IMPLEMENTATION OF ADAPTIVE DIGITAL FIR AND REPROGRAMMABLE MIXED-SIGNAL FILTERS USING DISTRIBUTED ARITHMETIC

A Dissertation
Presented to
The Academic Faculty

By

Walter G. Huang

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
December 2009

IMPLEMENTATION OF ADAPTIVE DIGITAL FIR AND REPROGRAMMABLE MIXED-SIGNAL FILTERS USING DISTRIBUTED ARITHMETIC

Approved by:

Dr. David V. Anderson, Advisor
Assc. Professor, School of ECE
Georgia Institute of Technology

Dr. Sung Ha Kang
Assistant Professor, School of Mathematics
Georgia Institute of Technology

Dr. Bonnie H. Ferri
Professor, School of ECE
Georgia Institute of Technology

Dr. James H. McClellan
Professor, School of ECE
Georgia Institute of Technology

Dr. Paul E. Hasler
Assc. Professor, School of ECE
Georgia Institute of Technology

Dr. Wayne H. Wolf
Professor, School of ECE
Georgia Institute of Technology

Date Approved: November 2009

DEDICATION

To my family, my Mom, my Dad, and my sister, for their love, support, and patience

ACKNOWLEDGMENTS

First of all, I would like to thank my family, my Mom, my Dad, and my sister, for their love, support, and patience. Their love means very much to me and inspired me through the challenges of graduate research to this meaningful milestone in my life. I want to tell them that "I love you. I am who and where I am today because of you. Thank you."

I would like to thank and to express my sincere gratitude to my dissertation advisor, Dr. David Anderson, for his endless support and patience. I can truly say he is the right dissertation advisor for me. Also, I would like to thank my dissertation committee members, Dr. Bonnie Ferri, Dr. Paul Hasler, Dr. Sung Ha Kang, Dr. James McClellan, and Dr. Wayne Wolf, for serving on my committee and for their comments and suggestions. Without my dissertation advisor and my committee members, I would still be mindlessly wandering the halls of Georgia Tech.

Finally, I would like to thank my lab mates and my social circle here at Georgia Tech for their friendship and support. They are true friends, and made my time at Georgia Tech unforgettable. I will cherish the times that we had together here in Atlanta, and I will miss the camaraderie that we shared. I take comfort in knowing we share this bond of friendship wherever we may go.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

When computational resources are limited, especially multipliers, distributed arithmetic (DA) is used in lieu of the typical multiplier-based filtering structures. However, DA is not well suited for adaptive applications. The bottleneck is updating the memory table. Several attempts have been done to accelerate updating the memory, but at the expense of additional memory usage and of convergence speed.

To develop an adaptive DA filter with an uncompromised convergence rate, the memory table must be fully updated. In this research, an efficient method for fully updating a DA memory table is proposed. The proposed update method is based on exploiting the temporal locality of the stored data and subexpression sharing. The proposed update method reduces the computational workload and requires no additional memory resources. DA using the proposed update method is called conjugate distributed arithmetic.

Filters can also be constructed from analog components. Often, for lower precision computations, analog circuits use less power and less chip area than their digital counterparts. However, digital components are often used because of their ease of reprogrammability. Achieving such reprogrammability in analog is possible, but at the expense of additional chip area.

A reprogrammable mixed-signal DA finite impulse response (FIR) filter is proposed to address the issues with reprogrammable analog FIR filters that are constructing compact reprogrammable filtering structures, non-symmetric and imprecise filter coefficients, inconsistent sampling of the input data, and input sample data corruption. These issues are successfully addressed using distributed arithmetic, digital registers, and epots.

Also, a mixed-signal DA second-order section (SOS), which is used as the building block for higher order infinite impulse response filters, was proposed. The type of issues with an analog SOS filter are similar to those of an analog FIR filter, which are the lack

of a compact reprogrammable filtering structure, the imprecise filter coefficients, the inconsistent sampling of the data, and the corruption of the data samples. These issues are successfully addressed using distributed arithmetic and digital registers.

CHAPTER 1

INTRODUCTION

Distributed arithmetic (DA) is commonly used for signal processing algorithms where computing the inner product of two vectors comprises most of the computational workload. This type of computing profile describes a large portion of signal processing algorithms, so the potential usage of distributed arithmetic is tremendous.

The inner product is commonly computed using multipliers and adders. When computed sequentially, the multiplication of two B-bit numbers requires from $B/2$ to B additions, and is time intensive. Alternatively, the multiplication can be computed in parallel using $B/2$ to B adders, but is area intensive [1, 2]. Whether a K-tap filter is computed serially or in parallel, it requires at least $B/2$ additions per multiplication plus the $K-1$ additions for summing the products together. In the best case scenario, $K \cdot (B+2)/2 - 1$ additions are needed for a K-tap filter using multipliers and adders.

A competitive alternative to using a multiplier is distributed arithmetic. It compresses the computation of a K-tap filter from K multiplications and $K-1$ additions into a memory table and generates a result in B-bit time using $B-1$ additions. DA significantly reduces the number of additions needed for filtering [3, 4]. This reduction is particularly noticeable for filters with high bit precision. This reduction in the computational workload is a result of storing the pre-computed partial sums of the filter coefficients in the memory table [1].

When compared with other alternatives, distributed arithmetic requires fewer arithmetic computing resources and no multipliers. This aspect of distributed arithmetic is a favorable one for computing environments with limited computational resources, especially multipliers. These type of computing environments can be found on older field-programmable gate arrays (FPGAs) and low-end, low-cost FPGAs. By using distributed arithmetic, these type of devices can be used for low latency, area constrained, high-order filters. Implementing such a filter using a multiplier based approach would be difficult.

1.1 Adaptive Filters using Distributed Arithmetic

Another important signal processing area is adaptive filtering. Adaptive filtering is extensively used in several signal processing applications including acoustic echo cancellation, signal de-noising, sonar signal processing, clutter rejection in radars, and channel equalization for communication and networking systems [5, 6]. For adaptive filtering applications, distributed arithmetic has not worked well for the following issues. First, the typical computational flow of distributed arithmetic for adaptive filtering requires a significant increase in the computational workload over the non-adaptive case and a noticeable increase in the computational time when constrained with limited computing resources. These additional resources are needed for updating the contents of the memory table associated with distributed arithmetic. For these applications, one of the typical advantages of distributed arithmetic that is the low computing requirement is significantly diminished or eliminated. For instance in a brute-force implementation, this straight forward updating method would take $(K/2-1)2^K+1$ additions for a K-tap FIR filter. When contrasted with the number of additions used for just filtering the data, this type of update method uses a factor of order $K \cdot 2^{(K-1)}/B$ more additions, where B is the bit precision of the input data for a conventional distributed arithmetic filter. For example when $K = 16$ and $B = 8$, a factor of order 2^{16} more additions are required to update the memory table than to compute the output sample. Several attempts have been done to accelerate the process of updating the memory. Although these approaches do reduce the amount of processing necessary to update the memory, this reduction is gained at the expense of additional memory usage and of convergence speed.

To address these issues, a new type of adaptive distributed arithmetic filter is proposed. The computational workload was significantly reduced by modifying the computational flow and utilizing an efficient method for updating the memory table contents without compromising the convergence speed or requiring additional memory resources. Details of these modifications for the proposed filter are provided in Section 3.1.

1.2 Reducing the Memory Usage of Adaptive Distributed Arithmetic Filters

Another aspect of distributed arithmetic that may dissuade one from using it is its relatively high memory usage. This issue is equally relevant to both non-adaptive and adaptive applications. Inherently, distributed arithmetic requires more memory than other alternatives; therefore, the memory ratio comparing distributed arithmetic with an alternative is always greater than one, and is typically significantly greater than one. For example, a 128-tap DA FIR filter without any memory optimizations will require a prohibitively large 2^{128} entries in the memory table.

To mitigate this issue for a particular type of adaptive DA filter, sliding-block distributed arithmetic was modified to significantly reduce the memory usage. This reduction was achieved by encoding the memory table contents. With additional modifications to the computational flow, the computing workload is reduced as well. Details of this modified sliding-block distributed arithmetic are given in Section 3.2.

1.3 Applications of Distributed Arithmetic Outside the Digital Domain: Mixed-Signal DA Filters

The filters described so far have used digital components. Filters can also be implemented using analog components. Typically, analog circuits use less power and less chip area than their digital counterparts for low precision computations. However, even with these advantages, the digital components are often preferred over the analog ones because of the ease of reprogrammability of the digital systems. A common method to achieve reprogrammability for analog systems is to utilize a bank of components to emulate the variability of key components such resistors or capacitors. A straight forward example of constructing a variable capacitor is to use a bank of capacitors whose size is scaled by factors of two and connected in parallel through a bank of switches. This type of variable component is reprogrammable using digital words. However, this type of structure dramatically increases the chip area consumed.

When analog circuits are used to implement filters with wide linear phase, which is important for applications such as image processing, a couple typical approaches exist for constructing such a filter. Each of these approaches has its own set of issues.

One common approach is to use traditional analog techniques using a combination of resistors, capacitors, transistors, and/or amplifiers. Typically, designing such filters with these components is time consuming, and its frequency response is not reprogrammable. As mentioned earlier, the filter can be redesigned with reprogrammability in mind; however, the chip area consumed increases dramatically and typically eliminates the benefit of low chip area when compared with their digital counterparts. This type of approach is commonly found for both continuous-time and discrete-time filters.

Filters with wide linear phase can also be implemented using a finite impulse response (FIR) filtering structures. The only requirement for making an FIR filter with linear phase is to ensure that the filter coefficients are symmetric. The type of symmetry, either even or odd, does not matter. When using digital components, this symmetry is easily achieved. In the analog domain, to achieve such symmetry is more difficult. The key point to achieving symmetry is making sure the filter coefficients are precise. If the filter coefficients are not precise, then the linear phase of the analog FIR filter is affected. Also if the filter coefficients are too imprecise, then the frequency response of the filter is affected. The issues that hamper symmetry and filter coefficient precision are process variations such as device mismatch. These issues can be overcome by using additional analog design techniques; however, these techniques come at the expense of consuming additional chip area. If reprogrammability of the filter coefficients is required, then these issues are further compounded, and the chip area consumed is increased even more when compared with a non-reprogrammable FIR filter.

Another issue with implementing an analog FIR filter is sampling and storing the input data especially for high-order filters. When using continuous-time components, two common ways exist to generate the appropriately delayed input samples. One approach is to use

a cascade of identical all-pass filters with a constant group delay appropriate for the desired sampling rate in the frequency range of interest. When using such a filtering structure, a couple problems need to be overcome. The first issue is related to the process variations such as the device mismatch. As stated earlier, this issue can be overcome by using additional analog design techniques at the expense of additional chip area. If the all-pass filter is not designed to tolerate process variations, then the group delay in the desired frequency range may not be constant, the group delay for each all-pass filter is different, and/or the group delay is not appropriate for the desired sampling rate. This lack of tolerance results is inconsistent sampling of the input data. The second issue is the noise that accumulates as the input signal is cascaded from one all-pass filter to the next. After cascading through a few all-pass filters, the input signal will become unusable. This effect limits the highest achievable order for an FIR filter. In turn, this limits the frequency roll-off and limits the variety of frequency responses that is obtainable.

The other approach when using continuous-time components for sampling and storing the input data for an analog FIR filter is to use a bank of all-pass filters with a group delay scaled according to the appropriate multiple of the sample period for the desired sampling rate in the frequency range of interest. This approach also is affected by issues related to process variations. These issues may result in the group delay in the desired frequency range to be not constant and/or to be not scaled accordingly to the appropriate multiple of the sample period. In other words, these issues result in an inconsistent sampling of the input data. The other downside of this approach is that each all-pass filter is not identical; therefore, this approach requires more design effort than the first one. For this approach, each all-pass filter must be designed for a different amount of group delay. In the first case, only one type of all-pass filter must be designed.

Discrete-time analog components are also used to sample and to store the input data for analog FIR filters. The typical way to sample and to store the input data using discrete-time components is to use a sample and hold circuit. By using these type of circuits, the issue

of inconsistently sampling the input data is minimized. These circuits can be configured a couple different ways. The first way is a cascade of sample and hold circuits connected to the same sample clock. As in the continuous-time case when using a cascade of all-pass filters, this approach has a significant issue with noise accumulation, and the input signal becomes unusable after being cascaded through a few sample and hold circuits. This accumulation of noise limits the highest achievable order for an FIR filter to a few filter taps.

Instead of connecting the sample and hold circuits in a cascaded fashion, another way is to connect each sample and hold circuit to the input signal where only one circuit is sampling the input signal every sample period and to connect the appropriate input samples with the appropriate filter coefficients. This connection of the input sample with the filter coefficient can be done in two ways. The first way is to connect the appropriate input sample to the appropriate filter coefficient through a switching matrix where each sample and hold circuit can be connected to each filter coefficient. This approach eliminates the noise accumulation issue when using a cascaded approach. The principle issue with this approach is that the switching matrix requires the filter length squared number of switches. In other words for a K-tap FIR filter, K^2 number of switches is needed for the switching matrix. The other way to connect the input sample to the filter coefficient is to rotate the filter coefficients to the appropriate input samples as if using a circular buffer. This approach eliminates the switching matrix; however, this approach does not eliminate the noise accumulation issue. Instead of accumulating noise in the input samples, the noise is accumulated in the filter coefficients.

In the proposed reprogrammable mixed-signal distributed arithmetic FIR filter, the typical issues associated with reprogrammable analog FIR filters, which are the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter coefficients, the inconsistent sampling of the input data, and the corruption of the input samples, are addressed. These issues are addressed using a combination of distributed arithmetic,

digital storage elements, and epots, which are compact, reprogrammable, and precise voltage references. Since the use of distributed arithmetic has not been done before using a combination of analog and digital components for a mixed-signal FIR filter, potential error sources due to using analog components are analyzed. Details of the proposed reprogrammable mixed-signal distributed arithmetic FIR filter and the analysis of the potential error sources are located in Chapter 4.

For applications that require a steep roll-off in frequency response, infinite impulse response (IIR) filters are better suited than finite impulse response (FIR) filters. Typically, IIR filters can be designed with frequency roll-off higher than FIR filters of the same order. Typically, high order IIR filters are constructed from second-order sections (SOSs). The construction of an analog IIR filter is similar to that of an analog FIR filter, and an analog IIR filter has issues similar to that of an FIR one, which are the lack of a compact reprogrammable filtering structure, the imprecise filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples. The key difference between the two is that a feedback path is present in addition to the feedforward path, which is also present in an FIR filter. The feedback path is constructed in a similar manner to the feedforward path except that it uses the output samples instead of the input samples.

In the proposed reprogrammable mixed-signal distributed arithmetic second-order section filter, the typical issues associated with reprogrammable analog IIR filters, which are the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples, are addressed. These issues are addressed using a combination of distributed arithmetic and digital storage elements. Since the use of distributed arithmetic has not been done before using a combination of analog and digital components for a mixed-signal IIR filter, potential error sources due to using analog components are analyzed. Details of the proposed reprogrammable mixed-signal distributed

arithmetic second-order section filter and the analysis of the potential error sources are located in Chapter 5.

1.4 Contributions

In this research, contributions are made in the following fields. The first one is in the development of an adaptive filter using distributed arithmetic. The issues addressed by this research are the lack of an efficient method to fully update the memory table of a distributed arithmetic adaptive filter, the usage of memory resources beyond that of the non-adaptive case, and the compromised convergence rate. The following contributions were made to address these issues and to develop an adaptive distributed arithmetic filter with an efficient method to fully update the memory table without using additional memory resources and with uncompromised convergence performance.

1. A new method for fully updating the memory table was proposed. By fully updating the memory table, the convergence performance of the adaptive filter remains unaffected; therefore, the proposed update method can be used to construct adaptive filtering structures using distributed arithmetic without a compromised convergence rate.

2. A new method for efficiently updating the entire memory table was proposed. The proposed update method reduced the computational workload for updating the memory table of a k-tap filter by about $k/2 - 1$ over brute-force.

3. By using a filter coefficient driven distributed arithmetic filtering structure, the additional memory resources required by most other types of adaptive distributed arithmetic filtering structures, which use an input driven memory table that is composed of the combinations of its filter coefficients, are eliminated.

In this research, the proposed memory update method is combined with a filter coefficient driven distributed arithmetic filtering structure. This combination is called conjugate distributed arithmetic (CDA).

After a through literature review of adaptive distributed arithmetic filtering structures, only one other type of adaptive DA that is called sliding-block distributed arithmetic (SBDA) also addresses the issues outlined above with an adaptive DA filter. Although CDA is not the only adaptive distributed arithmetic filtering structure that addresses these issues, CDA is advantageous in a variety of filter configurations.

1. Among the adaptive distributed arithmetic filters that fully updates its memory tables, CDA uses the least amount of memory. Its memory usage is only matched by the brute-force method; however, CDA reduces the number of operations required by about $k/2-1$ over brute force for a k-tap filter. Recall, only adaptive distributed arithmetic filters that fully updates its memory tables is able to maintain the convergence speed of the LMS algorithm.

2. CDA uses less memory than SBDA especially if the filter is broken up into few subunits. This advantage is useful when coded on a system with limited memory.

3. CDA uses fewer additions than SBDA when the bit precision of the filter coefficients in SBDA is greater than the number of additional memory table entries that need to be updated in CDA. Typically, this occurs when the coefficient bit precision, the number of subunits, the depth of the memory tables, or a combination of these three are low. A couple benefits of fewer additions are boosted sampling rate, or lower power usage.

In addition to CDA and SBDA, an alternative adaptive DA filter structure called SBDA-OBC was proposed and developed. Its memory update method is a modification of the one used in SBDA, and it has lower memory usage and fewer additions for most filter configurations over SBDA. The principle motivations for modifying SBDA are to encode

the memory using OBC such that the memory usage is reduced almost in half and to modify SBDA in such a way that when the memory is encoding using OBC that the computational workload for updating the memory table is reduced. The following are the observed savings of SBDA-OBC.

1. SBDA-OBC has the lowest memory requirements of any current mechanization for a coefficient driven, memory-based DA adaptive FIR filter. Specifically, SBDA-OBC uses about 50% less memory than SBDA when the filter length of the subunits is long.

2. SBDA-OBC has the fewest number of additions for a large number of filtering configurations among the current mechanizations for a coefficient driven, memory-based DA adaptive FIR filter. Specifically, SBDA-OBC needs about 50% less additions than SBDA when the filter length of the subunits is long.

Contributions were also made in the field of reprogrammable mixed-signal FIR filter design. The issues addressed by this research are the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter coefficients, the inconsistent sampling of the input data, and the corruption of the input samples. The following contributions were made to address these issues.

1. A reprogrammable mixed-signal FIR filter was proposed and developed to address the issues of the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter coefficients, the inconsistent sampling of the input data, and the corruption of the input samples.

2. Distributed arithmetic and epots were used to address the issue of a lack of a compact reprogrammable filtering structure for a reprogrammable mixed-signal FIR filter. A combination of distributed arithmetic and epots were used to construct a compact reprogrammable mixed-signal FIR filter.

3. epots were used to address the issue of non-symmetric and imprecise filter coefficients for a reprogrammable mixed-signal FIR filter. epots were used to precisely reprogram the filter coefficients. These filter coefficients can be programmed with such precision that a natural by-product is high filter coefficient symmetry.

4. Digital registers were used to address the issues of inconsistent input data sampling and of input sample data corruption for a reprogrammable mixed-signal FIR filter. The digital registers were used to sample the input data consistently and to eliminate input data corruption. Because the digital registers can be cascaded without concern for data corruption, this ability to cascade many input digital registers together allows for the construct of a high order FIR filter using the proposed reprogrammable mixed-signal FIR filtering structure.

5. An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed reprogrammable mixed-signal FIR filter. Deduced from this analysis, a guideline was generated of where to focus the design effort for the proposed reprogrammable mixed-signal FIR filter. The guideline is to use most of the design effort on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

Finally, contributions were made in the field of reprogrammable mixed-signal second-order section design. The issues addressed by this research are the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples. The following contributions were made to address these issues.

1. A reprogrammable mixed-signal second-order section was proposed and developed to address the issues of the lack of a compact reprogrammable filtering structure, the

imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples.

2. Distributed arithmetic and digital registers were used to address the issue of a lack of a compact reprogrammable filtering structure for a reprogrammable mixed-signal second-order section. A combination of distributed arithmetic and digital registers were used to construct a compact reprogrammable mixed-signal second-order section.

3. Digital registers were used to address the issue of imprecise feedback and feedforward filter coefficients for a reprogrammable mixed-signal second-order section. The digital registers were used to precisely reprogram the filter coefficients.

4. Independently clocked sample and hold circuits and a circular buffer of digital registers were used to address the issues of inconsistent sampling of the input and output data and of the corruption of the input and output samples for a reprogrammable mixed-signal second-order section. The independently clocked sample and hold circuits and the circular buffer of digital registers were used to consistently sample the input and output data and to minimize the corruption of the input and output samples.

5. An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed reprogrammable mixed-signal second-order section. Deduced from this analysis, a guideline was generated of where to focus the design effort for the proposed reprogrammable mixed-signal second-order section. The guideline is to use most of the design effort on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

CHAPTER 2

BACKGROUND MATERIAL

In the first section of this chapter, a review of distributed arithmetic for FIR filters is presented. This section includes material on common optimizations used with distributed arithmetic. These optimizations are commonly applied to adaptive distributed arithmetic filters.

The second section is a review of distributed arithmetic for adaptive applications, in particular using the LMS algorithm. This section focuses on the two different approaches for updating the memory tables used for distributed arithmetic, which are to partially update the memory tables or to completely update the memory tables. This section discusses the trade-offs between the two approaches in terms of computational workload and adaptive convergence speed.

2.1 Review of Distributed Arithmetic FIR Filters

One way to take advantage that the density of memory elements is growing faster than of computational logic elements is to use an implementation that utilizes memory for more than storage. One such mechanization is called distributed arithmetic (DA). In following sections, a brief background and common hardware optimizations of DA is provided.

2.1.1 Distributed Arithmetic

Distributed arithmetic (DA) was first introduced by Croisier et al. [7] and further developed by Peled and Lui [3]. DA is a multiplier-less implementation for computing the inner product of a pair of vectors [8], a common computation used in digital signal processing. It is well suited to implementing high throughput FIR filters and signal transformations such as discrete cosine transforms or fast fourier transforms. DA is a bit-serial computation that forms an inner product of a pair of vectors in a few steps by storing all possible combination sums of weights in a memory table. It is assumed that the inputs to the filter are represented

as B-bit 2's complement binary numbers with only the sign bit to the left of the radix point. A discrete-time linear finite impulse response filter generates the output $y[n]$ as a sum of delayed and scaled input samples $x[n]$. In other words,

$$y[n] = \sum_{k=0}^{K-1} w_k x[n-k]. \tag{2.1}$$

Let the signal samples to the filter be represented as B-bit 2's complement binary numbers,

$$x[n-i] = -b_{i0} + \sum_{l=1}^{B-1} b_{il} 2^{-l}, \quad i = 0, \ldots, K-1, \tag{2.2}$$

where b_{il} is the l^{th} bit in the 2's complement representation of $x[n-i]$. Substituting (2.2) into (2.1) and swapping the order of the summations yields

$$y[n] = -\left[\sum_{i=0}^{K-1} b_{i0} w_i\right] + \sum_{l=1}^{B-1}\left[\sum_{i=0}^{K-1} b_{il} w_i\right] 2^{-l}. \tag{2.3}$$

For a given set of w_i ($i = 0, \ldots, K-1$), the terms in the square braces may take only one of 2^K possible values, which may be stored in a memory table, denoted as the DA filtering memory table (DA-F-MEM). The entry in the DA-F-MEM addressed by r, is given by

$$\text{DA-F-MEM}_{(r)} = \sum_{i=0}^{K-1} c_i^{(r)} w_i, \quad r = 0, \ldots, 2^K - 1, \tag{2.4}$$

where $c_i^{(r)}$ is the i^{th} bit in the K-bit representation of the address r. In other words,

$$r \;=\; \sum_{i=0}^{K-1} c_i^{(r)} 2^i. \tag{2.5}$$

For each l, $l = 0, \ldots, B-1$, the term in the square braces in (2.3) is essentially the entry in the DA-F-MEM whose address is $\sum_{i=0}^{K-1} b_{il} 2^i$.

A 4-tap ($K = 4$) implementation of the DA FIR filter is shown in Figure 2.1. The DA-F-MEM contains all 16 possible combination sums of the filter weights $w_0, w_1, w_2,$ and, w_3. The bank of shift registers in Figure 2.1 stores 4 consecutive input samples ($x[n-i], i = 0, \ldots, 3$). The concatenation of rightmost bits of the shift registers becomes the address of the memory table. The shift register is shifted right at every clock cycle. The corresponding memory table entries are also shifted and accumulated B consecutive times to generate the

output $y[n]$ where B is the precision of the input data. The sign bit control is used to change the addition to subtraction for the sign bits which are included in the first expression square brackets in Eq. (2.3). In addition, computing the filtering operation by utiliziing the DA filter can be done in B clock cycles regardless of the size of the filter, K. Thus, obtaining a high throughput rate using the DA implementation, especially if $K \gg B$, is possible. Also due to the regular structure of DA, the logic complexity is low. A comprehensive tutorial review of DA linear filters is given in [4].

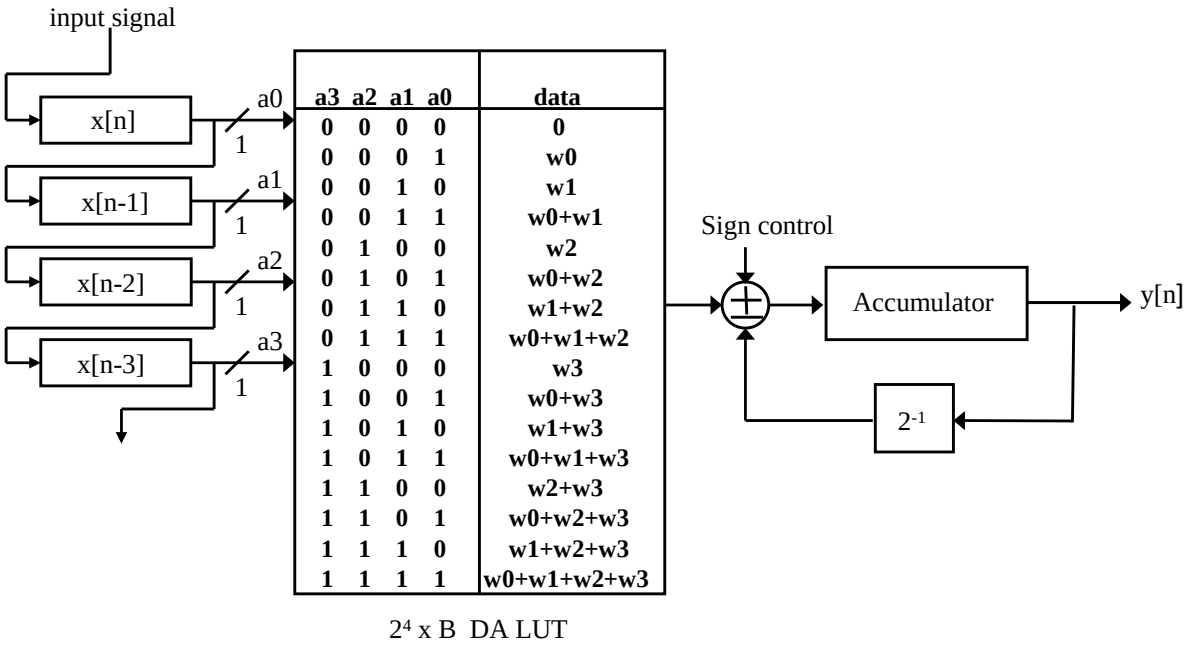


Figure 2.1. Block diagram of DA implementation of a 4-tap ($K = 4$) FIR filter. Each coefficient has B bits of precision (ex. B=16).

This speed advantage comes at a cost. First, distributed arithmetic is not well suited for adaptive filtering applications. When used for such applications, updating the associated memory tables with their appropriate values using brute-force methods utilizes a significant amount of additional hardware and considerably depreciates the throughput of the adaptive filter to the point where using distributed arithmetic has no advantage versus typical hardware implementations. Several past attempts have been made to implement adaptive filters using DA [9, 10], but the approximations that are used to modify standard

adaptation algorithms may not be suitable for many practical applications.

The second disadvantage is its high memory usage. By using some techniques presented in a distributed arithmetic tutorial paper by White [4], this problem can be mitigated to a significant extent. But even using these methods, the memory usage is still higher when compared to conventional implementations which may prevent the usage of distributed arithmetic in certain applications such as FIR filtering.

2.1.2 Common Optimizations

Even in its most fundamental implementations, DA utilizes an unreasonable amount of memory for high order filters. In this section, two common optimizations are presented that mitigates and reduces the amount of memory to within reasonable limits. The first optimization reduces the memory by splitting the filter up into smaller base filters. The second one reduces the memory by encoding the contents in offset binary coding (OBC). These optimizations are described in further detail in the sections below.

2.1.2.1 DA Filter Design for Large Filter Sizes

As the filter size increases, the memory requirements of the implementation described in the previous section grow exponentially. For example, a 128-tap DA FIR filter will use a prohibitively large 2^{128} entries in the DA-F-MEM. This problem may be alleviated by breaking up the filter into smaller base DA filtering units that utilize tractable memory sizes and then summing up the outputs of these units.

The summation in the square braces in Eq. (2.3) may be split so that K-tap filter is divided into m smaller filters each having k-tap DA base units ($K = m \times k$). Here it is assumed that K is not prime. Thus, Eq. (2.3) can be written as

$$y[n] = -\left(\sum_{j=0}^{m-1}\left[\sum_{i=jk}^{(j+1)k-1} b_{i0}w_i\right]\right) + \sum_{l=1}^{B-1}\left(\sum_{j=0}^{m-1}\left[\sum_{i=jk}^{(j+1)k-1} b_{il}w_i\right]\right)2^{-l} \tag{2.6}$$

The terms in parentheses in Eq. (2.6) may be implemented using m DA base units, each implementing the expression in square brackets. In Figure 2.2, the implementation of a 4-tap FIR filter based on Eq. (2.6), for $m = 2$ and $k = 2$ is shown.

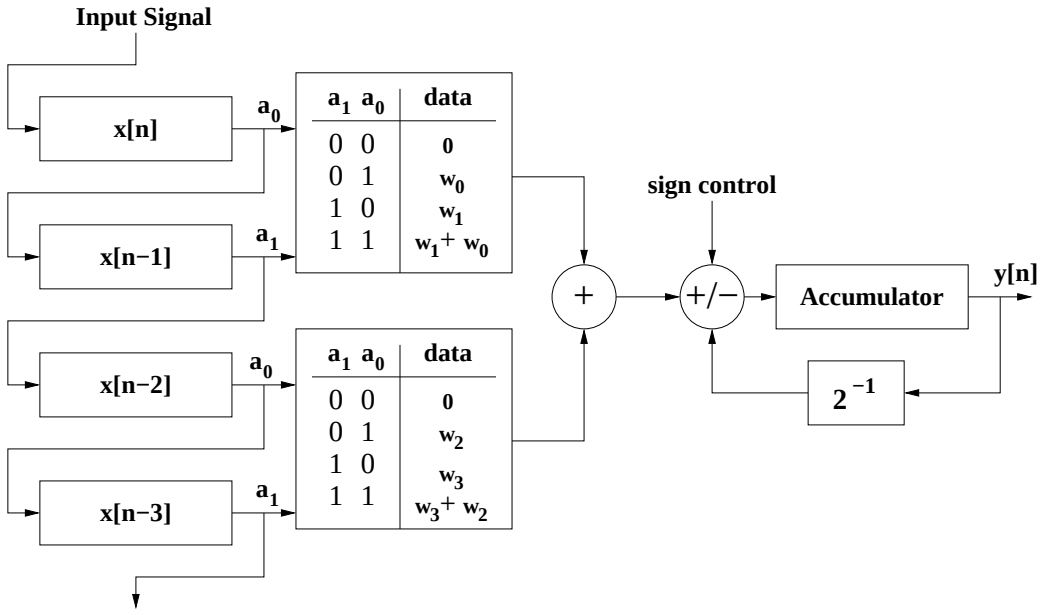


Figure 2.2. Implementation of 4-tap DA FIR filter using $m = 2$ and $k = 2$.

The total memory requirement for implementation of Eq. (2.6) is $m \times 2^k$ memory elements. The total number of clock cycles used for this implementation will be $B + \lceil \log_2(m) \rceil$; the additional second term is the number of clock cycles utilized to implement an adder tree to calculate the sums of the units. Thus, the decrease in throughput of this implementation is marginal. For instance if $K = 128$, a full memory table implementation would use 2^{128} memory elements. Instead, a designer can choose $k = 4$ and $m = 32$ to implement a filter with only 512 memory elements. The number of clock cycles utilized for this implementation would be 21 clock cycles. In comparison, the single memory table implementation would use 16 clock cycles.

2.1.2.2 Distributed Arithmetic using Offset Binary Coding

Offset binary coding can be used to halve the size of the memory tables in DA [4]. The derivation for OBC begins with writing the input, $x[n - i]$, as follows.

$$x[n-i] = \frac{1}{2}\{x[n-i] - (-x[n-i])\}, \quad i = 0, \ldots, K-1 \tag{2.7}$$

Next, x_i and $-x_i$ are written in two's complement notation as shown in Eq. (2.8) and Eq. (2.9) respectively and substituted back into Eq. (2.7) to yield Eq. (2.10).

$$x[n-i] = -b_{i0} + \sum_{l=1}^{B-1} b_{il} 2^{-l} \tag{2.8}$$

$$-x[n-i] = -\bar{b}_{i0} + \sum_{l=1}^{B-1} \bar{b}_{il} 2^{-l} + 2^{-(B-1)} \tag{2.9}$$

$$x[n-i] = \frac{1}{2}\left[-(b_{i0} - \bar{b}_{i0}) + \sum_{l=1}^{B-l} (b_{il} - \bar{b}_{il}) 2^{-l} - 2^{-(B-1)}\right] \tag{2.10}$$

Eq. (2.10) can be rewritten as Eq. (2.11) where the variable, c_{il}, is defined as $b_{il} - \bar{b}_{il}$. This change of variables translates $b_{il} = 0$ to a subtraction of $\frac{h[i]}{2}$ and translates $b_{il} = 1$ to an addition of $\frac{h[i]}{2}$ where previously $b_{il} = 0$ was an addition of zero and $b_{il} = 1$ was an addition of $h[i]$.

$$x[n-i] = \frac{1}{2}\left[-c_{i0} + \sum_{l=1}^{B-1} c_{il} 2^{-l} - 2^{-(B-1)}\right] \tag{2.11}$$

Now with $x[n-i]$ written in OBC notation, its incorporation into the FIR filtering equation is detailed below in Eq. (2.14) where $Q(b_l) = \sum_{i=0}^{K-1} \frac{h[i]}{2} c_{il}$ and $Q(0) = \sum_{i=0}^{K-1} \frac{-h[i]}{2}$.

$$y[n] = \sum_{i=0}^{K-1} h[i] x[n-i] \tag{2.12}$$

$$y[n] = \frac{1}{2} \sum_{i=0}^{K-1} h[i] \left[-c_{i0} + \sum_{l=1}^{B-l} c_{il} 2^{-l} - 2^{-(B-1)}\right] \tag{2.13}$$

$$y[n] = -Q(b_0) + \sum_{l=1}^{B-1} Q(b_l) 2^{-l} + 2^{-(B-1)} Q(0) \tag{2.14}$$

The next step is to use Eq. (2.14) and to identify what hardware maps to each term. As a preview to this mapping, a block diagram of the hardware implementation is shown in Figure 2.3. First, the B-bit input samples needs to be serialized to format them for usage with DA. This function is performed by the parallel input, serial output (PISO) units. They convert the K input samples into K-bit streams where the bit b_{il} is associated with the l^{th} bit of the bit stream for $x[n-i]$. These bit streams are outputted from the LSB, $l = 0$, first to the MSB, $l = B-1$, last. Then, these b_{il} bits are used to form the address, b_l that is equal to

ROM Address a_0	a_1	a_2	a_3	16-Word ROM Contents, Q
0	0	0	0	½(–h[0] – h[1] – h[2] – h[3]) = Q(0)
0	0	0	1	½(–h[0] – h[1] – h[2] + h[3]) = Q(1)
0	0	1	0	½(–h[0] – h[1] + h[2] – h[3]) = Q(2)
0	0	1	1	½(–h[0] – h[1] + h[2] + h[3]) = Q(3)
0	1	0	0	½(–h[0] + h[1] – h[2] – h[3]) = Q(4)
0	1	0	1	½(–h[0] + h[1] – h[2] + h[3]) = Q(5)
0	1	1	0	½(–h[0] + h[1] + h[2] – h[3]) = Q(6)
0	1	1	1	½(–h[0] + h[1] + h[2] + h[3]) = Q(7)
1	0	0	0	½(h[0] – h[1] – h[2] – h[3]) = Q(8)
1	0	0	1	½(h[0] – h[1] – h[2] + h[3]) = Q(9)
1	0	1	0	½(h[0] – h[1] + h[2] – h[3]) = Q(10)
1	0	1	1	½(h[0] – h[1] + h[2] + h[3]) = Q(11)
1	1	0	0	½(h[0] + h[1] – h[2] – h[3]) = Q(12)
1	1	0	1	½(h[0] + h[1] – h[2] + h[3]) = Q(13)
1	1	1	0	½(h[0] + h[1] + h[2] – h[3]) = Q(14)
1	1	1	1	½(h[0] + h[1] + h[2] + h[3]) = Q(15)

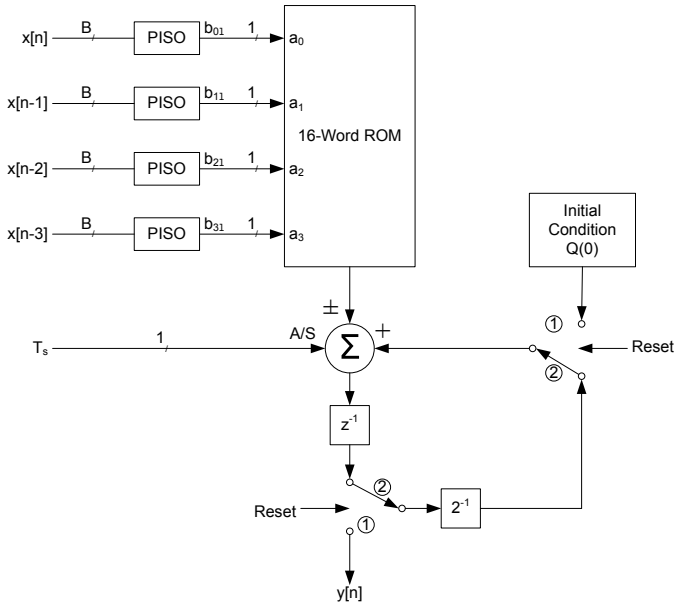


Figure 2.3. A block diagram of an input driven DA FIR filter using OBC.

$\sum_{i=0}^{K-1} b_{il}2^i$, for a memory that stores all the partial sums of $Q(b_l)$. Every possible value of $Q(b_l)$ is stored in a 2^K entry memory. For initialization of the filter, the switches are set to position 1. These switches are controlled by the Reset signal that is set to high at this time. With the present setting of the Reset signal, the output of the memory is added with the initial condition. This addition correlates with the term $2^{-(B-1)}Q(0)$, which is to add $Q(0)$ with the partial product of the LSB, in the equation. After the first word of data is outputted, the Reset signal goes low, and the switches move to position 2. Since the filter begins with the LSB, the T_s bit or the sign bit is set to "0". Now, the output of the memory is added with one half of the previous summation. This feedback loop corresponds to the summation term, $\sum_{l=1}^{B-1} Q(b_l)2^{-l}$. The filter continues operating in this manner until the MSB. At this time, the sign bit is set high and the output of the memory is subtracted from the previous summation. This operation correlates to the term $-Q(b_0)$. Finally to output the completed computation of $y[n]$, the Reset signal is set high, and the switches are set to position 1.

