DESIGN OF A REUSABLE DISTRIBUTED ARITHMETIC FILTER AND ITS APPLICATION TO THE AFFINE PROJECTION ALGORITHM

A Dissertation
Presented to
The Academic Faculty

By

Haw-Jing Lo

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2009

TABLE OF CONTENTS

LIST OF TABLES iv

LIST OF FIGURES v

SUMMARY viii

CHAPTER 1 INTRODUCTION 1
- 1.1 Distributed Arithmetic 3
  - 1.1.1 Mechanization of Distributed Arithmetic 4
  - 1.1.2 Memory Reduction 5
  - 1.1.3 Throughput Increase 11
  - 1.1.4 Power Reduction 12
- 1.2 Adaptive Filtering 14
  - 1.2.1 Least Mean-Square and Normalized Least Mean-Square Algorithms 16
  - 1.2.2 Recursive Least-Squares Algorithm 18
  - 1.2.3 Affine Projection Algorithm 18
  - 1.2.4 Fast Affine Projection Algorithm 19
  - 1.2.5 Fixed-Point Effects in Adaptive Filters 24

CHAPTER 2 REUSABLE DISTRIBUTED ARITHMETIC 28
- 2.1 FIR Filtering 29
- 2.2 FIR Filtering with RDA 29
  - 2.2.1 RDA Partial Product Generation 30
  - 2.2.2 Summation Block 31
  - 2.2.3 Accumulation Block 31
- 2.3 FIR Filtering with Multipliers 33

CHAPTER 3 COMPARISON OF THE RDA AND MM FIR FILTERS 38
- 3.1 Gate Implementations of Arithmetic Logic 40
- 3.2 Gate-level Comparison of RDA-FIR and MM-FIR 42
- 3.3 FPGA Comparison of RDA-FIR and MM-FIR 49

CHAPTER 4 RDA AND THE FAST AFFINE PROJECTION ALGORITHM 53
- 4.1 NLMS using Distributed Arithmetic 54
- 4.2 Fast Affine Projection Algorithm 57
  - 4.2.1 Iterative Methods to Solve $\mathbf{R}[n]\mathbf{p}[n] = \mathbf{b}$ 58
  - 4.2.2 Solving for $\mathbf{p}[n]$ 59
  - 4.2.3 Update of $\mathbf{R}[n]$ 60
  - 4.2.4 Computing the Reciprocal $\frac{1}{r_0[n]}$ 61
- 4.3 Optimizations to the FAP Algorithm 63
- 4.4 FAP Adaptive Filtering with RDA 69
- 4.5 FAP Adaptive Filtering with MM 71

CHAPTER 5 COMPARISON OF THE RDA AND MM FAP ADAPTIVE FILTERS 73
5.1 Gate-level Comparison of RDA-FAP and MM-FAP 74
5.2 Improving the Throughput of RDA-FAP 82
5.2.1 Binary Signed-Digit Numbers 82
5.2.2 BSD and RDA-FAP 84

CHAPTER 6 CASE STUDY: DIGITAL HEARING AIDS 87
6.1 Wide Dynamic Range Compression 90
6.2 Feedback Cancellation 94

CHAPTER 7 CONCLUSION AND FUTURE RESEARCH 98
7.1 Summary of Contributions 98
7.1.1 Reusable Distributed Arithmetic 98
7.1.2 DA NLMS Adaptive Filter 98
7.1.3 FAP Adaptive Filter 99
7.1.4 Implications for Digital Hearing Aids 99
7.2 Future Research Directions 100
7.2.1 Application of RDA to the FFT/IFFT 100
7.2.2 RDA-NLMS 100
7.2.3 Identifying New Applications 101

REFERENCES 102

LIST OF TABLES

Table 1.1 Adaptive Filter Criteria 15

Table 1.2 LMS Algorithm 16

Table 1.3 NLMS Algorithm 17

Table 1.4 AP Algorithm 19

Table 1.5 FAP Algorithm 21

Table 1.6 Adaptive Filter Comparison Summary 24

Table 3.1 RDA-FIR and MM-FIR FPGA synthesis results 52

Table 4.1 Number of operations sorted by type for NLMS and DA-NLMS 57

Table 4.2 Number of operations sorted by type for FAP and RDA-FAP 66

Table 6.1 Specifications for a WDRC filter. 91

Table 6.2 Specifications for feedback cancellation. 95

LIST OF FIGURES

Figure 1.1 Basic FIR filter block diagram. 3

Figure 1.2 Block diagram of a 4-tap DA with full table 5

Figure 1.3 Memory contents of a 4-tap DA with a full table 6

Figure 1.4 Block diagram of 4-tap DA with half table 7

Figure 1.5 Memory contents of a 4-tap DA with a half table 7

Figure 1.6 Block diagram of a 4-tap DA using OBC 8

Figure 1.7 Memory contents of a 4-tap DA using OBC 9

Figure 1.8 Block diagram of 4-tap DA with split memories 9

Figure 1.9 Memory contents of a 4-tap DA with split memories 10

Figure 1.10 Block diagram of 2-tap DA using 2-BAAT 11

Figure 1.11 Memory contents of a 2-tap DA using 2-BAAT 12

Figure 1.12 System identification example 16

Figure 1.13 Effect of varying p on the AP algorithm 20

Figure 2.1 Block diagram of a RDA partial product generator 30

Figure 2.2 Coefficient register contents for a $R = 2$, $B = 4$ RDA FIR filter 31

Figure 2.3 Block diagram of the RDA summation block 32

Figure 2.4 Block diagram of the RDA accumulation block 33

Figure 2.5 Block diagram of a L-tap RDA FIR filter. 34

Figure 2.6 Block diagram of the MM multiplier and memories block. 35

Figure 2.7 MM input samples memory update sequence 36

Figure 2.8 Block diagram of the MM FIR filter. 37

Figure 3.1 Ideal throughput and latency plots for RDA-FIR and MM-FIR. 39

Figure 3.2 Full adder built from nine gates. 40

Figure 3.3 Full adder built from nine gates with delay paths marked. 41

Figure 3.4 Gate delays of different arithmetic components. 42

Figure 3.5 Gate counts of different arithmetic components. 43
Figure 3.6 Number of multipliers needed for MM-FIR. 44
Figure 3.7 Gate count and latency for MM-FIR. 45
Figure 3.8 Number of multipliers needed for PMM-FIR. 47
Figure 3.9 Gate count and latency for PMM-FIR. 48
Figure 3.10 RDA-FIR and MM-FIR FPGA synthesis results 50
Figure 4.1 Simulation results of LMS, NLMS, and DA-NLMS 56
Figure 4.2 Division via lookup table 64
Figure 4.3 Flowchart of the FAP algorithm. 67
Figure 4.4 RDA-FAP test configurations 68
Figure 4.5 Comparison of RDA-FAP to NLMS and FAP 69
Figure 4.6 Effects of reducing bit precision 70
Figure 4.7 Noise cancellation results comparing RDA-FAP to NLMS and FAP. . . 70
Figure 4.8 RDA-FAP partial product generation block 71
Figure 4.9 MM-FAP multiplier and memories block 72
Figure 5.1 Ideal throughput and latency plots for RDA-FAP and MM-FAP. 75
Figure 5.2 Number of multipliers needed for MM-FAP. 76
Figure 5.3 Gate count and latency for MM-FAP. 78
Figure 5.4 Number of multipliers needed for PMM-FAP. 79
Figure 5.5 Gate count and latency for PMM-FAP. 80
Figure 5.6 RDA-FIR and MM-FIR FPGA synthesis results. 81
Figure 5.7 Input/Output characteristics of on-line arithmetic 83
Figure 5.8 NRDA-FAP coefficient update 84
Figure 5.9 Throughput comparison of RDA-FAP and NRDA-FAP. 86
Figure 6.1 Digital hearing aid block diagram 88
Figure 6.2 Different types of hearing aids. 89
Figure 6.3 WDRC block diagram 90

Figure 6.4 Feedback cancellation block diagram 94

SUMMARY

Digital signal processing (DSP) is widely used in many applications spanning the spectrum from audio processing to image and video processing to radar and sonar processing. At the core of digital signal processing applications is the digital filter which are implemented in two ways, using either finite impulse response (FIR) filters or infinite impulse response (IIR) filters. The primary difference between FIR and IIR is that for FIR filters, the output is dependent only on the inputs, while for IIR filters the output is dependent on the inputs and the previous outputs. FIR filters also do not suffer from stability issues stemming from the feedback of the output to the input that affect IIR filters.

In this thesis, an architecture for FIR filtering based on distributed arithmetic is presented. The proposed architecture has the ability to implement large FIR filters using minimal hardware and at the same time is able to complete the FIR filtering operation in minimal amount of time and delay when compared to typical FIR filter implementations. The proposed architecture is then used to implement the fast affine projection adaptive algorithm, an algorithm that is typically used with large filter sizes. The fast affine projection algorithm has a high computational burden that limits the throughput, which in turn restricts the number of applications. However, using the proposed FIR filtering architecture, the limitations on throughput are removed. The implementation of the fast affine projection adaptive algorithm using distributed arithmetic is unique to this thesis. The constructed adaptive filter shares all the benefits of the proposed FIR filter: low hardware requirements, high speed, and minimal delay.

CHAPTER 1
INTRODUCTION

Digital signal processing (DSP) is widely used in many applications ranging from audio and speech processing to image and video processing to radar and sonar processing. DSP algorithms are implemented in hardware using computers, specialized processors called digital signal processors (DSPs), field-programmable gate arrays (FPGAs), or custom built hardware called application-specific integrated circuits (ASICs). The choice of hardware platform depends on the requirements imposed by the application. ASICs are the traditional solution to high performance applications, but the high development costs and time-to-market factors prohibit the deployment of such solutions for certain cases. DSP processors offer high programmability, but the sequential execution nature of their architecture can adversely affect their throughput performance. FPGAs are somewhere in between ASICs and DSPs, offering programmability and improved performance through parallelization.

At the core of digital signal processing applications is the digital filter. Digital filters are generally used for:

- Separation of signals that have been combined
- Restoration of signals that have been distorted
- Transform operations

Signal separation is used when a signal has been corrupted with noise or other forms of interference or combined with other signals. Echo cancellation in a telecommunications network is an example of separation.

Signal restoration is used when the signal has been distorted. An example of signal restoration is channel equalization, where the losses of the channel are identified and compensated for.

Transform operations involve the conversion or mapping of the signal from one domain to another. The Fourier transform is such an operation. The Fourier transform converts the signal from a time-domain representation to a frequency-domain representation. Other transforms include the discrete cosine transform (DCT), which is used in image compression, and the modified DCT (MDCT), which is used in audio compression.

Digital filtering is implemented in two ways, using either finite impulse response (FIR) filters or infinite impulse response (IIR) filters. The primary difference between FIR and IIR is that for FIR filters, the output is dependent only on the inputs, while for IIR filters the output is dependent on the inputs and the previous outputs. FIR filters also do not suffer from stability issues that affect IIR filters.

In this thesis, an architecture for FIR filtering based on distributed arithmetic is presented. The proposed architecture has the ability to implement large FIR filters using minimal hardware and at the same time is able to complete the FIR filtering operation in minimal amount of time and delay when compared to typical FIR filter implementations.

The proposed architecture is then used to implement the fast affine projection adaptive algorithm, an algorithm that is typically used with large filter sizes. The implementation of the fast affine projection adaptive algorithm using distributed arithmetic is unique to this thesis. The constructed adaptive filter shares all the benefits of the proposed FIR filter: low hardware requirements, high speed, and minimal delay.

The proposed FIR filter and adaptive filter are then analyzed in the context of digital hearing aids. Example hearing aid types are presented along with comparison results.

Figure 1.1 shows the basic diagram of a FIR filter. For an adaptive filter, the coefficients w_k are not fixed, after every output sample is computed the coefficients are updated by an adaptive algorithm. It is evident that as the filter length (L) increases, the number of components, and therefore the hardware resources, increases. For large filter sizes, the number of components may become too large to be practical.

Many applications require large FIR filters. Pulsed radar uses matched filters which can

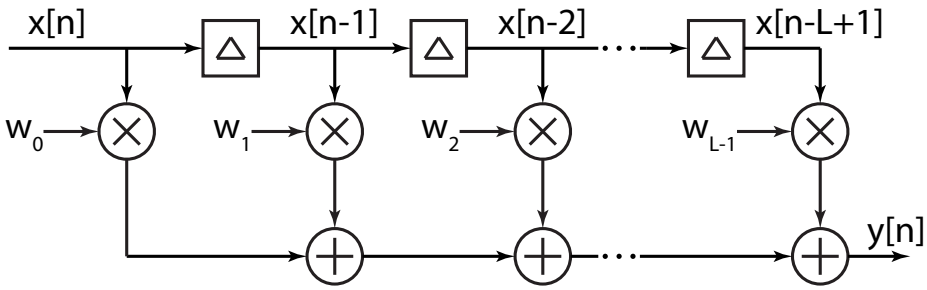


Figure 1.1. Basic FIR filter block diagram.

have filter lengths of up to 1000 points or more. Acoustic echo cancellation requires long filter lengths (> 10,000)to capture the echo which can take up to a few seconds to return. MDCT and the inverse, IMDCT, used in the encoding and decoding of audio streams such as MP3 and AAC, use filter lengths ranging from 128 points 1024 points. The examples listed not only require large filter lengths, but also require the filtering operation to be real-time, resulting in the need for high speed and low latency filtering methodologies.

The simplest way to achieve the speed and latency requirements of demanding applications is to simply use as much hardware resource as necessary. While this approach may be applicable to certain applications such as ground-based radar systems, power consumption demands for other applications such as portable music devices prevent such a direct approach.

1.1 Distributed Arithmetic

Distributed arithmetic (DA) is an efficient multiplication-free technique for calculating inner products first introduced by Croisier, *et al.* [1] and Zohar [2] and further developed by Peled and Liu [3] more than three decades ago. The multiplication operation is replaced by a mechanism that generates partial products and then sums the products together. The key difference between distributed arithmetic and standard multiplication is in the way the partial products are generated and added together. Since its introduction, distributed arithmetic has been widely adopted in many digital signal processing applications, including but not limited to digital filtering [4][5], discrete cosine transform [6][7], discrete Fourier

transform [8].

1.1.1 Mechanization of Distributed Arithmetic

Consider the following inner product of two N dimensional vectors $\mathbf{a}$ and $\mathbf{x}$, where $\mathbf{a}$ is a constant vector, $\mathbf{x}$ is the input sample vector, and y is the result.

$$y = \sum_{k=0}^{N-1} a_k x_k \tag{1.1}$$

Using B-bit 2's complement binary representation scaled such that $|x_k| < 1$ produces

$$x_k = -b_{k(B-1)} + \sum_{n=0}^{B-2} b_{kn} 2^{-(B-1)+n} \tag{1.2}$$

where b_{kn} are the bits (0 or 1) of x_k, $b_{k(B-1)}$ is the most significant bit, and b_{k0} is the least significant bit. Substituting Eqn. 1.2 into Eqn. 1.1 yields

$$y = \sum_{k=0}^{N-1} a_k \left[-b_{k(B-1)} + \sum_{n=0}^{B-2} b_{kn} 2^{-(B-1)+n} \right] \tag{1.3}$$

$$= -\sum_{k=0}^{N-1} a_k b_{k(B-1)} + \sum_{n=0}^{B-2} \left[\sum_{k=0}^{N-1} a_k b_{kn} \right] 2^{-(B-1)+n} \tag{1.4}$$

Equation 1.4 represents the distributed arithmetic computation. The values of b_{kn} are either 0 or 1, resulting in the bracketed term in Eqn. 1.4

$$\sum_{k=0}^{N-1} a_k b_{kn} \tag{1.5}$$

having 2^N possible values. Since $\mathbf{a}$ is a constant vector, the bracketed term can be pre-computed and stored in memory using either a lookup table (LUT) or ROM. For each bit depth, n, the lookup table is then addressed using the individual bits of the input samples, x_k which are typically stored in a long shift register chain, and the value read out is then accumulated, with the final result y appearing after B cycles. It is important to note that with this implementation, regardless of the lengths of the vectors $\mathbf{a}$ and $\mathbf{x}$, the final result y is computed in B cycles.

The direct or full table implementation of Eqn. 1.4 requires a memory size of 2×2^N words. The contents of the memory include all 2^N possible combinations of Eqn. 1.5 as

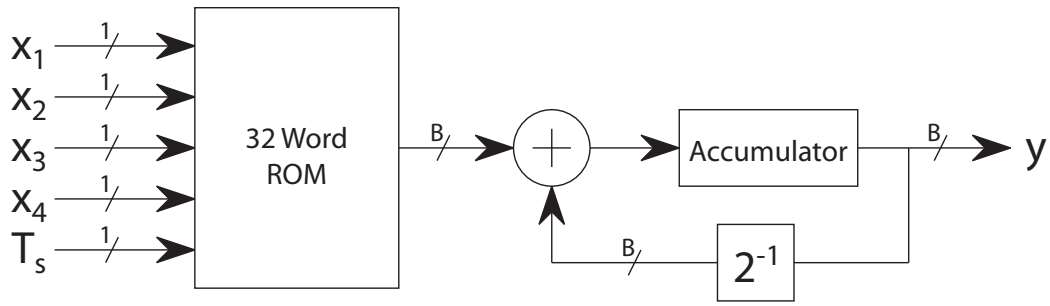


Figure 1.2. Block diagram of a 4-tap DA with full table. The full memory implementation requires a 32 word lookup table.

well as their negatives because of the term

$$-\sum_{k=0}^{N-1} a_k b_{k(B-1)} \tag{1.6}$$

which occurs when processing the sign bit. A block diagram of the full table implementation is shown in Fig. 1.2, with memory contents shown in Fig. 1.3

The ability of distributed arithmetic to reduce a multiply operation into a series of shifts and additions yields great potential for implementing various DSP systems at a significantly reduced area. However, this reduction in area comes at the cost of increased power and decreased throughput. This trade-off among area, throughput, and power has generated substantial research into making DA-based designs a more viable alternative to the standard multiply-accumulate designs for certain applications.

1.1.2 Memory Reduction

From Eqn. 1.5 it can be shown that for large values of N the memory size grows too large to be practical. However, the memory requirements can be reduced by a factor of 2 with a slight modification to the adder in Fig. 1.2. If the adder is changed to an adder/subtractor, the lower half of the memory in Fig. 1.3 can be removed. A block diagram of the half table implementation (2^N words) is shown in Fig. 1.4 with memory contents shown in Fig. 1.4.

Additional reduction in memory requirements can be achieved through the use of offset binary coding (OBC) [9], yielding a memory with contents shown in Fig. 1.7, with the block diagram of the OBC implementation in Fig. 1.6. Inspection of the memory contents

Memory Address	32 Word Memory Contents
00000	0
00001	a_0
00010	a_1
00011	$a_1 + a_0$
00100	a_2
00101	$a_2 + a_0$
00110	$a_2 + a_1$
00111	$a_2 + a_1 + a_0$
01000	a_3
01001	$a_3 + a_0$
01010	$a_3 + a_1$
01011	$a_3 + a_1 + a_0$
01100	$a_3 + a_2$
01101	$a_3 + a_2 + a_0$
01110	$a_3 + a_2 + a_1$
01111	$a_3 + a_2 + a_1 + a_0$
10000	0
10001	$-a_0$
10010	$-a_1$
10011	$-(a_1 + a_0)$
10100	$-a_2$
10101	$-(a_2 + a_0)$
10110	$-(a_2 + a_1)$
10111	$-(a_2 + a_1 + a_0)$
11000	$-a_3$
11001	$-(a_3 + a_0)$
11010	$-(a_3 + a_1)$
11011	$-(a_3 + a_1 + a_0)$
11100	$-(a_3 + a_2)$
11101	$-(a_3 + a_2 + a_0)$
11110	$-(a_3 + a_2 + a_1)$
11111	$-(a_3 + a_2 + a_1 + a_0)$

Figure 1.3. Memory contents of a 4-tap DA with a full lookup table implementation. Note the contents of the lower half are identical to the negative of the upper half of the memory.

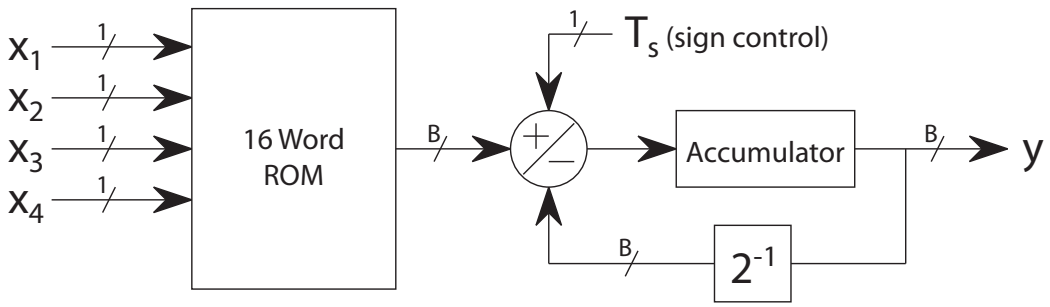


Figure 1.4. Block diagram of a 4-tap DA with half table. By replacing the adder with an adder/subtractor the memory size is reduced from 32 words to 16 words.

Memory Address	16 Word Memory Contents
0000	0
0001	a_0
0010	a_1
0011	$a_1 + a_0$
0100	a_2
0101	$a_2 + a_0$
0110	$a_2 + a_1$
0111	$a_2 + a_1 + a_0$
1000	a_3
1001	$a_3 + a_0$
1010	$a_3 + a_1$
1011	$a_3 + a_1 + a_0$
1100	$a_3 + a_2$
1101	$a_3 + a_2 + a_0$
1110	$a_3 + a_2 + a_1$
1111	$a_3 + a_2 + a_1 + a_0$

Figure 1.5. Memory contents of a 4-tap DA with a half lookup table implementation.

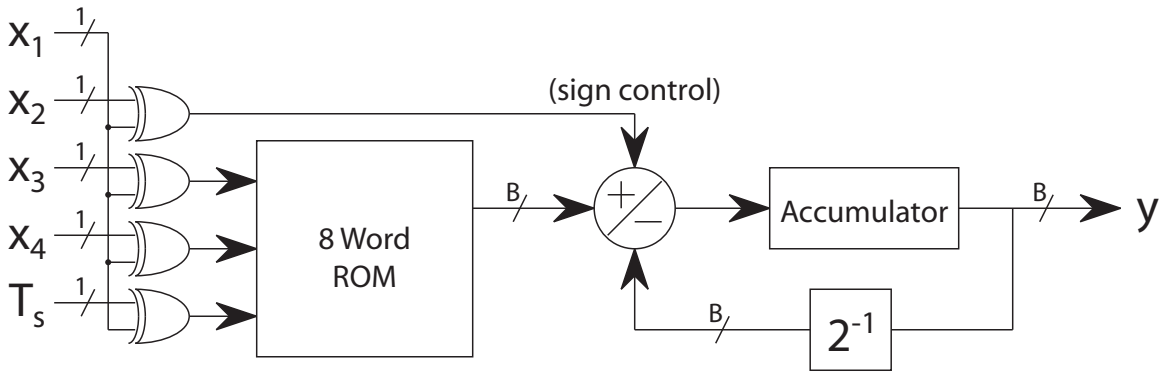


Figure 1.6. Block diagram of a 4-tap DA using OBC. Additional memory size reduction can be obtained from the use of offset binary coding.

shows that the values of the bottom half of the memory are identical to the negated values from the top half of the memory. As such, only half of the memory needs to be stored, resulting in a memory size of 2^{N-1} words.

Choi, *et al.* [10] reduced the memory size further, from 2^{N-1} of DA-OBC to 2^{N-2}, by exploiting the symmetry found in the contents of the memory. Their method reduces the memory by half at the cost of an adder/subtractor and some control logic. Yoo, *et al.* [11] iteratively applied the memory reduction technique by Choi, *et al.* [10] to obtain a LUT-less implementation of DA-OBC. Rather than having a lookup table consisting of combinations of coefficients, each entry in the memory contains a single coefficient. Extending this approach to the original DA architecture, each coefficient is replaced with a multiplexer/adder pair, negating the need for a LUT.

For certain applications, such as matrix transforms, it is possible to factorize the original transform matrix into smaller, sparse matrices. The combination of sparse matrix factorization with the OBC technique offers more memory reduction than using the OBC technique alone [12].

The discrete cosine transform (DCT) is another application that is well suited to DA implementations. Shams, *et al.* [13] propose a method where not only is the lookup table removed, but no storage elements are required for the coefficients. Their technique relies on

Memory Address	16 Word Memory Contents
0000	$-(1/2)(a_3 + a_2 + a_1 + a_0)$
0001	$-(1/2)(a_3 + a_2 + a_1 - a_0)$
0010	$-(1/2)(a_3 + a_2 - a_1 + a_0)$
0011	$-(1/2)(a_3 + a_2 - a_1 - a_0)$
0100	$-(1/2)(a_3 - a_2 + a_1 + a_0)$
0101	$-(1/2)(a_3 - a_2 + a_1 - a_0)$
0110	$-(1/2)(a_3 - a_2 - a_1 + a_0)$
0111	$-(1/2)(a_3 - a_2 - a_1 - a_0)$
1000	$(1/2)(a_3 - a_2 - a_1 - a_0)$
1001	$(1/2)(a_3 - a_2 - a_1 + a_0)$
1010	$(1/2)(a_3 - a_2 + a_1 - a_0)$
1011	$(1/2)(a_3 - a_2 + a_1 + a_0)$
1100	$(1/2)(a_3 + a_2 - a_1 - a_0)$
1101	$(1/2)(a_3 + a_2 - a_1 + a_0)$
1110	$(1/2)(a_3 + a_2 + a_1 - a_0)$
1111	$(1/2)(a_3 + a_2 + a_1 + a_0)$

Figure 1.7. Memory contents of a 4-tap DA using offset binary code. Only either the upper or lower half of the memory is used, since the memory contents are symmetrical around the midpoint.

the redundancy of calculations in the DCT. The typical DA ROM contents of the possible combinations of coefficients are mapped into an adder matrix array. This produces a very compact structure for computing the DCT. However this structure cannot be reconfigured for other filtering applications without being redesigned.

Through the introduction of additional arithmetic blocks, namely, adders, the size of

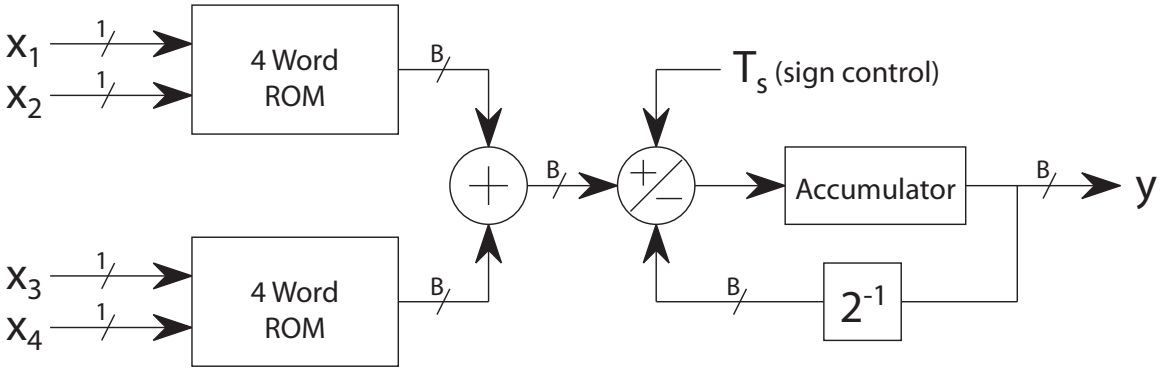


Figure 1.8. Block diagram of 4-tap DA with split memories. Total memory size is reduced in half from a half table implementation.

Memory Address	ROM 1 Contents	ROM 2 Contents
00	0	0
01	a_0	a_2
10	a_1	a_3
11	$a_0 + a_1$	$a_2 + a_3$

Figure 1.9. Memory contents of a 4-tap DA with split memories.

the memory can be further reduced [14]. The summation in the square brackets of Eqn. 1.4 can be split, yielding

$$y = -\sum_{k=0}^{N-1} a_k b_{k(B-1)} + \sum_{n=0}^{B-2} \left[\sum_{k=0}^{N/2-1} a_k b_{kn} + \sum_{k=N/2}^{N-1} a_k b_{kn} \right] 2^{-(B-1)+n} \tag{1.7}$$

$$= -\sum_{k=0}^{N-1} a_k b_{k(B-1)} + \sum_{n=0}^{B-2} \left[\sum_{k=0}^{N/2-1} a_k b_{kn} \right] 2^{-(B-1)+n} + \sum_{n=0}^{B-2} \left[\sum_{k=N/2}^{N-1} a_k b_{kn} \right] 2^{-(B-1)+n} \tag{1.8}$$

where the terms in the square brackets of Eqn. 1.8 each take $2^{N/2}$ values instead of 2^N.

Consider the memory in Fig. 1.5. This single 2^4 word, 4 bit address memory can be reduced to two 2^2 word, 2 bit address memories with the addition of an adder, yielding a factor of 2 in memory savings. This is illustrated in Fig. 1.8, with memory contents shown in Fig. 1.9. If a single memory has a capacity of $2^8 = 256$ words, splitting it into two $2^4 = 16$ words memories yields a factor of $\frac{256}{2\times16} = 8$ in memory savings. Rather than precomputing and storing all the possible combinations of Eqn. 1.5, some combinations are calculated online. This uniform memory splitting leads to an increase in latency and additional adder(s), but the area cost of the additional adder(s) is negligible compared to the savings obtained from memory reduction.

Another approach for realizing high-order filters (large memory) is to cascade smaller, lower-order filters to obtain the original higher-order filter [15]. Each stage of the cascade is a complete DA FIR filter, with the order determined by the desired performance of passband gain, stopband attenuation, passband ripple, and memory capacity. Memory is reduced significantly at the expense of increased latency and extra accumulation blocks from the individual filters.

