VECTORIZATION OF DIGITAL FILTERS FOR CVP

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

BAO LINH DANG
born in Hanoi, Vietnam

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

VECTORIZATION OF DIGITAL FILTERS FOR CVP

by BAO LINH DANG

Abstract

Digital filtering has been essential in digital communication systems. As a result, many attempts to implement digital filtering algorithms efficiently have been undertaken. Those attempts fall into: algorithm improvements, creating new suitable architectures for the existing algorithms or combining both approaches. This thesis describes the mapping of three important filtering algorithms, i.e. Finite Impulse Response, Decimation and Adaptive filters to a new DSP architecture - The Co-Vector Processor (CVP) - developed by Philips Research Laboratories, Eindhoven, The Netherlands. The main challenge in this work is to minimize the impact of the data dependencies present in the algorithms and to find optimized algorithm mappings to exploit the parallelism offered by CVP. In this report, different mapping approaches for the targeted algorithms will be investigated and implemented for CVP. The results expressed in the number of clock cycles and computation times of the different approaches will be compared and the best implementation will be chosen. In addition, the above results will also be compared with performance results of similar DSP architectures in the market. These comparisons will indicate the potential of the CVP architecture in respect to Digital Signal Processing algorithms. From the analysis of the mentioned filtering algorithms, this thesis also focuses on finding possible improvements for the CVP Instruction Set Architecture (ISA). We propose three modifications to the ISA which can lead to performance improvements for digital filtering algorithms in particular and vector processing in general.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-02

Committee Members :

Co-advisor:	Nur Engin (Philips Nat.Lab.)
Co-advisor:	Georgi Gaydadjiev (CE, TU Delft)
Chairperson:	Stamatis Vassiliadis (CE, TU Delft)
Member:	Alexander Yarovoy (IRCTR, TU Delft)

To my parents for their endless love and support

Contents

List of Figures viii

List of Tables ix

Acknowledgements xi

Glossary xiii

1 Introduction 1
- 1.1 Problem statement and Objectives . . . 2
- 1.2 Thesis Overview . . . 2

2 Co-Vector Processor Architecture 5
- 2.1 3G Challenges for modern DSP-based Architectures . . . 5
- 2.2 The Co-Vector Processor . . . 6
- 2.3 Programming with CVP - Tools and methodology . . . 9

3 The mapping of FIR filters 11
- 3.1 Vectorization Strategies for the basic case . . . 11
  - 3.1.1 The horizontal vectorization strategy . . . 12
  - 3.1.2 The vertical vectorization strategy . . . 14
- 3.2 The implementation of the basic case in CVP . . . 15
- 3.3 The general case . . . 17
- 3.4 Performance and Conclusions . . . 21

4 The Mapping of Decimation and Interpolation filters 23
- 4.1 Basic building blocks . . . 24
- 4.2 Vectorization strategies for the cases of small factors . . . 25
  - 4.2.1 Strategy 1 . . . 26
  - 4.2.2 Strategy 2 . . . 27
  - 4.2.3 Strategy 3 . . . 29
- 4.3 The general cases of arbitrary decimation factors . . . 32
- 4.4 Performance and Conclusions . . . 34

5 The mapping of Adaptive Filters 37
- 5.1 Applications of Adaptive Filters . . . 38
  - 5.1.1 Adaptive Equalization . . . 38
  - 5.1.2 Echo Cancellation . . . 39
  - 5.1.3 Adaptive Prediction . . . 39
  - 5.1.4 System Identification . . . 40
- 5.2 The FIR LMS Adaptive filter . . . 41
- 5.3 The Block Least Mean Square -BLMS- algorithm . . . 42

5.3.1 The BLMS algorithm . . . 42
5.3.2 Implementation of BLMS in the time domain . . . 44
5.3.3 Implementation of BLMS in the frequency domain . . . 47
5.4 The Block Exact LMS -BELMS . . . 50
5.4.1 Block Exact LMS 1 -BELMS1 . . . 51
5.4.2 Block Exact LMS 2 -BELMS2 . . . 53
5.5 Performance and Conclusions . . . 55

6 Conclusions **57**
6.1 Thesis's Contributions . . . 57
6.2 Future work . . . 58

A Programs for FIR filters **61**
A.1 Matlab program for results comparison . . . 61
A.2 Vertical Implementation . . . 62
A.3 Horizontal Implementation . . . 64
A.4 CVP-C program for FIR filter . . . 65

B Programs for Decimation filters **67**
B.1 Matlab program for Decimation filters . . . 67
B.2 Implementation of Parallelization Strategy 1 . . . 69
B.3 Implementation of Parallelization Strategy 2 . . . 71
B.4 Implementation of Parallelization Strategy 3 . . . 74

C Programs for Adaptive filters **77**
C.1 Implementation of BLMS algorithm in the time domain . . . 77

Bibliography **61**

List of Figures

1.1 Hardware Architecture of the Software-Modem project 1

2.1 The change in DSP speed measured by BDTImark 6
2.2 Detailed block diagram of CVP 8

3.1 The matrix representation of the N-tap FIR filter 12
3.2 Pseudo code for the vertical strategy 15
3.3 CVP's memory organization 16
3.4 Non-aligned vector read from the memory 17
3.5 The matrix FIR equation for the general case 18
3.6 Number of bits required to represent results of Multiply-accumulate operations 19
3.7 Pseudo code for the complete program 20
3.8 Comparison between the theoretical result and result computed by the CVP 21
3.9 Proposed integrated Scaling instruction 22

4.1 Decimated filter bank 23
4.2 M-fold Decimator and L-fold Interpolator 24
4.3 Decimation and Interpolation filters 25
4.4 Noble Identities for multirate systems 25
4.5 The matrix representation of a Decimation FIR filter with factor of 2 . . . 26
4.6 Pseudo code for the first strategy (M is the number of inputs) 27
4.7 Polyphase representations of Decimation and Interpolation filters 27
4.8 Block diagram of SHUFFLE unit 28
4.9 Shuffling a vector in CVP - the first half 29
4.10 Pseudo code for the second strategy 29
4.11 The matrix representation of a decimation FIR filter 30
4.12 Pseudo code for the strategy 3 31
4.13 The IFIR technique for efficient design of narrow-band filter [**?**] 33
4.14 Block diagram of a Decimation filter with factor M 33
4.15 Two stages representation of filter in Figure 4.14 34
4.16 Comparison between the theoretical result and results computed by CVP 35
4.17 Fractional decimation 35

5.1 General structure of an adaptive filtering system 37
5.2 Structure of an Adaptive equalizer 39
5.3 Structure of an Adaptive Echo Canceller [**?**] 39
5.4 Structure of an Adaptive Predictor 40
5.5 System Identification Model [**?**] 40
5.6 Structure of a Block LMS filter 43
5.7 Memory organization for the BLMS program 46
5.8 Implementation of Block LMS FIR algorithm in the frequency domain but coefficients are updated in time-domain 48

5.9 The complete implementation of Block LMS FIR algorithm in frequency domain (including the coefficient updating part) 48
5.10 Vector-dot operation on two 16-element vectors 55
5.11 The Multiply-Accumulate model [**?**] 56

List of Tables

3.1 The scheduling for strategy 1 (SENDL: Send a line; RCV: Receive a line; MUL: Multiplication; ASRAdi: Arithmetic Shift Right with double-word precision; DIADD: Intra-Add) 13
3.2 The scheduling for FIR program (SENDV: Send a vector; SSND: Send a scalar; RCV: Receive a vector; SRCV: Receive a scalar and N is the length of the filter) 17
3.3 States of the pointers used in the FIR implementation. h_0 is the first filter coefficient and X_0 is the first data vector 19
3.4 Performance of CVP on FIR filter 22

4.1 The scheduling for decimation FIR filter (SENDV: Send a vector; SSND: Send a scalar; RCV: Receive a vector; SRCV: Receive a scalar and N is the length of the filter) 32
4.2 Performance of CVP on various parallelization strategies 35

5.1 Performance comparison between LMS in the frequency- and time-domain 49
5.2 Number of cycles required by CVP's FFT implementation 50
5.3 Comparison between CVP's and Altivec's performance on the Block LMS algorithm 56

Glossary

3G	Third Generation of Mobile Communication
ADPCM	Adaptive Differential Pulse-Code Modulation
AMU	ALU and MAC unit
BDTI	Berkely Design Technology Inc
BELMS	Block Exact Least Mean Square
BLMS	Block Least Mean Square
CDMA	Code Division Multiple Access
CGU	Code Generation Unit
CVP	Co-Vector Processor
DSP	Digital Signal Processing
EEMBC	EDN Embedded Microprocessor Benchmark Consortium
FDD	Frequency Division Duplex
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GSM	Global System for Mobile Communication
IFFT	Inverse Fast Fourier Transform
IFIR	Interpolated FIR
IIR	Infinite Impulse Response
ISA	Instruction Set Architecture
LMS	Least Mean Square
MAC	Multiply and Accumulate
RF	Radio Frequency
SFU	Shuffle Unit
SIMD	Single Instruction Multiple Data
SLU	Shift Left Unit
SRU	Shift Right Unit
SW-MODEM	Software-Modem
TDD	Time Division Duplex
TD-SCDMA	Time Division Synchronous CDMA
UMTS	Universal Mobile Telecommunications System
VLIW	Very Long Instruction Word
VMU	Vector Memory Unit

1 Introduction

The demand toward high speed, integrated wireless telecommunication services have been increasing very rapidly in the last twenty years. As such, many new services have been adopted by service-providers and accordingly, the telecommunication networks have been evolving very fast, both in terms of the services diversity and mobile devices' complexity. Nowadays, a number of third generation wireless communication standards are under consideration, e.g. UMTS/TDD, UMTS/FDD, TD-SCDMA etc., and they are expected to co-exist with the 2G standards as well as their future extensions. Considering the above diversity, the need for a single flexible architecture that has the processing power to support different 3G standards (and the existing 2G standards) is emerging. The main advantage of such an approach is that it will provide ultimate flexibility and help its developer to remain competitive in different markets where different standards and services are adopted.

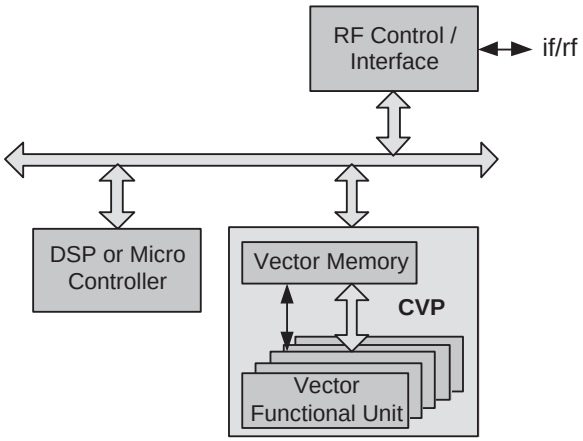

Figure 1.1: Hardware Architecture of the Software-Modem project

In response to this trend, a project called Software Modem (SW-MODEM) is being developed in Philips Research Laboratories, The Netherlands. The goal of the project is to design a programmable architecture that can support multiple Third Generation (3G) wireless standards as well as their developments in the future. The heart of the project is a co-processor referred as the Co-Vector Processor (CVP). The hardware framework of the SW-MODEM project is illustrated in Figure 1.1. When supplemented with a general purpose processor, CVP provides a powerful programmable solution for existing 3G baseband algorithms such as Rake receiver, digital filtering, channel coding/decoding etc. In general, CVP is a Very Long Instruction Word (VLIW) architecture with functional units supporting Single Instruction Multiple Data (SIMD) parallelism. With the

assumption that the targeted algorithms can be vectorized efficiently, CVP can deliver high processing power required by the 3G standards and meanwhile remain simple and a low-cost solution.

1.1 Problem statement and Objectives

Digital filters such as Finite Impulse Response (FIR), Infinite Impulse Response (IIR), Decimation, Adaptive filters etc., are one of the essential components in many digital systems. The importance of digital filtering is also indicated by its presence in most of the multimedia benchmark suites such as EDN Embedded Microprocessor Benchmark Consortium (EEMBC), Berkely Design Technology Inc.'s composite digital signal processing speed metric (BDTImark) and many others.

This thesis is an attempt to efficiently implement different digital filtering algorithms, i.e. FIR, Decimation and Adaptive filters on CVP. The main obstacle is to restructure or vectorize the original algorithms that are sequential [1] into block forms [2] suitable for the parallel architecture of CVP. The objectives of this project are three-fold:

- The first objective is to study various digital filtering algorithms, i.e. FIR filters, Decimation, Interpolation filters and Adaptive filters. The main target is to adapt the mentioned algorithms to the CVP's architecture. In other words, the original sequential algorithms will be restructured according to the SIMD and VLIW parallelism offered by CVP.
- The second objective of the thesis is to implement the investigated and restructured algorithms into CVP. The results of the mapping must be able to indicate the potential of CVP for the targeted digital filtering algorithms in the comparison with other benchmark results from other architectures such as Motorola's Altivec, TI's TMS320C64x or Analog Devices' TigerShark.
- From the above exploration, the last objective of the thesis is to find possible architectural bottlenecks suitable for improvement. More precisely, we also aim at exploring the possibilities to fit the CVP Instruction Set Architecture (ISA) better to the algorithms under focus.

1.2 Thesis Overview

With the above objectives in mind, the thesis is organized as follows:

- Chapter 2 introduces the requirements posed by the development of new generation of mobile telecommunication on DSP architectures design. After that, the general architectural description of the Co-Vector Processor (CVP) will be provided. The chapter will show how each requirement is addressed by CVP. In addition, the methodology and tools that will be used for programming and benchmarking throughout the thesis are also presented.

[1] A sequential algorithm is defined as an algorithm for calculating one output at a time

[2] Inversely, a block algorithm is defined as an algorithm for calculating multiple outputs at a time

- Chapter 3 starts the main part of the thesis by describing the mapping of the **Finite Impulse Response** (FIR) filter. In this chapter, we discuss two vectorization strategies, namely the Horizontal and the Vertical, for mapping FIR filters into CVP. Next, the advantages and disadvantages of the two strategies are listed and compared. We will discuss that the second strategy - The Vertical vectorization - is more efficient and suitable for implementation. Its performance on CVP is finally compared with the available FIR filter's benchmark results on various DSPs in the market. It will be shown that CVP has better performance than its counterparts.

- Chapter 4 presents the mapping of **Decimation and Interpolation** algorithms. Unlike the FIR algorithm, the decimation and interpolation filters involve distributed data in the memory. This poses a difficulty on the mapping since the data must be collected and rearranged before being processed. Three vectorization strategies for decimation filters with relatively small decimation factors will be discussed. We will show that the third strategy has the best performance. However, it also has a narrower application domain since it can only be used for the cases of decimation filters with factor of 2 and 4. Moreover, we will present that the three strategies can be applied for a general case with larger decimation factors. The performance of the three implementations concludes the chapter.

- Chapter 5 examines various vectorization strategies for **Adaptive** filters. The vectorization of the algorithm is difficult since an output is not only dependent on the inputs and filter coefficients but also on the previous outputs. We continue by investigating different block computation algorithms for adaptive filters where multiple outputs are calculated at once. We differentiate two kinds of block algorithms: the **Block Exact** and the approximate **Block** algorithms. The Block Exact methods have exactly the same response and results as the original sequential algorithm, but they also have more complicated structures. Meanwhile, the Block approaches offer much simpler structures but poorer performances in terms of convergence speed. After investigating these algorithm, we conclude that there is only one suitable adaptive algorithm for CVP, namely the Block LMS (BLMS). The performance comparison between the implementation of BLMS in CVP and in Altivec concludes the chapter.

- Finally, Chapter 6 will conclude the thesis by summarizing the contributions of the thesis and indicating the directions for future work.

Co-Vector Processor Architecture

2

In this chapter, several challenges and requirements that the present and coming generations of mobile communication imposes on modern computer architectures are discussed. From the general requirements, Sections 2.2 shows how the requirements are addressed by the Co-Vector Processor (CVP) and how CVP was designed to overcome those challenges. Finally, Section 2.3 introduces the CVP's tools and methodology for programming and benchmarking.

2.1 3G Challenges for modern DSP-based Architectures

Nowadays, programmable digital signal processors (DSPs) are pervasive in the wireless handset market for digital cellular telephony. In the last decades, there have been two distinct approaches to the implementation of mobile handsets. The first approach focuses on programmable DSPs which provide flexibility for future technologies and standards. Meanwhile, the second focuses on Application Specific Integrated Circuits (ASICs) which provide fast and low-cost alternatives for specific applications. The trend has been and will likely continue to be some combination of both approaches. However, in [**?**] and [**?**], the authors have persuasively argued that the **programmability requirement** is becoming more and more compelling, especially for the Third Generation of Mobile communication (3G) applications. Some of the reasons that lead to the above argument are as follows:

- In the recent years, there have been a number of different standards in different markets around the world, e.g. UMTS in Europe, CDMA2000 in America or TD-SCDMA in China.
- Telecommunication products, e.g. mobile phones, have evolved beyond their original functionality. Different from the 2G systems which are dedicated to real-time voice applications, 3G counterparts must deal with both real-time and non-real-time applications, e.g. streaming video, email, web-surfing etc.

Higher data transfer rates and complexity of 3G systems also pose increased demands on Digital Signal Processors's **processing power**. It is estimated that the complexity of 3G base-band processing could be measured in billions of operations per second. In [**?**], four different DSP processors were benchmarked using the BDTImark. Figure 2.1 shows the results of this comparison. It can be seen that the processing power required for Third Generation of Mobile communication (3G) is still far from what have been achieved.

Another challenge posing on today's architectures is to close the gap between the frequency of the processor which doubles every three years, and the memory access time

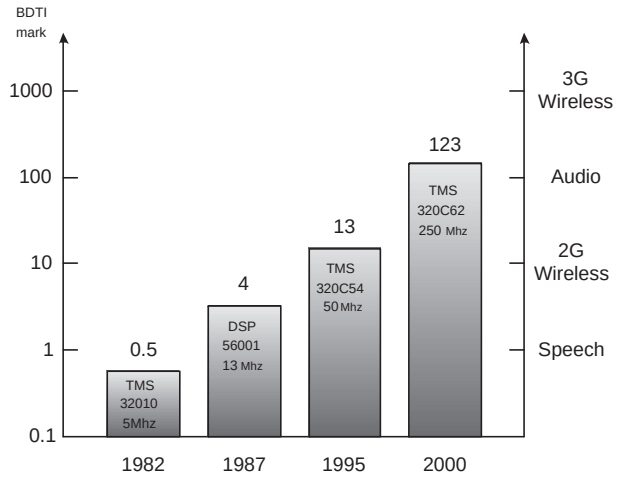

Figure 2.1: The change in DSP speed measured by BDTImark

that only increases by ∼7% per year [**?**]. Applications such as multimedia and communication applications use larger data structures than traditional applications. This issue becomes more problematic in the case of "multi-issue" processors [**?**]. In short, a modern DSP-based architecture must be designed to have **broad memory bandwidth**.