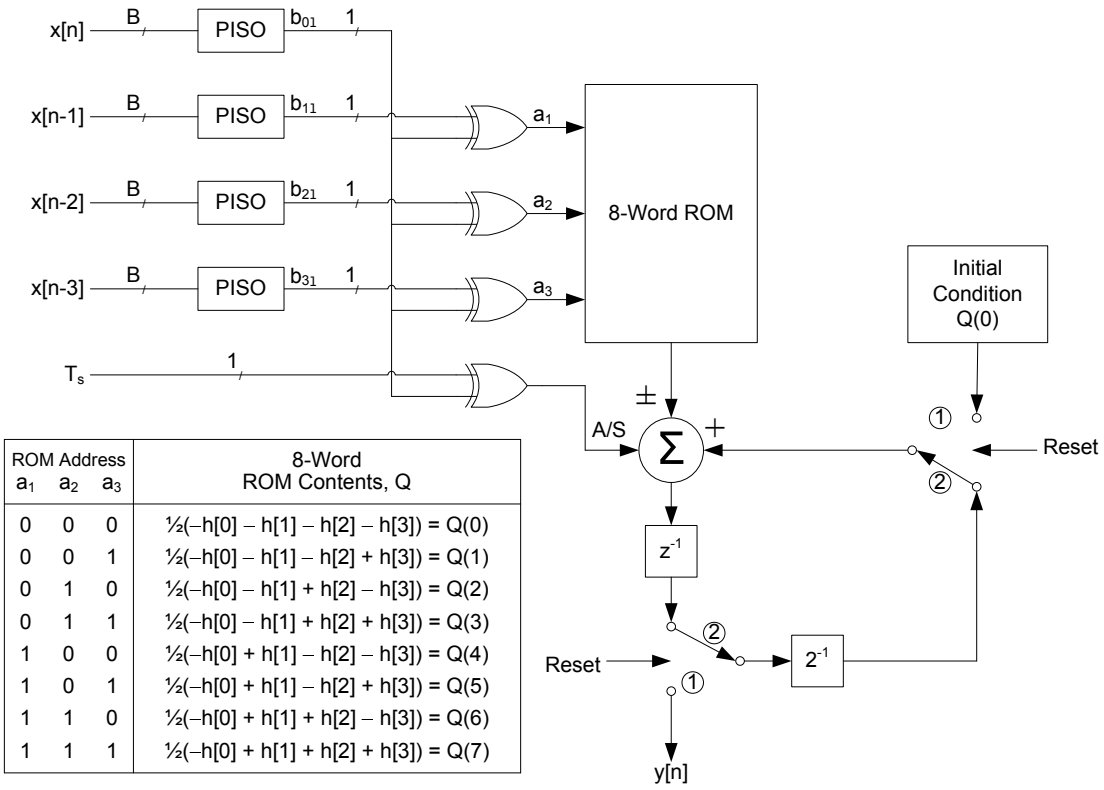
The memory usage can be reduced by a factor of two if one observes that the upper half of the ROM table in Figure 2.3 is a mirror image of the lower half. By using only the upper

half of the memory to generate the output, this restriction implies that only the address bits a_1 through a_{K-1} are used and that a_0 is equal to "0". With a_0 always equal to "0", a problem arises on how to add values where b_{0l} = "1", which implies that a positive $\frac{h[0]}{2}$ is included in the partial product for the l^{th} bit. A simple solution to generating the partial products where b_{0l} = "1" is to subtract the output from the ROM when T_s = "0" and to add when T_s = "1". However when b_{0l} = "0", the usage of the sign bit remains unchanged, which is that the output of the ROM is added when T_s = "0" and that the output of the ROM is subtracted when T_s = "1". To incorporate this functionally into the hardware, the control bit A/S is now the product of exclusive ORing the b_{0l} bit and the T_s bit together where previously the A/S bit was equal to the sign bit. An unfortunate byproduct of this modification though is the unintentional conversion of the addition of the partial product of the l^{th} bit for bits b_{1l} through $b_{(K-1)l}$ to a subtraction when b_{0l} = "1". To rectify this problem, the b_{1l} through $b_{(K-1)l}$ bits are exclusive ORed with b_{0l}. Now, the filter will generate the proper output using only half of the memory that was previously needed. Other than the K additional exclusive OR gates, the operation of the filter remains the same as before. A block diagram of the hardware implementation is shown in Figure 2.4.

The application of OBC to input driven DA for non-adaptive FIR filters is of tremendous benefit. This trade-off of reduced memory at the expense of slightly more hardware is worthwhile and is often taken. However for adaptive applications where the contents of the DA memory tables are changing, the benefit of OBC is greatly reduced because of the significant amount of hardware required to update the tables.

2.2 Review of Adaptive Distributed Arithmetic Filters

Although DA is well suited for non-adaptive filtering applications, DA is very difficult and burdensome to implement for adaptive purposes. The key issue with using DA for adaptive filters is how to efficiently update the contents of the memory tables, which are typically



ROM Address a_1	a_2	a_3	8-Word ROM Contents, Q
0	0	0	½(–h[0] – h[1] – h[2] – h[3]) = Q(0)
0	0	1	½(–h[0] – h[1] – h[2] + h[3]) = Q(1)
0	1	0	½(–h[0] – h[1] + h[2] – h[3]) = Q(2)
0	1	1	½(–h[0] – h[1] + h[2] + h[3]) = Q(3)
1	0	0	½(–h[0] + h[1] – h[2] – h[3]) = Q(4)
1	0	1	½(–h[0] + h[1] – h[2] + h[3]) = Q(5)
1	1	0	½(–h[0] + h[1] + h[2] – h[3]) = Q(6)
1	1	1	½(–h[0] + h[1] + h[2] + h[3]) = Q(7)

Figure 2.4. A block diagram of an input driven DA FIR filter using OBC with reduced memory usage.

loaded with the partial sums of the filter weights. These update methods for adaptive DA-based filters can be divided into two categories. The first is where the memory table is partially updated, and the second is where the memory table is fully updated. A brief literature review of implementations for both categories is given below.

2.2.1 Partially Updated DA Filters

Rather than updating each entry of a memory table, some adaptive DA filters only update a few entries per sample period [9-11]. The advantage of this approach is that it reduces the computational workload. However, this computational efficiency comes at the expense of convergence speed.

To begin the mathematical derivation of how to create a feasible partially updated distributed arithmetic adaptive filter, the i^{th} input sample, $x[n-i]$, can be expressed as a summation as shown in Eq. (2.15).

$$x[n-i] = \sum_{l=1}^{B} b_{(n-i)l} 2^{-l}, \quad i = 0, \ldots, K-1. \tag{2.15}$$

Although this notation is constant with the earlier one used in previous sections, the usage of vectors and matrices in place of summations will make mathematical manipulation in this case easier. The input sample $x[n-i]$ can be expressed as the product of two vectors as

$$x[n-i] = \mathbf{B}_{(n-i)} \cdot \mathbf{T}^T. \tag{2.16}$$

where $\mathbf{B}_{(n-i)} = \{b_{(n-i)1}\ b_{(n-i)2}\ \ldots\ b_{(n-i)l}\ \ldots\ b_{(n-i)B}\}$ and $\mathbf{T} = \{2^{-1}\ 2^{-2}\ \ldots\ 2^{-l}\ \ldots\ 2^{-B}\}$. $\mathbf{B}_{(n-i)}$ is the offset binary encoded bit-wise representation of the i^{th} input sample $x[n-i]$, and $\mathbf{T}$ is the binary scaling vector. The binary representation, $\mathbf{B}_{(n-i)}$, of the input sample $x[n-i]$ for $i = 0, 1, ..., K-1$ a form of offset binary coding where a "1" is equal to 1 and where a "0" is equal to -1. Note, this form of offset binary coding is different than the one used in Section 2.1.2.2 and in Section 3.2. For those other cases, a form of offset binary coding was used for a two's complement binary system and not for an unsigned magnitude one.

For convenience, the FIR filtering equation in summation form is repeated below as

$$y[n] = \sum_{i=0}^{K-1} w_i[n]x[n-i], \tag{2.17}$$

where $w_i[n]$ is the i^{th} filter weight and $x[n-i]$ is the i^{th} input sample. Eq. (2.17) can be vectorized into

$$y[n] = \mathbf{W}_n \cdot \mathbf{X}_n^T \tag{2.18}$$

where $\mathbf{W}_n = \{w_0[n]\ w_1[n]\ \ldots\ w_i[n]\ \ldots\ w_{K-1}[n]\}$ and $\mathbf{X}_n = \{x[n]\ x[n-1]\ \ldots\ x[n-i]\ \ldots\ x[n-(K-1)]\}$. $\mathbf{W}_n$ is the vector of filter weights, and $\mathbf{X}_n$ is the vector of input samples. By substituting Eq. (2.16) into Eq. (2.18), $\mathbf{X}_n$ is also equal to

$$\{\mathbf{B}_n\mathbf{T}^T\ \mathbf{B}_{(n-1)}\mathbf{T}^T\ \ldots\ \mathbf{B}_{(n-i)}\mathbf{T}^T\ \ldots\ \mathbf{B}_{(n-(K-1))}\mathbf{T}^T\},$$

which can be rewritten as $\mathbf{T} \cdot \{\mathbf{B}_n^T\ \mathbf{B}_{(n-1)}^T\ \ldots\ \mathbf{B}_{(n-i)}^T\ \ldots\ \mathbf{B}_{(n-(K-1))}^T\}$.

Now, a new matrix, $\mathbf{A}_n$, is defined as

$$\begin{aligned}\mathbf{A}_n &= \begin{bmatrix} \mathbf{B}_n^T & \mathbf{B}_{(n-1)}^T & \ldots & \mathbf{B}_{(n-i)}^T & \ldots & \mathbf{B}_{(n-(K-1))}^T \end{bmatrix} \\ &= \begin{bmatrix} b_{n1} & b_{(n-1)1} & & b_{(n-i)1} & & b_{(n-(K-1))1} \\ b_{n2} & b_{(n-1)2} & & b_{(n-i)2} & & b_{(n-(K-1))2} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ b_{nl} & b_{(n-1)l} & & b_{(n-i)l} & & b_{(n-(K-1))l} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{nB} & b_{(n-1)B} & & b_{(n-i)B} & & b_{(n-(K-1))B} \end{bmatrix}.\end{aligned}$$

and is called the partial product address matrix. This matrix is the concatenation of the bit-wise representation of the K input samples, $\mathbf{B}_{(n-i)}$ for $i = 0, 1, ..., K-1$, where the i^{th} column is the B-bit binary representation of $x[n-i]$ for $i = 0, 1, ..., K-1$. The l^{th} row in the matrix $\mathbf{A}_n$ represents the K-bit binary address used to find the l^{th} partial product in the distributed arithmetic memory table.

After defining this new matrix $\mathbf{A}_n$, the input sample vector $\mathbf{X}_n$ can be expressed as $\mathbf{T} \cdot \mathbf{A}_n$. By substituting this new expression for $\mathbf{X}_n$, Eq. (2.18) can be written as

$$y[n] = \mathbf{W}_n \cdot (\mathbf{T} \cdot \mathbf{A}_n)^T = \mathbf{W}_n \cdot \mathbf{A}_n^T \cdot \mathbf{T}^T. \tag{2.19}$$

Next, a new vector, $\mathbf{P}_n$, is defined as

$$\mathbf{P}_n = \mathbf{W}_n \cdot \mathbf{A}_n^T \tag{2.20}$$

and is denoted as the partial product vector. The l^{th} element in the vector $\mathbf{P}_n$ is the value of the l^{th} partial product of the distributed arithmetic computation.

By using this new vector $\mathbf{P}_n$ in Eq. (2.19), the distributed arithmetic equation for an FIR filter in matrix notation is

$$y[n] = \mathbf{P}_n \cdot \mathbf{T}^T. \tag{2.21}$$

The adaptive filter algorithm used in these partially updated DA filters is the least mean square (LMS) algorithm. In matrix notation, the LMS filter weight update equation is

$$\mathbf{W}_{n+1} = \mathbf{W}_n + 2\mu e \mathbf{X}_n. \tag{2.22}$$

For an adaptive algorithm, the updated partial product vector, $\mathbf{P}_{n+1}$, is equal to

$$\mathbf{P}_{n+1} = \mathbf{W}_{n+1} \cdot \mathbf{A}_n^T. \tag{2.23}$$

When updating the filter weights using the LMS algorithm and substituting Eq. (2.22) into Eq. (2.23), $\mathbf{P}_{n+1}$ becomes

$$\mathbf{P}_{n+1} = (\mathbf{W}_n + 2\mu e \mathbf{X}_n) \cdot \mathbf{A}_n^T. \tag{2.24}$$

where e is the signal error. Earlier, the input sample vector $\mathbf{X}_n$ was written as the product of the binary scaling vector $\mathbf{T}$ and of the partial product address matrix $\mathbf{A}_n$. Using this expression for $\mathbf{X}_n$, Eq. (2.24) can be written as

$$\mathbf{P}_{n+1} = (\mathbf{W}_n + 2\mu e \mathbf{T} \cdot \mathbf{A}_n) \cdot \mathbf{A}_n^T = \mathbf{W}_n \cdot \mathbf{A}_n^T + 2\mu e \mathbf{T} \cdot \mathbf{A}_n \cdot \mathbf{A}_n^T. \tag{2.25}$$

Noting the first product in Eq. (2.25) is equal to the updated partial product vector $\mathbf{P}_{n+1}$, Eq. (2.25) can be expressed as

$$\mathbf{P}_{n+1} = \mathbf{P}_n + 2\mu e \mathbf{T} \cdot \mathbf{A}_n \cdot \mathbf{A}_n^T. \tag{2.26}$$

Eq. (2.26) is the equation for updating the $(n+1)^{\text{th}}$ partial product vector using the previous n^{th} partial product vector plus the autocorrelation of the binary representation of the input samples $x[n-i]$ for $i = 0, 1, ..., K-1$.

Note, the matrix generated by the multiplication $\mathbf{A}_n \cdot \mathbf{A}_n^T$ is a $B \times B$ symmetrical matrix, and the diagonal elements of this matrix is equal to K. If the input signal is zero-mean, the successive input samples are uncorrelated, and the individual bits in the binary representation of each input sample $x[n-i]$ are uncorrelated, then the expectation of any non-diagonal element of $\mathbf{A}_n \cdot \mathbf{A}_n^T$ is equal to zero, and this product is simply the $B \times B$ identity matrix multiplied by K. Substituting $K \cdot \mathbf{I}_B$ for $\mathbf{A}_n \cdot \mathbf{A}_n^T$ in Eq. (2.26), the equation for updating the partial product vector becomes

$$\mathbf{P}_{n+1} = \mathbf{P}_n + 2\mu e\mathbf{T} \cdot K \cdot \mathbf{I}_B = \mathbf{P}_n + 2\mu eK\mathbf{T}. \tag{2.27}$$

If the product of $2\mu K$ is a power of two, then the update simply becomes the previous partial product vector plus the error shifted by a power of two times the binary scaling vector $\mathbf{T}$.

When Eq. (2.27) is taken literally, this update equation implies that only B partial products in the DA memory table need to be updated every sample period. The memory elements which are updated for the $(n+1)^{\text{th}}$ sample are determined by the partial product address matrix for the n^{th} sample. B additions are needed every sample period for a partially updated distributed arithmetic adaptive filter using a single memory table. However, adaptive filters using a partially updated memory table experience reductions in their convergence rate.

Notice for all the partial update methods proposed so far, the distributed arithmetic partial product address matrix is formed from the B-bit binary representation of the input samples and not from the B-bit binary representation of the filter coefficients. Methods like sliding-block distributed arithmetic, which will be described in Section 2.2.2.2, and the one proposed in Section 3.1 use a partial product address matrix composed of the binary bits of the filter coefficients. By using this approach, the problems of updating the DA memory table and updating the filter coefficients are broken into two separate ones.

2.2.2 Fully Updated DA Filters

Currently, two approaches exist to fully updating the memory tables in a distributed arithmetic filter. They are using the brute-force method and a form of DA called sliding-block distributed arithmetic. A summary of these type of adaptive distributed arithmetic methods are given below in Sections 2.2.2.1 and 2.2.2.2. The primary advantage of using a fully updated memory table over a partially updated one for an adaptive filter is the near-ideal convergence rate rather than a reduced one for a given adaptive algorithm.

2.2.2.1 Brute-Force Method

To implement an adaptive filter using a DA structure, the entries of the memory table, which contains all possible combination sums of the filter weights and are utilized for determining the required partial products, need to be recalculated and to be updated on a sample-by-sample basis to prevent a loss in the convergence rate. A brute-force implementation that updates each weight individually according to an adaptive filtering algorithm such as the one shown in Eq. (2.22), and then regenerates the memory table using the new weights, will be computationally expensive and time consuming, causing significant reduction in the filter throughput. To fully update a memory table using brute-force, this method would take $(K/2-1)2^K+1$ additions for a K-tap filter. When compared to a partially updating DA method, the brute-force method utilizes approximately $(K/B)2^{(K-1)}$ times more additions for a K-tap filter. For example a 16-tap filter using 8-bit input samples, the brute-force method uses approximately 128x more additions to fully update its memory table.

2.2.2.2 Sliding-Block Distributed Arithmetic

Sliding-block distributed arithmetic is a coefficient driven, rather than input driven [9, 11-29], mechanization of DA. Since the contents of the memory tables are changing over time, the key issue is developing a method that minimizes the number of additions needed to build those tables. In SBDA, the input data is collected in blocks and then the appropriate samples are windowed and convolved with the FIR filter coefficients as shown in Figure 2.5. This type of DA only changes the contents of one DA memory table every sample period.

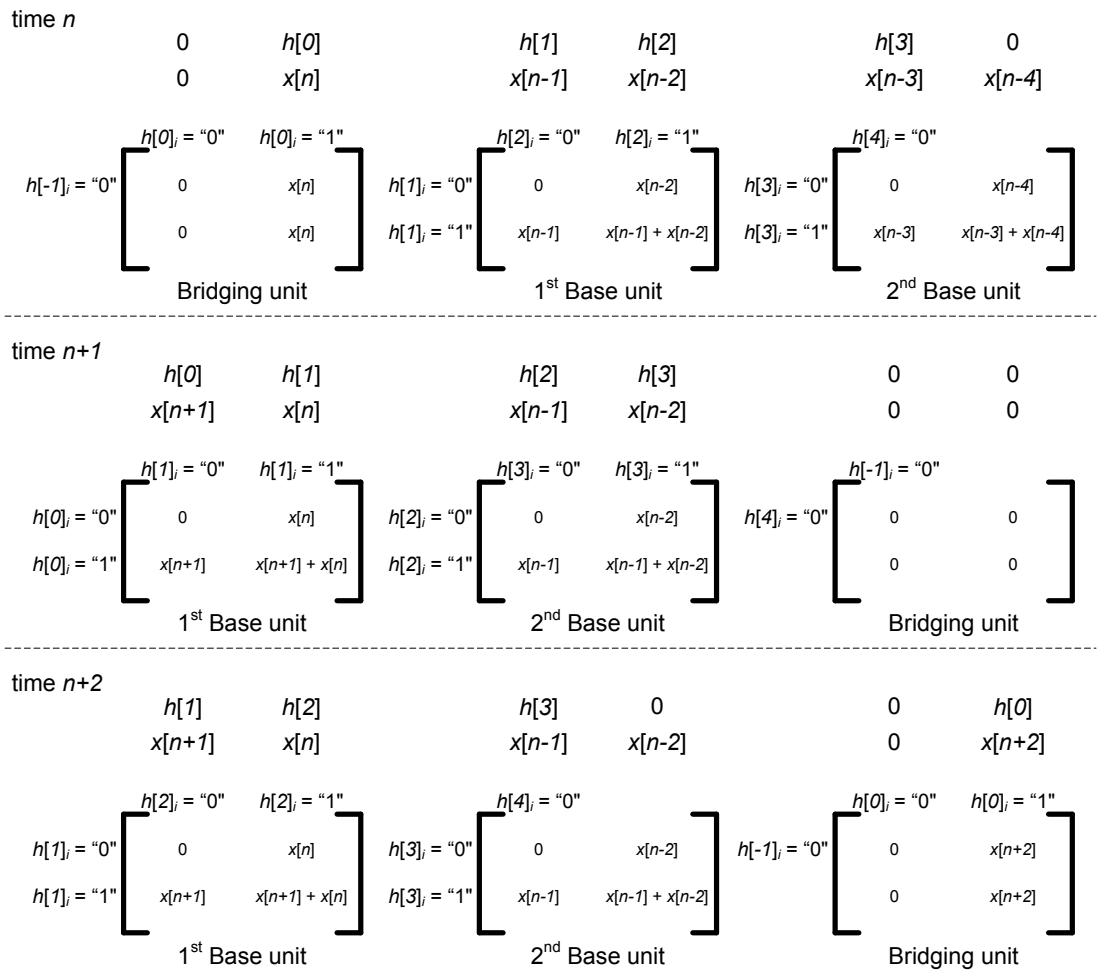


Figure 2.5. An illustration of SBDA in action, where $K = 4$, $m = 2$, and $k = 2$.

A summary of SBDA is given below.

The key issue in a coefficient driven DA mechanization is developing a method that minimizes the number of additions needed to build the DA memory tables. If the update were done using brute-force, then it would take $(k/2-1)2^k+1$ additions for a k-tap filter. In SBDA, an observation is noted that the contents of the memory change slowly over time. In other words, only the oldest input sample needs to be removed while the newest input sample needs to be added. Thus, only 2^{k-1} additions and 2^{k-1} subtractions for a total of 2^k computations are required for a k-tap filter. This results in a reduction in the number of operations necessary by about a factor of $k/2-1$ over brute-force. However, for SBDA to eliminate the subtractions, each DA memory table is associated with a fixed set of inputs. Because of this fact and assuming a K-tap filter is split into m sub-filters of k taps, $m+1$ base units are necessary. Every sample period, one of the base units is selected to update its outdated memory table with the k most recent input samples and is called the bridging unit.

The memory table being updated is constructed as follows. For simplification, the table is shown in a square matrix format. The assumption is that the oldest $k/2$ samples are associated with the $2^{k/2}$ columns while the $k/2$ most recent samples are associated with the $2^{k/2}$ rows. Each column is associated with a binary code , $c = \{c_{k/2}...c_0\}$, where c_0 is affiliated with the oldest sample and $c_{k/2}$ is affiliated with the $k/2$ oldest sample. On the other hand, another unique binary code, $r = \{r_{k/2}...r_0\}$, is associated with each row where r_0 is affiliated with the $k/2$ newest sample and $r_{k/2}$ is affiliated with the most recent sample. Now that the terminology has been described, the focus can return to how the DA memory table is generated. First, the table is initialized with all zeros. When the current sample arrives, it is added to the DA memory table as the oldest input sample since it was the first to come. In other words, this sample is added to any column where c_0 = "1". Then, the table is updated in the following order. First, the columns starting from c_1 to $c_{k/2}$ are updated. Subsequently, the rows are updated beginning from r_0 to $r_{k/2}$. With each

ensuing sample, they are added to their appropriate places in the table. After k samples, the DA memory table is now complete. As a simple example, a description of how the DA memory table is generated is considered for the 2-tap base unit case. At time n^-, the table is initialized to all zeros. When the sample $x[n]$ arrives, it is added to the rightmost column of the matrix. This column has a binary code of c = "1". At the next sample $x[n + 1]$, it is added to the bottommost row whose binary code is r = "1". Now, the current DA memory table is complete, and a new one begins to be generated at $n + 2$. Note in this example, the DA memory table with the oldest samples is shifted to the right. An illustration of this example is shown in Figure 2.6.

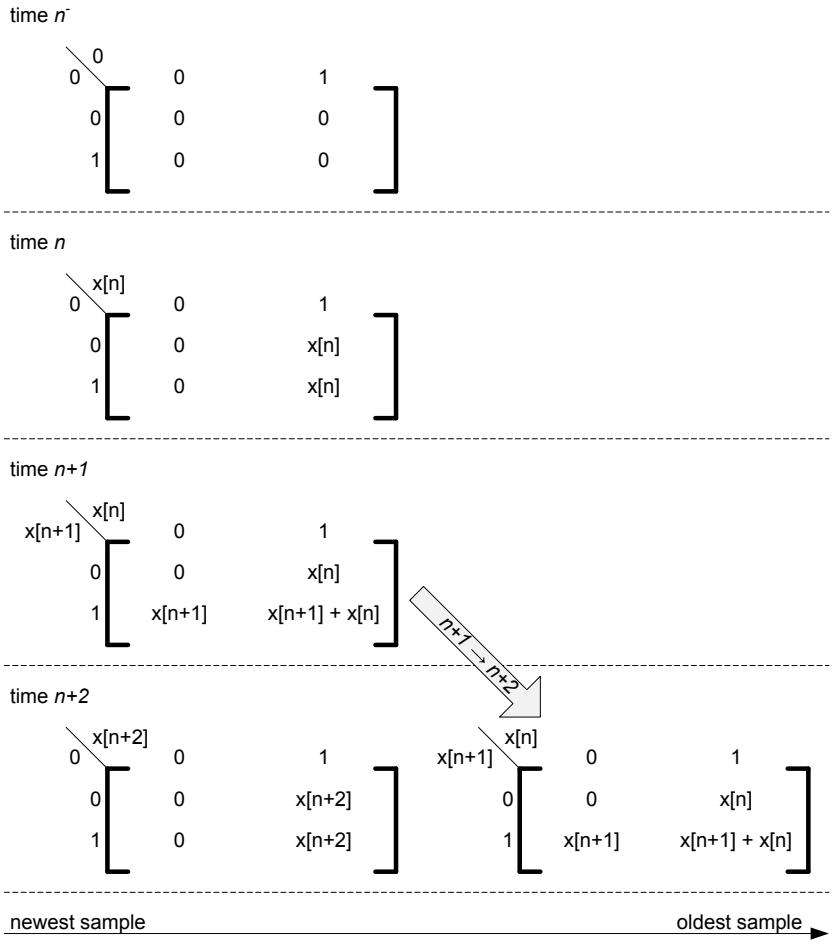


Figure 2.6. An illustration of the how the DA memory tables are generated over time in SBDA.

To account for the passage of time, the filter coefficients need to be aligned with the

proper data; therefore, they are passed from one base unit to the next as shown in Figure 2.7. The newest input sample needs to be aligned with the first filter coefficient, $h[0]$, and the

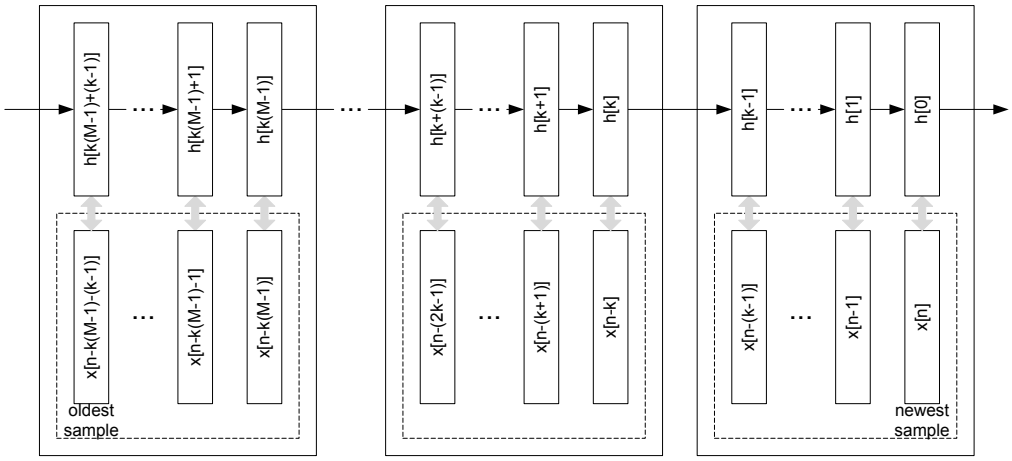


Figure 2.7. An illustration of the movement of the filter coefficients and input samples among multiple units.

oldest input sample needs to be aligned with the last filter coefficient, $h[k(m-1)+(k-1)]$. As time passes, the coefficients are passed to the right. In Figure 2.7, the data has been windowed and any data that occurs before and after what is shown is assumed to be zero. The shifting of the coefficients is necessary because the contents of the DA memory tables in the base units do not move as time passes.

Using the methods above for generating the DA memory table and for accounting of time, a description of how SBDA works is now provided. To assist in the explanation, an illustration of SBDA in action, where $K = 4$, $m = 2$, and $k = 2$, is shown in Figure 2.5. SBDA is filtering the input samples $x[n]$ through $x[n-3]$. However, the newest sample is not contained in any of the previous two base units, and the memory content of those units is fixed; therefore, a new DA memory table in a new base unit must be created. Note, that $x[n-4]$ in the second base unit is not used and the coefficient associated with it is zero. Now, the filter coefficients span the bridging unit and the two base units. At next cycle, $n+1$, the newest sample, $x[n+1]$, is added to the DA memory table. Since the generation of the new table is now complete, the bridging filter becomes the first base filter

and the previous first base filter the second base unit. The coefficients slide over to the left only to cover the two base units. This shift accounts for the elapse of one sample. Since the filter does not need any data from the previous second base unit, it becomes the new bridging unit and its memory contents are initialized to zero. At time $n+2$, the cycle begins again. Note at every sample, the coefficients are shifted to the left and when it gets to the beginning of the queue, it is shifted back to end of the queue like a circular buffer.

CHAPTER 3

NEW TECHNIQUES FOR ADAPTIVE DISTRIBUTED ARITHMETIC FILTERS

A couple of new techniques are proposed in this chapter to address the potential issues regarding the usage of distributed arithmetic in adaptive filters. These issues are the difficulty of updating the contents of the memory table and the relatively high memory usage. In the first part of this chapter, a new memory table update method is proposed in Section 3.1. Compared to the typical approach previously used to mitigating the memory table update issue, the newly proposed method is able to address this issue without introducing new ones. To address the memory usage issue, a previously known type of adaptive distributed arithmetic is combined with a memory reduction technique. In addition to reducing the size of the memory table, the proposed combination was modified to further reduce the computational workload for updating the memory table. The details of this modified combination is provided below in Section 3.2

3.1 New Memory Table Update Method

A new hardware architecture using conjugate distributed arithmetic (CDA) which is suitable for high throughput hardware implementations of LMS adaptive filters is presented. Unlike a traditional distributed arithmetic (DA) implementation where all possible combination sums of the filter coefficients are stored in a memory table, in the CDA architecture, all possible combination sums of the input signal samples are stored in the memory table and updated at the arrival of every sample using an efficient update procedure. In the CDA formulation, the filter weights are represented as B-bit 2's complement binary numbers,

$$w_i = -b_{i0} + \sum_{l=1}^{B-1} b_{il} 2^{-l}, \quad i = 0, \ldots, K-1, \tag{3.1}$$

where b_{il} is the l^{th} bit in the 2's complement representation of w_i. Substituting Eq. (3.1) into Eq. (2.1) and swapping the order of the summations yields

$$y[n] = -\left[\sum_{i=0}^{K-1} b_{i0}x[n-i]\right] + \sum_{l=1}^{B-1}\left[\sum_{i=0}^{K-1} b_{il}x[n-i]\right]2^{-l}. \tag{3.2}$$

Now, the memory table has all possible combination sums of the input signal samples $\{x[n], x[n-1], \ldots, x[n-K+1]\}$.

For fixed filters, the standard DA architecture is far more efficient than the CDA architecture since the memory table does not have to be updated every sample instance. On the other hand for an adaptive filter, the CDA architecture is superior for reasons described below. In an adaptive filter, *e.g.* LMS, the weights w_i are updated according to

$$w_i[n+1] = w_i[n] + \mu e[n]x[n-i] \tag{3.3}$$

where μ is the step size and $e[n]$ is error between the filtered output and a desired target signal. With the traditional DA architecture, the entire memory table will have to be recomputed using the updated set of weights. In the case of the CDA architecture, since the weights are individually stored (not as all possible combination sums in a 2^K element memory table), their update can be easily performed. The update of the memory table containing all possible combination sums of the input signal samples will also have to be performed; however, this can be done far more efficiently than recomputing the entire memory table as in the traditional DA implementation by using the procedure described in [30] for updating the DA-A-MEM. The details of this update procedure are provided in Section 3.1.1.

In this implementation, the term $\mu e[n]$ is quantized to one of L values, each selected to be some power of 2. This enables us to minimize the on-chip area usage by replacing the hardware multiplier by a simple barrel shifter. In other words, the product of the contents of the DA-A-MEM[n] with $\mu e[n]$ is approximated by a right shift of the contents of the DA-A-MEM[n]. It must be noted that while such an approximation does not affect the throughput, it causes a marginal degradation in the convergence of the distributed arithmetic adaptive filter (DAAF). In Figure 3.1, a MATLAB simulation comparison for the convergence of

the following two LMS implementations is provided: (i) the actual product $\mu e[n]\times$ DA-A-MEM$[n]$ is calculated and (ii) the term $\mu e[n]$ is quantized to one of $L = 8$ values (each value is a power of 2) and thus the product is implemented by a single shift operation. In both cases, a white Gaussian random noise signal with zero mean and unit variance was used as the input and the desired signal, $d[n]$, was generated by filtering the input with a 256-tap low-pass FIR filter. The $e[n]$'s used in the plot in Figure 3.1 for both the above mentioned cases are obtained by averaging 150 independent trials of the respective MATLAB simulations.

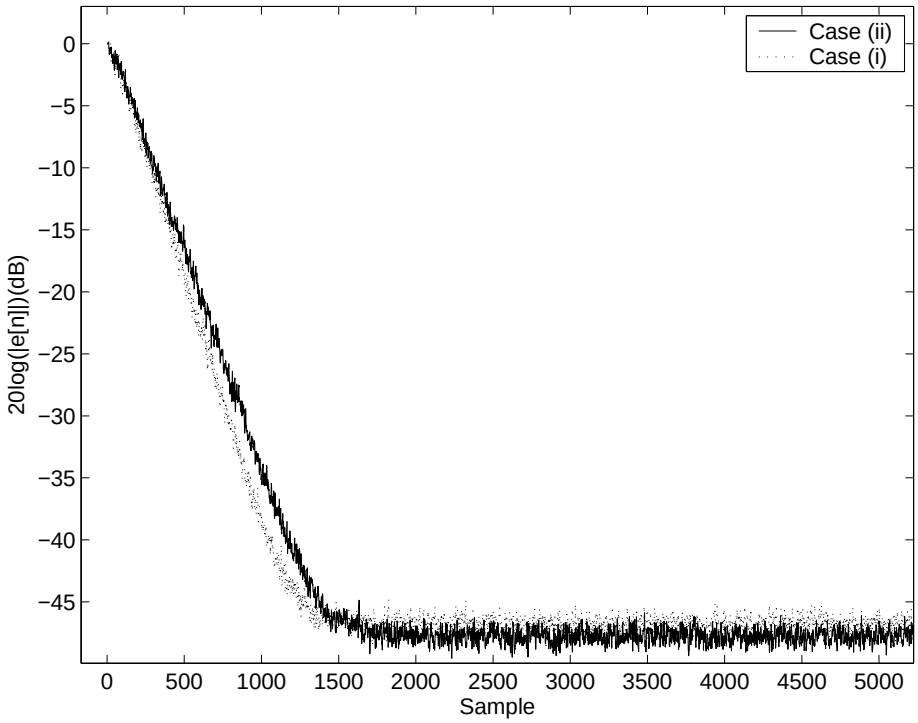


Figure 3.1. MATLAB simulation of convergence for LMS implementations Case (i) Actual $\mu e[n]\times$ DA-A-MEM$[n]$ is used and Case (ii) $\mu e[n]$ is quantized to one of $L = 8$ values (powers of 2). The $e[n]$'s used in the plot for both the above mentioned cases are obtained by averaging 150 independent trials of the respective MATLAB simulations.

3.1.1 Updating the Conjugate DA Memory Table

Let the state of the memory table at a sample time instance n be denoted MEM$[n]$. Figure 3.2 shows the update of the MEM$[n]$ from MEM$[n-1]$ for $K = 4$. It may be observed

that the contents of the even addressed locations (locations whose addresses have a 0 in the LSB) of the MEM[n] are the contents of the lower half (locations whose addresses have a 0 in the MSB) of the MEM[$n-1$]. Also, the contents of the odd addressed locations (locations whose addresses have a 1 in the LSB) of the MEM[n] can be obtained from the even addressed locations of the MEM[n] according to

$$\mathrm{MEM}_{(2l+1)}[n] = \mathrm{MEM}_{(2l)}[n] + x[n], l = 0, \ldots, 2^{K-1} - 1. \tag{3.4}$$

The update of the MEM[n] from MEM[$n-1$] can be summarized by the following two steps:

Step 1: The lower half of the MEM[$n-1$] is re-mapped to even addressed locations of the MEM[n] as shown by the arrows in Figure 3.2. Instead of physically moving the contents of the memory table, this re-mapping operation can be performed by a simple left-rotation of the K address lines of the memory table. Address rotation allows the physical contents of memory to remain the same, even as the external logic sees the table as re-mapped. It can be observed that the external address referring to a given physical address at the time n is the left-rotated version of the external address referring to the same physical address at the time $n-1$. Therefore, the effect of address rotation can be accomplished by connecting the external and the internal addresses via K, K-to-1 input multiplexers. The $\log_2(K)$ select lines of each of the K multiplexers are connected to the $\log_2(K)$ bits of a counter, which is incremented with the sample clock. Thus, by address rotation, the entire mapping of the memory table can be done instantaneously at the arrival of the new sample $x[n]$.

The address rotation is controlled by the DA filter update control module.

Step 2: It must be noted that the address rotation maps the upper half of the MEM[$n-1$], containing sums involving the oldest sample $x[n-4]$, to the odd addressed locations at time n. The entries in these odd addressed locations of the MEM[n] are overwritten by values obtained according to Eq. (3.4). In other words, the contents of the odd addressed locations of the MEM[n] are obtained by reading the contents of the corresponding preceding even

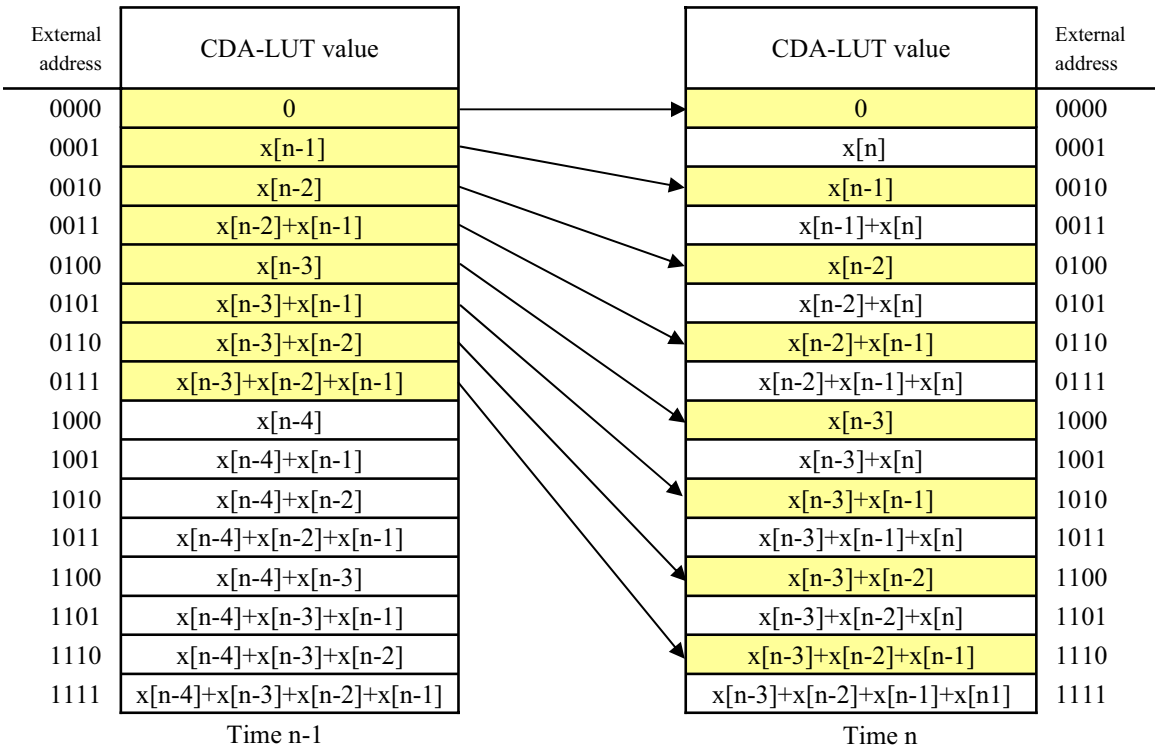


Figure 3.2. Update of the MEM[n] from MEM[$n-1$].

addressed locations, adding the newest sample $x[n]$ and then storing the result back at the odd addressed locations. The filter weights $w_i, i = 0, 1, \ldots, K-1$, may be updated according to Eq. (3.3) in parallel with the update of the LUT.

3.1.2 Architectures for large filter taps

In any DA based system as the filter order increases, the memory requirements of the implementation grow exponentially. For example, a 128-tap DA FIR filter will require a prohibitively large 2^{128} entries in the memory table. This problem may be alleviated by breaking up the filter into smaller base DA filtering units that require tractable memory sizes and then summing up the outputs of these units [4]. A K-tap filter may be divided into m smaller filters each having k-tap DA base units ($K = m \times k$).

A K-tap CDA-based adaptive filter may also be split similarly in a practical implementation [30]. The m units operate in cascade, with the oldest sample of the l^{th} unit passed on to the $l+1^{\text{th}}$ unit at the next sample instance. With this decomposition, the total memory

requirement is reduced from 2^K to $m \times 2^k$ words.

3.1.3 Performance Results

In this section, the performance of the proposed CDA-based adaptive filtering architecture is compared versus the traditional MAC-based architecture. Both architectures were implemented on an Altera Stratix FPGA clocked at 50MHz.

