GSFAP Adaptive Filtering Using Log Arithmetic for Resource-Constrained Embedded Systems

MILAN TICHY and JAN SCHIER
Academy of Sciences of the Czech Republic
and
DAVID GREGG
Trinity College Dublin

Adaptive filters are widely used in many applications of digital signal processing. Digital communications and digital video broadcasting are just two examples. Traditionally, small embedded systems have employed the least computationally intensive filter adaptive algorithms, such as normalized least mean squares (NLMS). This article shows that FPGA devices are a highly suitable platform for more computationally intensive adaptive algorithms. We present an optimized core which implements GSFAP. GSFAP is an algorithm with far superior adaptation properties than NLMS, and with only slightly higher computational complexity. To further optimize resource requirements we use logarithmic arithmetic, rather than conventional floating point, within the custom core. Our design makes effective use of the pipelined logarithmic addition units, and takes advantage of the very low cost of logarithmic multiplication and division. The resulting GSFAP core can be clocked at more than 80MHz on a one million-gate Xilinx XC2V1000-4 device. The core can be used to implement adaptive filters of orders 20 to 1000 performing echo cancellation on speech signals at a sampling rate exceeding 50kHz. For comparison, we implemented a similar NLMS core and found that although it is slightly smaller than the GSFAP core and allows a higher signal sampling rate for the corresponding filter orders, the GSFAP core has adaptation properties that are much superior to NLMS, and that our core can provide very sophisticated adaptive filtering capabilities for resource-constrained embedded systems.

This work was supported and funded by the Czech Ministry of Education, Project CAK No. IM0567, and also by the European Commission, Project AETHER No. FP6-2004-IST-4-027611.
The paper reflects only the authors' view and neither the Czech Ministry of Education nor the European Commission is liable for any use that may be made of the information contained herein.
Authors' addresses: M. Tichy (contact author), and J. Schier, Department of Signal Processing, Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic, Pod Vodarenskou vezi 4, 182 08 Prague 8, Czech Republic; email: {tichy, schier}@utia.cas.cz; D. Gregg, Department of Computer Science, O'Reilly Institute, Trinity College Dublin, Dublin 2, Ireland; email: David.Gregg@cs.tcd.ie.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1539-9087/2010/02-ART29 $10.00
DOI 10.1145/1698772.1698787 http://doi.acm.org/10.1145/1698772.1698787

Categories and Subject Descriptors: B.2.4 [**Arithmetic and Logic Structures**]: High-Speed Arithmetic—*Cost/Performance*; C.3 [**Special-Purpose And Application-Based Systems**]: Signal Processing Systems

General Terms: Algorithms, Design, Performance, Experimentation

Additional Key Words and Phrases: FPGA, DSP, logarithmic arithmetic, affine projection

ACM Reference Format:

Tichy, M., Schier, J., and Gregg, D. 2010. GSFAP adaptive filtering using log arithmetic for resource-constrained embedded systems. ACM Trans. Embedd. Comput. Syst. 9, 3, Article 29 (February 2010), 31 pages. DOI = 10.1145/1698772.1698787 http://doi.acm.org/10.1145/1698772.1698787

1. MOTIVATION

Adaptive filters are widely used in digital signal processing (DSP) for such applications as noise and echo cancellation, and in areas such as data communication systems. A filter applies some function to its input signal to perform the required transformation. An adaptive filter monitors its environment and adapts this function accordingly.

A wide variety of adaptive filtering algorithms have been proposed in the literature. The most important categories of algorithms are perhaps those based on *least mean squares* (LMS) [Widrow and Stearns 1985] and *recursive least squares* (RLS) [Kalouptsidis and Theodoridis 1993; Haykin 2002]. Algorithms based on LMS are fast and simple to implement, but suffer from slow convergence. The RLS-based algorithms converge much faster, but are usually considered too computationally expensive, particularly in applications like echo cancellation where filters with up to several hundred taps are required. Currently, the most widely used adaptive filtering algorithm in resource-constrained embedded systems is a variant of LMS called *normalized LMS* (NLMS) [Haykin 2002]. Faster-converging algorithms are usually considered too expensive for small embedded systems.

More recently, a category of algorithms based on *affine projection* (AP) [Ozeki and Umeda 1984], sometimes referred to as the *generalized NLMS*, have been developed, which provide some compromise between the slow convergence of LMS and the computational complexity of RLS.

In this article we show that a highly optimized core for a faster-converging, more computationally expensive algorithm can be comfortably implemented on a small (one-million-gate) FPGA. The algorithm is the *Gauss-Seidel fast affine projection* (GSFAP) [Albu et al. 2002a], which is perhaps one of the most efficient of the fast AP algorithms. To reduce the resource requirements we represent numbers using the *logarithmic number system* (LNS) [Swartzlander and Alexopoulos 1975; Yu and Lewis 1991; Coleman et al. 2000; Coleman et al. 2008]. Logarithmic arithmetic allows resource-efficient, low-latency multiplication and division at the cost of slightly more expensive addition and subtraction. The resulting GSFAP core can be clocked at over 80MHz on a one-million-gate Xilinx XC2V1000-4 device. To demonstrate the effectiveness of our design, we used the core to implement an adaptive filter of order 1000 performing echo cancellation on speech signals at a sampling rate of more than 50kHz.

The rest of this article is organized as follows. Section 2 provides further background on adaptive filtering. In Section 3 the GSFAP algorithm is described in more detail. The log arithmic number system is introduced in Section 4. Section 5 details the architecture of our GSFAP core. Finally, in Section 6 we evaluate the performance of this architecture.

2. ADAPTIVE FILTERING

The classical approach of digital adaptive filter design is to use the least mean square (LMS) [Widrow and Stearns 1985] algorithm or one of its modifications because these algorithms are simple and extensively described in literature, and thus their behavior is very well known. Such digital filters are relatively easy to implement even in very small electronic devices, and consequently are widely used in various applications. The disadvantage of the classical LMS filter design is its slow convergence. In contrast, the recursive least square (RLS) [Kalouptsidis and Theodoridis 1993; Haykin 2002] algorithm is known as an algorithm with very good convergence properties, however, it has high computational complexity and memory requirements.

