= \begin{bmatrix} D_y(0) & \\ & \mathbf{D}_z(0) \end{bmatrix}, D_{z(i,i)}(0) > 0.\end{aligned} \tag{3.49}$$

3.4.1 Direct Update of LD-RLS

The LD-RLS algorithm and other similar algorithms can be updated directly. In this part we present equations for updating the EF LD-RLS and DF LD-RLS algorithms which are used in implementations in the next chapter. The derivation is described in Appendix A.

EF LD-RLS

$$\begin{aligned} & L_{i,j}(k) = L_{i,j}(k-1) - \frac{f_j(k) g_i^{(j+1)}(k)}{\lambda + h_{j+1}(k)} \\ & D_i(k) = D_i(k-1) \frac{\lambda + h_{i+1}(k)}{\lambda(\lambda + h_i(k))} \\ \text{where} & \\ & \mathbf{f}(k) = L(k-1)\mathbf{d}(k) \\ & \mathbf{g}_i(k) = D_i(k-1) f_i(k) \\ & h_i(k) = \sum_{l=i}^{n} f_l(k) g_l(k), \quad h_{n+1}(k) = 0; \\ & g_i^{(m)}(k) = g_i^{m+1}(k) + D_m(k-1) L_{i,m}(k-1) f_m(k) \end{aligned} \tag{3.50}$$

DF LD-RLS

If input data are sufficiently excited ($h_2(k) > 0$)

$$
\begin{aligned}
L_{i,j}(k) &= L_{i,j}(k-1) - \frac{f_j(k) g_i^{(j+1)}(k)}{\alpha_j(k) + h_{j+1}(k)} \\
\\
\bar{D}_i(k) &= D_i(k-1) \frac{\alpha_i(k) + h_{i+1}(k)}{\alpha_i(k) + h_i(k)} \\
D_1(k) &= \frac{\bar{D}_1(k)}{\lambda}, \quad D_i(k) = \bar{D}_i(k) \quad \forall i \in (2..n)
\end{aligned}
\tag{3.51}
$$

where

$$
\begin{aligned}
&\mathbf{f}(k) = L(k-1)\mathbf{d}(k) \\
&\mathbf{g}_i(k) = D_i(k-1) f_i(k) \\
&h_i(k) = \sum_{l=i}^{n} f_l(k) g_l(k), \quad h_{n+1}(k) = 0; \\
&g_i^{(m)}(k) = g_i^{m+1}(k) + D_m(k-1) L_{i,m}(k-1) f_m(k) \\
&\alpha_1(k) = 1, \quad \alpha_i(k) = \psi(k) \quad \forall i \in (2..n) \\
&\psi(k) = \frac{h_2(k)}{h_2(k)(\lambda+1) - 1}
\end{aligned}
$$

If input data aren't sufficiently excited ($h_2(k) \to 0$)

$$
\begin{aligned}
D_1(k) &= \frac{D_1(k-1)}{\lambda(1 + h_1(k) - h_2(k))} \\
D_i(k) &= D_i(k-1) \quad \forall i \in (2..n)
\end{aligned}
\tag{3.52}
$$

Summary

In this part the basic derivation of the LD-RLS algorithm together with the direct update of EF LD-RLS and DF LD-RLS were presented to provide background information for the next chapter where the implementation of these algorithms is described.

3.5 Comparison of the Adaptive Algorithms

When a new algorithm is devised or implemented, it is necessary to compare the new algorithm with other existing algorithms. The comparison of an adaptive algorithm can

be done in several typical applications such as system identification, channel equalization (inverse modeling) or noise cancellation. Of course, the algorithms should be tested in a kind of application for which they were designed.

The comparison between the adaptive algorithms is based on using them in the same testing application with the same system and input signals. Then one or more parameters of the quality of the adaptation are compared. These observed parameters depend on the purpose of adaptation.

Characteristics often used for comparison are the *mean squared error* (MSE) and *i*n adaptive signal processing, and the *system error norm* (SEN) in adaptive control. The first characteristic MSE statistically reflects differences between output signal from the identified system and output signal from the adaptive filter. It can be calculated as

$$MSE = \frac{1}{n}\sum_{i=1}^{n} e_i^2, \tag{3.53}$$

which is the average of the square of the estimation error. It is often expressed in decibels (dB) in the literature as

$$MSE_{dB} = 10\log_{10}(MSE) \quad [dB]. \tag{3.54}$$

The second statistic, SEN reflects the squared norm of the difference between the true parameters of an unknown system and their estimated values. It can be expressed in decibels as

$$SEN_{dB} = 10\log_{10}(\|\hat{\Theta} - \Theta\|^2) \quad [dB]. \tag{3.55}$$

Most of the adaptive algorithms use some forgetting technique. Therefore all these algorithms have at least one heuristic parameter which is adjusted by hand. Typically it is forgetting factor in the LMS and RLS algorithms with exponential forgetting. In many papers where different algorithms for forgetting were compared, the authors used examples with one concrete value of the forgetting factor for all the compared algorithms.

And this poses questions such as "How was the forgetting factor chosen?" or "Why did they use just that value?" and the most important question "Can two algorithms be compared with the same value of the forgetting factor?". The last question means that the forgetting factor can have a different meaning or impact on the convergence

and behaviour of each algorithm. And, of course, some algorithms have more than one tunable parameters, e.g. in the maximally restricted forgetting in (Kulhavý, 1987) where the second heuristic parameter is used.

In practice, the forgetting factor is chosen from one's own experience and from the interval where the identification algorithm can guarantee robustness. But when algorithms are proposed and simulated, the algorithms should be compared under similar conditions. For example, when the forgetting factor ensures optimality of identification in some ways. We expect that such conditions are ensured for different settings (e.g. the value of the forgetting factor) of each algorithm. We guess that some algorithms can have much worse behaviour than other algorithms with the same value of their forgetting factor. And there may be a case when an algorithm is better than another with equal factors, and it is worse with "optimally" chosen factors.

In the following experiments we will try to answer the mentioned questions. These experiments are based on some of the RLS algorithms mentioned above. The SEN statistic was selected as the cost function to find the forgetting factor in which the algorithms are comparable. Table 3.1 shows all RLS algorithms with different forgetting methods which were used in the experiments.

Algorithm	**Implemented by**
Conventional RLS	(Diniz, 2007)
EF LD-RLS	(Peterka, 1982)
RLS with variable FF	(Navrátil and Bobál, 2005)
DF LD-RLS	(Kulhavý, 1983)
DDF RLS	(Cao and H., 2000)

Table 3.1: The RLS algorithms used in the experiment

All statistics used in the RLS algorithms depend on the input data (i.e. on the structure of the unknown system and its input data). Therefore we used the mean value of the SEN statistic as a cost function to find the value of the forgetting factor for each algorithm. We selected the mean value of the SEN statistic as an example for its simplicity.

$$\lambda_{comp} = \lambda \quad \text{if} \quad SEN(\lambda_{comp}) = \min_{\lambda} E(SEN(\lambda)), \quad \lambda \in (0,1) \tag{3.56}$$

We known that this selection is not really usable in real applications because such a cost function optimizes the forgetting factor for minimizing the difference between the estimated and true parameters. The more useful cost function should be more complex to respect the application requirements such as the convergence rate and tracking ability together.

To minimize the selected cost function we used an iterative algorithm of interval halving on $\lambda \in (0.1, 1)$ because we assumed the SEN is a continuous function of the forgetting factor with one minimum in all tested algorithms. An example of such a function is in Figure 3.8. The example also shows that changing the forgetting factor doesn't have to have significant influence on the cost function in some cases.

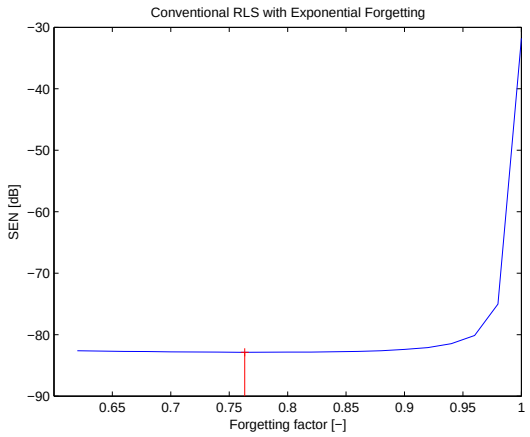


Figure 3.8: An example of a dependence of the mean value of the SEN statistic on the forgetting factor for an identification problem

Because the mean value of the SEN depends on the input data and their length (mainly in an identification problem) we prepared several groups of experiments. All algorithms were tested on problems of system identification, system identification with insufficient excitation of inputs, slow tracking of parameters and fast tracking of parameters. In the experiments we computed the average of the SEN from 2000 time samples.

We tried our concept on the following problems

- Identification of system with sufficiently excited inputs (Figure 3.9),

- Identification of system with insufficiently excited inputs (Figure 3.10),
- Tracking slow time-varying parameters with insufficiently excited inputs (Figure 3.11),
- Tracking fast time-varying parameters with insufficiently excited inputs (Figure 3.12),

The result of each experiment contains a set of charts

- the $MSE(k)$ with the forgetting factors evaluated to minimize the average SEN for each algorithm,
- the $SEN(k)$ with the forgetting factors evaluated to minimize the average SEN for each algorithm,
- the $MSE(k)$ with the same value of the forgetting factor $\lambda = 0.98$ for all algorithms,
- the $SEN(k)$ with the same value of the forgetting factor $\lambda = 0.98$ for all algorithms,
- the $MSE(k)$ with the same value of the forgetting factor $\lambda = 0.8$ for all algorithms,
- the $SEN(k)$ with the same value of the forgetting factor $\lambda = 0.8$ for all algorithms.

The experiments have been selected as simple and regular cases of using adaptive algorithms in such systems. We didn't try any extreme cases because these experiments should only demonstrate the proposed method. We used the "optimal" computed forgetting factors, the value $\lambda = 0.98$ used in (Parkum et al., 1992), and the very low value $\lambda = 0.8$ which is out of range of the commonly used interval of the forgetting factor. Charts in Figures 3.9, 3.10, 3.11 and 3.12 show that forgetting methods, which use the forgetting factor differently in the computation, have a slightly different behaviour than others for the same value of the forgetting factor. From the charts we see that for "optimal" values of the forgetting factors the SEN statistics are the best; it is because we use the mean value of SEN as a cost function to optimize the forgetting factor. For real applications a more complex cost function must be used.

Summary

This part discusses the common practice described in many papers concerning the comparison of adaptive algorithms with different forgetting methods which need not necessary

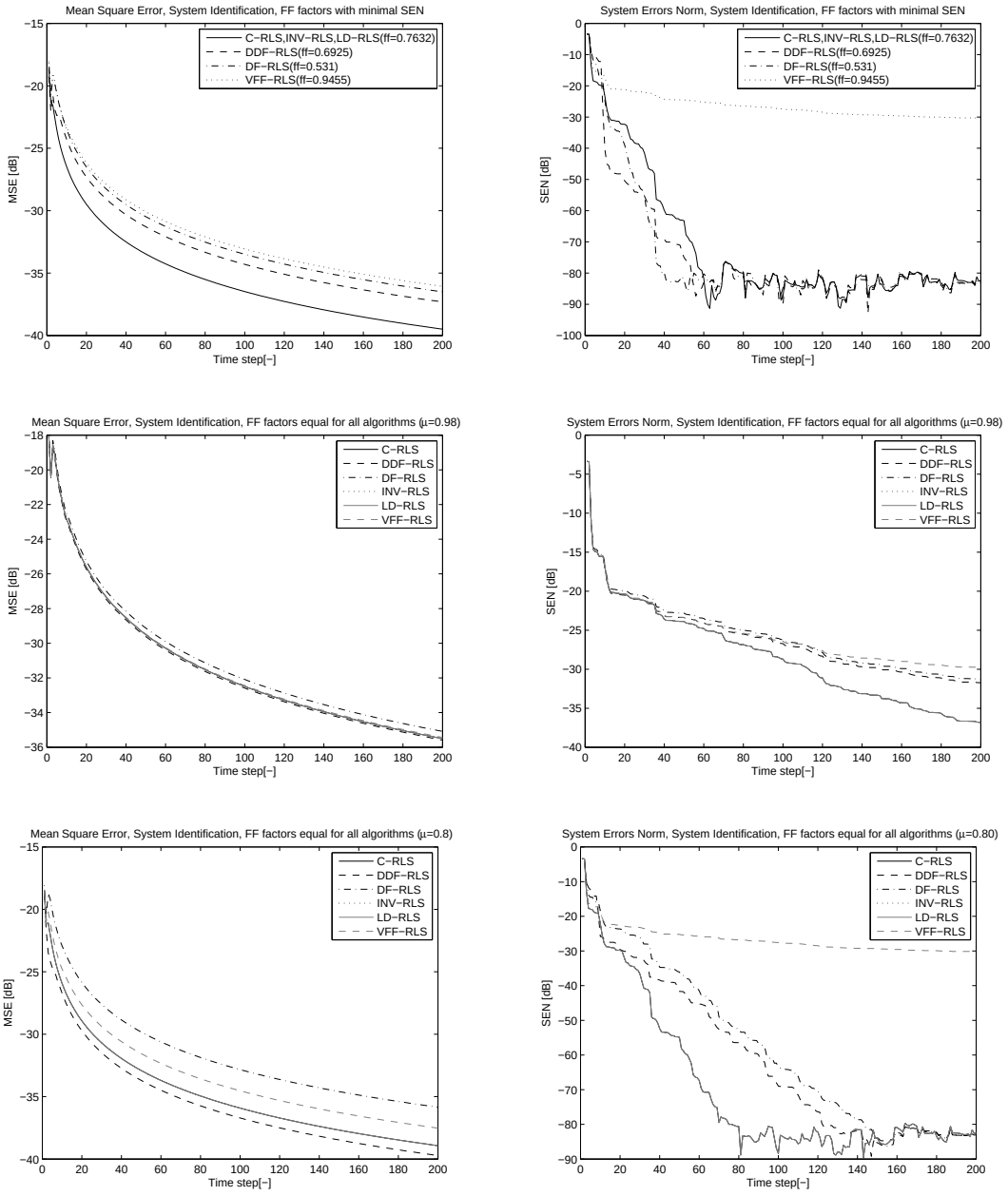


Figure 3.9: The comparison of different RLS algorithms for system identification. The first couple of graphs is for forgetting factors with minimal $E(SEN)$ in each algorithm, the second couple is for $\lambda = 0.98$ and the third couple is for $\lambda = 0.8$

be correct. Many authors compare their new adaptive algorithm or modification with others by using a uniform value of the forgetting factors. They don't analyze which value of the forgetting factor should be used.

We proposed a simple method to ensure that algorithms with different forgetting methods will be certainly comparable. It is based on searching the "optimal" value of the forgetting factor (or "optimal" values of all factors used in the forgetting algorithm) separately for each algorithm. Then the algorithms are compared with their own "optimal" forgetting factor instead of one common value of the forgetting factor. The "optimal" value can be searched from various statistics used to test the qualities of the adaptive algorithms. The statistic should be chosen according to the purpose of the concrete adaptive algorithm. Examples of such statistics are *mean squared error* (MSE) or *system error norm* (SEN). The value of both the statistics SEN and MSE always depends on the identified system together with the input data. Therefore the "optimal" values of forgetting factors depend on specific experiments, and they must always be evaluated.

To test our hypothesis we performed several experiments with the RLS algorithms and five different forgetting methods. We used them to identify an unknown system and parameter tracking with four sets of parameters and input vectors. We didn't try any extreme cases because these experiments were demonstrated the proposed method. Of course we know that the average of the SEN is not quite appropriate for finding the "optimal" value of forgetting factor, but it should only demonstrate the concept of comparability between algorithms with different forgetting methods. According to the performed experiments we can say that it is more appropriate to use forgetting factors evaluated from common optimization task for each algorithm independently than to use one selected forgetting factor for all algorithms when comparing adaptive algorithms with different forgetting methods. The difference between "optimally" selected factor for each algorithm and any common value of factor can be significant and it depends on concrete experiment used for comparison.

3.6 Summary

This chapter deals with adaptive algorithms with forgetting methods. First general information about adaptive algorithms used in linear models of unknown systems is discussed. Then there the *recursive least squares* (RLS) algorithm is briefly described. The next part summarizes the LDU decomposition used in LD-RLS algorithms.

The main part of the chapter is aimed at the description of forgetting methods used in the RLS algorithms. We described the classical exponential forgetting and not so commonly used directional forgetting.

The chapter ends with a discussion of comparability of adaptive algorithms with different forgetting methods. Authors of most papers where they propose new methods or modifications of a forgetting method use any value of the forgetting factor to compare their method with other existing methods. Because they don't discuss where or how they obtained the value they used, we proposed a method for comparing adaptive algorithms with different forgetting methods.

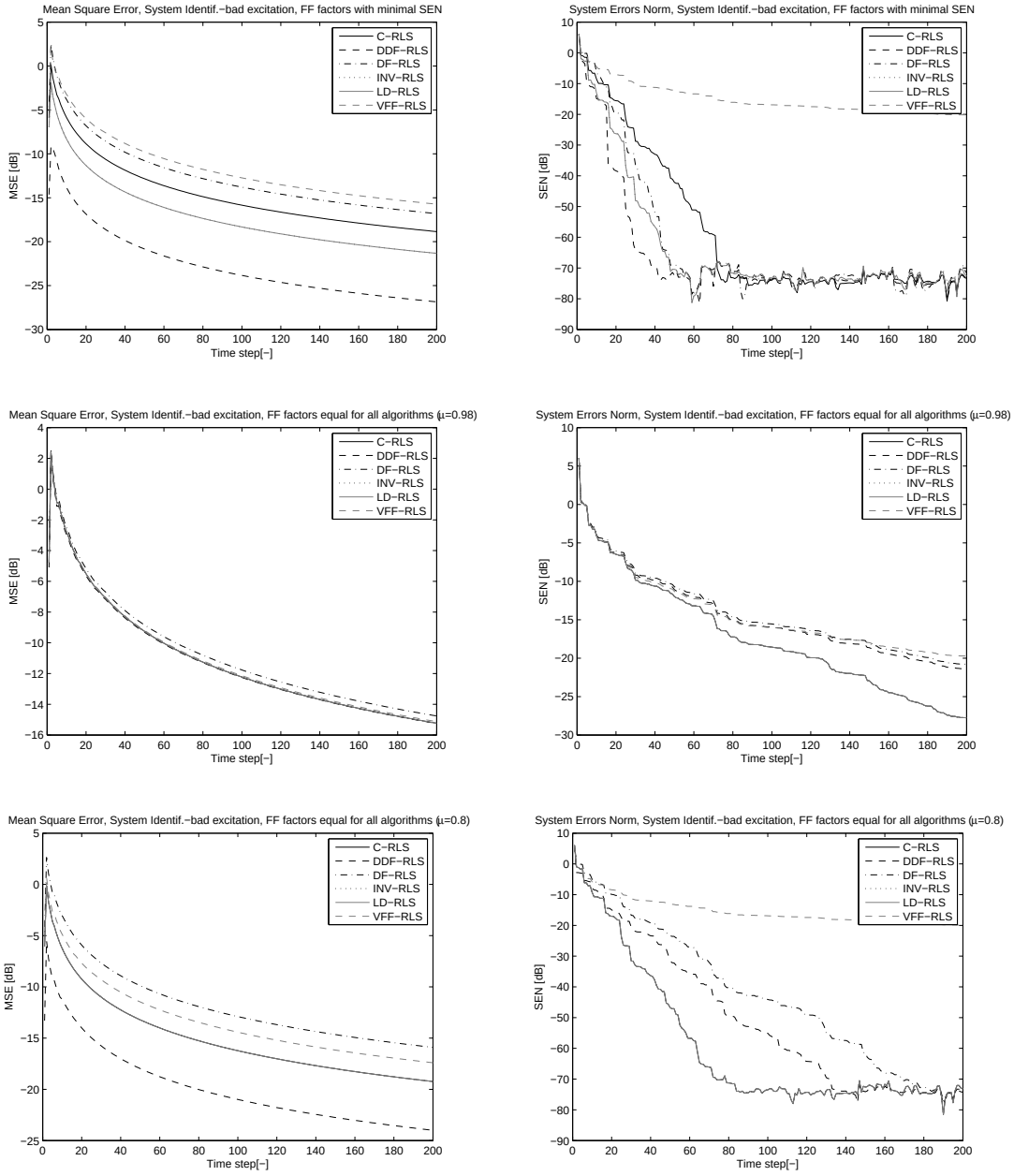


Figure 3.10: The comparison of different RLS algorithms for system with insufficient excitation of inputs. The first couple of graphs is for forgetting factors with minimal $E(SEN)$ in each algorithm, the second couple is for $\lambda = 0.98$ and the third couple is for $\lambda = 0.8$

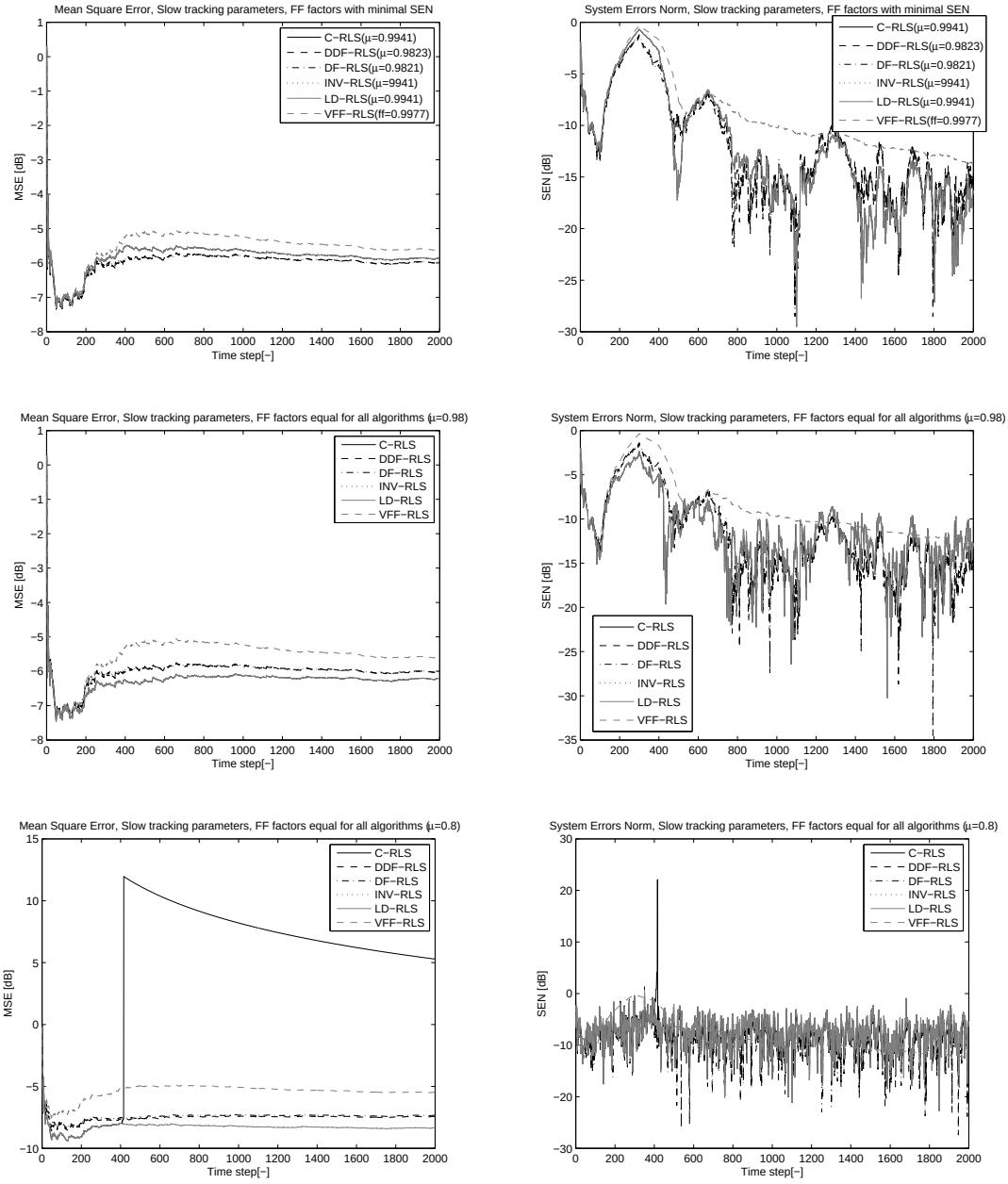


Figure 3.11: The comparison of different RLS algorithms for system with slow time-varying parameters. The first couple of graphs is for forgetting factors with minimal $E(SEN)$ in each algorithm, the second couple is for $\lambda = 0.98$ and the third couple is for $\lambda = 0.8$

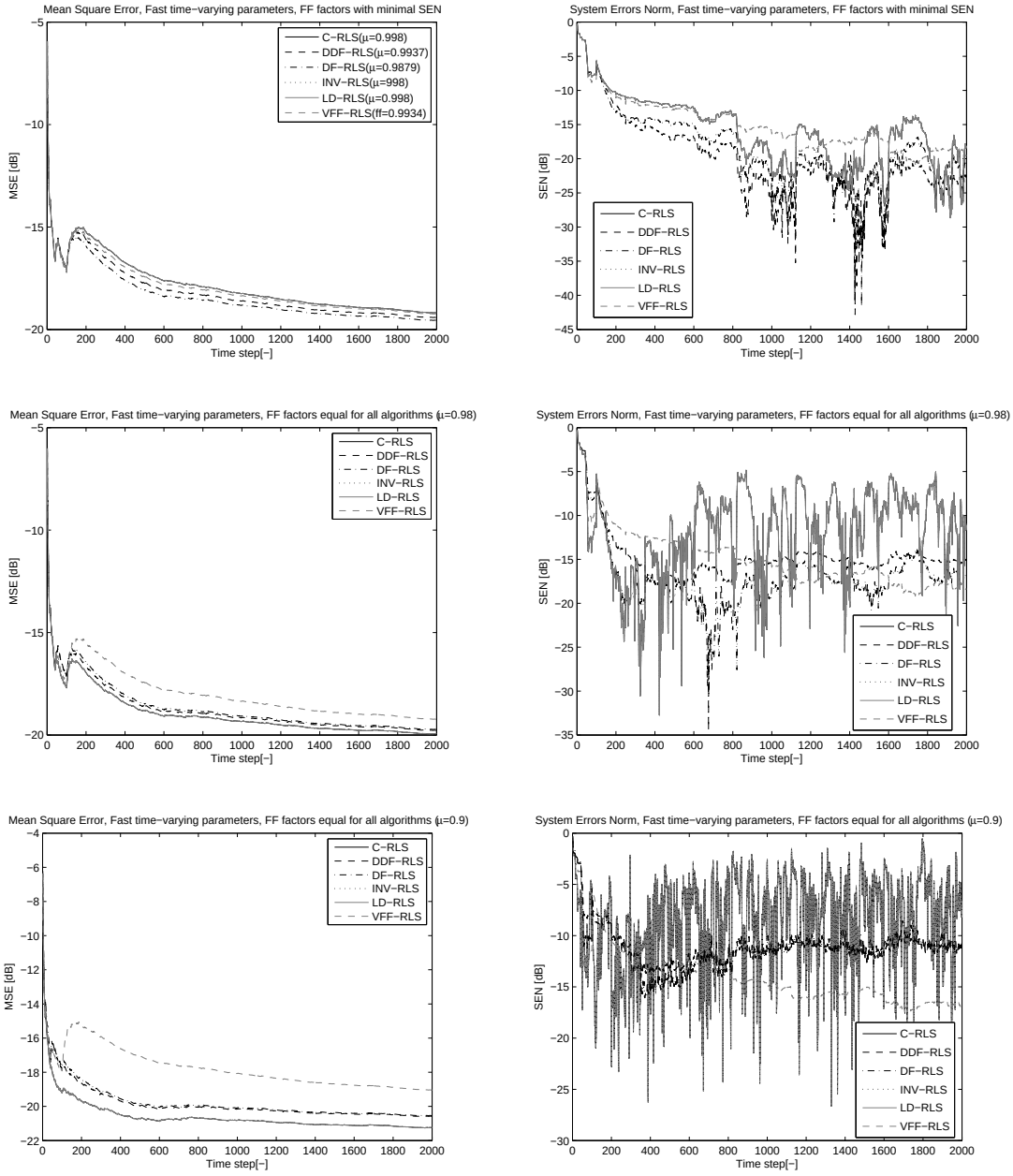


Figure 3.12: The comparison of different RLS algorithms for system with fast time-varying parameters. The first couple of graphs is for forgetting factors with minimal $E(SEN)$ in each algorithm, the second couple is for $\lambda = 0.98$ and the third couple is for $\lambda = 0.8$

Chapter 4

Implementation

This chapter deals with implementations of LD-RLS and especially the DF LD-RLS algorithm on embedded systems. First we will briefly discuss the implementation aspects on embedded systems. Because the term *embedded system* is obviously too wide, then the work will focus on two architectures with potential to be used in embedded systems, and in parallel processing (being aware of the fact that DF LD-RLS has limited possibilities for parallelisation).