3.1.3.1 Throughput

The throughput is defined as the number of signal samples processed by an adaptive filter per second. If t is the number of clock cycles required for filtering and updating the filter weights according to the adaptation algorithm, then

$$\text{Throughput} = \frac{\text{clock rate}}{t}. \tag{3.5}$$

The filtering operation uses B clock cycles. For a K-tap CDA adaptive FIR filter implemented in m sub units, each operating on k samples, the update of the memory table can be done in 2^{k-1} clock cycles as described in Section 3.1.1. The weights can be updated at the same time the memory table is being updated. Since the update of the weights usually does not take longer than updating the memory table, the total number of clock cycles for filtering, updating the memory table, and updating the weights is $B + 2^{k-1}$. Finally, the adder tree uses $\lceil \log_2(m) \rceil$ clock cycles. Thus, the overall K-tap adaptive filter utilizes $B + 2^{k-1} + \lceil log_2(m) \rceil$ clock cycles. In other words, the throughput for the proposed DA-adaptive filter is given by

$$\text{Throughput} = \frac{\text{clock rate}}{B + 2^{k-1} + \lceil \log_2(m) \rceil}. \tag{3.6}$$

The throughput of the proposed CDA adaptive filter is compared against a MAC-based adaptive filter. A plot of this comparison is shown in Figure 3.3. As shown in our previous work [30], the throughput of a DA based FIR filter does not vary much as the filter order is increased. In contrast, the throughput of the MAC-based approach is reduced by more than 1.5 orders of magnitude when the filter order grows from 16 taps to 1024 taps. For example

when the filter order is 16 taps, the throughput of a 4 MAC-based adaptive filter is equal to the throughput of a CDA adaptive filter with $k = 4$. But for a filter order of 1024 taps, a CDA adaptive filter with $k = 4$ has a throughput 48 times greater than a 4 MAC-based adaptive filter.

Despite an efficient method for updating the memory table, using a base filter order greater than 4 has severe consequences on the throughput. For example, the throughput for a CDA adaptive filter with $k = 8$ is at least five times lower than a CDA adaptive filter with $k = 2$. On the other hand, the throughput for a CDA adaptive filter with $k = 4$ is at most 1.25 times lower than a CDA adaptive filter with $k = 2$.

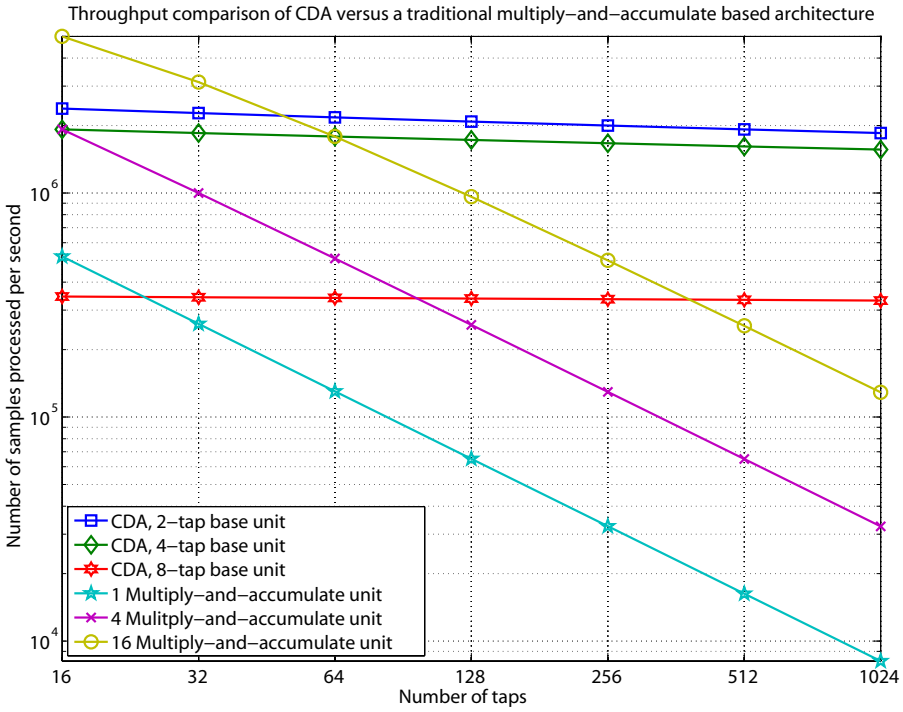


Figure 3.3. Throughput comparison of CDA versus a traditional multiply-and-accumulate based architecture.

3.1.3.2 Number of Logic Elements

In programmable logic systems, the size of the logic design is measured by the number of logic elements (LEs). For Altera's Stratix architecture, the LE is the smallest unit for implementing logic functions. Each LE uses a four-input memory table, a programmable

register and carry chain with carry select capability. Further details of LE can be found in [31]. Ten LEs are grouped into one logic array block (LAB) and the LABs are interconnected through a row and column-based network. For this analysis, the number of LEs, instead of the number of LABs, is considered as a metric for area usage since it has more detailed information about the actual chip area used for the implementation.

In terms of the number of logic elements used to synthesis a particular configuration, a CDA adaptive filter with $k = 8$ has the fewest logic elements for a given filter order than any CDA adaptive filter with $k < 8$ as illustrated in Figure 3.4. For a CDA adaptive filter of a certain filter order, the logic element usage decreases as k increases. For reference, the number of logic elements used in a 1, 4, and 16-MAC based adaptive filter is also shown in Figure 3.4.

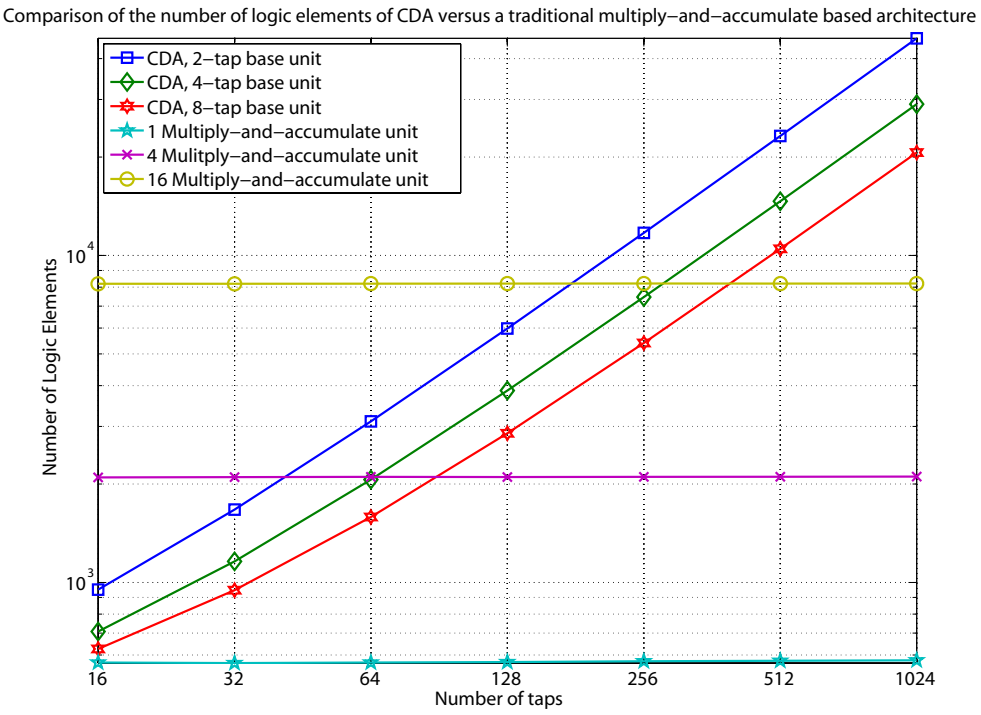


Figure 3.4. Comparison of the number of logic elements of CDA versus a traditional multiply-and-accumulate based architecture.

3.1.3.3 Logic Element Efficiency

In many situations, the amount of die area available for a certain function is limited so knowing how many logic elements are used is vital. But as transistor sizes shrink, the number of transistors that can be placed in a fixed die area increases and designers must come up with new ways of utilizing the additional transistors in an effective way. To measure the effective usage of these transistors, a metric of how efficiently each logic element is being used for the computation of the inner product is created. This metric is calculated by dividing the throughput by the logic element usage. Higher numbers means that the logic elements are being utilized for beneficial calculation rather than being idle and performing no computations.

Figure 3.5 is a plot that allows a designer to compare how efficiently each approach utilizes its logic elements. From the figure, the most logic element efficient design is CDA with $k = 4$ followed by CDA with $k = 2$. All other designs have approximately the same logic element efficiency and are about 2.5 times less efficient than CDA $k = 2$ and about 3.3 times less efficient than CDA $k = 4$. In other words, any MAC-based adaptive filter for a given filter order would need at least 3.3 times more logic elements to achieve the same throughput as a CDA adaptive filter with $k = 4$. Not surprisingly, the logic element efficiency for a MAC-based adaptive filter does not really change with the number of MACs because most of the logic elements used in a MAC-based adaptive filter is utilized for its MACs and the throughput for a MAC-based adaptive filter decreases linearly as more MACs are used.

3.1.3.4 Memory

Altera's Stratix architecture has three types of RAM blocks consisting of M512, M4K, and M-RAM blocks [31]. For each RAM block type, the FPGA synthesis software provided the number of blocks utilized in the design. In addition, these reports presented the memory usage in KB. For this analysis, measuring the amount of memory reported in KB is the appropriate metric especially when comparing the memory requirement with other methods

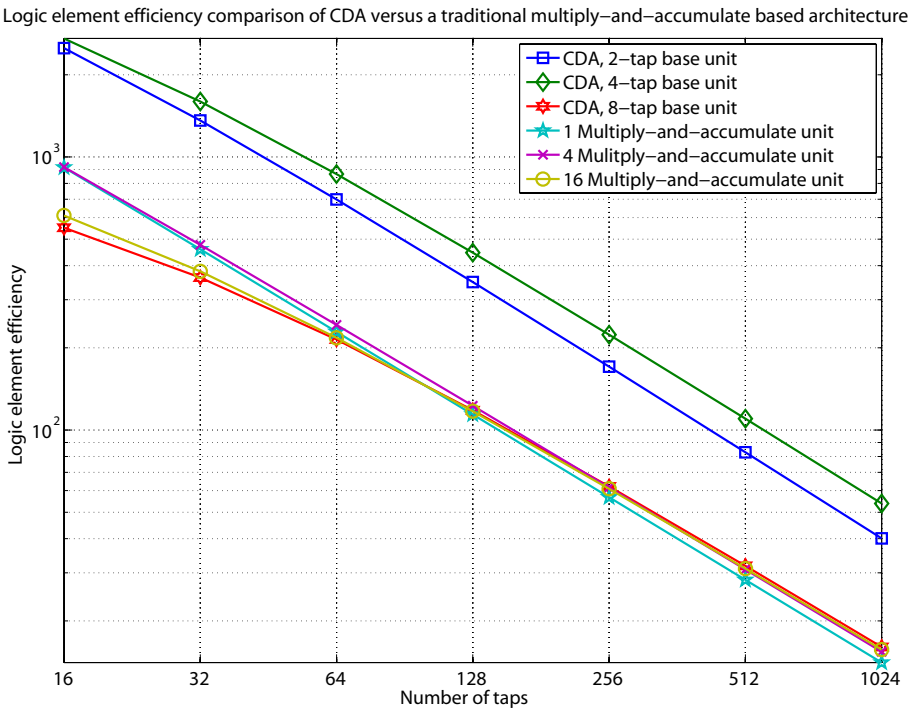


Figure 3.5. Logic element efficiency comparison of CDA versus a traditional multiply-and-accumulate based architecture.

such as DSP microprocessor implementations.

For the MAC-based adaptive filter, the amount of memory used did not vary with the number of MACs but is only dependent on the filter order as illustrated in Figure 3.6. These filters used the least amount of memory. For a given CDA adaptive filter order, the memory usage increased with k. Compared to the MAC-based adaptive filter, a CDA adaptive filter with $k = 2$, $k = 4$, and $k = 8$ used approximately 1.5, 2.5, and 16.6 times the memory used by a MAC-based one for a certain filter order. Regardless of the hardware implementation, the memory usage increased linearly with filter order.

3.1.3.5 Power Consumption Estimates

The power consumption estimates are obtained using the PowerPlay Power Analyzer tool of Altera's Quartus II software. A reference on this tool is located at [32]. For an accurate estimate, a simulation vector was created and utilized as the input to the analyzer. This step occurred after the place and route for maximum accuracy. By using a simulation vector, its

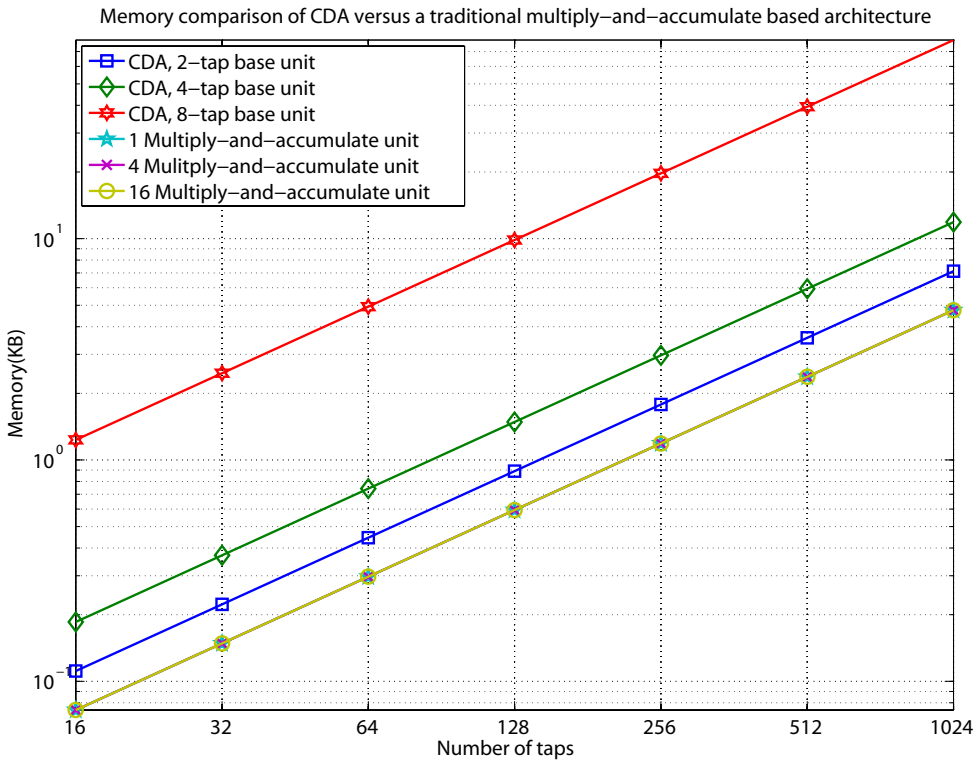


Figure 3.6. Memory comparison of CDA versus a traditional multiply-and-accumulate based architecture.

usage will reflect actual design behavior. As has been pointed out by [33-35], an accurate power estimate must filter out the glitches in the simulation vector so this setting in the PowerPlay Power Analyzer was activated. With these settings, an accurate power estimate, usually within 10 percent [36], can be generated [36, 37].

The dynamic thermal power for both CDA and MAC-based adaptive filters are shown in Figure 3.7. To easily compare power values for different CDA and MAC-based adaptive filter configurations, the dynamic thermal power was measured when the throughput for each configuration is the same. A CDA adaptive filter with $k = 4$ consumed less power than a CDA adaptive filter with $k = 2$ or $k = 8$ and consumed about the same amount of power as a 16 MAC-based adaptive filter. Although a CDA adaptive filter with $k = 4$ and a 16 MAC-based adaptive filter used approximately the same amount of power, the CDA architecture is able to achieve a higher throughput, if necessary, than the MAC-based architecture.

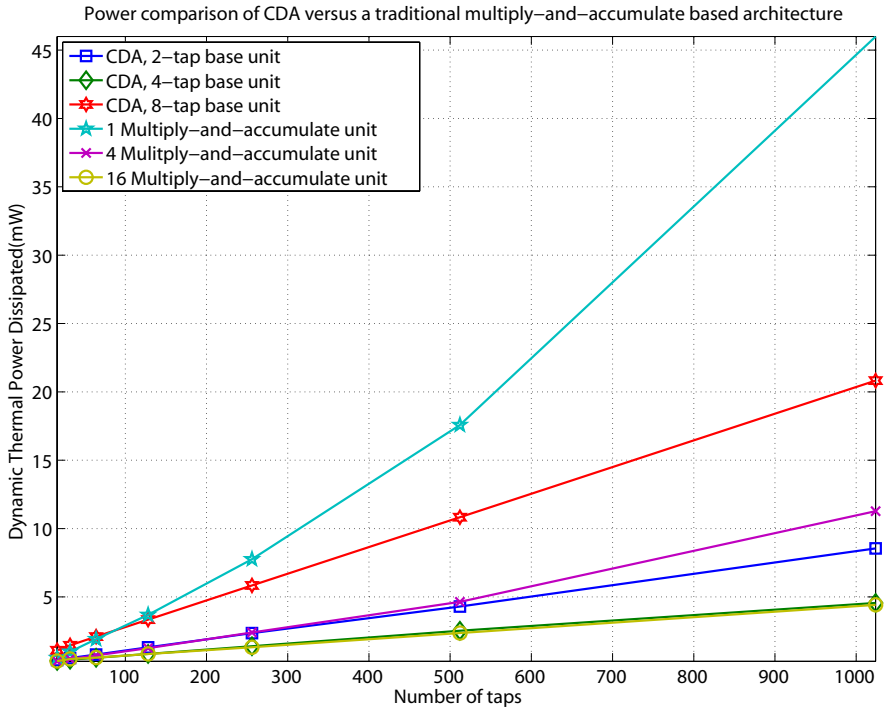


Figure 3.7. Power comparison of CDA versus a traditional multiply-and-accumulate based architecture.

3.1.4 Comparing CDA to SBDA

Although SBDA and CDA can be used for non-adaptive FIR filters, they were both purposefully created for adaptive applications. They both reduce the number of additions needed to update their DA memory tables, and they both provide easy access for modifying their filter coefficients unlike input driven DA. To demonstrate the ease of implementing an adaptive filter, both SBDA and CDA provide descriptions for mechanizations of the LMS algorithm.

The main distinction between SBDA and CDA is the manner in which the input data is partitioned. In CDA, the data is partitioned into individual samples; hence, every DA memory table must be updated every sample period. Only the data that is necessary for filtering is stored in the tables. On the other hand, for SBDA, the data is partitioned into blocks of k samples; therefore, the data must be windowed before it is filtered. By rotating the coefficients from one block to the next, only the base unit whose DA memory table contains the oldest samples needs to be updated. This base unit is designated as the bridging unit. Since it is the only component that is updating the content of its DA memory table, no extra additions are expended by the other base units. However since the bridging unit introduces an additional partial product that must be added in the DA computation, B_h, which is the bit precision of the filter coefficients, additional additions are required. In contrast for CDA, every base unit is recomputing the contents of their tables every sample period, which requires one addition per updated entry. In terms of memory usage, CDA uses less than SBDA because SBDA uses an extra table for its bridging unit. Also like SBDA, CDA is able to eliminate the need for subtractions when updating its memory contents. In other words, the oldest samples in the DA memory tables are removed through other means. However, this task is accomplished by a different method in CDA. It is achieved by remapping the DA memory table through address rotation. It only updates the necessary table entries by using the content already present plus the newest sample.

From a purely computational point of view such as executing the mechanizations on a microprocessor in sequential order, the following equations state the number of additions

and the amount of memory needed for SBDA and CDA.

$$SBDA_{additions} = B_h\left(\lfloor K/k \rfloor + 1\right) + \left(2^{k-1} - 1\right) \tag{3.7}$$

$$CDA_{additions} = B_h\left(\lfloor K/k \rfloor\right) + \left(\lfloor K/k \rfloor\right) \cdot \left(2^{k-1} - 1\right) \tag{3.8}$$

$$SBDA_{memory} = 2^k\left(\lfloor K/k \rfloor + 1\right) \tag{3.9}$$

$$CDA_{memory} = 2^k\left(\lfloor K/k \rfloor\right) \tag{3.10}$$

where B_h is the bit precision of the filter coefficients, K is the filter length, and k is the size of the base unit. CDA uses $\frac{1}{\lfloor K/k \rfloor + 1}$ less memory than SBDA, and this advantage is significant when $\lfloor K/k \rfloor$ is small as illustrated in Figure 3.8. However, when $\lfloor K/k \rfloor$ is large, this advantage is offset by a significant increase in the number of additions necessary for CDA over SBDA as shown in Figures 3.9 and 3.10, and is of minimal benefit as shown in Figure 3.8. A plot of the comparative advantage of CDA over SBDA when k is held constant as $\lfloor K/k \rfloor$ is swept for different values of B_h is shown in Figure 3.9. A plot of the comparative advantage of CDA over SBDA when B_h is held constant as $\lfloor K/k \rfloor$ is swept for different values of k is illustrated in Figure 3.10. When the comparative advantage is positive, CDA uses less adders than SBDA. When the comparative advantage is negative, SBDA uses less adders than CDA. Interestingly, CDA uses less adders than SBDA when $\frac{B_h}{\left(\lfloor K/k \rfloor - 1\right) \cdot \left(2^{k-1} - 1\right)} > 1$.

3.2 Encoding the Memory Tables using Offset Binary Coding

As shown in Section 2.1.2.2, the application of offset binary coding (OBC) to the memory table of a non-adaptive distributed arithmetic FIR filter is relatively straightforward. Unfortunately for a typical adaptive input-driven DA filter, the use of OBC just further complicates the difficult task of updating the memory table. However for sliding-block distributed arithmetic, the application of offset binary coding to its memory table becomes a reasonable combination.

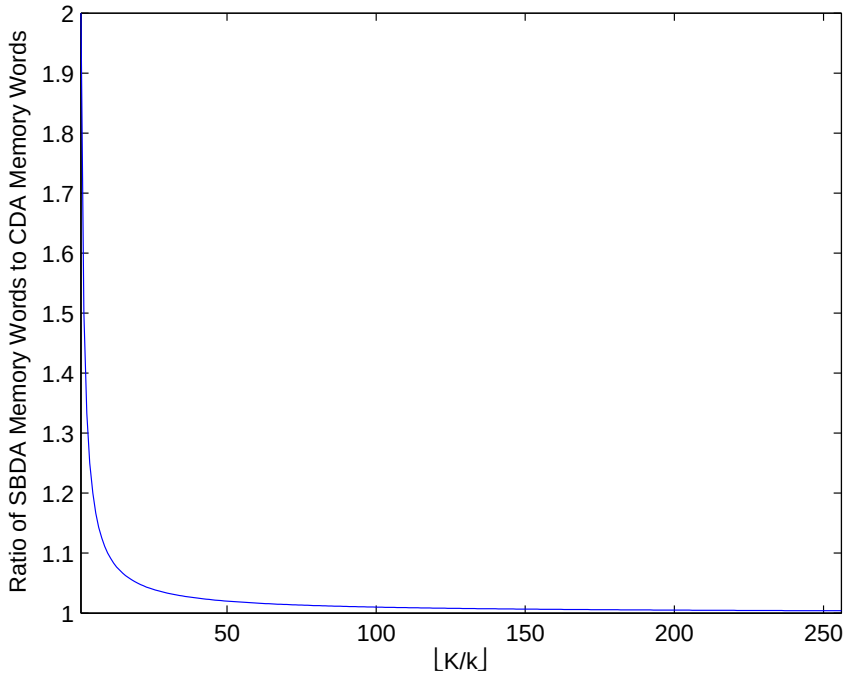


Figure 3.8. A plot of the $\frac{SBDA_{memory}}{CDA_{memory}}$ versus $\lfloor K/k \rfloor$.

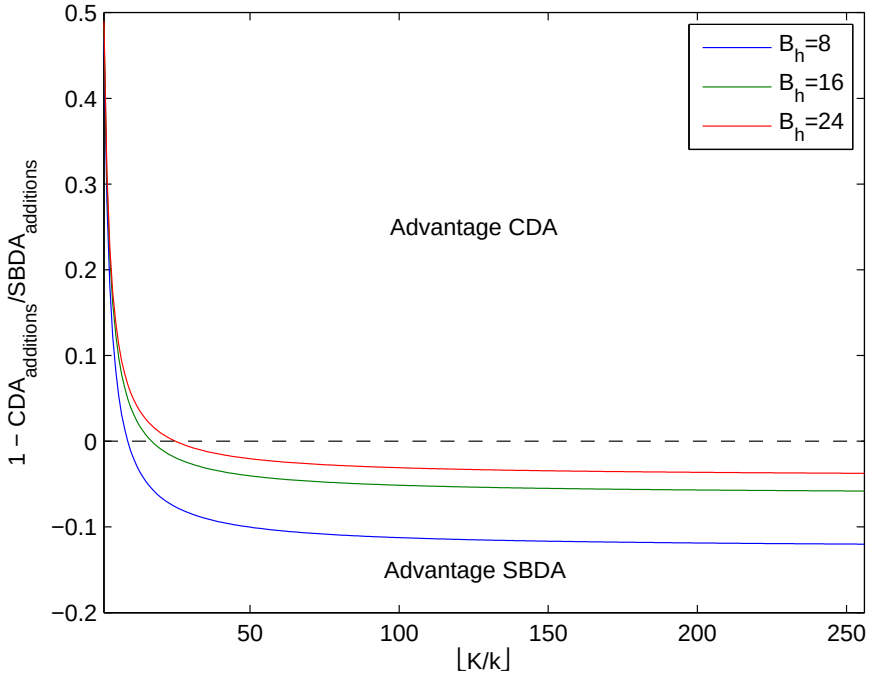


Figure 3.9. A plot of the $1 - \frac{CDA_{additions}}{SBDA_{additions}}$ versus $\lfloor K/k \rfloor$ for varying B_h when $k = 2$.

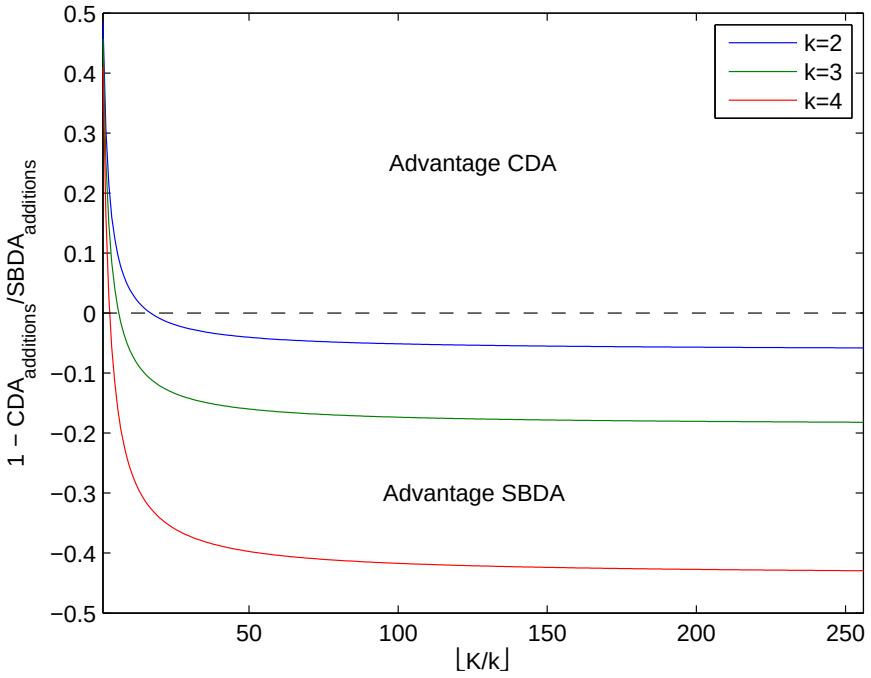


Figure 3.10. A plot of the $1 - \frac{CDA_{additions}}{SBDA_{additions}}$ versus $\lfloor K/k \rfloor$ for varying k when $B_h = 16$.

3.2.1 Combining Sliding-Block Distributed Arithmetic with Offset Binary Coding

By applying OBC to SBDA, the size of the memory tables can be reduced in half. However, this straight-forward combination of SBDA with OBC only yields a memory benefit. Still, an arithmetic operation either an addition or a subtraction is required to update each entry in the memory table being updated [1]. For a memory table not encoded using OBC, SBDA uses the same number of operations because only half of the memory table, which is twice the size of the one used when it is encoded in OBC, needs to be updated. Therefore, the number of computations needed is equal to the amount used in SBDA without OBC.

Recall that in SBDA when a DA memory table is about to be updated, it is initialized to zero. To generate the tables in OBC format, half of the most current input, $x[n]$, needs to be added or subtracted from every entry based on the bit stream for the current sample, b_{0l}. When b_{0l} = "0", $0.5x[n]$ is subtracted, and when b_{0l} = "1", $0.5x[n]$ is added. As an alternative, a block of the k most current samples can be collected and used to compute

the initial condition, $Q(0)$, so that the DA memory tables are initialized to it. With the exception of the first step where no update is required, each subsequent update only needs to add the current sample to the entries where b_{0l} = "1", which maps only to half of the table. This modification of SBDA is called SBDA-OBC.

A comparison of how the computational flow for updating a DA memory table encoded using OBC for a 3-tap filter differs between SBDA and SBDA-OBC is shown in Figure 3.11. For both SBDA and SBDA-OBC, three steps plus the initialization of the memory contents are required to completely update a memory table. One step is taken every sample period, and does not affect the filtering computation. For SBDA, eight additions are needed, and for SBDA-OBC, six additions are required. In the first step for SBDA, no additions are needed because the current sample plus zero is equal to the current sample. For each subsequent step, four additions are used. In the first step for SBDA-OBC, no additions are needed because no update is necessary since the memory table is already initialized to the correct values. For each subsequent step, it uses two additions at every step. Plus, two additions for computing $Q(0)$, which is necessary to update the memory table. In this case, SBDA-OBC uses 25% fewer additions than SBDA.

3.2.2 Differences in Implementation between SBDA and SBDA-OBC

As a point of reference, a description of a plausible SBDA implementation is given. The overall block diagram for such an implementation is shown in Figure 3.12. This implementation is composed of M SBDA processing units, a memory table update control unit, an adder tree, and a DA backend.

The DA backend is composed of an adder/subtractor, a multiplexer, and a couple data storage elements. This backend along with the adder tree is used to maximize resource sharing among the SBDA processing units and to eliminate the redundant hardware. This single backend plus the adder tree is used to replace the DA backends found in non-optimized SBDA processing units, and can be found in other DA implementations [30]. The purpose of this backend is to accumulate/decimate the output of the adder tree with the previously

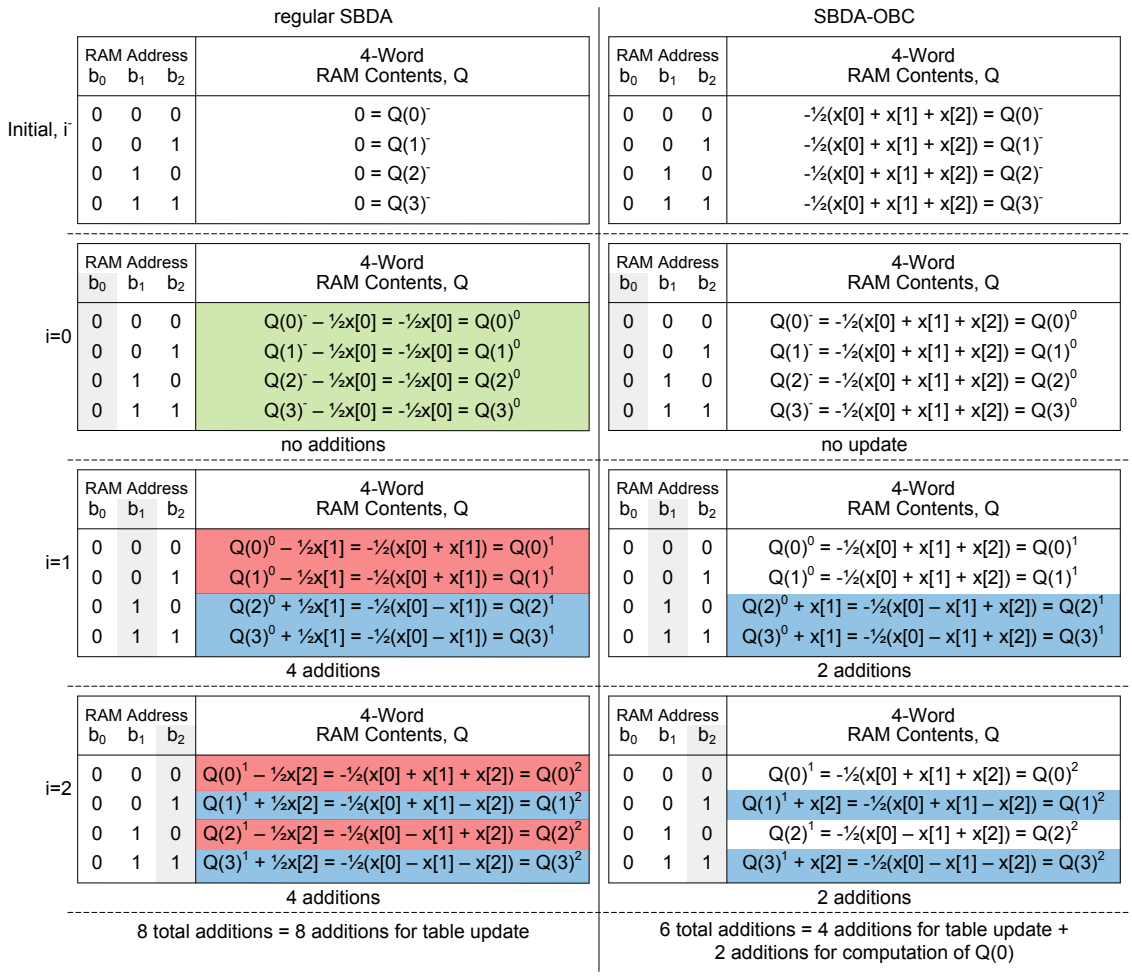


Figure 3.11. A comparison of how SBDA and SBDA-OBC generates a DA memory table encoded in OBC.

scaled partial products, to capture and to scale these intermediate accumulated values by a factor of one-half, and to capture the output y[n]. Note, the output of the adder tree is decimated from the accumulated value when processing data associated with the most significant bit, $l = 0$. This event usually occurs on the last computational cycle. Also note, the previously accumulated value is set to zero when beginning the DA computation. Normally, this event occurs when data associated with the least significant bit, $l = B - 1$, is being processed.

The primary function of the memory table update control unit is to generate the address sequence required to update a memory table, and to determine which memory table among

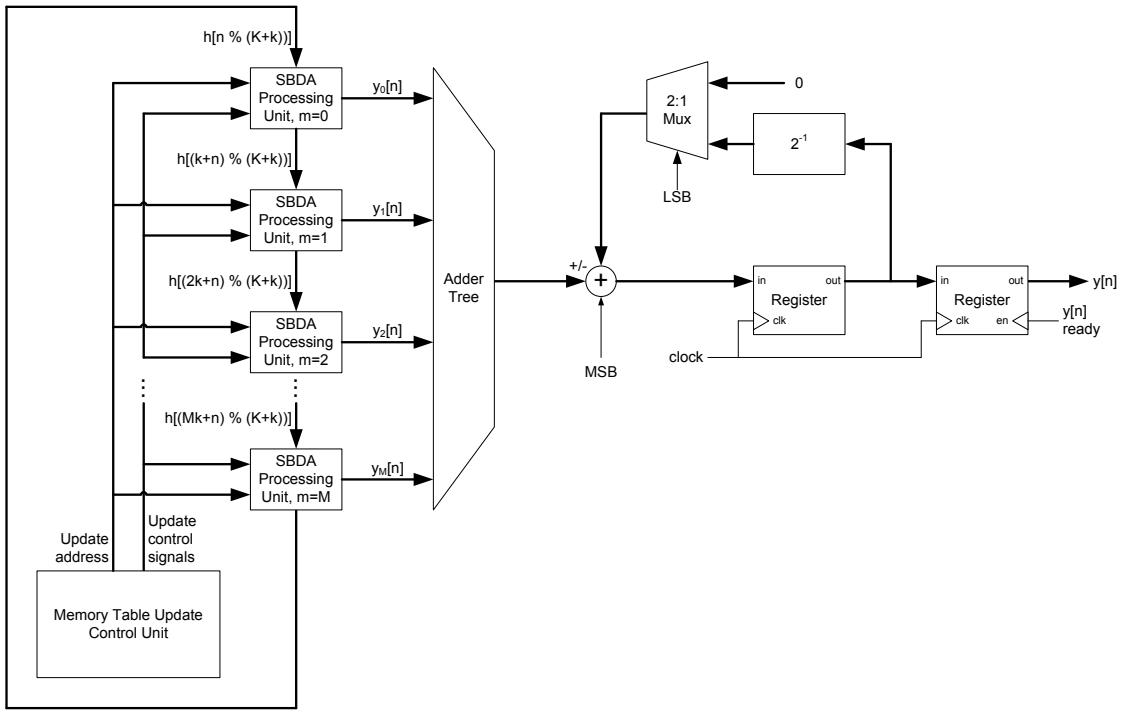


Figure 3.12. Overall block diagram of an SBDA implementation.

the M processing units needs their memory updated. Recall, only one memory table needs to be updated each sample period when using SBDA unlike other adaptive DA filter implementations. This unit updates the memory table with the oldest input samples, and over k sample periods refreshes the contents of the entire table. This process is described in more detail in Section 2.2.2.2 and in [1].

The M SBDA processing units are where the distributed arithmetic memory tables resides and where the filter weight updates and the DA filtering address generation are performed. A block diagram of this unit is shown in Figure 3.13. This unit is composed of a memory table, two multiplexers, an adder, and a weight update and address generation unit. The memory table is used to store all possible partial products. It has two address sources. One called the updating address is used when the contents of its memory table is being refreshed with the newest samples. This only occurs when its memory table contents become outdated. It takes k samples to refresh the table. During the first refresh cycle,

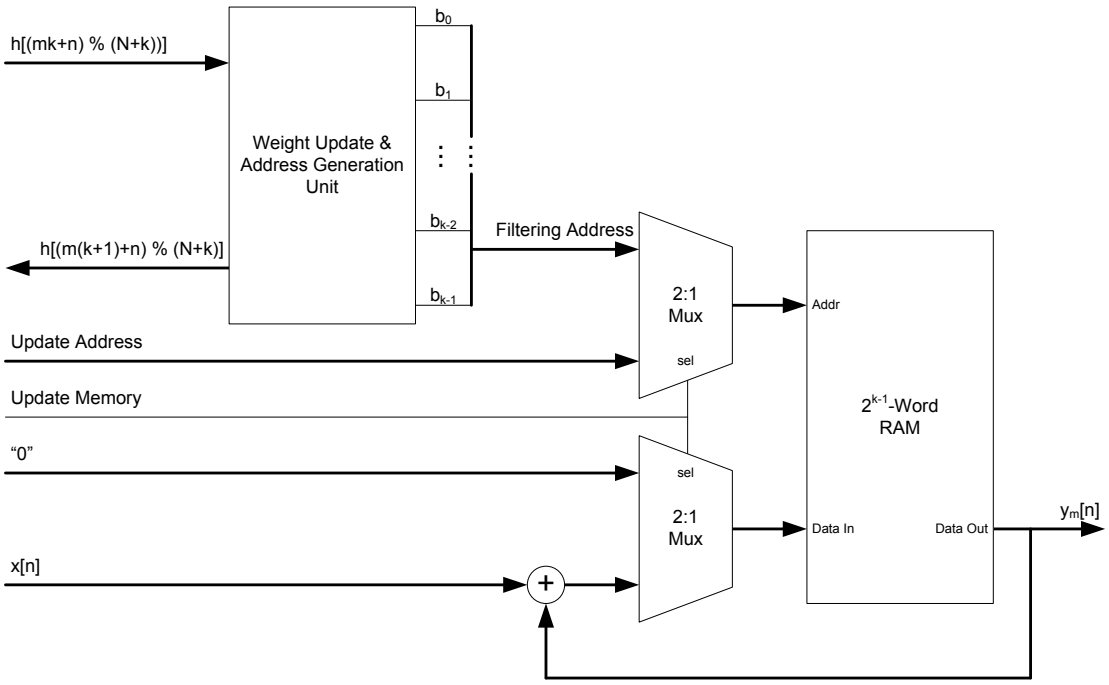


Figure 3.13. Block diagram of the SBDA processing unit.

which lasts one sample period, the entire table is initialized to zero. For the next $k-1$ refresh cycles, which last $k-1$ sample periods, the most recent input sample, $x[n]$, is inserted into the memory table using the contents stored at that address plus the input sample. All control such as the address sequence for updating the memory table every refresh cycle is provided by the memory table update control unit. However, the memory table can be used for filtering during this refresh period. Once the contents are fully updated, they are valid for another $K-k$ samples.