In order to reduce the computational complexity of the affine projection algorithm (APA) [Ozeki and Umeda 1984], a fast version of the affine projection algorithm (FAP) [Gay and Tavathia 1995] has been developed. Although the FAP algorithm converges quickly and has low computational requirements, it is actually a rather unpromising algorithm because it is numerically unstable, especially for nonstationary signals. This is due to the use of fast RLS (FRLS) [Cioffi and Kailath 1983; Slock and Kailath 1991] in the algorithm. However, a number of variants have been proposed that solve the numerical stability problems, while maintaining the advantages of FAP. The "modified" FAP (MFAP) [Liu et al. 1996; Kaneda et al. 1995] uses the matrix inversion lemma employed in the classical RLS algorithm, thus avoiding the problems with fast RLS, but at the cost of greater computational requirements. Conjugate gradient (CG) FAP [Ding 2000] uses results of the modified FAP, and uses the conjugate gradient method [Luenberger 1984] to deal with the matrix inversion. The FAP-based algorithm that we believe to be the most suitable for hardware implementation is Gauss-Seidel (GS) FAP [Albu et al. 2002a], which replaces the CG method with the Gauss-Seidel method [Hageman and Young 1981]. This algorithm has all the advantages of modified FAP and CGFAP, but has lower computational complexity, allowing an efficient implementation with fewer hardware resources.

Let us now summarize the computational complexities of algorithms mentioned in previous paragraphs (see Table I). The complexity of the LMS-based algorithms is $\mathcal{O}(L)$, typically $2L + 1$ multiply-accumulate (MACC) operations per iteration, where L is the filter order. The complexity of NLMS is similar, $2L+3$ MACC operations and 1 division per iteration. On the contrary, the complexity of the RLS-based algorithms is $\mathcal{O}(L^2)$. The memory requirements of RLS are also much higher than those of (N)LMS. Fast versions of the RLS algorithm with complexity $\mathcal{O}(L)$ exist, which partly solve the complexity issues. The RLS lattice algorithm requires $18L$ operations and the fast transversal filter (FTF) requires $8L$ operations, or $9L$ in the stabilized form. This, however, comes at

Table I. Summary of the Complexities of Adaptive Filtering Algorithms

Algorithm	Complexity [MACC ops]
LMS	$2L+1$
NLMS	$2L+3$
RLS	$\mathcal{O}(L^2)$
RLS Lattice	$18L$
FRLS	$8L$
FRLS Stabilized	$9L$
APA	$2L+\mathcal{O}(N^3)$
FAP	$2L+20N$
FAP Stabilized	$2L+24N$
Modified FAP	$2L+3N^2+12N$
CGFAP	$2L+2N^2+9N+1$
GSFAP	$2L+N^2+4N-1$

the expense of problems with numerical stability. For large filtering problems (several hundred taps) and real-time implementations, the differences in complexity between the LMS ($2L$)- and the FRLS ($9L$ or $18L$)-based algorithms can be significant. The complexity of FAP is $2L + \mathcal{O}(N)$, where N is the projection order. All other FAP variants mentioned above have complexity $2L + \mathcal{O}(N^2)$, where GSFAP is the least complex, with its complexity being $2L+N^2+4N-1$ MACC operations and N divisions per sample period. In most applications, especially those involving speech, the projection order is almost always very much smaller than the filter order, that is, $N \ll L$, so the time complexity is usually dominated by L rather than N^2. The echo or noise cancellation problems are good examples of such applications. In fact, the GSFAP algorithm has complexity very similar to the original FAP, while being numerically stable.

Using the fast affine projection algorithm brings new qualitative properties to the adaptive filter design. Its main attributes include RLS-like convergence and tracking abilities with LMS-like computational complexity. In other words, it is possible to use very high order digital adaptive filters with improved convergence properties, without a substantial increase in computational complexity.

3. THE GSFAP ALGORITHM

First, a brief review of the GSFAP algorithm is given. The GSFAP algorithm is summarized in Figure 1.

The filter order, that is, the number of filter coefficients, is denoted as L. Individual coefficients (weights) constitute the coefficient vector $\mathbf{w}_k$. All variants of FAP algorithms mentioned in this text do not use the actual (true) filter coefficients $\mathbf{w}_k$ in the adaptation process but use more efficient *alternate coefficient vector* update, introduced in Gay [1993]. This alternate coefficient vector is denoted $\hat{\mathbf{w}}_k$ and has the same number of elements. The number N is referred to as the *projection order*.

The parameters μ and δ are the relaxation factor and the regularization parameter, respectively. The former, μ, represents the algorithm's step-size parameter, which tells us how quickly we move towards the optimal solution (weights). The algorithm is stable for $0<\mu<2$. The latter, δ, is the regularization

Inputs: u_k excitation signal
d_k desired signal

Outputs: y_k filter output
e_k estimation error

Parameters:
$L \in \langle 20; 1000\rangle \subset \mathcal{N}$ filter order
$N \in \langle 2; 20\rangle \subset \mathcal{N}$ projection order
$\mu \in (0; 2) \subset \mathcal{R}$ relaxation parameter
$\delta \in \langle 10^{-3}; 1\rangle \subset \mathcal{R}$ regularization parameter

(0) Initialization:

$$\mathbb{R}_{-1} = \delta\,\mathbb{I}, \quad \mathbf{p}_{-1} = \tfrac{1}{\delta}\,\mathbf{b}, \quad \mathbf{b} = \begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix}^{\mathrm{T}}$$

$$\boldsymbol{\epsilon}_{-1} = \mathbf{0}, \quad \hat{\mathbf{w}}_{-1} = \mathbf{0}, \quad \mathbf{u}_{k<0} = \mathbf{0}, \quad \boldsymbol{\xi}_{k<0} = \mathbf{0}$$

(1) Update $\mathbb{R}_k$:

a) $u_k \dashrightarrow \mathbf{u}_k,\ \boldsymbol{\xi}_k$

b) $\mathbb{R}_k = \mathbb{R}_{k-1} + \boldsymbol{\xi}_k\boldsymbol{\xi}_k^{\mathrm{T}} - \boldsymbol{\xi}_{k-L}\boldsymbol{\xi}_{k-L}^{\mathrm{T}}$