The first of the two architectures is the well known systolic array. This architecture is researched here to confirm the hypothesis that the final structure of DF LD-RLS will be similar to the structure for the DF UD-RLS algorithm described in (Chisci and Mosca, 1987). The second reason to use this architecture is that the development in computer architectures is aimed at many-core computing (Silvano et al, 2010).

The second architecture is the *UTIA DSP Platform*[1] which has been developed recently. It is a flexible, re-programmable and reconfigurable hardware accelerator. The platform with pipelined floating-point operations such as addition, multiplication, division, multiply-accumulate and dot product is used and extended in this work.

In all of these implementations we will consider the worst case, when an algorithm is used to estimate parameters in one-step prediction or filtering, i.e. matrices L and D must be

[1]The UTIA DSP platform has been developed for DSP applications by J.Kadlec at the Department of Signal Processing, ÚTIA AV ČR, v.v.i, therefore its called UTIA DSP platform.

updated before the next data comes. It means the implementations must expect that they cannot be fully parallelized or pipelined due to data dependence in the DF LD-RLS algorithm as shown in the previous chapters.

The chapter ends with a discussion of a possible building systolic arrays with UTIA DSP platforms as processing elements.

4.1 Introduction

An architecture of the *processing unit* (PU) where the algorithm will be implemented is one of the crucial criteria of an implementation. This criterion is important mainly in embedded systems where the number of possible architectures is huge. The number of computing elements in one processing unit and their structure is one of the main properties along with efficiency of the implementation and power consumption of the PU. Therefore the development of future architectures for embedded systems is directed to multi-core and many-core processors (Silvano et al, 2010), i.e. to processing units with many processing elements. There are many structures of the processing elements in the parallel units, e.g. PE in a systolic array, SIMD, MIMD, stream processing, distributed processing.

Of course, the implementation depends on the structure of the algorithm, and it depends on the structure of the process and its model whose parameters are estimated. One of the characterizations of processes is based on the number of system inputs and outputs. Algorithms with the directional forgetting can be directly used only for systems with single output (SISO or MISO), but parameters of systems with multiple outputs (SIMO or MIMO) can be estimated separately in parallel. Then a set of estimators with directional forgetting can be used for each output of a MIMO system. Identification should use a correct model which will contain all relations between inputs and outputs to avoid erroneous estimations.

There are always more ways to implement an algorithm on embedded system, therefore the selected way depends on the required optimization criteria. In the past when algorithms were implemented on systems with a single microprocessor and small external memory, the main criterion was memory consumption and data organization in the memory because of the price of a memory was too high. Today's implementations are mainly

oriented to minimize the time of computation or efficiency and power consumption. The efficiency in parallel multi-core processing units is often expressed in terms of utilization of all processing elements. The utilization can be influenced by rescheduling of operations in the algorithm or swapping parts of the algorithm between processing elements.

The following criteria of algorithm implementation should be observed:

- utilization of processing elements, i.e. the ratio between the time when a PE is working and whole computation time.

$$U_{PE(i)} = \frac{Twork_{PE(i)}}{Tcomp}$$

- overall utilization, i.e. the combination of utilization over all PEs

$$U_{all} = \frac{\sum_{i=1}^{n} U_{PE(i)}}{n.Tcomp}$$

- numerical robustness,
- data throughput,
- speed of computation (input/output data sampling rate),
- latency for pipelined implementations
- computational complexity,
- amount of used resources,
- power consumption,

The preferred criterion depends on the purpose of the embedded system. For real-time systems the crucial criterion is the time of computation and the related data throughput and the computational complexity. For mobile systems it is mainly power consumption and the related amount of used resources and their utilization.

Implementations of the DF LD-RLS described in this chapter is essentially aimed at numerical robustness; improvement of some of the mentioned criteria will be only a side effect.

4.2 Implementation on a Systolic Array

In this section a possible implementation of the DF LD-RLS algorithm on a systolic array will be discussed. First we will briefly describe a generic systolic array. Implementation-specific details of the algorithm will be presented subsequently.

4.2.1 Systolic arrays

A *systolic array* (SA) is a network of PEs working in parallel with a regular structure and interconnections ((Kung, 1982)). Input and output data are processed by outer elements, and all data travel through the neighbouring elements. Each PE can work only with its local data or data from its neighbours. Systolic arrays with broadcast signals are also possible. In that case PEs can provide data to other PEs which aren't their neighbours. Such connections are dedicated to share only one signal, i.e. it is a point-to-point connection all the time. This structure has a drawback - it doesn't contains only short connections between PEs - which makes parameters of implementation worse for systolic arrays on FPGAs.

For maximal regularity the systolic array should have as few different PE flavours as possible, ideally it should contain PEs of one type only. Systolic arrays have an advantage over other architectures, they can process some complex algorithms with the computational complexity $\mathcal{O}(n^2)$, with the time complexity $\mathcal{O}(n)$ and in some cases even $\mathcal{O}(1)$. Decreasing the time complexity is always balanced with increasing the space complexity, i.e. the number of PE increases from $\mathcal{O}(n)$ up to $\mathcal{O}(n^2)$.

The concept of systolic arrays is convenient for implementations of an algorithm that work with vectors and matrices. Systolic arrays are suitable for data pipelining, but real use of this technique depends on the algorithm and its data dependences.

An example of a pipelined systolic array is shown in Figure 4.1, where the matrix multiplication is depicted. In this case each PE performs the same *multiply-accumulate* (MAC) function and also passes the input data to its neighbours. The input matrices are pushed through the SA in synchronous steps so that each PE multiplies the corresponding items from the input matrices.

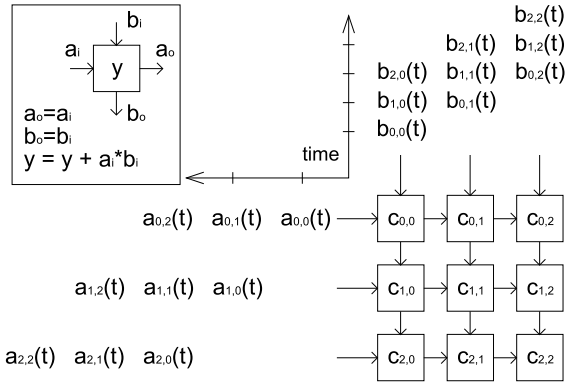


Figure 4.1: Matrix multiplication on systolic array

The big advantage of systolic arrays is their relatively simple practical use, when the implementation has only several types of elements and therefore the complexity of their programming is not growing when increasing the size of the array.

4.2.2 The EF LD-RLS Algorithm

At first the EF LD-RLS has been implemented in the form of a systolic array as a testing case. As shown in the previous chapter the EF LD-RLS has part of the algorithm identical to DF LD-RLS, so we can expect the structure of this part to be the same or at least similar.

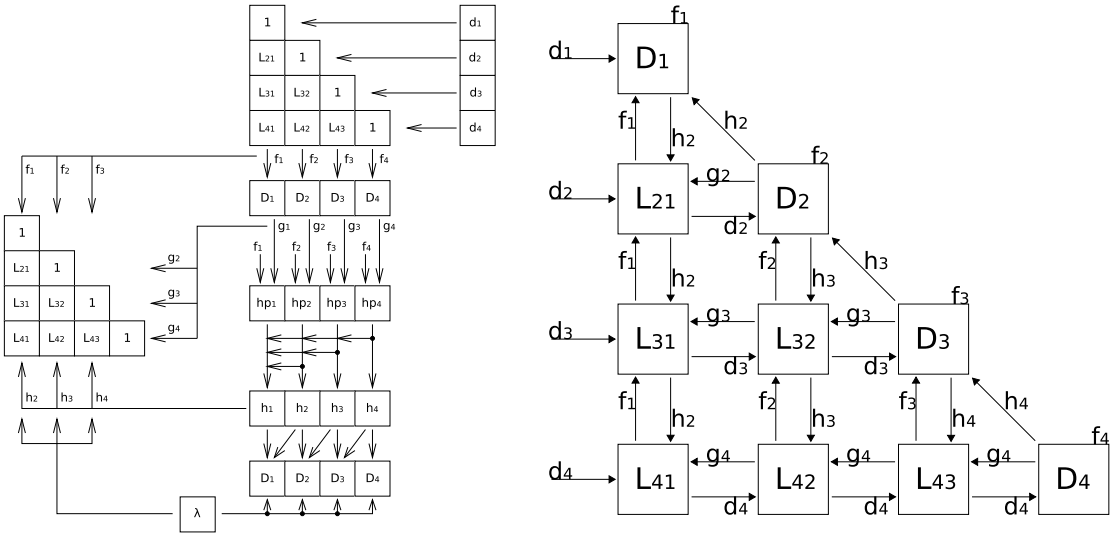


Figure 4.2: EF LD-RLS data dependence graph and PE dependence graph (for n=3)

Figure 4.2 depicts the data dependences in the algorithm and the dependence graph of PEs. The graphs are derived from Equations 3.50 in the Chapter 3. f is computed as a multiply-accumulate function of input vector d and matrix L. f is immediately used for computation of g with D. Vectors f and g together constitute hp which is an input to cumulative summation of vector h. The coefficients gl necessary to update matrix L are computed from the vector g and matrix L. The next coefficients are additions of items of vector h and forgetting factor λ. They are required to update values in both matrices L and D.

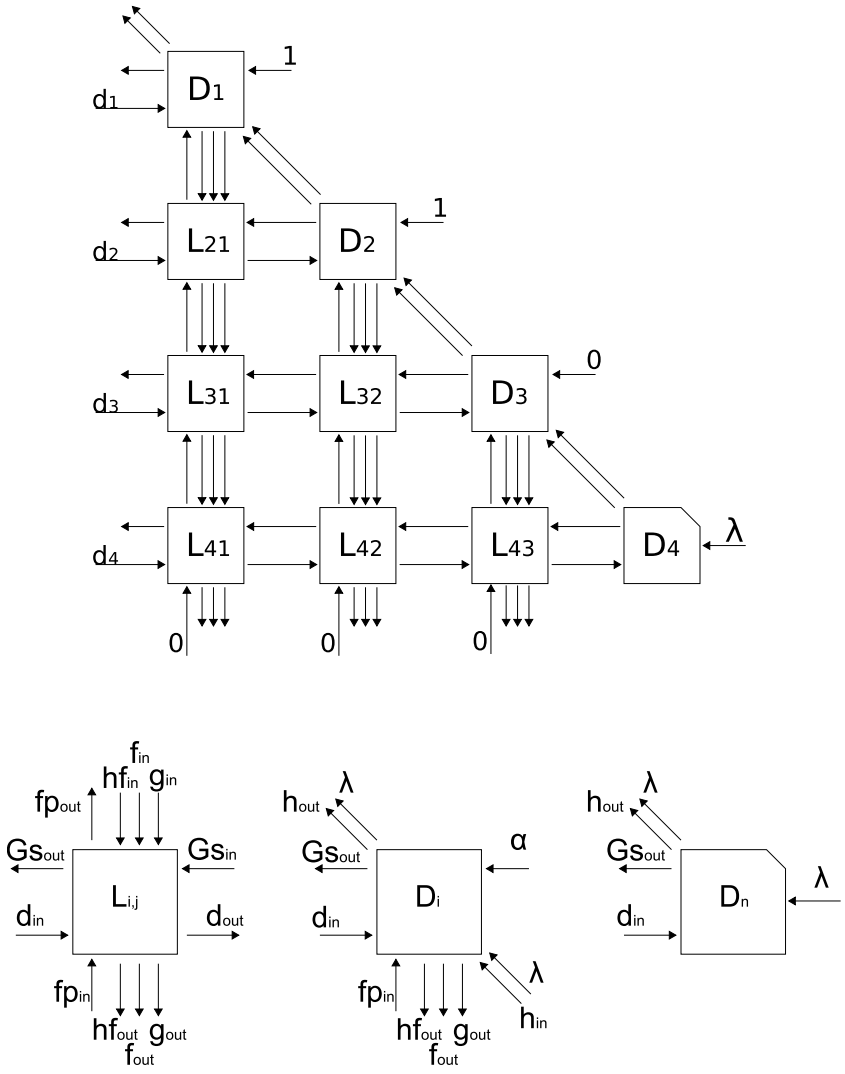


Figure 4.3: EF LD-RLS on a systolic array (for n=3)

The final systolic array with space complexity $\mathcal{O}(n^2)$ is shown in Figure 4.3.

The systolic array consists of three types of processing elements. The first type of PE uses elements of the matrix L and the second and third type use elements of the matrix D. The third type labeled D_n is a special case of the second type with different inputs and outputs but with a similar function.

	$\mathbf{L}_{i,j}$ **type**	$\mathbf{D}_i$ **type**	$\mathbf{D}_n$ **type**
	Registers: L	Registers: D, f, g, λ	Registers: D, f, g, λ
I.phase			
	Inputs: d_{in}, fp_{in}	Inputs: $d_{in}, fp_{in}, h_{in}, \lambda, \alpha$	Inputs: d_{in}, λ
	Outputs: d_{out}, fp_{out}	Outputs: $f_{out}, g_{out}, hf_{out},$	Outputs: $Gs_{out}, h_{out}, \lambda$
	$d_{out} = d_{in}$	$Gs_{out}, h_{out}, \lambda$	$f = d_{in}$
	$fp_{out} = fp_{in} + L.d_{in}$	$f = fp_{in} + d_{in}$	$g = D.f$
II.phase		$g_{out} = D.f$	$h_{out} = f.g$
	Inputs: $f_{in}, g_{in}, h_{in}, Gs_{in}$	$hf_{out} = \lambda + h_{in}$	$Gs_{out} = g$
	Outputs: $f_{out}, g_{out},$	$f_{out} = f$	$D = D\dfrac{1}{\lambda + f.g}$
	hf_{out}, Gs_{out}	$hp = f.g$	
	$f_{out} = f_{in}$	$h_{out} = h_{in} + hp$	
	$g_{out} = g_{in}$	$Gs_{out} = g$	
	$hf_{out} = hf_{in}$	if $\alpha = 1$ then	
	$Gs_{out} = Gs_{in} + L.g_{in}$	$D = D\dfrac{\lambda + h_{in}}{\lambda(\lambda + h_{out})}$	
	$L = L - \dfrac{f_{in}.Gs_{in}}{hf_{in}}$	else	
		$D = D\dfrac{\lambda + h_{in}}{\lambda + h_{out}}$	

Table 4.1: EF LD-RLS on a systolic array, operations in PEs

Particular types of processing elements perform FP operations shown in Table 4.1. The elements $L_{i,j}$ work in two phases, in the first phase the partial summation of f is prepared and it is completed in elements D_i . Then elements of matrix D is updated in SA elements D_i. f is used in the second phase to update elements of matrix L in SA elements $L_{i,j}$.

This proposed systolic array is only for study purposes, therefore it doesn't solve initialization of the array, i.e. how the initial values of L and D are loaded into the processing

elements.

4.2.3 The DF LD-RLS Algorithm

The LDU and UDL decompositions are very similar as described in Chapter 2. So we expect that the structure of the designed LD-RLS SA will be close to the structure of the UD-RLS SA. The implementation of the DF UD-RLS on systolic array has been described in (Chisci and Mosca, 1987).

At first the algorithm must be decomposed to fundamental vector and matrix operations. The result is based on Equations 3.51 in the Chapter 3 and it is relatively the same as in listing of Algorithm 2 in the part about the UTIA DSP platform.

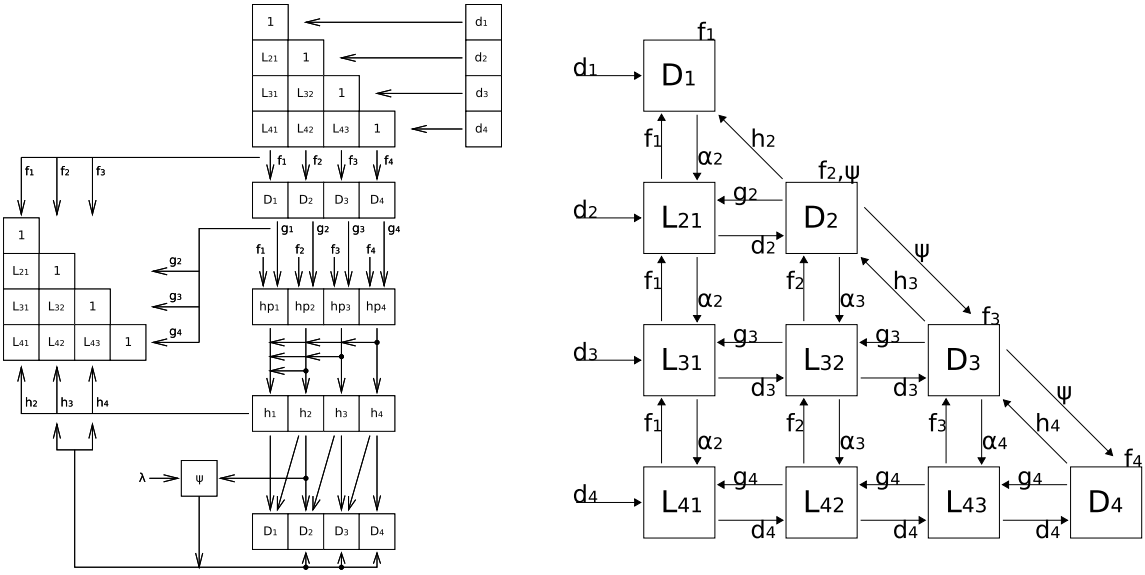


Figure 4.4: DF LD-RLS data dependence graph and PE dependence graph (for three estimated parameters)

Then the dependence graph must be prepared and analyzed. It is depicted in Figure 4.4. A one can see it has a structure similar to EF LD-RLS except for the forgetting factor. The directional forgetting factor is computed from forgetting factor λ chosen by the user and statistic $\zeta = h_2$ computed from the covariance matrix (i.e. from submatrices $L_{2:n,2:n}$ and $D_{2:n,2:n}$).

The dependence graph shows that the computation will be performed in two steps. In the first step, which is the same for the EF LD-RLS algorithm, variables f, g, h will be computed, and then the directional forgetting factor is evaluated and L and D matrices will be updated. The graph also determines the sequence of input data in time and movements of variables in the array. They must come in a correct order to the appropriate elements, therefore some delays will have to be introduced. If we make use of the obvious triangular shape of the array with $\frac{n(n+1)}{2}$ elements, the computation of the array will have time complexity $\mathcal{O}(n)$.

The proposed systolic array is shown in Figure 4.5. The systolic array consists of four types of processing elements. The first type works with elements of matrix L, and all others compute with elements of matrix D. The elements labeled D_n and D_1 are special cases of element D_i. Element D_1 computes the directional forgetting factor ψ apart from all variables computed in elements D_i.

The processing elements perform FP operations shown in Tables 4.2 and 4.3. All the processing elements work in two phases. In the first phase vectors f, g, h are evaluated along columns from left($j = 1$) to right($j = n$), where the incoming input data are shifted in time as shown in Figure 4.6. Data go through the array to the right side and then through the diagonal processing elements up to D_1, where the directional forgetting factor ψ is computed. Then if ψ is not equal to zero, elements of the matrices L and D are updated from the top elements to the bottom elements.

Figure 4.6 depicts the timing in the array, the left part depicts the first phase, and the right part is for the second phase. Small boxes with numbers in the left figure are delays which are shown to indicate the possibility to use the array in a synchronous mode when all input data arrive on the same time. Numbers written close to the processing elements are times when the elements work.

Figure 4.6 also shows that the proposed array has time complexity $\mathcal{O}(n) = 3n + 1$. This is a worse result than reached by the authors in (Chisci and Mosca, 1987), but they had an array where the number of elements was greater by $(n+1)$ elements. They reached the time complexity $\mathcal{O}(n) = 2n + 3$ of the DF UD-RLS algorithm with $\frac{(n+1)(n+2)}{2}$ processing elements. We use array with $\frac{n(n+1)}{2}$ processing elements.

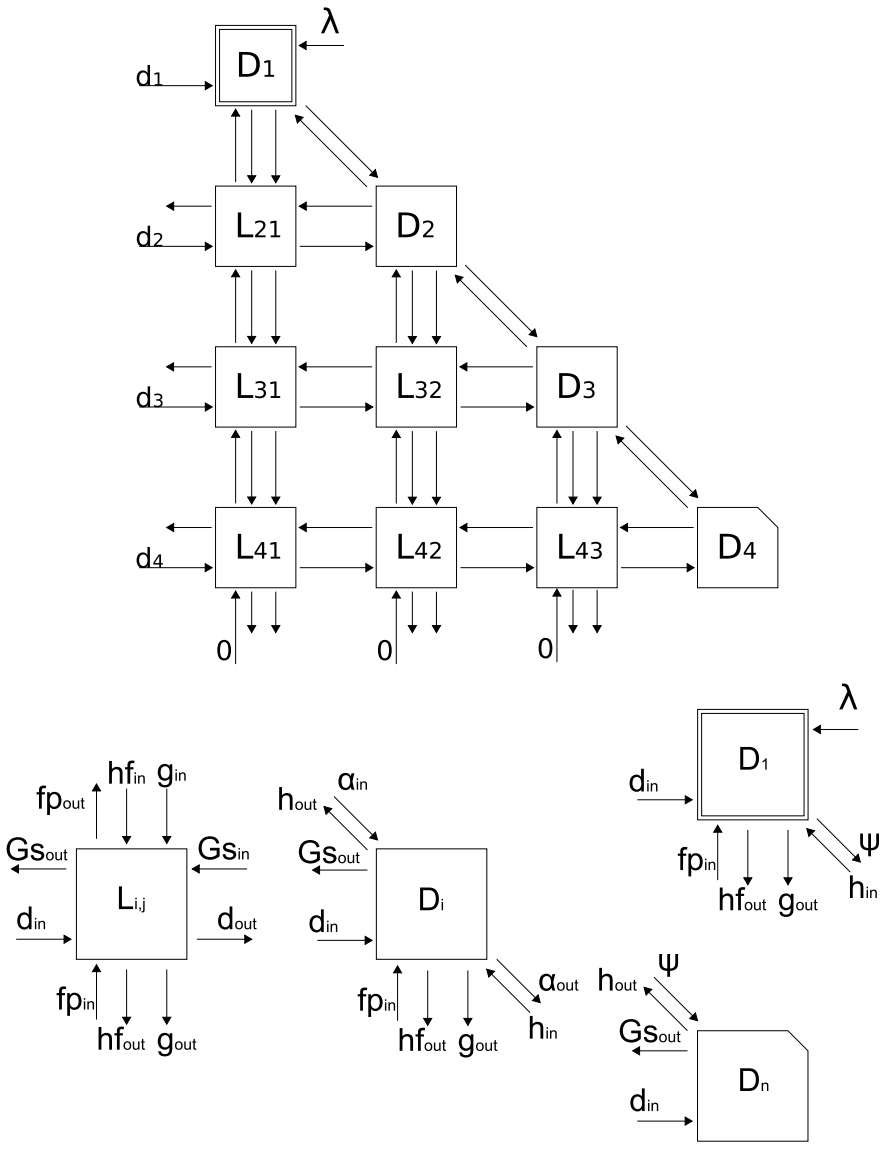


Figure 4.5: DF LD-RLS on a systolic array (for three estimated parameters)

	$\mathbf{L}_{i,j}$ **type**		$\mathbf{D}_i$ **type**
	Registers: L		Registers: D, f, g, h
I.phase		I.phase	
	Inputs: d_{in}, fp_{in}		Inputs: d_{in}, fp_{in}, h_{in}
	Outputs: d_{out}, fp_{out}		Outputs: h_{out}
	$d_{out} = d_{in}$		$f = fp_{in} + d_{in}$
	$fp_{out} = fp_{in} + L_{i,j}.d_{in}$		$g = D_i.f$
			$h = h_{in}$
			$h_{out} = h + f.g$
II.phase		II.phase	
	Inputs: g_{in}, hf_{in}, Gs_{in}		Inputs: α_{in}
	Outputs: $g_{out}, hf_{out}, Gs_{out}$		Outputs: $\alpha_{out}, g_{out}, Gs_{out}, hf_{out}$
	$g_{out} = g_{in}$		$g_{out} = g$
	$hf_{out} = hf_{in}$		$Gs_{out} = g$
	$Gs_{out} = Gs_{in} + L.g_{in}$		$\alpha_{out} = \alpha_{in}$
	if $hf_{in} \neq 0$ then		if $\alpha_{in} \neq 0$ then
	$L_{i,j} = L_{i,j} - hf_{in}.Gs_{in}$		$D_i = D_i \dfrac{\alpha_{in} + h}{\alpha_{in} + (h + f.g)}$
			$hf_{out} = \alpha_{in} + h$
			else
			$hf_{out} = 0$

Table 4.2: DF LD-RLS on a systolic array, operations in L and D_i PEs

D_1 type	**D_n type**
Registers: D, f, g, h, ψ	Registers: D, f, g, h
	I.phase
Inputs: $d_{in}, fp_{in}, h_{in}, \lambda$	Inputs: d_{in}
Outputs: g_{out}, hf_{out}, ψ	Outputs: h_{out}
$f = fp_{in} + d_{in}$	$f = d_{in}$
$g = D_1.f$	$g = D_n.f$
$h = h_{in}$	$h = f.g$
$g_{out} = g$	$h_{out} = h$
if $h_{in} \neq 0$ then	II.phase
$\psi_1 = \lambda.h - 1 + \lambda$	Inputs: ψ
$hf_{out} = 1 + h$	Outputs: Gs_{out}
if $\psi_1 \neq 0$ then	$Gs_{out} = g$
$\psi = \frac{\lambda}{\psi_1}$	if $\psi \neq 0$ then
$D_1 = D_1 \frac{1+h}{\lambda(1+(h+f.g))}$	$D_n = D_n \frac{\psi}{\psi + h}$
else	
$\psi = 0$	
$D_1 = D_1 \frac{1+h}{\lambda(1+(h+f.g))}$	
else	
$\psi = 0$	
$D_1 = \frac{D_1}{\lambda}$	
$hf_{out} = 0$	

Table 4.3: DF LD-RLS on a systolic array, operations in D_1 and D_n PEs

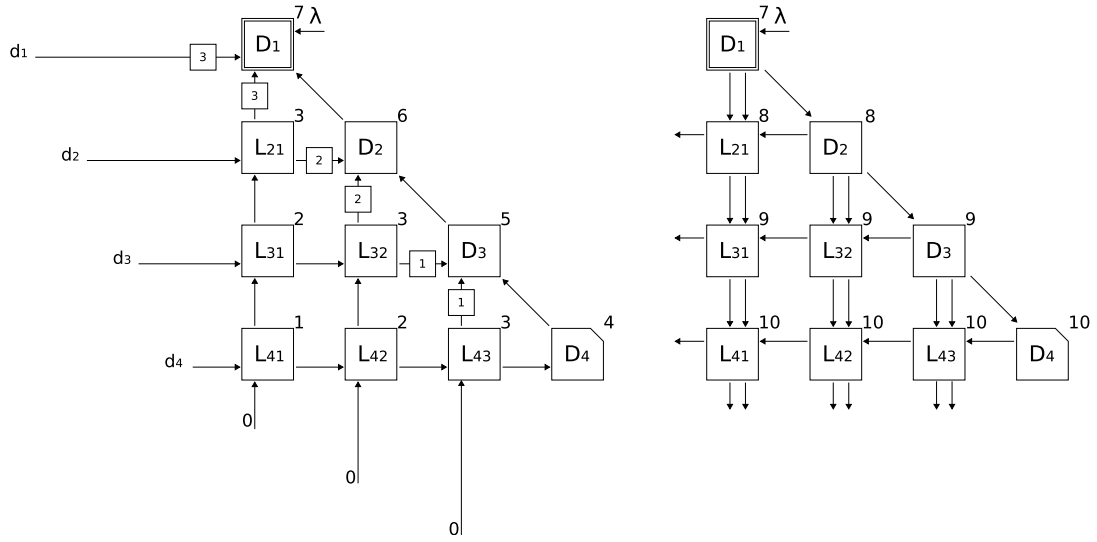


Figure 4.6: Timing of the DF LD-RLS systolic array (for three estimated parameters). Left side is the first phase and right side is the second mode

4.3 Implementation on the UTIA DSP platform

In this work the *UTIA DSP Platform*[2] is used for implementation of the DF LD-RLS algorithm. The UTIA DSP platform is a generic concept of a flexible, reprogrammable and reconfigurable hardware accelerator. The domain of use depends on the configuration of the platform, because its concept is based on the FPGA technology and its hardware reconfigurability. The platform is intended to be a hardware accelerator for a general-purpose processor in a system-on-chip.