1.1.3 Throughput Increase

The previous implementations described perform the inner product bit serially, *i.e.*, the individual bits of the input samples x_k are used one bit at a time to generate the output y. The rate at which the output can be computed depends directly on the rate at which the input bits are consumed and the number of bits used to represent a word. In order for the inner product computation to not be restricted by the number of bits, rather than utilizing a single bit at a time, additional speed can be obtained by ingesting multiple bits at a time [9]. The extra speed is achieved at the expense of increased memory resources. Figure 1.10 shows an example of a 2-bits-at-a-time (2-BAAT) DA system with memory contents shown in Fig. 1.11.

Garcia, *et al.* [16] applied the concept of a residue number system to mitigate the dependency of a bit serial system on bit precision. A residue number system represents a large integer using a set of smaller integers. Rather than computing a single arithmetic operation on two large integers, the arithmetic operation is broken down into many parallel operations on considerably smaller integers. The advantage of this system is that the number of required bits ingested serially is reduced.

Typical distributed arithmetic implementations when applied to recursive (IIR) filtering require two separate FIR filters. One filter processes the inputs $x[n]$ and the other filter processes the outputs $y[n]$. With each filtering requiring B clock cycles per output, where B is bit precision, the total number of clock cycles per output $y[n]$ is $2B$. Su, *et al.* [17]

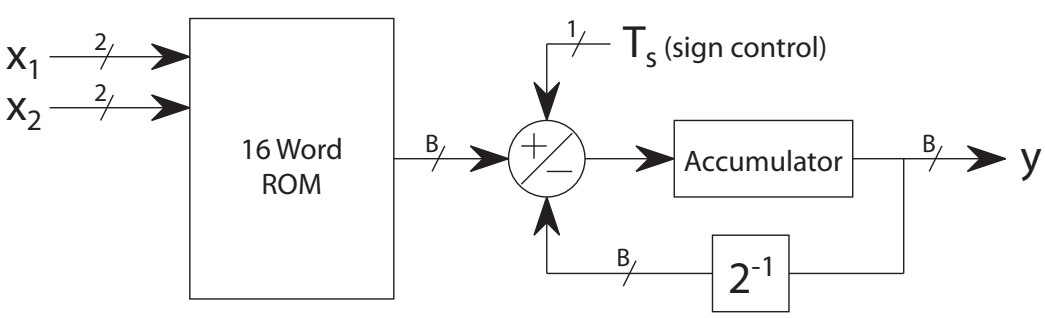


Figure 1.10. Block diagram of a 2-tap DA using 2-BAAT.

Memory Address	16 Word Memory Contents
0000	0
0001	a_0
0010	$2a_0$
0011	$3a_0$
0100	a_1
0101	$a_1 + a_0$
0110	$a_1 + 2a_0$
0111	$a_1 + 3a_0$
1000	$2a_1$
1001	$2a_1 + a_0$
1010	$2a_1 + 2a_0$
1011	$2a_1 + 3a_0$
1100	$3a_1$
1101	$3a_1 + a_0$
1110	$3a_1 + 2a_0$
1111	$3a_1 + 3a_0$

Figure 1.11. Memory contents of a 2-tap DA using 2-BAAT.

applied the sign digit number system [18] to remove carry chains that occur during addition/accumulation. With the carry chains removed, the bit serial operation can be performed MSB first with the intermediate results applied to the next iteration immediately.

The simplest way to increase throughput with a distributed arithmetic system is to simply truncate the bit serial input stream. This can be performed by simply dropping the desired lower bits, or performing the partial product generation starting from the MSB of the input data and stopping at a predetermined bit location [19].

1.1.4 Power Reduction

The power dissipated in a CMOS digital circuit is is composed of a static component and a dynamic component [20][21]. The static power dissipation can be formulated as

$$P_s = I_{leak} V_{DD} \tag{1.9}$$

where I_{leak} is the leakage current, and V_{DD} is the supply voltage referenced to ground. Ideally there should not be any current flowing through the transistors in the absence of any

switching activity; however there exists a leakage current that flows through the reverse-biased diode junction of the transistors. As transistor size decreases with advancing technology, power dissipation from leakage currents becomes an increasingly important factor.

The dynamic power component consists of dissipation from direct path currents and dissipation from the charging and discharging of capacitances. Direct path currents arise from the non-ideal (non-zero) signal transition times from low to high and high to low. In reality the finite slope creates a direct current path between V_{DD} and ground since both NMOS and PMOS transistors are conducting. The equation for power dissipation resulting from direct path current is

$$P_{dp} = I_{mean} V_{DD} \tag{1.10}$$

where I_{mean} is the average current during the transition period from low to high and high to low. The power dissipation due to the charging and discharging of capacitances is related to the frequency at which the circuit is operated. The relationship between power and frequency is formulated as

$$P_d = \alpha C V_{DD}^2 f, \quad 0 \leq \alpha \leq 1 \tag{1.11}$$

where α is the probability of switching, C is the capacitance at the switching node, and f is the frequency or rate of switching. It is important to note that the power dissipated through the charging and discharging of capacitances is independent of any physical attributes of the circuit [20].

For traditional DA systems, the system clock frequency, f_{sys}, is typically $B \cdot f_{samp}$, where B is the chosen bit precision and f_{samp} is the sampling clock frequency. For a fixed sampling frequency, if the bit precision increases, the system clock frequency would also increase, resulting in an increase in power consumption by a similar factor.

The techniques of the previous section lead to an increase in throughput, which results in the decrease of the required system clock frequency for a given sampling clock frequency, which ultimately leads to a decrease in dynamic power. The methods presented in

this section are directed at reducing the power consumption of DA architectures through other means.

One approach to reducing power dissipation is to minimize switching activity. Sacha, *et al.* [22] recoded the inputs using a nonredundant signed digit representation. The recoding process introduces zeros into the higher-order bits. When an all-zero lookup table address is encountered, the actual lookup and accumulation are suppressed, reducing power.

A similar approach was undertaken by Ramprasad, *et al.* [23] that utilized non-uniform memory size partitioning. For lookup table addresses that occur most frequently, the corresponding memory locations are placed into a small memory, with the next most frequent addresses in a slightly larger memory, and so forth. The rationale is that accesses to smaller memories dissipate less power [24]. This approach is application specific since it depends on the statistical properties of the input. As such, any benefits from this approach is lost once the statistics of the input changes.

The input shift register present in traditional DA implementations can be replaced by a memory [25]. Instead of shifting the data every system clock, pointers are used to reference the data. A remapping of bits is needed since the data in the memory does not change locations. This remapping is accomplished via a switch network that is updated at the sample clock frequency. Since the input data is no longer clocked at the high system clock frequency, significant power savings can be obtained.

1.2 Adaptive Filtering

Adaptive filtering is widely utilized in applications such as echo cancellation, noise cancellation, channel equalization, and system identification. In particular, finite impulse response (FIR) adaptive filters are more widely used than the infinite impulse response (IIR) counterpart for several reasons. FIR adaptive filters are more stable than IIR adaptive filters, since the filter coefficients, or weights, are more robust to quantization noise. FIR adaptive filter coefficients are also more easily updated. Adaptive algorithms range from very simple

Table 1.1. Adaptive Filter Criteria

Criteria	Description
Rate of convergence	Number of iterations needed
Misadjustment	Amount by which the final converged value differs from the true value
Robustness	Convergence behavior in the presence of noise
Computational requirements	Complexity, number of operations needed
Structure	How information flows in the algorithm, useful for hardware implementations
Numerical properties	Stability and accuracy

to very complex, with the choice of algorithm depending on the criteria presented in Tbl. 1.1.

An example of system identification is shown in Fig. 1.12. The goal of the adaptive algorithm is to adjust the coefficients of the adaptive filter, $\mathbf{w}[n]$, in order to minimize the error term $e[n]$ in the mean-squared sense. The adaptive algorithm essentially identifies a vector of coefficients $\mathbf{w}[n]$ that minimizes the following quadratic equation,

$$\xi[n] = E\left\{|e[n]|^2\right\} \tag{1.12}$$

where $e[n] = d[n] - y[n]$, $d[n]$ is the output from the unknown system (desired signal), $y[n]$ is the output from the adaptive filter, $E\{\cdot\}$ denotes the expected value, and $\xi[n]$ is the mean-squared error (MSE). Many methods exist to solve Eqn. 1.12, the most common being the method of steepest descent. The steepest descent method is an iterative process that estimates the solution vector at every time index by adding a correction term to the previous solution vector so that the current solution vector is closer to the optimal solution than the previous estimates. This process continues until the adaptive algorithm reaches steady-state, where the difference between the current solution vector and the optimal solution, or MSE, is at its minimum. In general, adaptive algorithms can be divided into four steps that are performed sequentially: filtering, computing the error, calculating the coefficient updates, and updating the coefficients. When split into this form, the primary difference

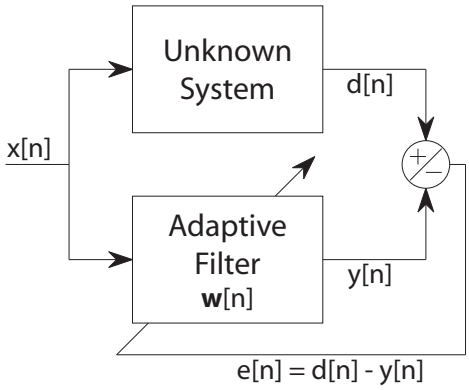


Figure 1.12. System identification example. The adaptive algorithm adjusts the filter coefficients $\mathbf{w}[n]$ so that $e[n]$ approaches zero.

between adaptive algorithms is in how the update calculation step is performed.

1.2.1 Least Mean-Square and Normalized Least Mean-Square Algorithms

The least mean-square (LMS) algorithm [26] is the most widely used FIR adaptive filtering algorithm. The algorithm is simple to implement, has low computational complexity, and the adaptation characteristics are adequate for most applications. The LMS algorithm is summarized in Tbl. 1.2. The term μ is a scaling factor, or step size, that controls the speed of adaptation, $\mathbf{w}[n]$ and $\mathbf{x}[n]$ are vectors of size L of the coefficients and input samples respectively. A drawback of the LMS algorithm is that the adaptation is sensitive to the magnitude of the input. The individual elements of the weight correction term $\hat{\mathbf{w}}[n]$ are directly proportional to the corresponding elements of the input $\mathbf{x}[n]$. As such, if the magnitude of $x[n]$ is large, the LMS algorithm suffers from a gradient noise amplification

Table 1.2. LMS Algorithm

Operation	Computation
Filtering	$y[n] = \mathbf{w}^T[n]\mathbf{x}[n]$
Compute Error	$e[n] = d[n] - y[n]$
Update Calculation	$\hat{\mathbf{w}}[n] = \mu e[n]\mathbf{x}[n]$
Coefficient Update	$\mathbf{w}[n] = \mathbf{w}[n-1] + \hat{\mathbf{w}}[n]$

problem [27][28].

The normalized LMS algorithm (NLMS) was introduced to overcome the sensitivity of the LMS algorithm to the inputs. The equations for the NLMS algorithm are listed in Tbl. 1.3. The update calculation of the NLMS algorithm differs from the LMS algorithm in that the weight correction term is normalized by $||\mathbf{x}[n]||^2$, which is the signal energy of $\mathbf{x}[n]$ and calculated using Eqn. 1.13.

$$||\mathbf{x}[n]||^2 = \sum_{i=0}^{L-1}(x[n-i])^2 \tag{1.13}$$

The normalization of the input by its signal energy enables the NLMS algorithm to converge faster than the LMS algorithm. The complexity of LMS and NLMS is $O(L)$.

When the inputs to the adaptive filter are highly correlated however, the convergence speed of the LMS and NLMS adaptive algorithms is reduced. Speech is an example of a highly correlated signal. The reason for the slow convergence speed is attributed to the fact that the LMS and NLMS algorithms minimize the error in the mean-squared sense. The convergence speed of MSE adaptive filters is dependent on the statistics of the input signals, in particular the autocorrelation of the input and the cross correlation between the input and desired signals [27]. The higher the correlation, the slower the convergence.

Table 1.3. NLMS Algorithm

Operation	Computation
Filtering	$y[n] = \mathbf{w}^T[n]\mathbf{x}[n]$
Compute Error	$e[n] = d[n] - y[n]$
Update Calculation	$\hat{\mathbf{w}}[n] = \mu e[n]\frac{\mathbf{x}[n]}{\|\|\mathbf{x}[n]\|\|^2}$
Coefficient Update	$\mathbf{w}[n] = \mathbf{w}[n-1] + \hat{\mathbf{w}}[n]$

1.2.2 Recursive Least-Squares Algorithm

The recursive least-squares (RLS) algorithm [29] has the fastest convergence speed when compared to the LMS and NLMS algorithms. The speed of convergence in the RLS algorithm is dependent on the input signals themselves, instead of the statistics of the signals. While the RLS algorithm converges faster than LMS and NLMS, particularly for correlated inputs, the computational complexity, $O(L^2)$, of the algorithm is very high, making it too expensive for many applications where long filter lengths are required, such as acoustic echo cancellation. Fast versions of the RLS algorithm have been proposed that reduce the computational complexity, however they suffer from numerical instability when implemented with finite precision computations.

1.2.3 Affine Projection Algorithm

The affine projection (AP) algorithm was proposed as a generalization to the NLMS algorithm [30], and offers a faster convergence rate for correlated signals with a computational complexity between the NLMS and RLS algorithms. The affine projection algorithm is defined by the equations in Tbl. 1.4, where $\mathbf{X}[n]$ is a $L \times p$ matrix containing the input samples $x[n]$, as expressed by the following,

$$\mathbf{X}[n] = \Big[\mathbf{x}[n], \mathbf{x}[n-1], \ldots, \mathbf{x}[n-p+1]\Big] \tag{1.14}$$

$$\mathbf{x}[n] = \Big[x[n], x[n-1], \ldots, x[n-L+1]\Big]^T \tag{1.15}$$

and

$$\mathbf{d}[n] = \Big[d_0[n], d_1[n], \ldots, d_{p-1}[n]\Big]^T \tag{1.16}$$

$$\mathbf{y}[n] = \Big[y_0[n], y_1[n], \ldots, y_{p-1}[n]\Big]^T \tag{1.17}$$

$$\mathbf{e}[n] = \Big[e_0[n], e_1[n], \ldots, e_{p-1}[n]\Big]^T \tag{1.18}$$

where p and L are the projection order and adaptive filter length respectively, $\mathbf{e}[n]$ is a vector containing the difference between the desired signal vector, $\mathbf{d}[n]$ and the output

signal vector, $\mathbf{y}[n]$, $\mathbf{w}[n]$ is a vector containing the weights of the adaptive filter, δ is the regularization variable for stability purposes and is typically very small, and μ is the step size.

The AP algorithm updates the coefficients using p input samples vectors $\mathbf{x}[n], \mathbf{x}[n-1], \ldots, \mathbf{x}[n-p+1]$, whereas the NLMS algorithm uses a single input $\mathbf{x}[n]$. If the projection order p is 1, then the affine projection algorithm reduces to the NLMS algorithm. Conversely, if the projection order p is L, then the AP algorithm becomes the RLS algorithm. As such, the convergence behavior and computational complexity of the AP algorithm can be controlled by varying the projection order. The effect of p on the convergence rate is shown in Fig. 1.13. Typical values of p range from 2 to 16. From the update calculation of Tbl. 1.4,

$$\hat{\mathbf{w}}[n] = \mu \mathbf{X}[n] \left(\mathbf{X}^T[n]\mathbf{X}[n] + \delta \mathbf{I} \right)^{-1} \mathbf{e}[n] \tag{1.19}$$

it can be noted that the affine projection is computationally expensive because of the number of samples utilized and the inversion of the autocorrelation matrix, $\mathbf{X}^T[n]\mathbf{X}[n] = \mathbf{R}[n]$. However, the complexity of the AP algorithm, $2Lp + O(p^2)$, is still less than that of the RLS algorithm.

1.2.4 Fast Affine Projection Algorithm

A crucial milestone in the implementation of the affine projection algorithm was the development of the fast affine projection (FAP) algorithm [31][32][33]. The FAP algorithm can have a high convergence rate rivaling RLS at a complexity relative to NLMS when

Table 1.4. AP Algorithm

Operation	Computation
Filtering	$\mathbf{y}[n] = \mathbf{X}^T[n]\mathbf{w}[n]$
Compute Error	$\mathbf{e}[n] = \mathbf{d}[n] - \mathbf{y}[n]$
Update Calculation	$\hat{\mathbf{w}}[n] = \mu \mathbf{X}[n] \left(\mathbf{X}^T[n]\mathbf{X}[n] + \delta \mathbf{I} \right)^{-1} \mathbf{e}[n]$
Coefficient Update	$\mathbf{w}[n] = \mathbf{w}[n-1] + \hat{\mathbf{w}}[n]$

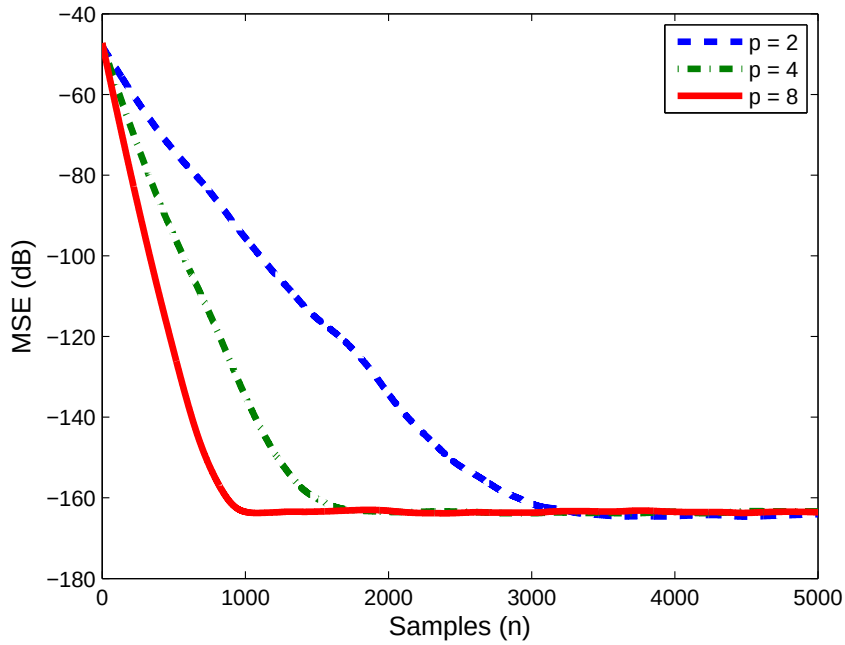


Figure 1.13. The effect of projection order p on convergence rate of the AP algorithm. The convergence curves are obtained by averaging 100 trials from a system identification setup.

the input parameters are configured properly. The fast affine projection algorithm achieves these performance gains and resource savings by using the same techniques that led to fast recursive least squares (FRLS) from RLS. The key parts from the development of the fast affine projection algorithm are as follows:

1. The approximation for the error vector

$$\mathbf{e}[n] = \begin{bmatrix} e[n] \\ (1-\mu)\underline{\mathbf{e}}[n-1] \end{bmatrix} \tag{1.20}$$

   where $\underline{\mathbf{e}}[n-1]$ is the upper $p-1$ elements of $\mathbf{e}[n-1]$

2. Use of sliding windowed FRLS to calculate the matrix inverse

Since the matrix inversion procedure is recursive, any errors present would accumulate and cause the system to become unstable. To prevent instability, the inversion process would have to be periodically reset to get rid of accumulated numerical errors. With floating-point systems this period restart may be acceptable, but with fixed-point systems

Table 1.5. FAP Algorithm

Operation	Computation
Filtering	$y[n] = \mathbf{x}^T[n]\mathbf{w}[n-1]$
Compute Error	$e[n] = d[n] - y[n] - \mu\bar{\mathbf{r}}^T[n]\underline{\boldsymbol{\eta}}[n-1]$
Update Calculation	$\mathbf{r}[n] = \mathbf{r}[n-1] + x[n]\boldsymbol{\alpha}[n] - x[n-L]\boldsymbol{\alpha}[n-L]$
	$\mathbf{e}[n] = \begin{bmatrix} e[n] \\ (1-\mu)\underline{\mathbf{e}}[n-1] \end{bmatrix}$
	$\boldsymbol{\varepsilon}[n] = \mathbf{P}[n]\mathbf{e}[n]$
	$\boldsymbol{\eta}[n] = \begin{bmatrix} 0 \\ \underline{\boldsymbol{\eta}}[n-1] \end{bmatrix} + \boldsymbol{\varepsilon}[n]$
	$\hat{\mathbf{w}}[n] = \mu\eta_{p-1}[n]\mathbf{x}[n-p+1]$
Coefficient Update	$\mathbf{w}[n] = \mathbf{w}[n-1] + \hat{\mathbf{w}}[n]$

the errors accumulate very quickly, forcing the restart period to be very small and impractical [34].

Table 1.5 lists the equations involved in the FAP algorithm. $\mathbf{X}[n]$ and $\mathbf{x}[n]$ are the same as the AP algorithm, $\mathbf{P}[n] = \mathbf{R}^{-1}[n]$ is the inverse of the $p \times p$ autocorrelation matrix $\mathbf{R}[n] = \mathbf{X}^T[n]\mathbf{X}[n]$, $\boldsymbol{\eta}[n]$ is a $p \times 1$ vector, η_{p-1} is the bottommost element of $\boldsymbol{\eta}[n]$, $\underline{\mathbf{e}}[n]$ and $\underline{\boldsymbol{\eta}}[n]$ are the $p-1$ upper elements of $\mathbf{e}[n]$ and $\boldsymbol{\eta}[n]$ respectively, $\bar{\mathbf{r}}[n]$ is the lower $p-1$ elements of $\mathbf{r}[n]$, and $\boldsymbol{\alpha}[n] = [x[n], x[n-1], \ldots, x[n-p+1]]^T$. The FAP algorithm has more equations than the AP algorithm, however the total computational complexity is significantly less, since fewer matrix operations are performed.

From Eqn. 1.20 it can be shown that if $\mu = 1$, then $\mathbf{e}[n]$ will only have one significant element, $e[n]$. This in turn simplifies the matrix inversion problem, since only p elements have to be calculated instead of p^2 elements of $\mathbf{R}^{-1}[n]$.

Define the following:

$$\mathbf{P}[n] = \mathbf{R}^{-1}[n] \tag{1.21}$$

$$\boldsymbol{\varepsilon}[n] = \mathbf{P}[n]\mathbf{e}[n] \tag{1.22}$$

Setting $\mu = 1$ in Eqn. 1.20 to obtain

$$\boldsymbol{\varepsilon}[n] = e[n]\mathbf{p}[n] \tag{1.23}$$

where $\mathbf{p}[n]$ is the leftmost column of $\mathbf{P}[n]$. The matrix inversion problem can then be obtained by solving a system of linear equations

$$\mathbf{R}[n]\mathbf{p}[n] = \mathbf{b} \tag{1.24}$$

where

$$\mathbf{b} = [1, 0, \ldots, 0]^T \tag{1.25}$$

is a length p vector containing 0 except for the first element which contains 1. Substituting Eqns. 1.22 and 1.23 into Eqn. 1.19,

$$\hat{\mathbf{w}}[n] = \mathbf{X}[n]\left(\mathbf{X}^T[n]\mathbf{X}[n] + \delta\mathbf{I}\right)^{-1}\mathbf{e}[n] \tag{1.26}$$

$$= \mu\mathbf{X}[n]\boldsymbol{\varepsilon}[n] \tag{1.27}$$

$$= \mu\mathbf{X}[n]\mathbf{p}[n]e[n] \tag{1.28}$$

The resulting equation no longer requires the direct calculation of the matrix inverse. The coefficient correction term can be computed by solving the linear system of equations to obtain the vector $\mathbf{p}[n]$.

Multiple methods exist to solve the system of linear equations represented by Eqn. 1.24. The conjugate gradient FAP (CGFAP) approach was developed to remove the stability issues with previous FAP implementations [34]. CGFAP solves the system of linear equations through the use of the conjugate gradient method, a numerical method that solves a linear system of equations whose matrix is positive definite and symmetric, as is the case of $\mathbf{R}[n]$. CGFAP has been simulated on a 16-bit fixed-point DSP with good results. The only drawback to this method is that the possible values of μ must be restricted to be close to 1 for the simplification of the matrix inversion problem of Eqn. 1.23 to hold.

Another method is to assume the autocorrelation matrix is Toeplitz [35]. The autocorrelation matrix in Eqn. 1.19, $\mathbf{X}^T[n]\mathbf{X}[n] = \mathbf{R}[n]$, is expressed as

$$\mathbf{R}[n] = \begin{bmatrix} r_0[n] & r_1[n] & \dots & r_{p-1}[n] \\ r_1[n-1] & r_0[n-1] & \dots & r_{p-2}[n-1] \\ \vdots & \vdots & \ddots & \vdots \\ r_{p-1}[n-p+1] & r_{p-2}[n-p+1] & \dots & r_0[n-p+1] \end{bmatrix} \tag{1.29}$$

where

$$r_\tau[n] = \sum_{i=0}^{N-1} x[n-i]x[n-i-\tau] \tag{1.30}$$

If $N \gg p$, which is the case for most applications, then

$$r_\tau[n] = r_\tau[n-1] = r_\tau[n-2] = \dots = r_\tau[n-p+1] \tag{1.31}$$

and $\mathbf{R}[n]$ can be approximated as

$$\hat{\mathbf{R}}[n] = \begin{bmatrix} r_0[n] & r_1[n] & \dots & r_{p-1}[n] \\ r_1[n] & r_0[n] & \dots & r_{p-2}[n] \\ \vdots & \vdots & \ddots & \vdots \\ r_{p-1}[n] & r_{p-2}[n] & \dots & r_0[n] \end{bmatrix} \tag{1.32}$$

which is a Toeplitz matrix [35]. The inverse of a Toeplitz matrix can be solved for using the Levinson-Durbin recursion [36][37][38].

The Gauss-Seidel fast affine projection algorithm (GSFAP) [39] takes the same approach as CGFAP, with the same assumptions made to reduce the complexity of the matrix inversion problem from p^2 to p. The two methods differ in the way the system of linear equations, Eqn. (1.24), is solved. GSFAP uses a single iteration of the Gauss-Seidel method.

The Gauss-Seidel method [40] is similar to the Jacobi method of solving a linear system of equations and guarantees convergence if the matrix is positive definite and symmetric.

Table 1.6. Adaptive Filter Comparison Summary

Criteria	LMS/NLMS	RLS	AP	FAP
Rate of convergence	slow	fast	fast	moderate fast
Misadjustment	depends on μ	very small	depends on μ	depends on μ
Robustness	good	average	good	good
Computational requirements	low	high	moderate high	moderate
Structure	good	average	average	average
Numerical properties	good	poor*	poor*	poor*

* Depends on how the matrix inverse is computed

The difference is that the computation of the current element uses updated values of preceding elements as soon as they are available. Like the Jacobi method, the Gauss-Seidel method also requires the matrix $\mathbf{R}[n]$ to be diagonally dominant, where

$$|r_{ii}| > \sum_{i \neq j} |r_{ij}| \tag{1.33}$$

In correlated systems it might not always be possible for Eqn. (1.33) to hold true. That, plus the same restrictions on μ as CGFAP, restricts the use of GSFAP for certain applications.

Since the introduction of GSFAP, other affine projection algorithms based on the Gauss-Seidel method have been introduced. The variants include LC-GSFAP [41], which applies the GSFAP algorithm to subband processing, MGSFAP [42], which allows the GSFAP algorithm to use any step size μ, among others. Other verions of the FAP algorithm exist as well, differing on the way the inverse of the autocorrelation matrix is estimated [33][43][44].

The various adaptive algorithms presented are summarized in Tbl. 1.6, and compared using the criteria introduced in Tbl. 1.1.

1.2.5 Fixed-Point Effects in Adaptive Filters

The implementation of adaptive filters using fixed-point precision requires careful consideration on multiple issues. The issues associated with the act of quantization include the

following:

- Noise from quantization of the inputs
- Effects of quantization of the coefficients
- Effects of rounding after performing arithmetic operations
- Overflow/underflow and the minimum number of bits

The conversion of an analog signal to a digital signal requires the signal to be quantized, where the digital version of the signal is represented using a finite number of bits. This process introduces noise into the system. The effect of quantization on the coefficients is to change the transfer function, or behavior, of the system, introducing error in the output. In the case of IIR filters, quantization of the coefficients can cause the filter to be unstable. Having more bit-precision reduces the quantization error, but increases the hardware resources. Conversely, reducing hardware complexity by reducing precision increases errors from quantization.

Fixed-point systems store data in fixed-width registers matching the precision of the data. However, when arithmetic operations are performed on the data, the result from the operation requires a larger register to hold to increased number of bits. For example, when two B-bit binary numbers are multiplied together, the product of the two numbers requires $2B$ bits. In order to maintain the original width, the result is either truncated or rounded.

Overflow occurs when the number to be represented in fixed-point form is too large, and underflow occurs when the number is too small. Consider a 3-bit two's complement number, k, capable of representing the values,

$$k \in \{-4, -3, -2, -1, 0, 1, 2, 3\} \tag{1.34}$$

If k is used to represent the number 5, then k would overflow because 5 exceeds the range of k. If k is used to represent the number 0.02, then k would underflow since the number would be rounded to 0.

The previously listed fixed-point issues from quantization affects the performance of an adaptive filter in several ways.

- Stability of adaptive algorithm
- Slower convergence
- Poorer steady-state performance

The stability of the fixed-point adaptive filter is dependent on the type of adaptive algorithm. Some algorithms such as RLS, are more sensitive to quantization errors while other algorithms such as LMS, are more robust. Quantization errors decrease the speed of convergence by reducing the accuracy of the computations. The inaccuracies then lead to a poorer steady-state performance. Instead of adapting until the system reaches steady-state (convergence), the adaptive filter may stop at an earlier point, resulting in a higher mean-squared error than would have been obtained with an infinite precision implementation.