(2) Update $\mathbf{p}_k$ (Gauss-Seidel iteration):

for $(i = 0;\ i < N;\ i = i + 1)$

$$p_{i,k} = \left[b_i - \sum_{j=o}^{i-1} R_{ij,k}p_{j,k} - \sum_{j=i+1}^{N-1} R_{ij,k}p_{j,k-1}\right] / \, R_{ii,k}$$

end;

(3) Compute e_k:

a) $y_k = \mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1} + \mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$

b) $e_k = d_k - y_k$

(4) Update $\boldsymbol{\epsilon}_k$ and $\hat{\mathbf{w}}_k$:

a) $\boldsymbol{\epsilon}_k = e_k\mathbf{p}_k + \begin{bmatrix}0\\ \bar{\boldsymbol{\epsilon}}_{k-1}\end{bmatrix}$

b) $\hat{\mathbf{w}}_k = \hat{\mathbf{w}}_{k-1} + \mu\mathbf{u}_{k-N+1}\epsilon_{N-1,k}$

Fig. 1. The Gauss-Seidel Fast Affine Projection (GSFAP) algorithm.

parameter for the autocorrelation matrix inverse. It prevents the autocorrelation matrix $\mathbb{R}$ from becoming singular.

Step 0 initializes the key variables. Steps 1 through 4 represent one iteration of the GSFAP algorithm. The index $\cdot_k$ is the “discrete time” (iteration) index; if more indices are given, it is always the last one which denotes the iteration index; for example, $p_{i,k}$ denotes i-th element of vector $\mathbf{p}_k$.

In Step 1, new data—input and desired signal samples u_k and d_k—are acquired; and the excitation signal vector $\mathbf{u}_k$, vector $\boldsymbol{\xi}_k$ (Step 1a) and the correlation matrix $\mathbb{R}_k$ (Step 1b) are updated.

The excitation signal vector $\mathbf{u}_k$ is a vector consisting of the delayed sequence of L input signal samples, that is,

$$\mathbf{u}_k = [\, u_k \;\; u_{k-1} \;\; \dots \;\; u_{k-L+1} \,]^{\mathrm{T}}. \tag{1}$$

The vector $\boldsymbol{\xi}_k$ consists of the delayed sequence of N input signal samples, which means that the vector $\boldsymbol{\xi}_k$ overlaps the first N elements of vector $\mathbf{u}_k$. As we will

see in Section 5, this property is important for saving hardware resources, on-chip Block RAMs in particular. The vector $\boldsymbol{\xi}_{k-L}$, required for the update of $\mathbb{R}_k$, has the same structure as $\boldsymbol{\xi}_k$ but it represents the state at time $k-L$.

The matrix $\mathbb{R}_k$ is an N by N matrix, which is the auto-correlation matrix of the excitation signal and it is symmetric and positive definite. Since the matrix $\mathbb{R}_k$ is symmetric, its update involves $N(N+1)$ multiply-accumulate operations, and $\frac{N(N+5)}{2}$ read and N^2 write memory accesses (provided that the whole matrix is held in memory—not only an upper or lower triangle).

In step 2, the vector $\mathbf{p}_k$ is calculated. This vector is in fact the first column of the inverse of matrix $\mathbb{R}_k$. This problem is equivalent to solving a set of linear equations $\mathbb{R}_k\mathbf{p}_k = \mathbf{b}$, where $\mathbf{b}$ is a vector of length N, in which all elements have the value zero, except for the first element which has the value one. As shown in Albu et al. [2002a], one iteration of the Gauss-Seidel (GS) method for solving a set of N linear equations provides a good estimate of the actual vector $\mathbf{p}_k$ using the vector $\mathbf{p}_{k-1}$ as initial value for each GS iteration. The symbol $R_{ij,k}$ denotes the ij-th element of the matrix $\mathbb{R}_k$ and the symbol $p_{i,k}$ denotes the i-th element of the vector $\mathbf{p}_k$. One full GS iteration requires N^2 additions/subtractions, $N(N-1)$ multiplications, N divisions, and $2N(N-1)$ read and N write memory accesses.

Step 3 represents an efficient method for calculation of the filter output y_k (Step 3a) and of the estimation error e_k (Step 3b) using alternate coefficient vector $\hat{\mathbf{w}}_k$ rather than original weight vector $\mathbf{w}_k$. Vector $\boldsymbol{\epsilon}$ is called the normalized estimation error vector and is of dimension N. The symbol $\bar{\epsilon}_{k-1}$ denotes an $N-1$ vector consisting of the $N-1$ uppermost elements of vector $\boldsymbol{\epsilon}_{k-1}$ and the symbol $\tilde{\mathbf{R}}_{0,k}$ represents an $N-1$ vector that consists of the $N-1$ lowermost elements of the first (left) column of the matrix $\mathbb{R}_k$. The calculation of the filter output y_k consists of two dot-product operations, one of length L and the other of length $N-1$, and of one additional multiply-accumulate to multiply the dot product $\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$ by μ and to add the result to the term $\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$. It is evident that the calculation of y_k requires $L + N$ multiply-accumulate operations and $2(L + N - 1)$ read memory accesses. Then, for calculation of e_k just one additional subtraction is required.

The normalized estimation error vector $\boldsymbol{\epsilon}_k$ and consequently the alternate coefficient vector $\hat{\mathbf{w}}_k$ are updated in Step 4. Both manipulations are based on a simple multiply-accumulate operation. After the normalized estimation error $\boldsymbol{\epsilon}_k$ (Step 4a) has been updated, the excitation signal vector $\mathbf{u}$ at time $k-N+1$ and the lowermost element of the newly updated vector $\boldsymbol{\epsilon}_k$ denoted as $\epsilon_{N-1,k}$ are used to update the alternate coefficient vector $\hat{\mathbf{w}}_k$ (Step 4b). The first operation requires N multiplications, $N-1$ additions and $2N-1$ read and N write memory accesses; the latter $L+1$ multiplications, L additions and $2L$ read and L write memory accesses. Finishing this step, one full iteration of the GSFAP is completed.

4. LOGARITHMIC ARITHMETIC

In order to maintain accuracy of the algorithm in the FPGA implementation, we decided to implement the computations using a floating-point-like arithmetic.

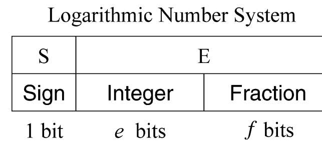

Fig. 2. Format of the Logarithmic Number System representation.