The basic implementation of the UTIA DSP platform with pipelined floating-point operations such as addition, multiplication and division is denoted as the *Basic Computing Element*[3] (BCE). The BCE platform is used and extended in this work, so it is described at first. Then we present all partial steps necessary for platform extensions and the modified methodology for implementing the algorithms.

4.3.1 The Basic Computing Element

This part briefly describes the BCE platform, information about the BCE platform can be also found in (Kadlec et al., 2007) and (Daněk et al., 2008).

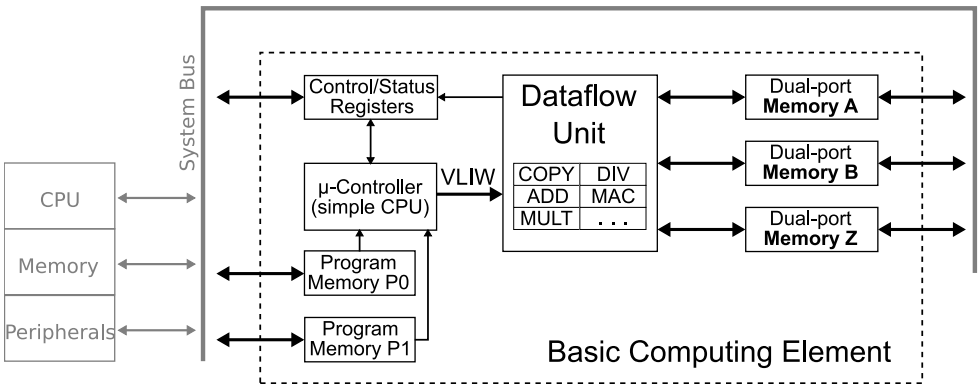


Figure 4.7: BCE from the point of view of the host CPU

[2]The UTIA DSP platform has been developed for DSP applications by J.Kadlec at the Department of Signal Processing, ÚTIA AV ČR, v.v.i, therefore its called UTIA DSP platform.

[3]The BCE platform has been developed and implemented by J.Kadlec. The implementation of the BCE platform is not an object of the thesis. The thesis is aimed at the implementation of algorithm on the BCE platform, extensions of the BCE platform, and the improvement of the implementation methodology.

Figure 4.7 depicts the BCE platform which consists of a *data-flow unit* (DFU), dual-ported data memories which can be connected to a host system for direct data exchange, a micro-controller (uC) and memories with programs for the micro-controller.

The BCE platform is highly configurable on several levels.

On the lowest level, i.e. hardware description in VHDL or another hardware description language, different computing units can be selected from libraries of units to achieve application requirements. Such units can be prepared for computation in fixed-point or floating-point, single, double or another precision, pipelined vector or matrix operations, etc. The original BCE used in this thesis contains three computing units. All of them use the pipelined floating-point single precision arithmetic. The pipeline latency for addition is 3 clock cycles (ClC); for multiplication it is 4 clock cycles and for division it is 16 clock cycles.

On the middle level, i.e. hardware integration as a model in Xilinx System Generator in the Matlab environment, the entire BCE accelerator is assembled. These tools allow simple integration and configuration of the platform parameters. For example, the following parameters can be configured: the number of computing units, the number of data memories, the micro-controller used, programmability of the data-flow unit from micro-controller, the number of data-flow units which allows to build accelerator with a SIMD-like structure. The data-flow unit consists of a finite state machine (FSM) which controls data paths and internal registers to perform the required macro-operation.

The DFU is able to read data or write data to each local data memory on each clock cycle. It also manages the initial and wind-up phases related to the pipelined operations. The initial configuration of the DFU used in the thesis is shown in Figure 4.8. It contains three computing units as mentioned above, three data memories and the Xilinx PicoBlaze3 as a re-programmable micro-controller with two interchangeable program memories. Switching between the program memories can be used as a substitute of slower hardware reconfiguration (Daněk et al., 2008).

The highest level is a program level where the micro-controller firmware is used to control the data-flow unit in the BCE by a sequence of operations or batches of operations. The micro-controller prepares VLIW instructions (*Very Long Instruction Word*) to control the DFU. Each VLIW contains the following information

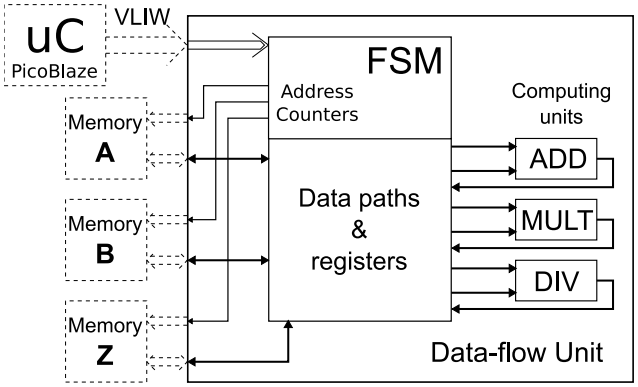


Figure 4.8: The origin BCE data-flow unit

- operations to be used,
- data memories that contain input arguments,
- data memory to store the result,
- the number of data items (the length of the vectors) to be processed in the operation.

A special level above the others is a program in the host CPU which controls the BCE accelerator, uploads firmwares to the micro-controller, and communicates with the BCE accelerator through the control and status registers. It also stores and reads data in the data memories. The user application (or system) in a CPU can prepare, compile and upload firmware to the BCE accelerator on the fly if necessary.

The data path is completely separated from the control path in the accelerator. Such a structure offers several advantages, it allows to change the data format and arithmetic for the same algorithm, to work with special data formats, to use the platform as a SIMD parallel accelerator. When the platform is used in a SIMD-like configuration, it contains one or more identical data-flow units with separate computational units and data memories, and it is one way how to increase data throughput.

The used BCE accelerator is suitable for operations with vectors because it uses pipelined FP computation units. So the next part is aimed at using FP operations.

FP Pipelined Operations in the BCE

The used BCE accelerator supports basic pipelined vector operations shown in Table 4.4. The operation COPY is realized in the data-flow unit, and it is used to copy data between data memories. It is useful when the sequence of operation needs the data distributed in different memories. The basic operations (ADD, MULT, DIV) are implemented directly in hardware as mentioned above. The other operations are performed in the DFU as sequences of the basic operations. All operations in Table 4.4 except DPROD are element-wise operations, i.e. each element of one input vector is processed with the corresponding element from the second vector, and the result is stored at the same position in the output vector as shown in the table.

Operation	**latency**[ClC]	**Description**
VCOPY	0	$Z_i = A_i$
VADD	3	$Z_i = A_i + B_i$
VSUB	3	$Z_i = A_i - B_i$
VMULT	4	$Z_i = A_i + B_i$
DPROD	≥ 3	$Z_0 = \sum_i (A_i * B_i)$
VMAC	8	$Z_i = Z_i + A_i * B_i$
DIV	16	$Z_i = A_i / B_i$

Table 4.4: Basic BCE FP operations

Utilization

The utilization of a computational unit for each pipelined operation[4] in the BCE can be expressed as

$$U_{op} = 100.\frac{n}{l_{op} + n + 2} \quad [\%] \tag{4.1}$$

where n is the number of elements in the vector calculated in the pipelined unit, and l_{op} is the latency of the pipelined unit. The latency is introduced by initialization of the pipeline.

[4]Operation which returns one result each clock cycle after a certain latency.

The utilization can be between 0% and 100%. It grows asymptotically to 100% with the rising number of elements in the computed vector. Therefore all equal operations in the algorithm should be combined in operations on the longest possible vectors. Because of memory reading/writing between operations 2 clock cycles must be added to the time of operation.

The utilization of one computational unit for the entire algorithm can be expressed as

$$U_{unit} = \frac{\sum n_i}{T_{alg}}, \tag{4.2}$$

where n_i is the number of vector elements in the operation i which is processed in the unit, and T_{alg} is the total time of the algorithm in clock cycles[5].

One of the main advantages of the BCE platform is its predictable duration of computation. The number of clock cycles for all three steps (pipeline initialization, data processing, wind-up) for all the operations is known. Thus the accelerator is suitable for real-time applications where the time of a computation must be known.

The BCE platform is a universal accelerator with possible on-the-fly software reconfiguration of its hardware function by changing the micro-controller firmware which controls the data flow between the computing units and data memories. Due to its universality the utilization of the computing units is low. If the utilization is the main criterion of an implementation the algorithm should be implemented in hardware as a new computation unit. The solution with an algorithm implemented in firmware has the advantage of rapid algorithm implementation possibly with fast debugging.

The basic BCE platform has separate data and control flow (i.e. the data don't go through the control micro-controller). There is only one feedback signal DONE from the DFU to the micro-controller. The separate data path enables to use the same firmware with the same control algorithm for computation in different arithmetic domains, e.g. single/double precision floating-point, fixed-point, etc. Alternatively the arithmetic can be switched on-the-fly through hardware reconfiguration. In this case the data transformation between different arithmetic should be solved. This can decrease the power consumption when the double precision FP units are used only when necessary.

[5] We assume that the unit is pipelined and processes one element per clock cycle.

The DF LD-RLS algorithm contains branching that depends on the decision if the inputs are excited sufficiently. The micro-controller makes the decision according to the result of a comparison, so a new signal from a newly added comparative unit must be added. Therefore the structure of the BCE platform must be modified at this point.

The LD-RLS algorithms work with data in vectors and triangular matrices. To maximize the utilization of the pipelined operations all data in the matrices should be vectorized. Therefore the BCE platform is extended with programmable address counters in the DFU for vectorizing such data in the matrices.

4.3.2 Implementation in the BCE Platform

The BCE platform allows to implement algorithms on several levels, or divide computation between these levels. As described in the previous subsections the platform has four levels where the algorithm can be implemented :

- software in the host CPU,
- firmware in the micro-controller,
- hard-wired data path control in the DFU,
- hardware IP cores with a new computing unit for the whole algorithm.

Each level has its advantages and disadvantages. On lower levels, a design and prototyping is more complicated, but the implementation is more efficient with better utilization. The choice where the algorithm should be implemented depends on selected criteria as shown in the book (Niemann, 1998).

For maximal performance the lowest level should be the most suitable, because there are minimal overheads with the operations. But there are drawbacks of implementations on lower levels, the major one is the development process. The process is slower and needs an experience in hardware design. The adjustment and debugging of the implemented algorithm is also slower and more complicated on lower levels.

For these reasons the best way to implement LD-RLS algorithms in the BCE platform is to divide the algorithms between the micro-controller firmware and the hardware control

of the data path. The DF LD-RLS algorithm has a high data dependence, therefore implementing it as one hardware IP core is impractical. On the other hand, its implementation in the host CPU software wouldn't be efficient enough.

Implementation Steps

Generally, the algorithm implementation in the BCE platform is done in the following steps:

- prepare the algorithm as a sequence of basic pipelined operations supported by the platform,
- the hardware part
  - implement the required but missing basic pipelined operations as HW IP cores for the selected arithmetic,
  - prepare and generate the hardware for the BCE accelerator,
- the software/firmware part
  - sort and group the operations according to the data dependences in the algorithm,
  - organize the data in the memories according to the algorithm data-flow graph and possible configurations of the DFU data paths,
  - write the firmware for the control micro-controller,
  - write the software (application) for the host CPU (MicroBlaze),
- simulate / test the implemented function

All these steps will be discussed in the same order for DF LD-RLS in the following text.

In many projects, people develop software tools which perform these steps automatically for platforms used in their projects. Unfortunately none of them is generic because each of these tools is specific for each platform or a class of platforms with the same architecture or structure. Several examples are shown in the book (Niemann, 1998). These tools must

analyze, synthesize and generate code for building software and/or hardware parts. The code is highly hardware specific.

For the BCE platform, there isn't any tool and therefore such a tool has been implemented as a part of this work. The developed tool automates the generation of the software part, i.e. generates source codes for the host CPU and the micro-controller firmware from an algorithm specified in the Matlab environment. The tool increases productivity in implementation of algorithms because algorithms can be developed and simulated in the Matlab environment and then the verified algorithm is automatically converted to the software/firmware for the BCE platform without any other intervention. The tool is described in the next chapter as one of the results.

Implementation of the two LD-RLS algorithms, i.e. LD-RLS with exponential and directional forgetting, are described in the following section. Then the text continues with the description of all implementation steps.

Algoritmus 1: EF LD-RLS suitable for the vector architecture

Data: L,D

Input: d,λ

f1 $f_i = L_{:,i} \cdot d_:$

f2 $g_i = f_i * D_i$

f3 $hp_i = g_i * f_i$

f4 $h_i = \sum_{j=i}^{n} hp_j$

f5 $hf_i = \lambda + h_i$

f6 $Gs_{i,j} = g_j * L_{i,j}$

f7 $Gl_{i,j} = \sum_{k=j}^{n} Gs_{i,k}$

f8 $Hl_{i,j} = f_i * Gl_{i,j}$

f9 $Du = D * hf_{i+1}$

f10 $\bar{D} = Du/hf_i$

f11 $\bar{D}_{1,2} = \bar{D}_{1,2}/\lambda$

f12 $Lu_{i,j} = Hl_{i,j}/hf_{j+1}$

f13 $\bar{L} = L - Lu$

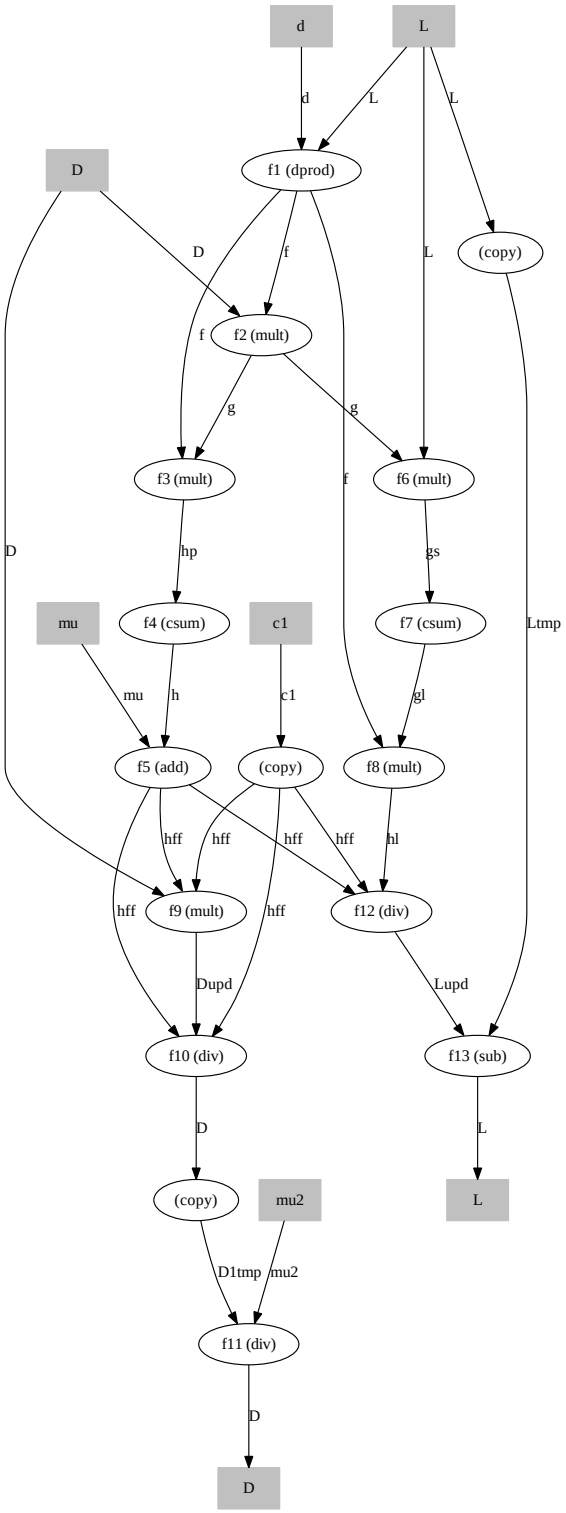


Figure 4.9: EF LD-RLS Data-flow graph

4.3.3 The EF LD-RLS Algorithm

The EF LD-RLS algorithm was implemented first. It was implemented mainly to test the developed automated tools and to compare the implementations of EF LD-RLS and DF LD-RLS.

Algorithm 1 is the implemented vectorized form of the EF LD-RLS algorithm. The data-flow graph of the algorithm is in Figure 4.9.

The first part of the EF LD-RLS algorithm, where the h and gl statistics are evaluated, is the same in both algorithms. Then the exponential forgetting factor λ is used to update all elements of matrices L and D.

The operations with operand $*$ are element-wise multiplications. All operations in the algorithm can be computed as pipelined vector operations with two exceptions: `f4` and `f7`. These two operations require *cumulative summation* (CSUM). The function can be implemented as a sequence of individual additions in program in the host CPU, or as a new operation of the BCE platform. The second option is certainly more efficient, and the implemented CSUM operation is described below.

In Figure 4.9 there are operations labeled "copy". These operations copy data from one memory to another. This is necessary when the input and output variables are the same or if both input variables are stored in the same memory.

4.3.4 The DF LD-RLS Algorithm

The DF LD-RLS algorithm is implemented in the BCE platform as listed in Algorithm 2. The first part of the algorithm, which is the same as for the EF LD-RLS algorithm, evaluates the statistic $\zeta = h_2 = \mathbf{z}^T C \mathbf{z}$. Then the algorithm branches in two ways. In one way, when input data doesn't bring a new information about the estimated parameters ($\zeta \leq \epsilon_0$), the L and D matrices don't get updated and then only the element D_1 is updated with the forgetting factor λ. The second way, when the matrices are updated, contains the computation of the directional forgetting factor ψ and update of L and D. There the algorithm checks if ψ is not too small ($\psi \leq \epsilon_0$). And according to this comparison the algorithm updates only the estimated parameters or it updates also elements of the

decomposed covariance matrix.

In the algorithm operations are organized to maximize the potential of the platform in vector computing. Hence the operations `f27` and `f28` are computed from temporary variables prepared in steps `f23-f26`. From this point of view, one problematic variable is ψ (`f13`,`f14`,`f22`) which is computed from scalar variables and constants. For the platform, this is the worst case because the operations have the greatest overhead (the pipelines must be initialized and wound-up to compute one value).

Algorithm 2 is depicted as a vector data-flow graph in Figure 4.10, and its control-flow graph is in Figure 4.11 where the branching of the algorithm is shown.

Due to branching in the algorithm a new operation for comparing two values is necessary. The function must be reachable from the data path, therefore it should be implemented in hardware as a new computing unit.

4.3.5 Design Flow of the Implementation

The design flow used for building the hardware and software of a system-on-chip with the BCE accelerator is shown in Figure 4.12. The flow consists of two main parts which can be divided as described below

- Hardware part
  - IP cores of the computational units
  - IP core of the BCE platform as a SoC peripheral
  - complete system on chip with the soft-core processor MicroBlaze
- Software part
  - firmware for the BCE micro-controller
  - software for the host processor MicroBlaze

In our case, the basic computing units for single precision floating-point arithmetic are generated by the Xilinx CoreGen tool (XCG). They are connected together with the configuration of the data-flow unit, data memories and the PicoBlaze micro-controller in