In response to the above requirements, many architectures that incorporate multiple functional and executional units as well as deep pipeline stages have been introduced to the market. Some successful examples are TigerSHARC of Analog Devices Inc, PowerPC with AltiVec vector parallel processing extension of Motorola, TMS320C6xxx from Texas Instruments [**?**, **?**, **?**]. However, as analyzed in [**?**, **?**], a better parallel processing solution is still needed for 3G's signal processing digital signal processing algorithms.

2.2 The Co-Vector Processor

In the light of the requirements presented in the previous section, the Co-Vector Processor is developed with the idea that a large number of computation-intensive algorithms will be efficiently executed on it. In this architecture, the main approach is to develop a vector processing engine that could exhaustively exploit the instruction level and data parallelism offered by the targeted 3G baseband algorithms such as rake receiver, digital filtering, channel coding/decoding, FFT etc. As a result, the CVP architecture employs many advanced techniques from VLIW and SIMD processors. Next, we show how CVP is designed to meet the three requirements.

- **Programmability Requirement**

  Programmability usually comes at the expense of decreasing performance and increasing the complexity of the Instruction Set Architecture. In an attempt to keep the architecture simple, the flow control part (branches, jumps, subroutines etc.) will be performed by a control processor. This control processor can be either a

general purpose processor or a conventional DSP. A large part of the software will be executed on this processor and only the computational extensive part - inner loops of DSP algorithms - will be executed on CVP.

Supplemented with the flow-control ability of the control processor, CVP also offers a rich instruction set including instructions that are specific for 3G processing, e.g. Intra-add, Correlate, Magnitude etc. and generic ones, e.g. ADD, MAC, SHIFT, Load/Store etc. Furthermore, CVP has two data paths, i.e. scalar path and vector path which when combined together, offer high flexibility and help with reducing the interaction with the host processor. Similar to many high performance DSP processors, CVP incorporates Zero-overhead Looping and several address-computation units. The detailed architecture of CVP is shown in Figure 2.2.

The CVP's ISA operates on a variety of data types: vectors or scalars. A vector can contain 32 elements of 8-bit single-word, 16 elements of 16-bit double-word or 8 elements of 32-bit quad-word. A scalar can be a single-word, a double-word or a quad-word. CVP supports both real and complex fixed-point arithmetics and 32-bit fixed-point representation could be the solution for applications requiring high-precision computation.

Unlike traditional vector processors, CVP supports only unit-stride vector accesses. This imposes a problem on data-packing when exploiting Instruction Level Parallelism exposed in many algorithms [**?**, **?**]. For instance, data gathering is very important for Decimation filters. This can be done easily in traditional vector processors where non-unit stride accesses are supported. In the case of CVP, this problem can be addressed by using the SHUFFLE (SFU) unit (Figure 2.2) that is able to rearrange elements of a vector in an arbitrary order.

- **High Processing Power Requirement**

Figure 2.2: Detailed block diagram of CVP

There are several ways to increase the processing power of a processor. One popular way to increase the processing ability is by increasing the pipeline depth - or the number of pipeline stages. However, this does not always help. Increasing the pipeline depth also means increasing the data dependency that causes stalls. Moreover, clock skew and pipeline register delay limit the performance gained by pipelining. To improve the performance beyond the gains afforded by increasing the pipeline depth, more execution units are added to conventional DSP architectures. However, this technique exposes problems that are difficult to solve, such as unfriendly compiler targets. Currently, most DSPs utilize "multi-issue" approach, e.g. Very Long Instruction Word (VLIW), Superscalar or Single Instruction Multiple Data (SIMD) to improve the processing power

Similarly, CVP employs VLIW approach to increase its performance. The architecture is divided into seven Functional Units (see Figure 2.2). More precisely, they are: Instruction Distribution Unit (IDU), Vector Memory Unit (VMU), Code Generation Unit (CGU), ALU and MAC Unit (AMU), Shuffle Unit (SFU), Shift Left Unit (SLU) and Shift Right Unit (SRU). With this architecture, in one cycle, CVP issue one very long instruction that contains seven execution segments. Another architectural paradigm employed by CVP is Single Instruction Multiple Data (SIMD). This technique allows the processor to execute multiple instances of the same operation using different data. The CVP architecture contains an array of 32 identical processing elements that can execute the same instruction.

- **High Memory Bandwidth Requirement**[1]

  To be consistent with the high processing load, high memory bandwidth is required. The high memory bandwidth is achieved by accessing the memory vector wide, i.e. 32 words of 8 bits during a single read or write access. CVP uses the Harvard architecture which separates program and data memory to gain bandwidth. Program memory fragments reside in the Instruction Distribution Unit and Data memory - Vector Memory locates in Vector Memory Unit. For better performance, vector-wide caches are utilized to store frequently repeated instructions in order to minimize the Instruction Memory for data fetches. Dedicated hardware is also used in Vector Memory Unit for memory address generation (acu0-acu7). Furthermore, to exploit the regularity of most DSP algorithms, CVP supports post-increment addressing mode which is proved to be very efficient for repetitive computations on a series of data stored at sequential memory locations.

2.3 Programming with CVP - Tools and methodology

In this thesis, the following tools are used for programming and benchmarking.

- **CVP's assembler**

  The assembler translates a program written in CVP's assembly language into a CVP-binary object file. This object file can be executed on either the CVP core (not yet available) or the CVP instruction-set Simulator. Programming in Assembly will have the following advantages:

  – Programming in Assembly allows better understand and utilization of the architecture.

  – Programs written in assembly will be more optimized since resources can be better allocated and instructions can be better pipelined.

  However, programming at the assembly level is difficult and error prone. Instruction scheduling will be complicated for large programs. Furthermore, The assem-

[1]The term "memory bandwidth" used here indicates the bandwidth of CVP's internal Vector memory (Figure 2.2).

bler is dependent on the Instruction Set Architecture (ISA) of CVP. Hence, programs will have to be rewritten if there are changes in the CVP's architecture. In this Software-Modem project, a higher level programming language, i.e. CVP-C, is under development but is not complete yet.

- **CVP's simulator**

  As mentioned above, the CVP instruction-set simulator is used to execute object files produced by the Assembler. The simulator is bit-true and almost cycle-true. Besides the options of viewing registers' and memory's contents, the number of executed instructions is also recorded during a simulation. This option is very convenient for benchmarking. As discussed in the Section 2.2, in each cycle CVP issues a single instruction. As a result, the number of cycles required to complete a task will actually be equal to the number of instructions executed. However, this measurement does not consider the stalls caused by non-aligned vector memory accesses. For this reason, the simulator is "almost cycle-true" for programs that cause cache misses. For example, a non-aligned vector read requires two cycles when there is a cache miss while it always needs only one in the simulator. For programs that do not cause cache misses, the simulator is cycle-true.

- **Matlab**

  Matlab programs are used during the thesis for the verification purposes. Provided the same inputs and parameters as in CVP-Assembly programs, results produced by the Matlab programs are then compared with results produced by CVP to prove the correctness of the implementation. Moreover, computation precision can also be investigated through this comparison since floating-point data is used in Matlab while CVP is a fixed-point architecture.

All of the programs in this thesis are written in CVP-Assembly using the above programming tools (the programs can be found in Appendix A, B, C). The results are compared with theoretical results produced by Matlab. The benchmarks results are produced with the assumption that data is already presented in the line cache. Stalls caused by non-aligned memory accesses are also discussed during the implementation and included in the final results.

To explain the algorithms in the following chapters, some notations are used. To denote a scalar value, small letters are used, e.g. h, x etc. Vectors are denoted by using upper case letters, e.g. H, X and matrixes are represented by upper case and boldface letters ($\mathbf{X}, \mathbf{H}$ etc.)

3 The mapping of FIR filters

The finite-impulse response (FIR) filter has been and continues to be one of the most fundamental processing elements in many digital signal processing (DSP) systems ranging from video and image processing to wireless communication. There has been a lot of effort to realize FIR filter efficiently by using different filter structures, e.g. transversal, symmetric transversal, lattice etc., both in the time domain and in the frequency domain (by using fast algorithms such as Fast Fourier Transform) [**?**, **?**, **?**].

In this chapter, the mapping of a FIR filter with 16-bit fixed point coefficients and inputs is described[1]. Starting from the basic case where 16 outputs are calculated from a N-tap FIR filter, we end up with the general case where a N taps FIR filter is used to calculate 16*M outputs. It is supposed that the number of inputs are multiple of 16 as it matches the architecture of CVP (256-bit wide data path). This assumption would not reduce the generality of the problem because normally, when the number of inputs is not a multiple-of-16 number and much larger than the number of filter coefficients, we could always use zero padding to extend the length of the input vector to the next proper number.

To exploit the parallelism offered by CVP, a programmer must first choose a proper parallelization strategy for the targeted algorithms. In other words, the investigated problems must first be vectorized or algorithmically restructured. This chapter is organized as follows: Section 3.1 discusses two vectorization strategies for calculating 16 outputs from an N-tap FIR filter. Section 3.2 presents the implementation of the two strategies. The implementation is generalized in Section 3.3 where 16*M outputs are calculated and finally, Section 3.4 provides the performance of CVP on the vectorized algorithm.

3.1 Vectorization Strategies for the basic case

This section describes two vectorization strategies. As will be discussed later, the implementation of the first strategy - the Horizontal - is not efficient and can not be applied for general case with arbitrary filter length. We show that the second strategy - the Vertical - exploits efficiently the architecture. This strategy uses both the vector and the scalar data paths available in CVP. Furthermore, this strategy does not depend on filter lengths. With these reasons, the vertical strategy is chosen for CVP implementation (Section 3.2). The implementation of the Horizontal strategy will be discussed briefly in Section 3.1.1.

[1]Because of the precision granularity of CVP is 8 bits, this is also the case for filters with coefficients and inputs encoded by 9-bit to 16-bit fixed-point representation

3.1.1 The horizontal vectorization strategy

A FIR filter is presented by

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k} \tag{3.1}$$

where $h'_k s \quad (0 \leq k \leq N-1)$ are the filter coefficients and $x'_k s$ are the inputs. This equation can also be represented in matrix form (for the case 16 outputs are calculated from an N-tap FIR filter)

$$\begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ y_{k+15} \end{bmatrix} = \begin{bmatrix} x_k & x_{k-1} & \cdots & x_{k-N+1} \\ x_{k+1} & x_k & \ddots & x_{k-N+2} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{k+15} & x_{k+14} & \cdots & x_{k-N+16} \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ \vdots \\ \vdots \\ h_{N-1} \end{bmatrix}$$

Figure 3.1: The matrix representation of the N-tap FIR filter

or using different notation:

$$Y = \mathbf{X} * H \tag{3.2}$$

where Y and H are the output and coefficient vectors respectively and $\mathbf{X}$ is the input matrix. An obvious way to parallelize this algorithm for CVP's SIMD-style parallel processing is to perform the multiplications horizontally and then "intra-add" the elements together to get a single output at a time.

$$y_k = h_0 x_k + h_1 x_{k-1} + \cdots + h_{N-1} x_{k-N+1} \tag{3.3}$$

In general, the Intra-Add operation is defined as follows:

Definition 1 *The Intra-Add of every L elements in a vector of length N:* $X = [x_0,\ x_1, \cdots,\ x_{N-1}]$ *is also a vector of length N:*

$$Intra\_Add_L(X) = R \quad (R = [r_0,\ r_1, \cdots,\ r_{N-1}]) \ where$$
$$r_k = \begin{cases} \sum_{i=k}^{k+L-1} x_i & if \quad (k\ mod\ L) = 0 \\ 0 & if \quad (k\ mod\ L) \neq 0 \end{cases}$$

Using the above definition, Equation 3.3 can be re-written in a compact form

$$y_k = Intra\_Add_N(X_k * H) \tag{3.4}$$

where $X_k = [x_k, x_{k-1}, \cdots, x_{k-N+1}]$ and N is the filter length. The vector multiplication in the above equation is element-wise.

The scheduling for a single loop in the 16-tap FIR implementation is shown in Table 3.1. In the first cycle, a data line is sent from the vector memory to AMU using the instruction SENDL. The stall in cycle 2 is a result of a cache-miss. In general, CVP needs a cycle to read a line from the main memory. At the beginning, the cache is assumed to be empty, hence there will be a stall cycle before the line is available in the cache to be read. In cycle 3, VMU sends the coefficient vector with the same instruction and AMU receives the data vector sent in the first cycle. After the coefficient vector is received in cycle 5, we start the multiplication (MUL), scaling (ASRA) and Intra-Add (IADD) in cycles 6,7 and 8 respectively.

The advantage of such implementation is that the program will contain only one loop and thus is simple and small in code size. However, this implementation would result in significant inefficiency since we have to scale the results of the multiplication before intra-adding. As mentioned before, each data sample is represented using 16-bit fixed point format. The result of a multiplication can be as long as 31-bit. Before any further calculations, the 31-bit results must be reduced to 16-bit or arithmetically scaled down 15 bits. As a consequence, scaling, "intra-adding" as well as storing the results back to memory must be performed inside the loop. The efficiency of the algorithm measured by the averaged number of cycles needed to get a single output will be very low. Furthermore, due to the fixed register's length in CVP, this strategy can only work efficiently with the cases where the filter's length is a multiple-of-16 number. If this constrain is not satisfied, there will be a waste of calculation. For instance when the filter length N is 24, two 16×16 multiplications are still needed to calculate one output which is exactly the same as the case of 32-tap filter. From the scheduling, this Horizontal Strategy spends about 7 cycles on calculating a single output. This results in a low efficiency of 1/7 or $\sim 15\%$.

Cycle	VMU		AMU	AMU
1	SENDL			
2	STALL		STALL	STALL
3	SENDL		RCV	
4			RCV	
5				MUL
6				ASRAdi
7				DIADD

Table 3.1: The scheduling for strategy 1 (SENDL: Send a line; RCV: Receive a line; MUL: Multiplication; ASRAdi: Arithmetic Shift Right with double-word precision; DIADD: Intra-Add)

From the above analysis and scheduling, the following remarks on the CVP's vectorization of FIR filter in particular and filtering algorithms in general can be made:

- The horizontal vectorization requires the multiplication of two vectors, i.e. data vector X_k and coefficient vector H_k. This requirement poses a difficulty since the AMU can only fetch one vector in a cycle. As a result, it needs three cycles (Table 3.1) to get the two vectors for a multiplication. Basically, there are two approaches to exploit the CVP architecture more efficiently. We can either use the SHIFT units as second source or we can find other vectorization strategies that exploits both the vector and scalar data-paths in CVP.
- One valuable characteristic of many digital filtering algorithms is the regularity in the computation pattern, namely the Multiply-Accumulate structure. In other words, the vectorization should be able to reserve the Multiply and Accumulate structures of those algorithms.

Examples stemming from these remarks can be seen in the next section - the Vertical strategy or in the Section 4.2.3 for Decimation filters.

3.1.2 The vertical vectorization strategy

An alternative approach is to vectorize the algorithm vertically or partition the input matrix **X** column-wise (or vertically). The FIR equation in Equation 3.1 can be rewritten as follows:

$$Y = X_0 * h_0 + X_1 * h_1 + \cdots + X_{N-1} * h_{N-1} \tag{3.5}$$

where

$$\begin{cases} X_0 = [x_k,\ x_{k+1},\ x_{k+2}, \cdots,\ x_{k+15}] \\ X_1 = [x_{k-1},\ x_k,\ x_{k+1}, \cdots,\ x_{k+14}] \\ X_2 = [x_{k-2},\ x_{k-1},\ x_k, \cdots,\ x_{k+13}] \\ \cdots\cdots \\ \cdots\cdots \\ X_{15} = [x_{k-15},\ x_{k-14},\ x_{k-13}, \cdots,\ x_k]. \end{cases}$$ [2]

In the vectorized expression 3.5, instead of calculating a single output at a time, 16 outputs will be calculated at once. This scheme needs N Multiply and Accumulate (MAC) operations to get the final 16 outputs. The operands of a MAC operation are an input vector (X_i) and a coefficient scalar (h_k). By the application of the broadcast register in ALU and MAC Unit (AMU), a scalar can be received from other functional units. It is then broadcast or replicated across an entire vector and hence can be used as an operand in the MAC operation. We have:

[2] Notice that the definition of X_k in this strategy is different from the previous one

$$Y = MAC(X_0,\ h_0) + MAC(X_1,\ h_1) + \cdots + MAC(X_{N-1},\ h_{N-1}) \tag{3.6}$$

The main disadvantage of this strategy is that it can only operate on input sequences of lengths that are a multiple of 16 number. In addition, the program will now contain 2 loops - an inner loop to calculate 16 outputs and an outer loop that evolves vertically along the input matrix (**X**). This will result in a program that is longer than the first strategy.

The pseudo code for the inner loop of this algorithm is shown in Figure 3.2

```
initialize_address_pointers();
LOOP(N times);
{
    load(data_vector X);
    load(coefficient_scalar h);
    multiply_accumulate(X, h);
    update_address_pointers();
}
```

Figure 3.2: Pseudo code for the vertical strategy

3.2 The implementation of the basic case in CVP

In this section, the algorithm shown in Figure 3.2 will be mapped to CVP. The inner loop contains only 4 tasks and can be packed into one CVP VLIW instruction. The most important part is to get data ready for MAC instructions; this includes initializing, updating pointers and loading pointed data into the registers.

Figure 3.3 shows the organization of the inputs and coefficients in the CVP's memory. Two pointer registers are needed for addressing an input vector and a coefficient scalar. At the initialization stage, the first pointer (e.g. acu0) is positioned at the address of the first data vector X_0 and the second (e.g. acu1) points to the beginning of the filter's coefficient vector (h_0). This coefficient will be sent directly from Vector Memory Unit (VMU) using "scalar-send" instruction (SSND).

As shown in Figure 3.1, X_0 is shifted to the left by one element to produce X_1, X_1 is shifted by one element to produce X_2 and so on. There are basically two ways to shift a vector in CVP, either by using the SHIFT Units or by fetching the vector directly from the memory after incrementing the pointer by 1 data element. The first approach requires more resources and increases the data traffic between the memory system and the register files. The second solution is done conveniently using the "Send-Vector" (SENDV) instruction of the VMU. When SENDV is executed, VMU fetches a vector starting from the pointed location by the pointer. The length of a vector depends on the size of each data element,i.e. 32*8-bit, 16*16-bit or 8*32-bit. In general, a non-aligned

access requires two cycles to read two lines from the memory. Since a line access requires only one cycle, the additional cycle is the penalty for performing non-aligned memory access. Because of the fact that many DSP algorithms have a linear access pattern, the requested vectors are consecutively stored in the memory. In this case, after the first access, the first line has already been in the internal buffer and thus, it requires only a cycle to finish the accesses.