Floating-point arithmetic units are often large and power-hungry, and usually have long pipeline latencies. Traditionally, multiplication, division, and square root were particularly expensive to implement. Recent Xilinx Virtex-2 and Virtex-4 FPGAs have included hard-wired logic, which implements 18-bit integer multiplication. These hardware multipliers can be used to implement the mantissa multiplication step in floating-point multiplication, greatly reducing the cost of floating-point multiplication units. However, even with hardware multipliers, such units are still large, and often have long latencies. Furthermore, floating-point division and square root remain expensive, and even floating-point addition units are far from cheap.

An alternative to floating-point is to represent real (non-integer) numbers in log arithmic form. The log arithmic number system (LNS) has been proposed many times, but has been little used since the advent of hardware floating-point units in general purpose processors. However, on an FPGA, multiplication, division, and square root of log arithmic numbers require only very simple logic to compute, making LNS ideal for algorithms that contain a large number of such operations. Addition and subtraction are more complex in LNS, but recent advances have made them practical even on small FPGAs.

4.1 LNS Data Representation

The log arithmic number system data representation is depicted in Figure 2. A log arithmic number consists of two parts: a sign digit S and a logarithm E that includes an integer part of length e and a fractional part of length f. The sign bit S is set to 0 if the number is positive and to 1 if the number is negative; E is the logarithm of the absolute value of a real number X to be represented.

For our representation in LNS, base-2 logarithm has been chosen. The logarithm E can thus be represented as a two's complement fixed-point value equal to $\log_2|X|$, where X is the value to be represented and $|\cdot|$ is the absolute value operator. Then, the value of X can be expressed as

$$X = (-1)^S \cdot 2^E. \quad (2)$$

It should be noted that the LNS representation can be considered as an extreme case of the floating-point number system with the significand always equal to 1 and the exponent represented as a fixed-point number (with integer and fractional parts) rather than an integer.

For the 32-bit LNS precision, the integer part is of length $e = 8$ and the fraction part is of length $f = 23$. There are two special values—zero and NaN—which have to be represented by a specific sequence of bits, where E has its leftmost bit set to 1 and all remaining bits to 0; if $S = 0$, the LNS number represents zero; if $S = 1$, the LNS number represents NaN.

The largest and the smallest values of the LNS fixed-point parts, denoted E_{max} and E_{min}, respectively, can be defined as

$$E_{max} = 2^{e-1} - 2^{-f} \quad (3)$$

$$E_{min} = -2^{e-1} + 2^{-f}. \quad (4)$$

Then, according to (2), the largest and the smallest positive real numbers representable by the LNS are $X^+_{max} = 2^{E_{max}}$ and $X^+_{min} = 2^{E_{min}}$, respectively.

The standard IEEE single-precision floating-point representation [Institute of Electrical and Electronics Engineers, Inc. 1985] uses a sign bit, 8-bit biased exponent, and (23 + 1)-bit significand. This format is able to represent signed values within the range $\approx 1.17 \cdot 10^{-38}$ to $3.4 \cdot 10^{38}$. In the equivalent 32-bit precision LNS representation, the integer and fractional parts are kept as coherent two's complement fixed-point value within the range ≈ -128 to 128, according to (4) and (3). Then, the real numbers representable by LNS are signed and within the range $\approx 2.9 \cdot 10^{-39}$ to $3.4 \cdot 10^{38}$.

4.2 LNS Operations

In the LNS, a value X is represented as the fixed-point quantity $i = \log|X|$, with an extra bit to indicate the sign of X and a special arrangement to accommodate zero and NaN. Base-2 logarithms are used, though in principle any base could be used.

Given two LNS values $i = \log_2|X|$ and $j = \log_2|Y|$, the LNS addition, subtraction, multiplication, division, and square root can be defined by the following set of equations:

$$\log_2(X + Y) = i + \log_2(1 + 2^{j-i}) \quad (5)$$

$$\log_2(X - Y) = i + \log_2(1 - 2^{j-i}) \quad (6)$$

$$\log_2(X \cdot Y) = i + j \quad (7)$$

$$\log_2\left(\frac{X}{Y}\right) = i - j \quad (8)$$

$$\log_2(\sqrt{X}) = \frac{i}{2}, \quad (9)$$

where in (5) and (6), without loss of generality, we choose $j \leq i$. In all these cases the sign bits are handled separately, using the same rules as in the case of floating-point operations.

It is evident that Equations (7), (8), and (9) can be implemented as simple as fixed-point addition, subtraction and shift operations, respectively. Unfortunately, as mentioned above, Equations (5) and (6) require evaluation of a nonlinear function

$$\mathcal{F}(r = j - i) = \log_2(1 \pm 2^r), \quad (10)$$

which can be seen in Figure 3. To evaluate these functions look-up tables are used containing values only at intervals through the functions. Intervening values are obtained by interpolation using the first-order Taylor series approximation. The look-up tables are kept reasonably small using the *error correction*

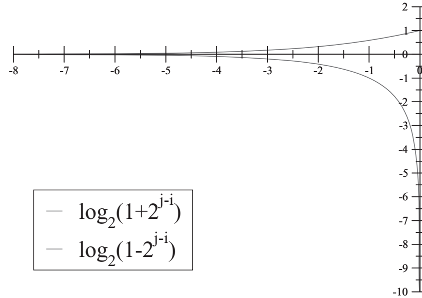

Fig. 3. Nonlinear functions that have to be evaluated using approximation, when performing the LNS addition and subtraction.

mechanism and for the case of log arithmic subtraction using the *range-shift algorithm* for values within the range $-0.5 < r < 0$. Both techniques as well as the approximation tables structure are described in Coleman [1995] and Coleman et al. [2000].

4.3 Hardware LNS Cores

The LNS arithmetic cores [Matousek et al. 2002] used for our implementation were originally developed within the project High Speed Logarithmic Arithmetic (HSLA). Since the completion of this project we have created an updated version with additional features, and higher clock speeds.

We use 32-bit and 19-bit LNS precisions, where the integer part is always of length $e = 8$ and the fraction part is of length $f = 23$ and $f = 10$, respectively. There are two special values—zero and NaN—that have to be represented by a specific sequence of bits, where E has its leftmost bit set to 1 and all remaining bits to 0; if $S = 0$, the LNS number represents zero; if $S = 1$, the LNS number represents NaN.