Algoritmus 2: DF LD-RLS suitable for the vector architecture

Data: L,D

Input: d,λ,$\lambda_1 = (\lambda - 1)$

f1 $f_i = L_{:,i} \cdot d_:$

f2 $g_i = f_i * D_i$

f3 $hp_i = g_i * f_i$

f4 $h_i = \sum_{j=i}^{n} hp_j$

f5 **if** $h_2 \leq \epsilon_0$ **then**

f6 $\alpha = 1 + hp_1$

f7 $\beta = \lambda * \alpha$

f8 $Dt = D_1/\alpha$

f9 $\bar{D}_1 = Dt$

else

f10 $gs = g * L$

f11 $gl = \sum_{j=i}^{n} gs_j$

f12 $hl = f * gl$

f13,f14 $\psi_d = (\lambda * h_2) + \lambda_1$

f15 **if** $\psi_d \leq \epsilon_0$ **then**

f16 $\alpha_{1:2} = 1 + h_{1:2}$

f17 $\beta = \lambda * \alpha_1$

f18 $\gamma = D_1 * \alpha_2$

f19 $\bar{D}_1 = \gamma/\beta$

f20 $Lupd_{:,1} = hl_{:,1}/\gamma_2$

f21 $\bar{L}_{:,1} = L_{:,1} - Lupd_{:,1}$

else

f22 $\psi = h_2/\psi_d$

f23,f24 $a_1(1:n) = \{1, \psi, \psi, ..., \psi\}$

f25,f26 $a_2(1:n+1) = \{h_1, h_2, ..., h_n, 0\}$

f27 $\gamma = a_1(1:n) + a_2(2:n+1)$

f28 $\delta = a_1(1:n) + a_2(1:n+1)$

f29 $Dupd = D * \gamma$

f30 $\bar{D} = Dupd/\delta$

f31 $\bar{D}_1 = \bar{D}_1/\lambda$

f32 $Lupd = hl/\gamma$

f33 $\bar{L} = L - Lupd$

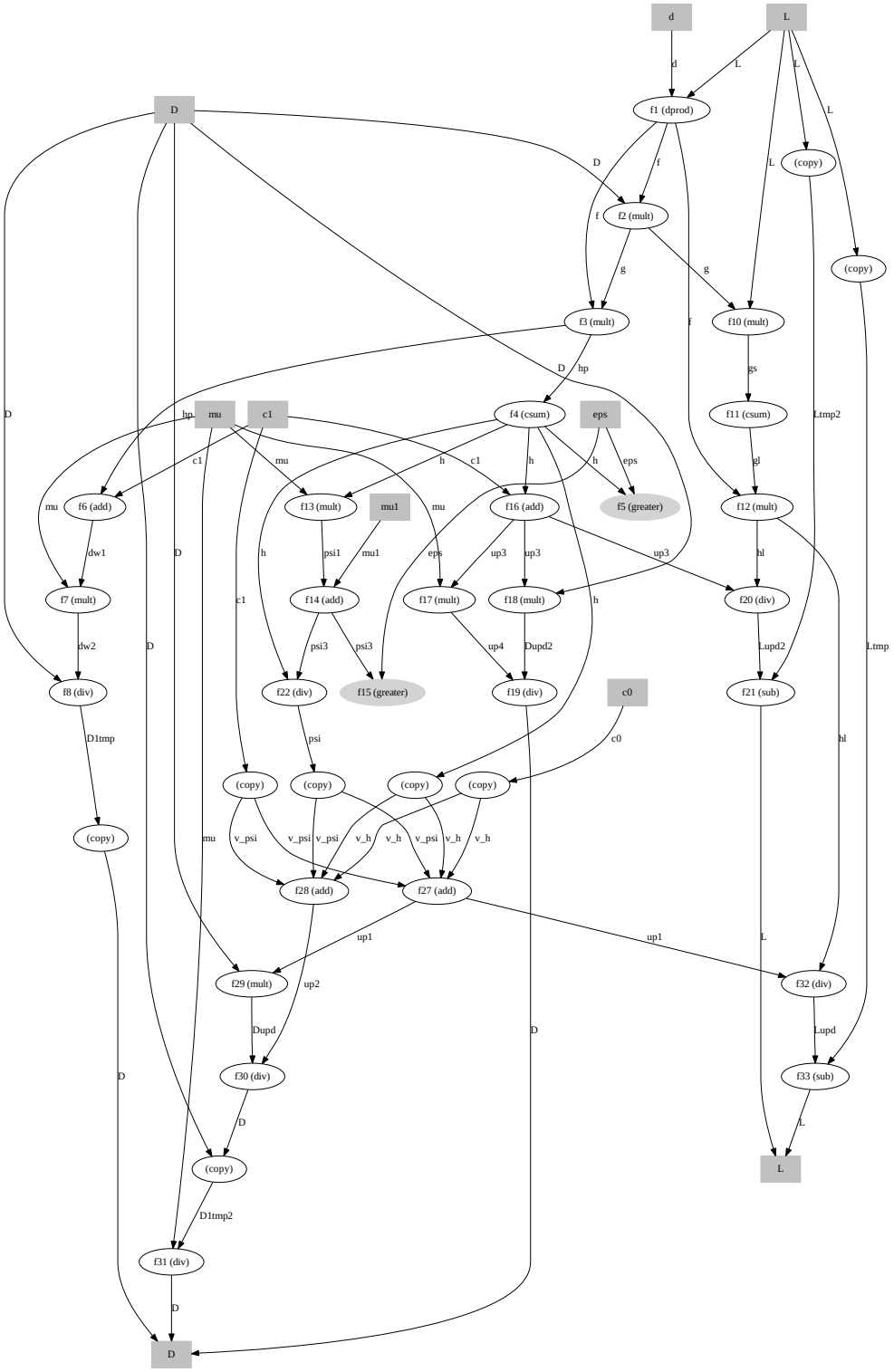


Figure 4.10: DF LD-RLS Data-flow graph generated by the automatic tool

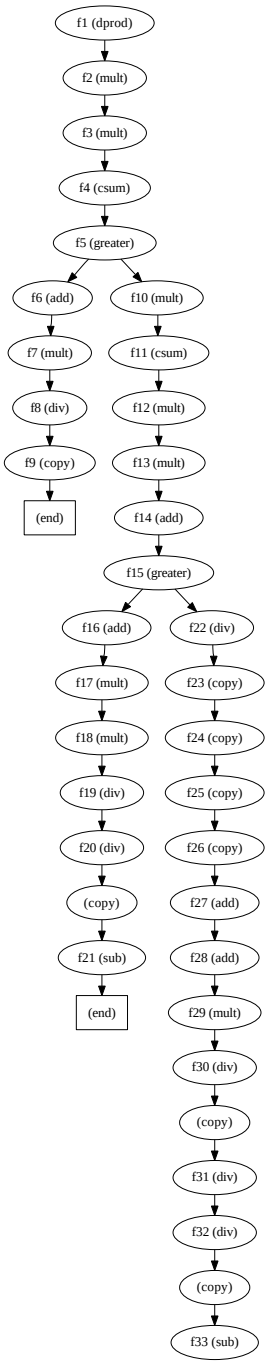


Figure 4.11: DF LD-RLS flowchart generated by the automatic tool

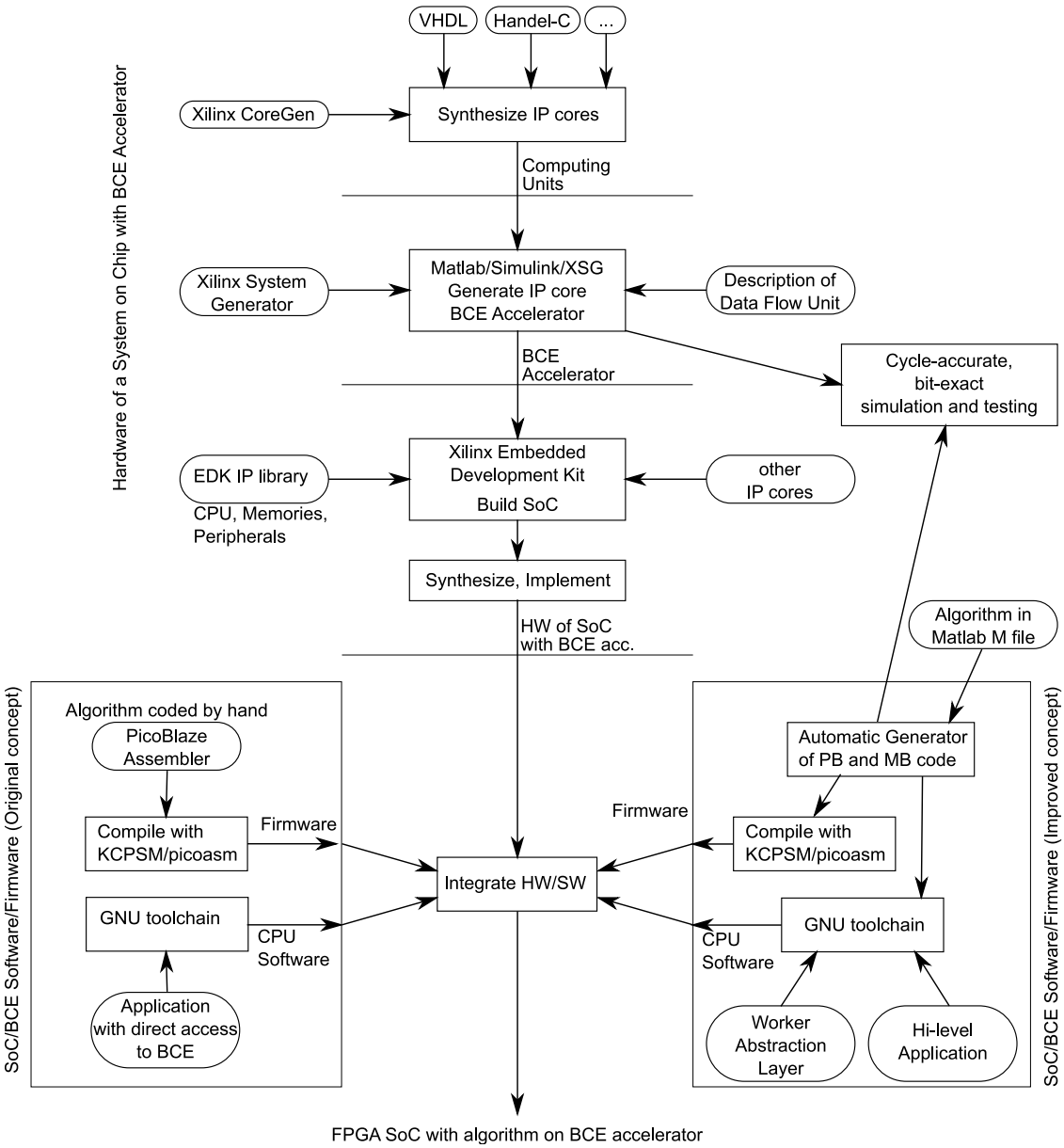


Figure 4.12: Design flow for building a SoC with the BCE accelerator

the Simulink model of the whole BCE accelerator in the Matlab/Simulink environment with the Xilinx System Generator tool (XSG). There a cycle-accurate and bit-exact simulation and testing can be performed with firmware prepared automatically based on the algorithm described in Matlab function.

In the Xilinx Embedded Development Kit environment (EDK), the IP core of the BCE accelerator generated by the XSG tool in a standard way is added as a peripheral into whole system together with a processor, memories and other peripherals. The system is synthesized and implemented in a bitstream for FPGA configuration. The final bitstream is a representation of the hardware of the system on chip with the BCE accelerator.

The system needs a software to perform the required functions. It consists of two parts at least. The first is an application which runs on the MicroBlaze processor. And the BCE accelerator needs a firmware. The firmware contains sequences of operations of the required algorithm. The sequence can be written by hand in the PicoBlaze assembler. Newly it can be automatically generated from an algorithm in the Matlab. This generator has been developed as apart of this thesis. Steps which have been modified for this work are described in the following text in the same order as the design flow of building SoC goes.

4.3.6 New Operations in the BCE Accelerator

The original BCE accelerator contains several basic vector operations, see table 4.4. Both algorithms, EF LD-RLS and DF LD-RLS need extra operations which must be added to the BCE accelerator if they are not to be implemented in software. And, of course, if we are trying to improve the implementation for speed and throughput, all operations must be as close to hardware as possible.

Both algorithms (Algorithm 1 and Algorithm 2) show which operations are required in addition to the common operations. Both the algorithms need cumulative summation, and the DF LD-RLS algorithm also needs comparison of a variable with a constant.

FP Comparator

The function for comparing two FP numbers is necessary for the DF LD-RLS algorithm. This function can be classified as a basic operation because it works directly with two values, and it can process them in a pipeline. The XCG tool contains this operation for the FP arithmetic, and so it has been prepared with the tool as an IP core with parameters shown in Table 4.5. The IP core is connected to the BCE data-flow unit similarly as all other basic operations (ADD, MULT, DIV). The DF LD-RLS algorithm needs this function for algorithm branching, therefore the operation has a non-standard connection in the path data unit. The result is normally stored in the data memory if necessary, but it also sets a new flag in the status register which is accessible to the BCE micro-controller. The flag is used for branching in the algorithm. The generated core *FP Comparator* (CMP unit) is pipelined and can process vectors. In the DF LD-RLS algorithm we need it only to compare a scalar variable with a constant.

Slices FF	4 input LUTs	Occupied Slices	Block RAMs	DSP48	**Latency[ClC]**
10	80	35	0	0	2

Table 4.5: Implementation parameters of the added FP comparator

FP Cumulative Summation

This function computes the vector of cumulative summations (CSUM) z from the input vector a

$$z(i) = \sum_{j=1}^{i} a(j). \tag{4.3}$$

The cumulative summation is necessary for LD-RLS algorithms, and since it can be implemented in more ways, it should be analyzed for more efficient implementation in the BCE platform. The serial implementation is directly visible from Equation 4.3 and the first diagram in Figure 4.13. In this case, the input values are summed sequentially. The function can also be implemented in parallel (the second diagram in Figure 4.13), but it has worse parameters for our BCE platform than the serial form as shown in the following analysis.

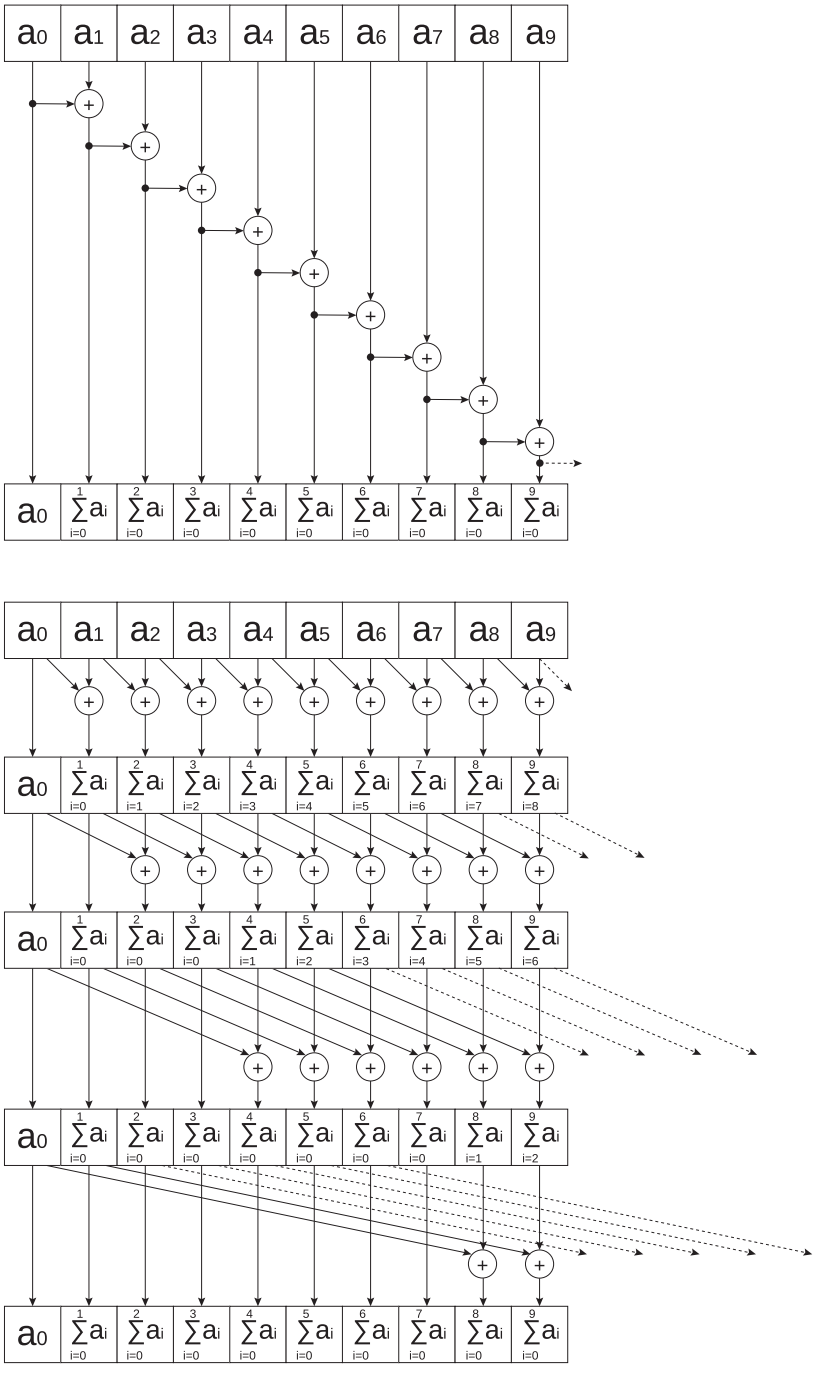


Figure 4.13: Two possible implementations of the cumulative summation (a-sequential; b-parallel)

The function has a data dependence in one direction (i.e. for the next output we need all previous inputs or the last output). In a serial implementation, the next value is computed only when the previous summation is evaluated, so the throughput of the CSUM function is determined by the adder latency. The first result is a direct copy of the first input, thus the delay of the operation for a vector of N numbers is

$$t(N) = 2 + (t_{adder} + 1) \cdot (N - 1) \quad [ClC]. \tag{4.4}$$

For the parallel implementation of the CSUM function in the BCE platform with one pipelined ADD unit, when each set of summation is processed as one pipelined operation, the delay of the operation with a vector of N numbers is

$$t(N) = 2 + (k \cdot t_{adder}) + \sum_{i=0}^{k-1} (N - 2^i), \tag{4.5}$$

where k is the number of sets of pipelined additions at one time.

$$2^k < N \leqq 2^{k+1}, k = \lfloor 1 + log_2(N - 1) \rfloor. \tag{4.6}$$

These two functions can be compared by the latency of the ADD unit and the number of summands.

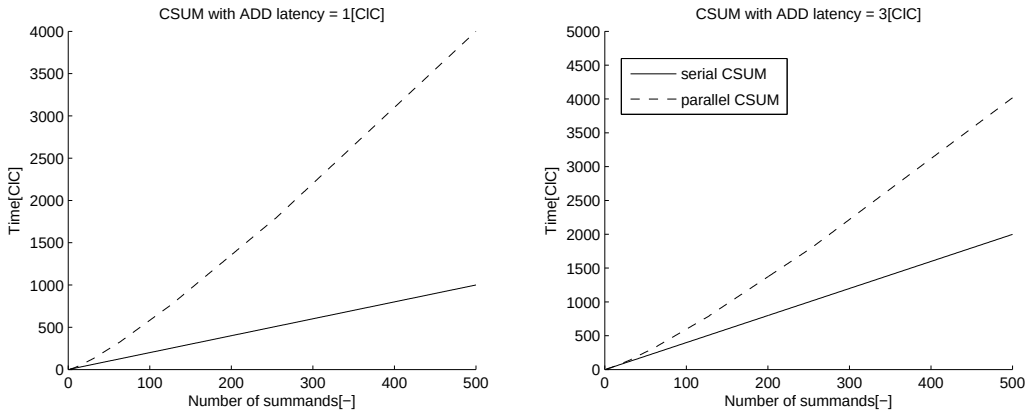


Figure 4.14: Time of computation of the CSUM operation in the BCE with one ADD unit(latency = 1 and 3 clock cycles)

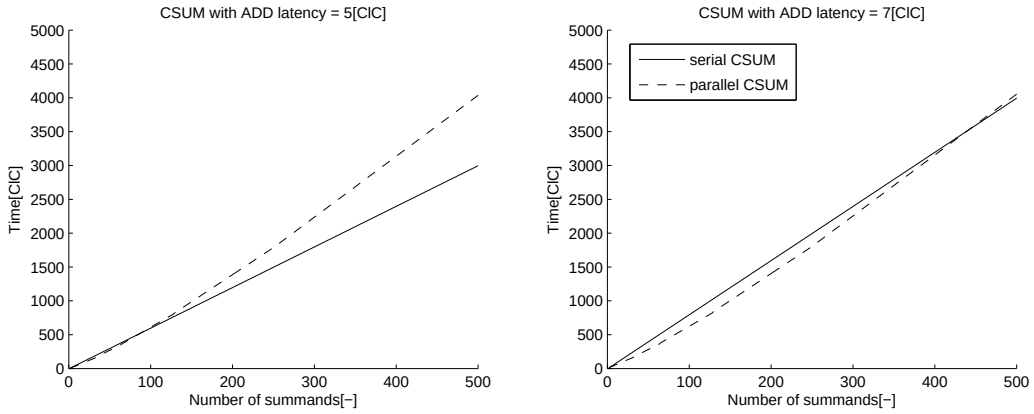


Figure 4.15: Time of computation of the CSUM operation in the BCE with one ADD unit(latency = 5 and 7 clock cycles)

A set of graphs in Figures 4.14 and 4.15 shows that the serial implementation of the CSUM function is faster than the parallel implementation in the BCE platform with one ADD unit if the latency is lower than 5 clock cycles, for higher latencies it depends on the number of summands.

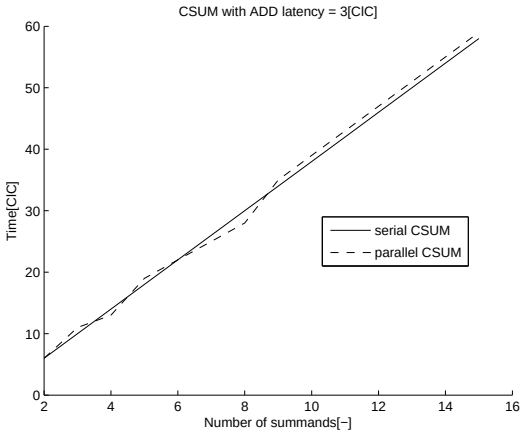


Figure 4.16: Detail of the beginning of the CSUM graph with the latency of ADD unit equal to 3 clock cycles

The graph in Figure 4.16 is a detailed view of the beginning of the graph in Figure 4.14 for the ADD latency equal to 3 clock cycles. It shows similar times for both implementa-

tions. The parallel realization has a non-linear dependence on the number of summands with jump shifts with new sets of summations, i.e. after the numbers of summands (N_s) is equal 2^k. It implies that parallel CSUM is the most effective for $N_s = 2^k$.

A general view of both realizations can be seen in Figure 4.17. The figure shows times of computation for both implementation in relation to the number of summands and latency of the ADD unit. The parallel CSUM is drawn with the dark surface, and the serial CSUM has the light surface. It shows that the serial CSUM is better for an ADD unit with low latencies. The border is depicted in the right figure, where the surfaces of the worse implementations are shown (it is a top view of the 3D graph).

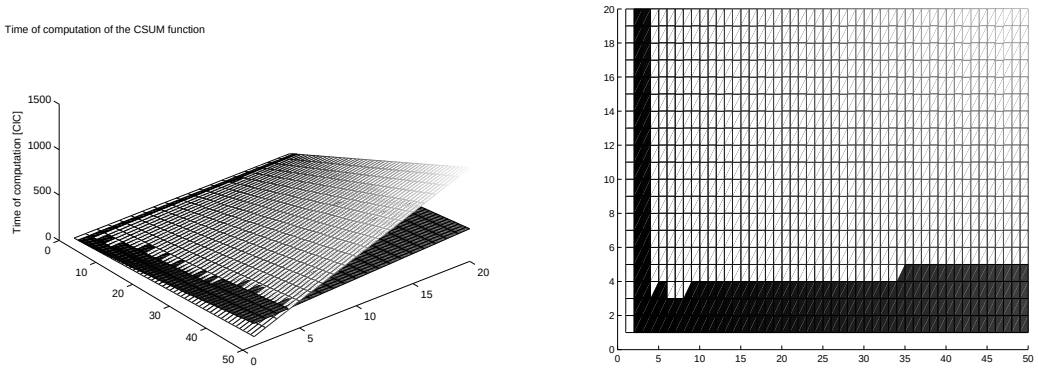


Figure 4.17: Times of computations for the serial and parallel CSUM. The dark surface is a parallel CSUM and the light surface is a serial CSUM.

The serial implementation has an advantage over the parallel implementation from the point of view of the BCE platform. It can use one memory for input and output operands simultaneously due to the intervals between reading the input memory when the addition is running. The parallel implementation can use one memory for input and output only if it skips one clock cycle between reading two input values and then the pipeline of the ADD unit will have only 50% utilization.

The analysis shows that the serial CSUM is better for the BCE platform with one ADD unit with the latency equal to 3 clock cycles.

4.3.7 Algorithm Vectorization

The used basic BCE platform contains pipelined computing units which have the highest throughput for operations with the longest vectors. To maximize the platform throughput we must optimize the data and operations in the algorithm.

Extension of the BCE Data Flow Unit

The LD-RLS algorithm can be computed as two nested loops with an element-wise computation, or it can be computed in vectors and triangular matrices. The basic BCE platform uses a DFU that works directly with vectors because of the pipelined computing units, and the data are organized in memories sequentially. The data must always be organized correctly in the data memories, i.e. as sequentially organized column vectors.

For this reason the DFU has been extended with programmable address counters to process all data as vectors (as shown in Figure 4.18). This extension has been done mainly to support triangular matrices which are necessary in the LD-RLS algorithms. For such shapes of data the address counters don't change the address linearly. This extension saves space in the data memories because all variables are organized in memories as linear vectors, and only elements with non-zero values from the diagonal and triangular matrices are stored. Triangular matrices stored this way obviously save about half the space. But their processing needs two nested address counters. It has a positive implication that loops controlled in the host CPU or in the micro-controller firmware can be moved closer to the computing units, which can save processing time.

For correct setting of both nested address counters their configuration in operations must contain the following information

- starting element in the variable (if it is a vector or matrix),
- increment in the inner loop,
- increment of the starting offset in the outer loop,
- flags how to use the increments in the loop (don't use, add, subtract).

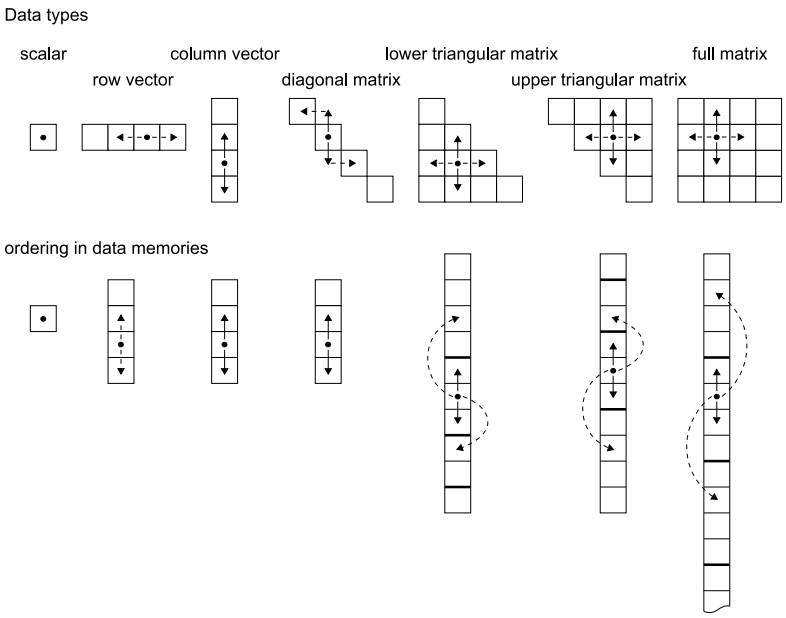


Figure 4.18: Vectorization of the basic types of variables in column vectors stored in the data memories. The arrows depict neighbour elements in the corresponding representations

Thus a VLIW instruction (see Figure 4.19) has been modified to take the required configurations of counters for all data memories. The figure depicts effects of the VLIW word on a generalized double loop processed in the modified DFU. The field *Dir* in the VLIW word controls multiplexers that select memories for operands $O1$,$O2$ and result R. Flags $RMode, O1Mode, O2Mode$ control if the counters get incremented or decremented or they stay the same.

The second modification of the DFU done in this work is addition of multiplexers and demultiplexers in the data path between the DFU and the data memories.

The original BCE platform supports the use of operations only with specific data memories. This solution has an advantage in a shorter critical path in the hardware design, thus the accelerator can run with a higher clock frequency. The platform supports the function for copying data from one memory to another, and it can compensate for the required but missing combinations of data memory organizations. For simpler algorithms with fewer variables this is sufficient.

In cases when the COPY function should be used many times in a complex algorithm

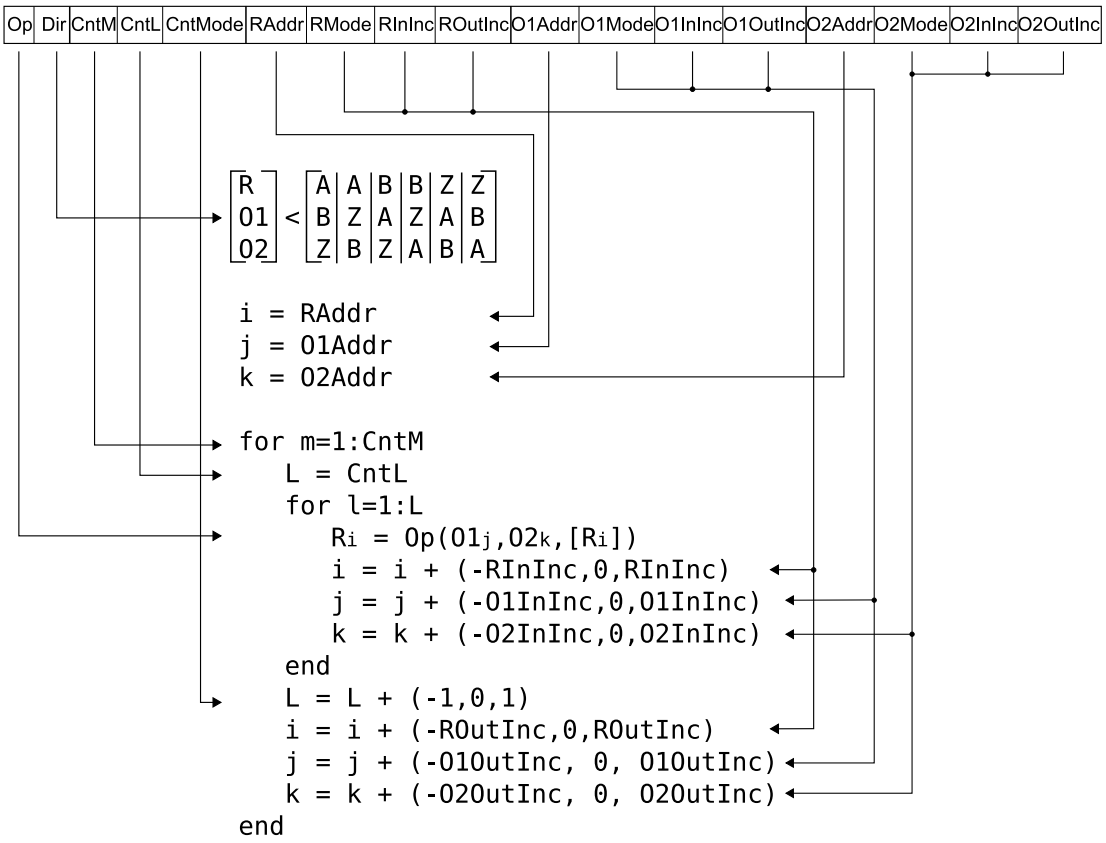


Figure 4.19: A new format of the internal VLIW word and its effect on the DFU

(e.g. DF LD-RLS), a better solution is to change the VLIW instruction to take not only the type of operation (COPY, ADD, MULT, ...), but also use of the data memories (which memory will contain operands and where to store the results). Thus the VLIW instruction will directly control the multiplexers and demultiplexers, and it will allow to use an operation with any combination of data memories.