The negative effects of fixed-point precision on adaptive filters can be minimized by increasing the internal bit-precision of the adaptive system. When an arithmetic operation is performed, instead of rounding the result to the input bit-precision, the bit-width of the result is kept. Further operations based on this increased precision number will also result in the expansion of the bit-width. For example the addition of two B-bit numbers, $a+b=c$, yields the number c which is represented using $B+1$ bits. If c is added to another number of the same precision, the answer is stored into a register of width $B+2$ bits. The process repeats until all the necessary computations for the output are complete. The computed final output of the adaptive filter is then rounded to the original bit-precision. In this manner errors arising from rounding can be mitigated at the small cost of increased register widths. Detailed analysis of the adaptive algorithm provides the bit-precision requirements for each stage of the algorithm, so that the appropriate register widths can be selected for every level of the algorithm.

The stability of the adaptive filter can be improved by making sure the internal precision is adequate. This is a different concept than increasing precision after every operation. In general by having enough precision overflows and underflows can be avoided, which improves the stability of adaptive filters under fixed-point conditions.

CHAPTER 2

REUSABLE DISTRIBUTED ARITHMETIC

Previous work on distributed arithmetic was heavily focused on the reduction of area, and partly focused on the increase in throughput and decrease in power dissipation. The work accomplished in this research leverages the advancements from prior work, culminating in the creation of a new distributed arithmetic filter design. This new design offers reduced area over the previous proposed designs. The design is termed reusable distributed arithmetic (RDA) since the driving force behind this work was to minimize area resources through the reuse of DA components, similar to the way multiplier-based architectures reuse the multipliers [45]. This new distributed arithmetic design is applied to FIR filtering and evaluated against other design approaches.

The original DA architecture stores all the possible binary combinations of the coefficients w_k in a memory or lookup table. It is evident that for large values of L, the size of the memory containing the precomputed terms grows exponentially too large to be practical. The memory size can be reduced by dividing the single large memory (2^L words) into m multiple smaller sized memories [14] each of size 2^k where $L = m \times k$. The memory size can be further reduced to 2^{L-1} and 2^{L-2} by applying offset binary coding and exploiting resultant symmetries found in the contents of the memories [9][10]. However, for very large values of L, the listed approaches still succumb to the limitations of storing the coefficient combinations in memory due to the exponential dependence of memory size on filter length.

RDA reduces the area requirements of the original DA in two ways: 1) by having a linear dependence of memory size on filter length, and 2) reusing the computation blocks much like the way multiply-accumulate units are reused.

2.1 FIR Filtering

Traditional implementations of an L-tap FIR filter

$$y[n] = \sum_{k=0}^{L-1} h[k]x[n-k] \tag{2.1}$$

where $h[k]$ are the filter coefficients and $x[n-k]$ are the input samples, require the use of L multiply-accumulate (MAC) units. Modern implementations employ M MAC units that are reused, where typically $M \ll L$. A multiplier-based architecture can be easily configured for a given set of speed, area, and power specifications. High throughput can be obtained by adding more multipliers. Smaller area can be obtained by using fewer multipliers. If high throughput and small area are desired, simply use a small enough number of multipliers and clock the system at a fast enough rate to meet throughput requirements, all the while acknowledging the trade-offs among area, speed, and power.

For distributed arithmetic-based FIR filters, similar configuration scenarios exist. When implementing FIR filters using DA, the area resource is dictated by the size of the lookup table, which in turn is dictated by the length of the filter and the bit precision. Throughput is proportional to bit precision, as is power consumption. One advantage that distributed arithmetic architectures possesses over multiplier architectures is a smaller hardware footprint for the same filtering operation. This advantage comes from implementing a multiply operation using smaller, simpler components. Another advantage is that a filtering operation with DA requires only B cycles to complete, regardless of filter length.

2.2 FIR Filtering with RDA

The RDA FIR filter is composed of the following blocks: partial product generators, a summation block, and an accumulation block. The reuse factor, R, of the RDA system determines the number of times the computation blocks are used during the processing of a single output sample. The input samples and coefficients are stored in registers, and the possible combinations of Eqn. 1.5 are generated online by sending the partial products into

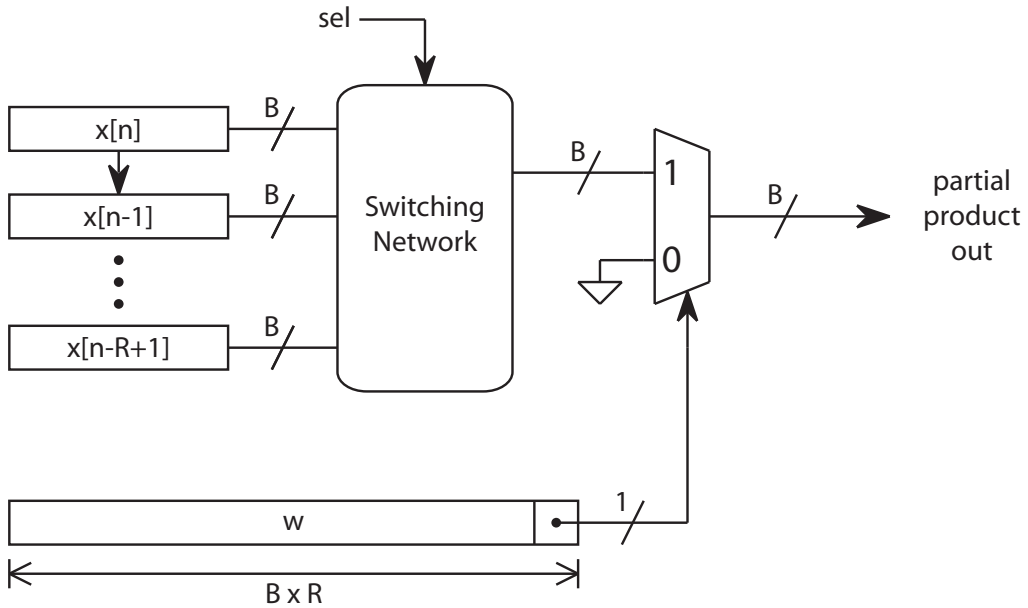


Figure 2.1. Block diagram of a RDA partial product generator. The bits of the coefficient register w are used a single bit at a time. Each bit is paired with the corresponding input sample $x[n-k]$. The switching network routes the correct input sample to the multiplexer.

the summation block. The unique aspect of RDA when compared against other DA implementations is that it has a linear instead of exponential dependence of memory on filter length and the coefficients instead of the input samples drive the system. This combination enables RDA to reuse the computation blocks much like the way MAC units are reused in traditional FIR filter implementations.

2.2.1 RDA Partial Product Generation

The RDA partial product generator is illustrated in Fig. 2.1. Typical distributed arithmetic filters are driven via the input samples. In order to facilitate the reuse of components, in RDA the coefficients drive the distributed arithmetic system, and the coefficients that drive the MUX are already multiplexed together. That is, the individual bits of the coefficients are stored in a single register in a multiplexed or interleaved fashion. Once all the bits of the coefficient register are used, the input samples are advanced forward one word at a time. For a $R = 2$ and $B = 4$ system the coefficient register contents are shown in Fig. 2.2, where $w_{A,n}$ is the n-th bit of w_A.

The operation of the RDA partial product generator is as follows. The contents of the coefficient register are shifted a single bit at a time to the right. Depending on the value of the least significant bit (LSB) position, the MUX then selects to output 0 when the LSB is 0, or the corresponding input sample $x[n]$ when the LSB is 1. The LSB of the coefficient register is connected to the most significant bit (MSB), so that after one complete filtering cycle the contents of the coefficient register remains unchanged, since the individual bits of the register are shifted back to their original positions.

2.2.2 Summation Block

The summation block consists of registered adders arranged in a tree formation. The adders are registered at the outputs for ease of implementation, *i.e.*, at every clock cycle the partial products advance forward one level in the tree. The diagram of the summation block is shown in Fig. 2.3.

2.2.3 Accumulation Block

The output from the summation block is shifted and accumulated in the accumulation block (Fig. 2.4), with the result ready after $R \times B$ clock cycles. As the output from the summation block arrives, it is accumulated for R cycles, shifted to the right a single bit, and added to the next output of the summation block. This cycle repeats until all $R \times B$ bits are processed. A normal approach towards the accumulation block using RDA would be to have multiple accumulators, one accumulator for each reuse. Once the partial product generation is complete for all R, the values in the accumulators are summed to obtain the output. The advantage of the multiplexed feedback accumulator over the normal approach is smaller area and lower latency, since a single accumulator is required versus R accumulators, and no extra adders are needed to compute the output. The complete RDA design is illustrated in Fig. 2.5.

$w_{B,3}$	$w_{A,3}$	$w_{B,2}$	$w_{A,2}$	$w_{B,1}$	$w_{A,1}$	$w_{B,0}$	$w_{A,0}$

Figure 2.2. Coefficient register contents for a $R = 2$, $B = 4$ RDA FIR filter

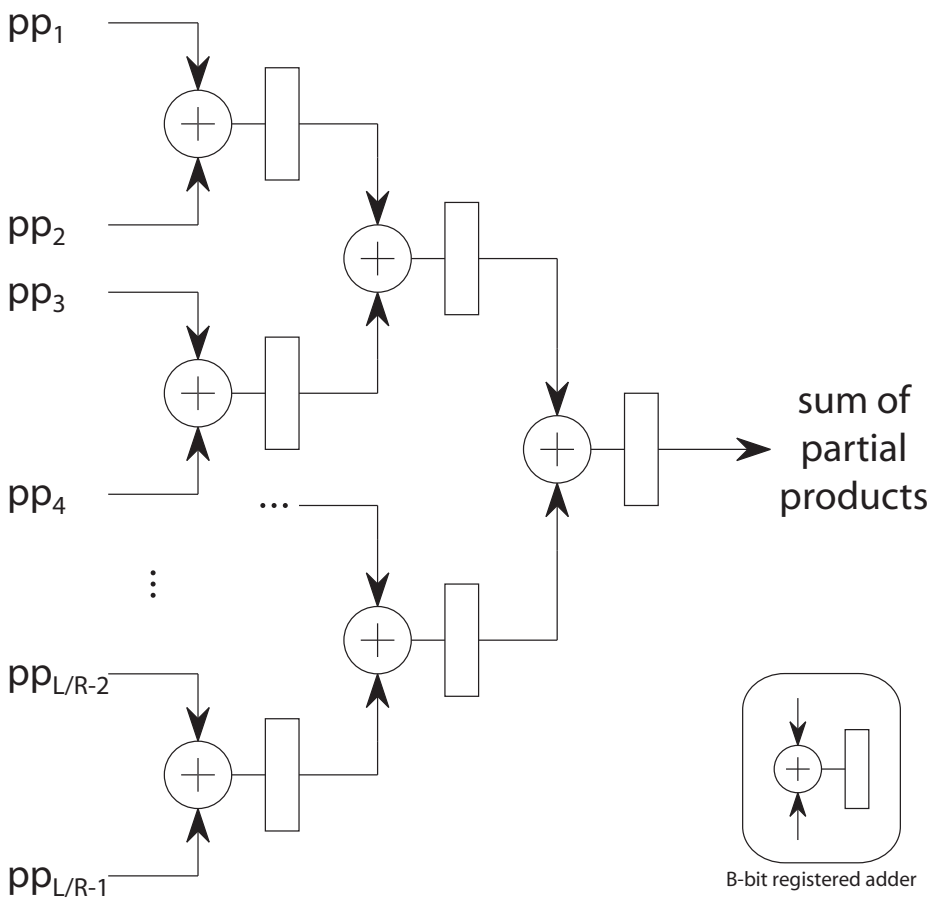


Figure 2.3. Block diagram of the RDA summation block. The adders area arranged in a tree configuration to minimize latency.

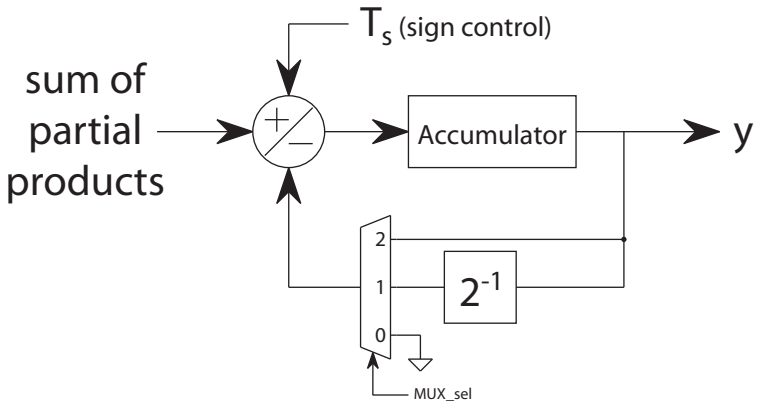


Figure 2.4. Block diagram of the RDA accumulation block. The inclusion of the multiplexer allows the accumulation to complete using only a single accumulator.

The combination of the multiplexed feedback accumulator and the coefficient driven partial product generator is what enable the reuse of all the computation blocks of RDA. Applying reusability to the traditional DA approach would require switching logic for both the input samples and the coefficients. In addition, since the lookup tables contain all the possible combinations of the coefficients, the number of required lookup tables would increase by the reuse factor. The coefficients are stored in an already multiplexed fashion in RDA so extra logic is not required for switching. RDA also does not require lookup tables, as such the increase in memory storage does not increase as quickly as the traditional DA approach.

2.3 FIR Filtering with Multipliers

Typical implementations of the FIR filtering equation utilize one or multiple multipliers, depending on the throughput requirements of the application. A multiplier-based FIR filter is constructed for comparison with the RDA approach to FIR (RDA-FIR) filtering. The multiple multiplier design is termed MM-FIR.

The multiple multiplier design consists of memories containing the coefficients and input samples. Instead of having multiple MAC units, multipliers are used, with a single accumulator as compared to multiple accumulators. Having a single or multiple accumulators does not change the operation of the MM-FIR filter, however by separating the

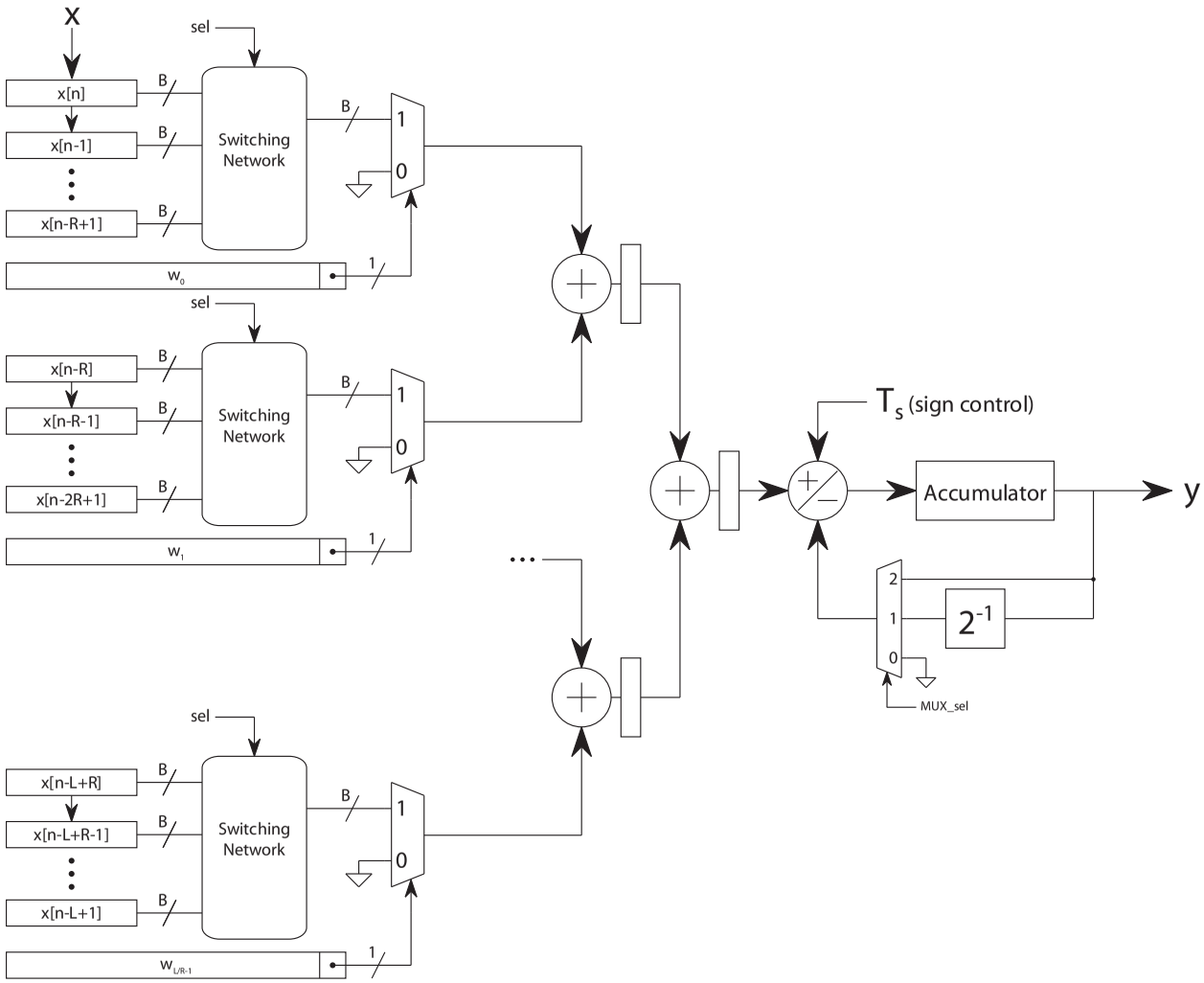


Figure 2.5. Block diagram of a *L*-tap RDA FIR filter.

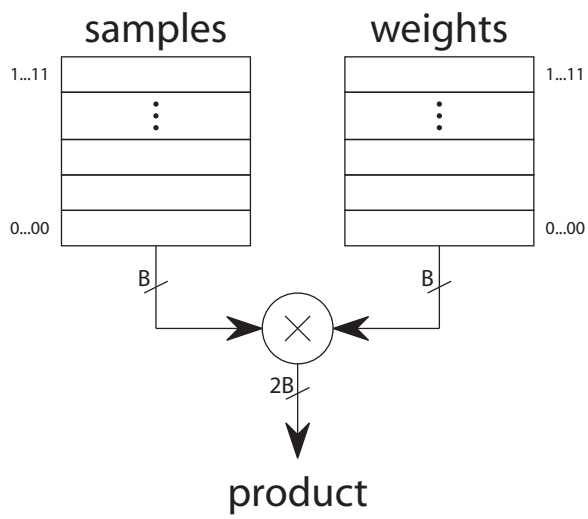


Figure 2.6. Block diagram of the MM multiplier and memories block.

accumulator from the MAC and pushing it to the end of the data path reduces the area of the design.

The multipliers compute in parallel, each multiplier requiring a memory for coefficients and a memory for input samples. The figure for the multiplier memories combination block is shown in Fig. 2.6. In contrast to the RDA-FIR system where the throughput is proportional to the bit precision, the throughput of an MM-FIR system is proportional to the number of times the multipliers are used. By having a large number of multipliers, the MM-FIR design can achieve high sampling rates.

The same location in all the coefficient memories is accessed simultaneously with a circular pointer system for incrementing the address during the filtering operation. After a full iteration, the pointer returns to the starting address. A similar pointer system is used for the memories holding the input samples.

As with the coefficient memories, the same location in all input sample memories is accessed simultaneously. The pointer loops through the $\frac{L}{M}$ addresses during a sample period. The stopping address of the pointer becomes the start address for the next sample period. This is important since the oldest input samples in each memory block are shifted to the next memory block. A block diagram illustrating this process is shown in Fig. 2.7. The

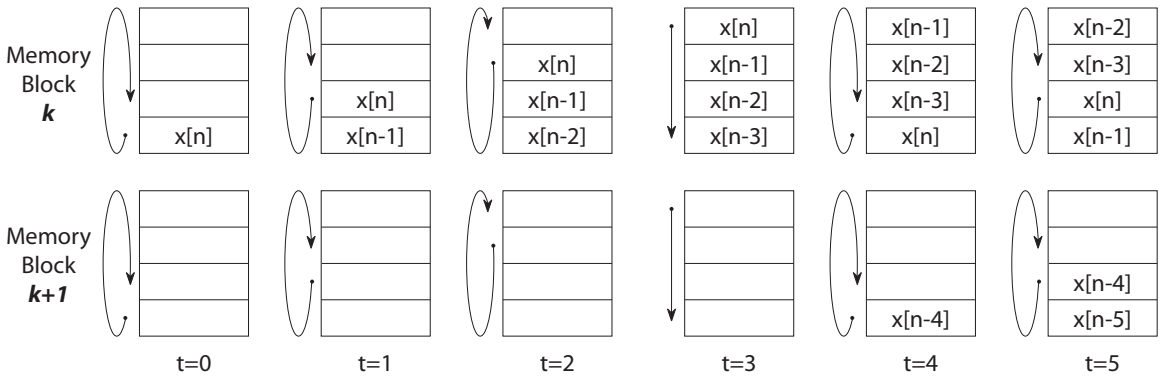


Figure 2.7. MM input samples memory update sequence. Each memory stores $\frac{L}{M}$ words, and each entry is updated with the entry from the previous memory block.

input samples are passed from one memory to the next at the sample clock.

Adders arranged in a tree structure are used to sum the products generated by the multipliers. The output of the adder tree is fed into the accumulator, with the output ready after $\frac{L}{M}$ cycles. This portion of the MM design is similar to the RDA design. The block diagram of the MM-FIR filter is shown in Fig. 2.8.

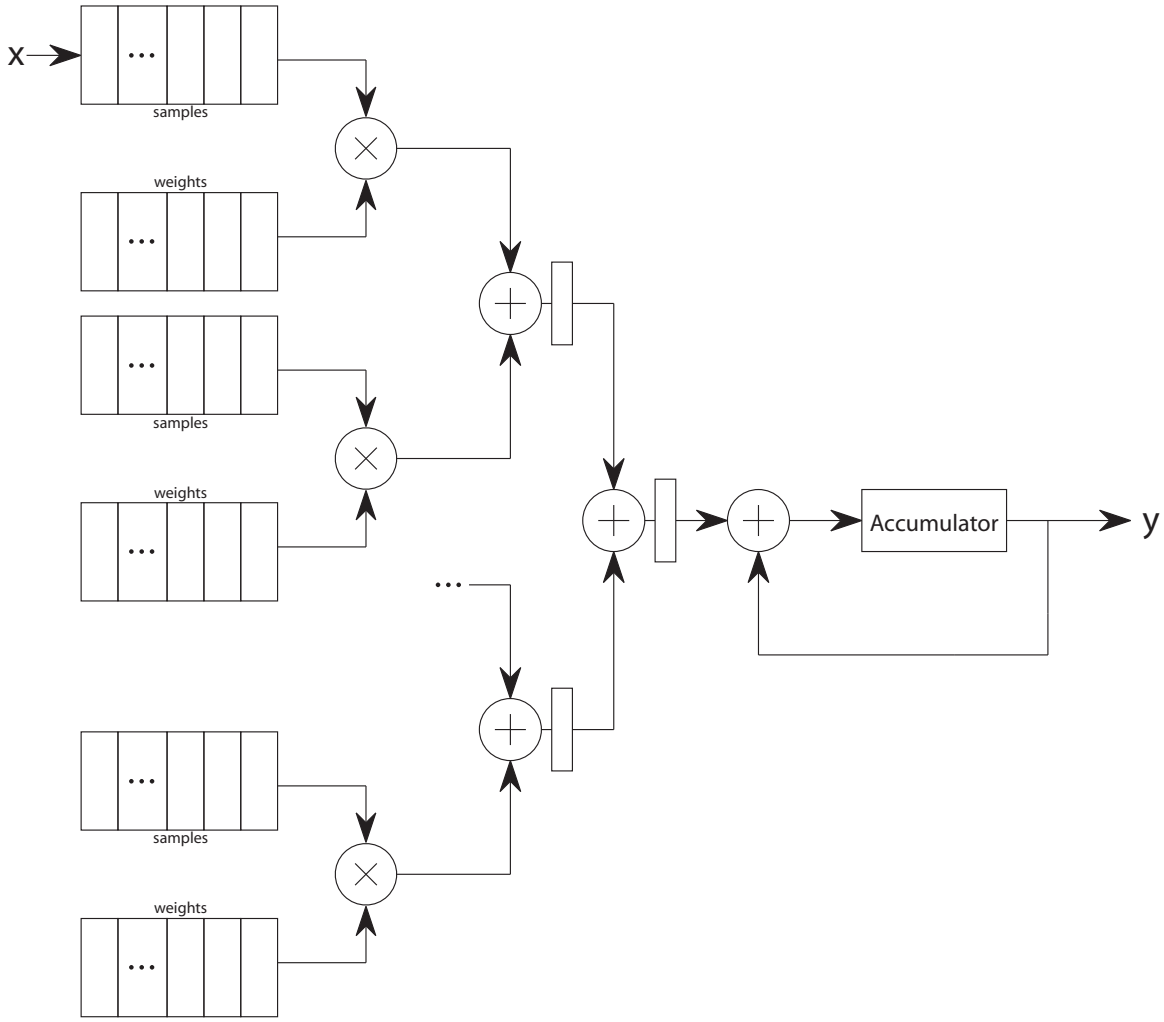


Figure 2.8. Block diagram of the MM FIR filter.

CHAPTER 3

COMPARISON OF THE RDA AND MM FIR FILTERS

The RDA and MM design approaches are used to implement FIR filters of varying filter lengths. The two designs are compared in terms of throughput, latency, and area. Throughput is defined as the number of samples processed per unit time. Latency is defined as the amount of time required to generate an output sample once the input sample has arrived. Area is defined as the amount of resources needed to implement the design.

From the models presented in the previous chapter, the throughputs of the RDA-FIR and MM-FIR systems are given by,

$$\text{Throughput}_{\text{RDA}} = [B]^{-1} \tag{3.1}$$

$$\text{Throughput}_{\text{MM}} = \left[\frac{L}{M}\right]^{-1} \tag{3.2}$$

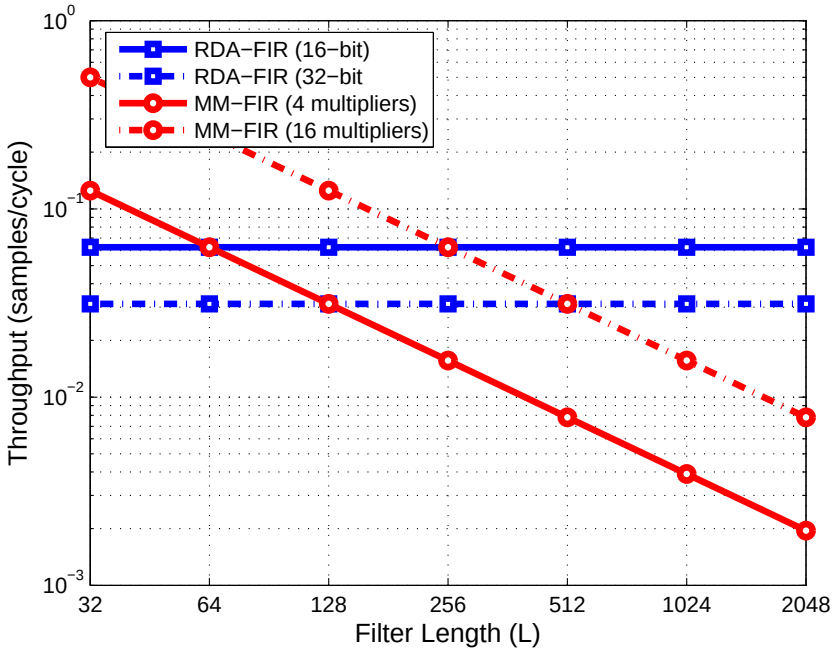
and the latencies are given by,

$$\text{Latency}_{\text{RDA}} = B + \log_2(L) \tag{3.3}$$

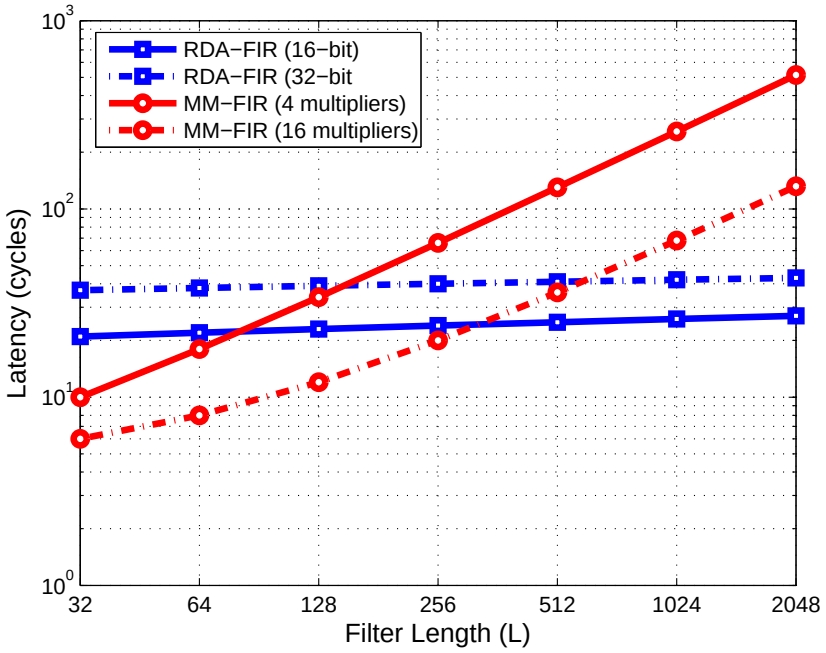
$$\text{Latency}_{\text{MM}} = \frac{L}{M} + \log_2(M) \tag{3.4}$$

where B is the bit precision, L is the filter length, and M is the number of multipliers. The throughput and latency for both designs are shown in Fig. 3.1(a) and 3.1(b) respectively. The number of multipliers was selected as $M = 4$ and $M = 16$, and the bit precision was varied between $B = 16$ and $B = 32$. The plots show the relationship between B, M, and L on throughput and latency of both designs.