Figure 3.3: CVP's memory organization

After one loop iteration, the pointers are updated to point to the next data vector and next coefficient scalar by post-incrementing by one element. Supposed that at the moment t, the pointer, e.g. acu0, points to vector $X_i = [x_i,\ x_{i+1}, \cdots,\ x_{i+15}]$. After being incremented by one element, at *t+1*, acu0 will point to the shifted vector $X_{i+1} = [x_{i+1},\ x_{i+2}, \cdots,\ x_{i+16}]$. As discussed above, the instruction SENDV is used to send the shifted vector to the AMU.

The scheduling for the FIR program is illustrated in Table 3.2. After the first three cycles (preamble part), the pipeline is filled, all the operations in one cycle will then be packed into one VLIW instruction and looped over. The stalls in cycle 2 and 3 are the result of non-aligned memory accesses (Figure 3.4) . To fetch a vector constituted from two memory lines, CVP requires two cycles to send both lines to the cache. It then needs one more cycle to read a scalar from the main memory. In the beginning of the scheduling, the cache is empty and thus it requires a total of 3 cycles from the first SENDV and SSND (Scalar-Send) to the first RCV and SRCV. However, from the cycle 4, both of the source lines are already present in the cache and the vector will be available right in the next cycle. As explained above, the data and coefficient pointers are updated to point to the next data element after each SENDV or SSND. After 16 SENDVs and 16 SSNDs, two new lines will be needed for the targeted vector and scalar. To send these two new lines, VMU requires 2 cycles and causes one stall cycle. In other words, every SENDV and SSND cause 1/16 stall cycle. For an N-tap FIR filter, this scheduling requires N SENDV and N SSND operations to calculate 16 outputs and results in $1/16 * N$ stall cycles. As such, to calculate M outputs from an N-tap filter, there will be $M/16 * 1/16 * N$ stall cycles.

Figure 3.4: Non-aligned vector read from the memory

Cycle	VMU	VMU		AMU	AMU	AMU
1	SENDV	SSND				
2	STALL	STALL		STALL	STALL	STALL
3	STALL	STALL		STALL	STALL	STALL
4	SENDV	SSND		RCV	SRCV	
5	SENDV	SSND		RCV	SRCV	MAC
6	SENDV	SSND		RCV	SRCV	MAC
	⋮	⋮		⋮	⋮	⋮
N+1	SENDV	SSND		RCV	SRCV	MAC
N+2	SENDV	SSND		RCV	SRCV	MAC
N+3				RCV	SRCV	MAC
N+4						MAC

Table 3.2: The scheduling for FIR program (SENDV: Send a vector; SSND: Send a scalar; RCV: Receive a vector; SRCV: Receive a scalar and N is the length of the filter)

3.3 The general case

In the previous section, 16 outputs were calculated at once. To move to the next 16 outputs, all pointers (an input data pointer and a coefficient pointer) need to be updated to the correct input vector and coefficient scalar, i.e. $X_{16(i+1)}$ and h_0 (provided that outputs from y_{16*i} to $y_{16*i+15}$ have been calculated in the previous step). The remaining process is the same as already described. In the program, only the input pointer needs to be updated. This is done by using LDBASE (Load-base) and LDOFFS (Load-offset) instructions. For the coefficient pointer, circular addressing is used so that after an inner-loop, the pointer returns to the beginning position and is ready for the next loop. The implementation contains two loops. The inner-loop calculates 16 outputs and the outer-loop evolves from one basic block to the next one.

Figure 3.5 is the matrix equation for the general case. For the first N outputs, the input matrix **X** is lower-triangular. To generate the zero upper part, a number of zero vectors need to be stored right before the actual inputs (Figure 3.3). The number of zero vectors (K) depends on the number of the filter's coefficients (N). We have:

$$K = \begin{cases} 1 & \text{if} \quad 1 \leq N \leq 16; \\ 2 & \text{if} \quad 17 \leq N \leq 32; \\ 3 & \text{if} \quad 33 \leq N \leq 64; \\ & \cdots\cdots \end{cases}$$

At the beginning, the input pointer is initialized to the address of the first actual input vector ($X_0 = [x_0,\ x_1, \cdots,\ x_{15}]$) and its increment is set to -2 (decrement 1 element). The coefficient pointer is initialized at the first coefficient (h_0), its increment is set to 2 (increment one element) and its boundary section is set to N*2 (N is the number of filter coefficients). After finishing all of the MAC operations, the coefficient

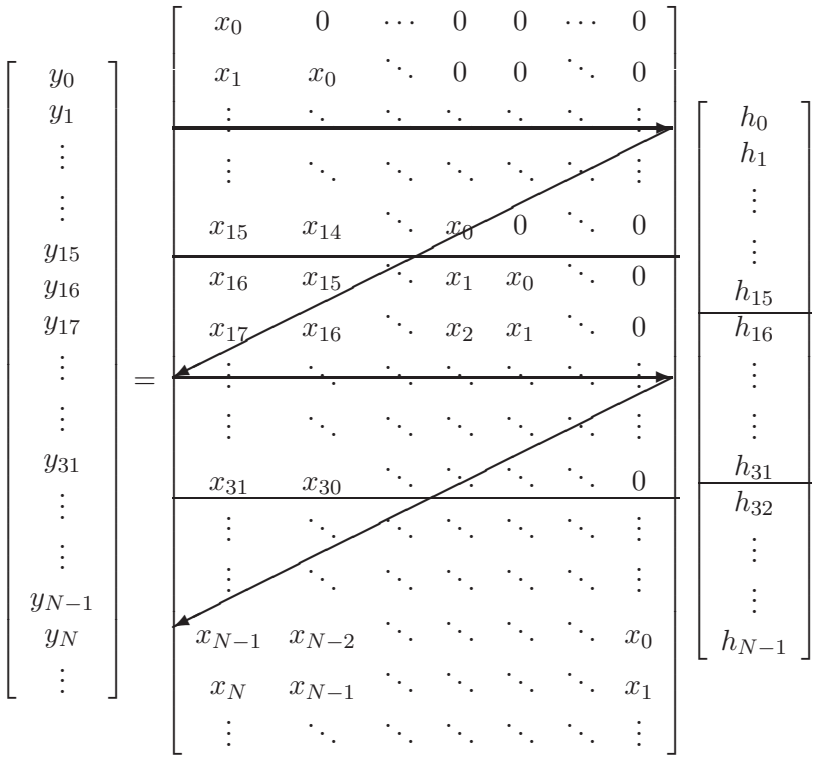

Figure 3.5: The matrix FIR equation for the general case

pointer automatically returns to the initialized states but the input pointer points to the vector $X_{-(N-1)} = [0, 0, \cdots, 0]$. To update the pointer to the right position ($X_{16} = [x_{16},\ x_{17}, \cdots,\ x_{31}]$), its increment must be set to $N * 2 + 32$ and then a SENDV is performed to update its offset. After the offset has been updated, the increment must be set to -2 again to start a new loop. The initialization and updating of the pointers are summarized in Table 3.3

		BASE	OFFSET	INCREMENT	BOUND
Coeff-pointer	Always	h_0	0	2	2*N
Data-pointer	Initialization	X_0	0	-2	0
	At the end of an Inner-loop	Unchanged	Unchanged	$2 * N + 32$	Unchanged
	At the end of an Outer-loop	Unchanged	Unchanged	-2	Unchanged

Table 3.3: States of the pointers used in the FIR implementation. h_0 is the first filter coefficient and X_0 is the first data vector

Figure 3.6: Number of bits required to represent results of Multiply-accumulate operations

Computation precision is also an issue in the implementation. In this section, the vectorization and implementation are for the 16-bit precision cases. After each multiplication, the result will be 31-bit long and can become longer after accumulation. Although CVP offers long accumulators (40 bits) with 8 extension bits for the cases of 16-bit precision, this problem may eventually lead to overflows. Figure 3.6 illustrates the number of bits required to represent the results of Multiply-Accumulate operations. The original operands are represented by 16 bits (1 sign bit and 15 value bits). At the beginning, a multiplication between two 16-bit numbers is performed and results in a 31-bit number. An accumulate or add following the multiplication can increase one bit of representation. To guarantee that there is no overflow, this accumulate process must be stopped when the results is 40 bit long. In other words, there are actually 9 extension bits for the case of 16-bit operands. As such, $2^9 = 512$ Multiply-Accumulates can be performed before overflows can occur.

However, to perform more than 512 MACs and to guarantee that there is no overflow in the implementation, a scaling must be performed after 512 MACs. In the context of this FIR implementation, only one Scaling is required after each inner-loop as long as the filter has less than 512 taps.

Figure 3.7 shows the pseudo code for the complete program to compute 16*M outputs with a N-tap FIR filter implemented with CVP.

```
input-pointer.base = &X0;
input-pointer.increment = -2;
coefficient-pointer.base = &h0;
coefficient-pointer.increment = 2;
coefficient-pointer.boundary = 2*N;
OUTTER_LOOP(M times);
    {
     send_a_vector(*input-pointer++);
     send_a_scalar(*coefficient-pointer++);
     receive_a_vector(), receive_a_scalar();
     INNER_LOOP(N-2 times);
          {
          MAC(X, h);
          send_a_vector(*input-pointer++);
          send_a_scalar(*coefficient-pointer++);
          receive_a_vector(), receive_a_scalar();
          }
     scaling_results();
     input-pointer.base = input-pointer.base + 2*N + 32;
     }
```

Figure 3.7: Pseudo code for the complete program

3.4 Performance and Conclusions

Figure 3.8: Comparison between the theoretical result and result computed by the CVP

Figure 3.8 shows the computed result of the above FIR algorithm implemented in CVP, the theoretical results computed in Matlab and the theoretical result when it is added with the rounding constant (0x8000). As mentioned before, the filter coefficients and data used in this implementation are 16-bit fixed-point numbers. During the computation, the 31-bit Multiply-Accumulate results must be reduced to 16-bit before any further calculation. This reduction is actually a truncation in which the 16 most significant bits are saved and the 16 least significant bits are discarded. To reduce the lost of precision caused by the truncation, a rounding constant, i.e. 0x8000 (or 1000 0000 0000 0000 in binary), should be added to the 31-bit results before performing the scaling. At this moment, however, we can not add the rounding constant to an accumulator since CVP does not support any accumulator operation. Through this example, we propose an improvement to the CVP architecture. The scaling operations should be integrated with a rounding constant addition. The proposed instruction is shown in Figure 3.9. Depending on the precision of the data, the value m of the rounding constant can be

0x80 when the precision is 8-bit, 0x8000 when the precision is 16-bit and 0x80000000 when the precision is 32-bit. The *src* field in the scaling instruction can be either an accumulator (acc0-acc7) or a register (amu0-amu15). The *amount* field specifies the scaling amount.

Figure 3.9: Proposed integrated Scaling instruction

In Figure 3.8, the errors between CVP results and the results computed by Matlab are depicted in the last two windows. As can be seen, the error between CVP result and the result added with rounding constant (in window 3) is evenly distributed around zero, meanwhile, the error between CVP result and the theoretical result is always positive (in window 2). The error between CVP result and the result added with rounding constant is also smaller. Its magnitude is about $5 * 10^{-5}$ compared with $15 * 10^{-5}$ of the case without rounding constant.

	Clock-rate (Mhz)	Clock cycles	Execution time (μs)
CVP	200	3728	18.6
ALTIVEC	600	9334	15.6
TIGERSHARC	300	7200	24
TMS320C64x	600	16243	27

Table 3.4: Performance of CVP on FIR filter

The performance results of various DSPs for calculating 1024 outputs with a 50-tap FIR filter are also listed in Table 3.4. From the analysis in Section 3.2, the number of stall cycles required to calculate 1024 outputs with a 32-tap FIR filter is: $1024/16*1/16*32 = 128$ *cycles*. This number is already included in Table 3.4.

As can be seen from the table, the performance of CVP is better than its counterparts. This can be explained by the advanced VLIW and SIMD architecture that CVP possesses and the parallel structure of the FIR algorithm. In this table, to compute the execution time for PowerPC with Altivec extension, we choose the PowerPC's clock-rate of 600Mhz, TigerShark's clock-rate of 300Mhz and TMS320C64 compared with CVP's clock-rate of 200Mhz.

The Mapping of Decimation and Interpolation filters

4

Multirate signal processing is another important tool in the modern signal processing systems. The idea is that we can separate different frequency bands in processed signals such as speech, image etc. and later reconstruct the original signal without any or with bearable losses. Since in such signals, low frequencies carry the large part of the information, we could always concentrate on the main bands that have been extracted and can continue processing them accordingly. As such, Multi-rate signal processing is basically concerned with problems in which more than one sampling rate is required in the system. In today's telecommunication systems, multirate systems in general, or decimation and interpolation filters in particular, can be found in many important applications, e.g. Digital Audio System, Subband Coding of Speech and Image Signals, ADC etc. [**?**]

Multirate processing involves two basic operations: decimation and interpolation. They are processes that transform the current sampling rate f_s to a new sampling rate f'_s. After decimation (a decimator with factor M takes every M^{th} sample), f'_s will be smaller than f_s and after interpolation, f'_s is larger than f_s. Fractional changes in the sampling rate (with factor of $\frac{L}{M}$) can be achieved by combining a decimator with factor M with an interpolator with factor L. Figure 4.1 shows an M-channel maximally decimated filter bank in which, M decimators form an analysis filter-bank which splits the signal x(n) into M sub-band signals. Each sub-band is then encoded prior to transmission. Similarly, at the other end of the system, M interpolators form a Synthesis filter bank which decodes, interpolates and finally combines M received sub-bands.

Figure 4.1: Decimated filter bank

In this chapter, the vectorization and mapping of Decimation and Interpolation filters to CVP are discussed. This chapter is organized as follows: Section 4.1 introduces the definitions and the basic building blocks of decimation and interpolation filters. Section 4.2 discusses various vectorization strategies for decimation FIR filters with small decimation factors and their implementations in CVP. For some special cases, another vectorization strategy which is more efficient could be used (Section 4.2.3). The implementation of interpolation that is simply the reversed process of decimation will be discussed briefly in Section 4.2.2. Section 4.3 extends the application domain of those

strategies discussed in Section 4.2 by showing that decimation filters with large decimation factors can be divided into multiple stages of smaller filters with smaller factors. Finally, Section 4.4 depicts the results of the vectorization strategies, their advantages as well as their disadvantages.

4.1 Basic building blocks

Figure 4.2 shows the two most basic building blocks in a multirate signal processing system, namely a decimator with the factor M and an interpolator with the factor L. Mathematically, they are characterized correspondingly by the equations 4.1 and 4.2.

Figure 4.2: M-fold Decimator and L-fold Interpolator

$$y_D(n) = x(Mn) \tag{4.1}$$

$$y_I(n) = \begin{cases} x(n/L) & if\ n\ is\ a\ multiple\ of\ L \\ 0 & otherwise \end{cases} \tag{4.2}$$

From Equation 4.1, an output at time n of a decimator is equal to the input at time Mn. As a consequence, only the input samples at the time index equal to a multiple number of M are retained and as such, we have performed a sampling-rate reduction. Inversely, interpolators are used to increase sampling rates. As shown in Figure 4.2(b) and Equation 4.2, the output $y_I(n)$ is obtained by inserting $L-1$ zeros between two adjacent samples of $x(n)$. As we might expect, the signal compression effects in the time-domain of decimators will cause the stretching effect in the frequency domain and thus aliasing. On contrary, an interpolation process corresponds to the stretch of signal in the time-domain and the imaging effect where extra copies of the basic spectrum are created in the frequency-domain.

The fundamental difference between the aliasing and imaging effects is that the aliasing effect causes loss of information but the imaging effect does not. To be able to reconstruct the signals, we have to prevent the aliasing as well as the imaging caused by decimation and interpolation. In most applications, a decimator is preceded by a bandpass filter to serve this purpose. Meanwhile, an interpolator is followed by bandpass filter to eliminate the undesired images. The combinations of a filter and a decimator or a interpolator is called decimation filters or interpolation filters accordingly (Figure 4.3).

It can also be proved that the sequence of a filter and a decimator or an interpolator in Figure 4.3 can be inverted. The prove of this equivalence is presented in [**?**]. These inversions are called the two **Noble Identities** and are shown in Figure 4.4. These

Figure 4.3: Decimation and Interpolation filters

identities are very valuable in many applications for efficient implementation of filters and filter banks.

Figure 4.4: Noble Identities for multirate systems

4.2 Vectorization strategies for the cases of small factors

In this section, an example of a decimation FIR filter with decimation factor of 2 is used. Based on this example, various vectorization strategies will be discussed and compared. This discussion will be expanded later to cover more general cases. Finally, the best solutions for different cases will be shosen.

The block diagram of the decimation filter shown in Figure 4.3(a) with decimation factor $M = 2$ is represented by:

$$y_{2n} = \sum_{k=0}^{N-1} h_k x_{2n-k} \tag{4.3}$$

where $h'_k s$ $(0 \leq k \leq N-1)$ are the filter coefficients and $x'_k s$ are the inputs. In other words, in the matrix form of FIR filter equation (Figure 4.5), only the even numbered outputs (y_{2*i}) are calculated. (Figure 4.5 shows the matrix representation of Equation 4.3)
or in vector form:

$$Y = \mathbf{X} * H \tag{4.4}$$

$$\begin{bmatrix} y_{2k} \\ y_{2k+2} \\ \vdots \\ \vdots \\ \vdots \\ y_{2k+30} \end{bmatrix} = \begin{bmatrix} x_{2k} & x_{2k-1} & \cdots & x_{2k-N+1} \\ x_{2k+2} & x_{2k+1} & \ddots & x_{2k-N+3} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{2k+30} & x_{2k+29} & \cdots & x_{2k-N+31} \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ \vdots \\ \vdots \\ h_{N-2} \\ h_{N-1} \end{bmatrix}$$

Figure 4.5: The matrix representation of a Decimation FIR filter with factor of 2

4.2.1 Strategy 1

The most straight-forward strategy is to calculate a single output at a time directly from the Equation 4.3.

$$y_{2n} = \sum_{k=0}^{N-1} h_k x_{2n-k} \tag{4.5}$$

With the Intra-Add operation defined in Chapter 3, Equation 4.5 can be re-written as follows:

$$y_{2n} = Intra\_Add_N(H * X_{2n}) \tag{4.6}$$

where

$$\begin{cases} H = [h_0,\ h_1,\ h_2, \cdots,\ h_{N-1}] \\ X_{2n} = [x_{2n},\ x_{2n-1},\ x_{2n-2}, \cdots,\ x_{2n-N+1}]. \end{cases}$$

In the next iteration, the next output will be calculated. We have

$$y_{2(n+1)} = Intra\_Add_N(H * X_{2n+2}).$$

Where $X_{2n+2} = [x_{2(n+1)},\ x_{2n+1},\ x_{2n}, \cdots,\ x_{2n-N+3}]$ is merely the shifted version of X_{2n} by two elements. This can be done by fetching the vector directly from the vector memory after incrementing the pointer by two data elements. The result of the multiplication (31 bits) is stored in an accumulator. At present, CVP supports Intra_Add operation but this instruction does not work with accumulators (acc0 - acc7). As a result, before performing Intra_Add, the result of the multiplication (31 bits) must be scaled down to 16 bits so that it fits into a register (amu0-amu15).

```
input-pointer.base = &X0;
input-pointer.increment = 2;
coefficient-pointer.base = &H;
LOOP(M/2 times);
    {
    load_data_vector(*input-pointer);
    load_coefficient_vector(*coefficient-pointer);
    multiply(X, H);
    scaling(15 bits);
    Intra_Add();
    input-pointer++;
    }
```

Figure 4.6: Pseudo code for the first strategy (M is the number of inputs)

The algorithm for this strategy is shown in Figure 4.6. This strategy has two main advantages. Firstly, the program contains only single loop, thus the program will be simple and small in terms of code size. Secondly, the decimation factor does not affect the algorithm. As can be seen in Figure 4.6, the decimation factor affects only the number of iterations and incrementing pointers which are trivial. However, as discussed in Section 3.1.1, the strategy has also shown a serious drawback that makes it very inefficient. Along with a multiplication in the loop body, a scaling and an Intra_Add must also be performed. This will decreases the program's efficiency drastically.