The weight update and address generation unit is where the filter coefficients are updated and where the filtering addresses are generated using those updated filter weights. For many adaptive algorithms, a bit-serial approach to updating the filter coefficients and to generating the filtering addresses can be used. An approach like this can significantly reduce the hardware needed. More detail about such an approach can be found in [1].

Now that a baseline SBDA implementation has been established, a plausible SBDA-OBC implementation can be described and contrasted with this baseline. An overall block

diagram of a plausible SBDA-OBC implementation is given in Figure 3.14, and a block diagram of its processing unit is given in Figure 3.15. Several key differences are identified

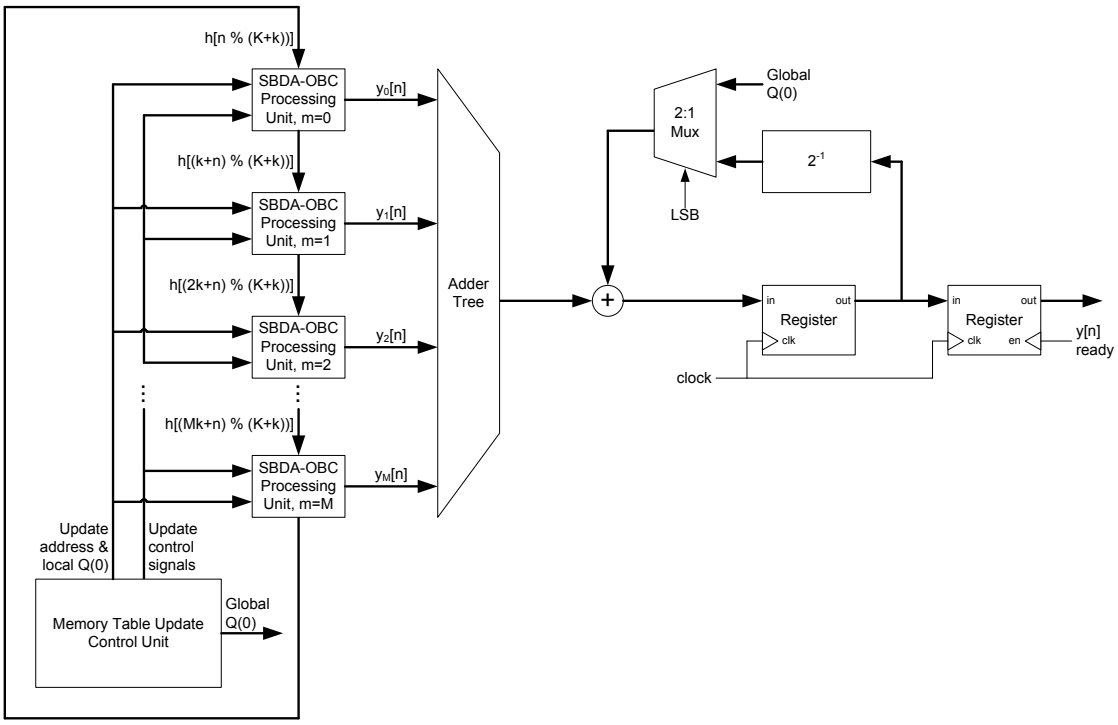


Figure 3.14. Overall block diagram of an SBDA-OBC implementation.

when comparing these figures with those for SBDA. First, a couple of these differences are found at the system level. For one, the memory table update control unit is somewhat modified. Now, it is also responsible for generating the local initial condition $Q(0)_{local}$ and the global initial condition $Q(0)_{global}$.

Before, a local initial condition was not necessary because the update memory table was initialized with zero; therefore, this value did not need to be passed into the processing unit. Also in SBDA, the memory tables are not encoded using OBC; hence, an initial condition is not required, while in SBDA-OBC, the memory tables are encoded using OBC. The local initial condition is computed by accumulating the k most recent input samples. After k sample periods, the accumulator is reset to zero, and the accumulation begins anew the next sample period. A new local initial condition is needed every k samples. As mentioned

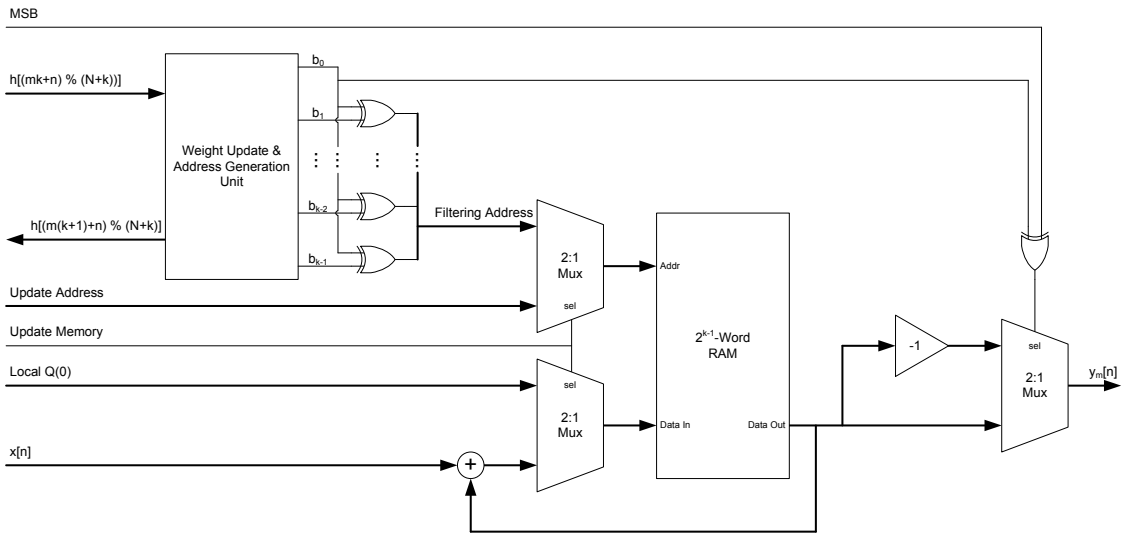


Figure 3.15. Block diagram of the SBDA-OBC processing unit.

earlier, the start of the computation needs to be delayed by k sample periods, since this value is needed to initialize the memory table. However, this delay only increases the latency of the system by k samples and does not decrease the throughput.

In addition to the local initial condition, a global initial condition needs to be calculated. This value is needed at the beginning of the DA computation, usually during the processing of the data associated with the least significant bit. The global initial condition is the summation of all the local initial conditions used in either the M or the $M + 1$ memory tables. Unlike the local initial condition, the global initial condition is computed using a moving averager of either M or $M + 1$ local initial conditions. The number of local initial conditions used is dependent on if the K-tap filter uses all $M + 1$ processing units or just M processing units. Like the local initial condition, a new global initial condition is computed every k samples. This global initial condition is necessary because of the common DA backend. If this DA backend was contained within each SBDA-OBC processing unit, then a global initial condition would not be needed. However, if the global initial condition is removed in this fashion, then the hardware for the overall system significantly increases, approximately M times additional DA backends. In light of this consequence, any hardware

associated with computing and with using a global initial condition is worthwhile.

At the processing unit level, a couple differentiations are identified. The most obvious are the exclusive OR gates used at the output of the weight update and address generation unit. These exclusive OR gates are needed to compress a k-bit address into a $k-1$-bit address as used for memory tables encoded using OBC. Remember, the size of an OBC encoded memory table is half the size of a regular one; hence, the necessity of compressing the address bits is required.

Another key difference is the need to be able to select an inverted output $-y_m[n]$ or a non-inverted output $y_m[n]$. This ability is necessary to implement any DA filter using offset binary coding. This selection must be done within the processing unit because this value is dependent on the local address bit b_0 and not some global variable. Therefore, this function cannot be incorporated into the DA backend on the system level.

3.2.3 Comparison of SBDA and SBDA-OBC

A logical starting point for making a comparison of SBDA and SBDA-OBC is to create formulas for certain critical measurement criteria. The measurements of importance are the data memory usage and the computational workload required for the computation of one sample. Since in the case of SBDA and SBDA-OBC, the only type of computation of significance that is used is addition. The computational workload is measured as the number of additions needed for both updating the memory table and filtering the data. These four formulas are listed below.

$$SBDA_{additions} = B_h\left(\lfloor K/k \rfloor + 1\right) + \left(2^{k-1} - 1\right) \tag{3.11}$$

$$SBDA - OBC_{additions} = B_h\left(\lfloor K/k \rfloor + 1\right) + \left(2^{k-2} + 1\right) \tag{3.12}$$

$$SBDA_{memory} = 2^k\left(\lfloor K/k \rfloor + 1\right) \tag{3.13}$$

$$SBDA - OBC_{memory} = \left(2^{k-1} + 1\right) \cdot \left(\lfloor K/k \rfloor + 1\right) \tag{3.14}$$

where B_h is the bit precision of the filter coefficients, K is the filter length, and k is the size of the sub-filter. A table of the computational workload and of the data memory usage for

different K when $B_h = 16$ and when k is selected such that the computational workload is minimized is given below in Table 3.1. If there are multiple configurations that minimize the computational workload, then the configuration with the lowest memory usage is reported.

Table 3.1. Computational Workload and Data Memory Usage for Various Filter Configurations when $B_h = 16$

	$k_{optimial}$*		# of Additions			# of Memory Words		
K	SBDA-OBC	SBDA	SBDA-OBC	SBDA-OBC**	SBDA	SBDA-OBC	SBDA-OBC**	SBDA
16	6	5	65	73	79	99	68	128
32	6	5	113	121	127	198	119	224
64	6	6	193	193	207	363	363	704
128	7	7	337	337	367	1235	1235	2432
256	8	7	593	625	655	4257	2405	4736
512	9	8	1041	1105	1167	14649	8385	16640
1024	10	9	1905	1953	2079	52839	29298	58368

* $k_{optimal}$ is defined as the k that minimizes the computational workload.
** The value provided is for the SBDA-OBC filter configuration when the size of the sub-filter is set to the $k_{optimal}$ for SBDA.

A plot of the relative advantage of SBDA-OBC over SBDA in terms of the number of additions needed for the computation of one sample versus the number of sub-filters when $B_h = 8, 16, 24$ and $k = 8$ is shown in Figure 3.16. From this figure, it is observed that SBDA-OBC is most beneficial when the bit precision of the filter coefficients is low and the filter is grouped into few sub-filters. This advantage diminishes as the number of additions needed for updating becomes numerically insignificant to the number of additions needed for filtering when either B_h is large, the filter is split into many sub-filters (i.e. $\lfloor K/k \rfloor$ is large), or a combination of both.

Figure 3.17 is a plot of the relative advantage of SBDA-OBC over SBDA in terms of the number of additions needed for the computation of one sample when k is varied and $B_h = 16$. It is observed that SBDA-OBC is most beneficial when the input samples are grouped into large blocks and the filter is grouped into few sub-filters. By increasing k, the ratio of computations used for updating over computations used for filtering increases; hence, the benefit of using SBDA-OBC over SBDA is significant.

An interesting observation is that SBDA-OBC is not beneficial when $k = 2$. In this case,

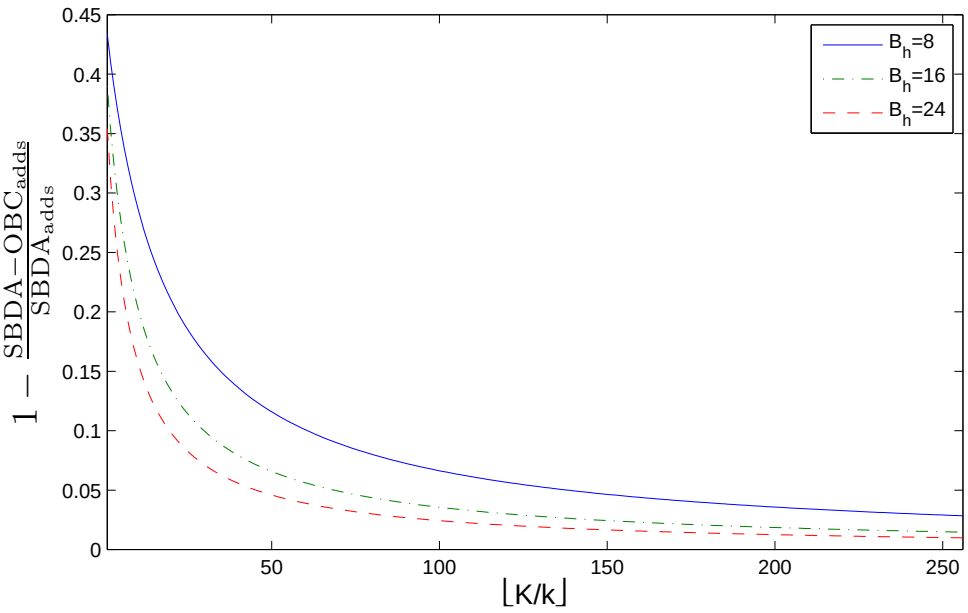


Figure 3.16. A plot of the $1 - \frac{SBDA-OBC_{additions}}{SBDA_{additions}}$ versus $\lfloor K/k \rfloor$ for varying B_h when $k = 8$.

the additional overhead associated with generating the initial condition, which is essential in the computational reduction of updating the memory table in other cases, in SBDA-OBC is significant. Specifically, in this case, SBDA requires one addition to update its memory table. However, SBDA-OBC requires two additions, one to update a memory table entry and another to compute the initial condition.

A plot of the relative advantage of SBDA-OBC over SBDA in terms of the memory usage when k is varied is shown in Figure 3.18. Note, this advantage is not dependent on K or on B_h. From the figure, it is observed that SBDA-OBC is most beneficial when the input samples are grouped into large blocks. Although SBDA-OBC still has a 25% advantage over SBDA when k is small, this advantage is diminished because of the slight memory overhead associated with SBDA-OBC. However, as the size of the memory tables increase exponentially, this overhead quickly becomes insignificant, and the advantage peaks at about 50%. This occurs when $k \approx 15$.

Since one of the primary focuses of this research is the reduction of the computational workload, it would be useful just to focus on only the updating portion. In the following

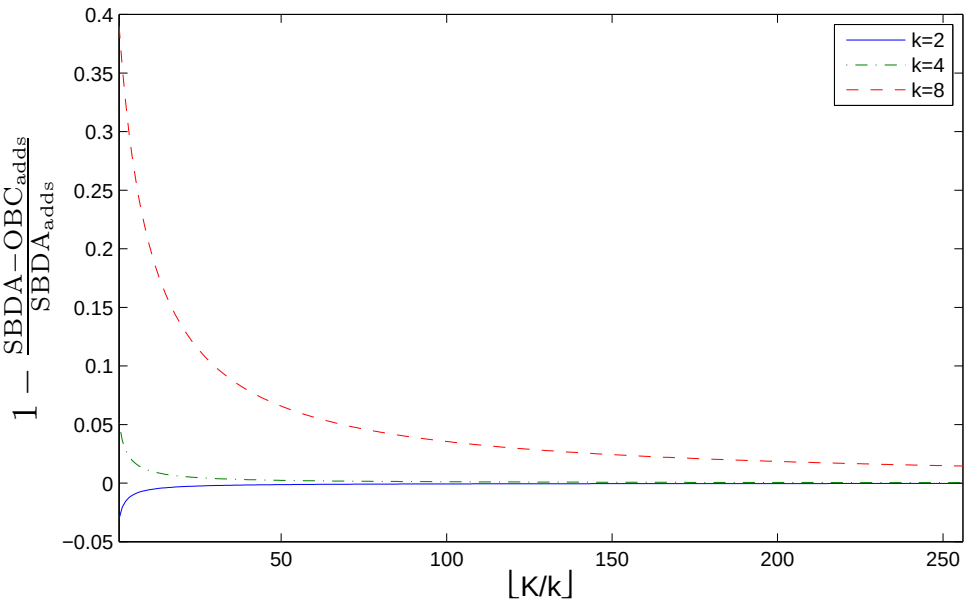


Figure 3.17. A plot of the $1-\frac{SBDA-OBC_{additions}}{SBDA_{additions}}$ versus $\lfloor K/k \rfloor$ for varying k when $B_h = 16$.

two equations, the number of additions required to update the DA memory table over k samples for SBDA and SBDA-OBC are given below. These equations are for a k-tap sub-filter. Eq. 3.16 has two terms because the first one, $(k-1)2^{k-2}$, is for updating the table and the second, $(k-1)$, is for computation of the initial condition.

$$SBDA_{adds,updating}(k) = (k-1)(2^{k-1}-1) \tag{3.15}$$

$$SBDA-OBC_{adds,updating}(k) = (k-1)2^{k-2} + (k-1) \tag{3.16}$$

Figure 3.19 is a plot of the relative advantage of SBDA-OBC over SBDA in terms of the number of additions needed for only updating the memory table when k is varied. This advantage is not dependent on K or B_h. It is observed that SBDA-OBC is most beneficial when the input samples are grouped into large blocks and only provides a benefit when $k > 3$. This figure reaffirms the observation made about Figure 3.17 when $k = 2$.

3.3 Summary

When computational resources are limited in particular multipliers, distributed arithmetic (DA) is used in lieu of the typical multiplier-based filtering structures. This advantage in

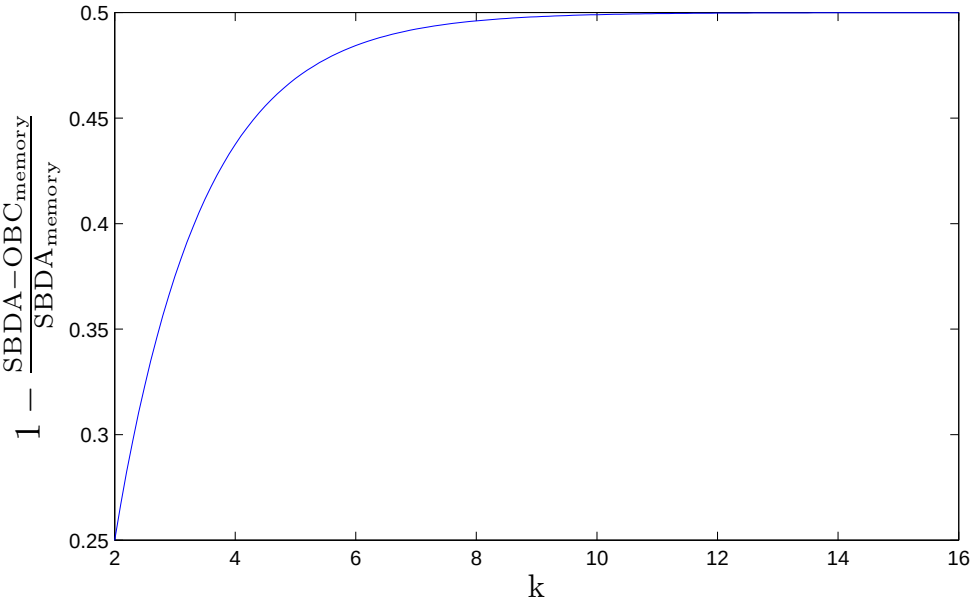


Figure 3.18. A plot of the $1-\frac{SBDA_{memory}}{SBDA-OBC_{memory}}$ versus k.

terms of computational resources is further extended when the bit precision of the filter is high. However due to the construction of the memory tables used for distributed arithmetic, it is not well suited for adaptive applications. The bottleneck when using DA for an adaptive filter is updating the memory table. Several attempts have been done to accelerate the process of updating the memory. Although these approaches do reduce the amount of processing necessary to update the memory, this reduction is gained at the expense of additional memory usage and of convergence speed. A more desirable solution is to develop a new approach for updating the memory table efficiently without using additional memory resources and compromising the convergence rate. In this thesis, such an approach is proposed and developed.

To develop an adaptive distributed arithmetic filter with a convergence rate that is not compromised, the memory table must be fully updated, and to realize that for an adaptive distributed arithmetic filter the memory table does not have to be composed of the combinations of the filter coefficients and be addressed by concatenating bits of the input data as in a traditional distributed arithmetic filtering structure. The memory table instead can

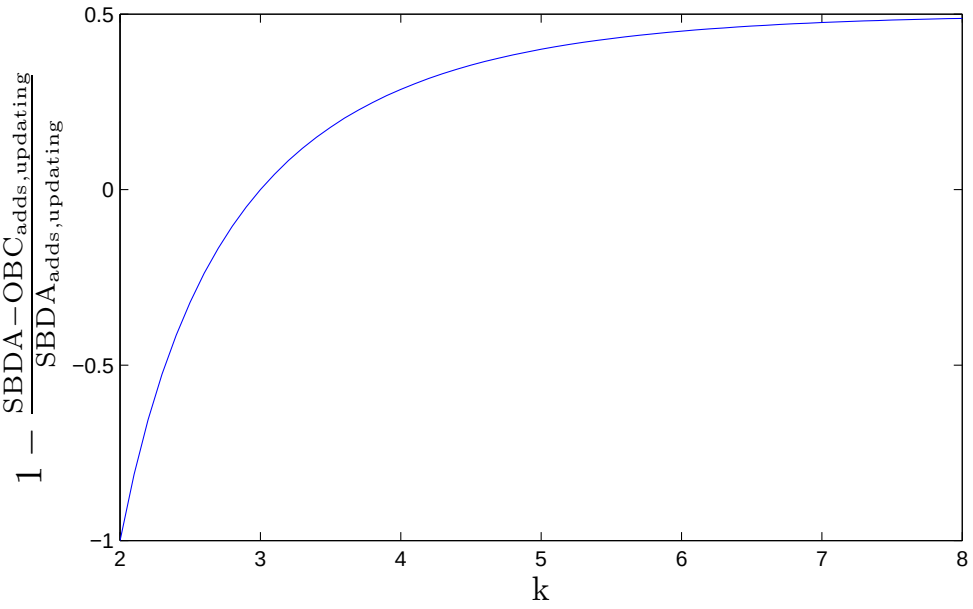


Figure 3.19. A plot of the $1 - \frac{SBDA-OBC_{adds,updating}}{SBDA_{adds,updating}}$ **versus** k**.**

composed of the combinations of the input samples and can be addressed by concatenating bits of the filter coefficients.

For an adaptive filter, the contents of the memory table must be updated regardless if the memory table is based on the filter coefficients or based on the input samples. In this research, an efficient method for fully updating a memory table that is composed of the combinations of the input samples and that is addressed by concatenating bits of the filter coefficients was proposed and developed. The proposed update method is based on exploiting the temporal locality of the stored data, which is a combination of the input samples that is determined by its memory address, and subexpression sharing. The proposed update method is possible because the data set, which is composed of the input samples, that the memory table is constructed from changes slowly. In other words, only the newest input sample needs to be added to the data set, and the oldest input sample needs to be removed from the data set. The proposed update method reduced the computational workload for updating the memory table by about $k/2-1$ over brute-force and required no additional memory resources. This type of distributed arithmetic that uses the proposed update

method is called conjugate distributed arithmetic (CDA). This type of update method is not applicable to a memory table composed of the combinations of the filter coefficients and addressed by concatenating bits of the input samples because the entire data set, which is composed of the filter coefficients, changes from sample to sample because of the adaptive nature of the filter.

The performance of CDA was compared against a traditional multiplier based approach and against sliding-block distributed arithmetic (SBDA), which is the only other adaptive distributed arithmetic approach that addresses the same issues as CDA. When CDA is compared against a traditional multiplier based approach, the following conclusions were deduced. The throughput of CDA remained relatively constant as the filter order increased while the throughput of a traditional multiplier based approach decreased quadratically with increasing filter order. Also, a CDA adaptive filter can use up to 3.3 times fewer logic elements than a traditional multiplier based adaptive filter to achieve a certain throughput and uses the same amount of memory as a non-adaptive distributed arithmetic FIR filter.

When CDA is compared against SBDA, the following conclusions were deduced. CDA uses $\frac{1}{\lfloor K/k \rfloor + 1}$ less memory than SBDA, and this advantage is significant when $\lfloor K/k \rfloor$ is small. However, when $\lfloor K/k \rfloor$ is large, this advantage is offset by an increase in the number of additions necessary for CDA over SBDA. Although in some situations, CDA uses less adders than SBDA when $\frac{B_h}{(\lfloor K/k \rfloor - 1) \cdot \left(2^{k-1} - 1\right)} > 1$.

In addition to developing a new type of adaptive distributed arithmetic called CDA, the only other type of adaptive distributed arithmetic that addressed the same issues as CDA called SBDA was modified to reduce the memory usage and to reduce the computational workload. SBDA was modified to encode the memory tables using offset binary coding (OBC). However, the computational workload for updating the memory table remained unchanged from the non encoded case for the same filter length. By modifying the value the memory table is initialized, the computational workload for updating the memory table is reduced for most filtering configurations. This modification of SBDA to encode the

memory tables using OBC and to initialize the memory table with the initial condition instead of zero as originally proposed in SBDA is called SBDA-OBC. When SBDA-OBC is compared with SBDA, the following conclusions were deduced. SBDA-OBC requires less memory than SBDA. The maximum memory usage advantage of SBDA-OBC over SBDA is approximately half, and this situation occurs when the size of the sub-filter is large. In terms of the computational workload, SBDA-OBC is most advantageous for large sub-filters and when the filter is split into few sub-filters. In this case, the computational workload is reduced almost in half.

3.4 Contributions

In the research proposed in this section, contributions are made in the development of an adaptive filter using distributed arithmetic. The issues addressed by this research are the lack of an efficient method to fully update the memory table of a distributed arithmetic adaptive filter, the usage of memory resources beyond that of the non-adaptive case, and the compromised convergence rate. The following contributions were made to address these issues and to develop an adaptive distributed arithmetic filter with an efficient method to fully update the memory table without using additional memory resources and with uncompromised convergence performance.

1. A new method for fully updating the memory table was proposed. By fully updating the memory table, the convergence performance of the adaptive filter remains unaffected; therefore, the proposed update method can be used to construct adaptive filtering structures using distributed arithmetic without a compromised convergence rate.

2. A new method for efficiently updating the entire memory table was proposed. The proposed update method reduced the computational workload for updating the memory table by about $k/2 - 1$ over brute-force.

3. By using a filter coefficient driven distributed arithmetic filtering structure, the additional memory resources required by most other types of adaptive distributed arithmetic filtering structures, which use an input driven memory table that is composed of the combinations of its filter coefficients, are eliminated.

In this research, the proposed memory update method is combined with a filter coefficient driven distributed arithmetic filtering structure. This combination is called conjugate distributed arithmetic (CDA).

After a through literature review of adaptive distributed arithmetic filtering structures, only one other type of adaptive DA that is called sliding-block distributed arithmetic (SBDA) also addresses the issues outlined above with an adaptive DA filter. Although CDA is not the only adaptive distributed arithmetic filtering structure that addresses these issues, CDA is advantageous in a variety of filter configurations.

1. Among the adaptive distributed arithmetic filters that fully updates its memory tables, CDA uses the least amount of memory. Its memory usage is only matched by the brute-force method; however, CDA reduces the number of operations required by about $k/2 - 1$ over brute force. Recall, only adaptive distributed arithmetic filters that fully updates its memory tables is able to maintain the convergence speed of the LMS algorithm.

2. CDA uses less memory than SBDA especially if the filter is broken up into few subunits. This advantage is useful when coded on a system with limited memory.

3. CDA uses fewer additions than SBDA when the bit precision of the filter coefficients in SBDA is greater than the number of additional memory table entries that need to be updated in CDA. Typically, this occurs when the coefficient bit precision, the number of subunits, the depth of the memory tables, or a combination of these three are low. A couple benefits of fewer additions are boosted sampling rate, or lower power usage.

In addition to CDA and SBDA, an alternative adaptive DA filter structure called SBDA-OBC was proposed and developed. Its memory update method is a modification of the one used in SBDA, and it has lower memory usage and fewer additions for most filter configurations over SBDA. The principle motivations for modifying SBDA are to encode the memory using OBC such that the memory usage is reduced almost in half and to modify SBDA in such a way that when the memory is encoding using OBC that the computational workload for updating the memory table is reduced. The following are the observed savings of SBDA-OBC.

1. SBDA-OBC has the lowest memory requirements of any current mechanization for a coefficient driven, memory-based DA adaptive FIR filter. Specifically, SBDA-OBC uses about 50% less memory than SBDA when the filter length of the subunits is long.

2. SBDA-OBC has the fewest number of additions for a large number of filtering configurations among the current mechanizations for a coefficient driven, memory-based DA adaptive FIR filter. Specifically, SBDA-OBC needs about 50% less additions than SBDA when the filter length of the subunits is long.

CHAPTER 4

DIGITAL-TO-ANALOG MIXED-SIGNAL DISTRIBUTED ARITHMETIC FIR FILTER

DA is computationally more efficient than MAC-based approaches when the input vector length is large. However, the trade-off for the computational efficiency is the increased power consumption and area usage due to the use of a large memory. These problems can be alleviated by utilizing mixed-signal circuit implementations for optimized DA performance, power consumption, and area usage. The proposed mixed-signal DA architecture [38-40], which was done in partnership with Erhan Özalevli, is built by utilizing the analog storage capabilities of floating-gate (FG) transistors for programmability. The circuit compactness is obtained through the application of the iterative nature of the DA computational framework, where many multipliers and adders are replaced with an addition stage, a single gain multiplication, and a coefficient array.

The computational efficiency of this DA implementation is demonstrated by configuring it as an FIR filter. The low-power implementations of these filters can readily ease the power consumption requirements of portable devices. Also, due to the serial nature of the DA computation, the power and area of this filter increase linearly with its order. Hence, this design approach allows for a compact and low-power implementation of high-order FIR filters.

In the next section, the DA computation is described. Subsequently, the hybrid distributed arithmetic architecture is explained, and the integration of tunable voltage references into the DA implementation is described. After that, the precise programming/tuning of these voltage references is explained. In addition, the theoretical analysis of second order effects in the design is given and the experimental results of this reprogrammble distributed arithmetic FIR filter are presented. In the last part of this chapter, previously reported FIR filter implementations are given and their design issues are summarized and compared with

the proposed implementation.

4.1 Analog Architecture

This hybrid DA architecture is composed of four components, which are a 16-bit shift register, an array of tunable FG voltage references (*epots*) [41, 42], inverting amplifiers (*AMP*), and sample-and-hold (*SH*) circuits, as illustrated in Figure 4.1. The DA computation is governed by the timing of the digital data and control bits and an illustration of the timing is shown in Figure 4.2. Digital inputs are introduced to the system by using a serial shift register. These digital input words are the digital bits, $b_{i,j}$ in Eq. (2.3), which select the epot voltages to form the appropriate sum of weights necessary for the DA computation at the j^{th} bit. The clock frequency of the shift register is dependent on the input data precision, K, and the length of the filter, M, and is equal to $M \cdot K$ times the sampling frequency. Once the j^{th} input word is serially loaded into the top shift register, the data from this register is latched at K times the sampling frequency. If the amount of area used by the shift registers is not a design concern, then ideally an M-tap FIR filter should have M shift registers. A clock that is K times faster than the sampling frequency would be used for this ideal configuration.

The analog weights of DA are stored by the epots. When selected, these weights are added by employing a charge amplifier structure composed of same size capacitors and a two-stage amplifier, AMP_1. The epot voltages, as well as the rest of the analog voltages in the system, are referenced to a reference voltage, $V_{ref} = 2.5V$. Since the addition operation is performed by using an inverting amplifier, the relative output voltage, when the *Reset* signal is enabled, is equal to the negative sum of the selected weights for $C_{in_i} = C_{FBamp_1}$. For the first computational cycle, the result of the addition stage is the summation, $\sum_{i=0}^{m-1} w_i b_{i(K-1)}$, in Eq. (2.3) which is the addition of weights for the LSBs of the digital input data.

In the feedback path of the system, a delay, an invert and a divide-by-two operations

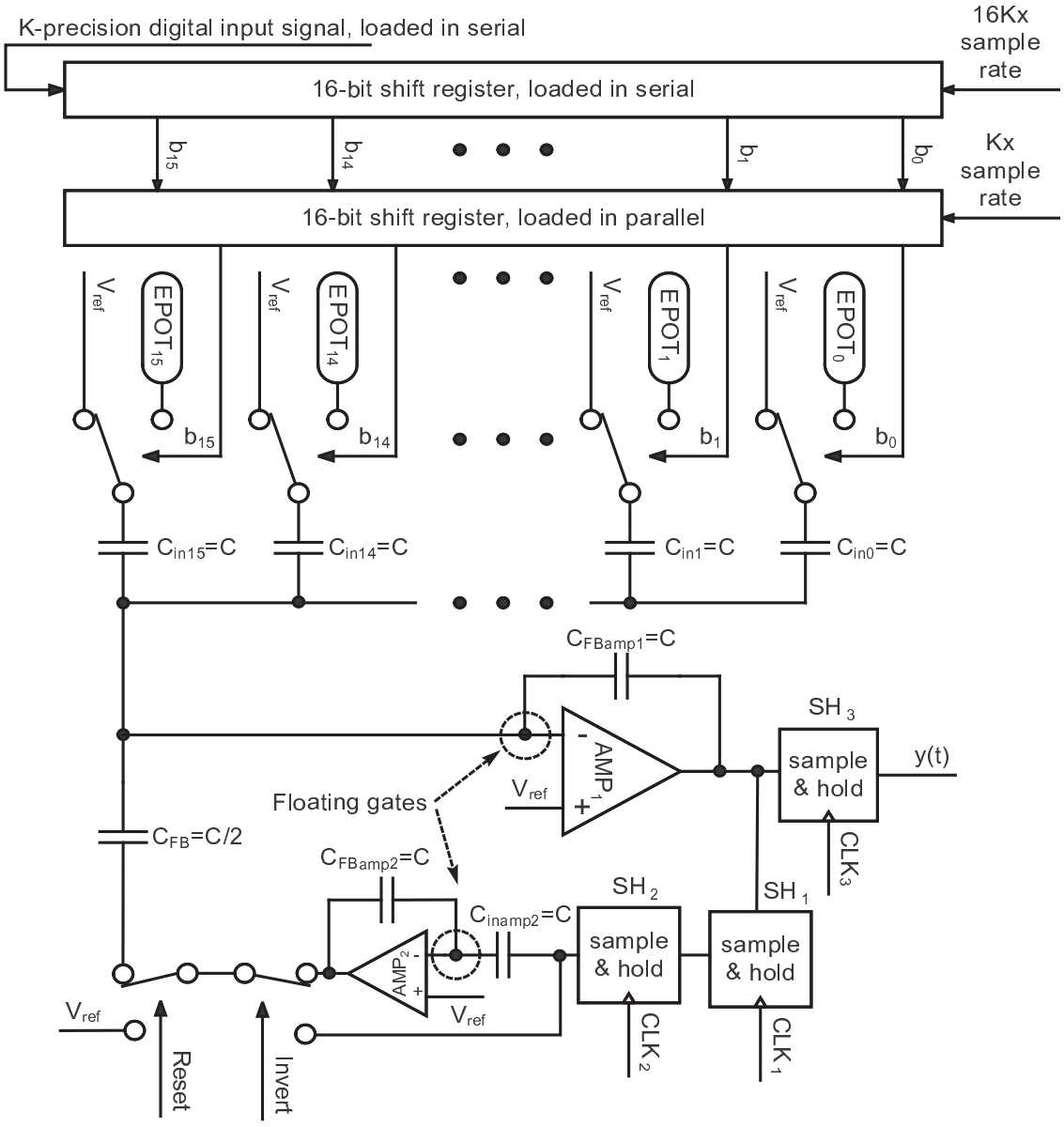


Figure 4.1. Implementation of the 16-tap hybrid FIR filter. b_i is the input bit for j^{th} cycle of operation and $y(t)$ is the output. Epots store the analog weights. Sample-and-holds, SHs, are used to obtain the delay and hold the computed output voltage.

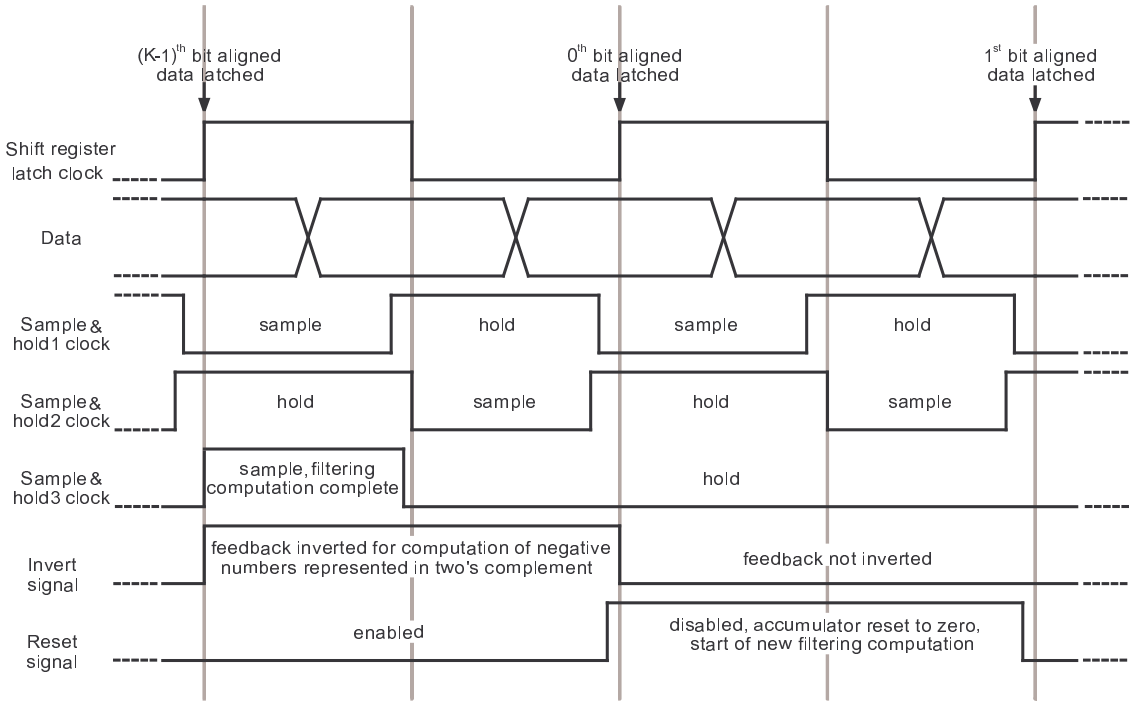


Figure 4.2. Digital clock diagram of the filter architecture.

are each used for the DA computation. For that purpose, sample-and-hold circuits, SH_1 and SH_2, and inverting amplifiers, AMP_1 and AMP_2, are employed in the implementation. The amplifier outputs which are used to feed it back into the system for the next cycle of the computation are stored in the SH circuits. Non-overlapping clocks, CLK_1 and CLK_2, are used to hold the analog voltage while the next stream of digital data is introduced to the addition stage. These clocks have a frequency of K times the sampling frequency. The stored data is then inverted relative to the reference voltage by using the second inverting amplifier, AMP_2, to obtain the same sign as the summed epot voltages. AMP_2 is identical to AMP_1 and has the same size input/feedback capacitors. After obtaining the delay and the sign correction, the stored analog data is fed back to the addition stage as delayed analog data. During the addition, it is also divided by two by using $C_{FB} = C_{FBamp_1}/2 = C/2$ which gives a gain of 0.5 when it is added to the new sum. This operation is repeated until the MSBs of the digital input data is loaded into the shift register. The MSBs are the $(K-1)^{th}$

bits and are used to make the computation 2s-complement compatible. This compatibility is achieved by disabling the inverting amplifier in the feedback path during the last cycle of the computation by enabling the *Invert* signal. As a result during the last cycle of the computation, the relative output voltage of AMP_1 is

$$V_{out_{amp_1}} - V_{ref} = -\sum_{i=0}^{M-1} \frac{C_{in_i}}{C_{FB_{amp_1}}} \left(V_{ref} - V_{epot_i}\right) b_{i0} + \sum_{j=1}^{K-1} 2^{-j} \sum_{i=0}^{M-1} \frac{C_{in_i}}{C_{FB_{amp_1}}} \left(V_{ref} - V_{epot_i}\right) b_{ij} \tag{4.1}$$

where the first term is the result of the calculation with the sign bits. Finally when the computation of the output voltage in Eq. (4.1) is finished, it is sampled by SH_3 using CLK_3 which is enabled once every K cycle. The computed voltage is stored in SH_3 until the next analog output voltage is ready. The new computation is started by enabling the *Reset* signal to zero out the effect of the previous computation. Then, the same processing steps are repeated for the next digital input data.

4.2 Analog Circuit Details

To achieve an accurate computation using DA, the circuit components are designed to minimize the gain and offset errors in the signal path. In this architecture, those components are the epots, the inverting amplifiers, and the sample-and holds.

The epot, shown in Figure 4.3, is modified from its original version [41] to obtain a low-noise voltage reference. It is a dynamically reprogrammable, on-chip voltage reference that uses a low-noise amplifier integrated with FG transistors and programming circuitry to tune the stored analog voltage. The amplifier in the epot circuit is used to buffer the stored analog voltage so that the epot can achieve low noise and low output resistance as well as the desired output voltage range. An array of epots is used for storing the filter weights; and during the programming, individual epots are controlled and read by employing a decoder.