Equations 3.1 to 3.4 represent the ideal throughput and latency, since the actual time required to carry out the arithmetic operations are not captured. As such, a more quantitative analysis of the two designs involves mapping the models to a gate-level description.



(a)



(b)

Figure 3.1. Ideal throughput and latency plots for RDA-FIR and MM-FIR.

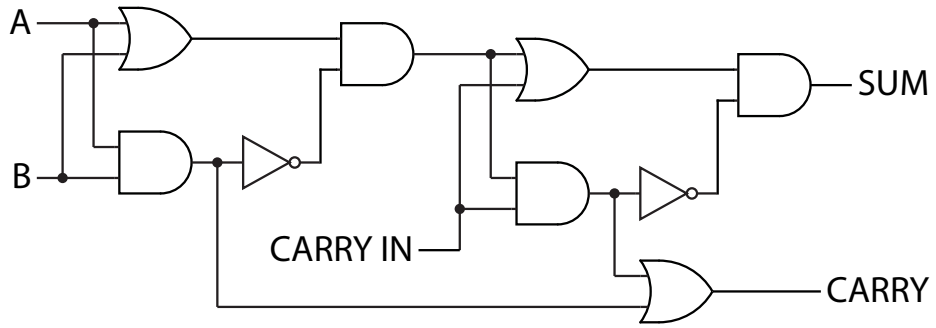


Figure 3.2. Full adder built from nine gates.

3.1 Gate Implementations of Arithmetic Logic

Digital hardware can be described in terms of logic gates. For example, FPGA synthesis results generally report a *total equivalent gate count* which provides a measure of the hardware complexity of the design. The information available from a gate-level analysis includes gate delay and gate count. Gate delay provides timing information, while the gate count provides area resource information for a particular logic function. Using the gate delay and gate count information of the arithmetic logic blocks enables a more quantitative comparison between the RDA-FIR and MM-FIR designs. The gate delay and gate count metrics are suitable for first-order comparisons, although the use of exact models is required for more accurate comparisons.

An example of gate delay and gate count information is presented using a 1-bit full adder. The schematic of the 1-bit full adder is shown in Fig. 3.2. It can be seen from Fig. 3.2 that the full adder is composed of 9 gates. The 1-bit full adder has two outputs, therefore there is a gate delay associated with each output. The gate delays for the SUM and CARRY outputs are 6 and 5 respectively. The delay paths are marked in Fig. 3.3, illustrating the critical path of the SUM and CARRY outputs.

The arithmetic components of the RDA-FIR and MM-FIR designs are adders and multipliers. The adders used in both designs are of the carry-lookahead type [46][47]. Carry-lookahead adders (CLA) are faster than the ripple-carry adders, and slightly more complex

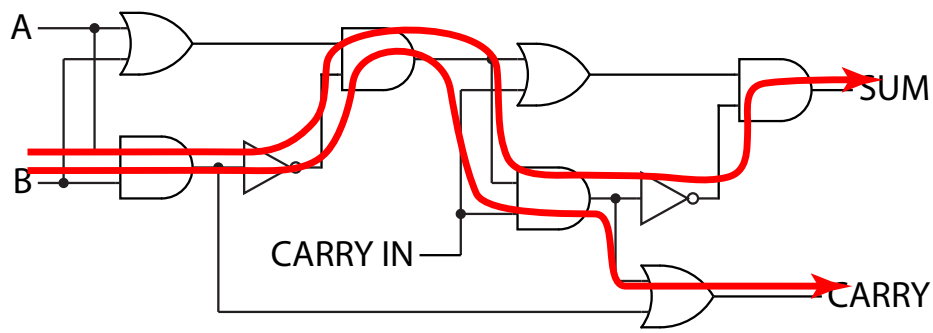


Figure 3.3. Full adder built from nine gates with delay paths marked.

than the ripple-carry adders. The adders are present in the summation block of both designs. The relation between gate count and gate delay of the CLA to bit precision is listed with the following equations [48].

$$T_{\mathrm{CLA}}(B) = 2 + 4\lceil \log_r(B) \rceil \tag{3.5}$$

$$A_{\mathrm{CLA}}(B) = 8B + \frac{1}{2}\left(3r + r^2\right)\left\lceil \frac{B-1}{r-1} \right\rceil \tag{3.6}$$

The term r refers to the r-bit lookahead logic block used, and usually $r = 4$. $T_{\mathrm{CLA}}(B)$ is the gate delay, and $A_{\mathrm{CLA}}(B)$ is the gate count for a B-bit carry-lookahead adder.

The multipliers are of the Dadda type. Dadda multipliers are fast multipliers that use an adder tree constructed with carry-save adders [49][50]. The Dadda multiplier has three steps in computing the multiplication. The two inputs are first used to create a matrix of bit products using an array of AND gates. Then the bit product matrix is reduced using the Dadda compression tree until two rows remain. The two rows are then summed together using a fast adder, in this case a CLA. The equations for gate delay and gate count for a B-bit Dadda multiplier with a 4-bit lookahead CLA are listed below [48].

$$T_{\mathrm{Dadda}}(B) = 1 + 6\left(\log_{1.44}(B) - 2\right) + 2 + 4\lceil \log_4(2B-2) \rceil \tag{3.7}$$

$$A_{\mathrm{Dadda}}(B) = 10B^2 - 4B - 26 \tag{3.8}$$

Figures 3.4 and 3.5 shows the dependence of gate delay and gate count on bit precision for carry-lookahead adder, ripple-carry adder (RCA), and the Dadda multiplier. From the

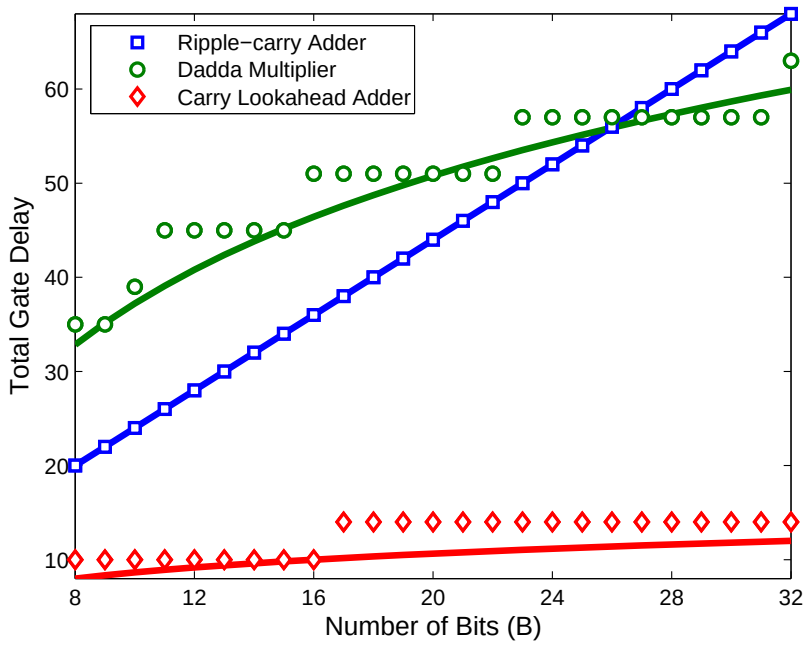


Figure 3.4. Gate delays of different arithmetic components.

gate delay plot it can be seen that the Dadda multiplier is faster than a ripple carry adder when the bit precision is large ($B > 27$). The logarithmic gate delay behavior of the carry-lookahead adder and Dadda multiplier can also be seen. The discrete points on the plot represent the actual gate delay, while the solid lines illustrate the trend of the gate delay versus bit precision.

3.2 Gate-level Comparison of RDA-FIR and MM-FIR

Using the gate delay equations, the arithmetic component that takes the longest amount of time for the RDA-FIR design is the adder at the output of the summation block, whereas for the MM-FIR design the slowest component is the multiplier. The ideal equations (Eqns. 3.1 to 3.4) for throughput and latency are modified to include the gate delay information. The resulting throughput and latency relationships are listed below.

RDA-FIR:

$$\text{Throughput}_{\text{RDA}} = [T_{\text{CLA}}(B + \log_2(L)) \times B]^{-1} \tag{3.9}$$

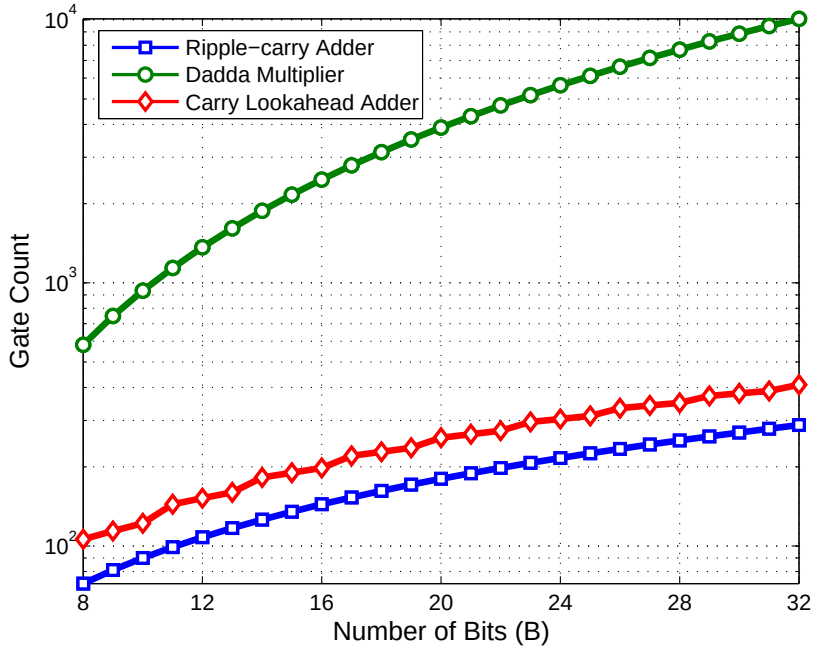


Figure 3.5. Gate counts of different arithmetic components.

$$\text{Latency}_{\text{RDA}} = T_{\text{CLA}}(B + \log_2(L)) \times (B + \log_2(L)) \tag{3.10}$$

MM-FIR:

$$\text{Throughput}_{\text{MM}} = \left[T_{\text{Dadda}}(B) \times \frac{L}{M}\right]^{-1} \tag{3.11}$$

$$\text{Latency}_{\text{MM}} = T_{\text{Dadda}}(B) \times \left(\frac{L}{M} + \log_2(M)\right) \tag{3.12}$$

The designs are then configured such that they operate at the same throughput for the minimum required area. This is performed by computing the number of multipliers, M, needed to ensure identical throughput. M is obtained by setting the throughput equations of the RDA-FIR and MM-FIR designs equal to each other.

$$M = \frac{L}{B} \cdot \frac{T_{\text{Dadda}}(B)}{T_{\text{CLA}}(B + \log_2(L))} \tag{3.13}$$

Substituting Eqn. 3.13 into the latency equation for MM-FIR,

$$\text{Latency}_{\text{MM}} = T_{\text{Dadda}}(B)\left[B\frac{T_{\text{CLA}}(B + \log_2(L))}{T_{\text{Dadda}}(B)} + \log_2\left(\frac{L}{B}\frac{T_{\text{CLA}}(B + \log_2(L))}{T_{\text{Dadda}}(B)}\right)\right] \tag{3.14}$$

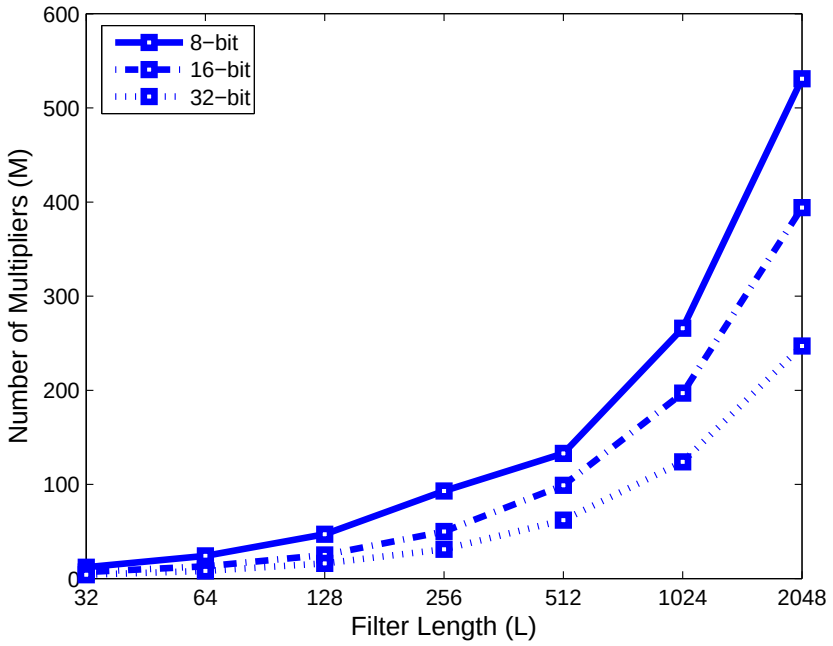
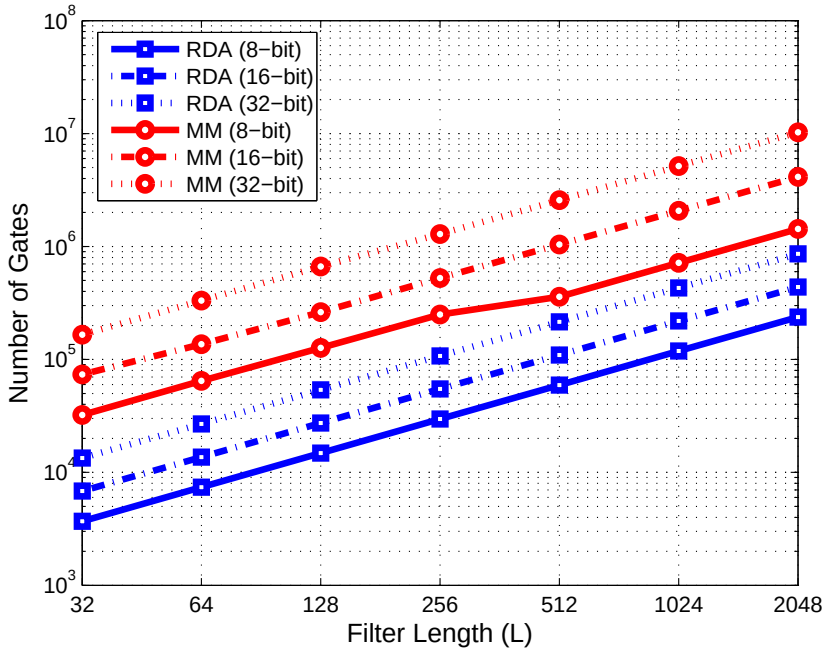


Figure 3.6. Number of multipliers needed for MM-FIR.

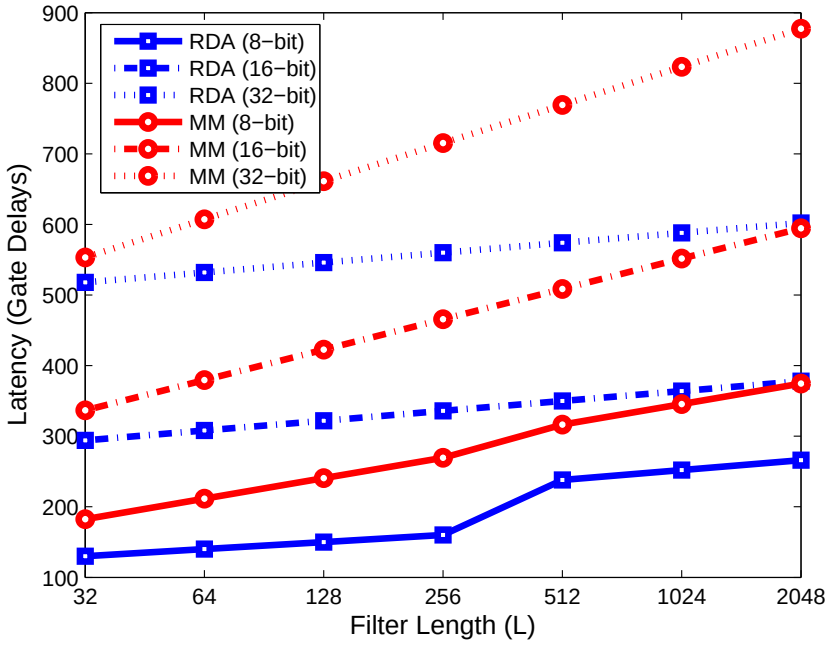
Figure 3.6 shows the number of multipliers needed to maintain identical throughput with the RDA-FIR design for varying filter length and bit precision. As filter length increases, the number of multipliers also increase, eventually requiring too many for practical hardware implementations. The number of multipliers also increases as the bit precision decreases. This is attributed to the throughput of the RDA-FIR design being inversely proportional to the bit precision. As bit precision decreases, the number of samples between clock cycles for the RDA-FIR filter increases, causing the MM-FIR filter to require more multipliers to maintain the same throughput.

The latency of the two designs is shown in Fig. 3.7(b). The results show that the MM-FIR design has a higher latency than the RDA-FIR design for all filter lengths and bit precisions. Again, this is due to the increase in number of multipliers to match the throughput of the RDA-FIR design.

The gate count results complete the gate-level comparison between the RDA-FIR and



(a)



(b)

Figure 3.7. Gate count and latency for MM-FIR.

MM-FIR design. The number of arithmetic components for both designs are added together, and using Eqns. 3.6 and 3.8, the total gate counts are computed and the results shown in Fig. 3.7(a). The gate count results take into account only the arithmetic logic blocks used, since the memory/register sizes used by both designs are identical. Not included as well are the controlling hardware blocks that regulate the filtering operation. The arithmetic blocks are sized properly to avoid any overflows. For example, for every level further down the adder tree in the summation block, the bit precision of the adder(s) at that level increase by a single bit. The MM-FIR design requires a larger number of gates than the RDA-FIR design. This is expected considering the area of a Dadda multiplier is proportional to B^2 while the area of a carry-lookahead adder is proportional to B. The MM-FIR design is on average ~10 times larger than the RDA-FIR design.

The performance of the MM-FIR filter can be improved if the limiting factor, the Dadda multiplier, is pipelined. Pipelining improves the throughput of a system by splitting a long computation process into a series of shorter processes that can then be executed concurrently [51]. The new throughput and latency equations for the pipelined MM-FIR (PMM) filter are listed below.

$$\text{Throughput}_{\text{PMM}} = \left[T_{\text{CLA}}(2B + \log_2(M)) \times \frac{L}{M}\right]^{-1} \tag{3.15}$$

$$\text{Latency}_{\text{PMM}} = T_{\text{CLA}}(2B + \log_2(M))\left(\frac{L}{M} + \log_{1.5}(B) + \log_2(M)\right) \tag{3.16}$$

The throughput of the PMM-FIR filter is dependent on the gate delay of the carry-lookahead adder instead of the multiplier, since the multiplier has been pipelined and split into smaller, more basic arithmetic logic blocks. The latency of the PMM design includes the contribution from the pipelined stages of the Dadda compression tree.

The throughput of the PMM-FIR design is again set equal to the throughput of the RDA-FIR design. Solving for the number of multipliers, M, to obtain the following equation.

$$M = \frac{L}{B} \times \frac{T_{\text{CLA}}(2B + \log_2(M))}{T_{\text{CLA}}(B + \log_2(L))} \tag{3.17}$$

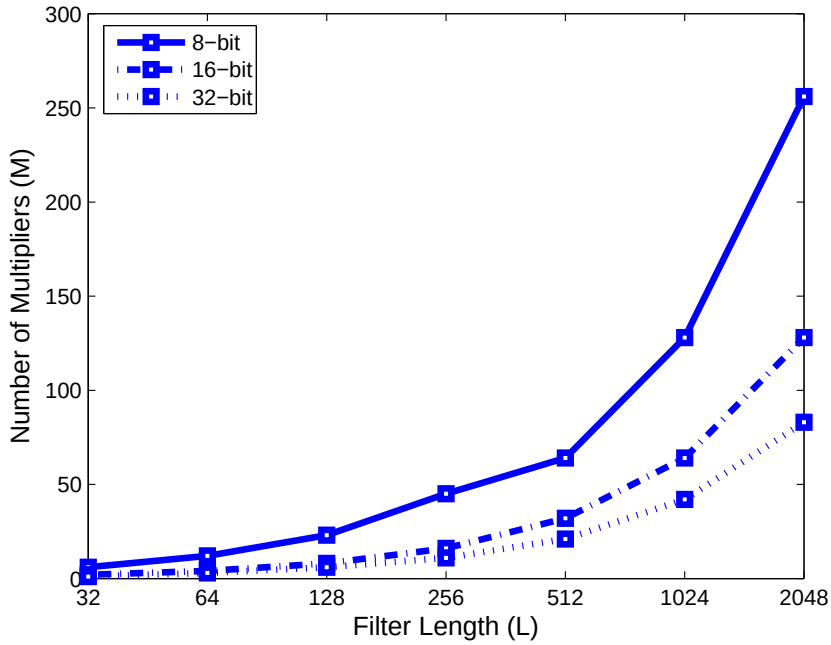


Figure 3.8. Number of multipliers needed for PMM-FIR.

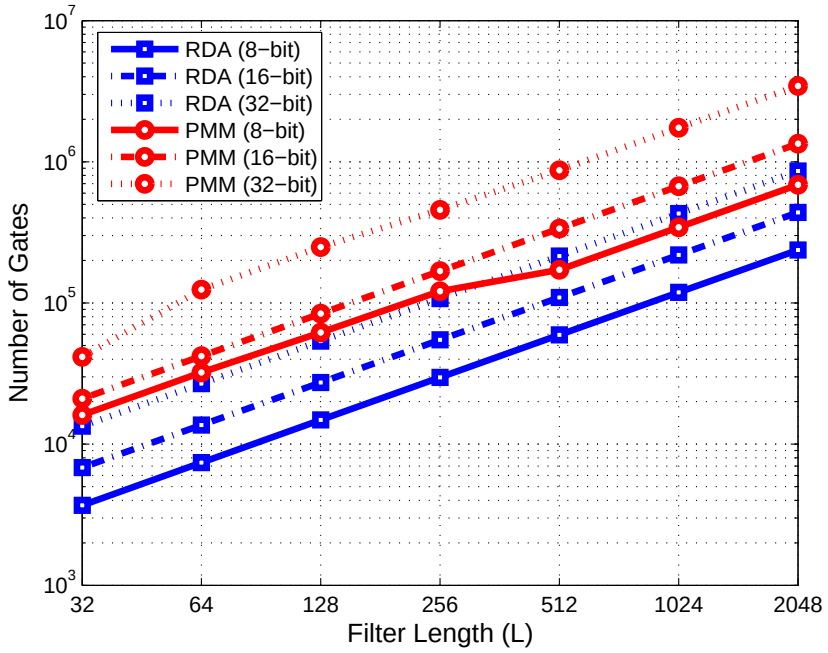
Substituting in Eqn. 3.5,

$$M = \frac{L}{B} \times \frac{2 + 4\lceil \log_4(2B + \log_2(M)) \rceil}{T_{\mathrm{CLA}}(B + \log_2(L))} \tag{3.18}$$

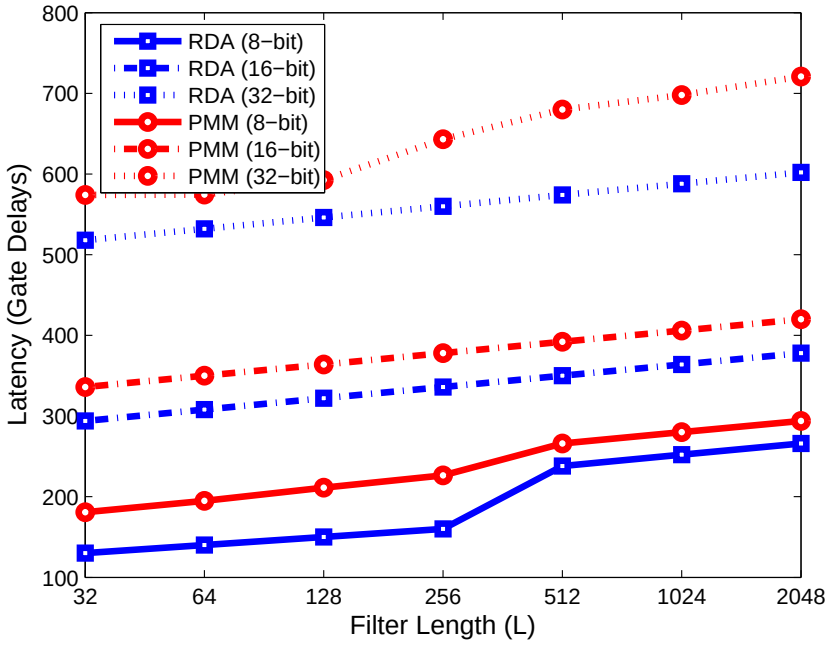
It can be seen that the term M appears on both sides of the equation. M is obtained by iteratively solving Eqn. 3.18. An initial guess is made for M, and the calculated result is used to compute the next value of M. The process repeats until the value of M is constant. Figure 3.8 shows the number of multipliers need by the PMM-FIR design to maintain the same throughput as the RDA-FIR design.

When compared against the number needed for the MM-FIR design, the pipelined design requires fewer multipliers because of the improved throughput from pipelining. The reduced number of multipliers translates to the gate count of the PMM-FIR design being lower than the MM-FIR design. However, the gate count of the PMM-FIR is still higher than that of the RDA-FIR design, as shown in Fig. 3.9(a). The PMM-FIR design is on average ~4 times larger than the RDA-FIR design.

The latency of the PMM-FIR design is lower than the MM-FIR design and more closely



(a)



(b)

Figure 3.9. Gate count and latency for PMM-FIR.

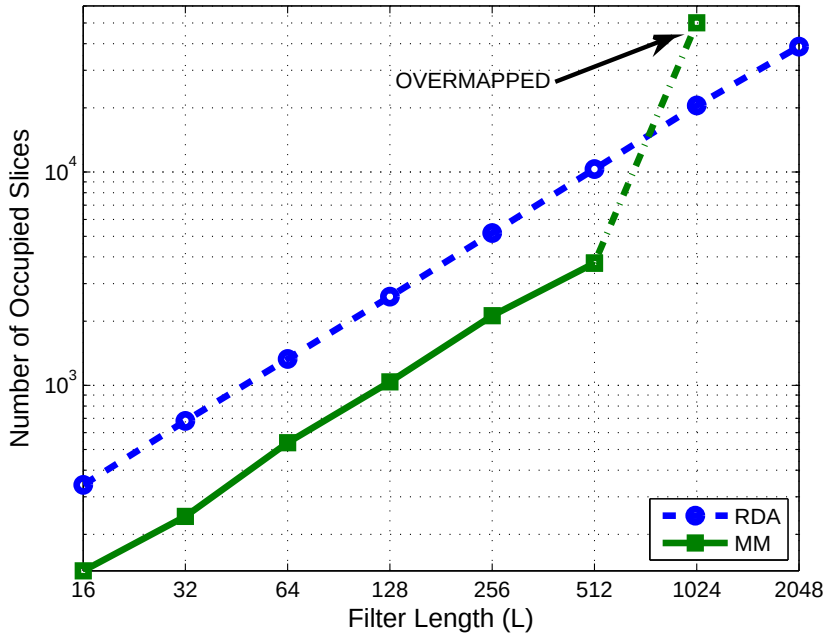
matches that of the RDA-FIR design. However, the latency of the PMM-FIR design is still larger than the RDA-FIR design for all values of B and L, as illustrated in Fig. 3.9(b).

3.3 FPGA Comparison of RDA-FIR and MM-FIR

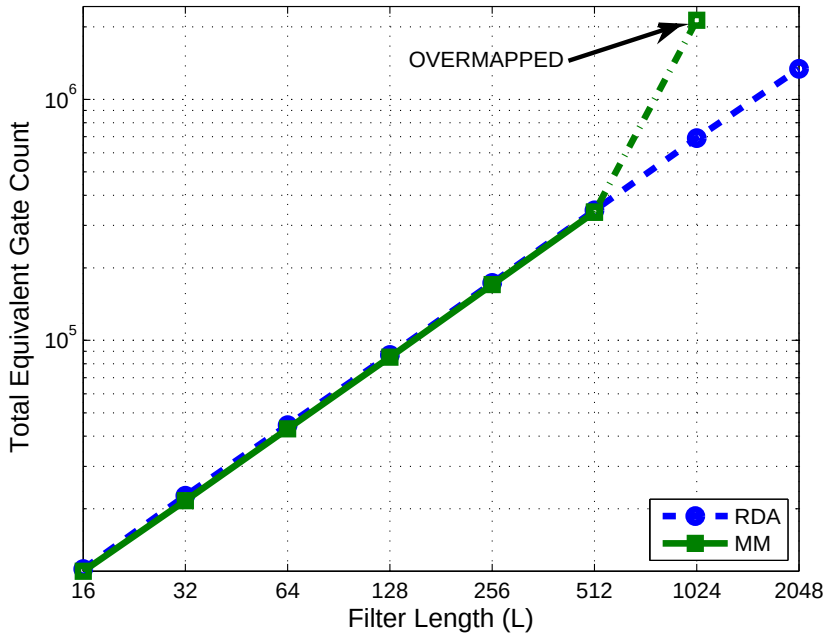
The RDA-FIR and MM-FIR designs were modeled using VHDL and compared using a Xilinx Virtex-4 LX100 (XC4VLX100) FPGA [45]. The selected FPGA offers a good balance between available number of logic elements and hardware multipliers. In particular the FPGA has 96 *XtremeDSP* slices which can be used to implement high speed multipliers. The VHDL models for both designs were synthesized using *Synplify Pro* and then place-and-routed using Xilinx ISE. The two designs were configured for 16-bit precision for filter lengths ranging from 16 taps to 2048 taps. The maximum sampling frequency of the RDA-FIR filter at each filter tap was obtained, and the MM-FIR filter was configured accordingly to have the same sampling frequency.