The solution decreases the maximal clock frequency of the BCE platform about 2 MHz due to the longer critical path in the design (for the clock frequency of the BCE platform 62.5MHz). A benefit of this modification is a simplification for users when they have more freedom in scheduling operations and variables in their algorithms.

The last modification of the DFU is the above mentioned feedback signal from the FP comparator (CMP) unit. A result of the CMP unit is not only stored in the data memory as a result of a normal operation, but it is saved as a logical (one-bit) flag in the status word which can be read by the micro-controller. Then the firmware can use the flag to branch the program according to its value.

All hardware modifications of the BCE platform are schematically depicted in Figure 4.20. The next step in the design flow is generation of the IP core of the BCE accelerator in a standard way by the XSG tool. The following text is aimed at software development which computes the DF LD-RLS algorithm.

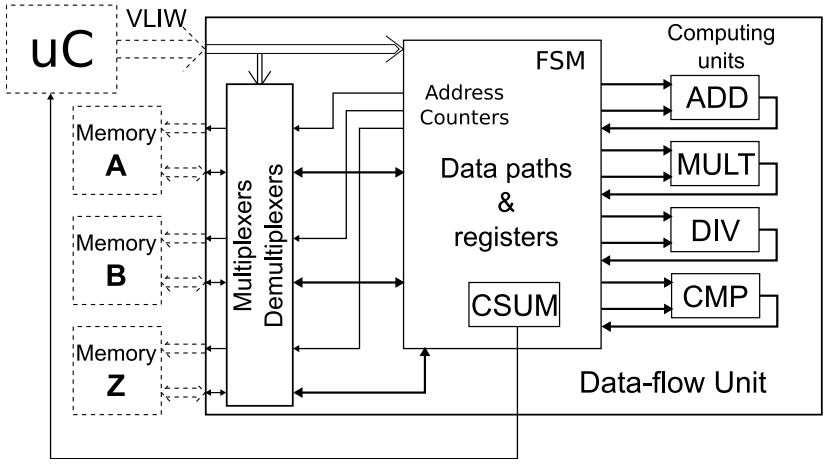


Figure 4.20: Diagram of the modified Data Flow Unit

4.3.8 Automatic Generator of Firmware for the BCE Platform

Software for a SoC with the BCE accelerator consists of two parts - an application in the MicroBlaze processor and firmware for the BCE micro-controller (PicoBlaze in our case). The firmware is originally written in assembler by hand.

The motivation to develop an automatic generator is mainly the simplification and acceleration of development and implementation of algorithms in the BCE platform. Then the generated code is errors free for any algorithm. The generator is written as a set of Matlab functions and its input is a special form of an algorithm in a Matlab script, so the algorithm can be tested before implementation and then implemented without manual interventions.

Another advantage of this approach is a possibility to write a scalable algorithm as a function of its order N, i.e. a universally written algorithm can be implemented for different sizes of its inputs, outputs and internal states by changing the value of order N during the generation. It is possible due to the description of variables and operations that contain their sizes as parametric polynomial functions with the parameter N.

The whole process of generation is still so simple that it can be implemented as an application in an embedded system where the BCE accelerator is connected. It would enable to use such systems with other tools to automate the generation and optimization of an accelerator.

Automated code generation for controlling the data path in the DFU unit consists of the following steps:

- generate a data-dependence graph from the algorithm
- determine the size of variables and their lifetime in the algorithm
- place variables in the data memories according to the restrictions from the possible data paths in hardware and the required placement (placing of variables in memories is independent of their sizes)
- compute offsets of variables in the memories (memories can be filled with/without the space reusing if the variable is no longer needed)

- generate the source code of the data path micro-controller firmware and two functions for initiation and starting a computing step from the host CPU.

Special Form of the Algorithms in the Matlab Environment

The generation starts from a Matlab script with an algorithm written as a sequence of operations in a special form. The generation tool can be extended with an algorithm which converts the algorithm in a normal form to the required one. But the tool needs some information which is not available in the normal form, e.g. a description of operations and their relations to hardware, the relation between the size of the variables and the algorithm order.

Inputs to the automatic generation are :

- A description of the platform data memory organization. Each data memory is described by its size (all sizes are in units according to the implemented algorithm, e.g. for single precision FP arithmetic it is a 32-bit word)
- A description of the operations. Each operation is described by:
  - a function which represents the behaviour of the operation when it is simulated,
  - a matrix of allowed combinations of transfers between data memories for arguments and results. Generally, operands have a fixed order because some operations are not commutative, therefore all allowed combinations must be in the matrix. In the version described in the thesis this matrix is not used because the BCE platform has been extended with mutliplexers and demultiplexers in data paths, and then the platform supports all combinations for all operations. It makes one of next steps, mapping variables in the data memories, significantly simpler. An example of such a table for operation DIV and the BCE used in the work is in Table 4.6. In the example, combinations where memory Z is a dividend is not allowed.
  - flags which indicate how many operands the function has, if the function has a result, and if the function has a control output which is used to control branching of the program in the firmware.

operand 1	operand 2	result
A	B	Z
B	A	Z
A	Z	B
B	Z	A

Table 4.6: An example of allowed combinations of data memories for an operation.

- A description of variables. Each variable is described by
  - an identifier used in the operations,
  - a type of a variable for generation of correct data vectorization in the memories and the corresponding settings for address counters in the DFU. It can be a scalar; column vector; row vector; diagonal matrix; lower triangular matrix; upper triangular matrix; or full matrix.
  - the length of the basic vector of the variable as a set of coefficients of polynomial functions of N.

    $$l(N) = a_0 + a_1 N + a_2 N^2 + ... + a_n N^n$$

    The size of the variable depends on its type – for all vector-like types it is directly the size of the variable; for a full matrix it is the length raised to the second power; and for triangular matrices it is equal to $\frac{l(l+1)}{2}$.
  - flags which indicate if the variable is a static constant which is set only once in the initialization phase, or if it is an input of the algorithm and must be set from the host CPU in each algorithm cycle, or an output variable which can be read after each cycle by the host CPU.
- A description of the algorithm as a sequence of operations. Each step is described by
  - a used operation from the defined set,
  - variables used as operands and to store results,
  - for each variable the description contains information for controlling the DFU address counters – initial offsets, and relative increments of the element index

in the inner and outer loops,

- an increment of the number of cycles in the inner loop for each cycle in the outer loop,
- a number of cycles in the inner and outer loops.
- a number of skipped operations if it is a control operation. The value is used when the flag in the status word is set.

The algorithm described this way can be used directly to run the simulation of the algorithm or to generate the code.

Generate Dependence Graphs of the Algorithm

The graph is generated as two matrices (an example of such matrices in Table 4.7) from the algorithm. The first matrix T is the translation matrix, and it describes the control flow in the algorithm. It is necessary because algorithms can contain branching with conditional operations. The second matrix V establishes relations between the steps and the variables. Each row in the matrix represents one step in the algorithm and contains positive values in the columns where the variables are used as inputs to the operation; a negative value is used for the output variable. The data-dependence graph is built from these matrices, and they also show the lifetime of the variables.

The dependence graph can be generated from the matrices in these steps:

- each operation in the algorithm (row in the matrices) is a node in the graph,
- for each column where the first row i with a positive value is before the first row j with a negative value ($i < j$) add a node with an input variable which must be initialized outside the algorithm.
- for each variable marked as output add a node.
- for each column add an edge from row i in the matrix (step in the algorithm, node in the graph) with a negative value to row j with a positive value if $j > i$ and any row between them doesn't contain a negative value. It is a simplified method usable when the algorithm doesn't contain any branching.

$$T = \left[\begin{array}{c|ccccc} & \mathbf{s1} & \mathbf{s2} & \mathbf{s3} & \mathbf{s4} & \mathbf{s5} \\ \hline \mathbf{s1} & & 1 & & & \\ \mathbf{s2} & & & 1 & & \\ \mathbf{s3} & & & & 1 & \\ \mathbf{s4} & & & & & 1 \\ \mathbf{s5} & & & & & \end{array}\right]$$

$$V = \left[\begin{array}{c|cccccccccc} & \mathbf{P_1} & \mathbf{x_1} & \mathbf{P_2} & \mathbf{x_2} & \mathbf{P_3} & \mathbf{x_3} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{y} \\ \hline \mathbf{s1} & 1 & 2 & & & & & -1 & & & & \\ \mathbf{s2} & & & 1 & 2 & & & & -1 & & & \\ \mathbf{s3} & & & & & & & 1 & 2 & -1 & & \\ \mathbf{s4} & & & & & 1 & 2 & & & & -1 & \\ \mathbf{s5} & & & & & & & & & 1 & 2 & -1 \end{array}\right]$$

Table 4.7: Example of data-dependence graph in form of matrices. Each row in matrix V corresponds to an operation in the algorithm and columns represents variables.

A graph for EF LD-RLS is in Figure 4.9 and a graph for DF LD-RLS is in Figure 4.10. The matrices are also used to map variables in the data memories as shown in the next part.

Mapping Variables in the Data Memories

The problem with mapping variables in the data memories can be divided in two separate problems on the condition that the sizes of the memories are sufficiently big the towards sizes of variables. The problem can be divided because all the operations in the BCE platform have access to the whole data memories, and we don't use any algorithm to balance the memory usage.

The first part is a placement of variables to a selected memory according to the operations. The second part is a computation of offsets of the variables and it will be discussed in

the next subsection. Algorithms for both parts are universally usable for any number of memories. For better clarity, we will describe the problem for the BCE platform with three data memories (A, B, Z).

We consider that each operation has defined the data memories which can be used with each operation. Memory selection is a mapping problem when one data memory from all possible memories (in our case they are labeled A, B and Z) is assigned to each variable so that all operations which work with the variable can access the selected memory for the required purpose. We consider that the BCE platform has only such operations which cannot use one data memory for more inputs/outputs, in other words, all variables related to the operation are mapped to different data memories.

Because of the modified platform allows to use all operations with all data memories, the mapping problem can be translated to the *vertex coloring*[6] algorithm. It can be solved with heuristic algorithm with the complete graph construction. The graph can be constructed from the data-dependence graph in form of matrices (matrix V in Table 4.7) in a simple way. Variables (columns in matrix V) are vertices and algorithm steps (rows in the matrix) represent edges between variables. We can distinguish between direct and indirect connections. The direct connection is between the input and output variables of one operations. The indirect bond is between two inputs of the same operation. In the current mapping algorithm both types of bonds (direct and indirect) are the same, but we assume that we will use type of bonds in the next version of the mapping algorithm for optimizing the solution. In the current version, the mapping algorithm only tries to construct complete graph by the recurrent merging of variables which are interconnected neither directly nor indirectly.

Algorithm 3 describes the heuristic algorithm used in the current version for mapping.

If dependence graph or other restrictions don't allow to map variables to the data memories, the mapping algorithm informs about error. There isn't implemented any algorithm to solve or improve mapping by manipulations of operations, since it hasn't be the aim of the work. The algorithm will be updated in the future work. The update will add an algorithm for solving simpler impossibilities of mapping by adding copy operation and

[6]Graph coloring, with coloring the vertices of a graph such that no two adjacent vertices share the same color.

Algoritmus 3: Mapping variables into data memories

Input: Data dependence matrix V of the algorithm

Output: Reduced matrix $\bar{V}$ describes a complete graph. If it has more columns than platform has data memories, variables cannot be mapped

$V_m = V$

RECURSION:

n = number of columns of V_m

m = number of rows of V_m

prepare c a vector of zeros (size=n)

`/* evaluate connectivity` c `for all columns,` $c_{(i)}$ `then contains number of connected columns with the i-th column (The connectivity for i-th column is computed as a number of columns which have a non-zero value on one of rows where the i-th column has a non-zero value.) */`

c = evalconnect(V_m)

for $i = 1 : n$ **do**

  if $c_{(i)} = m$ **then**

    $c_{(i)} = -1$ // the column is connected with all other columns

V_m = sortcolumns(V_m,c) // sort columns according to their connectivity $c_{(i)}$ in decreasing sequence

$i = 1$

while $i < n$ **do**

  $a = V_{(:,i)}$ // get the first column with the biggest $c_{(i)}$

  $j = i + 1$

  while $j < n$ **do**

    `/* find the first next column` b `which hasn't a non-zero value on any row from all rows where the` a `has a non-zero value. */`

    $b = V_{(:,j)}$

    if *nolinkbetween(a,b)* **then**

      // merge columns (variables) - output column will contain all non-zero values from both columns

      V = mergecolumn(V,i,j)

      go to RECURSION

    $j = j + 1$

  $i = i + 1$

$\bar{V} = V_m$ // no other possible reduction of the dependence graph

another improving of the mapping algorithm will be in balancing of utilization of the data memories.

Lifetime of Variables - Offsets in Memories

The lifetime of each variable is obtained from the matrix V as an interval between the step when the variable is set for the first time and the step when the variable is used for the last time. After the previous step, mapping variables in the data memories, we have variables sorted to N lists of variables, where N is the number of data memories. For each list, the algorithm tries to place the variable at the first empty place, and it tries to re-use the memory if a variable is no longer used.

In the present version, the greedy algorithm is used (as listed in Algorithm 4). The algorithm needs variables sorted according to their lifetimes and sizes. And it also needs the current offset which is set to the beginning of the memory. Then the first variable with the longest lifetime is placed at the current memory offset. If the variable is not necessary in all the steps, the algorithm tries to add the next variable without overlapping the lifetimes. All placed variables are removed from the list. If no other variable can be added, then the current offset is increased and the algorithm repeats until all variables are placed or the current offset exceeds the size.

Lifetimes of all variables in the EF LD-RLS algorithm are shown in Figure 4.21.

Placement in the data memories generated with Algorithm 4 is shown in Figure 4.22.

A note to the mapping algorithm

The mapping algorithm uses a heuristic algorithm for *vertices coloring* and then more solutions are possible. The implemented EF LD-RLS algorithm is an example, it can be mapped in two ways, when a part of the dependence graph has too few restrictions. The variables in the part can be mapped to more data memories as shown in Figure 4.23. In the figure boxes represent operations and ellipses are variables (letters in brackets are mapped memories). Variables with bigger labels can be mapped into two memories.

Algoritmus 4: Mapping variables inside the memories (computation of memory offsets)

Input: V = list of variables with their properties

sort V according to their lifetimes and sizes respectively
n = size of V
$V_i.offset = 0, \forall i \in 1, ..., n$
$o_g = 0$ (set the global offset to the beginning of memory)
prepare L = an array of zeros (size = the number of time steps)
prepare C an empty list of variables with currently assigned offsets
while V *is not empty* **do**
    $f = V_1$ (get the first variable)
    set $L_j = 1, \forall i \in lifetime(f)$
    $f \rightarrow C$ (add the variable to working list)
    while $L_j \neq 1, \forall j$ ***and*** *all unassigned variables haven't been tested* **do**
        get the next unassigned variable n
        if *the lifetime of* n *is only in the unmarked time steps in* l_w **then**
            add the variable n to the list v_w
            mark the time steps in l_w as used according to the lifetime of n
    assign the global offset o_g to all variables in the working list l_w
    increase o_g with the greatest size of all the variables in l_w

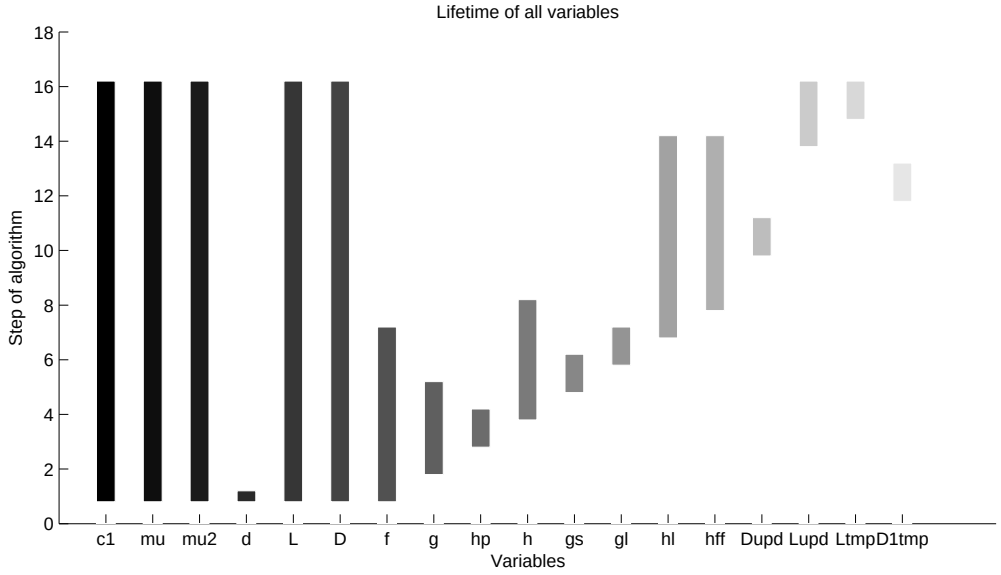


Figure 4.21: Lifetimes of all variables in the EF LD-RLS algorithm

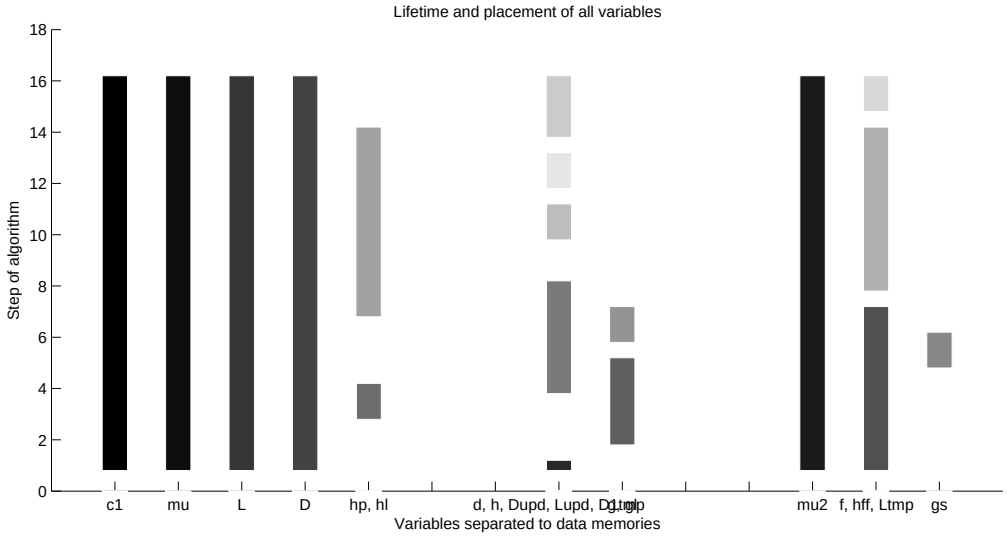


Figure 4.22: Lifetimes and placement of all variables in the EF LD-RLS algorithm. The greedy algorithm was used to place variables. The left, middle and right part of the figure shows variables placed in data memories A,B and Z respectively.

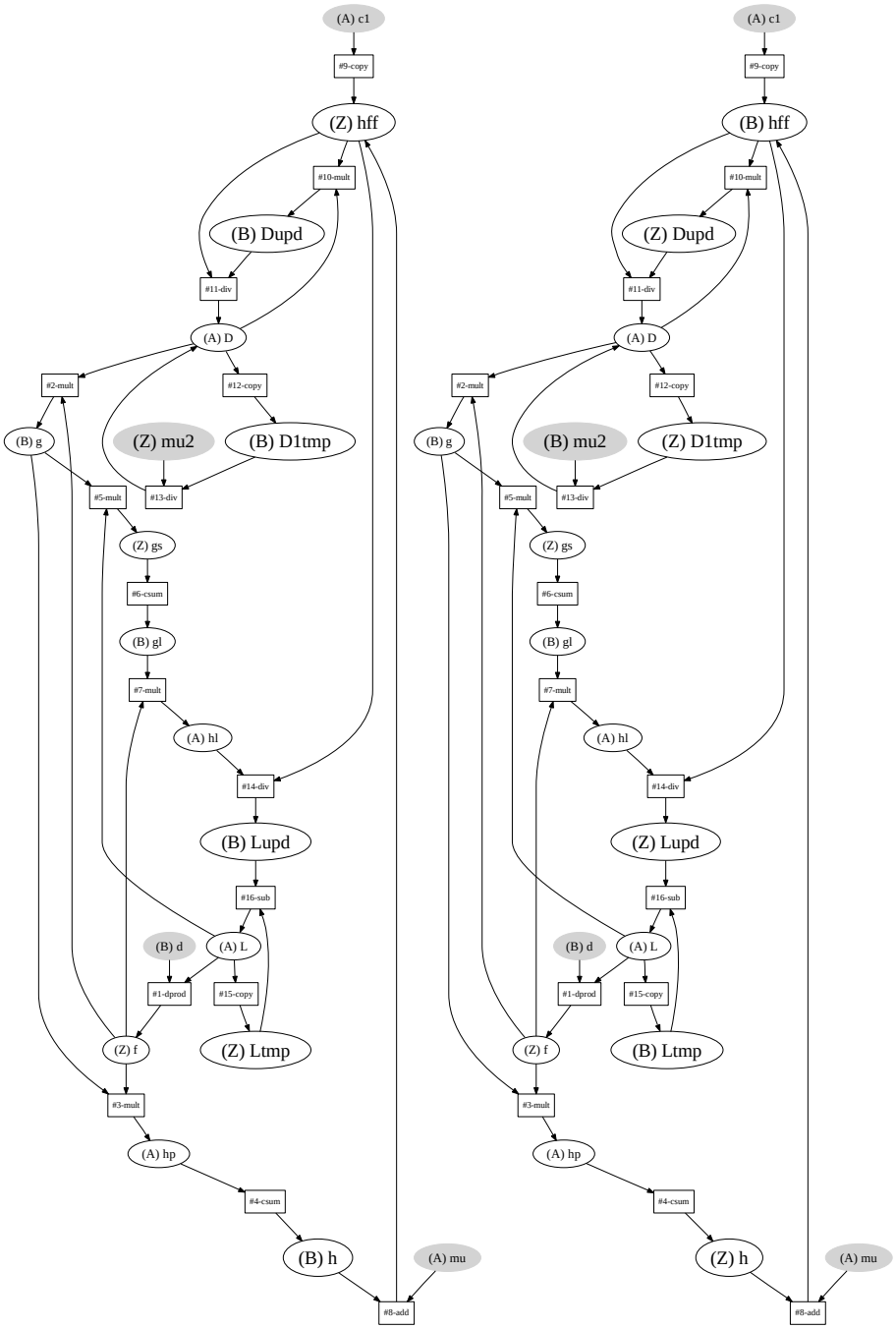


Figure 4.23: Data dependence graph of EF LD-RLS with two possible mapping of variables

The current version of the mapping tool finds the first possible mapping, but it can be worse than another. It can have significant influence on the occupation of the data memories. Figure 4.24 shows occupied data memories on the algorithm order. It depicts how the change of the placement of seven variables reduces occupied space by half. In case of the EF LD-RLS algorithm, the change of placement doesn't increase order of the algorithm because occupied space in all three memories must be lower than size of memory (straight line in the figure), and the second mapping (Figure 4.23) decreases size of used space only in one memory.

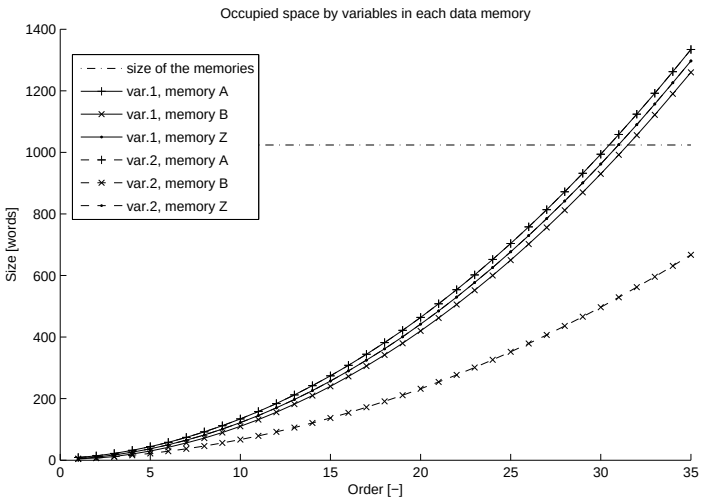


Figure 4.24: Space in the data memories occupied by variables in EF LD-RLS for two possible mappings

This issue is not considered in this version of the mapping tool. The tool only tries map variables with the required order, and it returns an error message if the variables cannot fit in the memories. Then steps to correct the graph can be done by hand.

Generate the Source Code for the Micro-Controller (BCE Firmware)

When the variables are mapped to the platform data memories, the source code of the micro-controller firmware can be generated as described in the next paragraphs.

The source code of the firmware is generated from the algorithm (represented as a sequence of operations with specified variables as operands and results); array of the mapping

variables to data memories and assigned offsets where are all variables placed. The generator produces codes which are sequences of macro-instructions. Each macro-instruction consists of the following parts

- preparing the VLIW instruction,
- waiting for the DONE signal until the previous operation is finished,
- sending the VLIW instruction to the DFU unit which starts the current operation,
- if it is a control operation (e.g. comparison in the DF LD-RLS algorithm) then wait for the DONE signal and use the result of the operation for branching in the program.

This sequence minimizes the total time of the whole algorithm, because the firmware processes these macro-instructions simultaneously with the computation in the DFU unit. Only if an operation is too short (an operation on too short data vectors), the hardware must wait until the micro-controller prepares the next VLIW instruction.

The source code for the used micro-controller PicoBlaze3 (PB) can be compiled to the binary code with the Xilinx tool *KCPSM3* or with the open source tool *picoasm*. The original tools support only compilation to ROM memories as a VHDL code. We use the *picoasm* tool with some modifications. Besides the VHDL code it generates also a C code with the firmware in the form of a static array, and a Matlab function which can be used to simulate the hardware in the Matlab environment.

Generate a Source Code for the Host CPU

The developed tool also automatically generates a source code of two functions for the host CPU which communicates with the BCE platform through the data and control memories. The CPU accesses the accelerator data memories directly, and therefore the procedure in C must be generated from the same array of variables as the firmware.