4.2.2 Strategy 2

Figure 4.7: Polyphase representations of Decimation and Interpolation filters

Another way to reduce the computation requirement is to use polyphase representation of the decimation and interpolation filters [**?**]. Figure 4.7 shows the block diagram of the 2-component polyphase representations of the decimation and interpolation filter shown in Figure 4.3 ($M = L = 2$).

In Figure 4.7(a), the input sequence (x_k) is divided into two smaller sequences. The first one contains the even-numbered input samples and the second contains the odd-numbered ones. As a result, the lengths of both input sequences (x_{2k} and x_{2k+1}) are half of the original (x_k). These two shorter sequences are then filtered by $E_0(z)$ and $E_1(z)$ whose lengths are again half of the original filter $H(z)$. The filters $H(z)$, $E_0(z)$, $E_1(z)$, $R_0(z)$ and $R_1(z)$ are related by the following formulas:

$$\begin{cases} H(z) = \sum_{n=-\infty}^{\infty} h(n)z^{-n} \\ E_0(z) \equiv R_0(z) = \sum_{n=-\infty}^{\infty} h(2n)z^{-n} \\ E_1(z) \equiv R_1(z) = \sum_{n=-\infty}^{\infty} h(2n+1)z^{-n} \end{cases}$$

For a N-tap FIR filter, single output requires N multiplications. Hence for M outputs, we needs MN multiplications. By using the polyphase representations, both the number of input samples and the length of each component FIR filters are reduced by the factor of 2. As a consequence, each decomposed filter - $E_0(z)$ or $E_1(z)$ - will be used to calculate only $M/2$ outputs. The total number of multiplications required will be $N/2M/2 + N/2M/2 = MN/2$ which is half of the original computation requirement. This example also shows show to re-arrange data with CVP's SHUFFLE unit. Unlike traditional vector processors, CVP does not support non-unit stride accesses. This poses a limitation on arranging and collecting data. To overcome this limitation, CVP includes a special unit to shuffle data elements in an arbitrary order (Figure 4.8).

Figure 4.8: Block diagram of SHUFFLE unit

The filtering process (FIR implementation) in Figure 4.7 has been already discussed in Chapter 3. The remaining problem is to implement the decimator and interpolator in CVP. Figure 4.9 shows how to decimate a vector with the factor of 2. By specifying the configurations or shuffle patterns, all elements in the vector are re-arranged accordingly. In CVP, it is possible to shuffle the odd target elements (each element is an 8-bit word) or the even elements by using ODD and EVEN instruction. A full shuffle is done by performing ODD and EVEN consecutively. The target elements will not be affected if the corresponding parts of the shuffling pattern are -1's. Since a decimated vector must be constructed from two source vectors, two shuffling patterns must be applied. The first pattern is used to obtain the first half of the targeted vector (Figure 4.9) and the second pattern is for the remaining part.

Figure 4.9: Shuffling a vector in CVP - the first half

After shuffling the input data, the decimated results are stored in memory and ready for the filtering operation. For the interpolation filter, the process is performed in reversed order. Filtering operations are executed first and then, the filtering results are interpolated by SHUFFLE unit (Figure 4.7 b, d).

The pseudo codes for this strategy is presented in Figure 4.10

```
shuffling_first_half()
storing_shuffling_result(*pointer1);
shuffling_second_half()
storing_shuffling_result(*pointer2);
filtering_first_half(&pointer1);
filtering_second_half(&pointer2);
adding_filtering_result();
```

Figure 4.10: Pseudo code for the second strategy

4.2.3 Strategy 3

The second strategy has improved the efficiency of the decimation filter but it still suffers from the shuffling operations. In some special cases, namely when the decimation factor is 2 or 4, another more efficient strategy can be used to further improve the efficiency .

The matrix equation of a decimation filter (with the decimation factor of 2) is shown in Figure 4.11.

$$\begin{bmatrix} y_k \\ y_{k+2} \\ \vdots \\ \vdots \\ y_{k+14} \\ \vdots \end{bmatrix} = \begin{bmatrix} x_k & x_{k-1} & \cdots & x_{k-N+1} \\ x_{k+2} & x_{k+1} & \ddots & x_{k-N+3} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{k+14} & x_{k+13} & \cdots & x_{k-N+15} \\ \vdots & \ddots & \ddots & \vdots \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ \vdots \\ \vdots \\ h_{N-2} \\ h_{N-1} \end{bmatrix}$$

Figure 4.11: The matrix representation of a decimation FIR filter

The above equation can be rewritten as follows

$$Y = X_0 * H_0 + X_1 * H_1 + \cdots + X_{N-1} * H_{N-1} \tag{4.7}$$

where

$$\begin{cases} X_0 = [x_k,\ x_{k+1},\ x_{k+2}, \cdots,\ x_{k+15}] \\ X_1 = [x_{k-1},\ x_k,\ x_{k+1}, \cdots,\ x_{k+14}] \\ \cdots\cdots \\ \cdots\cdots \\ X_{N-1} = [x_{k-N+1},\ x_{k-N+2},\ x_{k-N+3}, \cdots,\ x_{k-N+16}]. \end{cases}$$

and

$$\begin{cases} H_0 = [h_0,\ 0,\ h_0,\ 0, \cdots,\ h_0,\ 0] \\ H_1 = [h_1,\ 0,\ h_1,\ 0, \cdots,\ h_1,\ 0] \\ H_2 = [h_2,\ 0,\ h_2,\ 0, \cdots,\ h_2,\ 0] \\ \cdots\cdots \\ \cdots\cdots \\ H_{N-1} = [h_{N-1},\ 0,\ h_{N-1},\ 0, \cdots,\ h_{N-1},\ 0]. \end{cases}$$

Notice that X_1 is produced by shifting X_0 right one element and half of the elements in H_k are 0's, so Equation 5.4.2 can rewritten as follows:

$$Y = Intra\_Add_2(X'_1 * H'_1 + X'_3 * H'_3 + \cdots + X'_{N-1} * H'_{N-1}) \tag{4.8}$$

where

$$\begin{cases} X'_1 = [x_{k-1},\ x_k,\ x_{k+1}, \cdots,\ x_{k+14}] \\ X'_3 = [x_{k-3},\ x_{k-2},\ x_{k-1}, \cdots,\ x_{k+12}] \\ \cdots\cdots \\ \cdots\cdots \\ X'_{N-1} = [x_{k-N+1},\ x_{k-N+2},\ x_{k-N+3}, \cdots,\ x_{k-N+16}]. \end{cases}$$

and

$$\begin{cases} H'_1 = [h_1,\ h_0,\ h_1,\ h_0, \cdots,\ h_1,\ h_0] \\ H'_3 = [h_3,\ h_2,\ h_3,\ h_2, \cdots,\ h_3,\ h_2] \\ H'_5 = [h_5,\ h_4,\ h_5,\ h_4, \cdots,\ h_5,\ h_4] \\ \cdots\cdots \\ \cdots\cdots \\ H'_{N-1} = [h_{N-1},\ h_{N-2},\ h_{N-1},\ h_{N-2}, \cdots,\ h_{N-1},\ h_{N-2}]. \end{cases}$$

```
initialize_address_pointers();
LOOP(N/2 times);
    {
    load(data_vector X'_1);
    load(coefficient_vector  H'_1);
    multiply_accumulate (X'_1,  H'_1);
    }
scaling(15 bits);
Intra_Add_2();
update_address_pointers();
```

Figure 4.12: Pseudo code for the strategy 3

The pseudo codes for this strategy is presented in Figure 4.12.

As can be seen in the pseudo program in Figure 4.12, two source vectors, namely X'_k and H'_k, are required for each Multiply_Accumulate operation. This exposes a difficulty on mapping this algorithm to CVP. In each clock cycle, VMU can issue only one vector. The only solution is to use SHIFT units for issuing the second vector. One may notice that vector X'_1 can be shifted left by two elements and filled with two other elements, i.e. x_{k-2} and x_{k-3} to produce X'_3. To get the correct elements, e.g. x_{k-2} and x_{k-3} for the shifting operation, a double-size scalar (a quad-word scalar for the case of 16-bit precision or a double-word scalar for the case of 8-bit precision) is sent directly from Vector Memory Unit to the Shift Unit. It will be available for the SHIFTS operation in the next cycle.

Due to this limitation, the presented strategy can be used only for 2 cases:

- Decimation filters with decimation factor of 2 which are represented by 16-bit precision
- Decimation filters with decimation factor of 2 or 4 which are represented by 8-bit precision

The scheduling of this algorithm is shown in Table 4.1.

Cycle	VMU	VMU		AMU	AMU	AMU		SLU	SLU	SLU
1	SENDV	SSND								
2	STALL	STALL		STALL	STALL	STALL		STALL	STALL	STALL
3	STALL	STALL		STALL	STALL	STALL		STALL	STALL	STALL
4	SENDV	SSND		RCV				RCV	SRCV	
5	SENDV			RCV				RCV	SRCV	SHIFTS
6	SENDV	SSND		RCV	RCV	MAC		RCV	SRCV	SHIFTS
7	SENDV	SSND		RCV	RCV	MAC		RCV	SRCV	SHIFTS
	⋮	⋮		⋮	⋮	⋮		⋮	⋮	⋮
N/2+2	SENDV	SSND		RCV	RCV	MAC		RCV	SRCV	SHIFTS
N/2+3	SENDV			RCV	RCV	MAC				SHIFTS
N/2+4				RCV	RCV	MAC				
N/2+5						MAC				

Table 4.1: The scheduling for decimation FIR filter (SENDV: Send a vector; SSND: Send a scalar; RCV: Receive a vector; SRCV: Receive a scalar and N is the length of the filter)

4.3 The general cases of arbitrary decimation factors

Figure 4.13: The IFIR technique for efficient design of narrow-band filter [**?**]

In the section above, three vectorization strategies have been analyzed. However, the decimation factors used in the previous section were relatively small. Strategy 2 can be applied for any cases with decimation factors smaller than 16 and the Strategy 3 can be applied for the cases where decimation factors are equal to 2 or 4. However, the performance of Strategy 2 decreases linearly as the decimation factor increases since it requires more shuffling operations. In this section, we are going to discuss a solution for the cases of large decimation factors. We present that a decimation filter can be decomposed into multiple stages of smaller filters with smaller decimation factors. We then argue that it not only makes the implementation of decimation filters with arbitrary decimation factors possible in CVP, but the approach also has better performance than the single filter cases [**?**].

A general block diagram of a decimation filter with factor M is depicted in Figure 4.14

Suppose that the decimation factor M can be decomposed into M_1 and M_2, that is, $M = M_1 M_2$. Then, the decimation filter in Figure 4.14 can be rearranged as depicted in Figure 4.15(a). This is assuming that the combination of two filters $G(z)$ and $I(z)$ has the

Figure 4.14: Block diagram of a Decimation filter with factor M

same response as the original filter $H(z)$. Finally, by using a Noble Identity introduced in Section 4.1 , Figure 4.15(a) can be proved to be equivalent to Figure 4.15(b) - The multi-stages representation of the original filter.

Figure 4.15: Two stages representation of filter in Figure 4.14

As introduced above, the combination of two filters $I(z)$ and $G(z)$ must have the same response as the original filter $H(z)$. The designs of $I(z)$ and $G(z)$ can be done by using a technique called Interpolated FIR or IFIR. In fact, the IFIR itself is a narrow-band filter design technique and has nothing to do with Decimators and Interpolators. However, in the context of decimation filters, this technique has naturally allowed the design of multi-stages structures. An example of applying IFIR technique to the above Decimation filter with factor M is illustrated in Figure 4.13.

Figure 4.13(a) shows the specification of the original filter $H(z)$ in Figure 4.14. As explained above, our purpose is to find two filters $I(z)$ and $G(z)$ that when combined together, have the same response as of the original filter. For the simplicity of this discussion, we suppose that M_1 is 2. Instead of meeting the specification in 4.13(a), we try to meet a two-fold (M_1-fold in general) stretched specification in Figure 4.13(b). It can be seen that the filter $G(z)$ has double bandwidth so that its order is half of the targeted filter $H(z)$. This property also reduces the computation requirement by half. Figure 4.13(c) shows the magnitude response of $G(z^2)$. The filter now has two passbands. The left one is similar to the desired passband and the other is unwanted. To eliminate the unwanted band, another filter -$I(z)$- is used (Figure 4.13(d). Finally, the cascade of $I(z)$ followed by $G(z^2)$ in Figure 4.13(e) is the equivalence of the original filter $H(z)$.

The analysis above has shown the ability of designing cascaded structures for decimation filters. Generally, when M is large and we want to decompose the filter into k stages ($k > 2$), the above technique can always be applied. As presented, the decimation filter $H(z)$ with factor M can be decomposed into two decimation filters $I(z)$ and $G(z)$ with factors M_1 and M_2 respectively. In the same manner, the decimation filter $G(z)$ can be decomposed again into two other decimation filters with smaller factors. This process can continue as long as the decimation factor is not a prime number.

4.4 Performance and Conclusions

Figure 4.16: Comparison between the theoretical result and results computed by CVP

Figure 4.16 shows the graph of the computed results of the above strategies implemented in CVP and the theoretical result computed in Matlab. Small errors among the results due to quantization is depicted in the second window. We can see that the Strategy 2 has the lowest precision since the final result is the sum of two filtered outputs which were previous scaled. The Strategy 3 has the highest precision since the final result is scaled only once after several MAC operations.

Table 4.2 shows the performance and the code sizes of the implementation of the three vectorization strategies. The Strategy 3 has the best performance and moderate code size, but its main limitation is the application domain of the algorithm. It is only applicable in the cases of decimation filters which have the decimation factor of 2 and are coded by 16-bit fixed point representation or the cases of decimation filters which have the decimation factor of 4 and represented by 8-bit precision. However, this implementation can be used as a basic block to build up decimation filters with large decimation factors as illustrated in Section 4.3.

	Clock cycles	Code size (bytes)
Strategy 1	392	2,893
Strategy 2	224	6,232
Strategy 3	148	4,482

Table 4.2: Performance of CVP on various parallelization strategies

Figure 4.17: Fractional decimation

The second strategy's performance is moderate but this strategy can be applied in a wide range of decimation filters as well as interpolation filters (by simply reversing the operations order). Unfortunately, the performance of this strategy decreases linearly as the decimation or interpolation factors increase. The number of clock cycles required for the shuffling operations increase linearly with the increase of decimation or interpolation

factors. This limitation, however, exposes one of the drawbacks of CVP that requires improvement. At this moment, a full shuffle in CVP requires one "Odd" and one "Even" Shuffle. Moreover, this mechanism requires two shuffling patterns in the Vector memory. An improvement of full shuffling operation which completes in one cycle and requires only one pattern, will not only increase the performance of programs but also save memory space.

The implementation of fractional changes in sampling rates (with a factor of $\frac{L}{M}$) can be achieved by combining a decimator with factor M and an interpolator with factor L (Figure 4.17). By using the above vectorization strategies with decimation and interpolation filter, this application can also be implemented in CVP efficiently.

To summarize, this chapter described in details various algorithms for mapping decimation and interpolation filters to CVP. For each particular case, a suitable algorithm can be used to exploit the SIMD and VLIW parallelism offered by CVP. For very large decimation factor cases, the cascaded form of decimation filters should be used to reduce the decimation factors and the computation requirements.

The mapping of Adaptive Filters

5

Adaptive filtering is a valuable digital signal processing tool that finds applications in many areas such as interference and echo cancellation, channel equalization, linear prediction and spectra estimation. Different from other signal processing systems, adaptive systems require a minimum of prior knowledge regarding the signals or systems under analysis. In other words, in a communication system where the operating conditions are variable and the a-priori information about the input signals is insufficient, adaptive filters can adjust themselves to a certain criterion of optimality.

The objectives of any linear adaptive system is to form, via some continuous iterative processes, the optimum linear estimator for a given system or process. This research was first initiated by two seminal works by Wiener (paralleled by Kolmogorov) and by Kalman and Bucy [**?**, **?**, **?**]. In the past twenty years, a large number of computationally efficient algorithms for adaptive filtering have been developed. They are based on either a statistical approach, e.g. the Least-Mean Square (LMS) algorithm, or a deterministic approach, such as the Recursive Least-Square (RLS) algorithm. The main advantage of the LMS algorithm is its computational simplicity. Conversely, the RLS algorithm offers faster convergence but has a higher degree of complexity [**?**].

Figure 5.1: General structure of an adaptive filtering system

A general structure of an adaptive filtering system is depicted in Figure 5.1. In the system, a data sequence x[n] is input to a filtering system that has updatable parameters (coefficients). The filtered outputs (y[n]) are then compared with the target outputs (d[n]) to form error samples (e[n]) which are fed back the Adaptive algorithm in order to adjust the parameters of the filtering system. In general, adaptive filters contain two distinct parts: a filtering part and an adaptive part. The filtering part is designed to perform a desired processing function and the adaptive part must be able to adjust the filter's coefficients to improve the filtering performance. As mentioned above, adaptive filters can be used in various applications with different configurations of the filtering and adaptive part. Typically, the filtering part is either a Finite Impulse Response (FIR) filter or an Infinite Impulse Response (IIR) filter and the two most used adaptive algorithms are LMS and RLS. In this report, we focus on the combination of a FIR filter and the LMS algorithm due to its simplicity, efficiency and generality. Since adaptive

FIR filters have only adjustable zeros, they are free of stability problems that can be associated with IIR filters where both poles and zeros are adjustable.