The number of slices occupied by the RDA-FIR and MM-FIR designs illustrate the area resources needed by the two designs. The number of available slices is determined by the specific FPGA, and serves to limit the maximum synthesizable FIR filter for both RDA-FIR and MM-FIR approaches. However, the slice count does not include the resources needed by the dedicated hardware multipliers of the MM-FIR design. As such, the total equivalent gate count serves as a more accurate metric for comparing the two designs.

The comparison results (Tbl. 3.1) show that when dedicated hardware multipliers are available on the FPGA, the total equivalent gate count for the MM-FIR approach is lower than the RDA-FIR approach. This is unsurprising as the hardware multipliers were custom designed for maximum performance for minimum area. However once the number of available multipliers was exhausted, Xilinx will construct multipliers out of the logic fabric of the Virtex-4. This immediately reduces the performance of the MM-FIR design and increases area resources tremendously, ultimately running out of available hardware resources for increasing filter lengths (Figs. 3.10(a) and 3.10(b)).



(a) Occupied slices for varying L.



(b) Equivalent gate count for varying L.

Figure 3.10. (a) and (b) RDA-FIR and MM-FIR comparison results using a Xilinx Virtex-4 LX100 FPGA. The two designs were configured to have the same throughput for varying filter lengths. The results are obtained from synthesis with *Synplify Pro* and place-and-route with Xilinx ISE.

For synthesizable filter lengths, the MM-FIR design has a slightly lower total equivalent gate count than the RDA-FIR approach. The RDA-FIR filter is on average 2.66 percent larger than the MM-FIR filter, however a significantly larger filter (2048 taps) is capable of being placed onto the Virtex-4 LX100 FPGA than with the MM design (512 taps).

Examining Fig. 3.6 it can be seen that for a 512-tap FIR filter with 16-bit precision the MM-FIR design requires approximately 100 multipliers, which is approximately the same number as that available on the Virtex-4 LX100 FPGA. A larger FIR filter of the same precision would require more than 100 multipliers, which corresponds to the MM-FIR design being unable to fit onto the FPGA. The results from the FPGA synthesis comparison show that the using gate-level descriptions provide reasonably accurate comparisons. The use of gate-level metrics is carried throughout the rest of this research.

Table 3.1. Comparison of FPGA implementation results for varying filter lengths L

Xilinx Virtex-4 LX100		**Filter Length (L)**							
		16	32	64	128	256	512	1024	2048
RDA	Number of Occupied Slices	341	684	1329	2604	5173	10328	20522	38780
	Total Equivalent Gate Count	11245	22651	44505	87006	173226	346130	690346	1339328
MM	Number of Occupied Slices	135	243	538	1038	2122	3739	N/A	N/A
	Total Equivalent Gate Count	11020	21608	42922	85208	170559	340267	N/A	N/A

*N/A means the design was unable to fit on the FPGA.

CHAPTER 4

RDA AND THE FAST AFFINE PROJECTION ALGORITHM

Recently several methods for computing the fast affine projection have been introduced suitable for fixed-point implementations, differing primarily in how the matrix inversion problem is solved, as discussed in Chapter 1, Section 1.2. For example, the conjugate gradient FAP (CGFAP) [34] and Gauss-Seidel FAP (GSFAP) [39] approach the matrix inversion problem by reducing the problem into solving a linear system of equations using the conjugate gradient method and Gauss-Seidel method respectively. Oh, *et al.* assume the matrix is Toeplitz and solve for the inverse using the Levinson-Durbin recursion [35]. The majority of the proposed FAP algorithms have been implemented using digital signal processors (DSPs) because of the divisions involved in the coefficient update, multiple computations, and the non-trivial datapath of the algorithm. The sequential execution nature coupled with the large number of computations restrict the maximum throughput attainable by DSPs.

In this chapter a hardware-efficient, high-throughput, low-latency, fixed-point implementation of the FAP algorithm based on the Gauss-Seidel approach is presented. The proposed architecture achieves high throughput and low complexity in three ways: 1) by using reusable distributed arithmetic [45], 2) using single iteration LUT-based division, and 3) parallelization of the coefficient update computations. This proposed architecture is termed RDA-FAP. Construction of the RDA-FAP adaptive filter is preceded by the design and implementation of the DA-NLMS algorithm. It was shown in Section 1.2 that the FAP algorithm is a generalization of the NLMS algorithm. As such, developments from the DA-NLMS adaptive filter can and are applied to the RDA-FAP adaptive filter.

Implementation of an adaptive filter using the traditional distributed arithmetic architecture (Fig. 1.4) presents a unique challenge. The lookup table of Fig. 1.4 contains not only the coefficients, but the combinations of the coefficients as well (Fig. 1.4). A

direct implementation would require the entire table to be updated at every sample, significantly increasing the computational complexity of the adaptive filter. By using the RDA architecture, the increase in complexity can be avoided. This is attributed to an important feature of RDA, the linear dependence of memory on filter length. An adaptive filter with L coefficients would require only L instead of 2^L coefficients to be updated when using RDA. Combining the minimal coefficient update with the fast throughput and low latency of properties of RDA enables the design of fast adaptive filters [52].

4.1 NLMS using Distributed Arithmetic

Revisiting the coefficient calculation and update equations for the affine projection algorithm from Tbl. 1.4,

$$\mathbf{w}[n] = \mathbf{w}[n-1] + \mu\mathbf{X}[n]\left(\mathbf{X}^T[n]\mathbf{X}[n] + \delta\mathbf{I}\right)^{-1}\mathbf{e}[n] \tag{4.1}$$

where

$$\mathbf{X}[n] = \Big[\mathbf{x}[n], \mathbf{x}[n-1], \ldots, \mathbf{x}[n-p+1]\Big] \tag{4.2}$$

$$\mathbf{x}[n] = \Big[x[n], x[n-1], \ldots, x[n-L+1]\Big]^T \tag{4.3}$$

For the case where $p = 1$ and $0 < \delta \ll 1$, Eqn. 4.1 reduces to the weight update equation for the NLMS algorithm shown below.

$$\mathbf{w}[n] = \mathbf{w}[n-1] + \mu e[n]\frac{\mathbf{x}[n]}{||\mathbf{x}[n]||^2} \tag{4.4}$$

NLMS using distributed arithmetic can be achieved by extending the work accomplished with LMS using DA [53][54]. The DA-LMS adaptive filter proposed in [54] simplifies the weight update equation of the LMS algorithm. The LMS weight update equation from Tbl. 1.2 is presented below.

$$\mathbf{w}[n] = \mathbf{w}[n-1] + \mu e[n]\mathbf{x}[n] \tag{4.5}$$

The term $\mu e[n]$, where μ is the adaptation step size and $e[n]$ is the error, is multiplied with the input sampes $\mathbf{x}[n]$. Instead of computing the multiplication, the term $\mu e[n]$ is converted into a right shift operation on the bits of $\mathbf{x}[n]$, with the shift amount determined by the absolute value of $e[n]$.

Comparing Eqns. 4.4 and 4.5, the two weight update equations are identical except for the $||\mathbf{x}[n]||^2$ term. This term can be considered as an modulating factor of $\mu e[n]$. If the value of $||\mathbf{x}[n]||^2$ is small, the net result would be a larger value of $\mu e[n]$, which corresponds to a larger right shift amount on $\mathbf{x}[n]$, and vice versa. In general, the magnitude of the term $\mu e[n]$ generates a right shift (positive), and the magnitude of $||\mathbf{x}[n]||^2$ generates a left shift (negative). The resulting shift amount that is applied to $\mathbf{x}[n]$ is the sum of the two shifts. The DA-NLMS adaptive filter can thusly be obtained from the DA-LMS filter proposed in [54] by modifying only the logic that determines the shift amount, leaving the remaining architecture unchanged.

The term $||\mathbf{x}[n]||^2$, which is the signal energy of $\mathbf{x}[n]$, can be computed using an estimate,

$$\epsilon[n] = \beta\epsilon[n-1] + (1-\beta)|x[n]|^2, \quad 0 < \beta < 1 \tag{4.6}$$

For the case of the DA-NLMS adaptive filter, for simplicity of implementation the signal energy estimate is,

$$\epsilon[n] = \beta\epsilon[n-1] + (1-\beta)|x[n]|, \quad 0 < \beta < 1 \tag{4.7}$$

where β is expressed as,

$$\beta = 2^{-k}, \quad 0 < k < \infty \tag{4.8}$$

The reason for β being an inverse power of 2 is so that $\beta\epsilon[n-1]$ and $(1-\beta)|x[n]|$ can be implemented as shift right operations on $\epsilon[n-1]$ and $|x[n]|$, respectively. The $|\cdot|^2$ operation on $x[n]$ has been simplified to just performing the absolute value, $|\cdot|$. The approximations made in the implementation of the DA-NLMS adaptive filter are summarized below.

1. Quantization of $\mu e[n]$ into a shift right operation on $\mathbf{x}[n]$

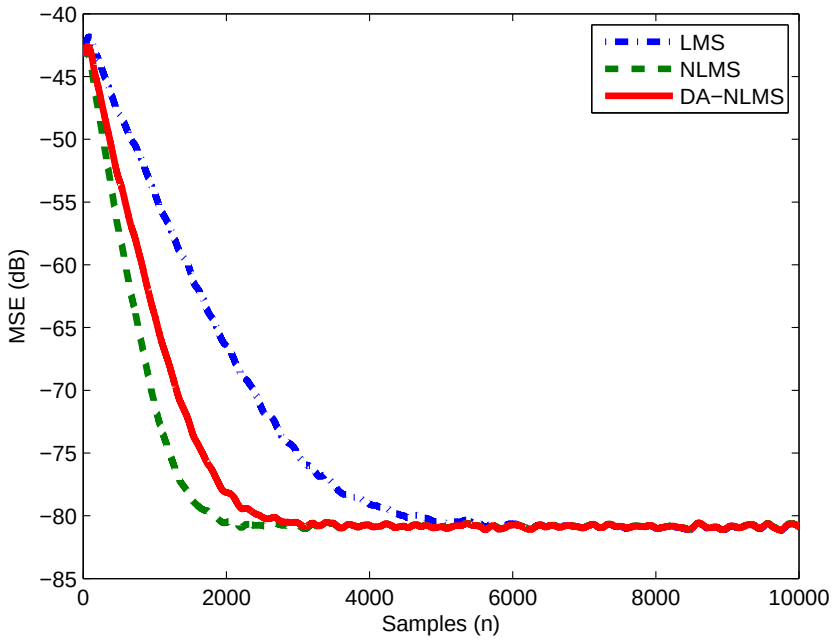


Figure 4.1. Simulation results of LMS, NLMS, and DA-NLMS from a system identification configuration with slightly correlated inputs. The convergence curves are obtained from averaging 100 runs.

2. Quantization of $||x[n]||^2$ into a shift left operation $\mathbf{x}[n]$

3. Simplification of the signal energy estimate calculation for $||\mathbf{x}[n]||^2$

Simulation results comparing the convergence behavior of LMS, NLMS, and DA-NLMS are shown in Fig. 4.1. The results show that the approximations degrade slightly the performance of the NLMS algorithm when compared to the full implementation version. However, the DA-NLMS algorithm still converges faster than the LMS algorithm.

The approximations of DA-NLMS also reduces the computational complexity of the adaptive algorithm. Table 4.1 lists the number of math operations required by the NLMS and DA-NLMS algorithms for one output sample. Table 4.1 does not represent the actual implementation of the algorithms, but the types of computations involved. The operations are grouped into three categories: addition (ADD), multiplication (MULT), and division (DIV). The step size μ is selected as an inverse power of 2 so that it can be implemented as a shift right operation for NLMS and DA-NLMS. The term $||\mathbf{x}[n]||^2$ is implemented using

Table 4.1. Number of operations sorted by type for NLMS and DA-NLMS

	NLMS			DA-NLMS*		
Operation	ADD	MULT	DIV	ADD	MULT	DIV
Filtering	$L-1$	L	0	$L-1$	L	0
Compute Error	1	0	0	1	0	0
Update Calculation	1	$L+1$	1	1	0	0
Coefficient Update	L	0	0	L	0	0
Total	$2L+1$	$2L+1$	1	$2L+1$	L	0

* Types of operations, not actual implementation

Eqn. 4.6 for NLMS. From Tbl. 4.1 the DA-NLMS has fewer operations than NLMS, and also removes the difficult and computationally expensive division. The methodologies used in the design of the DA NLMS adaptive filter are carried over into the design of the FAP adaptive filter.

4.2 Fast Affine Projection Algorithm

FIR filters based on reusable distributed arithmetic have been shown to be a hardware-efficient architecture for high order digital filters, requiring fewer resources than the multiplier-based counterpart, and at the same time freeing up those hardware resources to be used for other functions. This hardware efficiency is carried over to the FAP algorithm, reducing the latency and increasing the speed associated with filtering. Further speed increase can be obtained by computing as much of the FAP algorithm in parallel as possible.

The coefficient calculation and update equations for the FAP algorithm from Section 1.2 are presented below.

$$\mathbf{r}[n] = \mathbf{r}[n-1] + x[n]\boldsymbol{\alpha}[n] - x[n-L]\boldsymbol{\alpha}[n-L] \tag{4.9}$$

$$e[n] = d[n] - \mathbf{x}^T[n]\mathbf{w}[n-1] - \mu\bar{\mathbf{r}}^T[n]\underline{\boldsymbol{\eta}}[n-1] \tag{4.10}$$

$$\mathbf{e}[n] = \begin{bmatrix} e[n] \\ (1-\mu)\underline{\mathbf{e}}[n-1] \end{bmatrix} \tag{4.11}$$

$$\boldsymbol{\varepsilon}[n] = \mathbf{P}[n]\mathbf{e}[n] \tag{4.12}$$

$$\boldsymbol{\eta}[n] = \begin{bmatrix} 0 \\ \underline{\boldsymbol{\eta}}[n-1] \end{bmatrix} + \boldsymbol{\varepsilon}[n] \tag{4.13}$$

$$\mathbf{w}[n] = \mathbf{w}[n-1] + \mu\eta_{p-1}[n]\mathbf{x}[n-p+1] \tag{4.14}$$

where $\mathbf{w}[n]$ represents the coefficients of a L-tap filter, p is the projection order, $\mathbf{x}[n] = [x[n], x[n-1], \ldots, x[n-L+1]]^T$, $\mathbf{P}[n] = \mathbf{R}^{-1}[n]$ is the inverse of the $p \times p$ autocorrelation matrix $\mathbf{R}[n]$, $\boldsymbol{\eta}[n]$ is a $p \times 1$ vector, η_{p-1} is the bottommost element of $\boldsymbol{\eta}[n]$, $\underline{\mathbf{e}}[n]$ and $\underline{\boldsymbol{\eta}}[n]$ are the $p-1$ upper elements of $\mathbf{e}[n]$ and $\boldsymbol{\eta}[n]$ respectively.

By setting $\mu = 1$ in Eqn. 4.11, the vector $\mathbf{e}[n]$ is reduced to the scalar $e[n]$, reducing Eqn. 4.12 to,

$$\boldsymbol{\varepsilon}[n] = \mathbf{p}[n]e[n] \tag{4.15}$$

where $\mathbf{p}[n]$ is the leftmost column of $\mathbf{P}[n]$, and

$$\mathbf{R}[n]\mathbf{p}[n] = [1, 0, \ldots, 0]^T \tag{4.16}$$

which is a system of linear equations that can be solved using various methods [55].

4.2.1 Iterative Methods to Solve $\mathbf{R}[n]\mathbf{p}[n] = \mathbf{b}$

For ease of hardware implementation, two iterative methods, the Jacobi and Gauss-Seidel iterations, were considered to solve the linear system of equations represented by,

$$\mathbf{Ax} = \mathbf{b} \tag{4.17}$$

For a $N \times N$ matrix A the Jacobi iteration is,

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \ldots, N \tag{4.18}$$

and the Gauss-Seidel iteration is,

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \ldots, N \tag{4.19}$$

where $\mathbf{x} = [x_1, x_2, \ldots, x_N]^T$. The main difference between the two methods is that for the Jacobi method, $\mathbf{x}^{(k+1)}$ depends only on $\mathbf{x}^{(k)}$, whereas for the Gauss-Seidel method $\mathbf{x}^{(k+1)}$ depends on the already the computed elements of $\mathbf{x}^{(k+1)}$ and elements of $\mathbf{x}^{(k)}$ that have not been used yet. The Gauss-Seidel approach requires less storage than the Jacobi approach because the already used elements of $\mathbf{x}^{(k)}$ are overwritten with the next iteration's values. The Gauss-Seidel approach also typically converges faster. Based on the savings in storage and faster convergence, the Gauss-Seidel iterative method was selected.

4.2.2 Solving for $\mathbf{p}[n]$

The linear system to solve is (Eqn. 4.16) represented by the following ($p = 4$ example)

$$\begin{bmatrix} r_0[n] & r_1[n] & r_2[n] & r_3[n] \\ r_1[n] & r_0[n-1] & r_1[n-1] & r_2[n-1] \\ r_2[n] & r_1[n-1] & r_0[n-2] & r_1[n-2] \\ r_3[n] & r_2[n-1] & r_1[n-2] & r_0[n-3] \end{bmatrix} \begin{bmatrix} p_0[n] \\ p_1[n] \\ p_2[n] \\ p_3[n] \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.20}$$

Using one Gauss-Seidel iteration to solve for $\mathbf{p}[n]$,

$$p_0[n] = \frac{1}{r_0[n]} \tag{4.21}$$

$$p_1[n] = \frac{1}{r_0[n-1]}\Big(- r_1[n]p_0[n]\Big) \tag{4.22}$$

$$p_2[n] = \frac{1}{r_0[n-2]}\Big(- r_2[n]p_0[n] - r_1[n-1]p_1[n]\Big) \tag{4.23}$$

$$p_3[n] = \frac{1}{r_0[n-3]}\Big(- r_3[n]p_0[n] - r_2[n-1]p_1[n] - r_1[n-2]p_2[n]\Big) \tag{4.24}$$

It has been shown in [39] that a single iteration is enough for the FAP adaptive algorithm to converge. From the previous equations it seems that four divisions are needed. In reality only a single division (Eqn. 4.21) is needed at every given sample. The previously computed divisions are stored as reciprocals and reused. In this way for every sample only one division is needed instead of p. The total number of additions, multiplications, and

divisions required by one Gauss-Seidel iteration for projection order p are,

$$\text{Number of additions:} \quad \frac{1}{2}p^2 - \frac{3}{2}p + 1 \tag{4.25}$$

$$\text{Number of multiplications:} \quad \frac{1}{2}p^2 + \frac{1}{2}p - 1 \tag{4.26}$$

$$\text{Number of divisions:} \quad 1 \tag{4.27}$$

4.2.3 Update of $\mathbf{R}[n]$

Inspection of Eqn. 4.20 shows that at every sampling interval, the entries of $\mathbf{R}[n]$ are shifted down and to the right by one element. As such, only the newest/current entries of $\mathbf{R}[n]$ need to be computed. Expanding Eqn. 4.9,

$$r_0[n] = r_0[n-1] + x^2[n] - x^2[n-L] \tag{4.28}$$

$$r_1[n] = r_1[n-1] + x[n]x[n-1] - x[n-L]x[n-L-1] \tag{4.29}$$

$$\vdots$$

$$r_{p-1}[n] = r_{p-1}[n-1] + x[n]x[n-p+1] - x[n-L]x[n-L-p+1] \tag{4.30}$$

For every sampling interval, it is necessary to store the vector

$$\hat{\mathbf{x}}[n] = \Big[x[n], x[n-1], \ldots, x[n-L-p+1]\Big]^T \tag{4.31}$$

which has only p elements more than the vector $\mathbf{x}[n]$. Using this vector, the current entries of $\mathbf{r}[n]$ can be computed. In order to save on redundant multiplications it is possible to store the already computed values of $x[n]x[n-k]$. However this would come at an increase in storage elements, since it is necessary to store $p \times L$ values. Depending on the values of p and L the additional storage requirements could prove prohibitively expensive. Considering that RDA has the ability to maintain a high throughput for large filters ($L > 1024$), and as such is typically configured for large L, the RDA-FAP design does not store the already computed values to save area.

If $\mathbf{R}[n]$ is assumed to be Toeplitz [35], then Eqn. 4.20 is rewritten as,

$$\begin{bmatrix} r_0[n] & r_1[n] & r_2[n] & r_3[n] \\ r_1[n] & r_0[n] & r_1[n] & r_2[n] \\ r_2[n] & r_1[n] & r_0[n] & r_1[n] \\ r_3[n] & r_2[n] & r_1[n] & r_0[n] \end{bmatrix} \begin{bmatrix} p_0[n] \\ p_1[n] \\ p_2[n] \\ p_3[n] \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.32}$$

From Eqn. 4.32 the previous values of $\mathbf{r}[n]$ no longer need to be stored. The update of $\mathbf{r}[n]$ from Eqns. 4.28 to 4.30 is computed in place, where the updated $\mathbf{r}[n]$ overwrites the previous values $\mathbf{r}[n-1]$. The Gauss-Seidel iteration then becomes,

$$p_0[n] = \frac{1}{r_0[n]} \tag{4.33}$$

$$p_1[n] = \frac{1}{r_0[n]}\Big(- r_1[n]p_0[n]\Big) \tag{4.34}$$

$$p_2[n] = \frac{1}{r_0[n]}\Big(- r_2[n]p_0[n] - r_1[n]p_1[n]\Big) \tag{4.35}$$

$$p_3[n] = \frac{1}{r_0[n]}\Big(- r_3[n]p_0[n] - r_2[n]p_1[n] - r_1[n]p_2[n]\Big) \tag{4.36}$$

Since the previous values of $r_0[n]$ are no longer stored, there is no need to store the previously computed reciprocals either. The computational complexity of the Gauss-Seidel iteration remains the same, but the storage requirements decreased. The total number of additions and multiplications required to update $\mathbf{r}[n]$ for projection order p are,

$$\text{Number of additions:} \quad 2p \tag{4.37}$$

$$\text{Number of multiplications:} \quad 2p \tag{4.38}$$

4.2.4 Computing the Reciprocal $\frac{1}{r_0[n]}$

The value $\frac{1}{r_0[n]}$ can be computed using any standard division module. However, division is computationally expensive, requiring multiple clock cycles to compute the answer. Division algorithms can be categorized into slow and fast types.

Slow division algorithms such as the restoring and SRT algorithms [56] generate one bit of the quotient per iteration. This means that for a B-bit division, B iterations are needed

before the result is available. While the number of iterations may seem small, considering that a RDA-FIR filter can compute the output of a L-tap filter in approximately B cycles, the time required for division is no longer trivial. When applied to an adaptive filter in conjunction with RDA, the large iteration count hinders the throughput of the adaptive algorithm.

Fast division algorithms such as the Newton-Raphson division start with a close approximation of the final quotient first, then iterates towards the final solution, typically requiring only three to four cycles. A lookup table containing the estimates of the quotient is typically used to provide the starting approximated value. While fast division using the Newton-Raphson iteration requires a few cycles to complete, in order to maximize the throughput of the FAP algorithm, unneeded clock cycles should be removed.

The RDA-FAP design uses the lookup table concept of the Newton-Raphson division in computing $\frac{1}{r_0[n]}$. The reciprocal is computed by first normalizing $r_0[n]$ to the range of $1 \leq \hat{r}_0[n] < 2$, where $\hat{r}_0[n]$ is the normalized result. Then K bits from the fractional part of $\hat{r}_0[n]$ are used as the address into a lookup table that holds 2^K corresponding precomputed reciprocal values. The normalization operation is essentially finding, from MSB to LSB, the first 1 in the bits of $r_0[n]$. The location of the first 1 is noted, and K bits to the right of that 1 are used as the address into a lookup table. The quotient is read out from the table, and shifted by the amount and direction needed to normalize $r_0[n]$. Two examples of division via lookup table are presented to illustrate the workings of the division via lookup table method.

Consider the following 9-bit number $b = 0011.10110_2 = 3.6875_{10}$, and a 3-bit, 8 entries lookup table (Fig. 4.2). The number b is normalized by shifting one bit to the right so that $1 \leq \hat{b} < 2$, yielding $\hat{b} = 0001.11011_2 = 1.84238_{10}$. The lookup table uses 3-bit addressing, so the three bits to the right of the decimal point, 110_2, are used as the address into the table. From the table in Fig. 4.2 the address 110_2 corresponds to the value $0.10001_2 = 0.5313_{10}$. Since b was normalized by shifting one bit to the right, the value returned from the lookup

table is also shifted one bit to the right, obtaining $0.010001_2 = 0.2656_{10}$. The actual value of $1/b$ is 0.2712_{10}.

Consider another 9-bit number $g = 0000.01011_2 = 0.3438_{10}$ with the same 3-bit, 8 entries lookup table of Fig. 4.2. The number g is normalized by shifting two bits to the left, yielding $\hat{g} = 0001.01100_2 = 1.375_{10}$. The three bits to the right of the decimal point are 011_2, with that address corresponding to the value $0.10110_2 = 0.6875_{10}$. This time g was normalized by shifting two bits to the left, so the value returned from the lookup table is also shifted two bits to the left, yielding $10.110_2 = 2.75_{10}$, with the actual value of $1/g$ being 2.9087_{10}.

The two examples show that there is a small difference between the actual and approximated values. It will be shown in a later section that the discrepancy is not enough to upset the convergence behavior of the RDA-FAP algorithm. The two examples also show that the register holding the properly shifted value from the lookup table needs to be appropriately sized. If the number to be reciprocated is of the form,

$$\underbrace{XXX}_{K_I}.\underbrace{XXXXX}_{K_F} \tag{4.39}$$

where K_I and K_F are the number of integer and fractional bits respectively, and the contents of the lookup table are represented using K_L bits, the register holding the appropriately shifted output from the division lookup table needs to be sized as,

$$\underbrace{XXXXXX}_{K_F+1}.\underbrace{XXXXXX}_{K_I+K_L-2} \tag{4.40}$$

with a total width of $K_I + K_F + K_L - 1$. The RDA-FAP implementation uses a 4-bit address and $K_L = 7$ for the lookup table.

4.3 Optimizations to the FAP Algorithm

The complexity of the FAP algorithm was reduced in order to make the algorithm more hardware friendly. The FAP algorithm was optimized using similar methods as the optimizations carried out for the NLMS algorithm.

Memory Address	Contents (Binary)	Contents (Decimal)
000	1.00000	1
001	0.11100	0.875
010	0.11001	0.7813
011	0.10110	0.6875
100	0.10100	0.625
101	0.10011	0.5938
110	0.10001	0.5313
111	0.10000	0.5

Figure 4.2. Memory contents for the lookup table used by the division via LUT method.

The division via lookup table method searches from MSB to LSB for the first 1 in the bits of the number to be reciprocated, for the case of the FAP algorithm, the number is $r_0[n]$. Examining the equation for $r_0[n]$ with initial condition $r_0[-1] = 1$ and $x[n]$ normalized such that $-1 < x[n] < 1$,

$$r_0[n] = r_0[n-1] + x^2[n] - x^2[n-L] \tag{4.41}$$

the smallest value $r_0[n]$ can take is 1, because any subtraction due to $x^2[n-L]$ has an addition previously from $x^2[n]$. Using that relation, the largest possible value for $r_0[n]$ is identified as $L+1$. As such, the range of $r_0[n]$ is shown below.

$$1 \leq r_0[n] \leq L+1 \tag{4.42}$$

When this condition is applied to the division via LUT method, only the integer bits (K_I) need to be searched for the leading 1, instead of searching all the bits from MSB to LSB. In addition, since only the integer bits are considered, the register holding the shifted result from the lookup table is smaller, with a total width of $K_I + K_L - 1$.

The multiplication of the vector $\mathbf{p}[n]$ by the scalar $e[n]$ in Eqn. 4.15 can be implemented as a right shift operation on $\mathbf{p}[n]$ based on the magnitude of $e[n]$, as was done for the DA-NLMS implementation. The update of $\boldsymbol{\eta}[n]$ is then accomplished by adding the shifted version of $\mathbf{p}[n]$ to the previous values of $\boldsymbol{\eta}[n]$. Similarly, instead of computing, from Eqn. 4.14, $\mu\eta_{p-1}[n]\mathbf{x}[n-p+1]$, the correction term $\mu\eta_{p-1}[n]$ is quantized into an arithmetic

shift operation on $\mathbf{x}[n-p+1]$. The approximations made in the simplification of the FAP adaptive algorithm are used in the RDA-FAP design and are summarized below.

1. Simplification of the Gauss-Seidel iteration by assuming $\mathbf{R}[n]$ is Toeplitz
2. Converting the division operation from the Gauss-Seidel iteration into a table lookup
3. Quantization of $e[n]\mathbf{p}[n]$ into a shift right operation on $\mathbf{p}[n]$
4. Quantization of $\mu\eta_{p-1}[n]\mathbf{x}[n-p+1]$ into a shift right operation on $\mathbf{x}[n-p+1]$

Table 4.2 lists the number of math operations required by the FAP and RDA-FAP algorithms for one sample. As with the case for DA-NLMS, the table only lists the types of operations involved, and not the actual implementation of the algorithms. The FAP algorithm presented in Tbl. 4.2 uses one iteration of the Gauss-Seidel method. From the results the RDA-FAP algorithm uses less computations and requires no division. The processing steps and data dependencies of the FAP algorithm is shown in Fig. 4.3.

The fixed-point RDA-FAP was simulated in a system identification configuration (Fig. 4.4(a)) and compared against floating-point versions of FAP and NLMS. The input to the system is an autoregressive process represented by

$$x[n] = \omega[n] + \varphi x[n-1] \tag{4.43}$$

where $\omega[n]$ is white gaussian noise and $0 < |\varphi| < 1$. The results show that the quantization of the $e[n]\mathbf{p}[n]$ and $\eta_{p-1}[n]\mathbf{x}[n-p+1]$ operations into arithmetic shifts, along with the approximate reciprocation via LUT-based division, are valid optimizations (Fig. 4.5).