The first function initiates all variables marked to pre-set. This function should be called from an user application only one when computation is initiated. The example in the following listing is generated from EF LD-RLS algorithm for three estimated parameters.

```
    /* bce_init - initiate algorithm */
int bce_init(wal_worker_t *worker, float *p_c1, float *p_mu,
             float *p_mu2, float *p_L, float *p_D)
{
  int res = WAL_RES_OK;
  res |= wal_mb2dmem(worker, 0, 0, 0, p_c1, 1);    /* initiate variable 'c1' */
  res |= wal_mb2dmem(worker, 0, 0, 1, p_mu, 1);    /* initiate variable 'mu' */
  res |= wal_mb2dmem(worker, 0, 2, 0, p_mu2, 1);   /* initiate variable 'mu2' */
  res |= wal_mb2dmem(worker, 0, 0, 2, p_L, 10);    /* initiate variable 'L' */
  res |= wal_mb2dmem(worker, 0, 0, 12, p_D, 4);   /* initiate variable 'D' */
  return res;
}
```

The second function writes all input variables in the data memories, runs the accelerator, then it waits until the accelerator finishes, and then it reads all output variables from the data memories.

```
    /* bce_cycle - compute one cycle of the algorithm */
int bce_cycle(wal_worker_t *worker, float *p_d, float *p_pars)
{
  int res = WAL_RES_OK;
  res |= wal_mb2dmem(worker, 0, 1, 0, p_d, 4);  /* set input variable 'd' */
  res = wal_start_operation(worker, WAL_PBID_P0);   /* start algorithm in the accelerator */
  if (res!=WAL_RES_OK) return res;
  res |= wal_pb2mb(worker, NULL);                   /* wait for sync end of algorithm  */
  res |= wal_pb2mb(worker, NULL);
  res |= wal_end_operation(worker);
  res |= wal_dmem2mb(worker, 0, 1, 0, p_pars, 3);  /* get parameters */
  return res;
}
```

Then the main part of the user application can be

```
  wal_worker_t *worker;
...
  wal_init_worker(worker);
  bce_init(worker, &c1, &lmb, &lmb2, &L, &D);
...
  while(running) {
    /* measure data  to 'd' */
    bce_cycle(worker, float &d, &pars);
    /* use estimated parameters from 'pars' */
  }
...
```

The generated codes use a special low-level library for communication with the accelerator. The library *Worker Abstraction Layer* (WAL) has been developed to simplify and unify access to hardware accelerators based on UTIA DSP platform from user applications, in other words it offers unified *Application Programming Interface* (API). The library WAL has been developed for European project SMECY and for this thesis.

The library is too close to hardware and therefore it is divided into two parts - common API for using in applications and low level functions specific for each worker. This concept allows to use the same functions in user application for accelerators (based on the concept of the UTIA DSP platform) with various differences, e.g. different size and number of data and control memories, different functions and different format of the VLIW instruction. The common API hides differences between workers and it is very easy to use in applications. Following steps must be done:

1. Add library to compilation process
2. Include header files of the library
3. Define worker structure or use the macro WAL_REGISTER_WORKER
4. Initiate worker
5. Use worker (set worker firmware, run operations, ...)

Testing and Simulation

The generated firmware can be tested and simulated directly in the Matlab/Simulink environment on the initial model used to generate the IP core with the BCE platform. The XSG tool with Simulink provide a bit- and cycle-accurate simulation.

The model must be modified to initialize all memories of the accelerator. The EDK block in the model was replaced with a subsystem (Figure 4.25) with the second instances (Figure 4.26) of dual-ported memories used in the platform. This instances have contents initialized with Matlab functions. The memories with the micro-controller firmware are initiated by function directly generated from the automatic tools. The data memories can

be initiated also from the automatic generator which contains placement of variables in data memories.

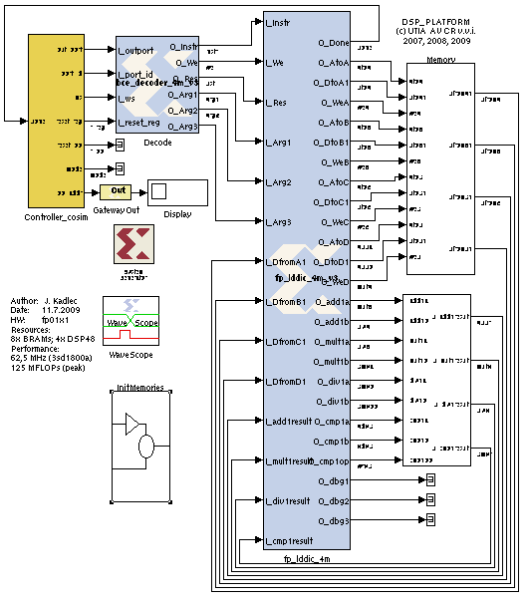


Figure 4.25: Simulink model for testing platform firmware

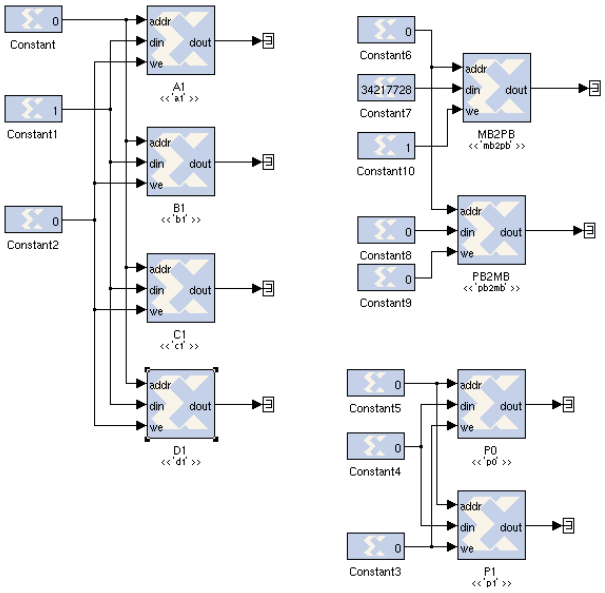


Figure 4.26: Simulink model for testing platform firmware - detail of init memories

Summary of implementation methodology

The implementation methodology for the UTIA DSP platform was described in this chapter. The methodology was improved during the thesis and the improvement is based on an automation of the development process in preparing firmware for the platform and software for the host CPU. The affected step in the development process is highlighted in Figure 4.27 which is schematically redrawn Figure 4.12.

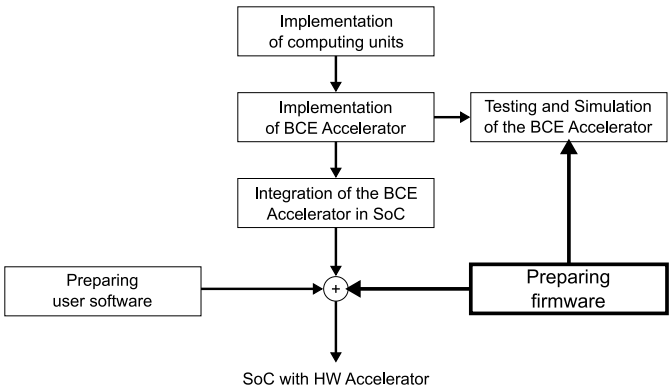


Figure 4.27: Schematic of the implementation methodology for the UTIA DSP platform

The improvement speedup of the implementation development. From experience the implementation of the firmware for EF LD-RLS by hand consumed about 20 hours, when the most of the time took placement of variables in the data memories and coding in the PicoBlaze assembler. The improvement of the development consumed about 30 minutes, when the algorithm in the Matlab environment had to be transformed to a special form.

4.3.9 Use SIMD Mode of the BCE platform for DF LD-RLS

In this section a possible use of the BCE platform in SIMD mode is discussed. The platform can work as a SIMD with more DFUs and computation units, hence in the following part we try to contemplate if the DF LD-RLS algorithm can be modified to use this feature. The basic platform can be prepared with one common control part of the data flow unit and more identical computing parts of the data flow unit, each computation part having its own data memories. The control of all the computing units is provided by one firmware in the common micro-controller (see Figure 4.28).

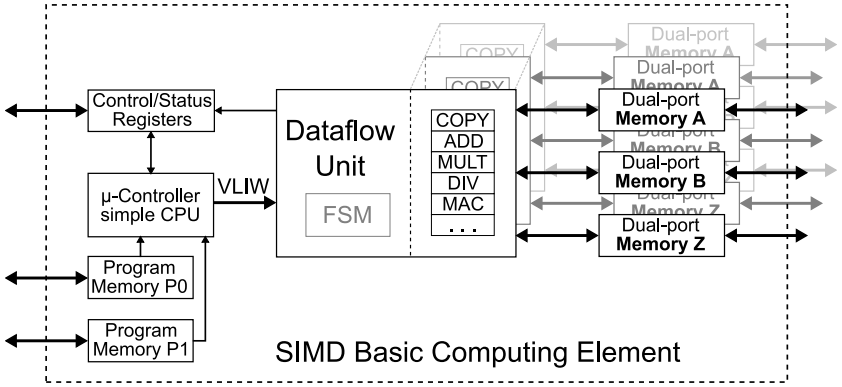


Figure 4.28: BCE in SIMD configuration

It highly increases the data throughput, but this feature cannot be used in all situations. The main condition to use the SIMD mode is when an algorithm is compact. It means such an algorithm which can process uniformly all data channels.

The DF LD-RLS algorithm contains branches that depend on processing the data, therefore the algorithm in the presented form cannot use the SIMD mode for simultaneous computation of independent data channels. Several changes in the algorithm must be done to implement it on SIMD for independent data channels.

The branching should be changed to operations whose results will be saved as new variables in the data memory. The variable should influence all operations in individual branches in the algorithm. It means the algorithm should execute the code in all the branches, and only the results of the valid branches should be used in the common parts.

The following code demonstrates this approach.

common part

$$b_{test} = \begin{cases} 1 & \text{if A less B} \\ 0 & \text{if A} \geq \text{B} \end{cases}$$

branch A

$$\ldots$$
$$C_A = b_{test} \cdot \ldots$$

branch B

$$\ldots$$
$$C_B = (1 - b_{test}) \cdot \ldots$$

common part

$$C = C_A + C_B$$

Such modifications for some algorithms need not be possible or profitable. Generally this method of using SIMD for algorithms with branching is practically usable for algorithms with few operations in branches.

We can try to use SIMD partially, it can be helpful in the case when we need to identify systems with the same inputs and more outputs. But it is usable only for systems which can be modeled with the *Moving Average* models, i.e. systems without dependence on previous output values. For these systems the common part of the L and D matrices is computed in one data flow unit, and then the estimated parameters for all outputs are computed simultaneously in the SIMD mode. In this case the structure of the BCE platform must be modified to a new structure (see Figure 4.29) where the common micro-controller controls one data flow unit (master) by the VLIW instruction. With a special instruction or with an added signal in VLIW instruction it can control all data-flow units together. Such a layout of the accelerator needs an extra shared memory at least. Or one of the data memories of each data flow unit wouldn't be connected to the system bus and the host CPU, but it would be shared with the master unit.

Theoretically, the proposed hybrid SISD/SIMD mode can be used for DF LD-RLS in these cases

- N independent channels with the same number of inputs and outputs (the structure of the identified system doesn't matter). It can be used with a modified algorithm

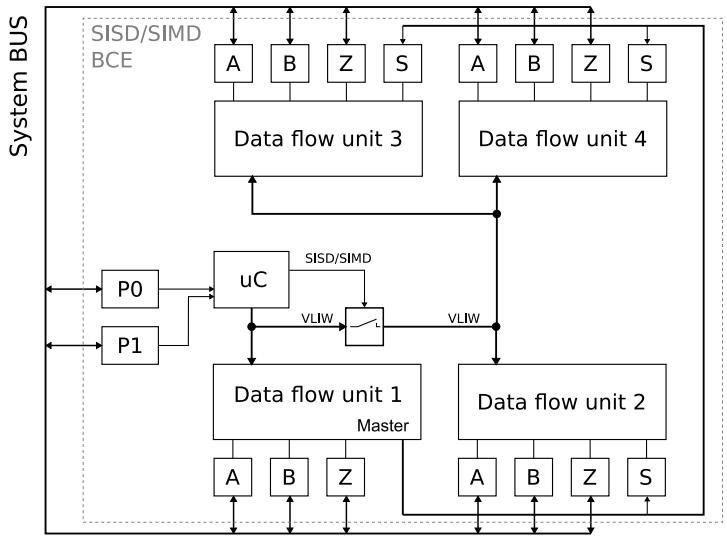


Figure 4.29: Modified BCE for switchable SISD/SIMD mode

with branches transformed to variables. Practically, for DF LD-RLS it is not of much use.

- N independent outputs of the MA models (FIR filters) with common inputs. It can be used with the hybrid SISD/SIMD mode of the BCE platform. In the common part the pre-computation of the ζ and update of the common part of L and D matrices are done, then parameters estimated separately can be computed in parallel. It can rapidly increase the throughput for filters with long data vectors. Practically, this organization is suitable for arrays of sensors.

For the presented reasons, using of SIMD for the DF LD-RLS algorithm is not too practical except in special cases when the parameters of a set of FIR filters with the same inputs are estimated.

4.4 Implementing a Systolic Array with the BCE Platform

This section discusses a conceptual proposal to use the BCE platform to build a systolic array. This concept hasn't been verified in a real hardware or in a simulator, it is only a theoretical contemplation.

The previous works with both architectures instigate us to use BCEs as processing elements in any structure based on systolic arrays. Such a structure can be one of the next steps in the development of the BCE platform.

4.4.1 Structure of a Systolic Array with BCE Accelerators

The concept is based on high configurability of the BCE platform and its preparation in the high level Matlab/Simulink environment. It allows to add more data memories to the DFU unit, and so each BCE platform (as a processing element in the array) can have as many neighbours as it needed (see Figure 4.30).

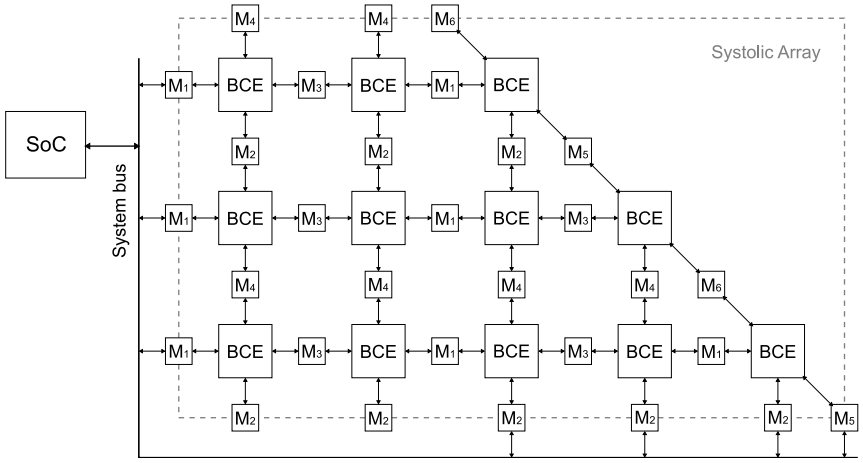


Figure 4.30: An example of the proposed concept of a systolic array with BCEs as Processing Elements

The data memories will be used as data links between the neighbouring PEs, and they will be able to transfer data in both directions. Outer memories which are used for the incoming and outgoing data, will be connected to the memory bus of the host system - system on chip.

A potential problem is the notification when new data sent to neighbours. Some notification flag must be added for each link and each direction.

Another potential problem is the proposed structure of the SA. Systolic arrays are based on fast and short links between PEs, but with a shared dual-port memory as a link between PEs all data can be stored and read with a limited speed - one word per clock cycle. Of

course the PEs can process data as a pipelined vector - the notification flag informs that the first value is written in memory and then the second PE can read the first value while the second is written by the first PE. This method supposes that both PEs run at the same sample rate.

The Logic of the flag can be as follows. Let's assume two PEs, one is a transmitting PE and one is a receiving PE. The transmitting PE has prepared data for the receiving PE. There is also one dual-port data memory between the PEs. The data memory is the common memory for all variables transfered between the PEs, and each variable has its specific position in the memory. Each variable also has a notification flag located in the data memory

At first, the transmitting PE checks the notification flag of the variable to detect if the receiving PE has already processed the previous value of the variable. If the flag is set, the transmitting PE can wait or perform another computation. If the flag is cleared, the variable has been stored in the data memory and the flag set. When the receiving PE reads the data, it will clear the flag (also if it no longer requires the data).

The computation in the current BCEs are started by the flags in the control register which are controlled by the host CPU. BCEs in systolic arrays should be started by the notification flags from their neighbours. In the BCE firmware all these notification flags should be tested to wait for all input variables necessary for the computation.

4.4.2 SoC with a Systolic Array

From the point of view of an entire system, it is impractical to connect each PE directly to the system bus like a BCE unit is normally connected.

The solution can be to use a special module which can be called a *PE concentrator*. It would be able to read the status register and write the control register of all BCEs in a systolic array. It would also upload the firmware to all BCEs. For these purposes it would have a one-way bus to broadcast the firmwares to more PEs at the same time and point-to-point links with each PE to enable firmware loads and to set or read the control/status registers.

The computation at the first BCE is started by the host CPU, by MicroBlaze in the case

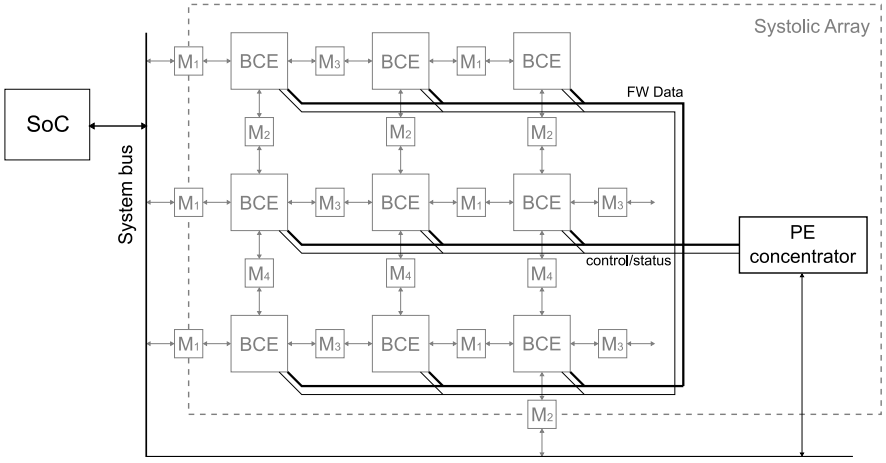


Figure 4.31: An example of the proposed concept of the PE concentrator

of our platform. Individual PEs in the systolic array start to compute when they have all their input data ready. In this concept, the notification flags described above notify that the data have been prepared and so the host CPU or another peripheral must set the notification flags in addition to the input data to start the computation.

4.5 Summary

Two possible implementations of the RLS algorithm based on the LDU decomposition with exponential and directional forgetting were designed and described in this chapter. The first implementation was designed for a triangular-shaped systolic array with $\mathcal{O}(n) = \frac{n(n+1)}{2}$ processing elements. The implementations demonstrate strong data dependences in the DF LD-RLS algorithm.

The main part of the chapter deals with an implementation of algorithms in the *Basic Computing Element* (BCE) accelerator with floating-point pipelined computing units. The platform is a vector accelerator for SoCs in FPGAs. It allows to change its accelerating function by program in the host CPU. Both the implemented algorithms require basic operations which the original platform doesn't provide, therefore the platform has been expanded with these operations.

The operation cumulative summation (CSUM) was analyzed to select the better of two possible implementations of the CSUM function. The original accelerator has also been

expanded with a crossbar switch between the data memories and the computing units. It enables to access all memories from all functions in the accelerator, and so implementations of algorithms are simpler and faster. A support for processing two nested loops for computations with diagonal and triangular matrices has been added. The last modification is the support of algorithms with branches.

The implemented algorithms were reordered to a suitable form for the vector architecture by hand. Then the following implementation steps were done automatically by a newly developed tool. The tool generates firmwares from algorithms written in the Matlab environment. It maps variables of an algorithm to the accelerator data memories, and it also computes the maximal possible order N of the algorithm.

At the end of the chapter possible implementation of the DF LD-RLS on a SIMD architecture has been discussed. The result of the analysis was that such an implementation is useful only for identification of a system which can be modeled by more MA models (FIR filters) with identical inputs. Then their parameters can be identified partially in parallel.

A structure of a systolic array has also been proposed that uses BCEs as processing elements. Its potential implementation has been discussed to resolve problematic issues.

Chapter 5

Results of the Implementation and Experiments

This chapter summarizes some properties of the implemented hardware accelerator. Parameters of the modified BCE platform and LD-RLS algorithms implemented in the UTIA DSP platform will be presented. Then the chapter will continue with several test cases.

5.1 The Modified BCE Platform

One of the comparable parameters for IP cores in an FPGA is the amount of resources required to implement the core in hardware. These values determine the occupied area of the chip and they depend on the configuration of the core and the FPGA device used.

All results presented in this section have been obtained from an implementation in a Xilinx 'Embedded Development HW/SW Kit - Spartan-3A DSP S3D1800A MicroBlaze Processor Edition' with Xilinx FPGA XC3SD1800A. The implementation has been realized with these tools: Mathworks Matlab/Simulink R2008b, Xilinx ISE Design Suite 11.4 (Xilinx CORE Generator, Xilinx System Generator, Xilinx Embedded Development Kit). The embedded system used for testing consisted of the 32-bit CPU MicroBlaze, 32KB SRAM, 128MB DDR SDRAM and peripherals: LEDs, buttons, timer, RS232.

The BCE accelerators have been prepared with the single precision FP computing units

ADD, MULT, DIV and furthermore the modified accelerator also used a CMP unit. Both versions had three data memories, each for 1024 values.

The resources required for the modified accelerator are shown in Table 5.1. The table also contains resources required for the original accelerator. The values are for the whole accelerators as pcore peripherals directly connected to the MicroBlaze SoC. The third column in the table contains values for the MicroBlaze SoC without any accelerator.

Resource	**Classic BCE**	**Modified BCE**	**MicroBlaze SoC**
Occupied Slices	2729	2848	8442
Slice Flip Flops	2381	2483	7119
4 input LUTs	4593	4837	11115
Block RAMs	10	10	34
DSP48	0	0	8

Table 5.1: The resources for the modified BCE accelerator with SP FP units

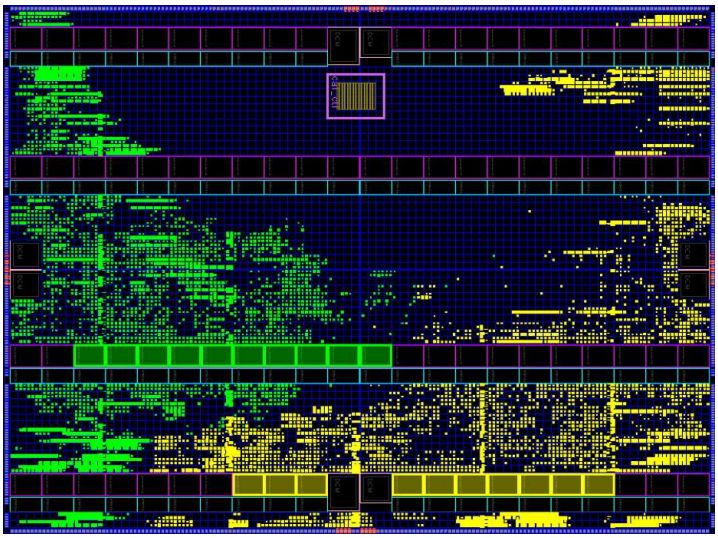


Figure 5.1: Floorplan of a SoC in an FPGA (Xilinx Spartan-3A DSP 1800) with both the BCE accelerators. The original accelerator is in green and the modified accelerator is in yellow

Modifications of the BCE accelerator require about one hundred more slices. It contains

an additional CMP unit, multiplexers to data memories in the data path and additional complex address counters with an extended format of the VLIW word. The maximal clock frequency of the modified BCE accelerator in this implementation was 69.7 MHz. The floorplan of the implemented system-on-chip can be seen in Figure 5.1. The system contains both the BCE accelerators, the original one (in green in the figure) and the modified one (in yellow in the figure).

The DF LD-RLS algorithm is not implemented as a classical stand-alone IP core, but it is implemented as a function of the modified UTIA DSP platform, i.e. it is a firmware for the modified BCE accelerator. Therefore the required resources are not appropriate parameters to describe such an IP core.

Much better and comparable parameters between accelerators based on the BCE platform can be: space required for the firmware; space required for variables in data memories; time to compute one pass of the algorithm; the maximal order of the algorithm (RLS order) with the same space in the data memories. These parameters for the two implementations described above are shown in Table 5.2.

Parameter	**EF LD-RLS**	**DF LD-RLS**
The size of the firmware[instr.words]	395	583
The maximal order of RLS	31	29
The time of one cycle (max.order) [ClC]	7340	9850 / 4875[1]
Occupied space in all three data memories together (max.order) [FP numbers]	2886	2361

Table 5.2: The BCE accelerator with SP FP units

The first parameter, the size of the firmware, is directly proportional to the algorithm complexity. It is an important parameter mainly to check if the algorithm can be implemented in the micro-controller program memory.

The maximal number of estimated parameters is the second parameter. It is given by the distribution of variables in the data memories and, of course, by their sizes which depend on the order of the RLS algorithm. The distribution of the variables in the data

[1]The first number was measured for sufficiently excited inputs when the parameters were updated; the second one was measured when the parameters weren't updated

memories is given by the computed algorithm and by the mapping of the variables in the memories which was done automatically by both parts of the mapping process. The value is relatively low because the BCE platform in the used version has only 3×1024 memory words, and the modification of the platform has been aimed at allowing the computation of the LD-RLS algorithm with minimal changes. The LD-RLS algorithms need to store the triangular matrix L and the diagonal matrix D, which is a vector, and because operations in the platform don't allow to use one memory for inputs and outputs, these two matrices must be duplicated in the memory to be able to update them. From this point of view the storage of the L and D matrices consume one whole memory for the order of the algorithm equal to 31 (matrices L and D occupy exactly 992 words for 30 estimated parameters). If we increase the size of the memories twice, the order can be increased only up to 42 estimated parameters, because the size of the variables L, g_s, g_l, h_l in both algorithm grows with $\frac{n(n-1)}{2}$.

The value in the table was reached by the mapping algorithm which tried to reuse the empty space after releasing other variables. For both algorithms, this strategy increases the maximal order by one parameter against the simple strategy when all variables occupy the whole space in memory all the time.

The number of clock cycles has been taken from a cycle-accurate simulation with real values as described in the chapter about implementation. The time of one cycle, shown in Table 5.2, isn't used for comparison with other implementations because it does not include communication with the host CPU or acquisition of new data. The communication contains mainly waiting for DDR memory where all data in the system were stored. This waiting cannot be predicted especially if a cache memory is used. Therefore all times presented in the next part are measured for storing the input data in the data memories in the accelerator and getting the output data from the data memories. The time of one cycle from Table 5.2 can be used as a teoretical boundary if the communication with CPU and data transfers don't take any time.

5.2 LD-RLS Algorithms on the UTIA DSP Platform

5.2.1 EF LD-RLS Accelerator

Figure 5.2 shows the averages of the measured times of computation one pass of the EF LD-RLS algorithm. The times are measured with transferring data from the CPU to the accelerator before computation and reading the data from the accelerator to the CPU after the computation.

It can be seen that the HW accelerator is faster for a system with six or more parameters. For a system with fewer parameters the algorithm in software is quicker, it is because the accelerator is pipelined. Transferring data from the system memory (DDR SDRAM) to accelerator memories and the initiation of pipelines in operations takes much time than computation in system CPU.