This chapter will be organized as follows: Section 5.1 briefly illustrates various applications of adaptive filters. Section 5.2 discusses the traditional sample-by-sample LMS algorithm and its disadvantages for implementing in CVP. The scalar structure of the LMS algorithm makes efficient implementation difficult. To overcome this problem, Section 5.3 and 5.4 present some techniques to vectorize the traditional LMS, namely Block LMS (BLMS) and Block Exact LMS (BELMS). We also show that the BELMS algorithms are not suitable for the CVP architecture. In contrast, the BLMS algorithm has a simple, fully vectorized structure that matches CVP. The performance of its CVP implementation is better than the performance of Motorola's AltiVec. Additionally, in Section 5.3, the implementations of BLMS in the time- and frequency-domain are also investigated.

5.1 Applications of Adaptive Filters

The most important feature of an adaptive filter is the ability to operate effectively in an unknown and changing environment as well as track time-varying characteristics of an input signal. With this ability, adaptive filters have been successfully applied in communications, control and image processing fields. Since CVP's target is the telecommunication market, it is important that adaptive filters can be efficiently implemented in CVP.

5.1.1 Adaptive Equalization

Adaptive equalization is used to eliminate the signal distortion introduced by the communication channels. The block diagram of an Adaptive equalizer is shown in Figure 5.2 [**?**, **?**].

Figure 5.2: Structure of an Adaptive equalizer

Initially, the equalizer is "trained" by transmitting a known test data sequence. By generating a synchronized version of the test signal in the receiver, the equalizer is supplied with a desired response d[n]. The estimation error is produced by subtracting the output y[n] from d[n]. This error is used to adjust the filter coefficients to their optimum values. When the training phase is completed, the adaptive equalizer tracks possible variations in the channel during transmission by using a receiver estimate of the transmitted sequence as a desire response. This receiver estimate is obtained by applying the equalizer output (y[n]) to a decision device - a Slicer.

5.1.2 Echo Cancellation

Another important application of adaptive filter is Echo cancellation. Dial-up telephone networks are used for low-volume infrequent data transmission. To provide full duplex operation, a device called a "hybrid" is used. Due to the impedance mismatch between the hybrid and the telephone channel, an "echo" is generated when there is data on the telephone lines. This unpleasant effect can be suppressed by installing an Adaptive Echo Canceller in the network.

Figure 5.3: Structure of an Adaptive Echo Canceller [**?**]

In Figure 5.3, x[n] is the far-end signal, d[n] is the echo of far-end signal plus near-end signal, y[n] is the estimated version of the echo and e[n] is the near-signal signal plus residual echo. The cancellation of the echo is achieved by estimating the echo signal and then subtracting it from the return signal.

5.1.3 Adaptive Prediction

A major application of adaptive prediction is the waveform coding of a speech signal. In this application, the adaptive filter is designed to exploit the correlation between adjacent speech samples so that the prediction error is much smaller than the speech signal itself. This prediction error is then quantized and sent to the receiver in order to reduce the number of bits required for transmission. This type of coding is called Adaptive Differential Pulse-Code Modulation (ADPCM) and provides good data rate compression.

Figure 5.4: Structure of an Adaptive Predictor

5.1.4 System Identification

System modelling is another important application of Adaptive filtering. The idea is that information about an unknown system can be retrieved by exciting the system under investigation with suitable signals and then collecting its responses. As illustrated in Figure 5.5, the unknown system is modelled by a filter whose coefficients can be adjusted by an adaptive algorithm. Both the unknown system and the filter are excited

by an input sequence x[n]. Their outputs are then compared with each other to produce estimation errors - e[n]. This estimation error represents the difference between the two systems. Intuitively, the adaptive algorithm must be able to minimize this error after some iterations. Finally, when the error has become sufficiently small, we can conclude that the resulting filter is the representation of the unknown system.

Figure 5.5: System Identification Model [**?**]

5.2 The FIR LMS Adaptive filter

The well-known LMS adaptive filter is actually a FIR digital filter of order $N-1$ in which the outputs are computed by the following equation:

$$y_k = \sum_{i=0}^{N-1} h_i(k) x_{k-i} \tag{5.1}$$

or

$$y_k = X_k^T H_k \tag{5.2}$$

where $h_i(k)$ is the i^{th} coefficient in the k^{th} iteration, x_k is an input sample and y_k is an output. X(k) and H(k) are $N \times 1$ vectors:

$$X_k = [x_k, x_{k-1}, \cdots, x_{k-N+1}]^T$$
$$H_k = [h_0(k), h_1(k), \cdots, h_{N-1}(k)]^T$$

After an iteration, the above filter is adjusted by the LMS adaptive algorithm. The detailed derivation of this adaptive algorithm is not discussed here and can be found in [**?**]. The final adaptive algorithm is described by Equation 5.3

$$H_{k+1} = H_k + 2\mu\epsilon_k X_k \tag{5.3}$$

where μ is the convergence constant and ϵ_k is the error at the k^{th} sample given by the difference between the desired output d_k and the actual output y_k

$$\epsilon_k = d_k - y_k \tag{5.4}$$

Equations 5.1, 5.3 and 5.4 completely describe the LMS adaptive filter. This sequential algorithm is not suitable for the CVP architecture because of the following reasons:

- As can be seen in Equation 5.1, only one output is calculated at a time. The only way to implement this equation is to vectorize it horizontally. As explained before, the performance of this strategy is less favorable than the Vertical strategy. For a 16-tap FIR filter, the horizontal strategy requires approximately 7 cycles to calculate one output compared with slightly more than 1 cycle/output of the vertical strategy.

- Equation 5.4 and 5.3 involve 3 scalar operations which are not suitable for the vector architecture of CVP. This scalar problem gets worse in Equation 5.4 since it needs two scalar operands. In the case of CVP, there is only one Broadcast register in AMU and thus can not be used to store 2 operands. The only solution is to store 2 scalar operands in 2 vectors. Instead of a simple scalar substraction, we have to perform a vector substraction which is a waste of resources. This complication increases the overhead of the implementation significantly.

With the above reason, it is predicted that the efficiency of the implementation will be low. For a 32-tap LMS filter, it needs approximately 8 cycles to complete Equation 5.1, 3 cycles to implement Equation 5.4 and about 15 cycles for Equation 5.3. Totally, this sequential approach requires about 30 cycles to calculate a single output. This is too inefficient to implement in CPV. In the next sections, we are going to investigate the block approaches to the LMS problem which will increase the efficiency when implemented in CVP.

5.3 The Block Least Mean Square -BLMS- algorithm

This section presents one of the most popular "block" or parallel techniques to realize LMS filter efficiently with parallel architectures. The notion of "sequential computation" was defined in [**?**]. In the context of digital filtering, sequential computation means scalar output values are calculated from the preceding inputs and outputs sequentially. Meanwhile, "block computation" means the calculation of a block of outputs from a block of inputs at a time. The main advantage of block techniques is that intermediate computations can be accomplished by using Fast Fourier Transform (FFT). Furthermore, the validity and characteristics of block techniques have already been proved [**?**, **?**, **?**, **?**, **?**].

Different from the original LMS algorithm where filter's coefficients are updated after each output calculation, in the block realization, filter's coefficients are kept unchanged during a number of samples N which is the block size. As a consequence, the behaviors of block adaptive algorithms are different from those of the traditional sample-to-sample LMS. In the case of correlated inputs, the convergence of BLMS algorithm is much slower than the LMS algorithm, resulting in larger number of iterations [**?**, **?**].

However, due to the employment of FFT, it has been proved that for filters with longer impulse response, a substantial computational gain can be achieved by implementing BLMS. For example, the ratio between the number of real multiplies required by BLMS and the number of real multiplies required by LMS is merely 0.062 when the filter length is 1024 [**?**].

Considering all of the above reasons and the algorithm's implementation simplicity, BLMS is considered to be suitable for implementation of adaptive filters in CVP. In this section, the algorithm description and its implementation in CVP is presented.

5.3.1 The BLMS algorithm

Figure 5.6 shows a general structure for a Block LMS filter.

Figure 5.6: Structure of a Block LMS filter

In the Figure 5.6, blocks of input data are formed by the Serial-Parallel box. The

matrix representation of the FIR operation is shown in Equation 5.5 where x_i is a data sample, $h_i(k)$ is the i^{th} coefficient in the k^{th} iteration, y_i is a output, L is the block size and N is the length of the filter.

$$\begin{bmatrix} y_{kL} \\ y_{kL+1} \\ \vdots \\ \vdots \\ \vdots \\ y_{(k+1)L-1} \end{bmatrix} = \begin{bmatrix} x_{kL} & x_{kL-1} & \cdots & x_{kL-N+1} \\ x_{kL+1} & x_{kL} & \ddots & x_{kL-N+2} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{(k+1)L-1} & x_{(k+1)L-2} & \cdots & x_{(k+1)L-N} \end{bmatrix} \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ h_{N-2}(k) \\ h_{N-1}(k) \end{bmatrix} \tag{5.5}$$

The implementation of this convolution in CVP was already discussed in Chapter 3. As mentioned above, the filter coefficients will be allowed to change only after a complete block is processed. Holding the weights constant during an L point block will limit the performance since, in order to keep the coefficients from too noisy, the convergence constant will be L times smaller than the constant in the original algorithm (Equation 5.6). The weight vector is updated by iterating Equation 5.3 over L points in the block. We have

$$H_{k+1} = H_k + \frac{2\mu_B}{L} \sum_{i=(k-1)L+1}^{kL} \epsilon_i X_i \tag{5.6}$$

where

$$\begin{aligned} X_k &= [x_{kL}, x_{kL-1}, \cdots, x_{kL-N+1}]^T \\ H_k &= [h_0(k), h_1(k), \cdots, h_{N-1}(k)]^T \\ Y_k &= [y_{kL}, y_{kL+1}, \cdots, y_{(k+1)L-1}]^T \end{aligned}$$

Equation 5.6 is shortened as follows

$$H_{k+1} = H_k + \frac{2\mu_B}{L}\underline{\Phi}_k = H_k + \frac{2\mu_B}{L}\mathbf{X}_k^T E_k \tag{5.7}$$

where the $\underline{\Phi}_k$ is an L element gradient estimate vector given by:

$$\begin{aligned} \underline{\Phi}_k &= [\phi_{0,k}, \phi_{1,k}, \cdots, \phi_{L-1,k}]^T \\ \phi_{i,k} &= X_{kN-i}^T E_k \\ \text{and } E_k &= [\epsilon_{kL}, \epsilon_{kL+1}, \cdots, \epsilon_{(k+1)L-1}]^T \end{aligned}$$

Clearly, the above coefficient update term ($\underline{\Phi}_k$) is a correlation operation which is implemented by the Correlation box in Figure 5.6. After calculating the updating term,

it is multiplied with the convergence constant ($\frac{2\mu_B}{L}$) and finally added up to the previous coefficient vector to finish one adaptive iteration.

The complete BLMS algorithm is time domain is summarized by the following equations

$$Y_k = \mathbf{X}_k H_k \tag{5.8}$$

$$E_k = D_k - Y_k \tag{5.9}$$

$$H_{k+1} = H_k + \frac{2\mu_B}{L}\mathbf{X}_k^T E_k \tag{5.10}$$

5.3.2 Implementation of BLMS in the time domain

In [**?**], it was proven that the algorithm is valid for any block size greater than or equal to one. However, the L equals to N case is preferred in most applications. This is because for L greater than N, the gradient estimate, which is computed over L input points, uses more input information than the filter $H(z)$ uses. This results in redundant calculations. For the cases when L is less than N, the filter length is larger than the input block being processed, which is a waste of filter weights.

In reality, the filter length varies from application to application and could be very large ($>$ 1000 taps), hence, it requires more than one iteration to complete one block computation. In this report, a case of a LMS 32-tap FIR filter will be investigated both in time and frequency domains. Data samples will be represented by 16_bit fixed_point precision. With this precision, the case of L = 32 is chosen to be implemented since it satisfies the "L=N" recommendation. Furthermore, this implementation can be applied for both the cases of small L's ($L \leq 16$) and large L's ($L \geq 16$).

The Filtering operation in Equation 5.5 is rewritten for the L = 32 case:

$$\begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ y_{k+15} \\ y_{k+16} \\ \vdots \\ y_{k+31} \end{bmatrix} = \begin{bmatrix} x_k & x_{k-1} & \cdots & \cdots & x_{k-31} \\ x_{k+1} & x_k & \ddots & \ddots & x_{k-30} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_{k+15} & x_{k+14} & \ddots & \ddots & x_{k-16} \\ x_{k+16} & x_{k+15} & \ddots & \ddots & x_{k-15} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_{k+31} & x_{k+30} & \cdots & \cdots & x_k \end{bmatrix} \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ h_{30}(k) \\ h_{31}(k) \end{bmatrix} \tag{5.11}$$

Equation 5.11 is actually an operation calculating 32 consecutive outputs of a FIR filter. After calculating 32 outputs, 32 errors are then calculated by subtracting the calculated outputs (y_k) from the desired output (d_k) :

$$\begin{bmatrix} \epsilon_k \\ \epsilon_{k+1} \\ \vdots \\ \epsilon_{k+15} \\ \epsilon_{k+16} \\ \vdots \\ \epsilon_{k+31} \end{bmatrix} = \begin{bmatrix} d_k \\ d_{k+1} \\ \vdots \\ d_{k+15} \\ d_{k+16} \\ \vdots \\ d_{k+31} \end{bmatrix} - \begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ y_{k+15} \\ y_{k+16} \\ \vdots \\ y_{k+31} \end{bmatrix} \tag{5.12}$$

In the body loop of the program, the two above steps are combined. The memory organization for the program is shown in Figure 5.7 where the errors, the updating terms (ϕ_k) and the filter coefficients are stored separately in the memory. The updating terms and the filter coefficients will be over-written every iteration.

Figure 5.7: Memory organization for the BLMS program

As explained above, the updating terms are actually the products of another convolution which will be calculated exactly in the same way as the outputs (y_k). The errors which have been stored in the memory from the previous step, now play the role of the filter coefficients.

$$\begin{bmatrix} \phi_0(k) \\ \phi_1(k) \\ \vdots \\ \vdots \\ \phi_{31}(k) \end{bmatrix} = \frac{2\mu_B}{L} \begin{bmatrix} x_k & x_{k+1} & \cdots & x_{k+31} \\ x_{k-1} & x_k & \ddots & x_{k+30} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{k-31} & x_{k-30} & \cdots & x_k \end{bmatrix} \begin{bmatrix} \epsilon_{0,k} \\ \epsilon_{1,k} \\ \vdots \\ \vdots \\ \vdots \\ \epsilon_{31,k} \end{bmatrix} \tag{5.13}$$

A small difference between Equation 5.13 and 5.11 is the transposition of the data matrix $\mathbf{X}$. To solve this ordering problem, Equation 5.13 is re-written as follows:

$$\begin{bmatrix} \phi_{31}(k) \\ \phi_{30}(k) \\ \vdots \\ \vdots \\ \phi_0(k) \end{bmatrix} = \frac{2\mu_B}{L} \begin{bmatrix} x_k & x_{k-1} & \cdots & x_{k-31} \\ x_{k+1} & x_k & \ddots & x_{k-30} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{k+31} & x_{k+30} & \cdots & x_k \end{bmatrix} \begin{bmatrix} \epsilon_{31,k} \\ \epsilon_{30,k} \\ \vdots \\ \vdots \\ \vdots \\ \epsilon_{0,k} \end{bmatrix} \tag{5.14}$$

Data matrix $\mathbf{X}$ in Equation 5.14 is now exactly the same as in Equation 5.11. In each loop, only one scalar coefficient value is read hence the transposition of the coefficient

vector does not affect the calculation.

To reverse the order of the calculated updating terms, shuffling is used. The usage of the SHUFFLE unit has already been discussed in Chapter 4

Finally, the updating terms are added to update the filter coefficients for the next iteration.

$$\begin{bmatrix} h_0(k+1) \\ h_1(k+1) \\ \vdots \\ \vdots \\ h_{31}(k+1) \end{bmatrix} = \begin{bmatrix} h_0(k) \\ h_1(k) \\ \vdots \\ \vdots \\ h_{31}(k) \end{bmatrix} + \begin{bmatrix} \phi_0(k) \\ \phi_1(k) \\ \vdots \\ \vdots \\ \phi_{31}(k) \end{bmatrix} \tag{5.15}$$

5.3.3 Implementation of BLMS in the frequency domain

In [**?**], the following properties have been concluded from the analysis of the implementation of BLMS in frequency domain:

- The time-domain block adaptive filter implemented in frequency-domain is equivalent to the frequency-domain adaptive filters (derived in the frequency domain) provided proper data sectioning. The two most common sectioning techniques are Overlap-save and Overlap-add [**?**].
- From the detailed analysis of the two most common sectioning techniques (Overlap-save and Overlap-add) in the frequency-domain, it is shown that the Overlap-save method is preferred over Overlap-add since it requires fewer computations. In the cases of fixed filters, the two methods do not make much difference. However, this is not the case for adaptive filters because the filter coefficients need to be updated after every iteration.

The convolution in Equation 5.5 can be implemented efficiently in frequency domain. To do this, the time-domain vectors are augmented and overlap-save or overlap-add method is used to ensure that circular convolution is equivalent to the desired linear convolution.

The augmented vectors are denoted in general by X_k^a, W_k^a, Y_k^a...etc but the actual contents depend on whether overlap-save or overlap-add method is used.

We also define the following notations:

$$\begin{cases} X_k(l) &= FFT\{X_k^a\} \\ H_k(l) &= FFT\{H_k^a\} \\ Y_k(l) &= FFT\{Y_k^a\} \\ E_k(l) &= FFT\{\underline{\epsilon}_k^a\} \\ \Phi_k(l) &= FFT\{\underline{\Phi}_k^a\} = E_k(l)X_k^*(l) \end{cases}$$

where the asterisk denotes complex conjugation. $FFT\{\underline{v}^a\}$ denotes the N' point Fast Fourier Transform of the elements of a $N' \times 1$ vector $\underline{v}^a$ for the discrete frequency index l.