As presented in Section 4.1, the 32-bit LNS is comparable to the IEEE single precision floating point, in terms of the range and precision. The 19-bit precision LNS format maintains the same range as 32-bit but has precision reduced to 10 fractional bits. It is comparable to the 16-bit floating-point formats used in commercial DSP devices. All units are available both in 32-bit and in 19-bit versions.

All hardware macros (cores) contain logic for handling exceptions, that is, evaluating invalid operations, performing overflow or underflow checking. The multiplication, division, and square root (MUL/DIV/SQRT) modules are implemented as macros with a latency of 1 cycle. The LNS addition/subtraction (ADD/SUB) module is designed as 8-stage fully pipelined arithmetic unit with two separate adder/subtracter pipelines. The look-up tables are stored in dual ported on-chip Block RAMs. The main reason the adder/subtracter units have two separate pipelines is the presence of two ports on the Block RAMs; a dual pipeline adder/subtracter is not much more expensive than one with a single pipeline. The architecture of individual LNS arithmetic units is described in detail in Tichy [2006].

Table II. Comparison of Underwood's Highly Optimized IEEE Single Precision Floating-Point Units with the 32-bit LNS Units in a Xilinx Virtex-II XC2V6000 (6-million gate) FPGA

	ADD		2-pipe ADD		MUL		DIV	
	FLP	LNS	FLP	LNS	FLP	LNS	FLP	LNS
Slice Flip Flops	696	—	1,392	1,702	821	35	2,476	35
4 input LUTs	611	—	1,222	2,135	722	139	2,220	145
Occupied Slices	496	—	992	1,648	598	83	1,929	82
Block RAMs	0	—	0	28	0	0	0	0
MULT18X18s	0	—	0	8	4	0	0	0
Latency [cc]	13	—	13	8	16	1	37	1
Clock rate* [MHz]	165	—	165	80	124	200	100	200

*Figures for clock rate are informative only and can vary according to the context in which the units are used. FLP data are related to XC2V6000-5 (speed grade 5); LNS data are related to XC2V6000-4 (speed grade 4).

4.4 Comparing LNS with Floating-Point

To demonstrate advantages and disadvantages of our LNS arithmetic units more clearly, we compare these arithmetic units with Underwood's IEEE floating-point (FLP) units [Underwood 2004]. Underwood developed highly optimized IEEE single (32-bit) and double (64-bit) precision floating-point units. Since our 32-bit LNS arithmetic corresponds to the IEEE single precision arithmetic [Institute of Electrical and Electronics Engineers, Inc. 1985], we exclude double precision floating-point and 19-bit LNS arithmetics from our comparisons. Table II shows the resource requirements and parameters of our 32-bit LNS units compared to Underwood's IEEE single precision floating-point units.

4.4.1 *Chip Area.* The major disadvantage of the LNS arithmetic is the number of on-chip Block RAMs occupied by the adder/subtracter (ADD/SUB) unit for storing look-up tables. LNS ADD/SUB units are always instantiated in pairs. The main reason the adder/subtracter unit has two separate pipes is the presence of two ports on Block RAMs, which means a dual-pipe ADD/SUB unit is not much more expensive than the one with a single pipe. It is evident that resource requirements for a pair of LNS ADD/SUB pipes is significantly higher than for a pair of standard 32-bit floating-point ADD/SUB units. With regard to area (slices), a pair of FLP ADD/SUB units occupies around 60% of a pair of LNS ADD/SUB units. This apparent drawback in size of units is, however, compensated for by other parameters and the advantages of other LNS units.

The LNS multiplier (MUL) unit occupies only a small fraction (around 14%) of the size of the floating-point multiplier unit—even when embedded (on-chip) multipliers are used. Perhaps the most common operation in many DSP and matrix algorithms is multiplication and addition. When we sum the resources required by a single multiply-add pipe, we can decide that using floating-point units is still preferable. The situation is different when using two multiply-add pipes: we can see that the LNS units require fewer resources (except for Block RAMs). Notwithstanding these facts, many DSP algorithms require division and/or square root operations. While figures for a floating-point square root unit are not available, data for a divider (DIV) unit (a LNS DIV unit occupies less than 5% of an FLP DIV unit) show that using the LNS architecture can result in a considerable advantage. One can argue that the number of Block

RAMs occupied by the LNS ADD/SUB unit is insuperable obstacle for a practical application of the LNS architecture. It is apparent that the decision about what kind of architecture or arithmetic would be the most suitable platform for a particular problem always depends on more than one factor. As we will see later in this section, the number of used Block RAMs is not always a limiting factor, even in considerations related to chip area.

4.4.2 *Clock Speed and Latencies.* Another important issue is the clock speed and latencies of functional units. Underwood's adder/subtracter unit can be clocked at up to 165MHz on a Virtex-II XC2V6000-5 FPGA (six-million gate, speed grade 5 device), with a latency of up to 13 clock cycles. The FLP multiplier unit can be clocked at about 125MHz on the same device with a latency of 16 clock cycles, whereas the FLP divider unit can be clocked at 100MHz, but with a latency of 37 clock cycles. Underwood's units are highly configurable, and the latency can be reduced at the cost of a corresponding reduction in clock speed. In contrast, latencies of the LNS adder/subtracter, multiplier, and divider units are 8, 1, and 1 clock cycles, respectively. The LNS adder/subtracter can be clocked at up to 80MHz; the LNS multiplier and divider at 200MHz. These figures were obtained for a Virtex-II XC2V6000-4 FPGA (six-million gate, speed grade 4 device) which is approximately 10%–15% slower than the Virtex-II XC2V6000-5 FPGA [Xilinx, Inc. 2005], so we can reasonably expect the LNS adder/subtracter unit to reach 90MHz on a speed grade 5 device. While the LNS adder/subtracter unit is about 40% slower than a coresponding floating-point unit, the multiplier and divider units are dramatically faster. If comparing the clock speed of LNS and FLP units, we have to apply figures of the slowest units: FLP DIV and LNS ADD/SUB.