The effects of bit precision on the convergence behavior of NLMS and RDA-FAP are shown in the next figure (Fig. 4.6). For decreasing bit precision, the convergence rate degrades more for RDA-FAP than fixed-point NLMS. However, for the same precision, RDA-FAP still converges faster than NLMS for correlated inputs. The faster decline is caused by the various approximations used to simplify the computation of the RDA-FAP algorithm.

Table 4.2. Number of operations sorted by type for FAP and RDA-FAP

	FAP			RDA-FAP*		
Operation	ADD	MULT	DIV	ADD	MULT	DIV
Filtering	$L-1$	L	0	$L-1$	L	0
Compute Error	p	$p-1$	0	p	$p-1$	0
Update Calculation	$\frac{1}{2}p^2+\frac{3}{2}p+1$	$L+\frac{1}{2}p^2+\frac{7}{2}p-1$	1	$\frac{1}{2}p^2+\frac{3}{2}p+1$	$\frac{1}{2}p^2+\frac{5}{2}p-1$	0
Coefficient Update	L	0	0	L	0	0
Total	$2L+\frac{1}{2}p^2+\frac{5}{2}p$	$2L+\frac{1}{2}p^2+\frac{9}{2}p-2$	1	$2L+\frac{1}{2}p^2+\frac{5}{2}p$	$L+\frac{1}{2}p^2+\frac{7}{2}p-2$	0

* Types of operations, not actual implementation

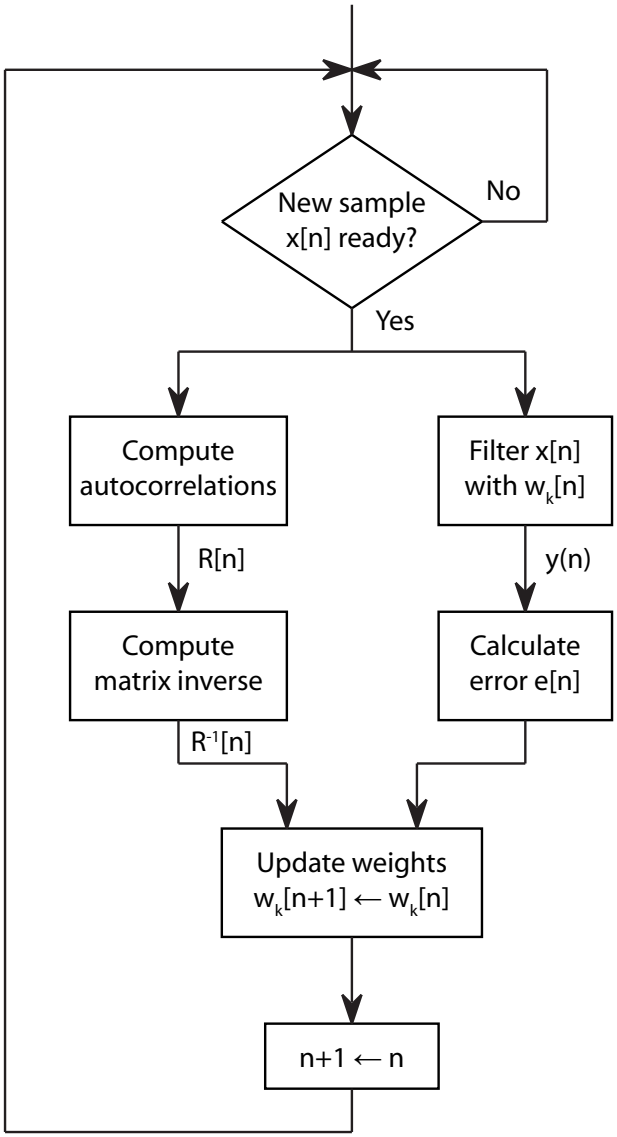
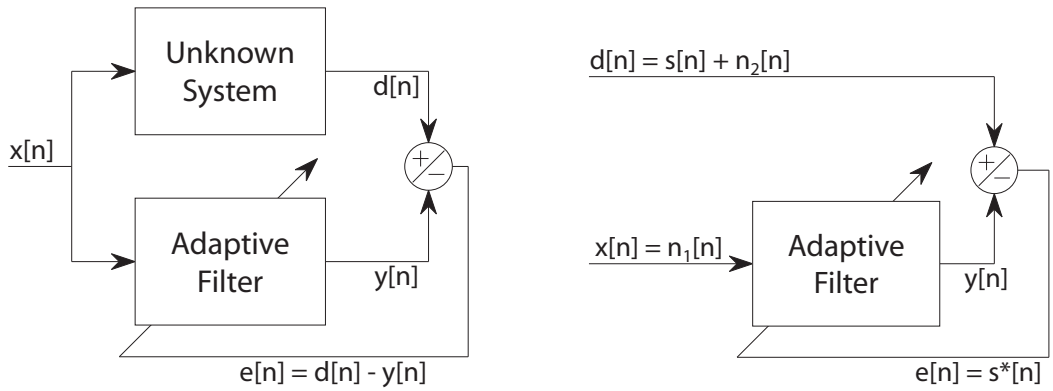


Figure 4.3. Flowchart of the FAP algorithm.



(a) System identification setup.

(b) Noise cancellation setup.

Figure 4.4. Test configurations used to evaluate the performance of the RDA-FAP adaptive filter.

The RDA-FAP adaptive algorithm was also simulated in a noise cancellation configuration (Fig. 4.4(b)). The noise cancellation testbench is formulated as follows,

$$n_1[n] = 0.8n_1[n-1] + \omega[n] \tag{4.44}$$

$$n_2[n] = -0.6n_2[n-1] + \omega[n] \tag{4.45}$$

$$x[n] = n_1[n] \tag{4.46}$$

$$d[n] = s[n] + n_2[n] \tag{4.47}$$

where $n_1[n]$ and $n_2[n]$ are autoregressive noise models, $\omega[n]$ is a zero-mean signal with normal distribution, $s[n]$ is the original speech signal. $x[n]$ and $d[n]$ are the input and desired signals, respectively. In this configuration the adaptive filter tries eliminate the noise from the speech signal. The error signal $e[n]$ is the recovered version of the input speech signal $s[n]$. Fig. 4.7 shows the convergence behavior of RDA-FAP compared to NLMS and FAP. The error shown is the difference between the actual speech signal and the recovered version. The results show that RDA-FAP converges faster than NLMS, at the same time converging slightly slower than the floating-point FAP algorithm. The FAP-based algorithms adapt very quickly to speech, resulting in the large jumps in the error curves. This occurs doing the voiced part of speech, which affects all three adaptive algorithms used. Because NLMS adapts slowly, the jumps are less pronounced. The jumps in the error can be removed by adding a voice activity detector to tell the adaptive filter to stop updating

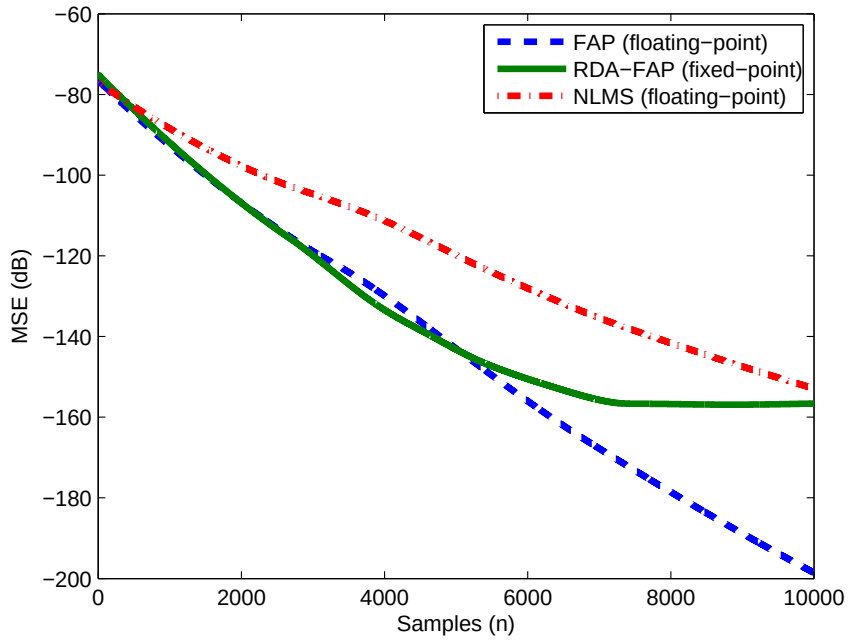


Figure 4.5. Convergence curves comparing the performance of RDA-FAP to NLMS and FAP. $L = 64$**,** $p = 4$**,** $\mu = 1$**, and** $\varphi = 0.95$**. The results are obtained by averaging 20 runs.**

during voiced segments.

4.4 FAP Adaptive Filtering with RDA

The RDA-FAP adaptive filter is composed of the RDA-FIR filter with supporting modules described in the previous sections for computing the update calculation. The partial product generator of the RDA-FAP adaptive filter is slightly different than the one used in the RDA-FIR filter. The block diagram of the modified partial product generator is shown in Fig. 4.8. The original RDA partial product generator is augmented with a 1-bit adder used for updating the coefficient.

The coefficients $\mathbf{w}[n]$ are updated by adding $\hat{\mathbf{w}}[n]$ to the previous coefficients $\mathbf{w}[n-1]$. Examining the update process for a single coefficient $w_k[n]$, the update equation is,

$$w_k[n] = w_k[n-1] + \hat{w}_k[n], \quad k = 0, 1, \ldots, L-1 \tag{4.48}$$

Typical implementations of the update equation would simply perform a parallel addition, *i.e.* all the bits used simultaneously to generate the sum. The RDA-FAP approach to Eqn.

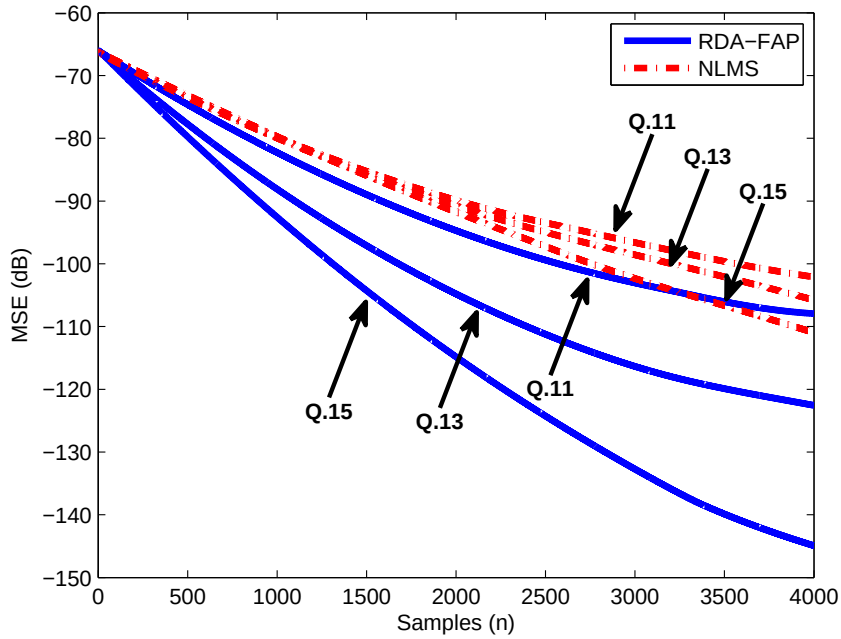


Figure 4.6. Convergence curves showing the effects of reducing bit precision on NLMS and RDA-FAP. $L = 64$**,** $p = 4$**,** $\mu = 1$**, and** $\varphi = 0.95$**. The results are obtained by averaging 20 runs.**

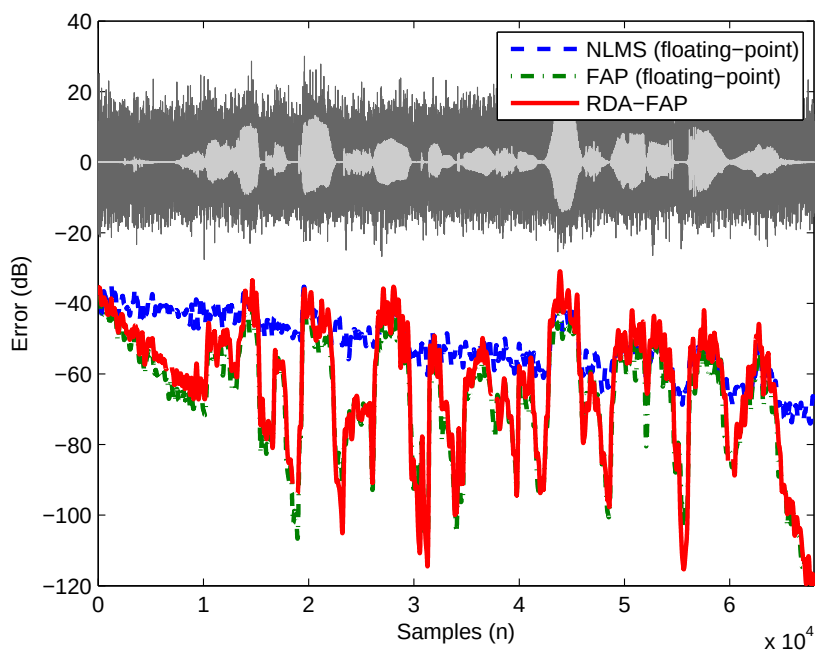


Figure 4.7. Noise cancellation results comparing RDA-FAP to NLMS and FAP.

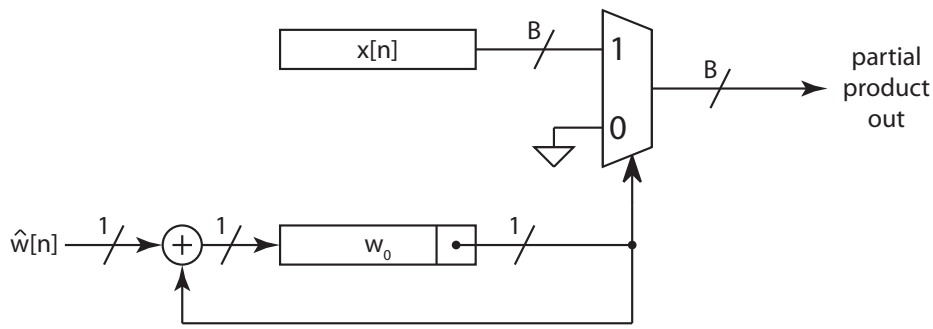


Figure 4.8. RDA-FAP partial product generation block.

4.48 is to perform the addition serially, where only a single bit of $w_k[n-1]$ and $\hat{w}_k[n]$ are used a a time, generating the sum result one bit at a time. This is only possible because of the unique structure of the RDA partial product generation block, where the coefficients are used to serially drive the system. The serial nature of the addition enables the coefficient update step to complete in B cycles, regardless of the length of the RDA-FAP filter.

The number of cycles needed to update the coefficients for typical implementations of any adaptive algorithm is dependent on the length of the filter. The RDA design enables the filtering phase of the adaptive filter and coefficients update phase to complete in constant time, independent of filter length. This ability is what enables the RDA-FAP adaptive filter to achieve very high throughput for long filter lengths.

4.5 FAP Adaptive Filtering with MM

The MM-FIR filter is used to implement the FAP algorithm with the same supporting modules as the RDA-FAP design. In this manner the comparison between RDA-FAP and MM-FAP involves only the core filtering and coefficient update operations. With the FAP algorithm, if L is very large, then for the MM design the computation of the coefficient correction term $\hat{\mathbf{w}}[n]$ would be completed before the filtering operation is finished. As such, the processing time required for computing the weight correction term can be neglected for the throughput and latency equations.

The block diagram of the new MM multiplier and memories block is shown in Fig. 4.9.

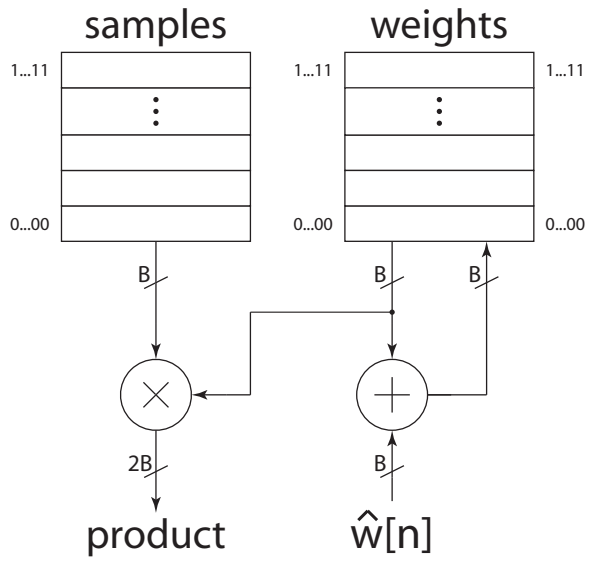


Figure 4.9. MM-FAP multiplier and memories block.

The original structure from the MM-FIR design was modified to incorporate a *B*-bit adder used in the update of the coefficient memory. The coefficient memory has been changed from a single-port memory to a dual-port memory. A dual-port memory is capable of reading from, and writing to, the memory at the same time but to different locations.

CHAPTER 5

COMPARISON OF THE RDA AND MM FAP ADAPTIVE FILTERS

FAP adaptive filters of different lengths were implemented using the RDA and MM design presented in the previous chapter. The two designs are compared in terms of throughput, latency, and area. Throughput is defined as the number of samples processed per unit time. Latency is defined as the amount of time required to generate an output sample once the input sample has arrived. Area is defined as the amount of resources needed to implement the design. The RDA-FAP and MM-FAP are designed to use the same adaptation block for this comparison *i.e.,* the supporting modules computing the coefficient correction term are identical. The reasoning behind using the same adaptation block is to focus the comparison on the filtering and update capabilities of both designs. In addition, for the filter sizes considered in this chapter, the update calculations are completed before the filtering operation is complete.

Neglecting the processing time for the computation of the weight correction term, the throughputs of the RDA-FAP and MM-FAP adaptive filters are given by,

$$\text{Throughput}_{\text{RDA}} = \left[\underbrace{B + \log_2(L)}_{Filtering} + \underbrace{B}_{Updating} \right]^{-1} \tag{5.1}$$

$$\text{Throughput}_{\text{MM}} = \left[\underbrace{\frac{L}{M} + \log_2(M)}_{Filtering} + \underbrace{\frac{L}{M}}_{Updating} \right]^{-1} \tag{5.2}$$

and the latencies by,

$$\text{Latency}_{\text{RDA}} = B + \log_2(L) \tag{5.3}$$

$$\text{Latency}_{\text{MM}} = \frac{L}{M} + \log_2(M) \tag{5.4}$$

where B is the bit precision, L is the filter length, and M is the number of multipliers. The equations for throughput show that the RDA-FAP and MM-FAP filters require more

cycles than their FIR counterparts while the latencies remain the same. This is because for adaptive filters the coefficients have to be updated before a new input sample can be processed. The latency of the adaptive filter is determined by how quickly the output can be computed, which in turn is governed by the latency of the FIR filter. The throughput and latency of the RDA-FAP and MM-FAP adaptive filters are shown in Fig. 5.1(a) and Fig. 5.1(b). The number of multipliers was selected as $M = 8$ and $M = 32$, and the bit precision was varied between $B = 16$ and $B = 32$. The plots show the relationship between B, M, and L on throughput of both designs.

5.1 Gate-level Comparison of RDA-FAP and MM-FAP

The gate-level comparison of RDA-FAP and MM-FAP uses the gate delay and gate count equations established in Chapter 3. The critical path arithmetic component for the RDA-FAP design is still the adder at the output of the summation block, and the slowest component for the MM-FAP design is the multiplier. The ideal equations (Eqns. 5.1 to 5.4) are modified to include the gate delay information. The resulting throughput and latency equations are shown below.

RDA-FAP:

$$\text{Throughput}_{\text{RDA}} = [T_{\text{CLA}}(B + \log_2(L)) \times [2B + \log_2(L)]]^{-1} \tag{5.5}$$

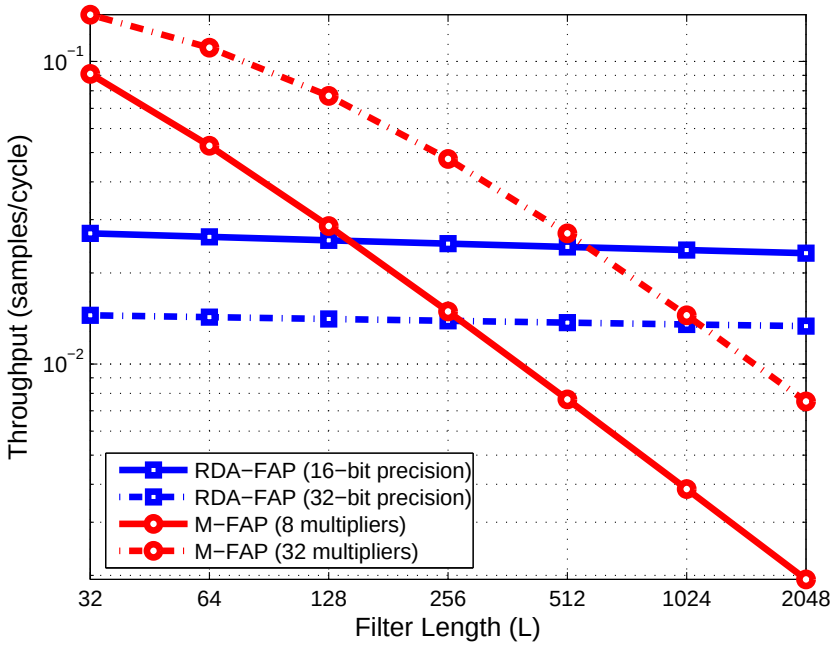
$$\text{Latency}_{\text{RDA}} = T_{\text{CLA}}(B + \log_2(L)) \times [B + \log_2(L)] \tag{5.6}$$

MM-FIR:

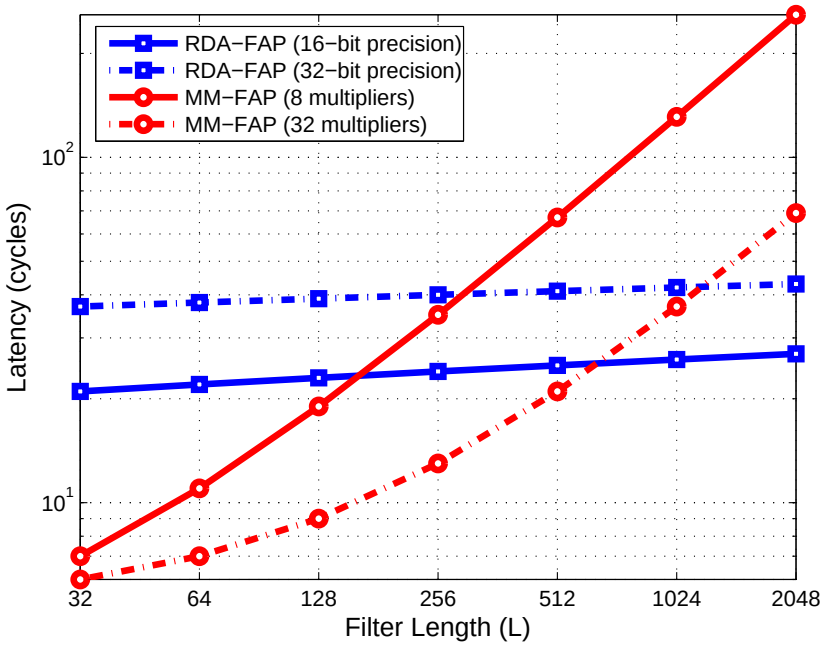
$$\text{Throughput}_{\text{MM}} = \left[T_{\text{Dadda}}(B) \times \left[\frac{2L}{M} + \log_2(M)\right]\right]^{-1} \tag{5.7}$$

$$\text{Latency}_{\text{MM}} = T_{\text{Dadda}}(B) \times \left[\frac{L}{M} + \log_2(M)\right] \tag{5.8}$$

The RDA-FAP and MM-FAP designs are then configured to have the same throughput. Identical throughput is achieved by setting the throughputs equal and solving for the



(a)



(b)

Figure 5.1. Ideal throughput and latency plots for RDA-FAP and MM-FAP.

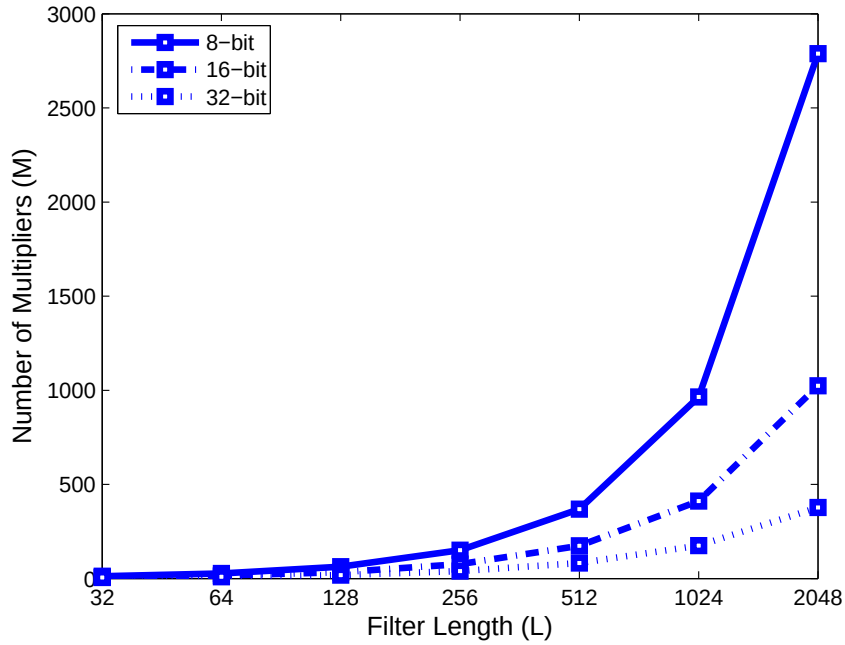


Figure 5.2. Number of multipliers needed for MM-FAP.

number of multipliers, M. The resulting equation for M is,

$$M = 2L\left(\frac{T_{\text{CLA}}(B + \log_2(L))}{T_{\text{Dadda}}(B)}\left[2B + \log_2(L)\right] - \log_2(M)\right)^{-1} \tag{5.9}$$

Equation 5.9 is solved iteratively since the variable M appears on both sides of the equation. The values of M are computed for varying filter lengths (L) and bit precision (B) and shown in Fig. 5.2. For large filter lengths, the number of multipliers needed by the MM-FAP design to maintain the same throughput as RDA-FAP design is too large to be practical. For example, for $L = 2048$ and $B = 16$, the number of multipliers needed is approximately 1000. If the filter length is reduced to $L = 512$ while maintaining the same bit precision, the number of multipliers needed is roughly 180, which is still a large number.

Using the number of multipliers obtained from solving Eqn. 5.9, the latency of the MM-FAP is shown in Fig. 5.3(b) along with the RDA-FAP latency. The results show that the MM-FAP design has higher latency than the RDA-FAP for all filter lengths and bit precisions except for one case, where $B = 32$ and $L = 32$. The results from this comparison match those obtained from the FIR case with the exception of the single outlier.

The gate counts of the two designs are computed in the same way as for the RDA-FIR and MM-FIR case. The number of arithmetic components used in both designs are added together, and the using the equations established in Section 3.1 of Chapter 3, the total gate counts are computed. The gate count includes the extra components needed to perform the coefficient update step (Eqn. 4.48). The RDA-FAP design has an additional 1-bit adder for every coefficient, and the MM-FAP design has a B-bit carry-lookahead adder for every multiplier. Not included in the gate count are the hardware blocks that regulate the filtering operation and the supporting modules that compute the coefficient correction term. The supporting modules are not included since the modules are the same for both RDA-FAP and MM-FAP. The results of Fig. 5.3(a) show that the MM-FAP design requires more gates than the RDA-FAP design. Again, the results are unsurprising considering the results from the FIR filter comparison. The MM-FAP design is on average ~16 times larger than the RDA-FAP design.

The multiplier of the MM-FAP adaptive filter is then pipelined to improve the throughput performance of the overall system. Pipelining should yield throughput gains similar to the MM-FIR filter. The new throughput and latency equations for the pipelined MM-FAP (PMM) adaptive filter are listed below.