We also define the following operation:

Definition 2 *A projection operator (F) constrains to zero all but N time_domain values corresponding to the inverse FFT (IFFT) of an $N' \times 1$ vector. We have:*

$$\begin{aligned} & \Phi_k(l) = [\Phi_k(0), \Phi_k(1), \cdots, \Phi_k(N'-1)] \\ \Phi'_k(l) \quad = \quad & F\{\Phi_k(l)\} = [\Phi_k(0), \Phi_k(1), \cdots, \Phi_k(N-1), \underbrace{0, \cdots, 0}_{N' - Nzeros}] \end{aligned}$$

then the Frequency_domain Algorithm is described by the following equations:

$$Y_k(l) = X_k(l) W_k(l) \tag{5.16}$$

$$H_{k+1}(l) = H_k(l) + \frac{2\mu_B}{L} \Phi'_k(l) \tag{5.17}$$

Figures 5.8 and 5.9 show the detailed block diagrams of the implementation of BLMS in frequency domain. We further differentiate two cases where the coefficient-updating part is implemented in the time domain (Figure 5.8) and in frequency domain (Figure 5.9).

Figure 5.8: Implementation of Block LMS FIR algorithm in the frequency domain but coefficients are updated in time-domain

Figure 5.9: The complete implementation of Block LMS FIR algorithm in frequency domain (including the coefficient updating part)

In this report, Overlap_save method is used to solve the block convolution problem. The augmented vectors will be:

$$\begin{cases} X_k^a & = & [x_{k-32}, \cdots, x_{k-1}, x_k, \cdots x_{k+31}] \\ H_k^a & = & [H_k^T, \underbrace{0, \cdots, 0}_{Nzeros}] \\ E_k^a & = & [\underbrace{0, \cdots, 0}_{Nzeros}, 0, \underline{\epsilon}_k^T] \end{cases}$$

With these augmented vector, Equation 5.16 is realized by:

$$\begin{aligned} Y_k = [y_k, \cdots, y_{k+31}]^T &= Last\ n\ terms\ of\ IFFT\{X_k(l) \odot H_k(l)\} \\ &= F\{X_k(l) \odot H_k(l)\} \end{aligned} \tag{5.18}$$

where operator $\odot$ denotes element-by-element multiplication.

Equation 5.17 for updating the coefficient vector in frequency domain is as follows:

$$H_{k+1}(l) = H_k(l) + \frac{2\mu_B}{L} FFT \begin{bmatrix} F\{E_k(l) \odot X_k^*(l)\} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{5.19}$$

where

$$F\{E_k(l) \odot X_k(l)\} = \ First\ n\ terms\ of\ IFFT\{E_k(l) \odot X_k^*(l)\} \tag{5.20}$$

Equation 5.18 and 5.19 complete the implementation of a BLMS 32-tap FIR filter in frequency domain.

In [**?**], the author proved that the number of multiplies required by the implementation of BLMS in the time-domain is much larger than in the frequency-domain when the FIR filter is long enough. This comparison is illustrated by Equation 5.21.

$$\frac{Frequency\ domain\ LMS\ real\ multiplies}{LMS\ real\ multiplies} = \frac{5(\log \frac{N}{2}) + 14}{N} \tag{5.21}$$

For different values of N, we have the following table:

n	$\frac{Frequency\ domain\ LMS\ real\ multiplies}{LMS\ real\ multiplies}$
32	1.2
64	0.69
256	0.21
1024	0.062

Table 5.1: Performance comparison between LMS in the frequency- and time-domain

Theoretically, as can be seen in Table 5.1, when the filter length is 1024, the ratio is merely 0.062. This comparison, however, is purely theoretical. From Figures 5.8 and 5.9, the most dominant parts of the implementation of BLMS in the frequency domain are the 5 FFT/IFFT blocks. Table 5.2 shows the approximate numbers of cycles required by the FFT implementation on CVP to calculate 32,64 and 1024 16-bit-real inputs [**?**].

For the cases of BLMS with block size $L = 32$, the 64-point FFT must be used. For 32 outputs, five 64-point FFT operations cost about 1250 cycles must be perform. The cost for completing the sectioning blocks is not comparable to the cost for performing FFT. However, the 1250 cycles for FFTs is still far worse than the cost of 200 cycles per 32 outputs when BLMS is implemented in the time-domain. The difference between the

Number of inputs	Number of Cycles
32	150
64	250
1024	3500

Table 5.2: Number of cycles required by CVP's FFT implementation

theoretical estimations and the approximation performances of CVP can be reasoned by the structure of FFT which requires many intermediate operations.

5.4 The Block Exact LMS -BELMS

The previous three sections have described the Block LMS algorithm and its implementation in the time and frequency domains. In spite of its ease for vectorization, the BLMS algorithm exposes many drawbacks. They are:

- As discussed in the previous section, the block size L is optimized when is equal to the filter's length N. For a number of applications where N is very large (> 1000), it requires large processing blocks. Furthermore, to apply fast algorithms like FFT (implementation of BLMS in the frequency-domain), Overlap-save or Overlap-add sectioning which are twice the filter length must be used. This again increases the block size. As the block size increases, the convergence of the algorithm gets slower (except in the case of uncorrelated inputs). As a result, it requires a large amount of computation to reach the desired state. Slow convergence speed is also a big issue for real-time applications.
- Another drawback of BLMS algorithms is presented in [**?**]. From Chapter 3, it has been presented that the heart of a FIR filtering problem is the Multiply-Accumulate operation. In reality, this operation can be implemented efficiently either in hardware or software. By applying fast algorithms to the filtering problems in the frequency-domain, we have broken the regular structure of the filtering process, namely the Multiply-Accumulate structure.

In this section, we are going to investigate two other LMS algorithms which can overcome the above drawbacks. These algorithms are mathematically equivalent to the original sequential LMS algorithm. Furthermore, they have block structures that can be implemented efficiently in parallel processors. Because of the equivalence with the original LMS algorithm and their block structures, the new algorithms are referred as Block Exact LMS - BELMS. We also differentiate BELMS1 which is described in Section 5.4.1 [**?**][**?**] and BELMS2 [**?**] discussed in Section 5.4.2.

5.4.1 Block Exact LMS 1 -BELMS1

In [**?**], the algorithm was introduced as Fast Exact LMS - FELMS. However, to emphasize the block structure of the algorithm, we refer to it as BELMS1 in this report.

Initially, let us consider the simplest case where the block size is 2. The general case will be presented at the end of this section. The original sample-by-sample LMS algorithm is described by:

$$\begin{aligned} y_k &= X_k^T H_k \\ H_{k+1} &= H_k + 2\mu\epsilon_k X_k \end{aligned}$$

or

$$\epsilon_k = d_k - X_k^T H_k \tag{5.22}$$
$$H_{k+1} = H_k + 2\mu\epsilon_k X_k \tag{5.23}$$

At time $k-1$, Equations 5.22 and 5.23 are:

$$\epsilon_{k-1} = d_{k-1} - X_{k-1}^T H_{k-1} \tag{5.24}$$
$$H_k = H_{k-1} + 2\mu\epsilon_{k-1} X_{k-1} \tag{5.25}$$

Substitute Equation 5.25 into 5.22 we obtain

$$\begin{aligned} \epsilon_k &= d_k - X_k^T H_{k-1} - 2\mu\epsilon_{k-1} X_k^T X_{k-1} \\ &= d_k - X_k^T H_{k-1} - \epsilon_{k-1} s(k) \end{aligned} \tag{5.26}$$

with $s(k) = 2\mu X_k^T X_{k-1}$.

Combining Equation 5.26 and 5.25 under matrix form results in:

$$\begin{bmatrix} \epsilon_{k-1} \\ \epsilon_k \end{bmatrix} = \begin{bmatrix} d_{k-1} \\ d_k \end{bmatrix} - \begin{bmatrix} X_{k-1}^T \\ X_k^T \end{bmatrix} H_{k-1} - \begin{bmatrix} 0 & 0 \\ s(n) & 0 \end{bmatrix} \begin{bmatrix} \epsilon_{k-1} \\ \epsilon_k \end{bmatrix} \tag{5.27}$$

or

$$\begin{bmatrix} 1 & 0 \\ s(n) & 1 \end{bmatrix} \begin{bmatrix} \epsilon_{k-1} \\ \epsilon_k \end{bmatrix} = \begin{bmatrix} d_{k-1} \\ d_k \end{bmatrix} - \begin{bmatrix} X_{k-1}^T \\ X_k^T \end{bmatrix} H_{k-1} \tag{5.28}$$

hence

$$\begin{bmatrix} \epsilon_{k-1} \\ \epsilon_k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -s(n) & 1 \end{bmatrix} \left[\begin{bmatrix} d_{k-1} \\ d_k \end{bmatrix} - \begin{bmatrix} X_{k-1}^T \\ X_k^T \end{bmatrix} H_{k-1} \right]. \tag{5.29}$$

The above equation is the block calculation for two errors. From the $L = 2$ case, we can generalize to the case in which the block size is L.

$$E_k = [S(k) + I]^{-1}[D_k - \mathbf{X}_k H(k - L + 1)] \tag{5.30}$$
$$H(k+1) = H(k - L + 1) + 2\mu\mathbf{X}^T(k)E_k \tag{5.31}$$

or

$$E_k = G(k)[D_k - \mathbf{X}_k H(k-L+1)] \tag{5.32}$$
$$H(k+1) = H(k-L+1) + 2\mu \mathbf{X}^T(k) E_k \tag{5.33}$$

where

$$E_k = [\epsilon_{k-L+1}, \epsilon_{k-L+2}, \cdots, \epsilon_k] \tag{5.34}$$
$$G(k) = [S(k) + I]^{-1} \tag{5.35}$$

$$S(k) = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ s_1(k-L+2) & 0 & \cdots & 0 \\ s_2(k-L+3) & s_1(k-L+3) & & 0 \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ s_{L-1}(k) & s_{L-2}(k) & \cdots s_1(k) & 0 \end{bmatrix} \tag{5.36}$$

$$s_p(k-q) = \mu X^T(k-q) X(k-q-p) \tag{5.37}$$

This approach has the advantage of converting the sequential LMS into an fixed FIR filter which can be efficiently implemented in parallel architectures, plus a small correction. Furthermore, the results of this algorithm are exactly the same as the traditional LMS.

Nevertheless, the main problem for implementing the algorithm in CPV is the matrix inversion $G(k) = [S(k) + I]^{-1}$. The problem gets worse when the block sizes increase. For the cases where the block sizes are small ($L \leq 4$), the matrix inversion can be calculated in advance, outside the program. In [**?**], the author successfully implemented the BELMS1 algorithm in the StarCore SC140. This processor has four ALUs and can compute four MACs in one cycle. With this reason, the block size was chosen to be 4 and $G(k)$ was a 4×4 matrix. To calculate $G(k)$, the matrix inversion was calculated manually outside the actual program. Each element - $s_p(k-q)$ - was calculated and plugged in to have the full inverted matrix. In this way, the calculation overhead of the updating terms is minimized. In other words, this algorithm is especially suitable for the case when the filter is very long and the block size is small.

The above technique, however, can not be applied to CVP. Because CVP can handle 16 MACs (with 16-bit precision) at a time, the best block size should be 16 which is too long for converting a matrix either manually or by a program.

5.4.2 Block Exact LMS 2 -BELMS2

The outputs are calculated as follows:

$$\begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ y_{k+15} \end{bmatrix} = \begin{bmatrix} X_k \\ X_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ X_{k+15} \end{bmatrix} \odot \begin{bmatrix} H_k^T \\ H_{k+1}^T \\ \vdots \\ \vdots \\ \vdots \\ H_{k+15}^T \end{bmatrix} \tag{5.38}$$

From the standard algorithm, the filter coefficients are updated every iteration.

$$H_k = H_{k-1} + \mu\epsilon_{k-1}X_{k-1} \tag{5.39}$$

or

$$\begin{aligned} H_{k+n} &= H_{k+n-1} + \mu\epsilon_{k+n-1}X_{k+n-1} \\ &= H_{k+n-2} + \mu\epsilon_{k+n-2}X_{k+n-2} + \mu\epsilon_{k+n-1}X_{k+n-1} \\ &= \cdots \\ &= \cdots \\ &= H_{k-1} + \mu\textstyle\sum_{i=-1}^{n-1}\epsilon_{k+i}X_{k+i} \end{aligned} \tag{5.40}$$

Equation 5.38 can now be rewritten as follows:

$$\begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ y_{k+15} \end{bmatrix} = \begin{bmatrix} X_k \\ X_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ X_{k+15} \end{bmatrix} H_{k-1}^T + \begin{bmatrix} X_k \\ X_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ X_{k+15} \end{bmatrix} \odot \begin{bmatrix} \mu\sum_{i=-1}^{-1}\epsilon_{k+i}X_{k+i}^T \\ \mu\sum_{i=-1}^{0}\epsilon_{k+i}X_{k+i}^T \\ \vdots \\ \vdots \\ \vdots \\ \mu\sum_{i=-1}^{14}\epsilon_{k+i}X_{k+i}^T \end{bmatrix}$$

or

$$Y = A + \mathbf{X} \odot B \tag{5.41}$$

The block update is derived form Equation 5.40 by setting n = 15. We have:

$$H_{k+15} = H_{k-1} + \mu \sum_{i=-1}^{14} \epsilon_{k+i}X_{k+i} \tag{5.42}$$

Equation 5.41 together with Equation 5.42 define the exact presentation of LMS FIR filter.

As can be seen in Equation 5.41, the term A is a fixed FIR operation on a fixed filter, i.e. H_{k-1}. To obtain B, we notice that each of its element is actually an intermediate result of another FIR operation, i.e. the last term in B.

We have

$$\mu \sum_{i=-1}^{14} \epsilon_{k+i} X_{k+i} = \mu \Delta H_k \tag{5.43}$$

where

$$\Delta H_k = [\Delta h_0, \Delta h_1, \cdots, \Delta h_{N-1}]$$

Each element of ΔH_k is calculated by the following equation

$$\begin{bmatrix} \Delta h_0 \\ \Delta h_1 \\ \vdots \\ \vdots \\ \Delta h_{N-1} \end{bmatrix} = \epsilon_{k-1} * \begin{bmatrix} x_{k-1} \\ x_{k-2} \\ \vdots \\ \vdots \\ x_{k-N} \end{bmatrix} + \epsilon_k * \begin{bmatrix} x_k \\ x_{k-1} \\ \vdots \\ \vdots \\ x_{k-N+1} \end{bmatrix} + \cdots + \epsilon_{k+14} * \begin{bmatrix} x_{k+14} \\ x_{k+13} \\ \vdots \\ \vdots \\ x_{k+15-N} \end{bmatrix}$$

or

$$\Delta h_j = h_{j+1} - h_j = \sum_{i=k-1}^{k+14} \epsilon_i x_{i-j} \tag{5.44}$$

The above equation is a FIR operation of the inputs x_{k+i} on a FIR filter with the error vector as the filter coefficients. This algorithm can overcome the matrix inversion problem as present on BELMS1. Unfortunately, this algorithm has additional difficulties for its implementation on CVP.

- The initial errors ($\epsilon_0 - \epsilon_{14}$) are required to start the algorithm. This can be done via some approximation. After some iterations, the algorithm converges to the exact value.

- This algorithm does not allow the "vertical parallelization" like in the case of FIR filter. The calculation of Equation 5.41 is a row-wise vector multiplication between two 16×16 matrices. As discussed in Chapter 3, each vector multiplication in CVP requires Scaling, Intra-Add which will decrease the algorithm efficiency.

5.5 Performance and Conclusions

With the above reasoning, we can conclude that the two Block Exact LMS algorithms are not suitable for CVP. However, the BELMS2 algorithm will be promising in case CVP could handle better the Vector-dot or the Multiply-sum operation. Moreover, this improvement would also increase the performance of the Horizontal strategy for FIR filter and of the Strategy 1 for Decimation filter. The Vector-dot and the Multiply-accumulate operation are two basic operations that find application in many digital signal processing algorithms. The Vector-dot operation is shown in Figure 5.10.

Figure 5.10: Vector-dot operation on two 16-element vectors

Given two vectors $X = [x_0, x_1, x_2, \cdots, x_{N-1}]$ and $Y = [y_0, y_1, y_2, \cdots, y_{N-1}]$, the vector-dot product z of X and Y is defined as:

$$z = X \otimes Y = x_0y_0 + x_1y_1 + \cdots + x_{N-1}y_{N-1}$$

CVP requires three instructions to implement the above operation. Firstly, multiply (MUL) or multiply-accumulate (MAC) instructions are required to perform the element-wise multiplication. After that, scaling (ASRA) is needed to reduce the number of bits representing the previous multiplication results. Finally, the separate products (x_iy_i) are added by applying the Intra-Add instruction (IADD). This structure can be improved. For example, the Motorola's Altivec architecture requires only two instructions to complete this task. Altivec's single "VEC_MSUMS"(Multiply and Sum) instruction includes both a MAC and a Scaling operations. Figure 5.11 presents a Multiply-Accumulate model that is integrated with both scaling and rounding operations. The CVP architecture can provide a multiply instruction for the first two steps, i.e. Multiply and Scale, and a MAC instruction for the whole operation (Multiply, Scale and Accumulate).

Figure 5.11: The Multiply-Accumulate model [**?**]

Contrary to BELMS algorithms, the Block LMS algorithm has a parallel structure that matches the CVP's architecture. In Table 5.3, the number of clock cycles required by CVP and AltiVec to calculate 128, 256, 512 and 1024 outputs are presented. The middle column shows the performance of the CVP in numbers of cycles required and

the right column is for the AltiVec implementation as discussed in [**?**]. We further note that the AltiVec's figures in Table 5.3 are obtained by an implementation of BLMS on a 24-tap FIR filter. Meanwhile, the implementation in CVP uses 32-tap FIR filter. This detail further facilitates the performance of CVP in comparison with Altivec.

	CVP (cycles)	Altivec (cycles)
128	762	1493
256	1514	2933
512	3018	5813
1024	6026	11573

Table 5.3: Comparison between CVP's and Altivec's performance on the Block LMS algorithm

6 Conclusions

In this thesis, three digital filtering algorithms, i.e. FIR, Decimation and Adaptive filtering have been investigated, vectorized and implemented on the CVP architecture. In this chapter, the main results achieved will be summarized in Section 6.1. In addition, Section 6.2 provides some possible directions for the future research.

6.1 Thesis's Contributions

The main contributions of this thesis are:

1. We explored and studied various digital filtering algorithms in details. Different approaches to vectorize the algorithms have been investigated and implemented with CVP. Although some specific study cases were used during the thesis to demonstrate the vectorization strategies and the implementations, we have always tried to preserve the generality of the original problems. In other words, the discussed vectorization can be applied for many cases in real applications. This discussion on each filter algorithm is as follows:

   (a) The Vertical vectorization strategy for FIR filters not only guarantees the efficiency of its implementations but it can also be applied for FIR filters of arbitrary lengths. Although the number of inputs (or outputs) for the strategy should be a multiple-of-16 number, it does not affect the generality since we can always use zero-padding to extend the length of the input sequence.