In addition to the clock speed, the latencies of arithmetic units are also important when comparing LNS with floating point. This is particularly true if an arithmetic module is on the critical path. As an example, we use implementation of our GSFAP core whose architecture is presented in Section 5. By inspection of Step 2 of the GSFAP algorithm in Figure 1, we can see that the divider is required in each iteration of the Gauss-Seidel procedure. The FLP divider has a latency of 37 clock cycles whereas the latency of the LNS divider is 1 clock cycle. Using the floating point would result in increasing of latency of the GSFAP core by $36N$ clock cycles. This is a significant difference in performance. Thus, LNS arithmetic units can result in a substantially faster design when compared with using floating point, depending on the mix of operations in the algorithm. The difference in performance may be particularly large when division or square root operations are on the critical path of the algorithm.

4.4.3 *Precision and Accuracy.* A final important issue is the precision and accuracy of operations. All floating-point operations are liable to a maximum half-bit rounding error [Koren 2002; Institute of Electrical and Electronics Engineers, Inc. 1985]. The LNS add and subtract operations show the same or smaller rounding error [Coleman and Chester 1999; Coleman et al. 2000], except for subtraction where operands are closely matched. Given that LNS multiplication and division are implemented as fixed-point addition and subtraction,

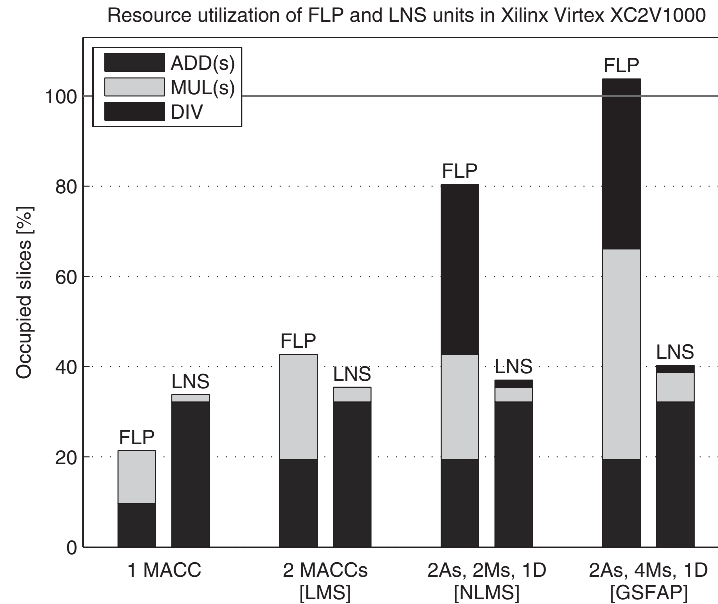

Fig. 4. Resource utilization of Underwood's highly optimized IEEE single precision floating-point units and of the 32-bit LNS units when used for implementation of various algorithms in a Xilinx Virtex-II XC2V1000 (1-million gate) FPGA.

respectively, no rounding takes place. In order to confirm the precision of LNS, we compared the signal-to-noise (SNR) ratios for the 32-bit floating-point and the 32-bit LNS implementations of several algorithms (using various parameters). The double precision floating-point implementations were used as a baseline. We obtained comparable results for both FLP and LNS in most cases. More details on these measurements can be found in Section 6.1. These results are not unexpected for algorithms with a roughly equal balance of additions and multiplications, because the additions still have a half-bit rounding error, although the multiplications introduce no rounding error. Clearly the benefits to be gained by using LNS will vary depending on the ratio of add/subtract to multiply/divide operations and the sign and range of the operands. A more detailed discussion on the precision and accuracy of the LNS arithmetic used in our implementation including the corresponding signal-to-noise ratios can be found in Coleman et al. [2008].

4.4.4 *Resource Utilization for Different Applications.* To demonstrate and compare FLP/LNS area requirements for various combinations of arithmetic units, we use a smaller Xilinx Virtex-II XC2V1000 FPGA as a normalized area. Figure 4 shows resource utilization of Underwood's single precision floating-point units and the 32-bit LNS units when used for implementation of various algorithms in a one-million gate Virtex-II XC2V1000 chip. For a single multiply-accumulate (MACC) module, we can see that using floating-point units would be more effective. In the case of two MACC modules, using an LNS unit requires slightly less resources (excluding on-chip Block RAMs). A typical example where

two MACCs can be effectively employed is the LMS algorithm. The most appropriate arithmetic for these kind of applications is not always clear. Fixed point has some attractions because of the simplicity of such algorithms, and provided the application can accept the precision and range of values offered by fixed point. However, the situation changes dramatically when there is a need for a divider. Typical examples of such applications are adaptive filters based on NLMS or GSFAP algorithms. It can clearly be seen that the area required by LNS units is less than 50% of the area needed by FLP units for the NLMS algorithm; using floating-point units for the implementation of GSFAP algorithm would not be feasible, given that four multipliers are required for an efficient implementation of the algorithm.

4.4.5 *Conversions.* In order to use the LNS arithmetic, input numbers may have to be converted to the LNS format and the output may have to be converted back to the required format. Conversions will be required if the whole system does not use the LNS to represent real numbers, and instead fixed or floating point numbers are used in other parts of the system.

On first sight, such conversions appear to be a major disadvantage of LNS arithmetic. However, the kind of applications that we target usually take their input from an analog to digital (A/D) converter and direct their output to a digital to analog (D/A) converter. Thus, conversion modules from fixed-point (FXP) to LNS and vice versa are required. Such units supporting FXP $\Leftrightarrow$ LNS conversions can be implemented in a relatively inexpensive way. Our conversion unit requires that the "FXP" numbers are in the format of two's complement fraction part of the fixed point value within the range $\langle -1; 1\rangle$.[1] The conversion of a fixed point value to the LNS format is based on decomposing the FXP number, using the look-up tables and log arithmic addition/subtraction operations. Since conversion modules employ an LNS ADD/SUB unit, they require only one supplementary Block RAM and a little additional logic beyond what is already required for ADD/SUB. In particular, the 24-bit fixed-point $\Leftrightarrow$ 32-bit LNS conversion unit occupies 246 slices (296 Flip-Flops and 250 4-input LUTs) of a Virtex-II device. The latency of the FXP $\Rightarrow$ LNS is 19 and of the LNS $\Rightarrow$ FXP is 46 clock cycles. Both operations are partially pipelined, so FXP $\Rightarrow$ LNS can process new data every second clock cycle, while LNS $\Rightarrow$ FXP every ninth clock cycle. If necessary this unit can also be used for FLP $\Leftrightarrow$ LNS conversions. The principles of conversions are described in Pohl et al. [2003].