In this architecture, epots and inverting amplifiers are the main blocks that use FG transistors to exploit their analog storage and capacitive coupling properties. A precise tuning of the stored voltage on FG node is achieved by utilizing the hot-electron injection

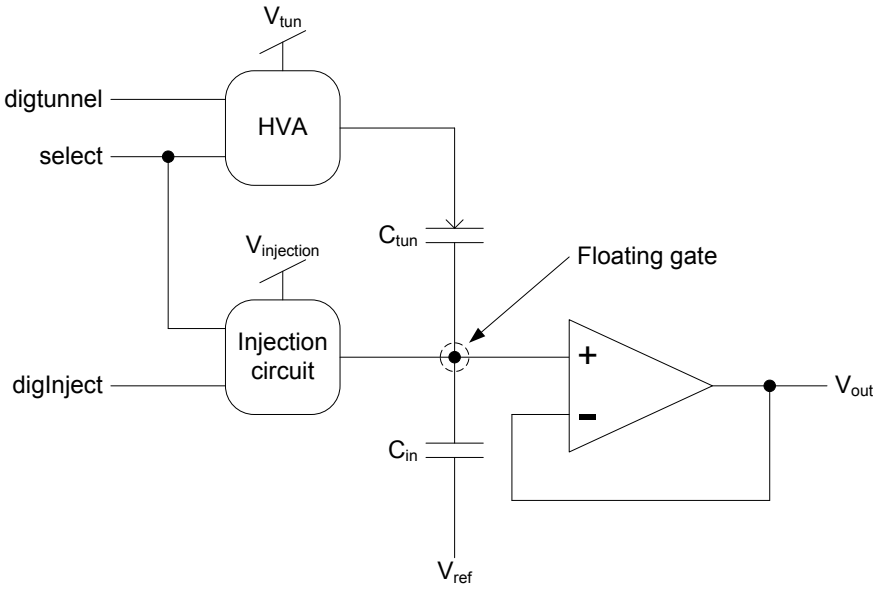


Figure 4.3. Schematic of the epot-based low-noise voltage reference.

and the Fowler-Nordheim tunnelling mechanisms. The epots employ FG transistors to store the analog coefficients of the inner product. In contrast, the inverting amplifiers use them not only to obtain capacitive coupling at their inverting-node, but also to remove the offset at their FG terminals.

One of the main advantages of exploiting FG transistors in this design is that the area allocated for the capacitors can be dramatically reduced. It is shown in [43] that epots can be utilized to implement a compact programmable charge amplifier DAC. This structure helps to overcome the area overhead, which is mainly due to layout techniques used to minimize the mismatches between the input and feedback capacitors. Similarly in this DA implementation, the unit capacitor, C, is set to $300fF$, and no layout technique is employed. As expected, due to inevitable mismatches between the capacitors, there will be a gain error contributed from each input capacitor. The stored weights are also used to compensate this mismatch. When the analog weights are stored to the epots, the gain errors are also taken into account to achieve accurate DA computation.

Unlike switched-capacitor amplifiers, the addition in this implementation is achieved

without resetting the inverting node of the amplifiers. This is because the floating-gate inverting-node of the amplifiers allow for the continuous-time operation. This design approach eliminates the need for multi-phase clocking or resetting. The inverting amplifiers are implemented by using a two-stage amplifier structure [44], shown in Figure 4.4, to obtain a high gain and a large output swing. Similar to the epots, the charge on the FG node

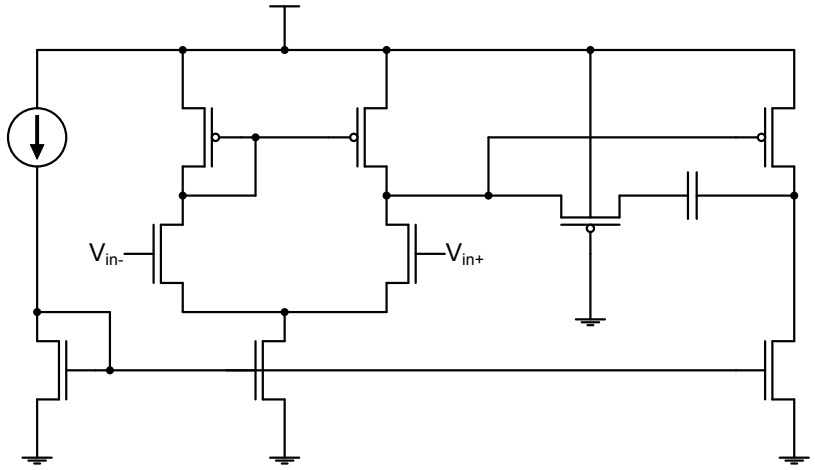


Figure 4.4. Schematic of the two-stage, high gain, large output swing inverting amplifier.

of these amplifiers is precisely programmed by monitoring the amplifier output while the system operates in the reset mode. In this mode, the shift registers are cleared and the Reset signal is enabled. Therefore, all the input voltages to the input capacitors including the voltage to the feedback capacitor, C_{FB}, are set to the reference voltage. These conditions ensure that the amplifier output becomes equal to the reference voltage when the charge on the FG is compensated. The charge on the FG terminal is tuned using the hot-electron injection and the Fowler-Nordheim tunnelling mechanisms. By using this technique, the offset at the amplifier output is reduced to less than 1mV.

Lastly, SH circuits need to be designed to simultaneously achieve high sampling speed and high sampling precision due to the bit-serial nature of the DA computation. Therefore, these circuits are implemented by utilizing the sample-and-hold technique using Miller hold capacitance [45], as illustrated in Figure 4.5. This compact circuit minimizes the

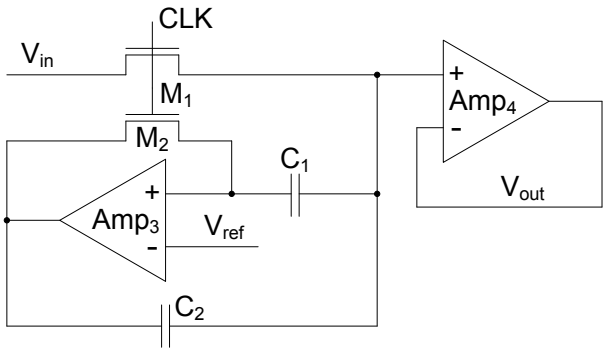


Figure 4.5. Schematic of the sample-and-hold circuit utilizing the Miller hold capacitance.

signal dependent error, while maintaining the sampling speed and precision by using the Miller capacitance technique together with Amp_3 shown in Figure 4.6. For simplification,

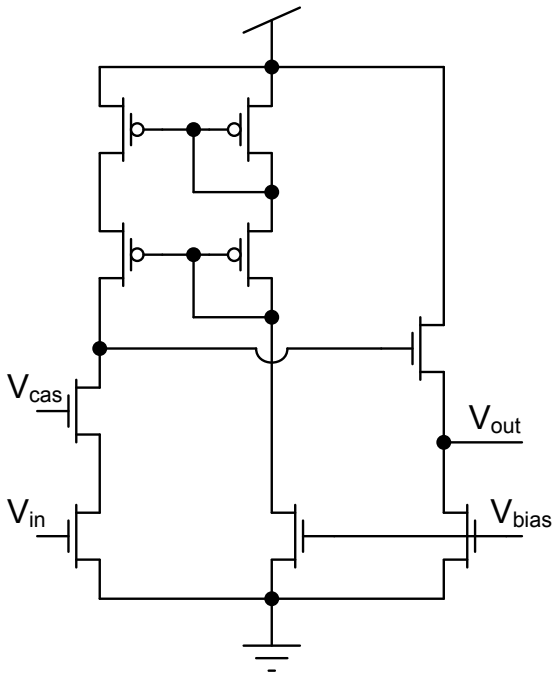


Figure 4.6. Schematic of Amp_3 used in Figure 4.5.

if we assume there is no coupling between M_1 and M_2, and amplifier, Amp_3, has a large gain, then the pedestal error contributed from turning switches (M_1 and M_2) off can be written as

$$\Delta V_{S1} + \Delta V_{S2} = \frac{\Delta Q_1(C_2 + C_{2B})}{C_{2B}(C_1 + C_2) + C_1C_2(A+1)} + \frac{\Delta Q_2}{C_2} \tag{4.2}$$

where ΔQ_1 and ΔQ_2 are the charges injected by M_1 and M_2, respectively. Also, A and C_{2B} are the gain and input capacitance of the amplifier, Amp_3. ΔQ_2 is independent of the input level, therefore ΔV_{S2} can treated as an offset. In addition, the error contributed by M_1, ΔV_{S1}, can be minimized by the Miller feedback, and this error decreases as A increases [45]. Due to serial nature of the DA computation offset in the feedback path is attenuated as the precision of the digital input data increases. Therefore, Amp_3 is designed to minimize mainly the signal dependent error, ΔV_{S1}.

Moreover, a gain-boosting technique [46] is incorporated into the SH amplifier, Amp_4, as shown in Figure 4.7, to achieve a high gain and fast settling. Two SH circuits are used in

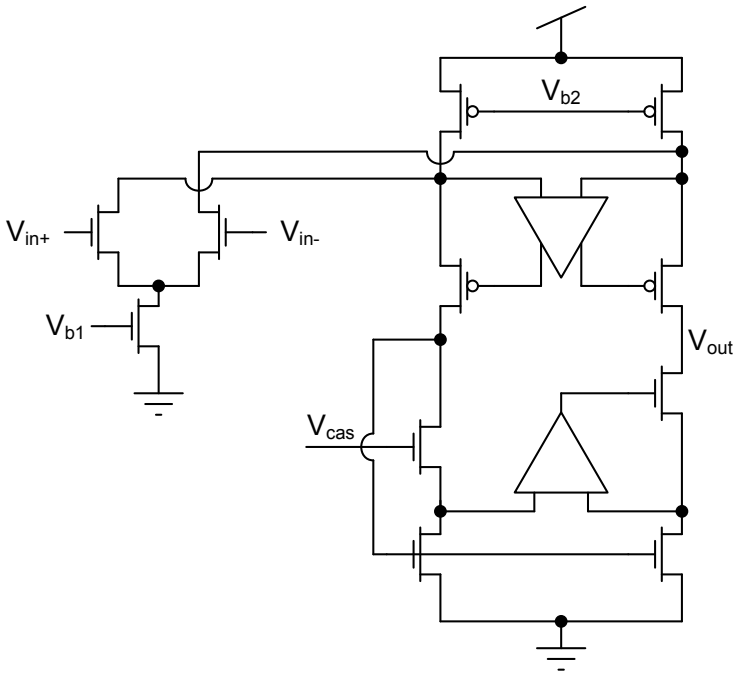


Figure 4.7. Schematic of Amp_4 used in Figure 4.5.

the feedback path to obtain the fixed delay for the sampled analog voltage. In addition, the third SH is utilized to sample and hold the final computed output once every K cycles. This SH uses a negative-feedback output stage [47], shown in Figure 4.8, to be able to buffer the output voltage off-chip. Due to the performance requirements of the system, these SH circuits consume more power than the rest of the system.

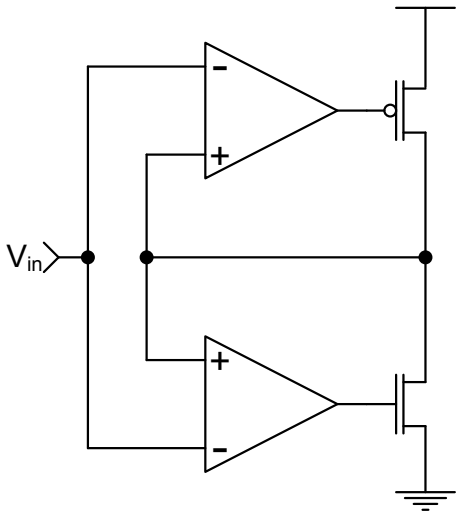


Figure 4.8. Schematic of the output buffer.

4.3 Programming the Analog Weights

The epots are incorporated into the design not only to store the weights of DA, but also to obtain reprogrammable tunability. In this section, programming of these epots is described.

The epot programming circuitry is shown in Figure 4.9. The stored voltage is tuned

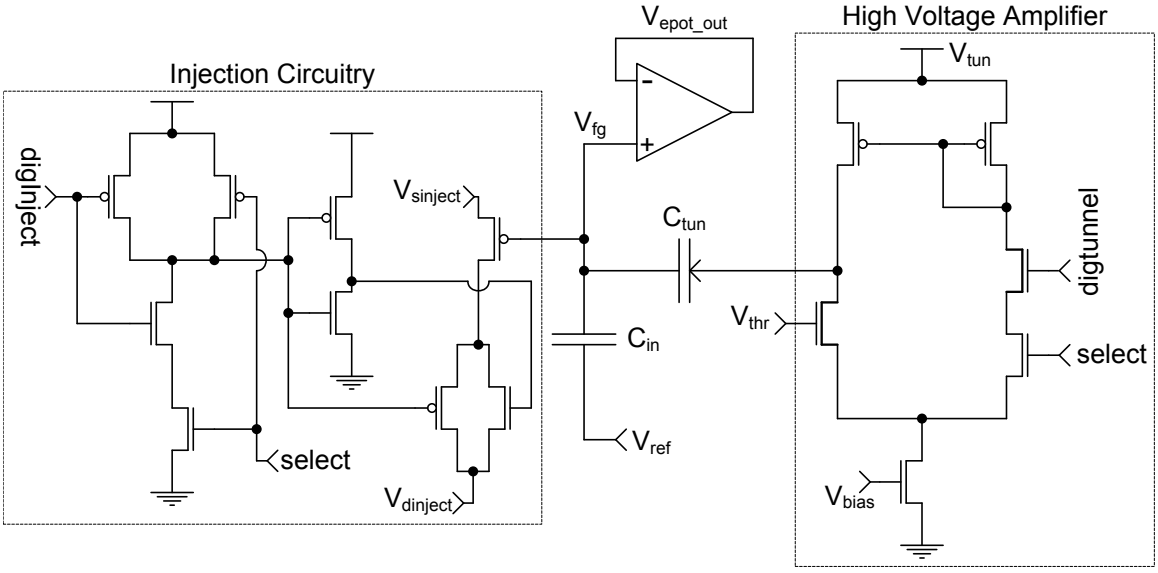


Figure 4.9. Schematic of the epot programming circuitry.

by the using Fowler-Nordheim tunnelling and the hot-electron injection mechanisms. The

tunnelling is utilized for coarse programming of the epot voltage, and used to reach $200mV$ below the target voltage. The purpose of undershooting is to avoid the coupling effect of the tunnelling junction on the floating gate when tunnelling is turned off. The tunnelling mechanism decreases the number of electrons, thus increasing the epot voltage. After selecting the desired epot by enabling its select signal, the tunnelling bit, digtunnel, is activated and a high voltage across the tunnelling junction is created. During programming, the high voltage amplifier is powered with $14V$.

In contrast to the coarse programming, the precise programming is achieved by using the hot-electron injection. The hot-electron injection mechanism decreases the epot voltage by increasing the number of electrons on the FG terminal. It is performed by pulsing a $6.5V$ across the drain and the source terminals of a pFET, as illustrated in Figure 4.9. As the FG voltage, V_{fg}, decreases, the injection efficiency drops exponentially since the injection transistor has better injection efficiency for smaller source-to-gate voltages. By keeping the FG potential at a constant voltage, the number of injected electrons, hence the output voltage change, is accurately controlled. To keep the FG at a constant potential, the input voltage of the epot, V_{ref}, is modulated during programming based on the epot voltage output, since the epot output is approximately at the same potential as V_{fg}.

After the programming, the tunnelling and injection voltages are set to ground to decrease the power consumption, and minimize the coupling to the floating-gate terminal. Also, V_{ref} is set to $2.5V$ to have the same reference voltage for all parts of the system. The epot voltage is programmed with respect to this voltage reference with an error less than 1mV for a $4V$ output range. The amount of charge that needs to be stored at an epot depends on the targeted weight and the gain error introduced by the input/feedback capacitors at the addition stage. During the programming, the Reset signal is enabled and all other capacitor inputs are connected to V_{ref} while periodically switching the targeted epot to find the voltage difference when epot is selected and unselected. This voltage is used to find the approximate value of the stored weight.

4.4 Analysis of Error Sources

In the analog domain, other sources of error exist for a distributed arithmetic FIR filter that is not observed in the digital domain. In this section, an analysis of these error sources that are generated by the analog components used in the proposed circuit is performed. These error sources are gain and offset errors, non-ideal weights, and noise in the signal path. In this section, the source of these errors are identified, and their contribution on DA are analyzed. The effect of non-ideal weights is mostly dependent on the application that DA is used for. For this particular implementation, their effect on FIR filtering applications is analyzed.

4.4.1 Computational Errors

As in serial digital-to-analog converters, the gain and offset errors are determined by the accuracy of the DA computation. If the error at the addition stage is due to the weight errors in the epots and the mismatch errors between the input capacitors, C_{in_i} (for $i = 1, 2, ...$), is assumed to be negligible, then the gain/offset errors and the noise in the data paths are the main sources of error. In this architecture, the inverting amplifiers, AMP_1 and AMP_2, are sources of gain and offset errors, and the sample-and-hold circuits, SH_1 and SH_2, are sources of offset error. In addition, the mismatch between C_{FB} and C_{FBamp_1} as well as between C_{FBamp_2} and C_{inamp_2} are sources of gain error. In this analysis, the effects of gain, offset, and random errors in the system were analyzed.

4.4.1.1 Gain Error

Unlike in the digital domain where a division by two is simply a shift of a bit, in the analog domain, this operation is achieved by employing an analog circuit. Usually, an error is generated by this circuit implementation, and the result of the division is 0.5 plus a gain error, Δ. The effect of Δ on the output of a DA computation, $y[n]$, is modelled by

$$y[n] = -\sum_{i=0}^{M-1} w_i b_{i0} + \sum_{j=1}^{K-1} (0.5 + \Delta)^j \sum_{i=0}^{M-1} w_i b_{ij} \tag{4.3}$$

The output error caused by Δ can be found by computing the difference of Eq. (2.3) and Eq. (4.3). For simplification, the term $\sum_{i=0}^{M-1} w_i b_{ij}$ is set to α. Therefore, the output error, ε, is reduced to the difference of two geometric sums and can be expressed as

$$\varepsilon = \alpha \frac{1-(0.5+\Delta)^K}{1-(0.5+\Delta)} - \alpha \frac{1-0.5^K}{1-0.5} = \alpha \frac{1-(0.5+\Delta)^K}{0.5-\Delta} - \alpha \frac{1-0.5^K}{0.5} \tag{4.4}$$

A plot of the output error due to gain error normalized by α for varying values of Δ and K is given in Figure 4.10. Since this system converts the digital input data to an analog

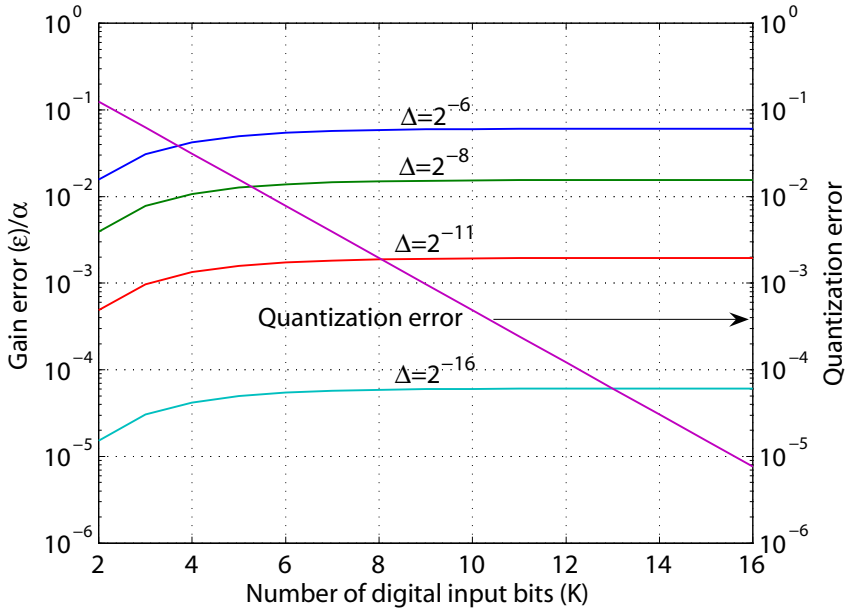


Figure 4.10. Computational error, ε/α, of the system due to quantization error and gain non-ideality, Δ.

output, the output error due to quantization is also provided. The intersection of output error (due to gain error) and quantization error curves is the minimum achievable output error of the proposed system and is used to determine the precision of an equivalent digital system. For example when $\Delta = 2^{-11}$, the two curves intersected at $\varepsilon/\alpha = 0.002$ and $K = 8$. This intersection point is the minimum computational error when $\Delta = 2^{-11}$, and the proposed system is equivalent to using an 8-bit digital DA at this intersection point. Also note that as K became large, ε/α approached a limit which is equal to $2\Delta/(0.5-\Delta)$.

4.4.1.2 *Offset Feedback Error*

Another source of error is the offset error. It is modelled as a constant error, δ, added to each j^{th} summation of weights, $\sum_{i=0}^{M-1} w_i b_{ij}$, as follows

$$y[n] = -\left[\delta + \sum_{i=0}^{M-1} w_i b_{i0}\right] + \sum_{j=1}^{K-1} 2^{-j}\left[\delta + \sum_{i=0}^{M-1} w_i b_{ij}\right] \tag{4.5}$$

After distributing $\sum_{j=1}^{K-1} 2^{-j}$ and then grouping the δ into one term, the error due to offset can be written as the summation of a geometric series.

$$error_{\text{offset}} = \delta\,\frac{1-0.5^K}{1-0.5} - 2\delta = \delta \cdot 2^{-(K-1)} \tag{4.6}$$

As K increases, the offset error in the feedback loop decreases. As a byproduct of how DA handles two's complement numbers, the last summation of weights, $\sum_{i=0}^{M-1} w_i b_{i0}$, is subtracted rather than added. This system decreases the offset error especially when the K is large. For $K = 8$ and $\delta = 100mV$, the offset error is $0.7813mV$.

4.4.1.3 *Random Feedback Error*

The random error is assumed to be Gaussian and is represented by X_j. The random variable X_j is added to the summation of weights at each j^{th} iteration, and all X_j's are independent and identically distributed.

$$y[n] = -\left[X_0 + \sum_{i=0}^{M-1} w_i b_{i0}\right] + \sum_{j=1}^{K-1} 2^{-j}\left[X_j + \sum_{i=0}^{M-1} w_i b_{ij}\right] \tag{4.7}$$

Once the term $\sum_{j=1}^{K-1} 2^{-j}$ is distributed and the $X_j's$ are collected into one summation, the mean and variance of $y[n]$ can be written as $\mu_Y = \mu_X\,\frac{1-0.5^K}{1-0.5} - 2\mu_X$ and $\sigma_Y^2 = \sigma_X^2\,\frac{1-0.25^K}{1-0.25}$, respectively. As K approaches infinity, the mean of the random error approaches zero, and the maximum variance of the random error is $\frac{4}{3}\sigma^2$.

4.4.2 Non-ideal Weight Errors for FIR filters

The errors due to non-ideal filter weights, such as random offset error, are caused by the limited precision of the epot programming and the epot noise. The effects of these errors are similar to the quantization effects in the digital domain which causes the linear difference

equation of an FIR filter to become a nonlinear one [48]. In addition, an analysis of the random time-varying error of the filter weights is provided.

4.4.2.1 Offset Error

An analysis similar to the one presented in [48] is given. Although the analysis provided is for a Type 2 FIR filter, the analysis can be generalized to any type of symmetric FIR filter [48]. An analogous analysis can be performed for nonsymmetric FIR filters.

4.4.2.2 Symmetric Offset Error

The variable $e[n]$ is equal to the difference between the output generated using the ideal filter weights and the output generated using the non-ideal filter weights. The frequency response for $e[n]$ can be written as $E(w) = \sum_{n=0}^{M-1} e[n]e^{-jwn}$. Assuming the FIR filter is Type-2 and the offset errors are of the same symmetry as the filter, $E(w)$ can be rewritten as a summation of cosines.

$$E(w) = e^{-jw\frac{M-1}{2}} \sum_{n=0}^{\frac{M}{2}-1} 2e[n]cos\left(w(\frac{2n+1}{2})\right) \tag{4.8}$$

Treating $e[n]$ as a random variable with a variance of σ_e^2 and using some trigonometric identities and Euler's rule, the variance of $E(w)$ can be written as follows

$$\sigma_E^2(w) = \sigma_e^2\left(M + \frac{sin(wM)}{sin(w)}\right) \tag{4.9}$$

$\sigma_E^2(w)$ can vary from zero to $2M\sigma_e^2$. Its frequency response for $M = 32$ is illustrated in Figure 4.11. The effects of symmetrical offset errors are similar to the effects of coefficient quantization in symmetrical digital FIR filters. These effects are reduced pass-band width, increased pass-band ripple, increased transition-band, and reduced minimum stop-band attenuation [48].

4.4.2.3 Nonsymmetric Offset Error

Unlike the previous analysis, $E(w)$ cannot be rewritten as a summation of cosines because the offset error is not symmetrical. Assuming $e[n]$ is a random variable with a variance of σ_e^2, the variance of $E(w)$, σ_E^2, is equal to $M\sigma_e^2$ for an M-tap FIR filter. Unlike the variance

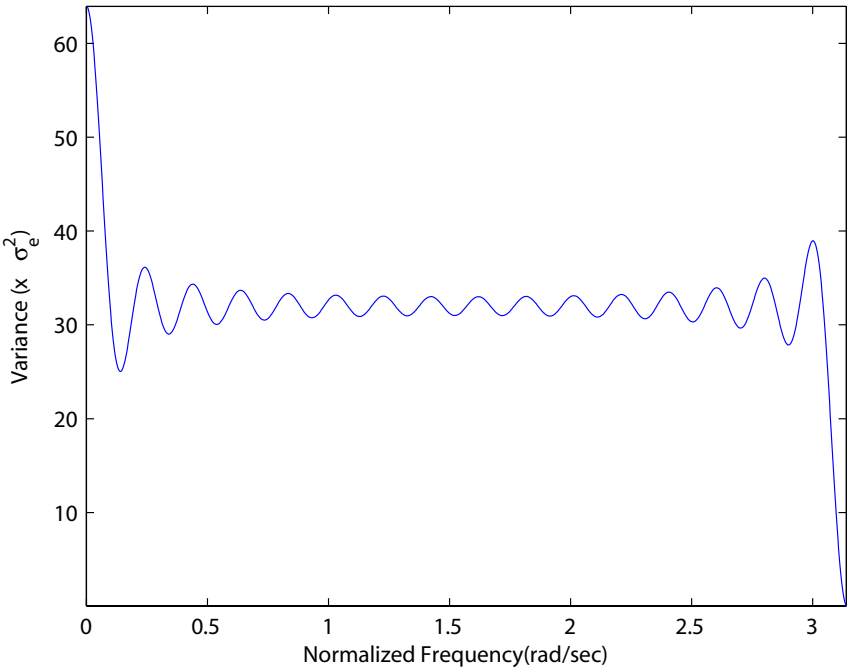


Figure 4.11. Frequency response of the variance for symmetric offset error, $M = 32$.

for symmetrical offset errors which varies with frequency, the variance for nonsymmetric offset errors is constant.

4.4.2.4 Random Error

The effects of time-varying random error on DA computation can be modelled as

$$y[n] = -\sum_{i=0}^{M-1}(w_i + e_{i0})b_{i0} + \sum_{j=1}^{K-1} 2^{-j} \sum_{i=0}^{M-1}(w_i + e_{ij})b_{ij} \tag{4.10}$$

Assuming each e_{ij} is a random variable that is independent and identically distributed, the error can be expressed as

$$error = -\sum_{i=0}^{M-1} e_{i0}b_{i0} + \sum_{j=1}^{K-1} 2^{-j} \sum_{i=0}^{M-1} e_{ij}b_{ij} \tag{4.11}$$

Since the above equation is just a summation of random variables, the parameter of significance for this analysis is the maximum variance of the random error, σ^2_{error}. For simplification, the analysis assumed that b_{ij} for all i and j is equal to 1. First, the variance of $-\sum_{i=0}^{M-1} e_{i0}b_{i0}$ is computed as $M\sigma^2$. Then, the variance of $\sum_{j=1}^{K-1} 2^{-j} \sum_{i=0}^{M-1} e_{ij}b_{ij}$ is calculated as $M\sigma^2 \frac{1-0.5^K}{1-0.5}$. These two variances added together are equal to a total variance,

$\sigma^2_{\text{error}} = M\sigma^2(3 - 0.5^{K-1})$, which approaches $3M\sigma^2$ when K is large.

4.5 Measurement Results

In this section, the experimental results from the proposed DA architecture, which is configured as an FIR filter, are presented. The measurement results are obtained using a Tektronix TDS 5034B oscilloscope from chips that were fabricated in a $0.5\mu m$ CMOS process. This 16-tap FIR filter is designed to run at $32/50kHz$ sampling frequency depending on the desired performance. The precision of the digital input data is set to 8 for these experiments. To meet this sampling rate, the data is loaded into the upper shift register at a rate of $3.84MHz$ for a $32kHz$ sampling frequency or $6.4MHz$ for a $32kHz$ sampling frequency.

To demonstrate the reprogrammability, the filter is configured as a comb, a low-pass, and a band-pass filter. The coefficients of these filters are shown in Table 4.1. Rather than

Table 4.1. Ideal and Actual (programmed epot voltages) coefficients of the comb, low-pass, and band–pass filters.

Filter	Comb		LPF		BPF	
Coefficients	Ideal	Actual (V)	Ideal	Actual (V)	Ideal	Actual (V)
1	0.4	2.0996	-0.0190	2.5192	0.033	2.4670
2	0	2.4994	-0.0390	2.5393	-0.064	2.5639
3	0	2.4994	0.0260	2.4738	-0.053	2.5530
4	0	2.5007	0.0160	2.4835	0.038	2.4617
5	0	2.5005	-0.0240	2.5239	0.047	2.4528
6	0	2.5000	-0.0360	2.5362	-0.054	2.5541
7	0	2.4999	0.0600	2.4401	-0.056	2.5561
8	0	2.4994	0.1800	2.3201	0.057	2.4425
9	0	2.4997	0.1800	2.3201	0.057	2.4427
10	0	2.5002	0.0600	2.4391	-0.056	2.5560
11	0	2.4998	-0.0360	2.5358	-0.054	2.5535
12	0	2.5002	-0.0240	2.5240	0.047	2.4527
13	0	2.5001	0.0160	2.4853	0.038	2.4616
14	0	2.5001	0.0260	2.4743	-0.053	2.5526
15	0	2.4997	-0.0390	2.5389	-0.064	2.5638
16	0.4	2.0996	-0.0190	2.5184	0.033	2.4669

using an oscilloscope, the actual voltages are measured using an Agilent model 3440 multimeter. The ideal coefficients are given to illustrate how close the epots are programmed to obtain the actual coefficients. The epots are programmed relative to a reference voltage, V_{ref}, which is set to $2.5V$. The error of the stored epot voltages are kept below $1mV$ to minimize the effect of weight errors on the filter characteristics.

An $858Hz$ sinusoidal output of the low-pass filter at a $50kHz$ sampling rate is illustrated in Figure 4.12. The spurious-free-dynamic-range (SFDR), which is the difference in

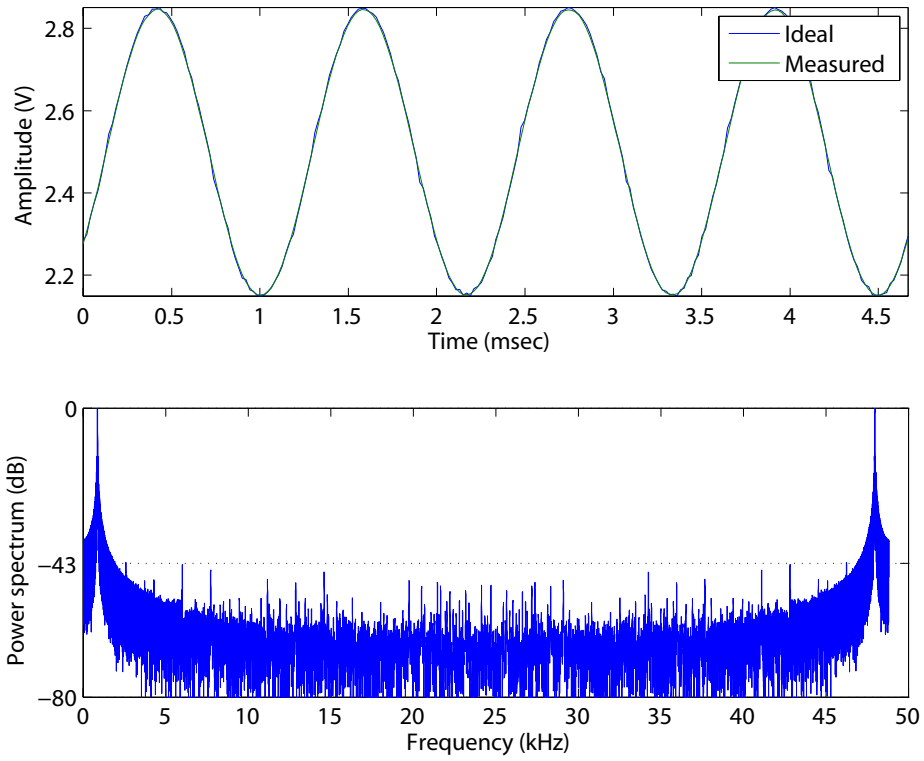


Figure 4.12. The output transient response and power spectrum of the low-pass filter for an input frequency of $858Hz$ with a $50kHz$ sampling frequency.

amplitude between the input frequency and the largest non-input frequency components, is measured to be $43dB$. For the comb filter with a $22kHz$ input signal frequency, it is observed that the SFDR does not degrade as shown in Figure 4.13. Although the input precision was set to 8 bits, the gain error in the system as well as noise in the experimental set-up limits the maximum achievable SFDR.

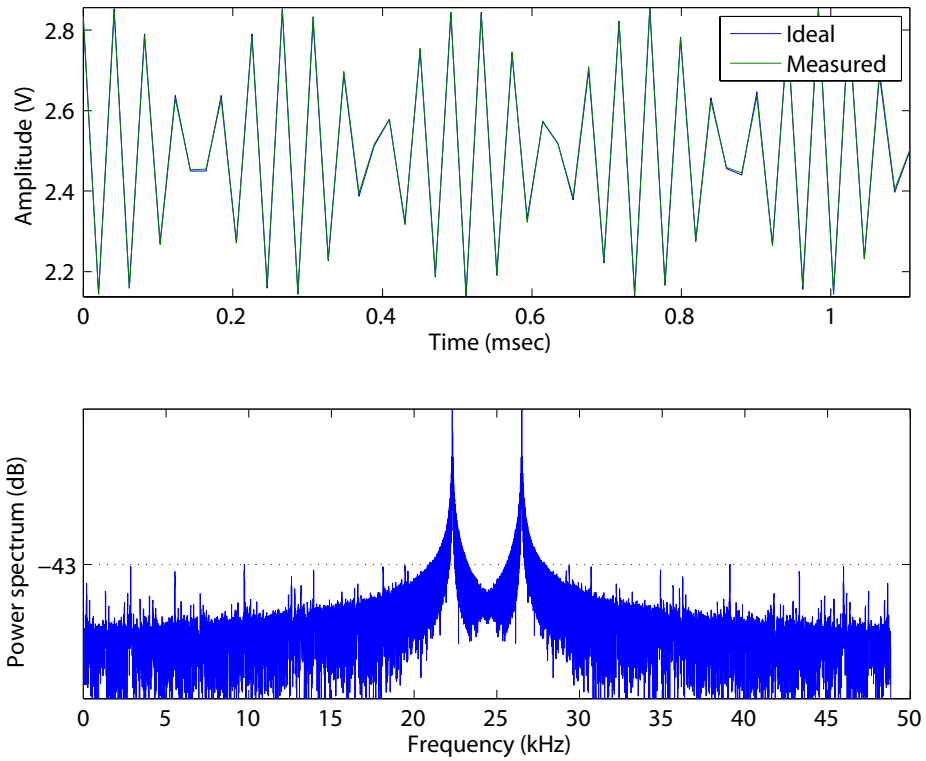


Figure 4.13. The output transient response and power spectrum of the comb filter for an input frequency of $22kHz$ with a $50kHz$ sampling frequency.

The second experiment is performed to characterize the magnitude and phase responses of the filters. For that purpose, a sinusoidal wave at a fixed sampling rate, $32/50KHz$, is generated using the digital data, and the magnitude and phase responses are measured by sweeping the frequency of the input sine wave from DC to $16/25kHz$. For this experiment, 256 data points are collected to accurately measure the frequency response of these filters. These responses follow the ideal responses closely even if the sampling rate is increased as illustrated in Figures 4.14, 4.15, and 4.16. Any variation in the frequency response

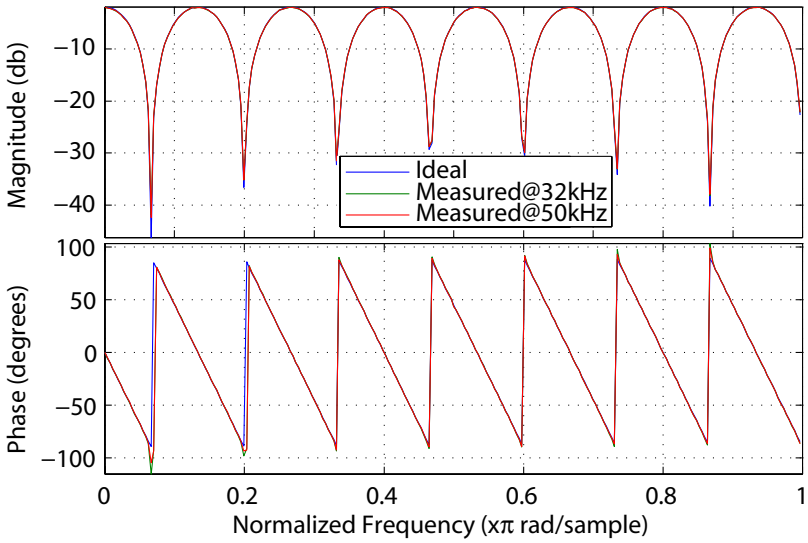


Figure 4.14. The magnitude and phase response of the comb filter at $32kHz$ and $50kHz$ sampling rates.

as the sampling rate increases is caused by the noise and offset in the feedback path as well as due to the performance degradation of the circuits. As the output signal amplitude becomes very low, the experimental set-up limits the resolvable magnitude and phase. As expected for a symmetrical FIR filter, the measured phase responses of comb, low-pass, and band-pass filters are linear.

The static power consumption of the fabricated chip is measured as $16mW$. Most of the power is consumed by the SH and inverting amplifier circuits. The die photo of the designed chip is shown in Figure 4.17. The system occupies around half of the $1.5 \cdot 1.5mm^2$ die area. The cost to increase the filter order is $0.011mm^2$ of die area and $0.02mW$ of power

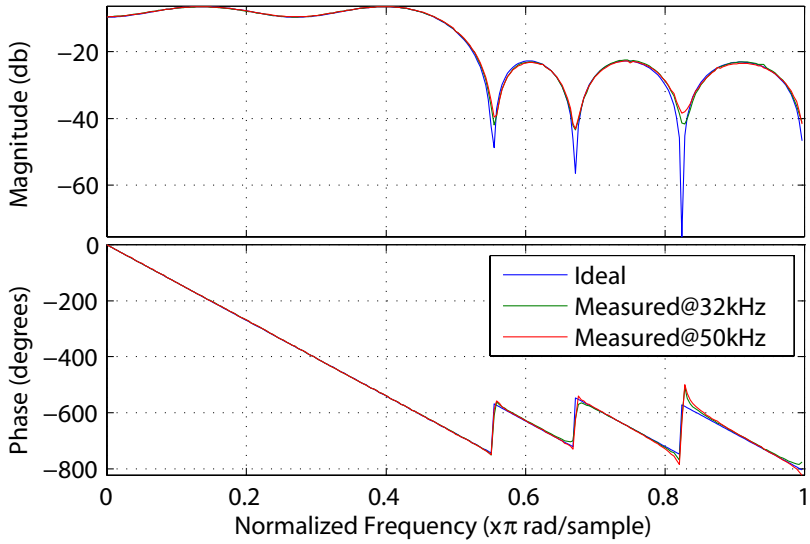


Figure 4.15. The magnitude and phase response of the low-pass filter at $32kHz$ and $50kHz$ sampling rates.