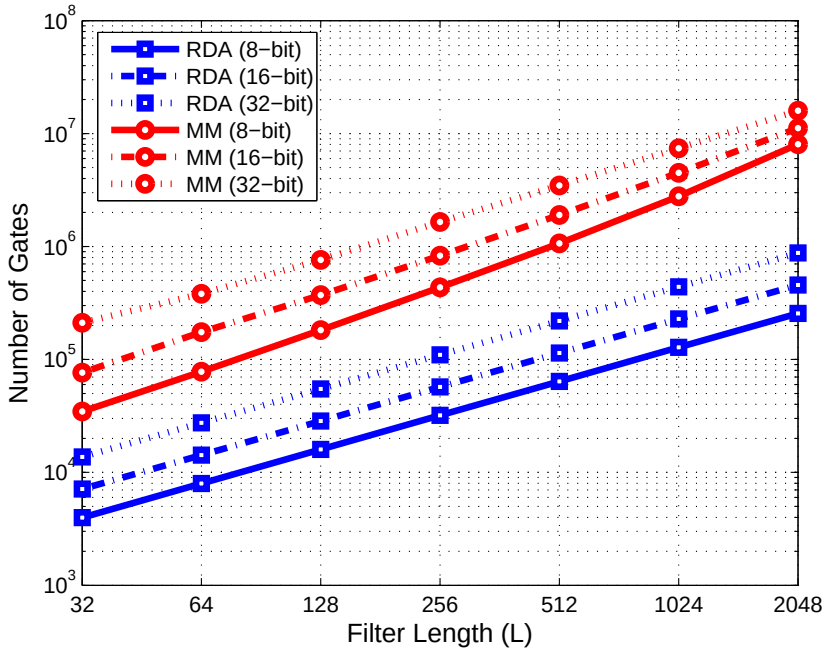
$$\text{Throughput}_{\text{PMM}} = \left[T_{\text{CLA}}(2B + \log_2(M))\left[\frac{2L}{M} + 2\log_{1.5}(M) + \log_2(M)\right]\right]^{-1} \tag{5.10}$$

$$\text{Latency}_{\text{PMM}} = T_{\text{CLA}}(2B + \log_2(M))\left[\frac{L}{M} + \log_{1.5}(M) + \log_2(M)\right] \tag{5.11}$$

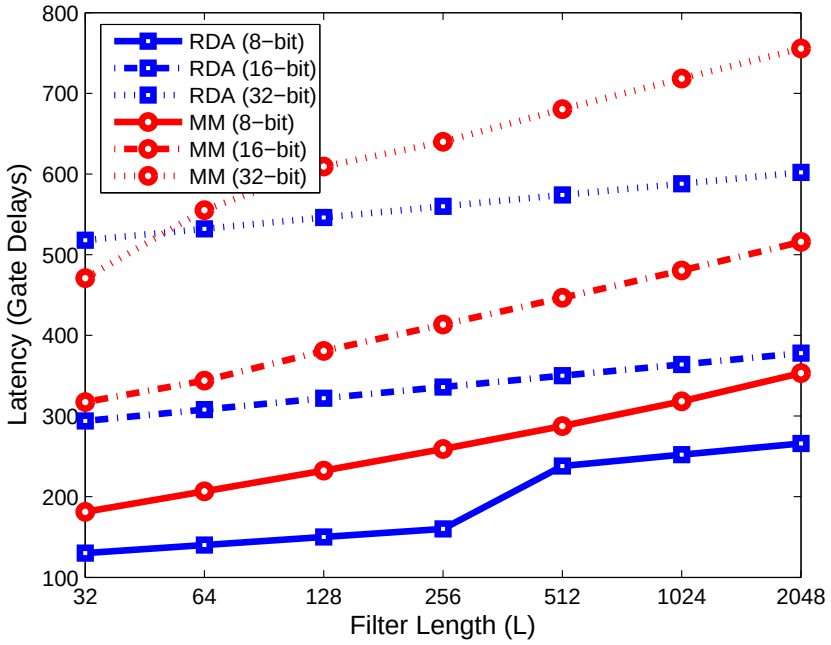
The throughput of the PMM-FAP adaptive filter is now dependent on the gate delay of the carry-lookahead adder instead of the multiplier, since the multiplier has been pipelined and split into smaller, more basic arithmetic logic blocks. The throughput includes the contribution from the pipelined stages of the Dadda compression tree.

Using the throughput equations of PMM-FAP and RDA-FAP, the number of multipliers needed to ensure identical throughput is computed, resulting in the following equation.

$$M = 2L\left(\frac{T_{\text{CLA}}(B + \log_2(L))}{T_{\text{CLA}}(2B + \log_2(M))}[2B + \log_2(L)] - 2\log_{1.5}(B) - \log_2(M)\right)^{-1} \tag{5.12}$$



(a)



(b)

Figure 5.3. Gate count and latency for MM-FAP.

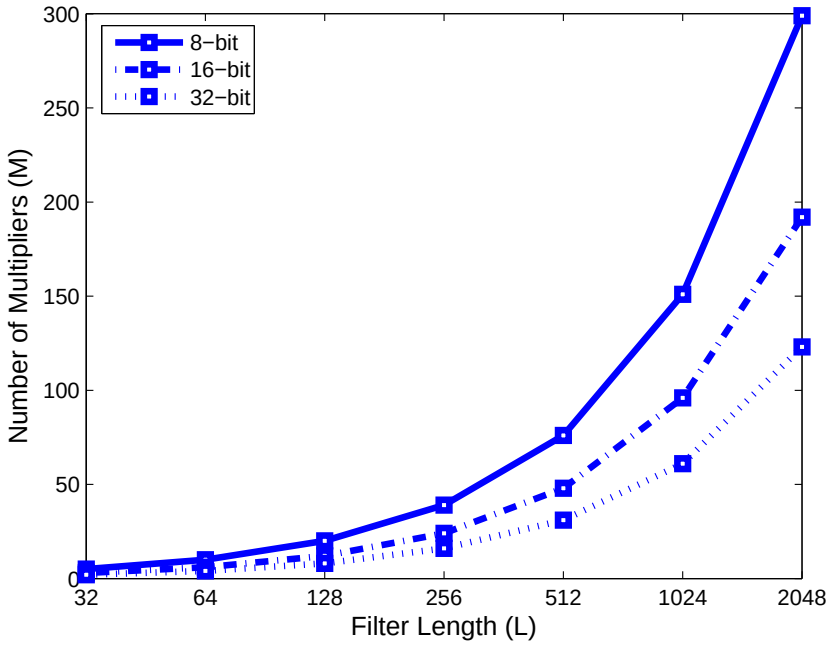
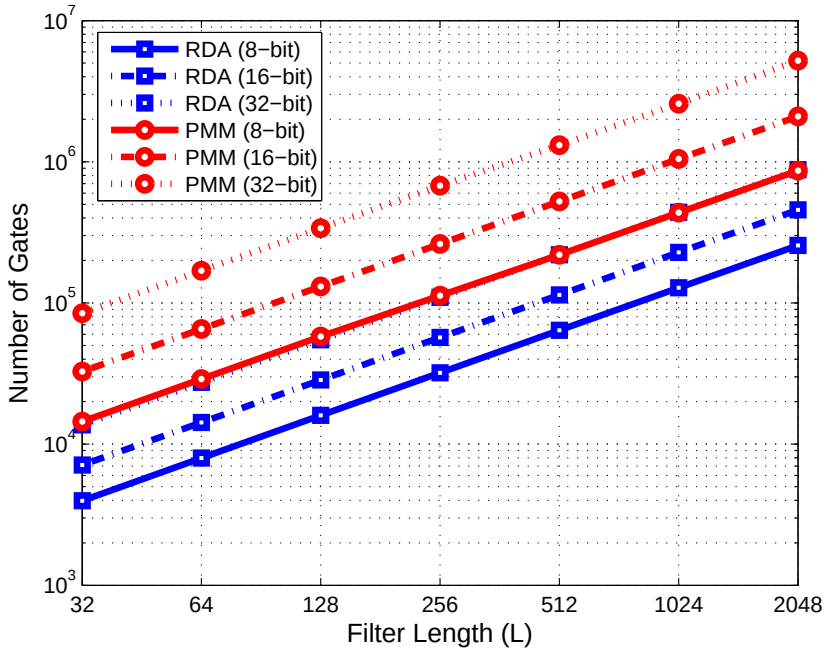


Figure 5.4. Number of multipliers needed for PMM-FAP.

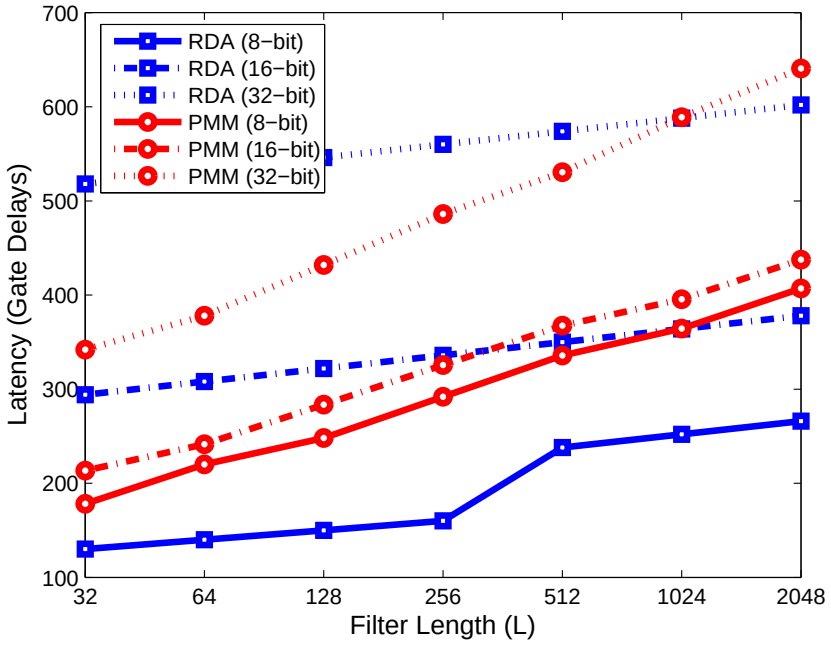
Equation 5.12 is solved by iteration, and the resulting number of multipliers are shown in Fig. 5.4. The number of multipliers needed by the PMM-FAP design to match the throughput of the RDA-FAP design is a lot less than that needed by the MM-FAP design. For example, for a $L = 2048$ length filter at 16-bit precision, the number of multipliers is approximately 200, instead of 1000 for the MM-FAP case.

The gate count of the pipelined MM-FAP adaptive filter is shown in Fig. 5.5(a). Since the PMM-FAP design requires fewer multipliers, the gate count of the PMM-FAP design is lower than the gate count of the MM-FAP design. The PMM-FAP design is still larger than the RDA-FAP design however. On average the PMM-FAP adaptive filter is ~5 times larger than the RDA-FAP design.

The latency results of Fig. 5.5(b) show that RDA-FAP no longer maintains a lower value of latency than PMM-FAP for all values of B and L. For moderate bit precision ($B > 8$), RDA-FAP has lower latency than the PMM-FAP when the filter lengths are large ($L > 512$). When the bit precision is low, ($B = 8$), then RDA-FAP is consistently lower latency than the



(a)



(b)

Figure 5.5. Gate count and latency for PMM-FAP.

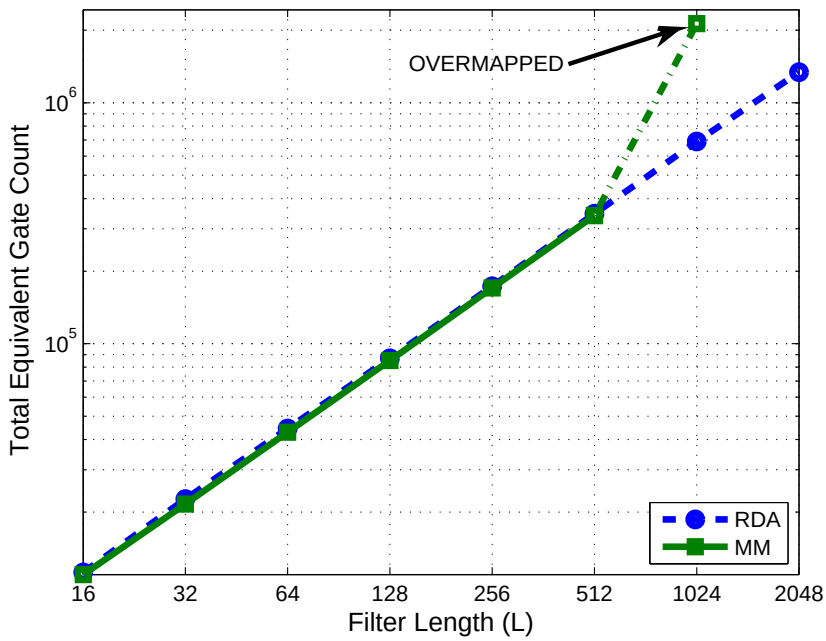


Figure 5.6. RDA-FIR and MM-FIR FPGA synthesis results.

PMM-FAP design. As stated earlier, the latency equation for the adaptive filter is actually the equation for the FIR filter case. The discrepancy between the RDA-FAP to PMM-FAP and RDA-FIR to PMM-FIR is attributed to the calculated value of M. In order for the PMM-FAP adaptive filter to match the throughput of the RDA-FAP adaptive filter, the computed number of multipliers is higher than the values computed for PMM-FIR versus RDA-FIR. This increase in multiplier count is what causes the latency of the PMM-FAP to be lower than that of RDA-FAP. The ratio of the number of multipliers for PMM-FAP ($M_{\text{PMM-FAP}}$) to the number of multipliers for PMM-FIR ($M_{\text{PMM-FIR}}$) is on average,

$$\frac{M_{\text{PMM-FAP}}}{M_{\text{PMM-FIR}}} = 1.33 \tag{5.13}$$

which corresponds to a 33% increase in the number of multipliers.

The FPGA results obtained for the RDA-FIR and MM-FIR comparisons (Ch. 3, Fig. 3.10(b)) are used to estimate the requirements of the RDA-FAP and MM-FAP adaptive filters. The total equivalent gate count is presented again here (Fig. 5.6) for convenience. The total equivalent gate count results show that the largest FIR filter based on multipliers

with the same throughput as the RDA-FIR filter that can be placed on the Xilinx Virtex-4 LX100 FPGA is 512 taps. This means that the 512-tap MM-FIR filter uses all 96 of the available multipliers on the FPGA, with none available to compute the FAP update calculations during the filtering phase. The RDA-FIR design on the other hand, has all 96 multipliers available. In this way the update calculations for the RDA-FAP adaptive filter can be performed while the filtering phase is being processed.

The result of the MM-FAP filter needing to process the filtering and update calculations separately is the increase in the number of clock cycles required between output samples. With the MM-FAP filter now requiring more clock cycles, in order to maintain the same throughput as RDA-FAP even more multipliers are required. Mapping this relation back to the FPGA means that the length of the largest MM-FAP adaptive filter capable of fitting onto the FPGA becomes noticeably smaller.

5.2 Improving the Throughput of RDA-FAP

The use of pipelining on the Dadda multipliers greatly improved the throughput and latency of the MM-FAP design. The pipelining concept essentially breaks long computation process into a series of shorter processes that can then be executed simultaneously. The adders of the RDA-FAP designed can also be pipelined [57], but considering that the sizes of the adders are not very large, any gains from pipelining would be small. Instead, the throughput of the RDA-FAP adaptive filter can be improved by using a different number format, binary signed-digit (BSD), for the internal calculations and exploiting the properties of the BSD format.

5.2.1 Binary Signed-Digit Numbers

The conventional binary number system represents integers as binary numbers of fixed length. The binary number of length n is an ordered sequence of binary digits,

$$(b_{n-1}, b_{n-2}, \ldots, b_1, b_0) \tag{5.14}$$

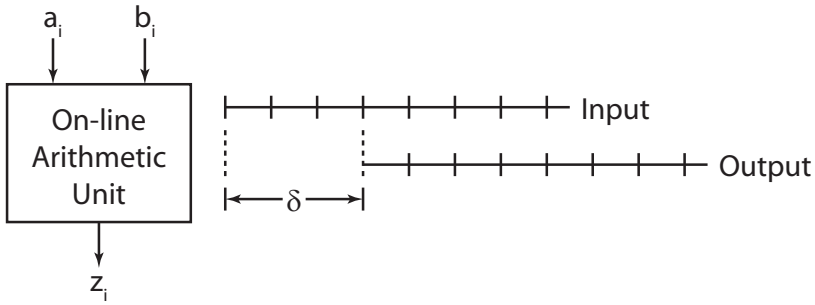


Figure 5.7. Input/Output characteristics of on-line arithmetic. a_i**,** b_i**, and** z_i **are the** i**-th digit of** a**,** b**, and** z **respectively.**

where each digit, or bit, can either be 0 or 1. Each digit in the binary number belongs to the set,

$$b_i \in \{0, 1\} \tag{5.15}$$

whereas the digits of binary signed-digit numbers belong to the set,

$$b_i \in \{\bar{1}, 0, 1\} \tag{5.16}$$

where $\bar{1}$ denotes the value -1. Binary signed-digit numbers have the property of carry-free addition, that is the addition of two BSD numbers does not generate a carry that propagates from LSB to MSB. The lack of a rippling carry enables the use of on-line (MSB-first) arithmetic [18][58][59].

When two binary numbers are added together, the sum is is computed from the least significant bit up to the most significant bit, or in a right to left fashion. Using MSB-first or on-line arithmetic, the sum is computed in the other direction, from left to right. The same right to left principle applies to other arithmetic operations such as multiplication and division. In this manner, on-line arithmetic makes it possible to overlap all arithmetic operations. On-line arithmetic modules however have an intrinsic on-line delay, δ, which is the number of digits that must be processed by the module before the first digit of the output is generated (Fig. 5.7). The value of δ is typically a small integer, *i.e.*, $0 < \delta < 4$.

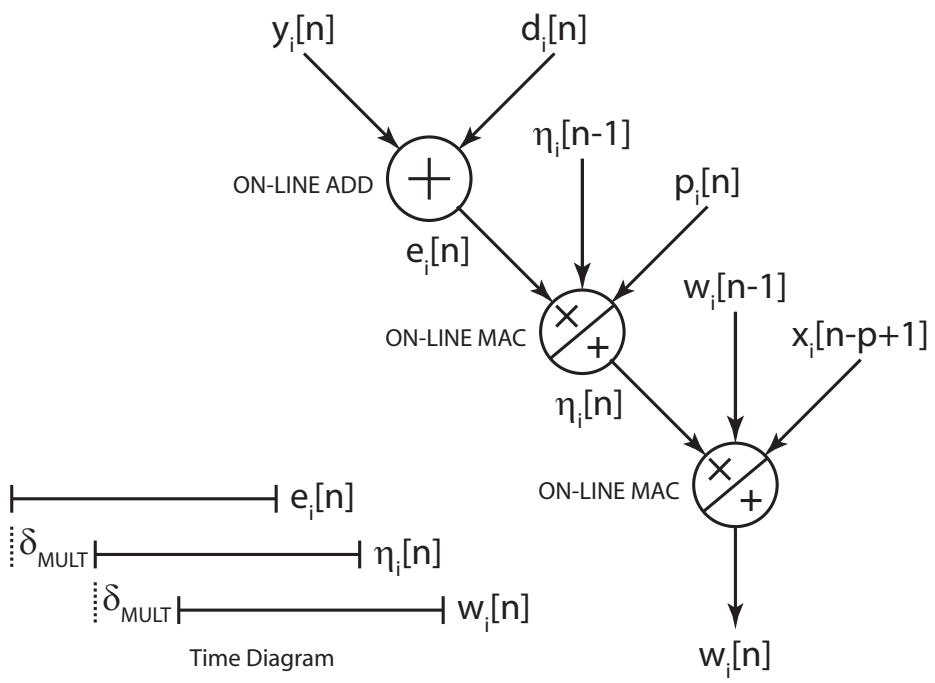


Figure 5.8. NRDA-FAP coefficient update using on-line arithmetic.

5.2.2 BSD and RDA-FAP

Modifications to the RDA-FAP adaptive filter were performed to incorporate the use of on-line arithmetic. The new RDA-FAP (NRDA-FAP) is architecturally the same as the original RDA-FAP design, except the operation of NRDA-FAP is slightly different. In addition, NRDA-FAP computes the FAP algorithm in its entirety, instead of quantizing certain multiplication operations into shift operations. In doing so it is expected for the NRDA-FAP to be slower, since more computations are being performed, however through the use of BSD and on-line arithmetic, the operations are overlapped resulting in a minor increase in clock cycles. The filtering phase of NRDA-FAP is the same as RDA-FAP with the minor difference that the coefficients driving the system are now fed from MSB to LSB instead of LSB to MSB. The corresponding change in the accumulation block is to change the direction of accumulation from shifting right to shifting left. In doing so the most significant bits of the output $y[n]$ are computed first, allowing for the remainder of the coefficient update calculations to begin. The computed bits of the accumulator are stable, since there are no carries generated when adding new values to the accumulator because of

the BSD format. The summation of the partial products in the adder tree is still computed in the same manner, the only difference is the partial products are represented in BSD form.

The calculations stemming from $y[n]$ are computed using on-line arithmetic. The bits of $y[n]$ are available serially, and used to compute $e[n]$ with the first bit result available after the delay associated with on-line addition (δ_{ADD}). The amount of time needed for the bits to settle in the accumulator is also δ_{ADD}. The computation of $\boldsymbol{\eta}[n]$ generates the serial output stream after δ_{MAC}, and the filter coefficients are then updated in a similar manner using an on-line multiply-accumulate module with associated delay of δ_{MAC}. The sequence of on-line operations is shown in Fig. 5.8.

The area of the NRDA-FAP design is larger than RDA-FAP because of the physical representation of the binary signed-digit number set. Since the binary number set includes -1, 0, and 1, two bits per digit are needed for storage instead of one bit. The input and coefficient storage for NRDA-FAP remains the same as RDA-FAP, but the intermediate registers used in the summation and accumulation block are now twice as large. The adders in the summation block are also of the BSD type, requiring slightly more gates than the CLA equivalent presented earlier. The schematics of the different adders needed for NRDA-FAP are presented by Kuninobu *et al.* [60].

The ideal throughput and latency equations for NRDA-FAP are shown below.

$$\text{Throughput}_{\text{NRDA}} = \left[B + \log_2(L) + 2\delta_{\text{ADD}} + 2\delta_{\text{MAC}}\right]^{-1} \tag{5.17}$$

$$\text{Latency}_{\text{NRDA}} = B + \log_2(L) \tag{5.18}$$

The latency equation remains unchanged from previously, but the throughput equation has been modified by replacing one of the B terms with $2\delta_{\text{ADD}} + 2\delta_{\text{MAC}}$. Figure 5.9 shows the throughput of the NRDA-FAP adaptive filter. The values for δ_{ADD} and δ_{MAC} are 2 and 4, respectively. The results from Fig. 5.9 show that the NRDA-FAP design requires less cycles than RDA-FAP except for the low precision case, $B = 8$.

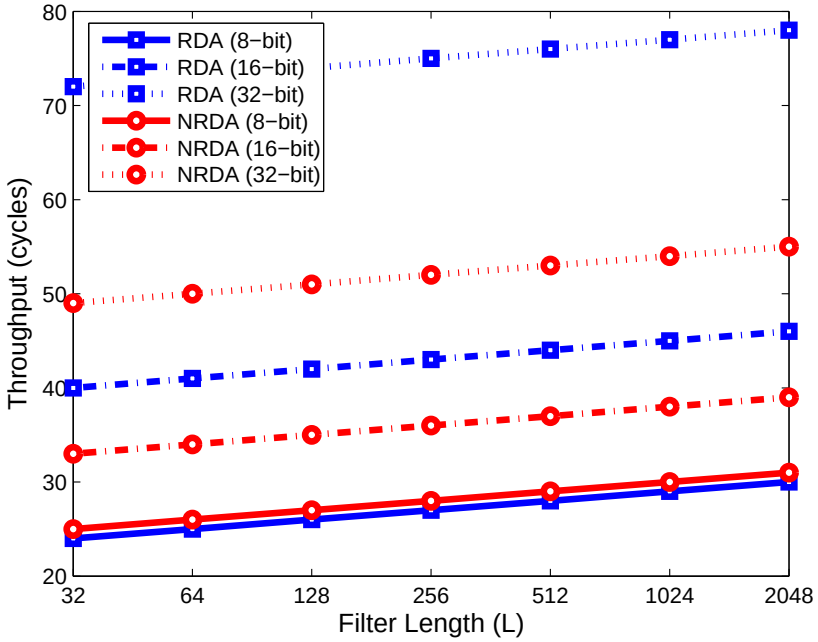


Figure 5.9. Throughput comparison of RDA-FAP and NRDA-FAP.

CHAPTER 6

CASE STUDY: DIGITAL HEARING AIDS

Hearing aids are used primarily to treat sensorineural hearing loss, which occurs when the cochlea and possibly auditory nerve are affected. Sensorineural loss changes the way loudness is perceived: soft sounds become inaudible, while loud or intense sounds are heard as loudly as persons who have normal hearing. The range of loudness for a hearing impaired person has been skewed, *i.e.*, the minimum loudness level has increased but the maximum loudness has remained the same. The approach used by hearing aids is to amplify the soft acoustic signals while reducing the loud signals. In addition to sound amplification, digital hearing aids also perform these essential functions: acoustic directionality, noise reduction, feedback cancellation, and sound classification.

Sound amplification can be performed using linear amplification or wide dynamic range compression (WDRC). Linear amplification is essentially a volume control, requiring the hearing aid wearer to adjust the volume as needed. WDRC on the other hand, provides different degrees of amplification based on the loudness of the sound, essentially functioning as an automatic volume control.

Acoustic directionality reduces noise originating from behind or to the side of the hearing aid wearer. This directional noise suppression involves either using directional microphones or omnidirectional microphones to generate a null pattern in the rear direction of the wearer.

Noise reduction in digital hearing aids operate by analyzing and processing the acoustic signal in many bands. Noise reduction is used for suppressing noise that maintains a relative constant sound pressure level over time, such as the noise in a car or train.

Hearing aid whistling occurs when amplified sounds feeds back from the receiver to the microphone. This can occur due to various factors such as poor fitting, changing ear size, small physical size, and other attributes of the hearing aid. Feedback cancellation removes

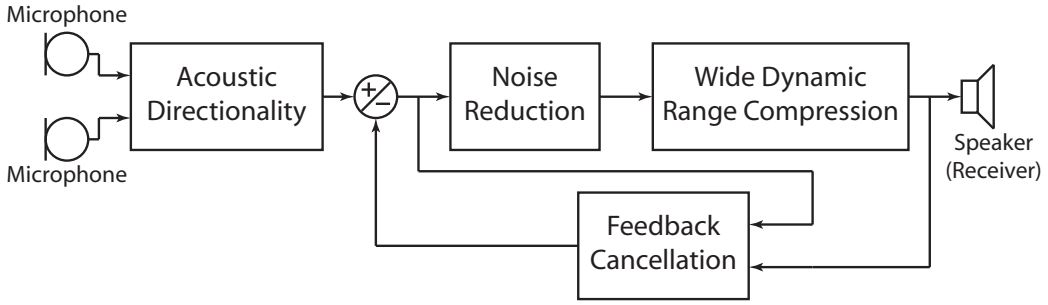


Figure 6.1. Simplified block diagram of a digital hearing aid.

the whistling effect using an adaptive filter that continuously monitors acoustic inputs of the hearing aid.

Sound classification is where the hearing aid identifies different acoustic environments and loads a different configuration based on the environment. For example, for speech inputs, the hearing aid would maximize intelligibility, while for music the hearing aid would try to have a broadband response.

The number of functions implemented vary, and is usually associated with the cost of the hearing aid. A "basic" hearing aid might only implement sound amplification, while a "mid-range" hearing aid would implement noise cancellation in addition to the "basic" features, and a "high-end" hearing aid would implement all the functions. A simplified block diagram of a digital hearing aid illustrating the different functions is shown in Fig. 6.1.

The various types and designs of hearing aids can be categorized into three basic styles. The three styles differ by size, placement, and hearing loss severity, as shown in Fig. 6.2. Each style of hearing aids has its advantages and disadvantages [61]. In general, the smaller the hearing aid, the smaller battery, and thusly the shorter the battery life.

Behind-the-ear (BTE) hearing aids are the largest type, composed of the hearing aid electronics in a plastic housing placed behind the ear with the processing acoustics signals sent into the ear via tubing and custom earmold, or electrically with a speaker placed in the ear canal. BTE hearing aids are typically used for mild to profound hearing loss.

In-the-ear (ITE) and in-the-canal (ITC) hearing aids are smaller than BTE hearing aids,

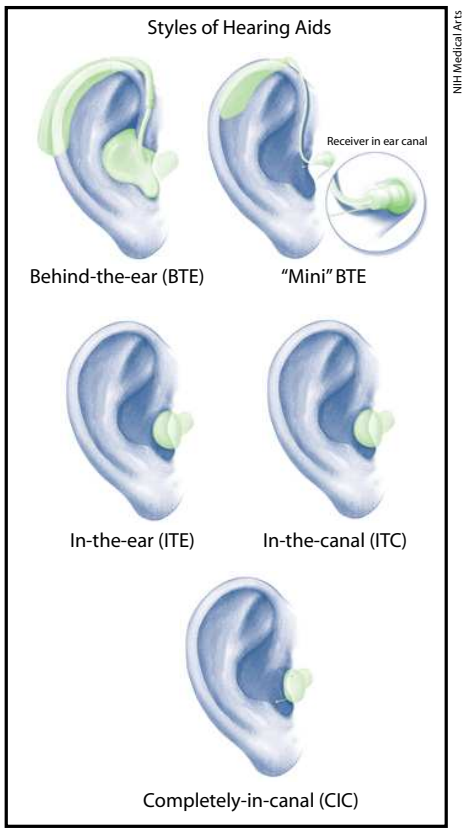


Figure 6.2. Different types of hearing aids.

fitting inside the outer ear. ITE hearing aids are used for mild to severe hearing loss. All the electronics are encased in a plastic housing that fits in the ear canal and extends to the outer ear, with no external wires or tubes.

Completely-in-canal (CIC) hearing aids are the smallest type of hearing aids. All the components are housed in a custom molded case that fits inside the ear canal. CIC hearing aids take advantage of the ear's natural sound collecting design, however, they are restricted to persons with ear canals large enough to accommodate the hearing aid. CIC hearing aids are used for mild to moderately severe hearing loss.

In this chapter the concept of RDA-FIR and RDA-FAP is applied to digital hearing aids. Example digital hearing aid designs are presented, along with a comparison between the RDA approach and the traditional multiplier (MM) approach. The results show the benefits of using RDA and the positive impact on battery life.