   (b) Three vectorization approaches for Decimation and Interpolation filters with small factors were investigated in Chapter 4. We have shown that decimation filters with arbitrary large factors can be decomposed into multiple stages of smaller decimation filters with smaller factors. With this reason, the three strategies can be generalized for decimation filters with arbitrary factors.

   (c) Adaptive filtering was the most difficult to vectorize due to the data-dependent structure of the algorithm. Various approaches have been studied but none of them is a "perfect" match for CVP. Although the implementation of BLMS has proved to be very efficient and general, it has a serious drawback. This algorithm has slow convergence speed which may not be bearable for many applications (especially for real-time applications). More study will have to be performed in order to find a good solution for the problem.

2. We implemented the investigated strategies, i.e. Horizontal and Vertical strategies for FIR filters, Strategies 1, 2 and 3 for Decimation filters and BLMS algorithm with CVP. The performance of the implementations have partially shown the potential of CVP in comparison with other architectures in the market.

3. Through the above algorithmic investigations, we also attempted to investigate the necessary modifications in the CVP architecture so that the its Instruction Set Architecture become more efficient for the digital filtering problems. The suggestions are as follows:

   - The performance of CVP on the Vector-Dot problem will be greatly improved if we can integrate the Scaling to Multiply or Multiply-accumulate instruction (as discussed in Section 5.5).
   - To increase the precision of the results after Scaling, a Rounding-Constant - 0x80000000 in the cases of 32-bit precision, 0x8000 in the cases of 16-bit precision or 0x80 in the cases of 8-bit precision - should be added before scaling. This operation should be integrated to the Scaling instruction.
   - The SHUFFLE instruction needs to be a "full" operation. In other words, the architecture should be able to shuffle a full vector with a single instruction. At this moment, a full vector shuffle requires an ODD and an EVEN shuffles to be completed. Furthermore, additional pattern registers (sfuc) will reduce the number of LOAD operations required and thus reduce the number of cycles required.

6.2 Future work

Even though we have accomplished significant insight towards the vectorization of different digital filtering algorithms for CVP's architecture, a lot of future work remains unaddressed. The following are some of the key areas for future research based on the results of this thesis:

- **Further improving the vectorization efficiency of the Adaptive filtering algorithms.**

  As discussed above, our work in this thesis has resulted in several efficient approaches to vectorize FIR, Decimation and Adaptive filters. However, there are many topics that are worth further investigation. Among the vectorization of the three filters, the results for Adaptive filtering were especially below the expectation. Some other approaches that deserve investigation are:

  - Closer investigation on **Block Exact** approaches. Since the Block Exact methods have exactly the same responses as the original algorithm, attempts to implement these approaches will result in computation gains while preserving the algorithms' important properties, e.g. the speed of convergence.
  - Implementation of approximate **Block** algorithms in the Frequency-domain is also an alternative that can be efficient when the filter's lengths are increased. This approach, however, requires efficient implementations of Fast Fourier Transform (FFT).

- **Further studying the vectorization of Interpolation filtering algorithms.**

In Chapter 4, we have investigated the vectorization of Decimation filters carefully. However, the vectorization of Interpolation filter has not fully studied. Although the Strategy 2 can be used for both Decimation and Interpolation filters, Strategy 1 and 3 have only been applied for Decimation filters. Further study A good solution for Interpolation filters will be an improvement for CVP for application from the Multirate Signal Processing field.

- **Continuing to explore other important digital filtering algorithms**

  In this thesis, only three important filtering algorithms, i.e. FIR, Decimation and Adaptive filtering have been studied. With the aim is the Telecommunication markets, others filtering algorithms such as IIR or different algorithm for Adaptive filtering (RLS, NLMS etc.) are envisioned to be important for CVP.

Programs for FIR filters

A.1 Matlab program for results comparison

```
%-------------------------------------------------------------------------
% This script is for the comparison purpose. The filtering result generated
% in Matlab of a 32-tap filter will be compared with that generated using
% CVP architecture. Both filter coefficients and data are converted to
% 16-bit fixed point representation.
%-------------------------------------------------------------------------

b_float = fir1(31,0.5);
x_float=[0.5979,0.9492,0.2888,0.8888,0.1016,0.0653,0.2343,0.9331,0.0631,0.2642,
0.9995,0.2120,0.4984,0.2905,0.6728,0.9580,0.7666,0.6661,0.1309,0.0954,0.0149,
0.2882,0.8167,0.9855,0.0174,0.8194,0.6211,0.5602,0.2440,0.8220,0.2632,0.7536,
0.6596,0.2141,0.6021,0.6049,0.6595,0.1834,0.6365,0.1703,0.5396,0.6234,0.6859,
0.6773,0.8768,0.0129,0.3104,0.7791,0.3073,0.9267,0.6787,0.0743,0.0707,0.0119,
0.2272,0.5163,0.4582,0.7032,0.5825,0.5092,0.0743,0.1932,0.3796,0.2764,0.7709,
0.3139,0.6382,0.9866,0.5029,0.9477,0.8280,0.9176,0.1131,0.8121,0.9083,0.1564,
0.1221,0.7627,0.7218,0.6516,0.7540,0.6632,0.8835,0.2722,0.4194,0.2130,0.0356,
0.0812,0.8506,0.3402,0.4662,0.9138,0.2286,0.8620,0.6566,0.8912];

output_matlab=fftfilt(b_float,x_float);          % Matlab's output

b_fixed = fixedpt_scale(b_float);
x_fixed = fixedpt_scale(x_float);

y1 = fftfilt(b_fixed, x_fixed);

% Estimated CVP's output with rounding constant

output_cvp = (y1+2^15)/2^30;

dummy=textread('test1.dmp','%s','whitespace','[ ( , ) ]');
count=2; new=[]; for k=1:6
   new=[new;dummy(count:(count+15))];
   count=count+17;
end res=str2num(char(new)); result=[]; for m=1:48
    for n=1:2
        temp(n+(m-1)*2)=res(3-n+(m-1)*2);
    end
end for i=1:6
```

```
    for j=1:16
    result(j+(i-1)*16)=temp(17-j+(i-1)*16)/2^15;
    end
end

different1 = output_matlab - output_cvp;
different2 = output_matlab - result;

%------------------------------------------------------------------------
%  Plot results
%------------------------------------------------------------------------

subplot(3,1,1);
plot(output_matlab,'r');
hold on;
plot(output_cvp,'b');
hold on;
plot(result,'g');

title('Filtering results of a 32-tap FIR filter calculated by
MATLAB and CVP');
ylabel('output caculated by MATLAB');
xlabel('n');
legend('by MATLAB', 'rounding added', 'by CVP',4);

subplot(3,1,2);
plot(different2);
title('Difference between the actual result and CVPs result');

subplot(3,1,3);
plot(different1);
title('Difference between the predicted result and CVPs result');
```

A.2 Vertical Implementation

```
.data

.address 0.0

.align   16

.dint zeros=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
.dint data0=[31391,22045,9519,16332,6946,32753,8658,2069,30576,7678,2140,3329,29125,9463,31102,19592]
.dint data1=[24695,8625,26936,7996,18357,20353,26850,569,32292,26763,9444,487,3126,4291,21828,25118]
.dint data2=[25529,10171,422,28732,22195,22475,20427,17682,5581,20858,6008,21611,19823,19730,7014,21615]
.dint data3=[9058,12439,6332,2434,16686,19087,23043,15014,16917,7443,391,2316,2435,22240,30365,10069]
.dint data4=[21353,23652,24991,4002,5124,29762,26612,3705,30067,27133,31054,16478,32328,20912,10287,25260]
.dint data5=[29202,21516,28247,7490,29942,15275,11148,27871,2660,1167,6979,13744,8918,28950,21730,24708]
```

```
.qint acucfg0 = [data5, 0, -2, 0,  coef0, 0, 2, 64]
.qint acucfg1= [coef1, 0, 32, 0, 0, 0, 0, 0]

.dint coef0=[14747,4823,-2786,-1877,1349,998,-745,-555,409,296,-209,-143,96,64,-46,-38]
.dint coef1=[-38,-46,64,96,-143,-209,296,409,-555,-745,998,1349,-1877,-2786,4823,14747]

.code

.stint  FIR_32

/*-------------------------------------------------------------------------*/
/*                          POINTERS INITIALIZATION                        */
/*-------------------------------------------------------------------------*/

VMU:: LDACUS(acu0, acucfg0);         /* acu0 - data5, acu1 - coef0 */

VMU:: LDBASE(acu4,960);              /* Initialize acu4 */

VMU:: LDINCR(acu4,1.0);

IDU:: loop(OUTER_LOOP,6);

VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
AMU:: SUBdi(amu3,amu3),RCV(amu,amu3);

IDU:: loop(30, LOOP_END)
VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
AMU:: RCV(vmu,amu0),SRCV(vmu);

LOOP_END: VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
          AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

VMU:: LDINCR(acu0,96)
AMU:: MACdi(amu0,bcst,acc1);

NOP;

AMU:: LSRAdi(acc1,15);

VMU:: SENDV(*acu0++)
AMU:: ADDdi(amu2,amu1);

VMU:: RCVL_AMU(*acu4++)
AMU:: SUBdi(amu0,amu0),RCV(amu,amu0);

OUTER_LOOP:  VMU:: LDINCR(acu0,-2)
             AMU:: SUBdi(amu2,amu2),RCV(amu,amu2);
NOP;
```

```
IDU:: EOS();
```

A.3 Horizontal Implementation

```
.qint acucfg0 = [0,0,0,coef0,0,2,2,zeros1]

.qint acucfg1 =[0,0,0,coef1,0,2,2,zeros2]

.dint coef0=[14747,4823,-2786,-1877,1349,998,-745,-555,409,296,-209,-143,96,64,-46,-38]
.dint coef1=[-38,-46,64,96,-143,-209,296,409,-555,-745,998,1349,-1877,-2786,4823,14747]

.code

int memwrite=960;

.stint  Decimation

/*-------------------------------------------------------------------------*/
/*                          POINTERS INITIALIZATION                        */
/*-------------------------------------------------------------------------*/

VMU:: LDACUS(acu0, acucfg0);

VMU:: LDACUS(acu2, acucfg1);

VMU:: LDBASE(acu4,memwrite);                /* Initialize acu4 */

VMU:: LDINCR(acu4,4);

VMU:: SENDV(*acu1);

VMU:: SENDV(*acu3) AMU:: RCV(vmu,amu2);     /* amu2 contains coef0 */

VMU:: SENDV(*acu0++) AMU:: RCV(vmu,amu3);   /* amu3 contains coef1 */

/*-------------------------------------------------------------------------*/

IDU:: loop(LOOP_END,96)
VMU:: SENDV(*acu2++)
AMU:: RCV(vmu,amu0);

AMU:: MACdi(amu0,amu2,acc3),RCV(vmu,amu1);

AMU:: MACdi(amu1,amu3,acc3);
NOP;

AMU:: ASRAdi(acc3,15);
```

```
AMU:: DIADDdi(amu6,16),RCV(amu,amu6);

VMU:: SENDV(*acu0++)
SRU:: RCV(amu,sru0);

AMU:: RCV(vmu,amu0),SUBdi(amu7,amu7),RCV(amu,amu7)
SRU:: SHIFT0(sru0,qword);

LOOP_END: VMU:: SENDV(*acu2++), SRCV(sru,*acu4++)
          AMU:: SUBdi(amu6,amu6),RCV(amu,amu6);

/*-------------------------------------------------------------------------------------------

NOP;

IDU:: EOS();
```

A.4 CVP-C program for FIR filter

```
#include "fxp.h"
#include "vmu_mem.h"
#include "cvp_acu.h"
#include "cvp_fu.h"

#define readl(name,var,addr) readl_##name(var,addr)
#define readv(name,var,addr) readv_##name(var,addr)
#define writel(name,addr,var) writel_##name(addr,var)
#define read_w(name,var,addr) read_w_##name(var,addr)
#define read_dw(name,var,addr) read_dw_##name(var,addr)
#define read_qw(name,var,addr) read_qw_##name(var,addr)
#define write_w(name,addr,var) write_w_##name(addr,var)
#define write_dw(name,addr,var) write_dw_##name(addr,var)
#define write_qw(name,addr,var) write_qw_##name(addr,var)

#define P_Q 8 #define P_W 32

#define INPUT_BASE  (address_t)(32)
#define ZEROS (address_t)(0)
#define OUTPUT_BASE  (address_t)(960)
#define COEFF_BASE (address_t)(256)

void fir_32()

#define CONST_1x2 (address_t)(-2) #define CONST_2   (address_t)(2)
#define BOUND     (address_t)(2*32) #define INC
(address_t)(1*32)
```

```
  address_t j = 0;
  address_t n = 0;
  address_t k = 0;
  address_t p = 0;
  address_t val = 15;
  address_t bound_1 = BOUND;
  address_t bound_0 = 0;
  address_t inc_1 = INC;
  address_t const_1x2_1 = CONST_1x2;
  address_t const_1x2_2 = CONST_2;
  address_t input = INPUT_BASE;
  const address_t output = OUTPUT_BASE;
  const address_t coeff = COEFF_BASE;

  address_t j_addr, n_addr, k_addr;
  vector_t acc0, acc1, temp1;

     scalar_t s;
     vector_t h;
     jindex_1: Cvp_acuAddMod(j_addr, j, input, j, const_1x2_1, bound_0);
     rd1:  readv(input, h, j_addr);
     kindex_1: Cvp_acuAddMod(k_addr, k, coeff, k, const_1x2_2, bound_1);
     rs2:  read_w(coef, s, k_addr);
        fir_loop: for (unsigned int i = 0; i < 31; i++)
                {

                        amuv_MACdi0(acc0, acc1, h, s);
                        jindex_0: Cvp_acuAddMod(j_addr, j, input, j, const_1x2_1, bound_0);
                        rd0:  readv(input, h, j_addr);
                        kindex_0: Cvp_acuAddMod(k_addr, k, coeff, k, const_1x2_2, bound_1);
                        rs1:  read_w(coef, s, k_addr);

                }
    amuv_LSRAdi0(acc0, acc1, 15);
    amuv_SND_1(temp1, acc1);
    nindex_1: Cvp_acuAddMod(n_addr, n, output, n, inc_1, 0);
    wr1: writel(output, n_addr, temp1);

}
```

Programs for Decimation filters B

B.1 Matlab program for Decimation filters

```
x_fixed = [19592,31103,9463,29124,3329,2140,7678,30576,2068,8657,32752,6947,
16332,9519,22046,31392,25120,21827,4289,3126,488,9444,26762,32293,570,26850,
20352,18357,7995,26935,8625,24694,21614,7016,19730,19821,21610,6010,20857,5580
,17682,20428,22476,22194,28731,423,10171,25530,10070,30366,22240,2435,2317,390,
7445,16918,15014,23042,19087,16685,2435,6331,12439,9057,25261,10286,20913,32329,
16479,31054,27132,30068,3706,26611,29763,5125,4001,24992,23652,21352,24707,21732,
28951,8919,13743,6980,1167,2661,27872,11148,15276,29943,7491,28246,21515,29203];

coef0 = [-38,64,-143,296,-555,998,-1877,4823,14747,-2768,1349,-745,409,-209,96,-46];

coef1 = [-46,96,-209,409,-745,1349,-2786,14747,4823,-1877,998,-555,296,-143,64,-38];

coef = [-38,-46,64,96,-143,-209,296,409,-555,-745,998,1349,-1877,-2786,4823,14747
,14747,4823,-2786,-1877,1349,998,-745,-555,409,296,-209,-143,96,64,-46,-38];

result_matlab = [];

result_temp = fftfilt(coef,x_fixed)/2^15;

for k=1:48
    result_matlab(k)=result_temp(k*2);
end

dummy=textread('decimation_ver2.dmp','%s','whitespace','[ ( , ) ]');
count=2;
new=[];
for k=1:3
   new=[new;dummy(count:(count+15))];
   count=count+17;
end res=str2num(char(new));
result=[];
for m=1:24
    for n=1:2
        temp(n+(m-1)*2)=res(3-n+(m-1)*2);
    end
end

for i=1:3
```

```
    for j=1:16
    result(j+(i-1)*16)=temp(17-j+(i-1)*16)/2^15;
    end
end

dummy=textread('decimation.dmp','%s','whitespace','[ ( , ) ]');
count=2;
new=[];

for k=1:6
   new=[new;dummy(count:(count+15))];
   count=count+17;
end

res=str2num(char(new));

res1=[]; for k=1:48
    res1(k)=res(2*k);
end

for m=1:6
    for n=1:8
        temp(n+(m-1)*8)=res1(9-n+(m-1)*8);
    end
end

result1 = temp/2^15;

dummy=textread('decimation_ver3.dmp','%s','whitespace','[ ( , ) ]');
count=2;
new=[];

for k=1:6
   new=[new;dummy(count:(count+15))];
   count=count+17;
end

res=str2num(char(new));
res1=[];

for k=1:48
    res1(k)=res(2*k);
end

for m=1:6
    for n=1:8
        temp(n+(m-1)*8)=res1(9-n+(m-1)*8);
    end
end

result2 = temp/2^15;
```

```
different = result_matlab/2^15 - result;
different1 = result_matlab/2^15 - result1;
different2 = result_matlab/2^15 - result2;
%------------------------------------------------------------------------
%  Plot results
%------------------------------------------------------------------------

subplot(2,1,1);
plot(result_matlab/2^15,'r');
hold on;
plot(result1,'g');
hold on;
plot(result,'b');
hold on;
plot(result2, 'm');

title('Decimation filter with factor of 2');
legend('by MATLAB', 'Strategy 1', 'Strategy 2', 'Strategy3', 4);
subplot(2,1,2);
plot(different1, 'r');
hold on;
plot(different, 'g');
hold on;
plot(different2, 'm');

title('Difference between the actual result and CVPs results');
legend('Between Matlab & Strategy 1', 'Between Matlab & Strategy 2', 'Between Matlab & Strate
```