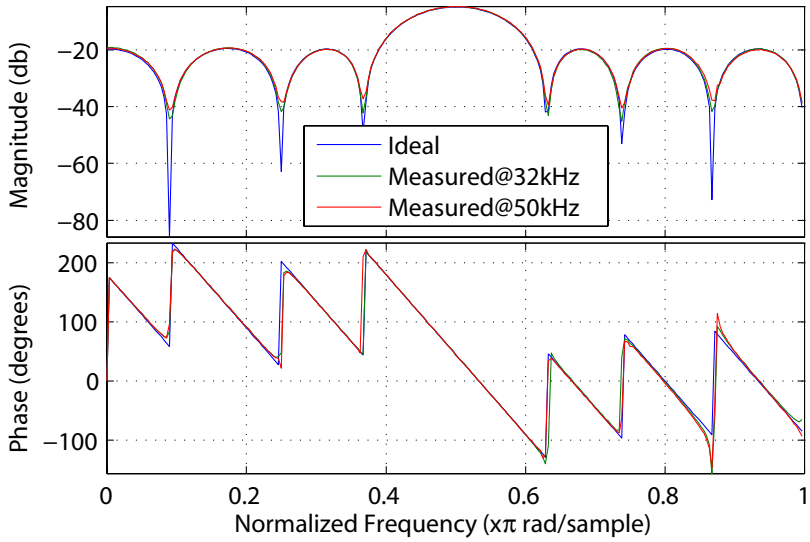


Figure 4.16. The magnitude and phase response of the band-pass filter at $32kHz$ and $50kHz$ sampling rates.

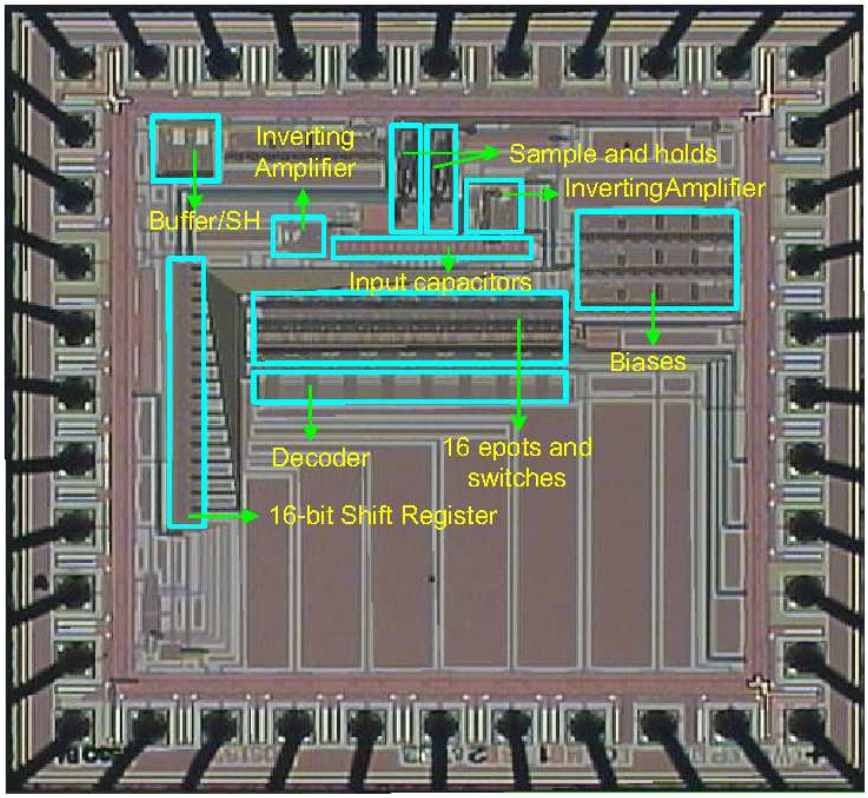


Figure 4.17. Die photo of the DA based FIR filter chip.

for each additional filter tap. This readily allows for the implementation of high-order filters. Lastly, the performance of the filter is summarized in Table 4.5.

Table 4.2. Performance and design parameters of the FIR filter.

Process	$0.5\mu m$, 2-*poly* CMOS
Power supply	$5V$
Reference voltage	$2.5V$
Epot Programming Resolution	100μV
Programming Mechanisms	Hot-Electron Injection and Electron Tunneling
Unit capacitor	$300fF$
Sampling frequency	$30/50KHz$
Input data precision	8
Number of filter taps	16
Increase in the power per tab	$0.02mW$
Increase in the area per tab	$0.011mm^2$
Total static power consumption	$16mW$
Used chip area	$\sim 1.125mm^2$

4.6 Summary

We have presented an implementation of a FIR filter exploiting the distributed arithmetic to minimize hardware complexity and floating-gate transistors to obtain programmable analog coefficients. The proposed architecture is well suited for implementing high-order FIR filters due to its low-area and power requirements for each additional FIR filter tap. Also, the programmable analog coefficients of this filter will enable the implementation of adaptive systems that can be used in applications such as noise cancellation and adaptive equalization.

4.7 Contributions

In this chapter, a reprogrammable mixed-signal FIR filter was proposed and developed. The issues addressed by the proposed reprogrammable mixed-signal FIR filter are the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter

coefficients, the inconsistent sampling of the input data, and the corruption of the input samples. The following contributions were made to address these issues.

1. A reprogrammable mixed-signal FIR filter was proposed and developed to address the issues of the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter coefficients, the inconsistent sampling of the input data, and the corruption of the input samples.

2. Distributed arithmetic and epots were used to address the issue of a lack of a compact reprogrammable filtering structure for a reprogrammable mixed-signal FIR filter. A combination of distributed arithmetic and epots were used to construct a compact reprogrammable mixed-signal FIR filter.

3. epots were used to address the issue of non-symmetric and imprecise filter coefficients for a reprogrammable mixed-signal FIR filter. epots were used to precisely reprogram the filter coefficients. These filter coefficients can be programmed with such precision that a natural by-product is high filter coefficient symmetry.

4. Digital registers were used to address the issues of inconsistent input data sampling and of input sample data corruption for a reprogrammable mixed-signal FIR filter. The digital registers were used to sample the input data consistently and to eliminate input data corruption. Because the digital registers can be cascaded without concern for data corruption, this ability to cascade many input digital registers together allows for the construct of a high order FIR filter using the proposed reprogrammable mixed-signal FIR filtering structure.

5. An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed reprogrammable mixed-signal FIR filter. Deduced from this analysis, a guideline was generated of where to focus the design effort for the proposed reprogrammable

mixed-signal FIR filter. The guideline is to use most of the design effort on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

To verify the feasibility of implementing a mixed-signal DA filter structure, equations were developed to model the effects of non-idealities on the performance. The non-idealities studied are gain and offset error, non-ideal coefficients, and noise in the signal path.

CHAPTER 5

FRAMEWORK FOR AN ANALOG-TO-ANALOG MIXED-SIGNAL DISTRIBUTED ARITHMETIC SECOND-ORDER SECTION FILTER

Although a digital-to-analog mixed-signal filter is useful, another practical filter topology would be if the input to output flow was both analog. In addition, converting the filter structure from a finite impulse response filter to an infinite impulse response (IIR) filter, in particular a second order section, would be of tremendous benefit. Although the filter would not be of linear phase, which would eliminate its usage from certain applications, an implementation of an analog, discrete-time second order section (SOS) filter is very commonly used.

This chapter will begin with an overview of applying distributed arithmetic to IIR filters, which a second-order section is a subset. Next, a description of the analog architecture is given. Also, this section includes a detailed description of the hardware operation of the proposed DA-based second-order section.

5.1 Distributed Arithmetic for Infinite Impulse Response Filters

Up to this point, distributed arithmetic has been used to implement FIR filtering structures. In this section, the application of distributed arithmetic to a generic infinite impulse response (IIR) filter is described. Since a second-order section is a type of IIR filter, this section is equally applicable to a second-order section.

The application of DA to an IIR filter is very similar to that of an FIR filter. As mentioned earlier, the output of a FIR filter is the inner product of the input and weight vectors, and DA is an efficient mechanization for computing an inner product, which makes DA well-suited for FIR filters. In the case of an IIR filter, DA is just as well-suited because the output of an IIR filter is the difference of the inner product of the feedback filter coefficients, w_a, and the output samples, $y[n]$, from the inner product of the feedforward filter

coefficients, w_b, and the sampled input, $x[n]$. Mathematically, this difference can be written as

$$y[n] = \sum_{i=0}^{K-1} w_b[i]\ x[n-i] + \sum_{j=1}^{L-1} w_a[j]\ y[n-j]. \tag{5.1}$$

for an IIR filter with K feedforward and L feedback coefficients. Note, this equation could have been just as easily been defined as the difference of two inner products instead of the summation of them without the loss of any meaning. Although this definition is not consistent with well known DSP references such as [49] and common DSP software development tools such as Mathworks Matlab, this definition was utilized to simplify the analysis of the effects of analog components in a mixed-signal DA second-order section implementation. This analysis is provided in the Section 5.5.

As shown earlier in the FIR filter case, the partial products used for DA can be generated by either the bit-slices of the digitized input samples or the quantized filter coefficients. Since in this case where an analog-to-analog input-to-output signal flow is desired, the partial products should be created using the bit-slices of the quantized filter coefficients, w_a and w_b. So let the feedforward and feedback filter coefficients be represented as B-bit 2's complement binary numbers,

$$w_a[j] = -a_{j0} + \sum_{l=1}^{B-1} a_{jl} 2^{-l}, \quad j = 1, \ldots, L-1, \text{ and} \tag{5.2}$$

$$w_b[i] = -b_{i0} + \sum_{l=1}^{B-1} b_{il} 2^{-l}, \quad i = 0, \ldots, K-1, \tag{5.3}$$

where a_{jl} is the l^{th} bit in the 2's complement representation of the j^{th} feedback filter coefficient, w_a, and b_{il} is the l^{th} bit in the 2's complement representation of the i^{th} feedforward filter coefficient, w_b. Substituting Eq. (5.2) and Eq. (5.3) into Eq. (5.1) and swapping the order of the summations for both the feedforward inner product, $\sum_{i=0}^{K-1} w_b[i]\ x[n-i]$, and the feedback inner product, $\sum_{j=1}^{L-1} w_a[j]\ y[n-j]$, yields

$$y[n] = -\left[\sum_{i=0}^{K-1} b_{i0} x[n-i]\right] + \sum_{l=1}^{B-1}\left[\sum_{i=0}^{K-1} b_{il} x[n-i]\right] 2^{-l} - \left[\sum_{j=1}^{L-1} a_{j0} y[n-j]\right] + \sum_{l=1}^{B-1}\left[\sum_{j=1}^{L-1} a_{jl} y[n-j]\right] 2^{-l}. \tag{5.4}$$

For a given set of $x[n-i]$ ($i = 0, \ldots, K-1$), the terms in the square braces for the feedforward inner product may take only one of 2^K possible values, and the terms in the square braces for the feedback inner product may take only one of $2^{(L-1)}$ possible values for a given set of $y[n-j]$ ($j = 1, \ldots, L-1$). These partial products for both the feedforward and feedback inner products may be stored into a memory table. The entry in the memory table for the feedforward path that is denoted as FF-MEM and that is addressed by r is given by

$$\text{FF-MEM}_{(r)} = \sum_{i=0}^{K-1} c_i^{(r)} x[n-i], \quad r = 0, \ldots, 2^K - 1, \tag{5.5}$$

where $c_i^{(r)}$ is the i^{th} bit in the K-bit representation of the address r. In other words,

$$r = \sum_{i=0}^{K-1} c_i^{(r)} 2^i. \tag{5.6}$$

For each l, $l = 0, \ldots, B-1$, the term in the first two square braces in Eq. (5.4) is essentially the entry in the FF-MEM whose address is $\sum_{i=0}^{K-1} b_{il} 2^i$. For the feedback path, the entry in its memory table that is denoted as FB-MEM and that is addressed by s is given by

$$\text{FB-MEM}_{(s)} = \sum_{j=1}^{L-1} d_j^{(s)} y[n-j], \quad s = 0, \ldots, 2^L - 1, \tag{5.7}$$

where $d_j^{(s)}$ is the $(j-1)^{\text{th}}$ bit in the $(L-1)$-bit representation of the address s. In other words, the address s can be expressed as

$$s = \sum_{j=1}^{L-1} d_j^{(s)} 2^{(j-1)}. \tag{5.8}$$

For each l, $l = 0, \ldots, B-1$, the term in the last two square braces in Eq. (5.4) is basically the entry in the FB-MEM whose address is $\sum_{j=1}^{L-1} a_{jl} 2^{(j-1)}$.

5.2 Analog Architecture

Building upon the earlier success of the digital-to-analog mixed-signal distributed arithmetic FIR filter, this structure can be modified to implement a second-order section. These type of filtering elements are useful as the fundamental building blocks of IIR filters designed using such methods as Butterworth, Chebyshev, and Elliptic. This design methodology is preferred because a single high order section is very sensitive to quantization errors,

which leads to stability issues, in the digital domain, and to process variations, which leads to device mismatch, in the analog domain. By using multiple second order sections, designing the desired filter is much less prone to these issues than using a single high order section. In addition to this conversion to a different filtering structure, this mixed-signal filter is targeted to be implemented on a field programmable analog array (FPAA).

5.3 Overview of the Field Programmable Analog Array

A field programmable analog array (FPAA) is a reprogrammable analog device that can implement multiple circuit topologies or filter types. These kind of devices are analogous to a field-programmable gate array (FPGA), which is used for the prototyping of digital circuits, in the digital domain. There are a few type of FPAA's available to use. The fundamental difference between these types of FPAAs is the basic circuit building block. Some FPAAs like the Anadigm AN120E04 [50] use a reprogrammable switched-capacitor circuit as its basic building block.

For this research, a floating-gate based FPAA is used. Specifically, the RASP 2.8 FPAA developed by the Integrated Computational Electronics Laboratory of the Georgia Institute of Technology under the direction of Dr. Paul Hasler [51]. Its fundamental building components are floating-gate switches, operational transconductance amplifiers (OTAs), and capacitors. In addition to these components, there are specialized components such as Gilbert Multipliers, floating-gate based current mirrors, floating-gate based multiple-input translinear elements (MITEs), I/O buffers, transmission gates, and NMOS and PMOS transistors. These components are grouped together into a larger unit called a computational analog block (CAB). Conceptually, these CABs are similar to the slices of a Xilinx FPGA or the logic cells of an Altera FPGA. The RASP 2.8 FPAA has thirty-two CABs that are connected through a floating-gate based routing fabric as shown in Figure 5.1. This FPAA has two different types of CABs. The most common CAB consists of three OTAs, four 500fF capacitors, two floating-gate based MITEs, a voltage buffer, a transmission gate, and

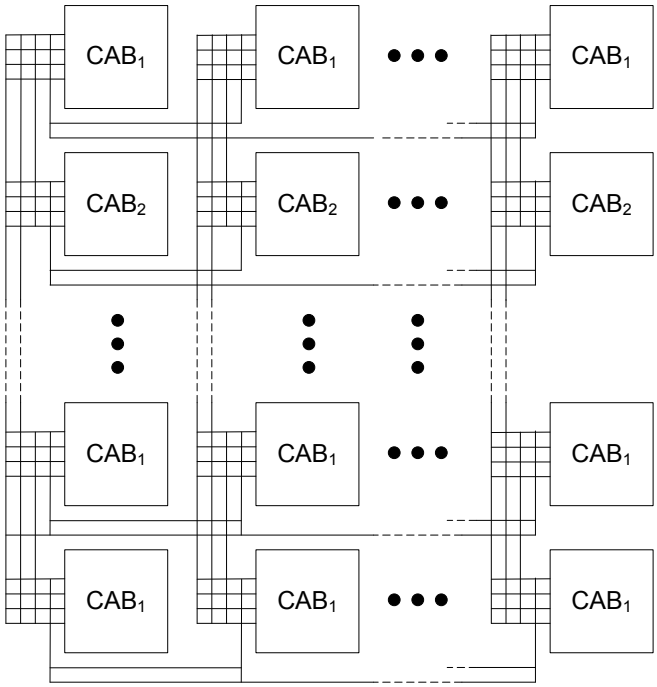


Figure 5.1. Overview of the CAB arrangement on the RASP 2.8 FPAA.

four NMOS or PMOS transistors. A schematic of this CAB is shown in Figure 5.2. The other type of CAB consists of a floating-gate based OTA, a pair of folded Gilbert multipliers, and a pair of floating-gate based current mirrors. A schematic of this CAB is shown in Figure 5.3.

5.4 FPAA Hardware Implementation

Using the digital-to-analog mixed-signal DA FIR filter as a template, an analog-to-analog mixed-signal second order section filter can be designed. However, unlike the earlier effort, which was fabricated as an ASIC, this filter was targeted for implementation on a RASP 2.8 FPAA. Although this FPAA uses operational transconductance amplifiers (OTAs), these devices are suitable substitutes for the operational amplifiers used in the earlier implementation.

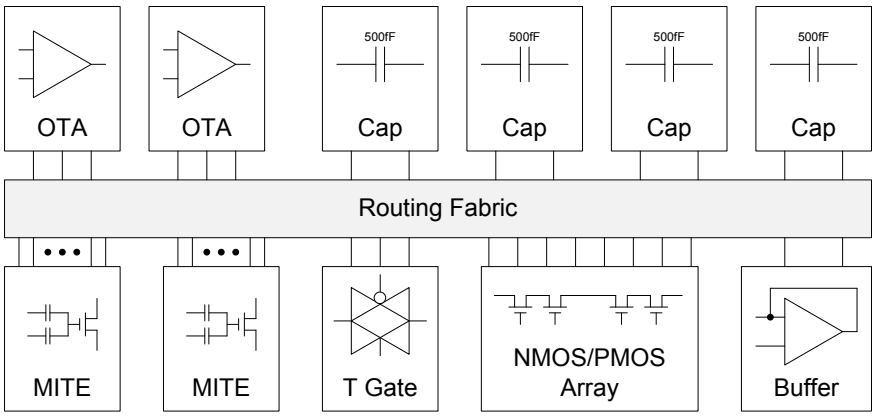


Figure 5.2. Component schematic of CAB_1 on the RASP 2.8 FPAA.

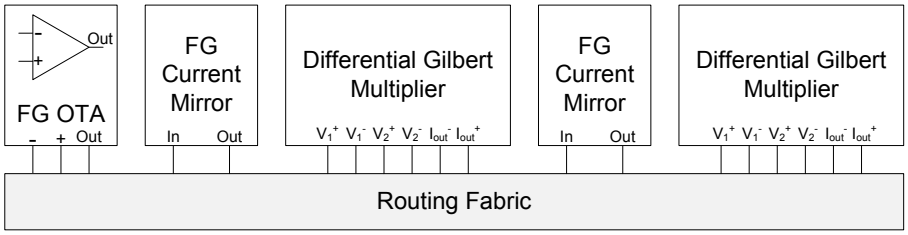


Figure 5.3. Component schematic of CAB_2 on the RASP 2.8 FPAA.

5.4.1 Overall Description of the Hardware

An overall hardware schematic of the proposed second-order section implementation is shown in Figure 5.4. The hardware shown in Figure 5.4 is used to implement nine functions. These functions are:

1. sampling and storing the input,
2. selecting the appropriate sampled and delayed inputs to be summed together for the $(l+1)^{\text{th}}$ distributed arithmetic computational cycle,
3. sampling and storing the output,
4. selecting the appropriate stored and delayed outputs to be summed together for the $(l+1)^{\text{th}}$ distributed arithmetic computational cycle,
5. delaying and storing the accumulation term for one computational cycle,

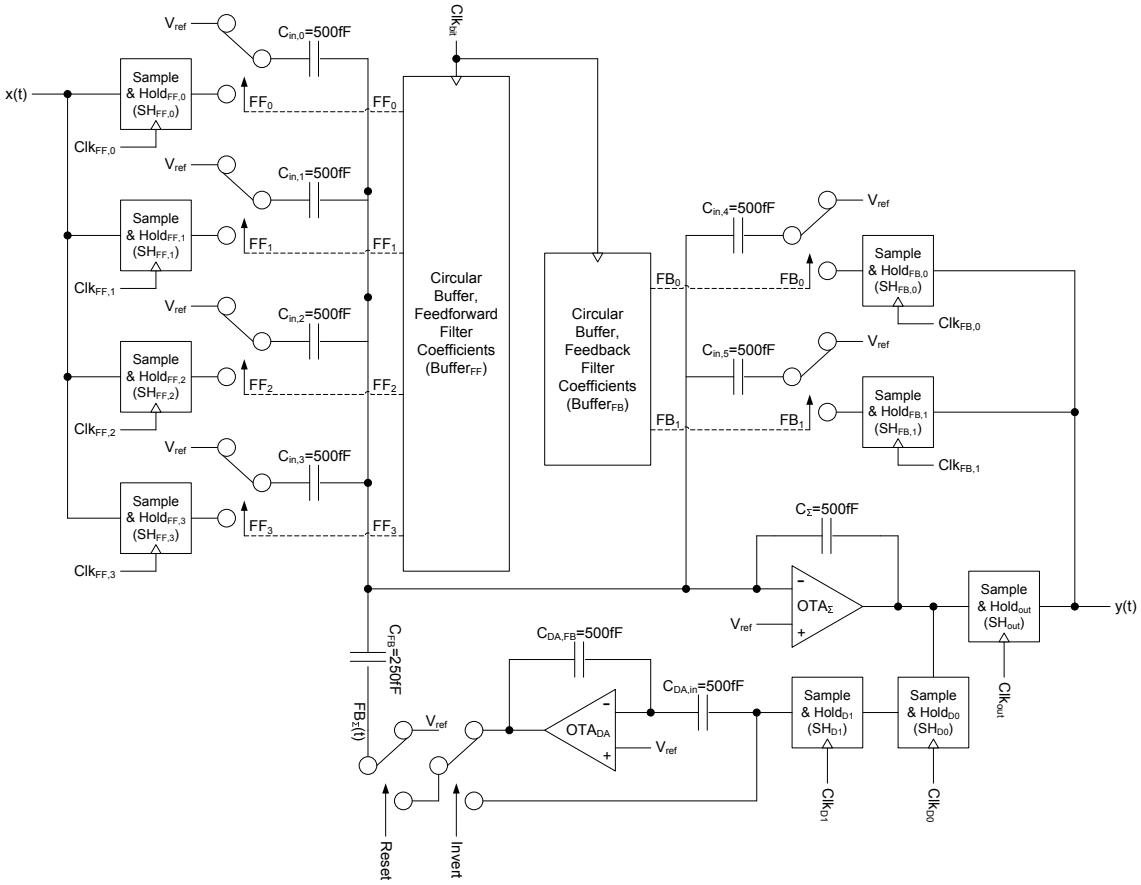


Figure 5.4. Component schematic of the mixed-signal second-order section.

6. inverting the accumulation term,

7. selecting if the feedback term is set to either the non-inverted or inverted accumulation term,

8. selecting if the feedback summation term is set to either the reference voltage or feedback term, and

9. summing the appropriately selected input and output samples together with half of the feedback summation term.

These functions are used to generate the output signal, $y(t)$, as necessitated by the distributed arithmetic mechanization of a second-order section. An overall description of how these functions are used to generate the output is given in the next section. For more details about the individual hardware components used in the schematic, they are located in Section 5.4.3. For more details about the hardware timing, they are located in Section 5.4.4.

5.4.2 Overall Description of the Hardware Operation

The output, $y[n]$, is generated in the following fashion. At the start of the distributed arithmetic computational cycle, two actions occur. First, one of the input sample and holds, $SH_{FF,i}, i = 0, 1, 2, 3$, is sampling the input signal, $x(t)$. The other is setting the feedback summation term, $FB_{\Sigma}(t)$, to the reference voltage, V_{ref}, by using the two-position switch controlled by the Reset signal. For most of the first computational cycle, the Reset signal is set to the supply rail of $2.4V$. This signaling level means that the feedback summation term $FB_{\Sigma}(t)$ is set to the reference voltage V_{ref}.

During the first computational cycle, the hardware is computing the partial products associated with the least significant bit, $l = B - 1$. The circular buffer $Buffer_{FF}$ determines which input samples are needed for the current feedforward partial product. It generates outputs FF_0 through FF_3. These outputs are used to control the two-position switches that connect either the reference voltage V_{ref} or the output of $SH_{FF,i}$ to the capacitor $C_{in,i}$ for

$i = 0, 1, 2, 3$.

The circular buffer $Buffer_{FB}$ determines which output samples are needed for the current feedback partial product. It generates outputs FB_0 through FB_1. These outputs are used to control the two-position switches that connect either the reference voltage or the output of $SH_{FB,i}$ to the capacitor $C_{in,i+3}$ for $i = 0, 1$.

The selected input and output samples are summed together and inverted using the operational transconductance amplifier OTA_Σ in an inverting configuration utilizing equal sized feedback and input capacitors, C_Σ and $C_{in,i}$ for $i = 0, 1, 2, 3, 4, 5$, to form the first accumulation term. The size of these capacitors is $500fF$. Unlike in the digital domain where an adder tree constructed of multiple adders would be needed, the multiple summations needed to form the l^{th} feedforward and feedback partial products and to add them together with half of the feedback summation term $FB_\Sigma(t)$ are performed by a single amplifier. At this point, the accumulation term is equal to $-\sum_{i=0}^{2} b_{i(B-1)}x[n-i] - \sum_{j=1}^{2} a_{j(B-1)}y[n-j]$.

The first accumulation term is then sampled by the sample and hold SH_{D0}. After the sample and hold SH_{D0} has finished sampling, its output is sampled by the sample and hold SH_{D1}. Once the sample and hold SH_{D1} has finished sampling, the first accumulation term has been successfully stored and delayed by one computational cycle through the use of sample and holds SH_{D0} and SH_{D1}.

Next, the inverse of the accumulation term is computed using the operational transconductance amplifier, OTA_{DA}. It is calculated using the amplifier OTA_{DA} in an inverting configuration utilizing equal sized feedback and input capacitors, $C_{DA,FB}$ and $C_{DA,in}$. The size of these capacitors is $500fF$. The input into the inverter is the delayed accumulation term, which is stored in and is the output of the sample and hold SH_{D1}, coupled through the input capacitor $C_{DA,in}$.

The inverted delayed accumulation term, which is the output of the inverting amplifier OTA_{DA}, and the non-inverted delayed accumulation term, which is the output of the sample and hold SH_{D1}, are fed as input into the two-position switch controlled by the Invert

signal. For all the computational cycles except for the last one, the Invert signal is set to ground. This signaling level means that the delayed accumulation term is to be inverted. By selecting the inverted delayed accumulation term, the output of the two-position switch controlled by the Invert signal is equal to $\sum_{i=0}^{2} b_{i(B-1)}x[n-i] + \sum_{j=1}^{2} a_{j(B-1)}y[n-j]$. This output is the one expected by Eq. (5.4) for a second-order section IIR filter after the first computational cycle.

Slightly before the start of the second computational cycle, the Reset signal is set to ground. This signaling level means that the feedback summation term $FB_{\Sigma}(t)$ is set to the output of the two-position switch controlled by the Invert signal. Presently, the output of the two-position switch controlled by the Invert signal is equal to the inverted delayed accumulation term; therefore, the output of the two-position switch controlled by the Reset signal is also equal to the inverted delayed accumulation term.

At the start of the second computational cycle, the circular buffers $Buffer_{FF}$ and $Buffer_{FB}$ advance their outputs to the next bit slice. Now, the second feedforward partial product, which is equal to $\sum_{i=0}^{2} b_{i(B-2)}x[n-i]$, the second feedback partial product, which is equal to $\sum_{j=1}^{2} a_{j(B-2)}y[n-j]$, and half of the feedback summation term $FB_{\Sigma}(t)$, which is equal to $\sum_{i=0}^{2} b_{i(B-1)}x[n-i] + \sum_{j=1}^{2} a_{j(B-1)}y[n-j]$, are summed together and inverted using the amplifier OTA_{Σ}. At this point, the accumulation term is equal to $-(\sum_{i=0}^{2} b_{i(B-2)}x[n-i]) - (\sum_{j=1}^{2} a_{j(B-2)}y[n-j]) - (\sum_{i=0}^{2} b_{i(B-1)}x[n-i] + \sum_{j=1}^{2} a_{j(B-1)}y[n-j])/2$.

The ratio of the input capacitors $C_{in,i}$ for $i = 0, 1, 2, 3, 4, 5$ and C_{FB} to the feedback capacitor C_{Σ} determines the inverted gain of the input signal as referenced to V_{ref}. For all input signals except the feedback summation term, the inverted gain is equal to -1; therefore, the size of the input capacitors $C_{in,i}$ for $i = 0, 1, 2, 3, 4, 5$ is equal to the size of the feedback capacitor C_{Σ}. To achieve the required inverted gain of -0.5 for the input signal $FB_{\Sigma}(t)$, the size of its input capacitor C_{FB} is equal to half the size of the feedback capacitor C_{Σ}. For this implementation, the size of C_{Σ} is $500fF$; therefore, the size of $C_{in,i}$ for $i = 0, 1, 2, 3, 4, 5$ is $500fF$, and the size of C_{FB} is $250fF$. Since no $250fF$ capacitors

are available on the RASP 2.8 FPAA, an equivalent capacitance is constructed from two $500fF$ capacitors connected in series.

The rest of the second computational cycle is completed as before in the first one. The current accumulation term is stored and delayed by the sample and holds SH_{D0} and SH_{D1}. The delayed accumulation term is inverted by the inverting amplifier OTA_{DA}, and the inverted delayed accumulation term is passed through the two-position switch controlled by the Invert signal into one of the inputs of the two-position switch controlled by the Reset signal.

For the computation and for the scaled accumulation of the remaining partial products until the last one, the computational flow remains unchanged from the second one. During the last computational cycle, the hardware is computing the partial products associated with the most significant bit, $l = 0$. At the start of the last computational cycle, the Invert signal is set to the supply rail of $2.4V$. This signaling level means that the non-inverted delayed accumulation term is connected to one of the inputs of the two-position switch controlled by the Reset signal. Remember, the Reset signal is still set to ground; therefore, the feedback summation term $FB_{\Sigma}(t)$ is equal to the non-inverted delayed accumulation term. In other words, $FB_{\Sigma}(t)$ is equal to $-(\sum_{l=1}^{B-1}[\sum_{i=0}^{2} b_{il}x[n-i]]2^{-l} + \sum_{l=1}^{B-1}[\sum_{j=1}^{2} a_{jl}y[n-j]]2^{-l})$ or to $-(\sum_{l=1}^{B-1}\{\sum_{i=0}^{2} b_{il}x[n-i] + \sum_{j=1}^{2} a_{jl}y[n-j]\}2^{-l})$. Although the sign of this term does not agree with Eq. (5.4), this inconsistency is corrected when half of the feedback summation term is inverted by the amplifier OTA_{Σ}.

In a similar manner as before, the feedforward and feedback partial products associated with the most significant bit $l = 0$ and half of the feedback summation term $FB_{\Sigma}(t)$ are summed together using the amplifier OTA_{Σ}. Now that the computation is complete, the output of second-order section, which is the output of the amplifier OTA_{Σ}, is sampled by the sample and hold, SH_{out}.

Once the output of the second-order section is sampled and before the start of the next sample period, the output of the sample and hold SH_{out} is sampled by one of the output

sample and holds, $SH_{FB,i}$ for $i = 0, 1$. The sample and holds $SH_{FB,i}$ are used to store the sampled and delayed outputs, $y[n-1]$ and $y[n-2]$, which are used in the computation of the feedback partial products for next sample period.

5.4.3 Description of the Hardware Subcomponents

There are several components used to implement the proposed mixed-signal distributed arithmetic second-order section filter. A description of each subcomponent and which functions they are mapped to are given below. Note, some functions require more than one subcomponent.

5.4.3.1 Implementation of the Sample and Hold Circuit

For any function that requires sampling a signal, either the input, the output, or the accumulation term, a sample and hold circuit needs to be implemented. The functions that need a sample and hold circuit are the functions enumerated 1, 3, and 5. For this research, a relatively simple sample and hold circuit was chosen. This design only requires an OTA, a capacitor, and a transmission gate. All of these components are available on the RASP 2.8 FPAA. However, the OTA required needs to be able to source $40\mu A$, and the maximum current that a standard RASP 2.8 FPAA OTA can source is $10\mu A$. To overcome this limitation, four standard RASP 2.8 FPAA OTAs are connected in parallel. In essence, an OTA with transistor sizings that is four times that of the standard RASP 2.8 FPAA OTA was constructed. These components are connected as illustrated in Figure 5.5.

As shown in Figure 5.5, the OTA is connected in a buffering configuration whose reference voltage is tied to the charge stored on the capacitor. This capacitor is connected to the input through a single position switch. The functionality of the single position switch is implemented using a transmission gate as shown in Figure 5.6.

To overcome the charge injection issues, the straight forward solution of using larger capacitors was used. In this case, the size of the sample and hold circuit used in this implementation of a mixed-signal distributed arithmetic second-order section is $9pF$. Although

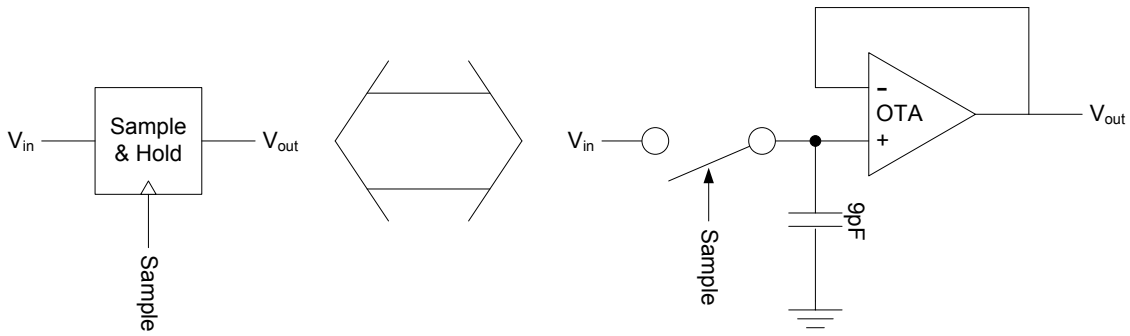


Figure 5.5. Component schematic of the sample and hold circuit.

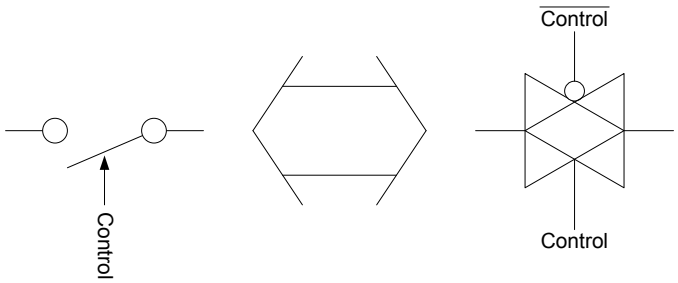


Figure 5.6. Component schematic of the 1-position switch.

the RASP 2.8 FPAA does not have a $9pF$ capacitor, eighteen of the $500fF$ capacitors are connected in parallel to achieve the necessary capacitance. When used in the final SOS implementation, the error is less than $1mV$.

5.4.3.2 Implementation of the Feedforward Circular Buffer

A description of the hardware used to store the feedforward filter coefficients is described in this section. This component is needed by function number 2 listed at the beginning of Section 5.4.1 whose function is to sum the sampled input according the selected digital coefficient bit slice. This component is used to control which input samples, $\{x[n]\ x[n-1]\ x[n-2]\}$, are needed for the l^{th} partial product in the DA computation of the output, $y[n]$. In other words, this unit determines which input samples are needed to compute the summation $\sum_{i=0}^{K-1} b_{il}x[n-i]$ during the l^{th} cycle of the DA computation. In this implementation for a second-order section, the four most recent input samples are stored. Because this component needs to control four input sources, four B-bit shift registers are needed. The shift out of the preceding shift register is connected to the shift in of the next one. The shift registers are connected in a circular fashion, which mean the shift out of the last unit is connected to the shift in of the first one. The first shift register is initialized with zeros. The rest of the shift registers are initialized in reverse feedforward filter coefficient order such that the last unit is initialized with filter coefficient $w_b[0]$. A block diagram for this component is shown in Figure 5.7.

Although this unit could be implemented on the FPAA, it would be more reasonable to implement this component on a digital exclusive prototyping platform such as an FPGA. The functionality of this component could also be done on a microcontroller and a 4-bit general purpose I/O port.

5.4.3.3 Implementation of the Feedback Circular Buffer

The feedback circular buffer is used to control which output samples, $\{y[n-1]\ y[n-2]\}$, are needed for the l^{th} partial product in the DA computation of the output, $y[n]$. In other words, this unit determines which output samples are needed to compute the summation

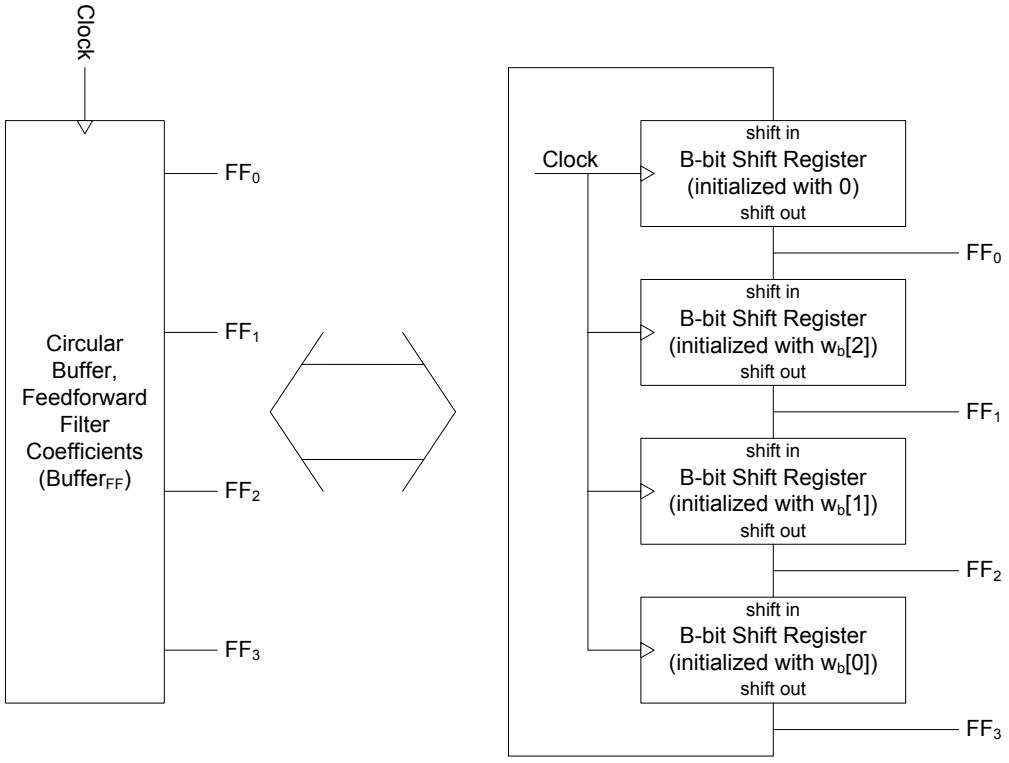


Figure 5.7. Component schematic of the circular buffer for the feedforward filter coefficients.

$\sum_{j=1}^{L-1} a_{jl} y[n-j]$ during the l^{th} cycle of the DA computation. This component is needed by function number 4. In this implementation for a second-order section, the two most recent output samples are stored. Because this component needs to control two input sources, two B-bit shift registers are needed. These components are connected just as they were in the feedforward case. The shift registers are initialized in reverse feedback filter coefficient order such that the first unit is initialized with $w_a[2]$ and the last unit is initialized with filter coefficient $w_a[1]$. A block diagram for this component is shown in Figure 5.8.

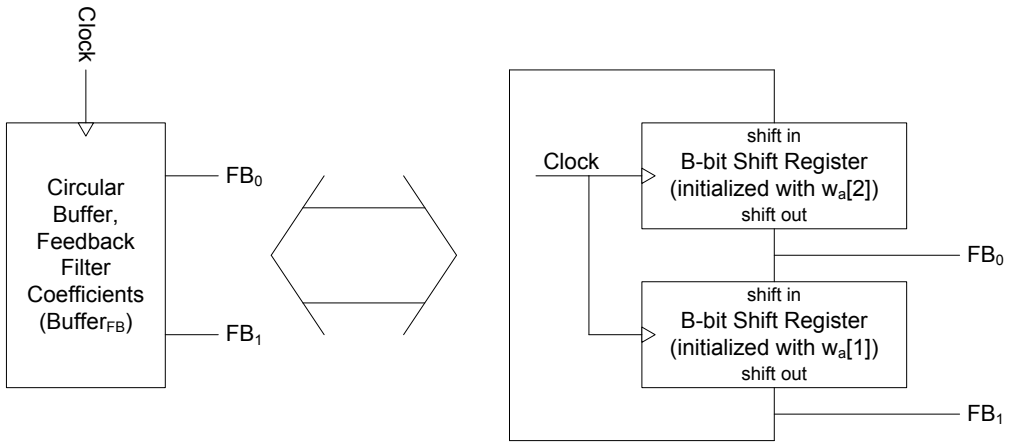


Figure 5.8. Component schematic of the circular buffer for the feedback filter coefficients.