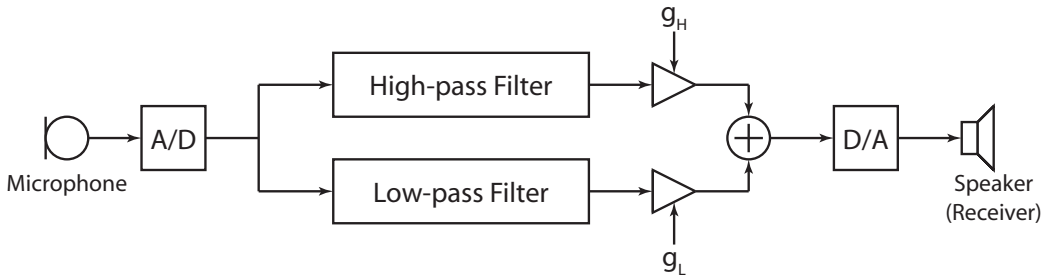


Figure 6.3. Block diagram of a digital hearing aid using WDRC.

6.1 Wide Dynamic Range Compression

Wide dynamic range compression is a method for compensating for the abnormal loudness growth that affects people with hearing loss by amplifying soft sounds by more and loud sounds by less. The block diagram of a digital hearing aid using a two-channel WDRC system is shown in Fig. 6.3. Typical implementations divide the high-pass and low-pass filters into further sub-filters, using 20 filters with approximately 100 coefficients each to cover the range from 200 Hz to 10 kHz [62].

The RDA-FIR and MM-FIR designs are compared to each other in the framework of WDRC. The operating specifications for a filter used in WDRC are shown in Tbl. 6.1. The RDA-FIR and MM-FIR filters are configured using the specifications listed. The delay of a 16-bit Dadda multiplier implemented in 0.18 μm process is shown to be 1.9 ns [50]. The delay of a 16-bit carry lookahead adder is extrapolated from the actual Dadda multiplier delay and the gate delay relation from the equations presented in Section 3.1. The gate delays for a 16-bit Dadda multiplier and 16-bit carry lookahead adder are computed as 51 and 10 respectively. Using this ratio the gate delay for the 16-bit CLA in 0.18 μm process is estimated to be 0.372 ns.

The gate count equations from Section 3.1 are used to compute the number of multipliers so that the MM-FIR design has approximately the same gate count as the RDA-FIR design. For a 16-bit 128-tap FIR filter, the number of multipliers is computed to be $M = 2$. This normalizing step is performed so that the comparison between the two designs focuses on speed and power.

Table 6.1. Specifications for a WDRC filter.

Specification	Variable	Value
Sampling frequency	f_s	20 kHz
Latency	T_l	< 1 ms
Filter length	L	128
Precision	B	16

The WDRC specifications list a sampling frequency of 20 kHz, which translates to 50 µs between successive output samples. Substituting the previous information into the throughput equation for RDA-FIR to find the operating system period, T_{RDA},

$$\text{Throughput}_{\text{RDA}} = B \times T_{\text{RDA}} \tag{6.1}$$

$$50 \times 10^{-6} = 16 \times T_{\text{RDA}} \tag{6.2}$$

$$T_{\text{RDA}} = 3.125 \times 10^{-6} \tag{6.3}$$

$$f_{\text{RDA}} = 1/T_{\text{RDA}} = 320 \text{ kHz} \tag{6.4}$$

and for the MM-FIR,

$$\text{Throughput}_{\text{MM}} = \frac{L}{M} \times T_{\text{MM}} \tag{6.5}$$

$$50 \times 10^{-6} = 64 \times T_{\text{MM}} \tag{6.6}$$

$$T_{\text{MM}} = 0.7813 \times 10^{-6} \tag{6.7}$$

$$f_{\text{MM}} = 1/T_{\text{MM}} = 1.28 \text{ MHz} \tag{6.8}$$

The sampling frequency of WDRC is relatively low, so the adders in the summation block of both designs no longer need to be registered, that is, the summation block completes all the summations in a single cycle dictated by T_{RDA} and T_{MM}. The amount of time needed by the summation block of both designs to complete, $T_{\text{SUM-RDA}}$ and $T_{\text{SUM-MM}}$, is computed and compared against T_{RDA} and T_{MM} to ensure that $T_{\text{SUM-RDA}} < T_{\text{RDA}}$ and $T_{\text{SUM-MM}} < T_{\text{MM}}$. Computing $T_{\text{SUM-RDA}}$,

$$T_{\text{SUM-RDA}} = \log_2(L) \times T_{\text{CLA}}(B) \tag{6.9}$$

$$= 7 \times 0.372 \times 10^{-9} \tag{6.10}$$

$$= 2.6 \text{ ns} \tag{6.11}$$

and $T_{\text{SUM-MM}}$,

$$T_{\text{SUM-MM}} = \log_2\left(\frac{L}{M}\right) \times T_{\text{CLA}}(2B) \tag{6.12}$$

$$= 4 \times 0.573 \times 10^{-9} \tag{6.13}$$

$$= 2.29 \text{ ns} \tag{6.14}$$

The results show that the summation block for both designs completes the additions within the time frame specified by T_{MM} and T_{RDA} by a large margin. The latencies are then computed using the following equations,

$$\text{Latency}_{\text{RDA}} = B \times T_{\text{RDA}} + T_{\text{SUM-RDA}} \tag{6.15}$$

$$= 16 \times 3.125 \times 10^{-6} + 2.6 \times 10^{-9} \tag{6.16}$$

$$\approx 50 \text{ µs} \tag{6.17}$$

$$\text{Latency}_{\text{MM}} = \frac{L}{M} \times T_{\text{MM}} + T_{\text{SUM-MM}} \tag{6.18}$$

$$= 64 \times 0.7813 \times 10^{-6} + 2.29 \times 10^{-9} \tag{6.19}$$

$$\approx 50 \text{ µs} \tag{6.20}$$

which show that both the RDA-FIR and MM-FIR designs meet the latency requirements of WDRC.

The power consumption of the RDA-FIR and MM-FIR implementations of the WDRC filters are estimated and compared. While the two designs are configured to have approximately the same gate count, the operating system frequencies are different. From Eqns. 6.4 and 6.8, the RDA-FIR design requires a system clock of 320 kHz while the MM-FIR design requires a system clock of 1.28 MHz. The system clock of the RDA-FIR design is four times slower than the MM-FIR system clock. Focusing on the low-pass and high-pass filters of Fig. 6.3, the difference in clock frequencies results in the RDA-FIR design consuming one quarter of the power of the MM-FIR design, assuming the identical capacitive

loads and switching probability considering the two designs have similar gate complexity. The actual difference is calculated by including the power consumption of the rest of the WDRC hearing aid. Using Eqn. 1.11, the power consumption of the WDRC system can be represented as,

$$P = P_F + P_S \tag{6.21}$$

$$= C_F V_{DD}^2 f_F + P_S \tag{6.22}$$

where P_F is the power consumed by the low and high-pass filters and P_S is the power used by the remainder of the WDRC system. C_F and f_F are the capacitance and frequency of the filtering block, respectively. Rewriting the previous equations in terms of current I,

$$I = C_F V_{DD} f_F + I_S \tag{6.23}$$

where I_S is the current used by the remainder of the WDRC system. Rewriting Eqn. 6.23 for the MM-FIR and RDA-FIR cases,

$$I_{\text{MM}} = C_F V_{DD} f_F + I_S \tag{6.24}$$

$$I_{\text{RDA}} = \frac{1}{4} C_F V_{DD} f_F + I_S \tag{6.25}$$

This set of relations is applied to an example hearing aid.

The Digi-Ear D1 [63] is a basic in-the-ear (ITE) digital hearing aid that implements WDRC. The battery of the Digi-Ear D1 is a Type 312 with a capacity of 110 mAh, meaning that it can supply 1 mA for 110 hours or 2 mA for 55 hours and so forth. The datasheet of the Digi-Ear D1 lists the total current draw to be 0.85 mA. Given the simplicity of the WDRC system, it can be approximated that the current required is split equally between the filters and the rest of the system. Using Eqn. 6.24 and the specifications of the Digi-Ear D1, C_F is computed to be 2.767×10^{-10} F. Substituting the value of C_F into Eqn. 6.25, I_{RDA} is calculated to be 0.53 mA.

The current consumption values are translated into battery life in terms of number of days. Instead of using 24 hours per day, hearing aid companies typically quote battery life

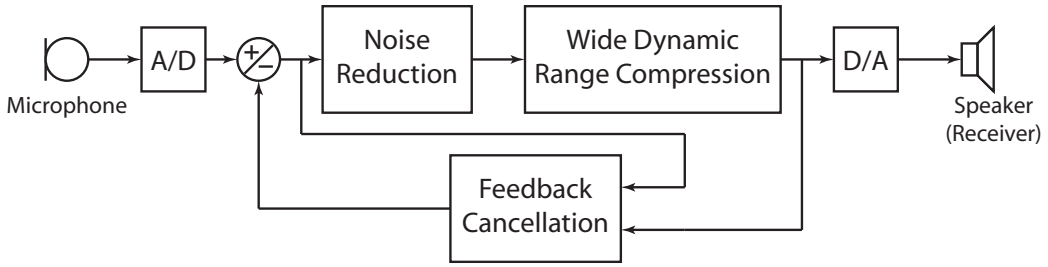


Figure 6.4. Block diagram of a digital hearing aid with noise reduction, WDRC, and feedback cancellation.

values in terms of 16 hour days, since it is assumed that the hearing aid is turned off when the user is asleep. The battery life of the RDA-FIR WDRC system is 13 days as opposed to the 8 days of the MM-FIR WDRC system. The results show that for a comparison using first order approximations, the WDRC system based on RDA-FIR lasts 5 days longer than the typical MM-FIR approach.

6.2 Feedback Cancellation

The digital hearing aid used in this example consists of modules performing noise reduction, wide dynamic range compression, and feedback cancellation. The structures used for noise suppression and WDRC are very similar, therefore the module used for noise reduction can be combined with the WDRC module [62]. The adaptive filter used in feedback cancellation monitors and cancels feedback signals with up to 3 ms delay, and typically uses the NLMS algorithm. With a sampling frequency of 20 kHz, the 3 ms delay equates to a 60-tap adaptive filter. For this comparison the number of taps used is 64. The block diagram of the digital hearing aid used in this example is shown in Fig. 6.4. The noise reduction block is the same as the WDRC block presented in the previous section. The specifications for the feedback cancellation block is summarized in Tbl. 6.2.

Using the same analysis procedure as the WDRC comparison, the RDA-FAP and MM-FAP designs are configured to have the same gate count. The gate count equations from Section 3.1 result in the MM-FAP requiring 1 multiplier. Continuing the analysis, the

Table 6.2. Specifications for feedback cancellation.

Specification	Variable	Value
Sampling frequency	f_s	20 kHz
Latency	T_l	< 1 ms
Filter length	L	64
Precision	B	16

throughput of the feedback cancellation module using RDA-FAP is,

$$\text{Throughput}_{\text{RDA}} = T_{\text{RDA}} \times [2B + \log_2(L)] \tag{6.26}$$

$$50 \times 10^{-6} = T_{\text{RDA}} \times [32 + 6] \tag{6.27}$$

$$T_{\text{RDA}} = 1.32 \times 10^{-6} \tag{6.28}$$

$$f_{\text{RDA}} = 1/T_{\text{RDA}} = 758 \text{ kHz} \tag{6.29}$$

and for the MM-FAP design,

$$\text{Throughput}_{\text{MM}} = T_{\text{MM}} \times \left[\frac{2L}{M} + \log_2(M)\right] \tag{6.30}$$

$$50 \times 10^{-6} = T_{\text{MM}} \times [128 + 0] \tag{6.31}$$

$$T_{\text{MM}} = 3.91 \times 10^{-7} \tag{6.32}$$

$$f_{\text{MM}} = 1/T_{\text{MM}} = 2.56 \text{ MHz} \tag{6.33}$$

The clock frequencies (f_{RDA} and f_{MM}) needed by the RDA and MM approaches for feedback cancellation are approximately twice that needed by the WDRC system. A quick check of the RDA-FAP and MM-FAP feedback cancellation modules show that both approaches meet the latency specified in Tbl. 6.2. The latency of the RDA-FAP and MM-FAP are based on the same equations as RDA-FIR and MM-FIR, and with the higher clock frequencies needed, both designs are guaranteed to meet the latency minimum of 1 ms. The RDA-FAP clock frequency is approximately 3.4 times slower than the MM-FAP clock frequency.

The approximate power consumption of the feedback cancellation module is computed

in the same manner as for the WDRC example. The dynamic power consumption of only the feedback cancellation (FC) module is represented by,

$$P_{FC} = C_{FC} V_{DD}^2 f_{FC} \tag{6.34}$$

where C_{FC} and f_{FC} are the capacitance and frequency of the feedback cancellation block, respectively. Rewriting the previous equation in terms of current I,

$$I_{FC} = C_{FC} V_{DD}^2 f_{FC} \tag{6.35}$$

and the total current consumption of the digital hearing aid with noise suppression, WDRC, and feedback cancellation is given by,

$$I = \underbrace{C_{FC} V_{DD} f_{FC}}_{\text{feedback cancellation}} + \underbrace{C_F V_{DD} f_F}_{\text{WDRC + noise reduction}} + \underbrace{I_S}_{\text{everything else}} \tag{6.36}$$

Rewriting Eqn. 6.36 for the MM and RDA cases,

$$I_{\text{MM}} = C_{FC} V_{DD} f_{FC} + C_F V_{DD} f_F + I_S \tag{6.37}$$

$$I_{\text{RDA}} = \frac{1}{3.4} C_{FC} V_{DD} f_{FC} + \frac{1}{4} C_F V_{DD} f_F + I_S \tag{6.38}$$

Equations 6.37 and 6.38 are applied to the Digi-Ear DS digital hearing aid [63].

The Digi-Ear DS is the high-end digital hearing aid of the Digi-Ear product line, incorporating the functions represented by Eqn. 6.36. The Digi-Ear DS is also an ITE hearing aid that uses Type 312 batteries with a 110 mAh capacity. The datasheet for the Digi-Ear DS lists the current draw as 1.1 mA. Using the values previously computed from the WDRC example, the feedback cancellation block using multipliers consumes 0.25 mA. Using Eqns. 6.37 and 6.38 along with the previously computed values, the feedback cancellation block using RDA consumes 0.074 mA. Combining the current consumption of the blocks for the RDA implementation, the current draw of the digital hearing aid is computed to be 0.6 mA.

The current consumption values are then translated into battery life using the 16 hour per day metric commonly used by hearing aid manufacturers. The battery life of the RDA-based digital hearing aid is approximately 11 days, while the battery life of the MM-based

hearing aid is approximately 6 days. The use of RDA enables the digital hearing aid to last twice as long as the traditional multiplier-based design.

The use of reusable distributed arithmetic increases the battery life of digital hearing aids by a non-trivial amount. For the case of a simple digital hearing aid that only implements wide dynamic range compression, RDA increases battery life by 1.6 times. Applying RDA to a full-featured digital hearing aid almost doubles the battery life when compared to the typical multiplier approach.

CHAPTER 7

CONCLUSION AND FUTURE RESEARCH

New architectures for FIR filtering and the fast affine projection adaptive filter were proposed, implemented using FPGAs, and compared against typical architectures. The new architectures were also evaluated for digital hearing aid applications. In this chapter, the work accomplished is summarized along with future research directions.

7.1 Summary of Contributions

The contributions of this thesis are summarized in this section.

7.1.1 Reusable Distributed Arithmetic

In Chapter 2, a new architecture for FIR filtering based on distributed arithmetic was developed. The proposed RDA architecture requires less area than the typical DA and multiplier-based approaches. It was then shown in Chapter 3 that when configured for the same throughput, the RDA-FIR filter was 10 times smaller than the MM-FIR filter and 4 times smaller than the PMM-FIR filter. In both cases the RDA-FIR filter has a lower latency. FPGA synthesis results from Chapter 3 show that the largest filter capable of being synthesized for the RDA case was 2048 taps, while for the MM case was 512 taps. For the FPGA designs both RDA-FIR and MM-FIR were configured to have the same throughput. The RDA architecture forms the basis for the high-throughput low-latency fast affine projection adaptive filter.

7.1.2 DA NLMS Adaptive Filter

In Chapter 4 a fixed-point NLMS adaptive filter was implemented using a division free normalization method based on arithmetic left and right shifts of the data, reducing the complexity of the NLMS algorithm by the removal of a division operation. The DA-NLMS adaptive filter was developed as an extension to the DA-LMS adaptive filter previously

designed by other members of the research group. Simulation results show that the normalization method used degrade slightly the performance of the NLMS algorithm when compared to the full implementation version. However, the DA-NLMS algorithm still converges faster than the LMS algorithm. The normalization method introduced for the DA-NLMS adaptive filter is carried on into the design of the FAP adaptive filter.

7.1.3 FAP Adaptive Filter

A high-throughput, low-latency adaptive filter was introduced in Chapter 4. The adaptive filter implements the fast affine projection algorithm, an adaptive algorithm that is typically implemented with long filter lengths. Reusable distributed arithmetic forms the core of the RDA-FAP adaptive filter. The division free normalization method from the DA-NLMS adaptive filter were used in conjunction with additional optimizations to reduce the complexity of the FAP algorithm. The RDA-FAP adaptive filter again requires less area than the MM-FAP adaptive filter when configured for the same throughput, being approximately 16 times smaller. The RDA-FAP adaptive filter also has lower latency than the MM-FAP adaptive filter. The RDA-FAP design is 5 times smaller than the pipelined version of the MM-FAP adaptive filter, however depending on the filter length and bit precision, the PMM-FAP might have lower latency. In general the results show that for large filter sizes RDA-FAP requires the least amount of area and has a lower latency than both MM-FAP and PMM-FAP.

7.1.4 Implications for Digital Hearing Aids

In Chapter 6 the RDA-FIR and RDA-FAP filters were applied to digital hearing aids. Digital hearing aid functions consist primarily of filtering and adaptive filtering. Since the sampling rates for digital hearing aids are relatively low, the comparison between RDA-based designs and MM-based designs was performed by normalizing based on size. The RDA-based designs and MM-based designs were first configured to have the same area, and based on that configuration the system clock frequency was determined. The results show

that RDA-based designs only need to be clocked at a fraction of the MM-based designs, resulting in RDA-based designs having significantly longer battery life than MM-based designs, roughly two times the battery life.

7.2 Future Research Directions

Directions for future research are presented in this section.

7.2.1 Application of RDA to the FFT/IFFT

The FFT is a widely used transform and can be found in applications ranging from audio processing to radar. FFT architectures based on distributed arithmetic have been proposed, but the smaller size of RDA in relation to other DA approaches should yield a smaller FFT design. Certain applications such as radar require very large FFT sizes, at least 4096 points, and also require high throughput. High throughput independent of filter length is a strength of RDA, as such the implementation of a small and efficient FFT/IFFT module could be very beneficial for demanding applications. For example, having a fast and low power FFT/IFFT processing core could enable real-time synthetic aperture radar (SAR) streams from unmanned aerial vehicles (UAVs). Current implementations of synthetic aperture radar using UAVs involve SAR data being processed on the ground, *i.e.* the SAR data is captured and recorded, then processed when the UAV returns from its flight.

7.2.2 RDA-NLMS

RDA has been shown to be an efficient way of performing filtering and adaptive filtering. In order to compare RDA to other DA approaches to adaptive filtering, implementation of a NLMS adaptive filter using RDA is needed. In this manner the comparison between RDA-NLMS and DA-NLMS in terms of speed, area, and power could provide more insight into improving the RDA design.

7.2.3 Identifying New Applications

The high-throughput of RDA can potentially enable new applications that were previously limited by speed, area, and power constraints. The identification of such applications would involve interactions between many different research groups. New applications can also provide additional design challenges that were previously unknown. The combination of new applications and new challenges can yield interesting and promising results.

REFERENCES

[1] A. Croisier, D. Esteban, M. Levilion, and V. Rizo, "Digital filter for PCM encoded signals US Patent 3,777,130," 1973.

[2] S. Zohar, "New Hardware Realizations of Nonrecursive Digital Filters," *IEEE Transactions on Computers*, vol. C-22, no. 4, pp. 328–338, 1973.

[3] A. Peled and B. Liu, "A New Hardware Realization of Digital Filters," *IEEE Transactions on ASSP*, vol. 22, no. 6, pp. 456–462, 1974.

[4] F. Taylor, "An analysis of the distributed arithmetic digital filter," *IEEE Transactions on ASSP*, vol. 34, no. 5, pp. 1165–1170, 1986.

[5] M. Martinez-Peiro, J. Valls, T. Sansaloni, A. Pascual, and E. Boemo, "A comparison between lattice, cascade and direct form FIR filter structures by using a FPGA bit-serial distributed arithmetic implementation," in *Proc. 6th IEEE International Conference on Electronics, Circuits and Systems ICECS '99*, vol. 1, pp. 241–244 vol.1, 1999.

[6] Y.-H. Chan and W.-C. Siu, "On the realization of discrete cosine transform using the distributed arithmetic," *IEEE Transactions on Circuits and Systems I*, vol. 39, no. 9, pp. 705–712, 1992.

[7] H.-C. Chen, J.-I. Guo, T.-S. Chang, and C.-W. Jen, "A memory-efficient realization of cyclic convolution and its application to discrete cosine transform," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 3, pp. 445–453, 2005.

[8] H.-C. Chen, J.-I. Guo, C.-W. Jen, and T.-S. Chang, "Distributed arithmetic realisation of cyclic convolution and its DFT application," *IEE Proceedings -Circuits, Devices and Systems*, pp. 615–629, 2005.

[9] S. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE ASSP Magazine*, vol. 6, no. 3, pp. 4–19, 1989.

[10] J.-P. Choi, S.-C. Shin, and J.-G. Chung, "Efficient ROM Size Reduction for Distributed Arithmetic," in *Proc. ISCAS 2000 Geneva Circuits and Systems The 2000 IEEE International Symposium on*, vol. 2, pp. 61–64 vol.2, 2000.

[11] H. Yoo and D. V. Anderson, "Hardware-efficient Distributed Arithmetic Architecture for High-order Digital Filters," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05)*, vol. 5, pp. v/125–v/128 Vol. 5, 2005.

[12] S. Chandrasekaran and A. Amira, "Novel Sparse OBC based Distributed Arithmetic Architecture for Matrix Transforms," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2007*, pp. 3207–3210, 2007.

[13] A. Shams, A. Chidanandan, W. Pan, and M. Bayoumi, "NEDA: a low-power high-performance DCT architecture," *IEEE Transactions on Signal Processing*, vol. 54, no. 3, pp. 955–964, 2006.

[14] C. Burrus, "Digital Filter Structures described by Distributed Arithmetic," *IEEE Transactions on Circuits and Systems*, vol. 24, no. 12, pp. 674–680, 1977.

[15] R. Babic, M. Solar, and B. Stiglic, "High order FIR digital filter realization in distributed arithmetic," in *Proc. th Mediterranean Electrotechnical Conference*, pp. 367–370 vol.1, 1991.

[16] A. Garcia, U. Meyer-Base, A. Lloris, and F. Taylor, "RNS implementation of FIR filters based on distributed arithmetic using field-programmable logic," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS '99*, vol. 1, pp. 486–489 vol.1, 1999.

[17] C.-L. Su, Y.-T. Hwang, and C.-W. Jen, "A novel recursive digital filter based on signed digit distributed arithmetic," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS '97*, vol. 3, pp. 2104–2107 vol.3, 1997.

[18] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. 10, pp. 389–400, 1961.

[19] R. Amirtharajah and A. Chandrakasan, "A micropower programmable DSP using approximate signal processing based on distributed arithmetic," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 2, pp. 337–347, 2004.

[20] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*. Addison Wesley, 2000.

[21] J. Rabaey, A. A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2002.

[22] J. Sacha and M. Irwin, "Input recoding for reducing power in distributed arithmetic," in *Proc. IEEE Workshop on Signal Processing Systems SIPS 98*, pp. 599–608, 1998.

[23] S. Ramprasad, N. Shanbhag, and I. Hajj, "Low-power distributed arithmetic architectures using nonuniform memory partitioning," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS '99*, vol. 3, pp. 470–473 vol.3, 1999.

[24] N. Hajj, C. Polyckronopoulos, and G. Stamoulist, "Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors," in *Proc. International Symposium on Low Power Electronics and Design*, pp. 70–75, 1998.

[25] N. Tan, S. Eriksson, and L. Wanhammar, "A power-saving technique for bit-serial DSP ASICs," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS '94*, vol. 4, pp. 51–54 vol.4, 1994.

[26] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*. Prentice Hall, 1985.

[27] S. Haykin, *Adaptive Filter Theory*. Prentice Hall, 1996.

[28] M. H. Hayes, *Statistical Digital Signal Processing and Modeling*. Wiley, 1996.

[29] B. Farhang-Boroujeny, *Adaptive Filters: Theory and Applications*. John Wiley and Sons, 1998.

[30] K. Ozeki and T. Umeda, "An adaptive filtering algorithm using an orthogonal projection to an affine subspace and its properties," in *Electron. Comm. Japan*, vol. 67-A, pp. 19–27, 1984.

[31] S. Gay, "A fast converging, low complexity adaptive filtering algorithm," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics Final Program and Paper Summaries*, pp. 4–7, 17–20 Oct. 1993.

[32] S. Gay and S. Tavathia, "The fast affine projection algorithm," in *Proc. International Conference on Acoustics, Speech, and Signal Processing ICASSP-95*, vol. 5, pp. 3023–3026, 9–12 May 1995.

[33] S. Douglas, "Efficient approximate implementations of the fast affine projection algorithm using orthogonal transforms," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP-96*, vol. 3, pp. 1656–1659, 7–10 May 1996.

[34] H. Ding, "A stable fast affine projection adaptation algorithm suitable for low-cost processors," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP '00*, vol. 1, pp. 360–363, 5–9 June 2000.

[35] S. Oh, D. Linebarger, B. Priest, and B. Raghothaman, "A fast affine projection algorithm for an acoustic echo canceller using a fixed-point DSP processor," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP-97*, vol. 5, pp. 4121–4124, 21–24 April 1997.

[36] N. Levinson, "The Wiener RMS error criterion in filter design and prediction," *J. Math. Phys.*, vol. 25, pp. 261–278, 1947.

[37] J. Durbin, "The fitting of time series models," *Rev. Inst. Int. Stat.*, vol. 28, pp. 233–243, 1959.

[38] W. F. Trench, "An algorithm for the inversion of finite Toeplitz matrices," *J. Soc. Indust. Appl. Math.*, vol. 12, pp. 515–522, 1964.

[39] F. Albu, J. Kadlec, N. Coleman, and A. Fagan, "The Gauss-Seidel fast affine projection algorithm," in *Proc. IEEE Workshop on Signal Processing Systems (SIPS '02)*, pp. 109–114, 16–18 Oct. 2002.

[40] R.Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, "Templates for the solutions of linear systems: Building blocks for iterative methods," *SIAM*, 1994.

[41] E. Chau, H. Sheikhzadeh, and R. Brennan, "Complexity reduction and regularization of a fast affine projection algorithm for oversampled subband adaptive filters," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '04)*, vol. 5, pp. V–109–12, 17–21 May 2004.

[42] F. Albu and C. Kotropoulos, "Modified Gauss-Seidel affine projection algorithm for acoustic echo cancellation," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05)*, vol. 3, pp. iii/121–iii/124, 18–23 March 2005.

[43] M. Tanaka, S. Makino, and J. Kojima, "A block exact fast affine projection algorithm," *IEEE Transactions on Speech and Audio Processing*, vol. 7, no. 1, pp. 79–86, 1999.

[44] Q. Liu, B. Champagne, and K. Ho, "On the use of a modified fast affine projection algorithm in subbands for acoustic echo cancelation," in *Proc. IEEE Digital Signal Processing Workshop*, pp. 354–357, 1996.

[45] H.-J. Lo, H. Yoo, and D. V. Anderson, "A reusable distributed arithmetic architecture for FIR filtering," in *Proc. 51st Midwest Symposium on Circuits and Systems MWSCAS 2008*, pp. 233–236, 2008.

[46] A. Weinberger and J. L. Smith, "A logic for high-speed addition," in *National Bureau of Standards Circular*, vol. 591, pp. 3–12, 1958.

[47] O. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, pp. 67–91, Jan. 1961.

[48] J. Swartzlander, E.E. and G. Goto, *Digital Design and Fabrication*. CRC Press, 2008.

[49] L. Dadda, "Some schemes for parallel multipliers," in *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[50] K. Bickerstaff, J. Swartzlander, E.E., and M. Schulte, "Analysis of column compression multipliers," in *Proc. 15th IEEE Symposium on Computer Arithmetic*, pp. 33–39, 2001.

[51] J. Jump and S. Ahuja, "Effective pipelining of digital systems," *IEEE Transactions on Computers*, vol. C-27, no. 9, pp. 855–865, 1978.

[52] H.-J. Lo and D. V. Anderson, "A hardware-efficient implementation of the fast affine projection algorithm," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, submitted.

[53] Huang, W. and Krishnan, V. and Allred, D. and Heejong Yoo, "Design analysis of a distributed arithmetic adaptive fir filter on an fpga," in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, vol. 1, pp. 926–930, 9–12 Nov. 2003.

[54] D. Allred, H. Yoo, V. Krishnan, W. Huang, and D. V. Anderson, "LMS adaptive filters using distributed arithmetic for high throughput," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52, pp. 1327–1337, July 2005.

[55] G. Strang, *Linear Algebra and Its Applications*. Brooks Cole, 1988.

[56] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Springer, 1994.

[57] L. Dadda and V. Piuri, "Pipelined adders," *IEEE Transactions on Computers*, vol. 45, pp. 348–356, March 1996.

[58] B. Parhami, "Carry-free addition of recoded binary signed-digit numbers," *IEEE Transactions on Computers*, vol. 37, pp. 1470–1476, Nov. 1988.

[59] B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Transactions on Computers*, vol. 39, pp. 89–98, Jan. 1990.

[60] S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of high speed MOS multiplier and divider using redundant binary representation," in *Proc. 8th Symp. Comput. Arithmetic*, pp. 80–86, 1987.

[61] H. Dillon, *Hearing Aids*. Thieme, 2001.

[62] A. Schaub, *Digital Hearing Aids*. Thieme, 2008.

[63] Hearing Central LLC Group.