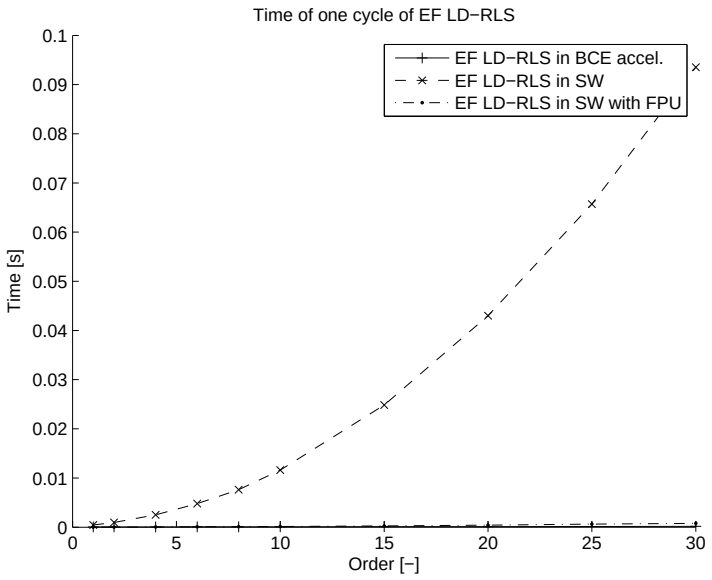


Figure 5.2: The computation time of one pass of the EF LD-RLS algorithm.

Figure 5.5 shows how the performance increases with the growing number of parameters. It's because the utilization of pipelined computing units is higher.

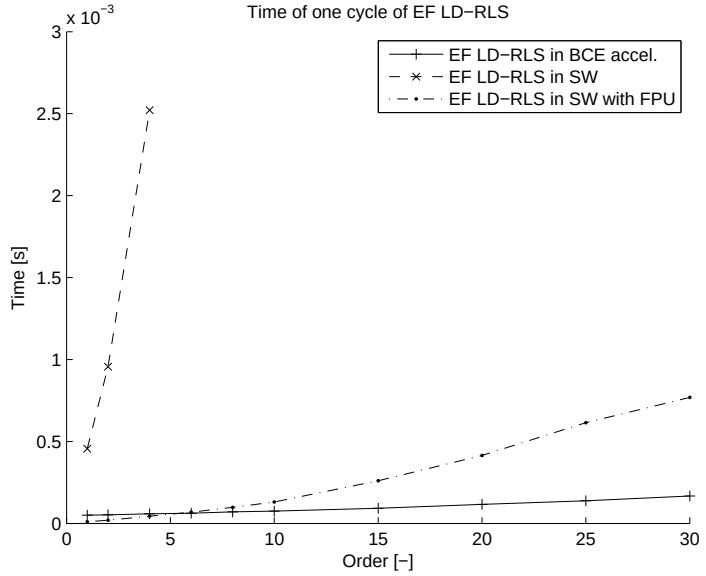


Figure 5.3: The lower segment of the chart in Figure 5.2.

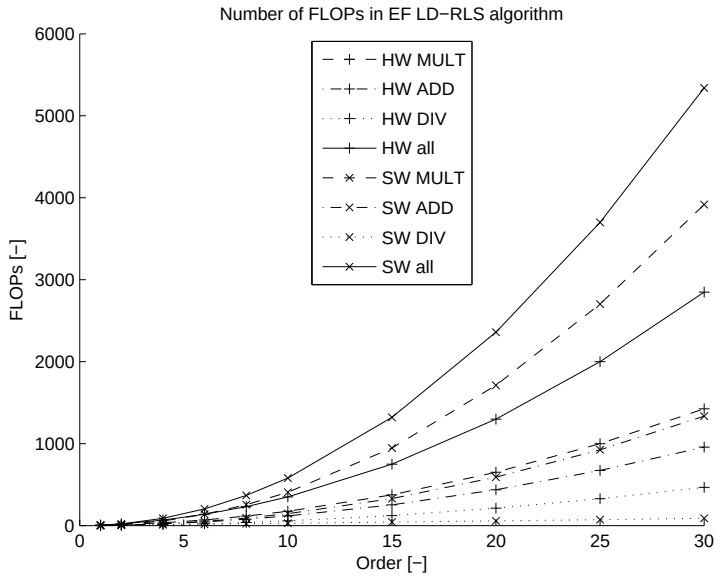


Figure 5.4: The number of FP operations in hardware and software implementation of the EF LD-RLS algorithm in UTIA DSP platform

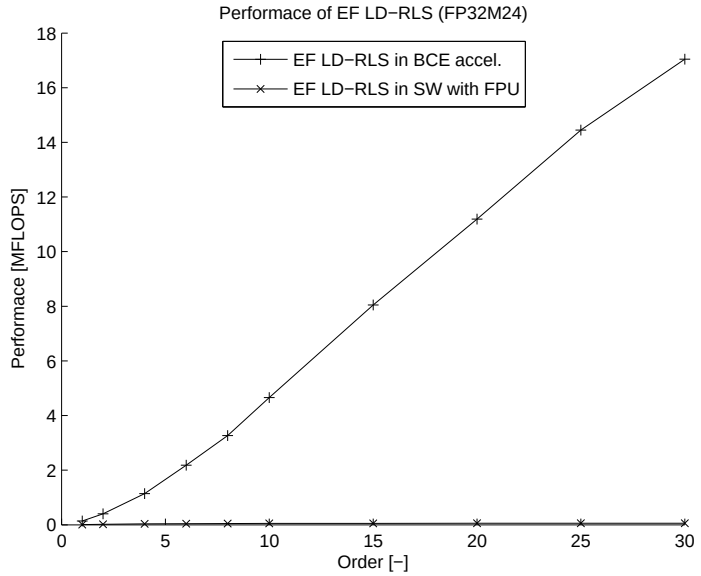


Figure 5.5: Accelerator performance for the EF LD-RLS

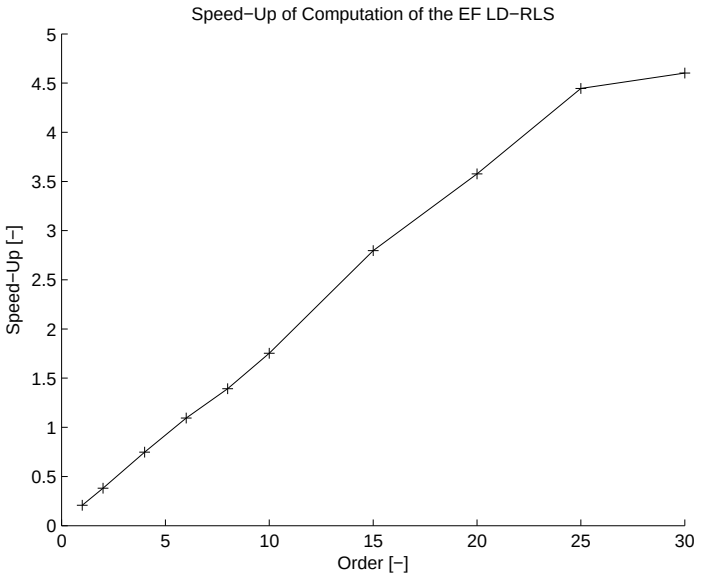


Figure 5.6: Accelerator speed-up for the EF LD-RLS

The last Figure 5.6 shows speed-up of the hardware accelerator against computation in software with FP co processor. For computation in software without FPU the speed-up is about hundred times better than for software with FPU. The speed-up is so low because the algorithm of EF LD-RLS is not suitable for BCE accelerator due to high data dependencies in LD-RLS algorithms.

5.2.2 DF LD-RLS Accelerator

Figure 5.7 shows the averages of the measured times of computation one pass of the DF LD-RLS algorithm. The times are measured with transferring data from the CPU to the accelerator before computation and reading the data from the accelerator to the CPU after the computation. The computation is more complicated, therefore the hardware accelerator is faster from the computation in software from two estimated parameters.

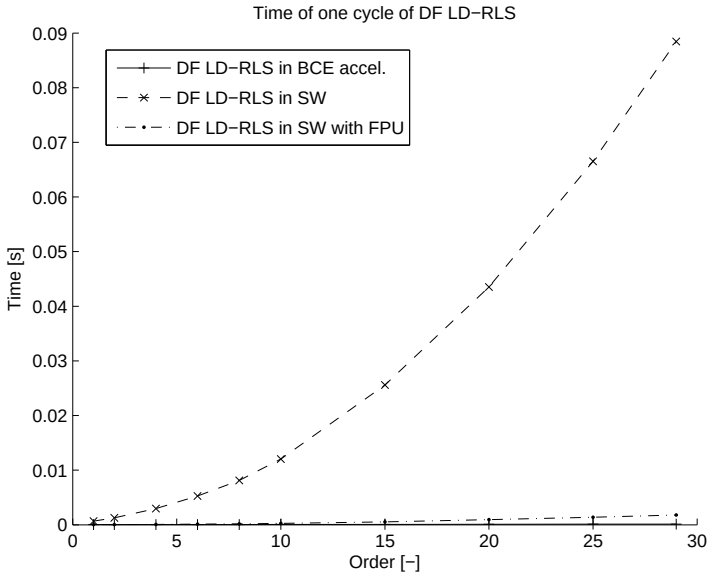


Figure 5.7: The computation time of one pass of the DF LD-RLS algorithm.

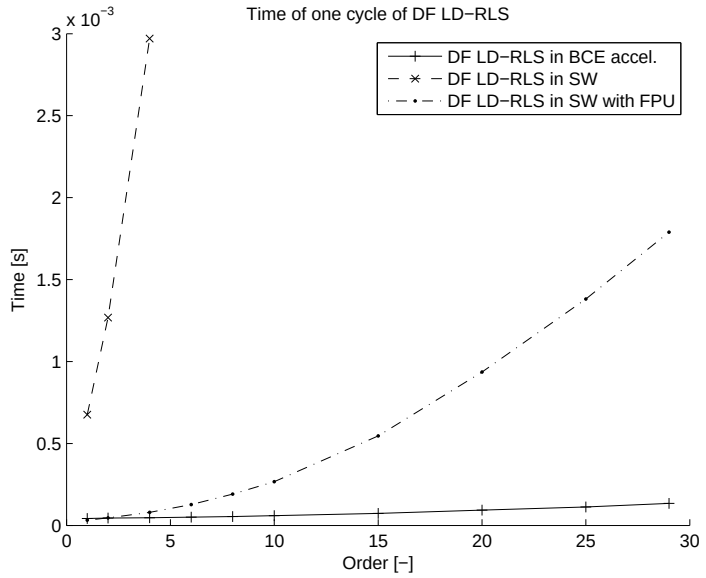


Figure 5.8: The lower segment of the chart in Figure 5.7.

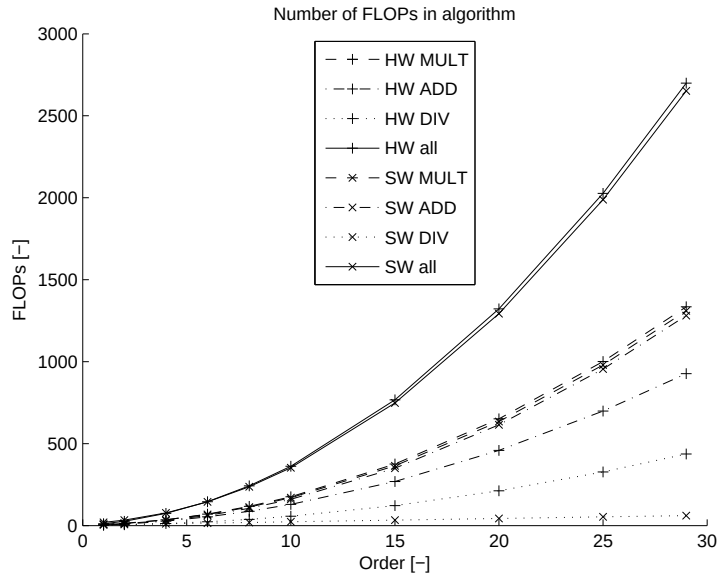


Figure 5.9: The number of FP operations in hardware and software implementation of the DF LD-RLS algorithm in UTIA DSP platform

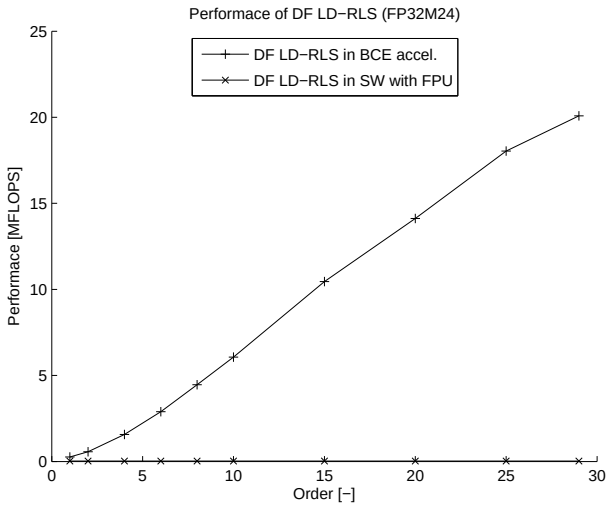


Figure 5.10: Accelerator performance for DF LD-RLS

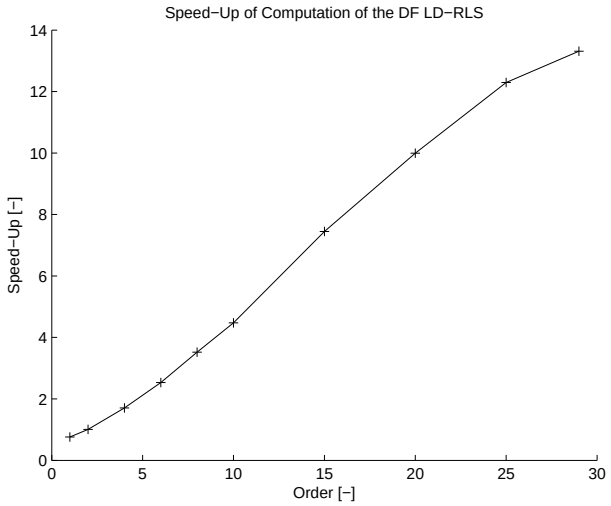


Figure 5.11: Accelerator speed-up for DF LD-RLS

5.3 Test Cases

Both the algorithms EF LD-RLS and DF LD-RLS were tested with several cases of adaptive identification. In this section some of them are presented for illustration

- System identification
- Slow parameter tracking
- Fast parameter tracking

Parameters and input signals for all cases were generated automatically, therefore the charts don't show the main advantage of the directional forgetting which is ability to avoid windup when inputs are not sufficiently excited. It can be seen in the case with fast time-varying parameters.

5.4 Summary

This chapter summarizes some of parameters of the modified BCE accelerator. In comparison with the original BCE accelerator it provides more flexibility and usability. Presented charts with performances of the implementated LD-RLS algorithms show that algorithms with high data dependencies are not suitable for pipelined architectures and their speed-ups are not so significant.

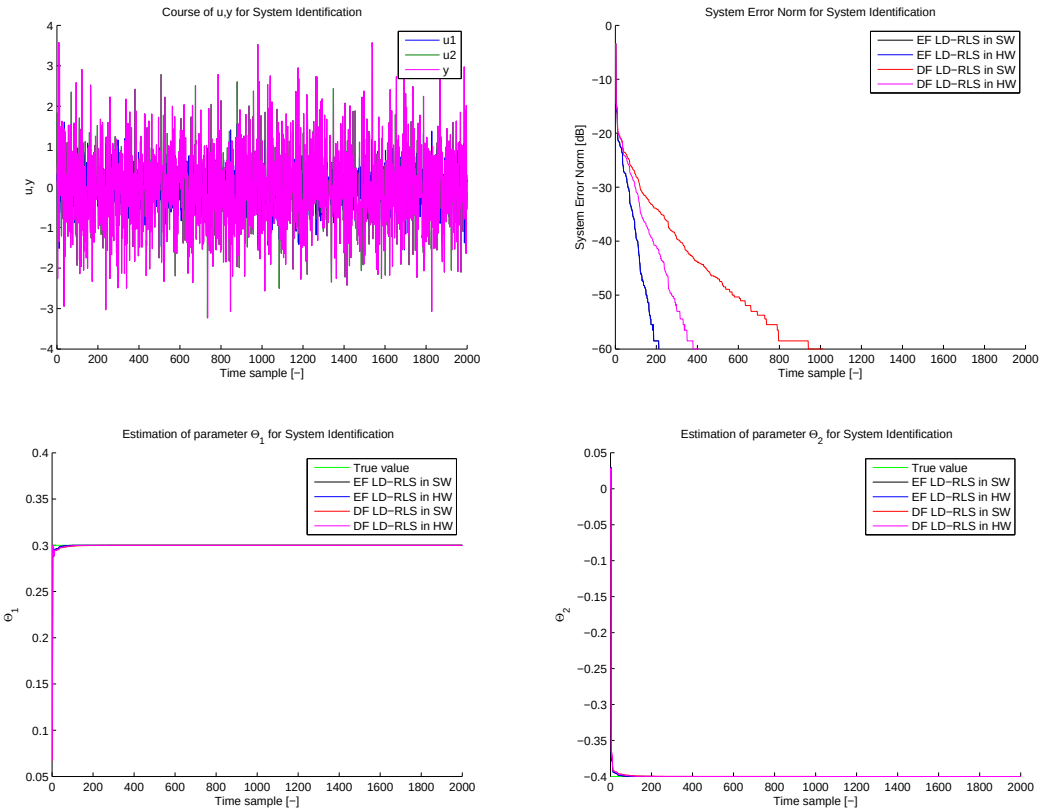


Figure 5.12: Test case: System identification - Evolution of parameters

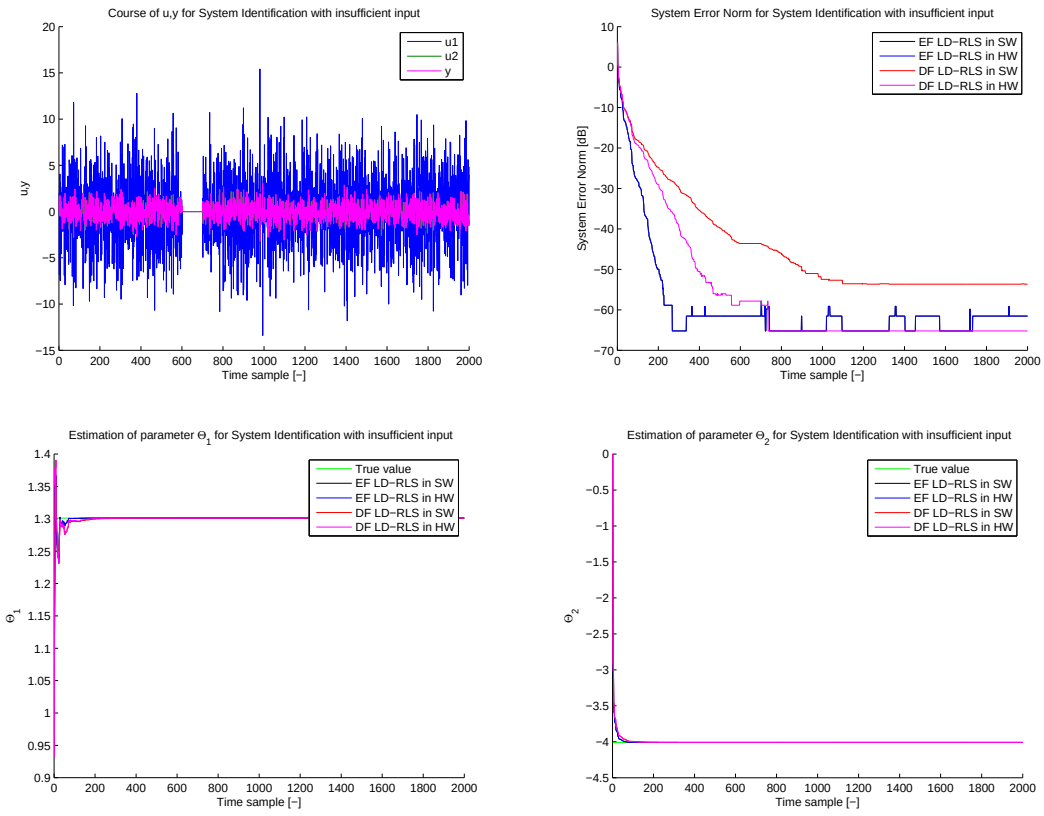


Figure 5.13: Test case: System identification with insufficiently excited inputs

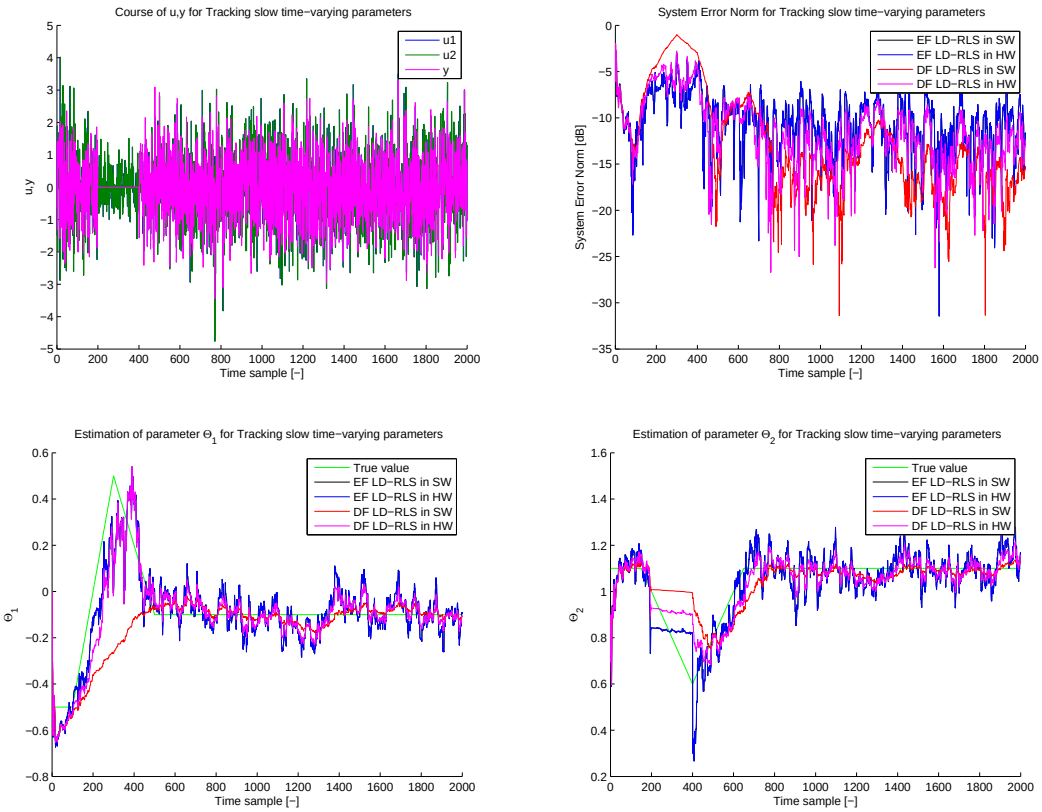


Figure 5.14: Test case: Tracking slow time-varying parameters

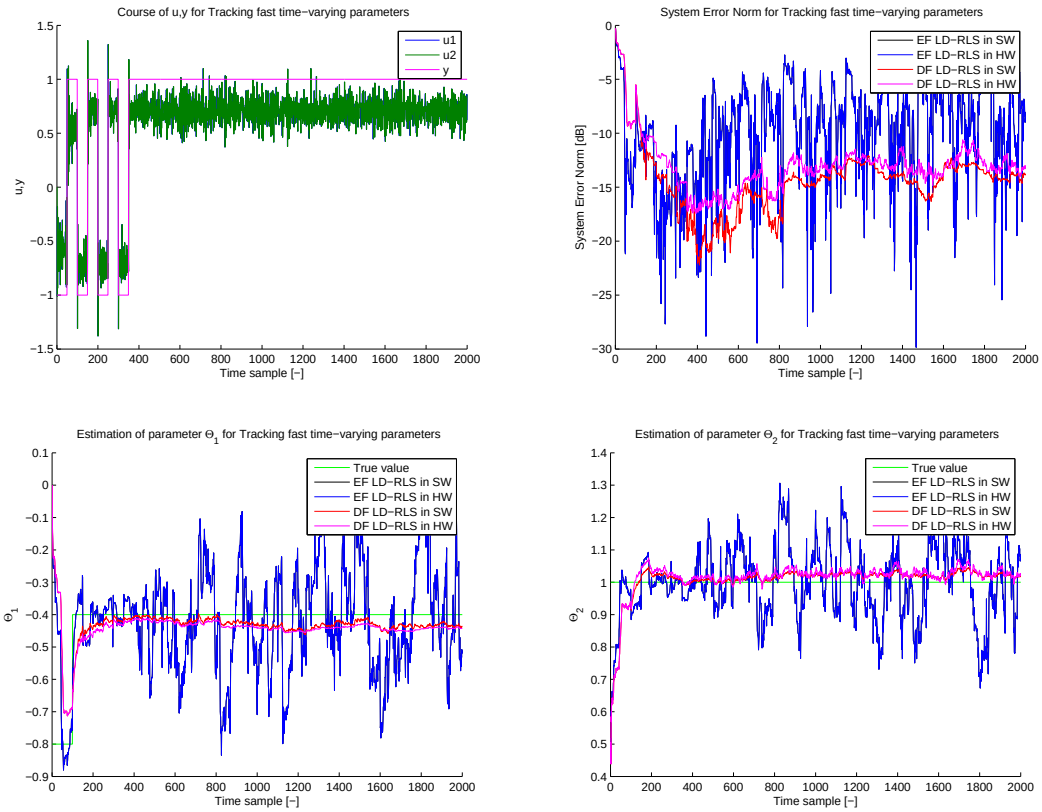


Figure 5.15: Test case: Tracking fast time-varying parameters

Chapter 6

Conclusion

The thesis deals with an implementation of the recursive least squares based on the LDU decomposition (LD-RLS) with directional forgetting.

Today's implementations of adaptive algorithms use mainly the least mean square (LMS) algorithms for their simplicity and low computational complexity which result in high speed and throughput. RLS algorithms aren't so often used, and if so, they use mainly QR decomposition with exponential forgetting for parallel signal processing applications.

The LD-RLS algorithm is interesting for control applications to identify an unknown system or to track time-varying parameters. It is interesting because its solution directly contains the estimated parameters and their uncertainties can be evaluated in a simple way. The next advantage is the possibility to use a priori information about the identified system and its parameters. From the point of view of the implementation this algorithm has an advantage that it doesn't need to compute the roots. The disadvantage of the LD-RLS algorithm is its higher computational complexity than for LMS and more complicated data dependences in the algorithm. The complexity of the LD-RLS with exponential forgetting is $\mathcal{O}(n) = \frac{7}{2}n^2 + \frac{13}{2}n - 3$ FLOP, where n is the number of parameters.

Directional forgetting (DF) was devised 25 years ago, but it is completely omitted in current implementations of RLS algorithms. The idea of the directional forgetting is to discard only part of the old information which is replaced by a new information in new data. From the point of view of the implementation the complexity of the DF LD-RLS

algorithm is $\mathcal{O}(n) = 6n^2 - 2^n$ FLOP.

Two possible implementations of the EF LD-RLS and DF LD-RLS algorithms were designed and described in the thesis. The first implementation was designed for a systolic array with $\mathcal{O}(n) = \frac{n(n+1)}{2}$ processing elements. It demonstrated the high data dependences in the EF LD-RLS and DF LD-RLS algorithms.