Although this unit could be implemented on the FPAA, it would be more reasonable to implement this component on a digital exclusive prototyping platform such as an FPGA. The functionality of this component could also be done on a microcontroller and a 2-bit general purpose I/O port.

5.4.3.4 Implementation of the Two-position Switch

The two-position switch is used to select between two input signals dependent on the signaling level of the control signal. This component is used for function numbers 2, 4, 7, and 8. It is composed of two transmission gates connected in parallel. At all times, one of the two transmission gates is transparent; however, only one transmission gate is transparent at the same time. These transmission gates are controlled by a single control signal.

One transmission gate is transparent when the control signal is set to the supply rail of $2.4V$, and the other one is transparent when the control signal is set to ground. Since this transmission gate is fully complementary, which is that it is composed of both NMOS and PMOS transistors, the control signal and its inverse is needed to control the transparency of the transmission gate. The control signal and its inverse is connected to the transmission gates as shown in Figure 5.9.

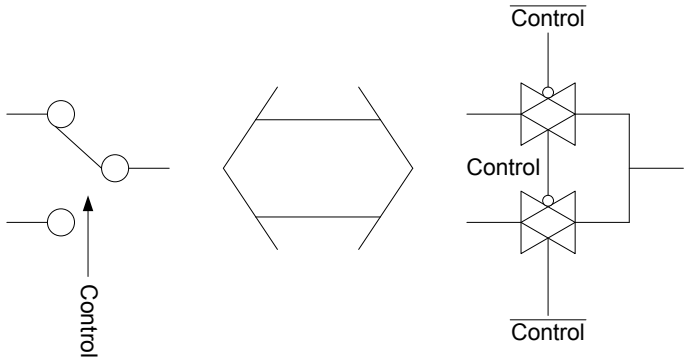


Figure 5.9. Component schematic of the 2-position switch.

5.4.3.5 Implementation of the Operational Transconductance Amplifier OTA_{Σ}

As mentioned earlier, the RASP 2.8 FPAA has operational transconductance amplifiers available to use. However, the maximum current that each OTA can source is limited to around $10\mu A$. Since this OTA is connected to several components, it needs to be able to provide around $260\mu A$, which is more than $10\mu A$. To achieve the required amperage, twenty-six of the standard RASP 2.8 FPAA OTAs are connected in parallel. By connecting the OTAs in this fashion, it has the same effect as constructing the standard RASP 2.8 FPAA OTA out of transistors that are twenty-six times larger than originally specified.

5.4.3.6 Implementation of the Operational Transconductance Amplifier OTA_{DA}

For this amplifier, the ability to source around $40\mu A$ of current is required. Like before for the amplifier OTA_{Σ}, four of the standard RASP 2.8 FPAA OTAs are connected in parallel to achieve the necessary amperage. By connecting the OTAs in this fashion, it has the same

effect as constructing the standard RASP 2.8 FPAA OTA out of transistors that are four times larger than originally specified.

5.4.4 Description of the Hardware Timing

To help follow the overall hardware timing of the proposed second-order section implementation, a timing diagram is shown in Figure 5.10. The distributed arithmetic operation starts at the beginning of the sample period. In this case, this event is the rising edge of the sample clock. Note in the implementation, no devices are driven by the sample clock; however, all timing events are relative to this clock.

A signal called the bit clock, Clk_{bit}, is used to clock individual DA computational cycles. The duration of a computational cycle is equivalent to the period of the bit clock. For every sample clock period, B computational cycles occur. Recall, B is the bit precision of the filter coefficients. Therefore, the bit clock is B times the sample frequency. In the Figure 5.10, the bit precision of the filter coefficients is four. Note, the bit clock and the sample clock are in phase, and the rising edge of the first computational cycle is aligned with the rising edge of the sample clock. The timing details for the other signals are given in the sections below.

5.4.4.1 Timing of $Clk_{FF,i}$ for the Input Sample and Holds $SH_{FF,i}$

Four sampling clocks, $Clk_{FF,i}$ for $i = 0, 1, 2, 3$, are used to control when each input sample and hold, $SH_{FF,i}$ for $i = 0, 1, 2, 3$, samples the input signal, $x(t)$. The sample and holds $SH_{FF,i}$ for $i = 0, 1, 2, 3$ samples the input when their respective sampling clock $Clk_{FF,i}$ for $i = 0, 1, 2, 3$ is high and holds when the sampling clock is low. In this implementation, each sample and hold only samples once every four sample periods. The sampling clocks $Clk_{FF,i}$ for $i = 1, 2, 3$ are the same as the sampling clock $Clk_{FF,0}$ except that the rising edges of $Clk_{FF,i}$ for $i = 1, 2, 3$ are separated from the rising edge of $Clk_{FF,0}$ by i sample period(s).

The rising edges of $Clk_{FF,i}$ for $i = 0, 1, 2, 3$ occurs in phase with the rising edge of the sample clock. The $Clk_{FF,i}$ for $i = 0, 1, 2, 3$ is high for a duration longer than the sampling

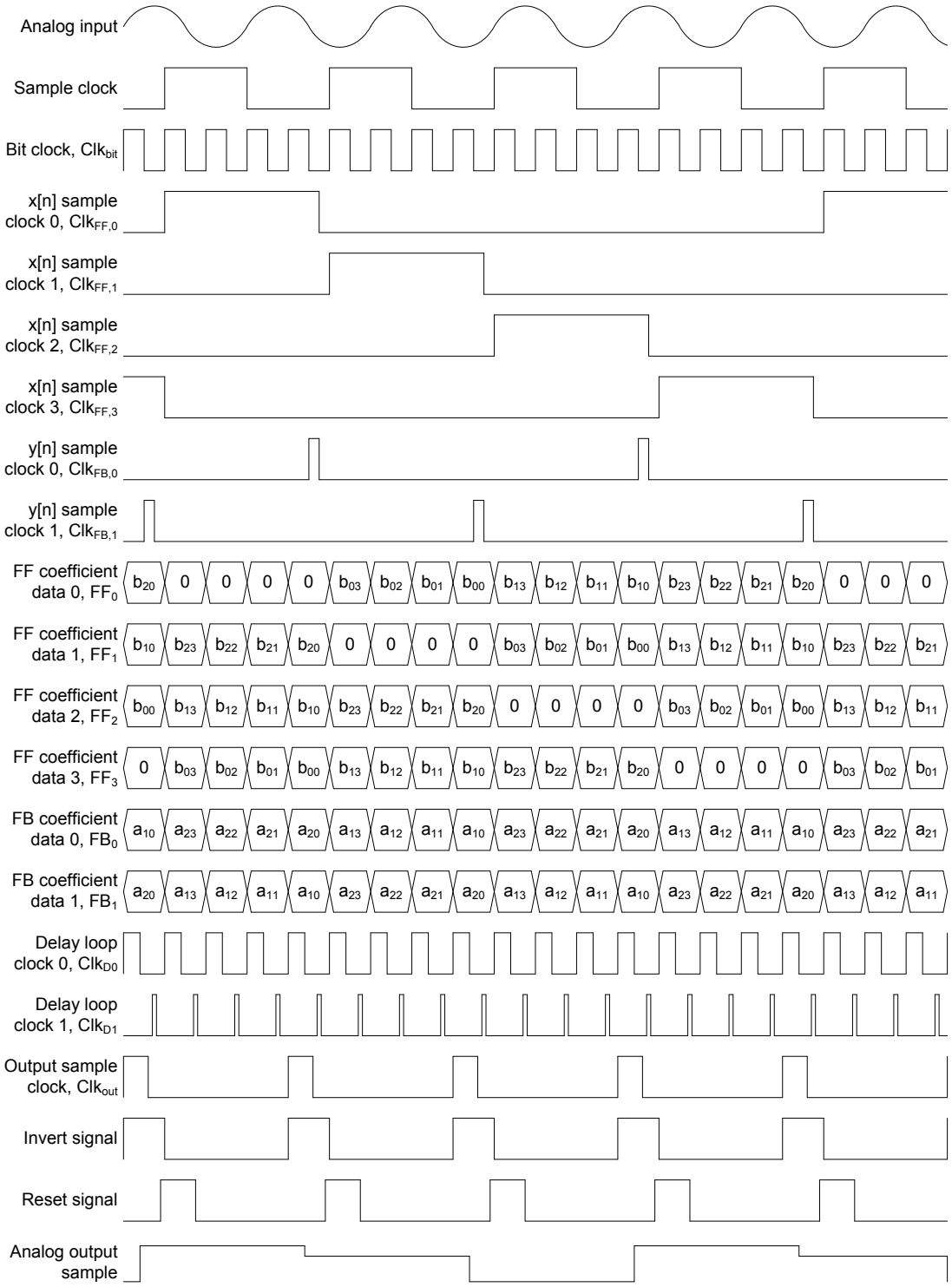


Figure 5.10. Timing diagram for the proposed second-order section implementation using distributed arithmetic.

settling time and for a duration shorter than the sampling period. The sampling of the input must be completed before the next sampling event minus its holding settling time. A timing diagram of $Clk_{FF,i}$ for $i = 0, 1, 2, 3$ is shown in Figure 5.11.

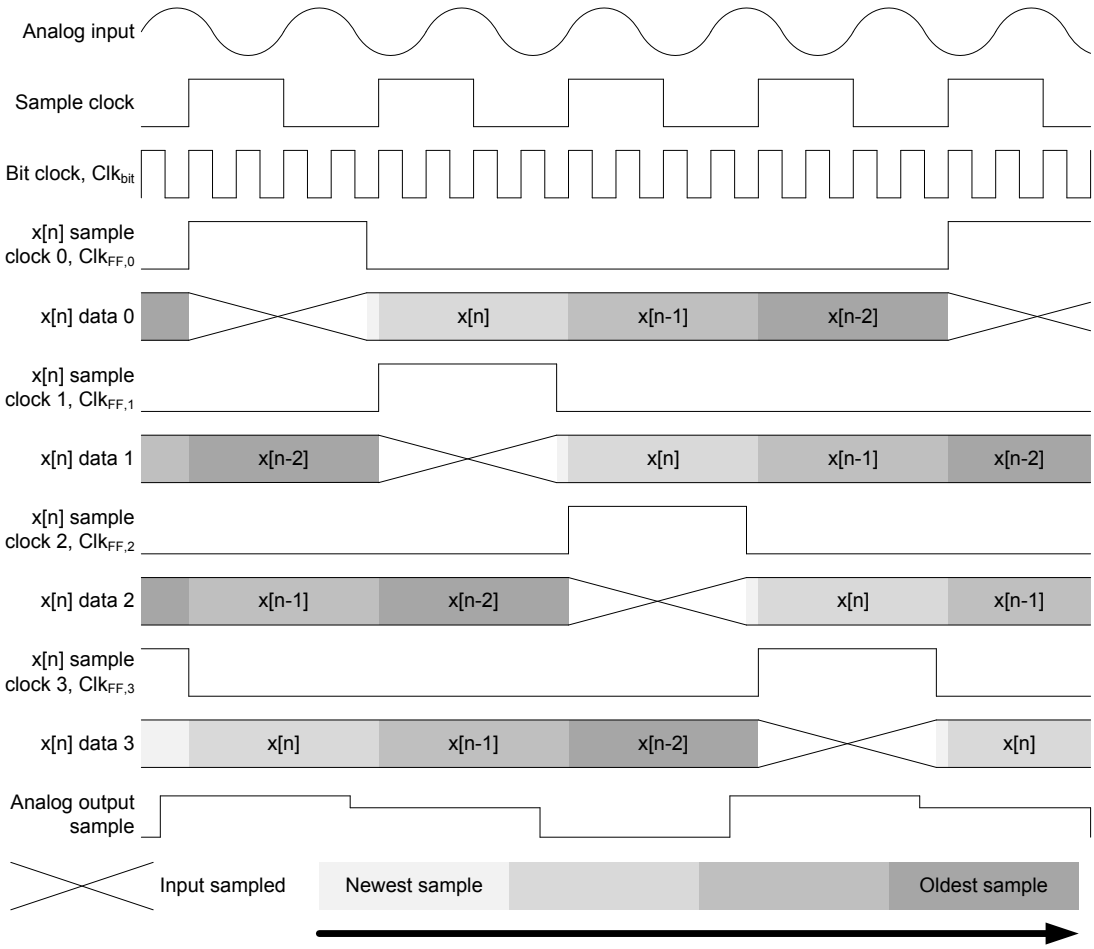


Figure 5.11. A time lapse diagram showing how the sampled input data changes.

Since the input samples, $x[n - i]$ for $i = -1, 0, 1, 2$, are not sampled using a sample and hold array configured like a serial shift register where the input of the i^{th} sample and hold is connected to the output of the $(i-1)^{\text{th}}$ one, the association of each sample and hold with the input samples changes as new input data is collected. Figure 5.11 is a time lapse diagram of these changing relationships.

When an input sample and hold first samples the input signal, this sample is labeled as

$x[n+1]$. Although this second-order section implementation is casual, processing using this data does not begin until the start of the next sample period. This additional sample period of latency is used to give the input sample and hold ample time to sample the input signal and to not dedicate any computation time for sampling the input. Note, this sampling method allows for higher computational rates. Since this input sample remains in its original sample and hold, this sample becomes $x[n]$ when the next sample period begins. After i sample periods from the initial sampling event, the input sample stored in this sample and hold is $x[n-i+1]$. This process of temporal aging is used for all input sample and holds.

Other sampling approaches do exist. This method was selected because it minimizes the number of sampling events and limits the amount of sampling error per input sample due to a sampling event to a single instance rather than multiple ones.

5.4.4.2 Timing of $Clk_{FB,j}$ for the Output Sample and Holds $SH_{FB,j}$

Two sampling clocks, $Clk_{FB,j}$ for $j = 0, 1$, are used to control when each output sample and hold, $SH_{FB,j}$ for $j = 0, 1$, samples the output signal, $y(t)$. The sample and holds $SH_{FB,j}$ for $j = 0, 1$ samples the input when their respective sampling clock $Clk_{FB,j}$ for $j = 0, 1$ is high and holds when the sampling clock is low. In this implementation, each sample and hold only samples once every two sample periods. The sampling clock $Clk_{FB,1}$ is the same as the sampling clock $Clk_{FB,0}$ except that the rising edge of $Clk_{FB,1}$ is separated from the rising edge of $Clk_{FB,0}$ by one sample period(s).

$Clk_{FB,j}$ for $j = 0, 1$ can go high and begin sampling the output anytime during the last computational cycle. For this implementation, the sampling clock $Clk_{FB,j}$ for $j = 0, 1$ goes high slightly before SH_{out} has finished sampling the output sample, $y[n]$. However, the rising edge of $Clk_{FB,j}$ for $j = 0, 1$ should occur early enough that this switching event does not affect the sampling of the output sample $y[n]$ by SH_{out}.

When the clock $Clk_{FB,j}$ for $j = 0, 1$ goes low and begins to hold the output signal, this timing event is the more crucial one. Remember, the sample and holds $SH_{FB,j}$ for $j = 0, 1$ are sampling the output of SH_{out} and cannot finish sampling until SH_{out} has finished

sampling $y[n]$ and its output has settled. Also, $Clk_{FB,j}$ for $j = 0, 1$ must remain high longer than its sampling settling time. Therefore, the falling edge of $Clk_{FB,j}$ for $j = 0, 1$ cannot occur for a period equivalent to the holding settling time of the sample and hold SH_{out} plus the sampling settling time of its sample and hold $SH_{FB,j}$ after the falling edge of the sampling clock, Clk_{out}, for SH_{out}. In addition, the falling edge of $Clk_{FB,j}$ for $j = 0, 1$ must occur before the next sampling event minus its holding settling time. A timing diagram of $Clk_{FB,j}$ for $j = 0, 1$ is shown in Figure 5.12.

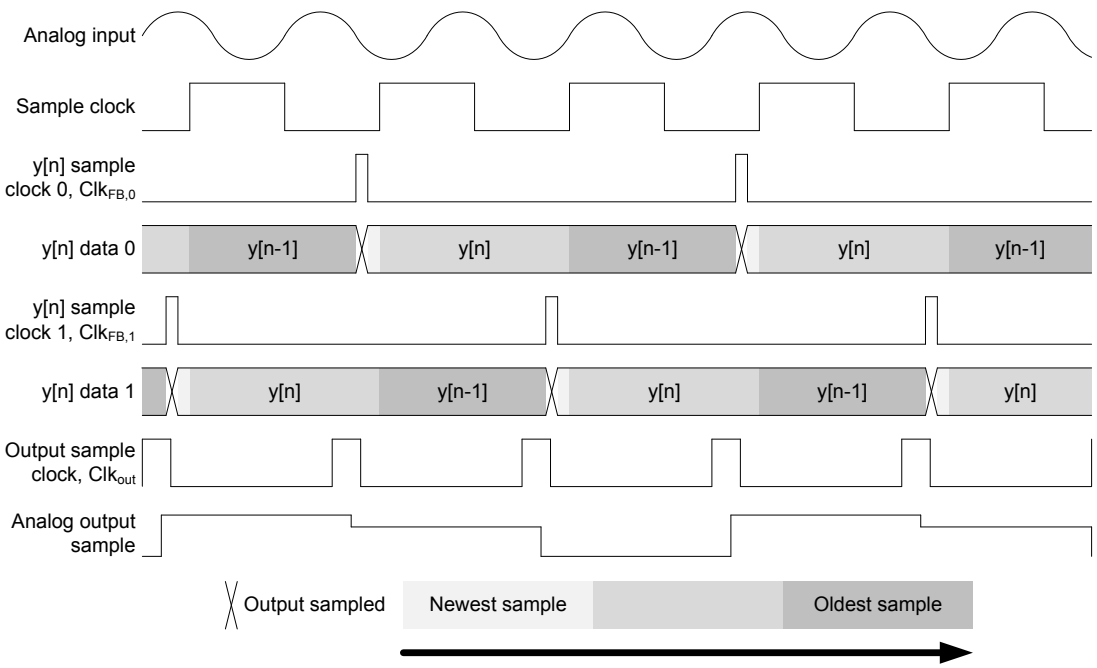


Figure 5.12. A time lapse diagram showing how the sampled output data changes.

Since the output samples, $y[n - j]$ for $j = 1, 2$, are not sampled using a sample and hold array configured like a serial shift register, the association of each sample and hold with the output samples changes as new output data is collected. Figure 5.12 is a time lapse diagram of these changing relationships.

When an output sample and hold first samples the output signal, this sample is labeled as $y[n]$. Unlike in the input sample case, the sampling of the output signal cannot be delayed without consequence. If the use of this output sample was delayed by one sample

period, then the difference equation for the output sample $y[n]$ becomes $\sum_{i=0}^{K-1} w_b[i]x[n-i]+\sum_{j=1}^{L-1} w_a[j]y[n-(j+1)]$, which is not the same as the desired difference equation of Eq. (5.1) for a second-order section filter. Because the temporal age of the output sample is relative to when $y[n]$ was sampled, the output sample $y[n]$ must be used immediately at the start of the next sample period as $y[n-1]$. After j sample periods from the initial sampling event, the output sample stored in this sample and hold is $y[n-j]$. This process of temporal aging is used for all output sample and holds $SH_{FB,j}$ for $j = 0, 1$. Like in the input sampling case, this method was selected because it minimizes the number of sampling events and limits the amount of sampling error per output sample $y[n-j]$ for $j = 0, 1$ due to a sampling event to a single instance rather than multiple ones.

5.4.4.3 Timing of FF_i for the Circular Buffer $Buffer_{FF}$

The signals FF_i for $i = 0, 1, 2, 3$ of the circular buffer $Buffer_{FF}$ are used to control which input samples $x[n-i]$ for $i = 0, 1, 2, 3$ are required for computing the current feedforward partial product. As mentioned earlier, the feedforward partial product associated with the least significant bit slice, which is the concatenation of the feedforward filter coefficients LSBs, is computed during the first computational cycle, while the feedforward partial product associated with the most significant bit slice, which is the concatenation of the feedforward filter coefficients MSBs, is computed during the last computational cycle. Each bit slice is composed of four bits. One bit for each input sample and hold. Note for a second-order section filter, only three of the input sample and holds are required for computing the output sample $y[n]$; therefore, the bit associated with the input sample and hold that is sampling the input signal $x(t)$ is set to zero. Also notice in Figure 5.10, the feedforward filter coefficients plus a zero are being rotated in a circular fashion to match the appropriate input sample and hold with the correct feedforward filter coefficient. Remember in this implementation, the feedforward filter coefficients are moved to the correct input samples rather than the other way around.

For the timing of the signals FF_i for $i = 0, 1, 2, 3$, a new feedforward bit slice is outputted on these signals at the start of every computational cycle. At the beginning of the DA computation, FF_i for $i = 0, 1, 2, 3$ outputs the feedforward bit slice associated with the least significant bit, $l = B - 1$, at the rising edge of the sample clock. Each subsequent feedforward bit slice is outputted on signals FF_i for $i = 0, 1, 2, 3$ at the rising edge of the bit clock Clk_{bit}. The feedforward bit slice associated with the most significant bit, $l = 0$, is outputted on these signals at the start of the last, B^{th}, computational cycle or $B - 1$ bit periods after the rising edge of the sample clock.

5.4.4.4 Timing of FB_j for the Circular Buffer $Buffer_{FB}$

The signals FB_j for $j = 0, 1$ of the circular buffer $Buffer_{FB}$ are used to control which output samples $y[n - j]$ for $j = 0, 1$ are required for computing the current feedback partial product. As mentioned earlier, the feedback partial product associated with the least significant bit slice, which is the concatenation of the feedback filter coefficients LSBs, is computed during the first computational cycle, while the feedback partial product associated with the most significant bit slice, which is the concatenation of the feedback filter coefficients MSBs, is computed during the last computational cycle. Each bit slice is composed of two bits. One bit for each output sample and hold. Notice in Figure 5.10, the feedback filter coefficients are being rotated in a circular fashion to match the appropriate output sample and hold with the correct feedback filter coefficient. Remember in this implementation, the feedback filter coefficients are moved to the correct output samples rather than the other way around.

For the timing of the signals FB_j for $j = 0, 1$, a new feedback bit slice is outputted on these signals at the start of every computational cycle. At the beginning of the DA computation, FB_j for $j = 0, 1$ outputs the feedback bit slice associated with the least significant bit, $l = B - 1$, at the rising edge of the sample clock. Each subsequent feedforward bit slice is outputted on signals FB_j for $j = 0, 1$ at the rising edge of the bit clock Clk_{bit}. The feedback bit slice associated with the most significant bit, $l = 0$, is outputted on these signals at

the start of the last, B^{th}, computational cycle or $B-1$ bit periods after the rising edge of the sample clock.

5.4.4.5 *Timing of Clk_{D0} for the Sample and Hold SH_{D0}*

The sampling clock Clk_{D0} is used to control when the sample and hold SH_{D0} samples the output signal $y(t)$. The accumulation term is sampled when Clk_{D0} is high, and the sampled voltage is held in the sample and hold SH_{D0} when Clk_{D0} is low.

The sample and hold SH_{D0} is the first sample and hold in a pair of sample and holds that are connected in series. The purpose of these sample and holds is to store and to delay the accumulation term. Two sample and holds with non-overlapping sampling phases are needed to prevent transparency in the distributed arithmetic feedback path. If this path was transparent, then an unintended feedback in the DA computation would occur, and the accumulation term would grow uncontrollably.

To store and to delay the accumulation term properly, both the sample and holds must be allowed to sample its input in the proper sequence within one computational cycle. To allow SH_{D1} enough time to sample, SH_{D0} begins sampling its input immediately. In other words, the rising edge of Clk_{D0} occurs at the same time as the rising edge of the bit clock Clk_{bit}. Then, the sample and hold SH_{D0} must sample its input long enough to allow its output to settle. Also, SH_{D0} must finish sampling early enough to allow SH_{D1} to wait for at least the holding settling time of SH_{D0} before sampling its input, to allow SH_{D1} to sample its input, and to allow SH_{D1} to settle its output after entering its holding phase. In other words, the falling edge of Clk_{D0} occurs at least a period of the sampling settling time for SH_{D0} after the rising edge of Clk_{D0}, occurs at least a period of the holding settling time for its sample and hold before the rising edge of Clk_{D1}, and occurs at least a period of the holding settling time for its sample and hold plus the sampling and holding settling times for SH_{D1} before the rising edge of Clk_{bit}. This timing pattern for Clk_{D0} repeats every computational cycle.

5.4.4.6 Timing of Clk_{D1} for the Sample and Hold SH_{D1}

The sampling clock Clk_{D1} is used to control when the sample and hold SH_{D1} samples the output signal $y(t)$. The output signal $y(t)$ is sampled when Clk_{D1} is high, and the sampled voltage held in the sample and hold SH_{D1} when Clk_{D1} is low.

The sample and hold SH_{D1} is the second sample and hold in the pair that store and delay the accumulation term. To prevent transparency in the DA feedback path, SH_{D1} cannot sample at the same time as SH_{D0} and can only begin to sample the output of SH_{D0} after SH_{D0} has finished sampling the accumulation term plus some holding settling time of SH_{D0}. This requirement means that the rising edge of the sampling clock Clk_{D1} is at least for a period of the holding settling time of SH_{D0} after the falling edge of Clk_{D0}. Also, the sample and hold SH_{D1} must finish sampling for a period of its holding settling time before the start of the next computational cycle, and the sampling phase of SH_{D1} must be long enough to allow its output to settle. This requirement means that the falling edge of Clk_{D1} occurs at least a period of the sampling settling time for SH_{D1} after the rising edge of Clk_{D1} and occurs at least a period of the holding settling time for SH_{D1} before the rising edge of Clk_{bit}. This timing pattern for Clk_{D1} repeats every computational cycle.

5.4.4.7 Timing of Clk_{out} for the Sample and Hold SH_{out}

The sampling clock Clk_{out} is used to control when the sample and hold SH_{out} samples the output signal $y(t)$. This sampled voltage is the output sample $y[n]$ and is used by the sample and holds $SH_{FB,j}$ for $j = 0, 1$ to generate the delayed output samples, $y[n - (j + 1)]$ for $j = 0, 1$. The output signal $y(t)$ is sampled when Clk_{out} is high, and the sampled voltage held in the sample and hold SH_{out} when Clk_{out} is low.

The periodicity of the sampling clock Clk_{out} is one period of the sample clock. The sampling clock Clk_{out} is active high and is sampling the output signal $y(t)$ only during the last, B^{th}, computational cycle. For all other computational cycles, Clk_{out} is low. The rising edge of the sampling clock Clk_{out} is aligned with the rising edge of the B^{th} computational cycle. The B^{th} computational cycle begins $B - 1$ bit periods after the start of the DA

computation, which is the rising edge of the sample clock. Clk_{out} must remain high long enough such that the output of the amplifier OTA_{Σ} has become settled after the last bit slice has been shifted into the computation of the last feedforward and feedback partial products by the circular buffers $Buffer_{FF}$ and $Buffer_{FB}$. The sample and hold SH_{out} must finish sampling the output signal $y(t)$ before the sample and hold SH_{D1} begins sampling the output of SH_{D0} and must allow enough time for the holding settling time of its output. Therefore, the falling edge of Clk_{out} must occur for a period of the holding settling time of SH_{out} before the rising edge of SH_{D1}. The sample and hold SH_{out} must be in its holding phase before SH_{D1} begins sampling because the delayed accumulation term is not used during the last DA computational cycle for the computation of the output sample $y[n]$.

5.4.4.8 Timing of the Invert Signal

The Invert signal is used to control a two-position switch that connects the active low input of the two-position switch controlled by the Reset signal to the inverted delayed accumulation term when the Invert signal is low and to the non-inverted delayed accumulation term when the Invert signal is high.

The Invert signal has a periodicity of one sample period. The Invert signal is active high and is not inverting the delayed accumulation term only during the last, B^{th}, computational cycle. For all other computational cycles, the Invert signal is low. This non-inversion of the delayed accumulation term is needed during the last DA computational cycle to generate the correct output for a DA-based filter using two's complement digital filter coefficients.

The rising edge of the Invert signal is aligned with the rising edge of the B^{th} computational cycle. The delayed accumulation term must remain non-inverted until the output signal $y(t)$ has been sampled by SH_{out} plus its holding settling time. Therefore, the falling edge of the Invert signal must occur at least a holding settling time of the sample and hold SH_{out} after the falling edge of Clk_{out}.

5.4.4.9 *Timing of the Reset Signal*

The Reset signal is used to control a two-position switch that connects the feedback summation term $FB_{\Sigma}(t)$ to the reference voltage V_{ref} when the Reset signal is low and to the output of the two-position switch controlled by the Invert signal when the Reset signal is high. The B^{th} computational cycle begins $B-1$ bit periods after the start of the DA computation, which is the rising edge of the sample clock.

The Reset signal has a periodicity of one sample period. The rising edge of the Reset signal can occur a holding settling time of the sample and hold SH_{out} after the falling edge of the Clk_{out} during the last computational cycle. By setting the Reset signal to high before the start of the next sample period, this action has no effect on the current output signal because it has already been sampled by SH_{out}, and $FB_{\Sigma}(t)$ must be set to V_{ref} for the next sample period. During most of the first computational cycle of the next sample period, the Reset signal continues to be high. The feedback summation term $FB_{\Sigma}(t)$ must remain connected to the reference voltage V_{ref} until the accumulation term has been sampled by the sample and hold SH_{D0} for the first phase of storing and of delaying the accumulation term and before the start of the next computational cycle. The falling edge of the Reset signal must occur at least a holding settling time of the sample and hold SH_{D0} after the falling edge of Clk_{D0} yet before the next rising edge of the bit clock.

5.5 Analysis of the Computational Error Sources

For this implementation, the gain and offset errors are determined by the accuracy of the DA computation. If the error at the addition stage is due to the sampled input errors in the sample and hold circuits and the mismatch errors between the input capacitors, $C_{in,i}$ (for $i = 0, 1, 2, ..., 5$), is assumed to be negligible, then the gain/offset errors and the noise in the data paths are the main sources of error. In this architecture, the inverting amplifiers, OTA_{Σ} and OTA_{DA}, are sources of gain and offset errors, and all the sample-and-hold circuits are sources of offset error. In addition, the mismatch between C_{FB} and C_{Σ} as well as between

$C_{DA,FB}$ and $C_{DA,in}$ are sources of gain error. In this analysis, the effects of gain, offset, and random errors in the system were analyzed.

5.5.1 Gain Error

In Section 4.4, an analysis of the error of the division by two computational circuit was performed; however, this analysis was for an FIR filter, and not for a second-order section. The key difference in the analysis for a second-order section is its analysis will contain two gain error sources rather than the one error source in the FIR case. A gain error is generated in both the feedforward and feedback signal paths. As before, the divide by two computational circuit is the source of this error. In this section, the gain error for the feedforward signal path is denoted as Δ_{ff}, and the gain error for the feedback signal path is designated as Δ_{fb}. The effects of Δ_{ff} and Δ_{fb} on the output of the DA computation for a second-order section, $y[n]$, are modelled by

$$\begin{aligned} y[n] = &-\left[\sum_{i=0}^{K-1} b_{i0}x[n-i]\right] + \sum_{l=1}^{B-1}\left[\sum_{i=0}^{K-1} b_{il}x[n-i]\right](0.5+\Delta_{ff})^l \\ &-\left[\sum_{j=0}^{L-1} a_{j0}y[n-j]\right] + \sum_{l=1}^{B-1}\left[\sum_{j=0}^{L-1} a_{jl}y[n-j]\right](0.5+\Delta_{fb})^l. \end{aligned} \tag{5.9}$$

For simplification, the term $\sum_{i=0}^{K-1} b_{il}x[n-i]$ is set to α for all l, and the term $\sum_{j=0}^{L-1} a_{jl}y[n-j]$ is set to β for all l. Therefore, Eq. (5.4) can be written as

$$y[n] = -\alpha + \sum_{l=1}^{B-1} \alpha\, 0.5^l - \beta + \sum_{l=1}^{B-1} \beta\, 0.5^l, \tag{5.10}$$

and Eq. (5.9) can be written as

$$y[n] = -\alpha + \sum_{l=1}^{B-1} \alpha\, (0.5+\Delta_{ff})^l - \beta + \sum_{l=1}^{B-1} \beta\, (0.5+\Delta_{fb})^l. \tag{5.11}$$

The output error caused by Δ_{ff} and Δ_{fb} can be found by computing the difference of Eq. (5.10) and Eq. (5.11). Now, the output error, ε, can be expressed as

$$\begin{aligned}
\varepsilon &= \left(\sum_{l=1}^{B-1} \alpha\,(0.5+\Delta_{ff})^l - \sum_{l=1}^{B-1} \alpha\,0.5^l\right) + \left(\sum_{l=1}^{B-1} \beta\,(0.5+\Delta_{fb})^l - \sum_{l=1}^{B-1} \beta\,0.5^l\right) \\
&= \left(\alpha\frac{1-(0.5+\Delta_{ff})^B}{1-(0.5+\Delta_{ff})} - \alpha\frac{1-0.5^B}{1-0.5}\right) + \left(\beta\frac{1-(0.5+\Delta_{fb})^B}{1-(0.5+\Delta_{fb})} - \beta\frac{1-0.5^B}{1-0.5}\right) \\
&= \alpha\left(\frac{1-(0.5+\Delta_{ff})^B}{0.5-\Delta_{ff}} - \frac{1-0.5^B}{0.5}\right) + \beta\left(\frac{1-(0.5+\Delta_{fb})^B}{0.5-\Delta_{fb}} - \frac{1-0.5^B}{0.5}\right). \qquad (5.12)
\end{aligned}$$

Despite this simplification, Eq. (5.12) is still not very intuitive. However if the worst case scenario is considered, which in many cases is the primary design concern, the output error due to gain error is maximized when β is equal to α. Eq. (5.12) can be further simplified by setting a new varilabe, Δ_{diff}, to be the difference of the two gain errors from the feedforward and feedback data paths, Δ_{ff} and Δ_{fb}, or in other words, Δ_{diff} is equal to $\Delta_{ff} - \Delta_{fb}$. With this new variable, Δ_{fb} can be rewritten as $\Delta_{ff} - \Delta_{diff}$. If Δ_{diff} is an order of magnitude smaller than Δ_{ff}, then Δ_{fb} is equal to Δ_{ff}. With this simplification and assumption, the output error ε can be expressed as

$$\varepsilon = 2\alpha\left(\frac{1-(0.5+\Delta_{ff})^B}{0.5-\Delta_{ff}} - \frac{1-0.5^B}{0.5}\right). \qquad (5.13)$$

A plot of the output error due to gain error normalized by α for varying values of Δ_{ff} and B is given in Figure 5.13. Since this system uses digital filter coefficients to generate an analog output from sampled analog data, the output error due to quantization is also provided. The intersection of output error (due to gain error) and quantization error curves is the minimum achievable output error of the proposed system and is used to determine the precision of an equivalent digital system. For example when $\Delta_{ff} = 2^{-11}$, the two curves intersected at $\varepsilon/\alpha = 0.0037$ and $B = 7$. This intersection point is the minimum error when $\Delta = 2^{-11}$ and that the proposed system is equivalent to using a 7-bit digital DA. Also note that as B became large, ε/α approached a limit which is equal to $2\Delta_{ff}/(0.5-\Delta_{ff})$.

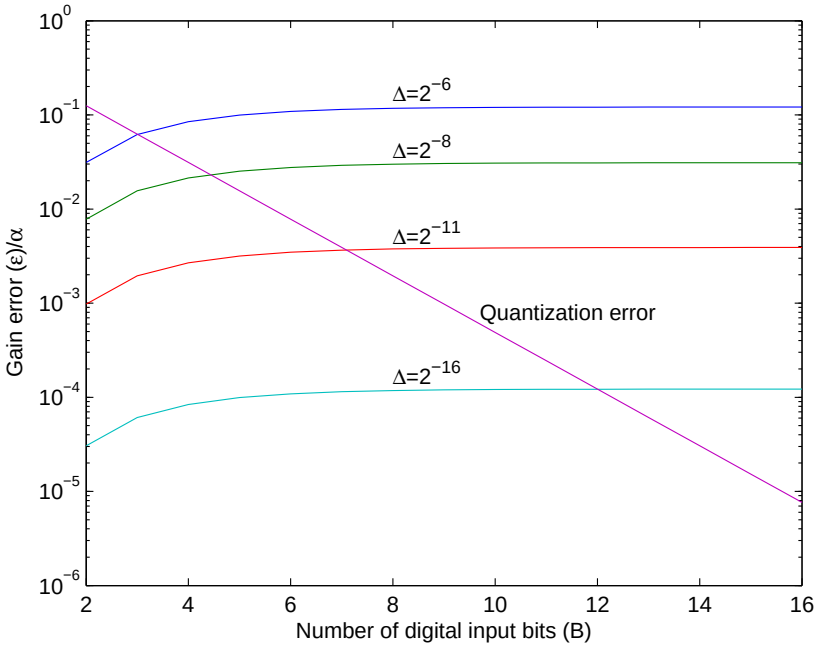


Figure 5.13. Computational error, ε/α, of the system due to quantization error and gain non-ideality, Δ_{ff}.

5.5.2 Offset Feedback Error

Another source of error is the offset error. As mentioned in the previous section, a second-order section has two computational feedback paths related to DA, one each for the feedforward and for the feedback inner product computational paths; therefore, it has two sources of offset error. The offset error that is generated by computing the inner product associated with the feedforward filter coefficients is modelled as a constant error, δ_{ff}, added to each l^{th} partial product, $\sum_{i=0}^{K-1} b_{il}x[n-i]$, and the offset error that is generated by computing the inner product associated with the feedback filter coefficients is modelled as a constant error, δ_{fb}, added to each j^{th} partial product, $\sum_{j=0}^{L-1} a_{jl}y[n-j]$. When considering these sources of error, the output, $y[n]$, can be written as

$$\begin{aligned} y[n] = &-\left[\delta_{ff} + \sum_{i=0}^{K-1} b_{i0}x[n-i]\right] + \sum_{l=1}^{B-1} 2^{-l}\left[\delta_{ff} + \sum_{i=0}^{K-1} b_{il}x[n-i]\right] \\ &-\left[\delta_{fb} + \sum_{j=0}^{L-1} a_{j0}y[n-j]\right] + \sum_{l=1}^{B-1} 2^{-l}\left[\delta_{fb} + \sum_{j=0}^{L-1} a_{jl}y[n-j]\right]. \end{aligned} \tag{5.14}$$

After distributing $\sum_{i=1}^{B-1} 2^{-l}$, grouping the δ_{ff}'s into one term, and grouping the δ_{fb}'s into a single term, the error due to the offset can be written as the difference of two geometric series summations.

$$\begin{aligned} error_{\text{offset}} &= \left(\delta_{ff}\ \frac{1-0.5^B}{1-0.5} - 2\delta_{ff}\right) + \left(\delta_{fb}\ \frac{1-0.5^B}{1-0.5} - 2\delta_{fb}\right) \\ &= \delta_{ff} \cdot 2^{-(B-1)} + \delta_{fb} \cdot 2^{-(B-1)} \\ &= \left(\delta_{ff} + \delta_{fb}\right) \cdot 2^{-(B-1)} \end{aligned} \tag{5.15}$$

The value of δ_{fb} has as equal a likelihood of being equal to α as to being equal to $-\alpha$ due to the manufacturing variances for fabricating CMOS circuits. Rather than expressing the error in terms of δ_{ff} or δ_{fb}, Eq. (5.15) can be expressed in terms of a new variable called δ_{avg} as

$$error_{\text{offset}} = 2\delta_{avg} 2^{-(B-1)} = \delta_{avg} 2^{-(B-2)}, \tag{5.16}$$

where δ_{avg} is equal to the average of the offset error for the two computational paths, or in other words, δ_{avg} is equal to $(\delta_{ff} + \delta_{fb})/2$. There is noticeable similarity between Eq. (4.6) and Eq. (5.16). Assuming δ is equal to δ_{avg}, the most notable difference is that the output error for the second-order section is twice as large when compared to the FIR filtering case. This result is to be expected because there are two sources of error, which are the DA feedback paths for the computation of the feedforward and of the feedback inner products, in the case of a second-order section while there is only one source of error, which is just for the feedforward inner product DA computational path, for an FIR filter.

As seen in the FIR when B increases, the offset error in the feedback loops decreases. Again, this result is a byproduct of how DA handles two's complement numbers. In DA for the computation of each inner product, the partial product associated with the most significant bit or the sign bit is subtracted rather than added as is the case for all the other partial products. In other words for the DA mechanization of the feedforward inner product, the partial product, $\sum_{i=0}^{K-1} b_{i0} x[n-i]$, is deducted rather than accumulated into the final result. Similarly for the DA mechanization of the feedback inner product, the partial product,

$\sum_{j=0}^{L-1} a_{j0}y[n-j]$, is subtracted rather than added to the output. This systematic attenuation of the offset error is significant when the B is large. For example when $B = 8$ and $\delta_{avg} = 100mV$, the offset error is $1.5625mV$, which is a reduction of 64x.