B.2 Implementation of Parallelization Strategy 1

```
.data

.address 0.0
.align   16

.qint acucfg0 = [0,0,0,coef0,0,4,2,zeros1]
.qint acucfg1 = [0,0,0,coef1,0,4,2,zeros2]

.dint coef0=[14747,4823,-2786,-1877,1349,998,-745,-555,409,296,-209,-143,96,64,-46,-38]
.dint coef1=[-38,-46,64,96,-143,-209,296,409,-555,-745,998,1349,-1877,-2786,4823,14747]

.code

int memwrite=960;

.stint  Decimation
```

```
/*-------------------------------------------------------------------------*/
/*                          POINTERS INITIALIZATION                        */
/*-------------------------------------------------------------------------*/

VMU:: LDACUS(acu0, acucfg0);

VMU:: LDACUS(acu2, acucfg1);

VMU:: LDBASE(acu4,memwrite);              /* Initialize acu4 */

VMU:: LDINCR(acu4,4);

VMU:: SENDV(*acu1);

VMU:: SENDV(*acu3)
AMU:: RCV(vmu,amu2);                      /* amu2 contains coef0 */

VMU:: SENDV(*acu0++)
AMU:: RCV(vmu,amu3);                      /* amu3 contains coef1 */

/*-------------------------------------------------------------------------*/

IDU:: loop(LOOP_END,48)
VMU:: SENDV(*acu2++)
AMU:: RCV(vmu,amu0);                      /* amu0 contains the low-vector */

AMU:: MACdi(amu0,amu2,acc3),RCV(vmu,amu1); /* amu1 contains the high-vector */

AMU:: MACdi(amu1,amu3,acc3);
NOP;

AMU:: ASRAdi(acc3,15);

AMU:: DIADDdi(amu6,16),RCV(amu,amu6);

VMU:: SENDV(*acu0++)
SRU:: RCV(amu,sru0);

AMU:: RCV(vmu,amu0),SUBdi(amu7,amu7),RCV(amu,amu7)
SRU:: SHIFT0(sru0,qword);

LOOP_END: VMU:: SENDV(*acu2++), SRCV(sru,*acu4++)
          AMU:: SUBdi(amu6,amu6),RCV(amu,amu6);

/*-----------------------------------------------------------------------------------------*/

NOP;

IDU:: EOS();
```

B.3 Implementation of Parallelization Strategy 2

```
.int SFU_PAT1=[-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,29,28,25,24,21,20,17,16,13,12,9,8,5,4,1,0]
.int SFU_PAT2=[29,28,25,24,21,20,17,16,13,12,9,8,5,4,1,0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]

.int SFU_PAT3=[-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,31,30,27,26,23,22,19,18,15,14,11,10,7,6,3,2]
.int SFU_PAT4=[31,30,27,26,23,22,19,18,15,14,11,10,7,6,3,2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]


.dint coef0=[-46,96,-209,409,-745,1349,-2768,14747,4823,-1877,998,-555,296,-143,64,-38]
.dint coef1=[-38,64,-143,296,-555,998,-1877,4823,14747,-2768,1349,-745,409,-209,96,-46]


.qint acu0cfg =  [64,32,0,SFU_PAT1,0,32,0,data0]
.qint acu1cfg = [64,32,0,SFU_PAT3,0,32,0,data0]
.qint acu2cfg = [32,2,0,coef0,0,-2,0,30.0]
.qint acu3cfg = [32,2,0,coef1,0,-2,-2,60.0]

.code

int memwrite1 = 960;
int memwrite2 = 1920;

.stint  decimation

VMU:: LDACUS(acu0, acu0cfg);

VMU:: LDACUS(acu2, acu1cfg);

VMU:: LDBASE(acu4,memwrite1);

VMU:: LDINCR(acu4,1.0);

VMU:: LDBASE(acu6,memwrite2);

VMU:: LDINCR(acu6,1.0);

VMU:: LDBASE(acu5,memwrite2);

IDU:: loop(Loop_end,3)
VMU:: LDINCR(acu5,1.0);

/*-------------------------------------------------------------------------*/

VMU:: SENDL(*acu1++);

VMU:: SENDL(*acu1++)
SFU:: CONF(sfuc0);
```

```
VMU:: SENDL(*acu0++)
SFU:: CONF(sfuc1);

VMU:: SENDL(*acu0++)
SFU:: ODD(sfuc0),RCV(vmu);

SFU:: EVEN(sfuc0),RCV(vmu);

SFU:: ODD(sfuc1);

SFU:: EVEN(sfuc1);

Loop_end: VMU:: RCVL_SFU(*acu4++);

/*-------------------------------------------------------------------------*/

IDU:: loop(Loop_end1,3);

VMU:: SENDL(*acu3++);

VMU:: SENDL(*acu3++)
SFU:: CONF(sfuc0);

VMU:: SENDL(*acu2++)
SFU:: CONF(sfuc1);

VMU:: SENDL(*acu2++)
SFU:: ODD(sfuc0),RCV(vmu);

SFU:: EVEN(sfuc0),RCV(vmu);

SFU:: ODD(sfuc1);

SFU:: EVEN(sfuc1);

Loop_end1: VMU:: RCVL_SFU(*acu5++);

VMU:: LDACUS(acu0, acu2cfg);           /* acu0 - data5, acu1 - coef0 */

IDU:: loop(OUTER_LOOP,3);

/*-------------------------------------------------------------------------*/

VMU:: SENDV(*acu0++),SSND(*acu1++,dword) AMU::
SUBdi(amu3,amu3),RCV(amu,amu3);

IDU:: loop(14, LOOP_END)
VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
AMU:: RCV(vmu,amu0),SRCV(vmu);

LOOP_END: VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
```

```
          AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

VMU:: LDINCR(acu0,64)
AMU:: MACdi(amu0,bcst,acc1);
/*-------------------------------------------------------------------------*/
NOP;

AMU:: LSRAdi(acc1,15);

VMU:: SENDV(*acu0++)
AMU:: ADDdi(amu2,amu1);

VMU:: RCVL_AMU(*acu4++)
AMU:: SUBdi(amu0,amu0),RCV(amu,amu0);

OUTER_LOOP:  VMU:: LDINCR(acu0,-2)
             AMU:: SUBdi(amu2,amu2),RCV(amu,amu2);

VMU:: LDACUS(acu0, acu3cfg);           /* acu0 - data5, acu1 - coef0 */

VMU:: LDBASE(acu5,30.0);
VMU:: LDOFFS(acu5,0);

IDU:: loop(OUTER_LOOP1,3);

/*-------------------------------------------------------------------------*/

VMU:: SENDV(*acu0++),SSND(*acu1++,dword) AMU::
SUBdi(amu3,amu3),RCV(amu,amu3);

IDU:: loop(14, LOOP_END1)
VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
AMU:: RCV(vmu,amu0),SRCV(vmu);

LOOP_END1: VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
           AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

VMU:: LDINCR(acu0,64)
AMU:: MACdi(amu0,bcst,acc1);
/*-------------------------------------------------------------------------*/
NOP;

AMU:: LSRAdi(acc1,15);

VMU:: SENDV(*acu0++)
AMU:: ADDdi(amu2,amu1);
```

```
VMU:: RCVL_AMU(*acu5++)
AMU:: SUBdi(amu0,amu0),RCV(amu,amu0);

OUTER_LOOP1:  VMU:: LDINCR(acu0,-2)
              AMU:: SUBdi(amu2,amu2),RCV(amu,amu2);

/*-------------------------------------------------------------------------*/

VMU:: LDOFFS(acu4,0);

IDU:: loop(LAST_LOOP,3);

VMU:: SENDV(*acu4++);

VMU:: SENDV(*acu5++)
AMU:: RCV(vmu,amu0);

AMU:: RCV(vmu,amu1);

AMU:: ADDdi(amu0,amu1);

LAST_LOOP: VMU:: RCVL_AMU(*acu6++);

NOP;
NOP;

IDU:: EOS();
```

B.4 Implementation of Parallelization Strategy 3

```
.qint acucfg0 = [0,32,0,coef0,0,0,-2,data2]
.qint acucfg1 = [0,0,0,0,0,-4,-6,data2]


.dint coef0=[-38,-46,-38,-46,-38,-46,-38,-46,-38,-46,-38,-46,-38,-46,-38,-46]
.dint coef1=[64,96,64,96,64,96,64,96,64,96,64,96,64,96,64,96]
.dint coef2=[-143,-209,-143,-209,-143,-209,-143,-209,-143,-209,-143,-209,-143,-209,-143,-209]
.dint coef3=[296,409,296,409,296,409,296,409,296,409,296,409,296,409,296,409]
.dint coef4=[-555,-745,-555,-745,-555,-745,-555,-745,-555,-745,-555,-745,-555,-745,-555,-745]
.dint coef5=[998,1349,998,1349,998,1349,998,1349,998,1349,998,1349,998,1349,998,1349]
.dint coef6=[-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786]
.dint coef7=[4823,14747,4823,14747,4823,14747,4823,14747,4823,14747,4823,14747,4823,14747,4823,14747]
.dint coef8=[14747,4823,14747,4823,14747,4823,14747,4823,14747,4823,14747,4823,14747,4823,14747,4823]
.dint coef9=[-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877,-2786,-1877]
.dint coef10=[1349,998,1349,998,1349,998,1349,998,1349,998,1349,998,1349,998,1349,998]
.dint coef11=[-745,-555,-745,-555,-745,-555,-745,-555,-745,-555,-745,-555,-745,-555,-745,-555]
.dint coef12=[409,296,409,296,409,296,409,296,409,296,409,296,409,296,409,296]
.dint coef13=[-209,-143,-209,-143,-209,-143,-209,-143,-209,-143,-209,-143,-209,-143,-209,-143]
.dint coef14=[96,64,96,64,96,64,96,64,96,64,96,64,96,64,96,64]
.dint coef15=[-46,-38,-46,-38,-46,-38,-46,-38,-46,-38,-46,-38,-46,-38,-46,-38]
```

```
.code

int memwrite=960;

.stint  Decimation

/*-------------------------------------------------------------------------------*/
/*                          POINTERS INITIALIZATION                              */
/*-------------------------------------------------------------------------------*/

VMU:: LDACUS(acu0, acucfg0);
VMU:: LDACUS(acu2, acucfg1);

VMU:: LDBASE(acu4,memwrite);

VMU:: LDINCR(acu4,32);

VMU:: SENDV(*acu0++),SSND(*acu2++,qword);

VMU:: SENDV(*acu1++),SSND(*acu2++,qword)
AMU:: RCV(vmu,amu0)
SLU:: RCV(vmu,slu0),SRCV(vmu);

IDU:: loop(loop_end,13)
VMU:: SENDV(*acu1++),SSND(*acu2++,qword)
AMU:: RCV(vmu,amu1)
SLU:: SHIFTS(slu0),RCV(slu,slu0),SRCV(vmu);

loop_end:  VMU:: SENDV(*acu1++),SSND(*acu2++,qword)
           AMU:: RCV(vmu,amu1),RCV(slu,amu0),MACdi(amu0,amu1,acc2)
           SLU:: SHIFTS(slu0),RCV(slu,slu0),SRCV(vmu);

VMU:: SENDV(*acu1++)
AMU:: RCV(vmu,amu1),RCV(slu,amu0),MACdi(amu0,amu1,acc2)
SLU:: SHIFTS(slu0);

AMU:: RCV(vmu,amu1),RCV(slu,amu0),MACdi(amu0,amu1,acc2);

AMU:: MACdi(amu0,amu1,acc2); NOP;

AMU:: LSRAdi(acc2,15);

IDU:: EOS();
```

Programs for Adaptive filters

C.1 Implementation of BLMS algorithm in the time domain

```
.data

.address 0.0
.align   16


.dint zero1=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
.dint zero2=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
.dint data0=[31391,22045,9519,16332,6946,32753,8658,2069,30576,7678,2140,3329,29125,9463,31102,19592]
.dint data1=[24695,8625,26936,7996,18357,20353,26850,569,32292,26763,9444,487,3126,4291,21828,25118]
.dint data2=[25529,10171,422,28732,22195,22475,20427,17682,5581,20858,6008,21611,19823,19730,7014,21615]
.dint data3=[9058,12439,6332,2434,16686,19087,23043,15014,16917,7443,391,2316,2435,22240,30365,10069]
.dint data4=[21353,23652,24991,4002,5124,29762,26612,3705,30067,27133,31054,16478,32328,20912,10287,25260]
.dint data5=[29202,21516,28247,7490,29942,15275,11148,27871,2660,1167,6979,13744,8918,28950,21730,24708]


.dint d0=[6836,-2851,-3112,-3422,-2397,188,214,989,-1395,3248,-7399,-3913,-800,-2709,1801,2536]
.dint d1=[31999,21660,11615,9811,13547,15193,18142,13044,17456,13927,4514,680,9613,16913,26651,26243]
.dint d2=[24464,22891,14147,12508,16492,16456,16543,19572,24477,26053,25026,3974,-851,4596,11224,24820]
.dint d3=[14728,9393,14048,14291,21225,25062,20090,18571,13286,11133,8283,13629,22289,17815,11820,13090]
.dint d4=[11767,9479,5003,4207,4231,21068,26784,20304,9669,10922,6212,943,411,13485,20328,26666]
.dint d5=[26236,31162,22436,7097,7848,20273,19325,15792,25498,26286,30398,26628,23150,22010,18173,18235]

.int SFU_PAT1=[1,0,3,2,5,4,7,6,9,8,11,10,13,12,15,14,17,16,19,18,21,20,23,22,25,24,27,26,29,28,31,30]



.qint acucfg0 = [0,0,0,0,0,-2,0,data0]
.qint acucfg1 = [64,2,0,30.0,0,32,0,d0]

.code

.stint  FIR_32

/*-------------------------------------------------------------------------*/
/*                          POINTERS INITIALIZATION                        */
/*-------------------------------------------------------------------------*/
VMU:: LDACUS(acu0, acucfg0);          /* acu0 - data0, acu1 - coef0 */

VMU:: LDBASE(acu4,960);               /* Initialize acu4 */
```

```
VMU:: LDINCR(acu4,1.0);

VMU:: LDBASE(acu5,1440);              /* Initialize acu5 */

VMU:: LDINCR(acu5,1.0);

VMU:: LDBASE(acu1,1920);              /* Initialize acu1 */

VMU:: LDINCR(acu1,2);

VMU:: LDBOUND(acu1,64);

VMU:: LDACUS(acu2, acucfg1);

VMU:: LDBASE(acu6,SFU_PAT1);

IDU:: loop(OUTER_LOOP,2);

/*-------------------------------------------------------------------------*/

VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
AMU:: SUBdi(amu3,amu3),RCV(amu,amu3);

IDU:: loop(30, LOOP_END)
VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
AMU:: SUBdi(amu4,amu4),RCV(amu,amu4),RCV(vmu,amu0),SRCV(vmu);

LOOP_END: VMU:: SENDV(*acu0++),SSND(*acu1++,dword)
          AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

VMU:: SENDL(*acu2++)
AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

VMU:: LDINCR(acu0,96)
AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu4);

/*-------------------------------------------------------------------------*/

NOP;

AMU:: LSRAdi(acc1,15);          /* scale accumulator, result is in amu2 */

VMU:: SENDV(*acu0++)
AMU:: SUBdi(amu4,amu2);         /* send the result stored in amu2 */

VMU:: RCVL_AMU(*acu4++)         /* store result to memory */
AMU:: SUBdi(amu0,amu0),RCV(amu,amu0);

OUTER_LOOP:  VMU:: LDINCR(acu0,-2)
             AMU:: SUBdi(amu2,amu2),RCV(amu,amu2);
```

```
/*-------------------------------------------------------------------------------*/
/*       Finish the FIR operation, starting the updating part                    */
/*                         Errors are stored from 30th line,                     */
/*-------------------------------------------------------------------------------*/
       VMU:: LDINCR(acu0,-64);

       VMU:: SENDV(*acu0++);

       VMU:: LDINCR(acu0,-2);

       IDU:: loop(Error_LOOP,2);

/*-------------------------------------------------------------------------------*/

VMU:: SENDV(*acu0++),SSND(*acu3++,dword)
AMU:: SUBdi(amu3,amu3),RCV(amu,amu3);

IDU:: loop(30, LOOP_END1)
VMU:: SENDV(*acu0++),SSND(*acu3++,dword)
AMU:: SUBdi(amu4,amu4),RCV(amu,amu4),RCV(vmu,amu0),SRCV(vmu);

LOOP_END1: VMU:: SENDV(*acu0++),SSND(*acu3++,dword)
           AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

AMU:: MACdi(amu0,bcst,acc1),RCV(vmu,amu0),SRCV(vmu);

VMU:: LDINCR(acu0,96)
AMU:: MACdi(amu0,bcst,acc1);

/*-------------------------------------------------------------------------------*/

NOP;

AMU:: LSRAdi(acc1,15);          /* scale accumulator, result is in amu2 */

VMU:: SENDV(*acu0++)
AMU:: SUBdi(amu4,amu2);         /* send the result stored in amu2 */

VMU:: RCVL_AMU(*acu5++)         /* store result to memory */
AMU:: SUBdi(amu0,amu0),RCV(amu,amu0);

Error_LOOP:  VMU:: LDINCR(acu0,-2)
             AMU:: SUBdi(amu2,amu2),RCV(amu,amu2);

VMU:: LDINCR(acu5,-64);

VMU:: SENDL(*acu5++);

VMU:: LDINCR(acu5,32);
/*-------------------------------------------------------------------------------*/
/*                  Reverse the order of the updating vector                     */
```

```
/*-------------------------------------------------------------------------------*/

VMU:: SENDL(*acu6);

IDU:: loop(2, Shuffle)
SFU:: CONF(sfuc0);

VMU:: SENDL(*acu5);

SFU:: RCV(vmu);

SFU:: ODD(sfuc0);

SFU:: EVEN(sfuc0);

Shuffle: VMU:: RCVL_SFU(*acu5++)
         AMU:: RCV(sfu,amu6);

/*-------------------------------------------------------------------------------*/
/*                  Add the old coefficients with updating part                  */
/*-------------------------------------------------------------------------------*/
VMU:: LDOFFS(acu5,0);

IDU:: loop(update_loop,2)
VMU:: LDINCR(acu1,32);

VMU:: SENDL(*acu1);

VMU:: SENDL(*acu5++)
AMU:: RCV(vmu, amu8);

AMU:: RCV(vmu, amu9);

AMU:: ADDdi(amu8,amu9);

update_loop:  VMU:: RCVL_AMU(*acu1++);

NOP;

IDU:: EOS();
```

Curriculum Vitae

BAO LINH DANG was born in Hanoi, Vietnam on the 24th of July 1978. In 1996, he was admitted into Hanoi University of Technology. He received his bachelor degree with distinction in Electronics and Telecommunication Engineering at Hanoi University of Technology in May 2001.

In November 2001, he started his MSc study in Electrical Engineering at Delft University of Technology (TU Delft), The Netherlands. In September 2002, he started working on his MSc thesis at Philips Research Laboratories (Nat.Lab.), Eindhoven under the supervision of Nur Engin (Philips Nat.Lab.) and Georgi Gaydadjiev (Computer Engineering Lab - CE, TU Delft). The CE Laboratory is part of the Department of Electrical Engineering, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, and is chaired by Prof. Stamatis Vassiliadis. His MSc thesis is entitled: "Vectorization of Digital Filtering Algorithms". His research interests include: Embedded Systems, Computer Architecture, Digital Signal Processing (DSP) and DSP for Communication and Telecommunication Networks.