In most DSP applications hundreds or thousands of log operations must typically be performed for every data sample that needs to be converted. The only major exception to this that we are aware of is image processing, an area that we do not focus on. In view of this fact and considering figures presented in the previous paragraph, conversions involved in a system that uses the LNS are not a limiting factor.

---

[1]Note that many A/D converters simply return an N-bit (often 8-, 16-, 20- or 24-bit) two's complement number as their output, and it is up to the designer of the system to interpret this value as a fixed point number and scale it if necessary. All our implementations have used a linear A/D converter, and thus require an FXP $\Leftrightarrow$ LNS converter. However, some other A/D converters return a value that is non-linearly related to the phenomenon it is measuring.

Also, it should be noted that in communications and audio-processing, so-called *companding* A/D and D/A converters [Kikkert 1974; Tsividis et al. 1990; Whitehouse 2006] with log arithmic characteristics are used. These converters, employing a-law or μ-law algorithm, are usually implemented either in hardware, or in a DSP directly at the point of data conversion. Using the output of one of these algorithms would probably result in savings in the terms of FPGA resource utilization, since no special LNS conversion would be needed. However, we have not studied this option yet in the terms of SNR and resulting precision, and we feel that this topic goes somewhat beyond the scope of this article.

4.4.6 *Summary.* Taking all the facts stated above into consideration, we can conclude that a 32-bit LNS arithmetic unit can perform with substantially better area usage, speed, and accuracy than a floating point unit, particularly when more multipliers or dividers are required. A further advantage of using LNS arithmetic is that the designer must focus only on efficiently using ADD/SUB units, because they are the only substantial hardware units. The other units are so small and fast that they could be replicated to simplify the design or reduce routing delays. In contrast, designing a system using FLP arithmetic may be more complicated because the adders, multipliers, dividers, and square root units are all substantial pieces of hardware that must be used efficiently to get good overall performance.

Lastly, it should be noted that Haselman et al. [2005] also discuss the advantages and disadvantages of the LNS arithmetic. Although those authors come to similar conclusions, their work completely omits information on the accuracy of their implementation of LNS add/subtract operations in relation to the size of the approximation tables.

5. GSFAP ARCHITECTURE

In this section we present the architecture of our GSFAP core and describe the structure of its individual functional units. The algorithm employs one LNS addition/subtraction dual-pipe unit (denoted ADD/SUB `A` and `B`—two separate, parallel pipelines), four multiplication units (denoted MUL `A`, `B`, `C`, and `D`) and one division (denoted DIV `A`). Nonscalar data structures, that is, vectors and a matrix, are stored in on-chip dual-port Block RAMs.

The top-level architecture of the GSFAP unit is depicted in Figure 5. Blocks in the diagram roughly correspond to individual steps of the algorithm presented in Figure 1. The blocks and operations depicted on the left-hand side of the diagram employ the first ADD/SUB pipeline (pipeline `A`) while the operations on the right-hand side employ the second pipeline of the ADD/SUB unit (pipeline `B`). The block denoted "`UX` update" does not use ADD/SUB unit at all and the block denoted "`WW[0,1]` update" utilizes both pipelines. The time line on the far right side of the diagram indicates the clock cycles during which different parts of the design are active in the GSFAP based filter with the parameters $L = 1000$ (filter order) and $N = 9$ (projection order). For example, the right-hand dot-product unit (which computes $\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$) is active from clock cycle 2 to cycle 1043, and uses pipeline `B` of the ADD/SUB unit.

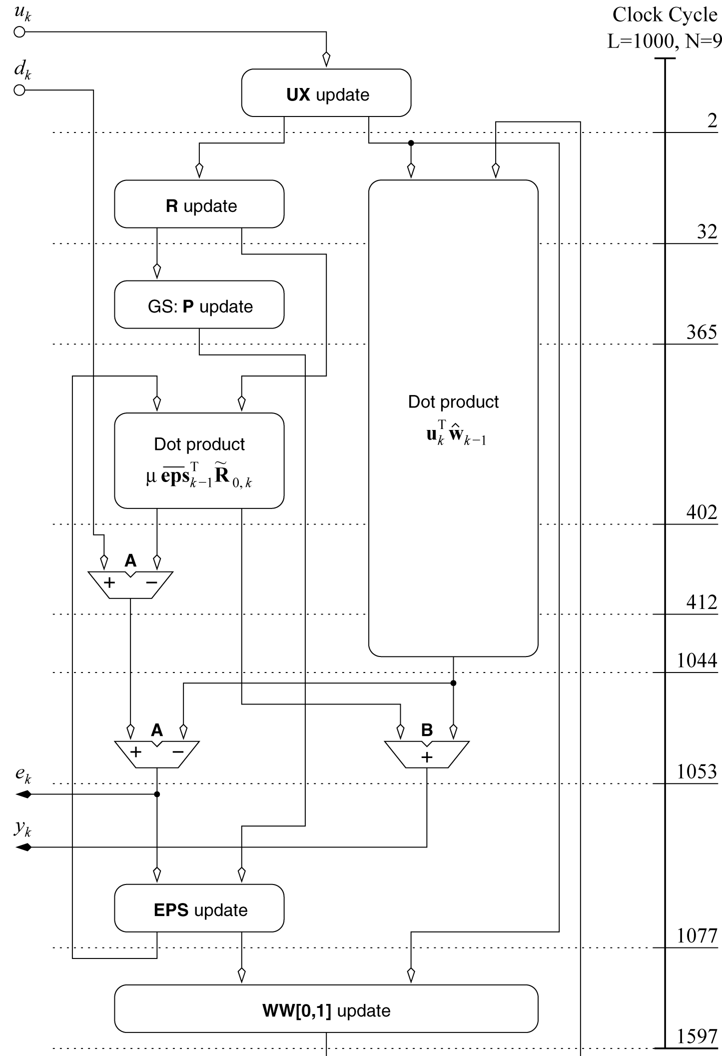

Fig. 5. Block diagram and data dependencies of the top-level architecture of GSFAP unit.