The second implementation described in this thesis used concept of the vector-like UTIA DSP platform which has been developed recently. The basic version of the platform is the *Basic Computing Element*(BCE) which is a flexible, re-programmable and reconfigurable hardware accelerator with pipelined floating-point operations for addition, multiplication, division, multiply-accumulate and dot product. The platform had to be modified in hardware for the implementation of the EF LD-RLS and DF LD-RLS algorithms. The platform was expanded with new operations for comparison and cumulative summation. The two possible implementations of the cumulative summation operation were analyzed to choose implementation better for the BCE platform. The platform was also expanded with a crossbar switch between the data memories and the computing units. It enabled to access all memories from all computing units. The support for the second nested loop was also added to the platform. The last, but for the DF algorithm the most important, modification was the support for algorithms with branches.

The implemented algorithms were reordered to a suitable form for the vector architecture by hand. All the following implementation steps were done automatically by a newly developed tool. The tool maps variables of an algorithm to the accelerator data memories. Then it generates firmware for the platform and code for the host CPU from an algorithm written in the Matlab environment. The generated code for the host CPU communicates with the platform through a specially developed library *WAL* which unifies the access to different accelerators based on the UTIA DSP platform. The platform was integrated in a system on a chip with the soft-core processor MicroBlaze in an FPGA. The prepared SoC contains one accelerator based on the modified BCE platform with switchable EF LD-RLS and DF LD-RLS algorithm with the maximal order $N = 31$ or $N = 29$ respectively.

When we studied existing papers about RLS algorithms, we found that many authors compared their new adaptive algorithms or modifications with others by using a uniform value of the forgetting factors, and they didn't analyze which value of the forgetting

factor should be used. A simple method was proposed to ensure that algorithms with different forgetting methods are certainly comparable. The method is based on searching the "optimal" value of the forgetting factor (or "optimal" values of all factors used in the forgetting algorithm) separately for each algorithm.

Objectives Revisited

The dissertation objectives are briefly reviewed and the results reached are presented:

- To summarize theoretical background for implementation of recursive least squares based on the LDU decomposition with directional forgetting.

  Theoretical background for the implementation of the EF LD-RLS and DF LD-RLS has been described in Chapter 2 and Chapter 3.

- To design a structure of LD-RLS with directional forgetting mapped to an architecture based on systolic arrays.

  In the first part of Chapter 4, specifically in Section 4.2, description of proposed structure of a systolic array for EF LD-RLS and DF LD-RLS has been presented.

- To extend the vector-like UTIA DSP platform with functions for implementing LD-RLS with directional forgetting.

  The UTIA DSP platform has been described in the second part of Chapter 4,specifically in Section 4.3. This section contains parts with a description of the performed modifications.

- To implement effectively LD-RLS with directional forgetting on the vector-like UTIA DSP platform.

  The implementation has been described in Section 4.3 and the results of the implementation have been presented in Chapter 5.

- To improve the implementation methodology for the UTIA DSP platform.

  The improved implementation methodology for the UTIA DSP platform has been described in Section 4.3.

- To develop tools for the improved methodology.

  Tools have been developed as scripts and functions for the Matlab environment. The concept of their behaviour has been described in Chapter 4 Sub-section 4.3.8.

- To discuss the possibility to use the UTIA DSP platform in a Network on Chip.

  The possibility of using UTIA DSP platform in a Network on Chip has been discussed in the Chapter 4 Section 4.4.

Summary of Contributions

The author's contribution includes

- the extension of the basic UTIA DSP Platform with the operation cumulative summation, crossbar switch in the data path, the support of the second nested loop and the support of algorithms with branches,
- the implementation of the vector form of EF LD-RLS and DF LD-RLS algorithms in the extended UTIA DSP Platform,
- the improvement of the implementation methodology for the UTIA DSP Platform,
- the development and implementation of the automatic generator of a firmware for the UTIA DSP Platform from an algorithm in the Matlab envorinment,
- the development of the host CPU software library for a unified access to the UTIA DSP Platform,
- the suggestion of the generic method to compare different adaptive algorithms with different forgetting factors.

Bibliography

Baleani, M., Gennari, F., Jiang, Y., Patel, Y., Brayton, R. K. and Sangiovanni-Vincentelli, A. (2002), Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform, *in* 'CODES'02: Proceedings of the tenth international symposium on Hardware/software codesign', ACM, New York, NY, USA, pp. 151–156.

Barr, M. (1998), *Programming Embedded Systems in C and C++*, O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN 1565923545.

Bittanti, S., Bolzern, P. and Campi, M. (1990*a*), 'Convergence and exponential convergence of identification algorithms with directional forgetting factor', *Automatica* **26**(5), 929 – 932.

Bittanti, S., Bolzern, P. and Campi, M. (1990*b*), 'A counter example to the exponential convergence of the directional forgetting algorithm', *International Journal of Adaptive Control and Signal Processing* **4**(3), 237–244.

Björck, A. (1996), *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia.

Bobál, V., Bohm, J., Prokop, R. and Fessl, J. (1999), *Praktické aspekty samočinně se nastavujících regulátorů:algoritmy a implementace*, VUTIUM, Brno.

Campi, M. (1994), 'Performance of rls identification algorithms with forgetting factor: A ϕ-mixing approach', *J. Math. Systems, Estimation and Control* **7**(3), 1–25.

Cao, L. and H., S. (2000), 'A directional forgetting algorithm based on the decomposition of the information matrix', *Automatica* **36**(11), 1725 – 1731.

Chisci, L. and Mosca, E. (1987), 'Parallel architectures for rls with directional forgetting', *International Journal of Adaptive Control and Signal Processing* **1**(1), 69 – 88.

Chokshi, R., Berezowski, K. S., Shrivastava, A. and Piestrak, S. J. (2009), Exploiting residue number system for power-efficient digital signal processing in embedded processors, *in* 'CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems', ACM, New York, NY, USA, pp. 19–28.

Daněk, M., Kadlec, J., Bartosinski, R. and Kohout, L. (2008), Increasing the level of abstraction in fpga-based designes, *in* 'Proceedings 2008 International Conference on Field Programmable Logic and Applications'.

Diniz, P. S. R. (2007), *Adaptive Filtering: Algorithms and Practical Implementation*, Springer-Verlag New York, Inc., Secaucus, NJ, USA. ISBN 0387312749.

Dogançay, K. (2008), *Partial-Update Adaptive Signal Processing*, Academic Press, Oxford. ISBN 978-0-12-374196-7.

Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L. and White, A. (2003), *Sourcebook of parallel computing*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-871-0.

Flynn, M. (1966), 'Very high-speed computing systems', *Proceedings of the IEEE* **54**(12), 1901 – 1909.

Glesner, S., Geiß, R. and Boesler, B. (2002), 'Verified code generation for embedded systems', *Electronic Notes in Theoretical Computer Science* **65**(2), 19 – 36. COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002).

Golub, G. H. and Van Loan, C. F. (1996), *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*, 3rd edn, The Johns Hopkins University Press. ISBN 0801854148.

Gunnarsson, S. (1996), Combining tracking and regularization in recursive least squares identification, *in* 'Proceedings of the 35th Conference on Decision and', Vol. 3, pp. 2551–2552.

Hagglund, T. (1983), New estimation techniques for adaptive control, PhD thesis, Lund Institute of Technology, Lund, Sweden.

Hanselmann, H. (1987), 'Implementation of digital controllers—a survey', *Automatica* **23**(1), 7–32.

Haykin, S. (1996), *Adaptive filter theory (3rd ed.)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-322760-X.

Heřmánek, A. (2005), Study of the next generation equalization algorithms and their implementation, PhD thesis, Université Paris XI, UFR Scientifique d'Orsay.

Jiang, J. and Zhang, Y. (2004), 'A revisit to block and recursive least squares for parameter estimation', *Computers and Electrical Engineering* **30**(5), 403 – 416.

Kadlec, J. (1986), Probabilistic Identification of Regression Model in a Fixed-Point Arithmetic, PhD thesis, Institute of Information Theory and Automation of the Academy of Sciences of the Czech Republic, Prague, Czechoslovakia.

Kadlec, J., Gaston, F. and Irwin, G. (1992), Parallel implementation of restricted parameter tracking, *in* 'Third IMA International Conference on Mathematics in Signal Processing'.

Kadlec, J., Gaston, F. M. F. and Irwin, G. W. (1995), 'The block regularised parameter estimator and its parallelisation', *Automatica* **31**(8), 1125 – 1136.

Kadlec, J., Gaston, F. M. F. and Irwin, G. W. (1997), 'A parallel fixed-point predictive controller', *International Journal of Adaptive Control and Signal Processing* **11**(5), 415–430.

Kadlec, J., R., B. and Daněk, M. (2007), Accelerating microblaze floating point operations, *in* 'Proceedings 2007 International Conference on Field Programmable Logic and Applications (FPL)'.

Kulhavý, R. (1983), Směrové zapomínání a průběžná identifikace systémů s pomalu se měnícími parametry, Technical report, Ústav teorie informace a automatizace ČSAV, Praha, CZ.

Kulhavý, R. (1987), 'Restricted exponential forgetting in real-time identification', *Automatica* **23**(5), 589 – 600.

Kulhavý, R. and Kárný, M. (1984), Tracking of slowly varying parametrs by directional forgetting, *in* 'Proceedings 9th IFAC World Congress', Vol. 10, pp. 78–83.

Kulhavý, R. and Zarrop, M. B. (1993), 'On a general concept of forgetting', *International Journal of Control* **58**(4), 905–924.

Kuneš, M., Heřmánek, A. and Tichý, M. (2009), Reducing power measurements of utia dsp platform by cloack-gating technique, report on experimental results, Technical report, Ústav teorie informace a automatizace AV ČR, v. v. i.

Kung, H. (1982), 'Why systolic architectures?', *Computer* **15**(1), 37–46.

Labrecque, M., Yiannacouras, P. and Steffan, J. G. (2007), 'Custom code generation for soft processors', *SIGARCH Comput. Archit. News* **35**(3), 9–19.

Lindoff, B. (1997), 'On the optimal choice of the forgetting factor in the recursive least squares estimator'.

Lindström, A., Nordseth, M. and Bengtsson, L. (2003), Vhdl library of nonstandard arithmetic units, Technical report, Chalmers University of Technology.

Ljung, L. (1998), *System Identification: Theory for the User (2nd Edition)*, Prentice Hall PTR. ISBN 0136566952.

Ljung, L. and Gunnarsson, S. (1990), 'Adaptation and tracking in system identification—a survey', *Automatica* **26**(1), 7–21.

Matousek, R., Tichý, M., Pohl, Z., Kadlec, J., Softley, C. and Coleman, N. (2002), Logarithmic number system and floating-point arithmetics on fpga, *in* 'FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications', Springer-Verlag, London, UK, pp. 627–636.

Moonen, M. (1993), A systolic array for recursive least squares computations - part ii: Mapping directionally weighted rls on an svd updating array, Technical report, ESAT - Katholieke Universiteit Leuven.

Moonen, M. (1995), 'Systolic algorithms for adaptive signal processing', *Institute for Mathematics and Its Applications* **69**, 125–138.

Navrátil, P. and Bobál, V. (2005), Adaptivní řízení systému tří nádrží v prostředí matlab simulink, *in* 'Technical Computing Prague 2005 13th Annual Conference Proceedings', VŠCHT, Praha.

Niemann, R. (1998), *Hardware/Software CO-Design for Data Flow Dominated Embedded Systems*, Kluwer Academic Publishers, Norwell, MA, USA. ISBN 0792382994.

Omondi, A. and Premkumar, B. (2007), *Residue Number Systems: Theory and Implementation*, Imperial College Press, London, UK, UK. ISBN 1860948669, 9781860948664.

Parkum, J., Poulsen, N. K. and Holst, J. (1990), Selective forgetting in adaptive procedures, *in* 'The 11th IFAC World Congress in Tallinn', Vol. 3, pp. 180–185.

Parkum, J., Poulsen, N. K. and Holst, J. (1992), 'Recursive forgetting algorithms', *International Journal of Control* **55**(1), 109–128.

Peterka, V. (1981), Bayesian approach to system identification, *in* 'Trends and Progress in System Identification, P. Eykhoff, Ed', Pergamon Press, pp. 239–304.

Peterka, V. e. a. (1982), Algoritmy pro adaptivní mikroprocesorovou regulaci technologických procesů, Technical report, Ústav teorie informace a automatizace ČSAV, Praha, CZ.

Pohl, Z. (2008), Adaptive Order Linear Predictor for Speech Coding Algorithms, PhD thesis, FEL ČVUT.

Proakis, J. G., Nikias, C. L., Rader, C. M., Ling, F., Moonen, M. and Proudler, I. K. (2001), *Algorithms for Statistical Signal Processing*, Prentice Hall PTR, Upper Saddle River, NJ, USA. ISBN 0130622192.

Ramanujam, J., Hong, J., Kandemir, M. and Narayan, A. (2001), Reducing memory requirements of nested loops for embedded systems, *in* 'DAC '01: Proceedings of the 38th annual Design Automation Conference', ACM, New York, NY, USA, pp. 359–364.

Rangarao, K. V. and Mallik, R. K. (2006), *Digital Signal Processing: A Practitioner's Approach*, Wiley Publishing. ISBN 0470017694.

Schier, J. (1994), Parallel Algorithms for Robust Adaptive Identification and Square-Root LQG Control Synthesis, PhD thesis, FJFI ČVUT.

Silvano et al, C. (2010), 2parma: Parallel paradigms and run-time management techniques for many-core architectures, *in* '2010 IEEE Annual Symposium on VLSI'.

Sjö, A. (1992), 'Updating techniques in recursive least-squares estimation'.

Stenlund, B. and Gustafsson, F. (2002), Avoiding windup in recursive parameter estimation, *in* 'Preprints of reglermöte'.

Svoboda, A. (1957), Rational numerical system of residual classes, *in* 'Stroje na zpracovani informaci', Vol. V, pp. 1–29.

Tichý, M. (2006), Fast Adaptive Filtering Algorithms and their Implementation using Reconfigurable Hardware and Log Arithmetic, PhD thesis, FEL ČVUT.

Tokhi, M. O., Hossain, M. A. and Shaheed, M. H. (2003), *Parallel computing for real-time signal processing and control*, Springer. ISBN 1852335998.

Yazdi, H. S., Yazdi, M. S. and Mohammadi, M. R. (2009), 'A novel forgetting factor recursive least square algorithm applied to the human motion analysis', *International Journal of Applied Mathematics and Computer Sciences* **5**(2), 128–135.

Yokoyama, Y., Minseok, K. and Arai, H. (2006), Implementation of systolic rls adaptive array using fpga and its performance evaluation, *in* 'Vehicular Technology Conference, 2006. VTC-2006 Fall'.

Zhao, Y. and Malik, S. (1999), Exact memory size estimation for array computations without loop unrolling, *in* 'DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference', ACM, New York, NY, USA, pp. 811–816.

Zheng, Y. and Lin, Z. (2003), 'Recursive adaptive algorithms for fast and rapidly time-varying systems', *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on* **50**(9), 602 – 614.

Appendix A

The derivation of direct update of L and D

The appendix contains derivation of the direct update of matrices L and D in the LD-RLS algorithms. The derivation can be found in (Peterka, 1982).

Matrices $L(k)$ and $D(k)$ can be updated directly instead of the information matrix $V(k)$ update. Its derivation will be shown for the exponential forgetting in the following text.

The LD factorization of matrix $V(k)$ has been introduced in Equation 3.43. The update of matrices $L(k-1)$ and $D(k-1)$ is taken from Equation 3.28 and is

$$L(k)D(k)L^T(k) = \left[\lambda \left(L(k-1)D(k-1)L^T(k-1)\right)^{-1} + d(k)d^T(k)\right]^{-1}. \tag{A.1}$$

It can be rewritten to the form

$$L(k)D(k)L^T(k) = \frac{1}{\lambda}L(k-1)ML^T(k-1), \tag{A.2}$$

where

$$M = \left[D^{-1}(k-1) + f\frac{1}{\lambda}f^T\right]^{-1}, \tag{A.3}$$

and

$$f = L^T(k-1)d(k). \tag{A.4}$$

Because matrix M is positive definitive, its factorization must exist and it is

$$M = H\bar{D}H^T, \tag{A.5}$$

where H is a lower triangular matrix with ones on the diagonal and $\bar{D}$ is a diagonal matrix. We need find the factorization of Equation A.5. Then the update is

$$\begin{aligned} L(k) &= L(k-1)H \\ D(k) &= \tfrac{1}{\lambda}\bar{D} \end{aligned} \quad . \tag{A.6}$$

Here we show only results of the derivation. Full derivation is shown in (Peterka, 1982).

The update of the diagonal matrix D is

$$D_{ii}(k) = D_{ii}(k-1)\frac{\sigma^2_{(i+1)}}{\lambda\sigma^2_{(i)}}, \tag{A.7}$$

where we use

$$\sigma_i^2 = 1 + \sum_i^n D_{ii} f_i^2. \tag{A.8}$$

The update of the lower triangular matrix L is

$$L_{ij}(k) = L_{ij}(k-1) - \frac{f_j g_i^{(j+1)}}{\sigma^2_{j+1}}, \tag{A.9}$$

where g_i is

$$g_i = \sum_{l=1}^{i-1} D_{ll} L_{il} f_l + D_{ii} f_i. \tag{A.10}$$

The Equations A.4 and A.7-A.10 describes the update of the EF LD-RLS algorithm.

Appendix B

DF LD-RLS unfolded data-flow graph

This appendix shows unfolded data-flow graph of DF LD-RLS for 3 parameters. This graph shows impossibility to parallel algorithm due to data dependence of directional forgetting. The highlighted way is a cumulative summation of $h_2 = \zeta = \varphi(t)^T \mathbf{C}(t-1)\varphi(t)$.

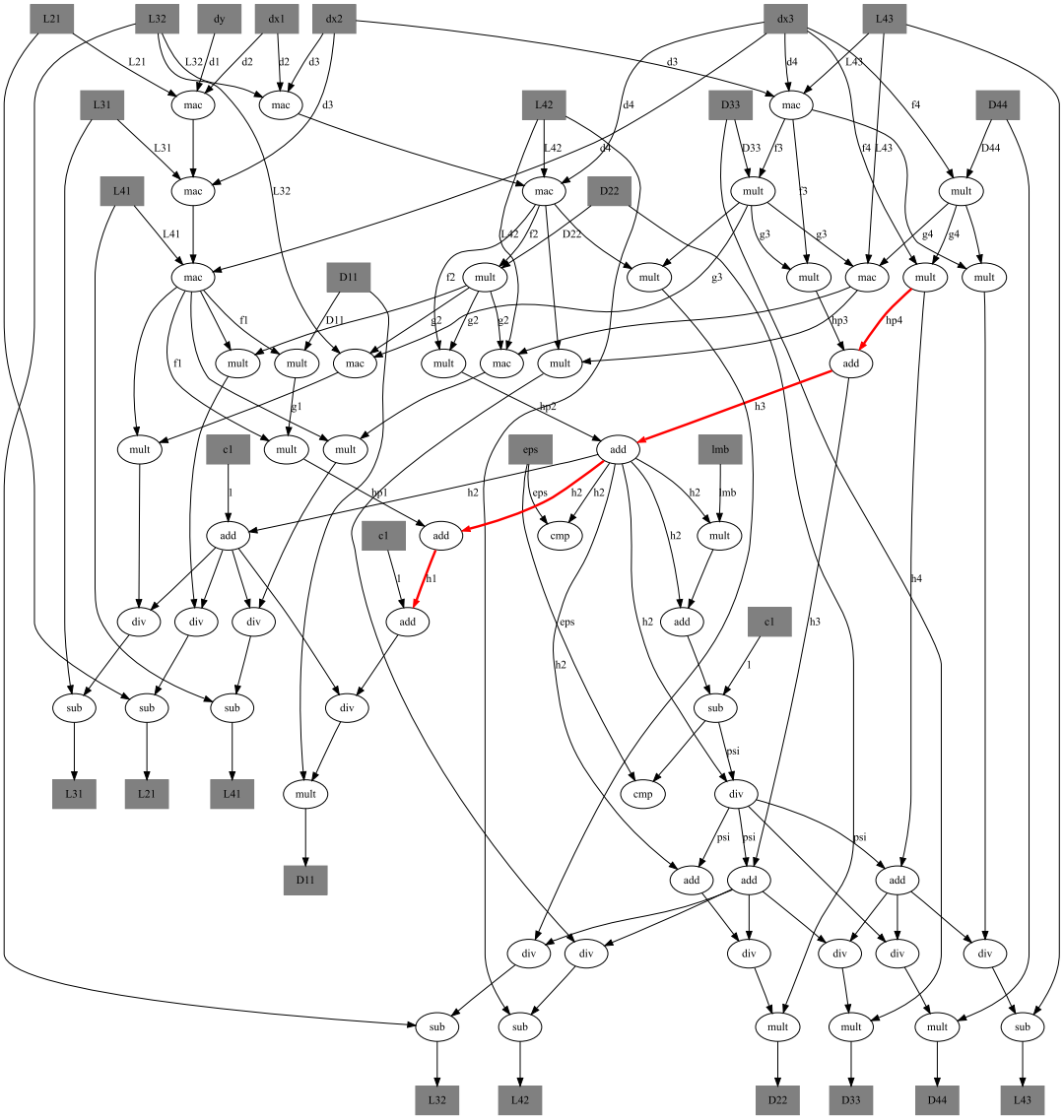


Figure B.1: Unfolded Data-flow graph of DG LD-RLS

List of Author's Publications

The author percentage share in all publications is equal to $100\%/n_a$, where n_a is the number of authors.

Journal Publications

[A1] R. Matoušek, M. Daněk, Z. Pohl, **R. Bartosinski**, and P. Honzík. Reconfigurable System-on-a-Chip. *Syndicated*, 5(2):1–3, 2005.

[A2] **R. Bartosinski**, P. Píša. Jednotka pro řízení pohybu s FPGA a operačním systémem RT Linux. *Automa*, 11(5):46–49, 2005.

Conference Publications

[B1] **R.Bartosinski**, P.Píša. Universal Motion Controller Platform with Real-Time Linux and FPGA. In *Proceedings of the 6th International Scientific-Technical Conference on Process Control (Říp 2004)*, p.266, Pardubice, 2004. University of Pardubice.

[B2] **R.Bartosinski**, M.Daněk, P.Honzík, R.Matoušek. Dynamic reconfiguration in FPGA-based SoC designs. *FPGA 2005 - ACM/SIGDA Thirteenth ACM International Symposium on Field-Programmable Gate Arrays*, p.274, Monterey, 2005. ACM.

[B3] **R.Bartosinski**, M.Daněk, P.Honzík, R.Matoušek. Dynamic reconfiguration in FPGA-based SoC designs. *DDECS 2005 - Proceedings of the 8th IEEE Workshop on Designs and Diagnostics of Electronic Circuits nad Systems*, pp 129-136, Sopron, 2005. University of West Hungary.

[B4] **R.Bartosinski**, M.Daněk, P.Honzík, R.Matoušek. Dynamic reconfiguration in FPGA-based SoC designs. *ACACES 2005. Advanced Computer Architecture and Compilation for Embedded Systems*, p.35-38, Ghent, 2005. HiPEAC Network of Excellence.

[B5] **R.Bartosinski**, P.Stružka, L. Waszniowski. PEERT-blockset for processor expert andmatlab/simulink integration. *Technical Computing Prague 2005 13th Annual Conference Proceedings*, p.1-8, Praha, 2005. VŠCHT.

[B6] **R.Bartosinski**, Z.Hanzálek, P.Stružka, L.Waszniowski. Processor Expert Enhances Matlab Simulink Facilities for Embedded Software Rapid Development *ETFA 2006 Proceedings*, p.625-628, Piscataway, 2006. IEEE.

[B7] **R.Bartosinski**, J.Kadlec. Hardware co-simulation with communication server from MATLAB/Simulink. In *Technical computing Prague 2006. 14th annual conference proceedings*. p. 13-20. 2006, Prague.

[B8] **R.Bartosinski**, Z.Hanzálek, P.Stružka, L.Waszniowski. Integrated Environment for Embedded Control Systems Design. *Proceedings 21st International Parallel and Distributed Processing Symposium*. p.147, Piscataway, 2007. IEEE.

[B9] J.Kadlec, **R.Bartosinski**, M.Daněk. Accelerating MicroBlaze Floating Point Operations. In *Proceedings 2007 International Conference on Field Programmable Logic and Applications (FPL)*. Delft : IEEE, 2007, FPL 2007, Amsterdam.

[B10] **R.Bartosinski**, J.Kadlec. Simulation of MCU hardware peripherals. In *Technical Computing Prague 2007. 15th annual conference proceedings*, 2007, Prague.

[B11] P.Stružka, L.Waszniowski, **R.Bartosinski**, T.Bysterský. Design of Control Application Using Processor Expert Blockset. In *Technical Computing Prague 2007. 15th annual conference proceedings*, 2007, Prague.

[B12] **R.Bartosinski**, M.Daněk, P.Honzík, J.Kadlec. Modelling Self-Adaptive Networked Entities in Matlab/Simulink. In *Technical Computing Prague 2007. 15th annual conference proceedings*, 2007, Prague.

[B13] M.Daněk, J.Kadlec, **R.Bartosinski**, L.Kohout. Increasing the Level of Abstraction in FPGA-based Designes. In *International Conference on Field Programmable Logic and Applications 2008*, Heidelberg, 2008. Kirchhoff Institute for Physics.

[B14] M.Daněk, J.-M.Philippe, **R.Bartosinski**, P.Honzík, Ch.Gamrat. Self-Adaptive Networked Entities for Building Pervasive Computing Aschitectures. In *International Conference on Evolvable Systems: From Biology to Harware, 8th International Conference, ICES 2008*, Heidelberg : Springer, 2008. Praha.

Non-Peer Reviewed and Unpublished Work

[C1] M.Líčko, **R.Bartosinski**, M.Kühl A High-Level Design Approach Towards FPGA-based Filter Design. *8th International Student Conference on Electrical Engineering, POSTER 2004*, FEL ČVUT v Praze, 2004.

[C2] **R.Bartosinski**. Dynamic Partial Reconfiguration on Xilinx FPGA. In *POSTER 2005 [CD-ROM]*. CTU Faculty of Electrical Engineering, 2005.

[C3] **R.Bartosinski**. Adaptive Identification Based on RLS Algorithms in Embedded Systems. PhD Thesis Proposal. 2006.

[C4] **R.Bartosinski**. RCCOM knihovna - podpora hardware-in-the-loop simulace na FPGA z prostředí Matalb/Simulink. Application Note, 2007, ÚTIA AV ČR.

[C5] L.Kafka, **R.Bartosinski**, M.Daněk. Accessory Tools for Partial Dynamic Reconfiguration on Xilinx FPGAs. Application Note, 2007, ÚTIA AV ČR.

[C6] **R.Bartosinski**. RCCOM - podpora komunikace s kartami s FPGA z prostředí Matlab a Matlab/Simulink. Application Note, 2007. ÚTIA AV ČR.

[C7] **R.Bartosinski**. Knihovna Processor Expert-Simulink. Licensed software, 2008. ÚTIA AV ČR.

[C8] **R.Bartosinski**. Processor Expert AutoSAR-Simulink Library. Licensed software, 2008. ÚTIA AV ČR.