5.5.3 Random Feedback Error

As before for the FIR filtering case, the random error is assumed to be Gaussian and is represented by S_l. The random variable S_l is added to the summation of weights at each l^{th} iteration, and all S_l's are independent and identically distributed. With the these sources of random error added to the feedback paths for both the feedforward and feedback DA computations, the output, $y[n]$, can be expressed as

$$y[n] = -\left[S_0 + \sum_{i=0}^{K-1} b_{i0}x[n-i]\right] + \sum_{l=1}^{B-1} 2^{-l}\left[S_l + \sum_{i=0}^{K-1} b_{il}x[n-i]\right] - \left[S_0 + \sum_{j=0}^{L-1} a_{j0}y[n-j]\right] + \sum_{l=1}^{B-1} 2^{-l}\left[S_l + \sum_{j=0}^{L-1} a_{jl}y[n-j]\right]. \tag{5.17}$$

Once the term $\sum_{l=1}^{B-1} 2^{-l}$ is distributed and the $S_l's$ are collected into one summation, the mean and variance of $y[n]$ can be written as $\mu_Y = 2\mu_S \frac{1-0.5^B}{1-0.5} - 4\mu_S = 4\mu_S([1-0.5^B]-1)$ and $\sigma_Y^2 = 2\sigma_S^2 \frac{1-0.25^B}{1-0.25}$, respectively. As B approaches infinity, the mean of the random error approaches zero, and the maximum variance of the random error is $\frac{8}{3}\sigma^2$.

5.6 FPAA Results

The filtering structure detailed in the previous section are simulated using the WinSpice simulation tool. WinSpice uses the EKV model to describe the behavior of the individual transistors. This model provides a better model of subthreshold behavior than other models. This aspect of the WinSpice simulation tool is critical because the devices used in the RASP 2.8 FPAA are typically operated in subthreshold.

To demonstrate the capability of the proposed second-order section filter implementation, it was configured as a low pass filter. The feedforward filter coefficients, $[b_0 b_1 b_2]$, were set to [0.5 0.25 0.125], and the feedback filter coefficients, $[a_0 a_1 a_2]$, were set to [1

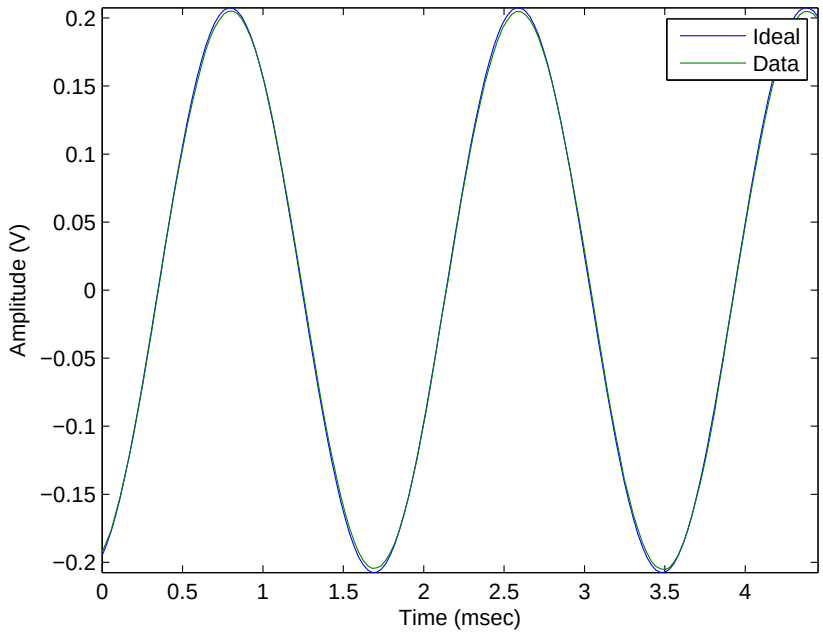


Figure 5.14. The transient response of the DA-based second-order section when configured as a low–pass filter for an input frequency of $558.036Hz$ **at a sampling rate of** $35.714kHz$**.**

-0.25 -0.125]. These coefficients were quantized to four bits of precision. The sampling rate was set to $35.714kHz$, and the reference voltage, V_{ref}, was set to $1.2V$ or at the mid-point between ground and the supply rail of $2.4V$. The provided input to the system was a sinusoid with a magnitude of $0.15V$ and no DC offset. This magnitude was selected to avoid generating a voltage that either exceeded the supply rail or fell below ground.

Two types of results were collected. They are the output transient responses and the magnitude and phase response.

The output transient responses are presented first. Figure 5.14 is the transient response of the low-pass filter for an input frequency of $558.036Hz$. The ideal transient response is in blue, and the observed result is in green. As can be seen from the figure, the ideal and the observed responses are nearly indistinguishable from one another. The average magnitude difference between the two responses was slightly more than $9mV$.

Figure 5.15 is the transient response of the low-pass filter for an input frequency of

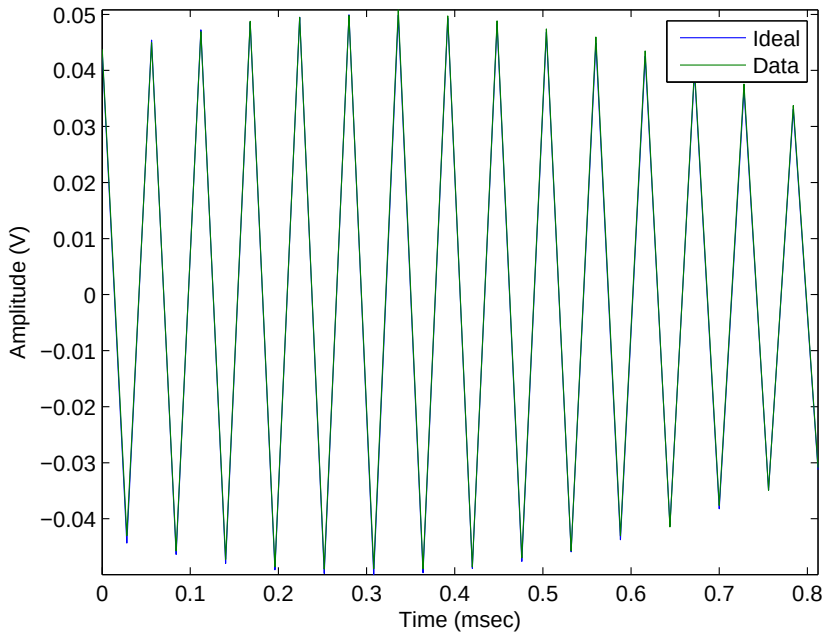


Figure 5.15. The transient response of the DA-based second-order section when configured as a low–pass filter for an input frequency of $17.578kHz$ **at a sampling rate of** $35.714kHz$**.**

$17.578kHz$. The earlier response was used to illustrate the transient response at low frequencies where this response is used to illustrate the transient response at high frequencies, in particular close to the Nyquist sampling rate. Like before, the ideal transient response is in blue, and the observed result is in green. As can be seen from the figure, the ideal and the observed responses are again nearly indistinguishable from one another. The average magnitude difference between the two responses was slightly more than $21mV$.

The next two figures are the magnitude and phase response. These figures were generated using the transient responses of sixty-four different frequencies spaced evenly from zero to sixty-three over sixty-fourths the Nyquist frequency. Figure 5.16 is the magnitude response of the low-pass filter. The ideal response is in blue, and the observed result is in green. Overall, the observed magnitude response is relatively close to the ideal one. The maximum difference in magnitude between the ideal and the observed responses is slightly more than four tenths of a decibel.

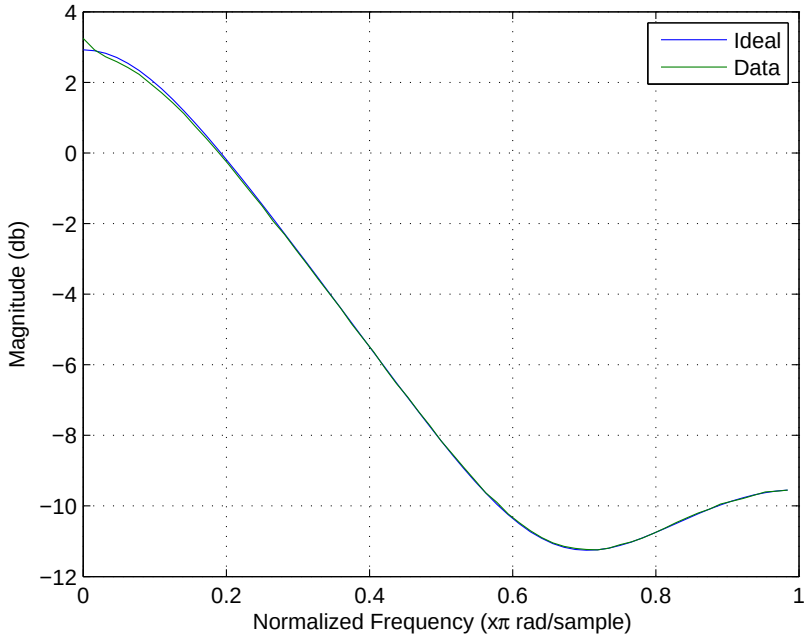


Figure 5.16. The magnitude response of the DA-based second-order section when configured as a low–pass filter.

Figure 5.17 is the phase response of the low-pass filter. As before, the ideal response is in blue, and the observed result is in green. Overall, the observed phase response is relatively close to the ideal one. The maximum phase difference between the ideal and the observed responses is less than five and a half degrees.

5.7 Summary

The implementation of a second-order section filter using distributed arithmetic was successfully accomplished. This implementation is targeted for use on a field-programmable analog array (FPAA), in particular the RASP 2.8 FPAA, in conjunction with an FPGA or a microcontroller. The results were collected using the WinSpice simulation software. Using this software, the transient responses of a low pass filter for sixty-four frequencies were collected. These transient responses were then analyzed and utilized to generate a sixty-four point magnitude and phase response. When comparing the observed magnitude response with the ideal one, the result was within five tenths of a decibel. These results showed that a

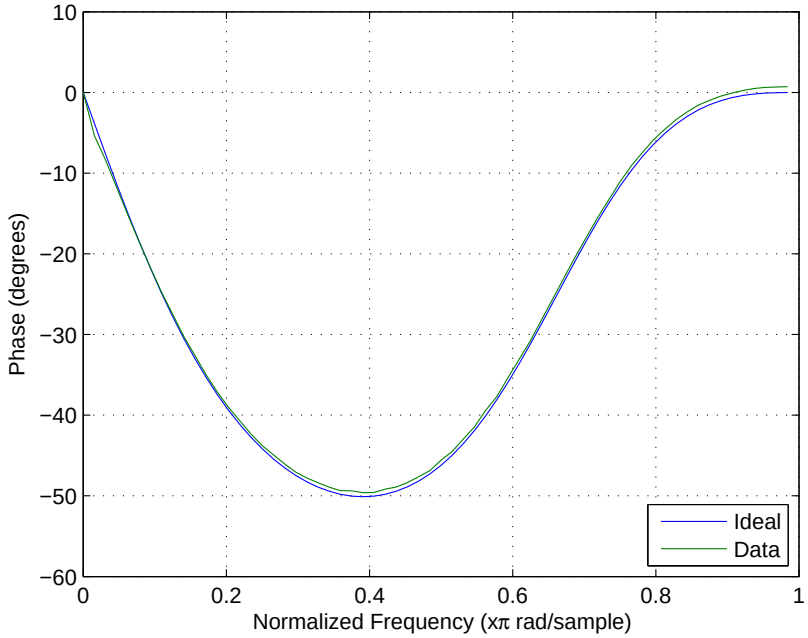


Figure 5.17. The phase response of the DA-based second-order section when configured as a low-pass filter.

second-order section using distributed arithmetic was successfully created from the RASP 2.8 FPAA components.

5.8 Contributions

In this chapter, a reprogrammable mixed-signal second-order section was proposed and developed. The issues addressed by the proposed reprogrammable mixed-signal second-order section are the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples. The following contributions were made to address these issues.

1. A reprogrammable mixed-signal second-order section was proposed and developed to address the issues of the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of

the input and output data, and the corruption of the input and output samples.

2. Distributed arithmetic and digital registers were used to address the issue of a lack of a compact reprogrammable filtering structure for a reprogrammable mixed-signal second-order section. A combination of distributed arithmetic and digital registers were used to construct a compact reprogrammable mixed-signal second-order section.

3. Digital registers were used to address the issue of imprecise feedback and feedforward filter coefficients for a reprogrammable mixed-signal second-order section. The digital registers were used to precisely reprogram the filter coefficients.

4. Independently clocked sample and hold circuits and a circular buffer of digital registers were used to address the issues of inconsistent sampling of the input and output data and of the corruption of the input and output samples for a reprogrammable mixed-signal second-order section. The independently clocked sample and hold circuits and the circular buffer of digital registers were used to consistently sample the input and output data and to minimize the corruption of the input and output samples.

5. An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed reprogrammable mixed-signal second-order section. Deduced from this analysis, a guideline was generated of where to focus the design effort for the proposed reprogrammable mixed-signal second-order section. The guideline is to use most of the design effort on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

CHAPTER 6

CONCLUSIONS, CONTRIBUTIONS AND DIRECTIONS FOR FUTURE RESEARCH

The conclusions and contributions of this research are given in the first two sections. In addition, directions for future research are given in the last section of this chapter.

6.1 Conclusions

When computational resources are limited in particular multipliers, distributed arithmetic (DA) is used in lieu of the typical multiplier-based filtering structures. This advantage in terms of computational resources is further extended when the bit precision of the filter is high. However due to the construction of the memory tables used for distributed arithmetic, it is not well suited for adaptive applications. The bottleneck when using DA for an adaptive filter is updating the memory table. Several attempts have been done to accelerate the process of updating the memory. Although these approaches do reduce the amount of processing necessary to update the memory, this reduction is gained at the expense of additional memory usage and of convergence speed. A more desirable solution is to develop a new approach for updating the memory table efficiently without using additional memory resources and compromising the convergence rate. In this thesis, such an approach is proposed and developed.

To develop an adaptive distributed arithmetic filter with a convergence rate that is not compromised, the memory table must be fully updated, and to realize that for an adaptive distributed arithmetic filter the memory table does not have to be composed of the combinations of the filter coefficients and be addressed by concatenating bits of the input data as in a traditional distributed arithmetic filtering structure. The memory table instead can composed of the combinations of the input samples and can be addressed by concatenating bits of the filter coefficients.

For an adaptive filter, the contents of the memory table must be updated regardless if the memory table is based on the filter coefficients or based on the input samples. In this research, an efficient method for fully updating a memory table that is composed of the combinations of the input samples and that is addressed by concatenating bits of the filter coefficients was proposed and developed. The proposed update method is based on exploiting the temporal locality of the stored data, which is a combination of the input samples that is determined by its memory address, and subexpression sharing. The proposed update method is possible because the data set, which is composed of the input samples, that the memory table is constructed from changes slowly. In other words, only the newest input sample needs to be added to the data set, and the oldest input sample needs to be removed from the data set. The proposed update method reduced the computational workload for updating the memory table by about $k/2-1$ over brute-force and required no additional memory resources. This type of distributed arithmetic that uses the proposed update method is called conjugate distributed arithmetic (CDA). This type of update method is not applicable to a memory table composed of the combinations of the filter coefficients and addressed by concatenating bits of the input samples because the entire data set, which is composed of the filter coefficients, changes from sample to sample because of the adaptive nature of the filter.

The performance of CDA was compared against a traditional multiplier based approach and against sliding-block distributed arithmetic (SBDA), which is the only other adaptive distributed arithmetic approach that addresses the same issues as CDA. When CDA is compared against a traditional multiplier based approach, the following conclusions were deduced. The throughput of CDA remained relatively constant as the filter order increased while the throughput of a traditional multiplier based approach decreased quadratically with increasing filter order. Also, a CDA adaptive filter can use up to 3.3 times fewer logic elements than a traditional multiplier based adaptive filter to achieve a certain throughput and uses the same amount of memory as a non-adaptive distributed arithmetic FIR filter.

When CDA is compared against SBDA, the following conclusions were deduced. CDA uses $\frac{1}{\lfloor K/k \rfloor + 1}$ less memory than SBDA, and this advantage is significant when $\lfloor K/k \rfloor$ is small. However, when $\lfloor K/k \rfloor$ is large, this advantage is offset by an increase in the number of additions necessary for CDA over SBDA. Although in some situations, CDA uses less adders than SBDA when $\frac{B_h}{(\lfloor K/k \rfloor - 1) \cdot \left(2^{k-1} - 1\right)} > 1$.

In addition to developing a new type of adaptive distributed arithmetic called CDA, the only other type of adaptive distributed arithmetic that addressed the same issues as CDA called SBDA was modified to reduce the memory usage and to reduce the computational workload. SBDA was modified to encode the memory tables using offset binary coding (OBC). However, the computational workload for updating the memory table remained unchanged from the non encoded case for the same filter length. By modifying the value the memory table is initialized, the computational workload for updating the memory table is reduced for most filtering configurations. This modification of SBDA to encode the memory tables using OBC and to initialize the memory table with the initial condition instead of zero as originally proposed in SBDA is called SBDA-OBC. When SBDA-OBC is compared with SBDA, the following conclusions were deduced. SBDA-OBC requires less memory than SBDA. The maximum memory usage advantage of SBDA-OBC over SBDA is approximately half, and this situation occurs when the size of the sub-filter is large. In terms of the computational workload, SBDA-OBC is most advantageous for large sub-filters and when the filter is split into few sub-filters. In this case, the computational workload is reduced almost in half.

Filters can also be constructed from analog components. Often for lower precision computations, analog circuits use less power and less chip area than their digital counterparts. However even with these advantages, digital components are often used because of their ease of reprogrammability. Achieving such reprogrammability with analog circuits is possible, but comes at the expense of additional chip area. This increase often eliminates the area advantage of using analog circuits over digital ones.

When wide linear phase is desired, a couple typical approaches exist for constructing an analog filter with wide linear phase. However, each approach has it own set of issues. A common issue among these approaches is the lack of reprogrammability or the lack of a compact structure to provide reprogrammability. When using an FIR filtering approach in the analog domain to generate a filtering structure with wide linear phase, a couple issues exist. One difficulty is obtaining symmetric and precise filter coefficients. The other issues are inconsistent sampling of the input signal and the corruption of the stored input data samples.

In this research, a reprogrammable mixed-signal distributed arithmetic FIR filter is proposed to address the issues with reprogrammable analog FIR filters of constructing compact reprogrammable filtering structures, non-symmetric and imprecise filter coefficients, inconsistent sampling of the input data, and input sample data corruption. These issues are addressed by using a combination of distributed arithmetic, digital storage elements, and epots, which are compact, reprogrammable, and precise voltage references.

The issue of constructing compact reprogrammable filtering structures is addressed by using a combination of distributed arithmetic and epot analog storage elements. By using a distributed arithmetic filtering structure in the analog domain, the additional analog components needed to make the filter reprogrammable is compressed into a few components. These additional analog components needed for the distributed arithmetic computational feedback path is reused for every bit instead of being dedicated to a single bit. In other words, the amount of hardware needed to support reprogrammability is compressed from multiple bits into a single bit. The full bit precision of the computation is achieved in DA by computing the output sample in a bit-serial fashion. By compressing the hardware using distributed arithmetic, the potential amount of chip area saved could be significant, up to a factor of the bit precision of the filter.

When using the proposed FIR filtering structure, compact and precise reprogrammable

analog storage elements are needed for the filter coefficients to address the issues of re-programmability and of non-symmetric and imprecise filter coefficients. Note, the degree of symmetry achieved is determined by the achievable precision of the filter coefficients; therefore, the more precise the filter coefficients can be programmed then the higher degree of symmetry that is present in the filter coefficients. To address this requirement, epots are used. Epots are reprogrammable analog voltage references [41]. For the compact and precise epot needed for this application, the epots in [42] were used. The compactness and precision of these epots were achieved by using floating-gates.

To address the issues of inconsistent input data sampling and input sample data corruption, digital registers are used to stored the input data. By storing the data in digital storage elements such as flip-flops, the volatility of the stored input samples is eliminated. By eliminating this volatility and a well designed clock distribution tree, the input data is consistently sampled, and the data storage elements can be cascaded without concern for data corruption. This ability to cascade many input data storage elements together allows for the construct of high order FIR filter using the proposed mixed-signal filtering structure.

To demonstrate the reprogrammability of the proposed mixed-signal FIR filter, the filtering structure was successful reprogrammed as a low-pass, band-pass, and comb filter. This success is illustrated by frequency responses that are very close to the ideal ones. These frequency responses also illustrate the high degree of precision the epots were programmed. The epots were programmed with enough accuracy that the phase response, which are very sensitive to imprecisely programmed filter coefficients, of the programmed filters are nearly linear. The measured spurious-free-dynamic-range (SFDR), which is the difference in amplitude between the input frequency and the largest non-input frequency components, of an 8-bit FIR filter is $43dB$, which is close to the expected bit resolution for eight bits.

An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed FIR

filter. The sources of error considered were gain error in the signal path, offset error in the signal path and weights, and noise in the signal path and weights. Each source of error was analyzed independently of one another. The key conclusions of these analyses are that offset error sources are attenuated by the distributed arithmetic computational process, that the variance of random error sources are amplified by the distributed arithmetic computational process, and that the gain error in the signal path determines the bit precision of the proposed filter given a certain digital bit precision. From a design standpoint, these conclusions mean that most of the design effort should be placed on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

In addition to the proposed mixed-signal distributed arithmetic FIR filter, a mixed-signal distributed arithmetic second-order section (SOS) was proposed and developed. Second-order sections are used as the building blocks for higher order infinite impulse response (IIR) filters. IIR filters are better suited than FIR filters for applications that require a steep roll-off in the frequency response. The type of issues with an analog second-order section filter are similar to those of an analog FIR filter, which are the lack of a compact reprogrammable filtering structure, the imprecise filter coefficients, the inconsistent sampling of the data, and the corruption of the data samples. The key difference between the two is that a feedback path is present in addition to the feedforward path, which is also present in an FIR filter. The feedback path is constructed in a similar manner to the feedforward path except that it uses the output samples instead of the input samples.

In the proposed reprogrammable mixed-signal distributed arithmetic second-order section filter, the typical issues associated with reprogrammable analog IIR filters, which are the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples, are addressed. These issues are addressed using a combination of distributed arithmetic and digital storage elements.

The issue of reprogrammability is addressed by using distributed arithmetic and by using digital storage elements. By using a distributed arithmetic filtering structure in the analog domain, the amount of hardware needed to support reprogrammability is compressed from multiple bits into a single bit. By compressing the hardware using distributed arithmetic, the potential amount of chip area saved could be significant, up to a factor of the bit precision of the filter.

The ease of reprogrammability is further enhanced by using digital storage elements such as flip-flops to store the filter coefficients. Reprogramming the filter coefficients becomes as simple as loading a new data word into a digital register. Also by using digital storage elements, the precision of the feedback and feedforward filter coefficients are known and do not change over time. The usage of digital registers to store the filter coefficients effectively eliminates any issues related to precision.

Inherently in a second-order section, the issue of input and output sample data corruption when using a cascade of sample and hold circuits is limited. This issue and the issue related to inconsistent sampling of the input and output data can be eliminated by using independently clocked sample and hold circuits; however for an analog system, the only feasible structure to connect the appropriate input and output sample to the appropriate feedforward and feedback filter coefficient is to use a large switching matrix. In the proposed mixed-signal second-order section, the switching matrix is eliminated because the filter coefficients are stored using digital registers. By using digital registers, the filter coefficients can be stored in a circular buffer without concern of data corruption as would occur in an analog system. By storing the filter coefficients in a circular buffer, the filter coefficients can be easily aligned with the appropriate input or output sample.

Simulation results of the proposed mixed-signal second-order section were generated. These results were used to generate the frequency response for a low-pass filter. The proposed filtering structure successful generated a frequency response close to the ideal one. The maximum difference in magnitude between the ideal and the observed responses is

slightly more than four tenths of a decibel. The maximum phase difference between the ideal and the observed responses is less than five and a half degrees.

An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed second-order section. The sources of error considered were gain error in the signal path, offset error in the signal path, and random noise in the signal path. Each source of error was analyzed independently of one another. The key conclusions of these analyses are that offset error sources are attenuated by the distributed arithmetic computational process, that the variance of the random error sources are amplified by the distributed arithmetic computational process, and that the gain error in the signal path determines the bit precision of the proposed filter given a certain digital bit precision. From a design standpoint, these conclusions mean that most of the design effort should be placed on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

6.2 Contributions

In this research, contributions are made in the following fields. The first one is in the development of an adaptive filter using distributed arithmetic. The issues addressed by this research are the lack of an efficient method to fully update the memory table of a distributed arithmetic adaptive filter, the usage of memory resources beyond that of the non-adaptive case, and the compromised convergence rate. The following contributions were made to address these issues and to develop an adaptive distributed arithmetic filter with an efficient method to fully update the memory table without using additional memory resources and with uncompromised convergence performance.

1. A new method for fully updating the memory table was proposed. By fully updating the memory table, the convergence performance of the adaptive filter remains unaffected; therefore, the proposed update method can be used to construct adaptive filters using distributed arithmetic without a compromised convergence rate.

2. A new method for efficiently updating the entire memory table was proposed. The proposed update method reduced the computational workload for updating the memory table by about $k/2 - 1$ over brute-force.

3. By using a filter coefficient driven distributed arithmetic filtering structure, the additional memory resources required by most other types of adaptive distributed arithmetic filtering structures, which use an input driven memory table that is composed of the combinations of its filter coefficients, are eliminated.

In this research, the proposed memory update method is combined with a filter coefficient driven distributed arithmetic filtering structure. This combination is called conjugate distributed arithmetic (CDA).

After a through literature review of adaptive distributed arithmetic filtering structures, only one other type of adaptive DA that is called sliding-block distributed arithmetic (SBDA) also addresses the issues outlined above with an adaptive DA filter. Although CDA is not the only adaptive distributed arithmetic filtering structure that addresses these issues, CDA is advantageous in a variety of filter configurations.

1. Among the adaptive distributed arithmetic filters that fully updates its memory tables, CDA uses the least amount of memory. Its memory usage is only matched by the brute-force method; however, CDA reduces the number of operations required by about $k/2 - 1$ over brute force. Recall, only adaptive distributed arithmetic filters that fully updates its memory tables is able to maintain the convergence speed of the LMS algorithm.

2. CDA uses less memory than SBDA especially if the filter is broken up into few subunits. This advantage is useful when coded on a system with limited memory.

3. CDA uses fewer additions than SBDA when the bit precision of the filter coefficients in SBDA is greater than the number of additional memory table entries that need

to be updated in CDA. Typically, this occurs when the coefficient bit precision, the number of subunits, the depth of the memory tables, or a combination of these three are low. A couple benefits of fewer additions are boosted sampling rate, or lower power usage.

In addition to CDA and SBDA, an alternative adaptive DA filter structure called SBDA-OBC was proposed and developed. Its memory update method is a modification of the one used in SBDA, and it has lower memory usage and fewer additions for most filter configurations over SBDA. The principle motivations for modifying SBDA are to encode the memory using OBC such that the memory usage is reduced almost in half and to modify SBDA in such a way that when the memory is encoding using OBC that the computational workload for updating the memory table is reduced. The following are the observed savings of SBDA-OBC.

1. SBDA-OBC has the lowest memory requirements of any current mechanization for a coefficient driven, memory-based DA adaptive FIR filter. Specifically, SBDA-OBC uses about 50% less memory than SBDA when the filter length of the subunits is long.

2. SBDA-OBC has the fewest number of additions for a large number of filtering configurations among the current mechanizations for a coefficient driven, memory-based DA adaptive FIR filter. Specifically, SBDA-OBC needs about 50% less additions than SBDA when the filter length of the subunits is long.

Contributions were also made in the field of reprogrammable mixed-signal FIR filter design. The issues addressed by this research are the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter coefficients, the inconsistent sampling of the input data, and the corruption of the input samples. The following contributions were made to address these issues.

1. A reprogrammable mixed-signal FIR filter was proposed and developed to address the issues of the lack of a compact reprogrammable filtering structure, the non-symmetric and imprecise filter coefficients, the inconsistent sampling of the input data, and the corruption of the input samples.

2. Distributed arithmetic and epots were used to address the issue of a lack of a compact reprogrammable filtering structure for a reprogrammable mixed-signal FIR filter. A combination of distributed arithmetic and epots were used to construct a compact reprogrammable mixed-signal FIR filter.

3. epots were used to address the issue of non-symmetric and imprecise filter coefficients for a reprogrammable mixed-signal FIR filter. epots were used to precisely reprogram the filter coefficients. These filter coefficients can be programmed with such precision that a natural by-product is high filter coefficient symmetry.

4. Digital registers were used to address the issues of inconsistent input data sampling and of input sample data corruption for a reprogrammable mixed-signal FIR filter. The digital registers were used to sample the input data consistently and to eliminate input data corruption. Because the digital registers can be cascaded without concern for data corruption, this ability to cascade many input digital registers together allows for the construct of a high order FIR filter using the proposed reprogrammable mixed-signal FIR filtering structure.

5. An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed reprogrammable mixed-signal FIR filter. Deduced from this analysis, a guideline was generated of where to focus the design effort for the proposed reprogrammable mixed-signal FIR filter. The guideline is to use most of the design effort on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

Finally, contributions were made in the field of reprogrammable mixed-signal second-order section design. The issues addressed by this research are the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples. The following contributions were made to address these issues.

1. A reprogrammable mixed-signal second-order section was proposed and developed to address the issues of the lack of a compact reprogrammable filtering structure, the imprecise feedback and feedforward filter coefficients, the inconsistent sampling of the input and output data, and the corruption of the input and output samples.

2. Distributed arithmetic and digital registers were used to address the issue of a lack of a compact reprogrammable filtering structure for a reprogrammable mixed-signal second-order section. A combination of distributed arithmetic and digital registers were used to construct a compact reprogrammable mixed-signal second-order section.

3. Digital registers were used to address the issue of imprecise feedback and feedforward filter coefficients for a reprogrammable mixed-signal second-order section. The digital registers were used to precisely reprogram the filter coefficients.

4. Independently clocked sample and hold circuits and a circular buffer of digital registers were used to address the issues of inconsistent sampling of the input and output data and of the corruption of the input and output samples for a reprogrammable mixed-signal second-order section. The independently clocked sample and hold circuits and the circular buffer of digital registers were used to consistently sample the input and output data and to minimize the corruption of the input and output samples.

5. An analysis of potential error sources generated by using analog components was performed to determine the effects that these analog components have on the proposed

reprogrammable mixed-signal second-order section. Deduced from this analysis, a guideline was generated of where to focus the design effort for the proposed reprogrammable mixed-signal second-order section. The guideline is to use most of the design effort on minimizing the variance of random errors sources and maximizing the precision of the amplifiers used.

6.3 Directions for Future Research

The following items listed below are directions for future research.

- In this thesis in Chapter 4, the construction and the performance of a 16-tap mixed-signal FIR filter was presented. By using an external operational amplifier as a summation element, multiple 16-tap mixed-signal FIR filters could be combined together to create high-order FIR filtering systems.

- In this thesis, the construction and the performance of a second-order section mixed-signal distributed arithmetic filter was studied. Using this second-order section as a building block, filters with more demanding frequency responses can be constructed using multiple sections. To minimize the effects of noise when cascading multiple analog components in series, this high-order filter can be constructed using second-order sections connected in parallel.

- Although the RASP 2.8 FPAA is capable of synthesizing the proposed mixed-signal second-order section, the capabilities could be further enhanced by optimizing the RASP 2.8 FPAA for the proposed mixed-signal filters.

- A folded mixed-signal FIR filter could be developed to further enhance the linear phase response of an FIR filter. A mixed-signal FIR filter constructed in such a manner should have more symmetric filter coefficients. The linear phase is improved by improving the symmetry of the filter coefficients.

- To increase the variety of applications that a mixed-signal distributed arithmetic filter could be used, an analog-to-analog mixed-signal distributed arithmetic FIR filter could be designed. This type of filter would complement the proposed digital-to-analog mixed-signal DA FIR filter, and constructed in a similar manner to the analog-to-analog mixed-signal distributed arithmetic second-order section.

REFERENCES

[1] D. L. Jones, "Efficient computation of time-varying and adaptive filters," *IEEE Transactions on Signal Processing*, pp. 1077–1086, March 1993.

[2] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, Inc, 1st ed., 1979.

[3] A. Peled and B. Liu, "A new hardware realization of digital filters," *IEEE Transactions on A.S.S.P.*, vol. ASSP-22, pp. 456–462, December 1974.

[4] S. A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, vol. 6, pp. 4–19, July 1989.

[5] S. Haykin, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice Hall, 1996.

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

[7] A. Croisier, D. J. Esteban, M. E. Levilion, and V. Rizo, "Digital filter for PCM encoded signals." U.S. Patent No. 3,777,130, issued April, 1973.

[8] S. G. Smith and P. B. Denyer, *Serial-Data Computation*. Kluwer Academic Publishers, 1st ed., 1988.

[9] C.-H. Wei and J.-J. Lou, "Multimemory block structure for implementing a digital adaptive filter usiing distributed arithmetic," *IEE Proceedings, Part G: Electronic Circuits and Systems*, vol. 133, pp. 19–26, Februray 1986.

[10] C. F. N. Cowan and J. Mavor, "New digital adaptive-filter implementation using distributed-arithmetic techniques," *IEE Proceedings, Part F: Communications, Radar and Signal Processing*, vol. 128, pp. 225–230, August 1981.

[11] C. F. N. Cowan, S. G. Smith, and J. H. Elliott, "A digital adaptive filter using a memory-accumulator architecture: Theory and realization," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 31, pp. 541–549, June 1983.

[12] C. F. N. Cowan and P. F. Adams, "Non-linear system modelling: Concept and application," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 9, pp. 444–447, March 1984.

[13] M. J. Smith, C. F. N. Cowan, and P. F. Adams, "Nonlinear echo cancellers based on transpose distributed arithmetic," *IEEE Transactions on Circuits and Systems*, pp. 6–18, January 1988.

[14] A. Sinha and A. P. Chandrakasan, "Energy efficient filtering using adaptive precision and variable voltage," in *Proceedings of the IEEE International ASIC/SOC Conference*, pp. 327–331, September 1999.

[15] R. Amirtharajah, T. Xanthopoulos, and A. Chandrakasan, "Power scalable processing using distributed arithmetic," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 170–175, August 1999.

[16] T. Xanthopoulos and A. P. Chandrakasan, "A low-power dct core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization," *IEEE Journal of Solid-State Circuits*, pp. 740–750, May 2000.

[17] G. Cherubini, "Analysis of the convergence behavior of adaptive distributed-arithmetic echo cancellers," *IEEE Transactions on Communications*, pp. 1703–1714, November 1993.

[18] G. Cherubini, "Nonlinear self-training adaptive equalization for partial-response systems," *IEEE Transactions on Communications*, vol. 42, pp. 367–376, February/March/April 1994.

[19] S. Chivapreecha, A. Jaruvarakul, N. Jaruvarakul, and K. Dejhan, "Adaptive equalization architecture using distributed arithmetic for partial response channels," in *Proceedings of the IEEE International Symposium on Consumer Electronics*, pp. 1–5, June 2006.

[20] Y.-T. Hwang and J.-C. Han, "A novel FPGA design of a high throughput rate adaptive prediction error filter," in *Proceedings of the IEEE Asia Pacific Conference on ASICs*, pp. 202–205, August 1999.

[21] Y.-T. Hwang and C. S. Lin, "Bit level VLSI design of high throughput DLMS adaptive IIR filters," in *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, vol. 2, pp. 1399–1402, August 1997.

[22] Y.-T. Hwang and C. S. Lin, "VLSI design of DLMS adaptive IIR filters for high speed echo cancellation," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, pp. 341–350, November 1997.

[23] Y.-T. Hwang and W.-C. Lin, "A high throughput rate and low circuit complexity QAM channel equalizer design based on bit serial scheme," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, pp. 558–567, October 1999.

[24] G. L. Sicuranza, A. Bucconi, and P. Mitri, "Adaptive echo cancellation with nonlinear digital filters," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 9, pp. 130–133, March 1984.

[25] Y. Tsunekawa, K. Takahashi, S. Toyoda, and M. Miura, "High-performance VLSI architecture of multiplierless LMS adaptive filters using distributed arithmetic," *Electronics and Communications in Japan, Part 3*, vol. 84, pp. 1–12, 2001.

[26] G. L. Sicuranza and G. Ramponi, "Distributed arithmetic implementation of nonlinear echo cancellers," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 10, pp. 1617–1620, April 1985.

[27] G. L. Sicuranza and G. Ramponi, "Adaptive nonlinear digital filters using distributed arithmetic," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, pp. 518–526, 1986.

[28] K. Takahashi, Y. Tsunekawa, N. Tayama, and K. Seki, "Analysis of the convergence condition of LMS adaptive digital filter using distributed arithmetic," *IEICE Transactions Fundamentals of Electronics, Communications and Comoputer Sciences*, vol. E85-A, pp. 1249–1256, 2002.

[29] G. Ramponi and G. L. Sicuranza, "The sign algorithm in memory-oriented implementations of nonlinear adaptive digital filters," in *Proceedings of the European Conference on Circuit Theory and Design*, pp. 537–540, September 1985.

[30] D. J. 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*, vol. 52, no. 7, pp. 1327–1337, 2005.

[31] "Stratix device family data sheet." Altera Corporation, http://www.altera.com/literature/lit-index.html.

[32] "Quartus II handbook, volume 3, chapter 8." Altera Corporation, http://www.altera.com/literature/hb/qts/qts_qii53013.pdf.

[33] J. H. Anderson and F. N. Najm, "Power estimation techniques for FPGAs," *IEEE Transactions on Very Large Scale Integration Systems*, pp. 1015–1027, October 2004.

[34] H. G. Lee, K. Lee, Y. Choi, and N. Chang, "Cycle-accurate energy measurement and characterization of FPGAs," *Analog Integrated Circuits and Signal Processing*, pp. 239–251, March 2005.

[35] E. Todorovich, E. Boemo, F. Angarita, and J. Valls, "Statistical power estimation for FPGA's," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications*, pp. 515–518, August 2005.

[36] "Stratix III programmable power." Altera Corporation, http://www.altera.com/literature/wp/wp-01006.pdf.

[37] "Stratix II vs. Virtex-4 power comparison & estimation accuracy." Altera Corporation, http://www.altera.com/literature/wp/wp_s2v4_pwr_acc.pdf.

[38] E. Özalevli, W. Huang, P. E. Hasler, and D. V. Anderson, "VLSI implementation of a reconfigurable mixed-signal finite impulse response filter," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 2168–2171, May 2007.

[39] E. Özalevli, W. Huang, P. E. Hasler, and D. V. Anderson, "A reconfigurable mixed-signal VLSI implementation of distributed arithmetic used for finite-impulse response filtering," *IEEE Transactions on Circuits and Systems, Part I: Regular Papers*, vol. 55, pp. 510–521, March 2008.

[40] E. Özalevli, P. E. Hasler, D. V. Anderson, and W. Huang, "Reconfigurable mixed-signal VLSI implementation of distributed arithmetic," *U.S. Patent*, March 2008. #7,348,909.

[41] R. R. Harrison, J. A. Bragg, P. Hasler, B. A. Minch, and S. Deweerth, "A CMOS programmable analog memory cell array using floating-gate circuits," *IEEE Transactions on Circuits and Systems*, vol. 48, pp. 4–11, January 2001.

[42] E. Özalevli, *Exploiting Floating-Gate Transistor Properties in Analog and Mixed-Signal Circuit Design*. PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, United States, August 2006.

[43] E. Özalevli, C. M. Twigg, and P. Hasler, "10-bit programmable voltage-output digital-analog converter," in *Proceedings of the International Symposium on Circuits and Systems*, vol. 6, pp. 5553–5556, May 2005.

[44] W. C. Black, D. J. Allstot, and R. A. Reed, "A high performance low power CMOS channel filter," *IEEE Journal of Solid-State Circuits*, vol. 15, pp. 929–938, December 1980.

[45] P. J. Lim and B. A. Wooley, "A high-speed sample-and-hold technique using a Miller hold capacitance," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 643–651, April 1991.

[46] K. Bult and G. J. G. M. Geelen, "A fast-settling CMOS opamp for SC circuits with 90-db DC-gain," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1379–1384, December 1990.

[47] K. E. Brehmer and J. B. Wieser, "Large swing CMOS power amplifier," *IEEE Journal of Solid-State Circuits*, vol. 18, pp. 624–629, December 1983.

[48] S. K. Haykin, *Digital Signal Processing: A Computer-Based Approach*. McGraw-Hill, 2nd ed., 2001.

[49] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 2nd ed., 1999.

[50] "AN120E04 datasheet reconfigurable FPAA." Anadigm, Inc., http://www.anadigm.com/_doc/DS021000-U004.pdf.

[51] A. Basu, C. M. Twigg, S. Brink, P. Hasler, C. Petre, S. Ramakrishnan, S. Koziol, and C. Schlottmann, "RASP 2.8: A new generation of floating-gate based field programmable analog array," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 213–216, September 2008.