5.1 Excitation Signal Vector (Block RAM UX) Update

In the following paragraphs, a description of individual components and data structures is given. The first task is to acquire new input signal samples u_k and d_k, and to update the excitation signal vector $\mathbf{u}_k$. It is also necessary to update two vectors $\boldsymbol{\xi}_k$ and $\boldsymbol{\xi}_{k-L}$ which are needed for the update of correlation matrix $\mathbb{R}_k$. Another part of the algorithm where input signal samples are needed is the update of the alternate coefficient vector $\hat{\mathbf{w}}_k$: it is necessary to keep a delayed excitation signal vector $\mathbf{u}_{k-N+1}$ (see Step 4b in Figure 1). As mentioned in Section 3, the vector $\boldsymbol{\xi}_k$ overlaps the first N elements of vector $\mathbf{u}_k$. With respect to the structure of $\mathbf{u}_k$ it is apparent that the first $L-N+1$ elements of $\mathbf{u}_{k-N+1}$ overlap the last $L-N+1$ elements of $\mathbf{u}_k$. Similarly, the last $N-1$ elements

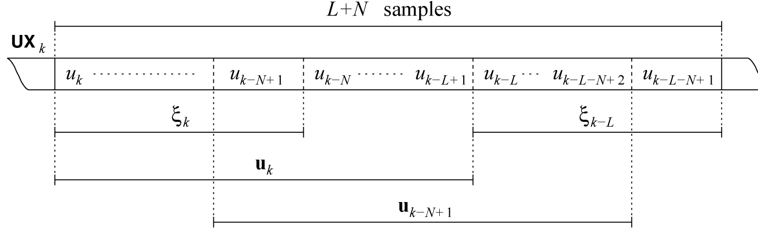

Fig. 6. Contents of the UX Block RAM consisting of a delayed sequence of the most recent $L + N$ input signal samples.

of $\mathbf{u}_{k-N+1}$ are overlapped by the first $N - 1$ elements of $\boldsymbol{\xi}_{k-L}$. Thus all vectors consisting of a sequence of input signal samples, $\mathbf{u}_k$, $\mathbf{u}_{k-N+1}$, $\boldsymbol{\xi}_k$ and $\boldsymbol{\xi}_{k-L}$, can be stored in a single memory block of length $L + N$ that we refer to as `UX` memory (or vector). The contents of `UX` memory at time k is shown in Figure 6.

The module for updating this `UX` storage is denoted "`UX` update" in Figure 5. The `UX` storage is implemented as a circular buffer, to avoid shifting each time a new input signal sample is acquired. The state of the buffer is kept in the register `UX_state` that is of the same width as the number of `UX` address lines. To update `UX`, we only need to take two steps: (1) decrement the register `UX_state` and (2) write a new data sample u_k to `UX` to the position given by the value in that register, that is, to `UX[UX_state]`. Since this operation consists of storing one value into a Block RAM and decrementing a single register, it takes, together with acquiring new data, only 2 clock cycles regardless of L or N.

5.2 Update of the Correlation Matrix $\mathbb{R}_k$

The next module of the GSFAP unit is denoted "`R` update"; its job is to update the autocorrelation matrix of the excitation signal $\mathbb{R}_k$. To update the correlation matrix, we need to calculate

$$\mathbb{R}_k = \mathbb{R}_{k-1} + \boldsymbol{\xi}_k \boldsymbol{\xi}_k^{\mathrm{T}} - \boldsymbol{\xi}_{k-L} \boldsymbol{\xi}_{k-L}^{\mathrm{T}}. \tag{11}$$

Since this matrix is symmetric, its update requires $N(N + 1)$ multiply-accumulate operations, and $\frac{N(N+5)}{2}$ read and N^2 write memory accesses (provided that the whole matrix is held in memory—not just an upper or lower triangle). However, it can be shown [Tichy 2006] that we can get the same result at a much lower cost. Let us define the *correlation vector*

$$\mathbf{r}_k = \left[r_{0,k} \;\; r_{1,k} \;\; \ldots \;\; r_{N-1,k} \right]^{\mathrm{T}}, \tag{12}$$

which is actually the first column (or row) of the matrix $\mathbb{R}_k$. This correlation vector can be updated in each iteration as follows.

$$\mathbf{r}_k = \mathbf{r}_{k-1} + \xi_{0,k} \boldsymbol{\xi}_k - \xi_{0,k-L} \boldsymbol{\xi}_{k-L}, \tag{13}$$

where $\xi_{0,k}$ and $\xi_{0,k-L}$ are the first (uppermost) elements of the vectors $\boldsymbol{\xi}_k$ and $\boldsymbol{\xi}_{k-L}$, respectively. The matrix $\mathbb{R}_k$ can be constructed so that each element of the original matrix $\mathbb{R}_{k-1}$ is shifted diagonally (along the main diagonal) by 1 and the vector $\mathbf{r}_k$ is used as the first column/row of the matrix $\mathbb{R}_k$, where $\mathbf{r}_k$ has been updated according to (13). The operation is schematically depicted in

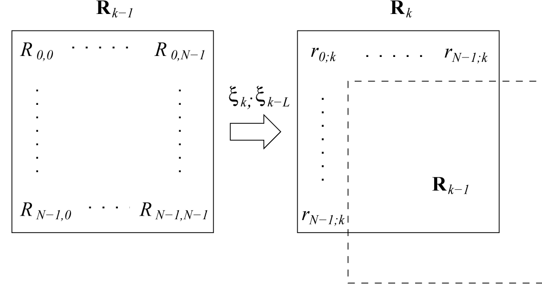

Fig. 7. Schematic representation of the update of matrix $\mathbb{R}_k$ by shifting the original matrix $\mathbb{R}_{k-1}$ diagonally and replacing its first column/row with the updated correlation vector $\mathbf{r}_k$.

Figure 7. On the assumption that we can implement shifting of the left-upper submatrix diagonally without excessive memory access (see "reindexing" in the following), this procedure requires only $2N$ multiply-accumulate operations, and N read and $2N - 1$ write memory accesses, which represents dramatic savings of operations compared to the updating of $\mathbb{R}_k$ according to (11).

A fundamental question is how to store the matrix $\mathbb{R}_k$ in memory. This matrix is symmetric, which suggests that resource usage could be minimized by storing only its upper or lower triangle. However, we decided to store the whole matrix in a Block RAM rather than its upper/lower triangle, even though it almost doubles memory requirements. The reasons for such decision are as follows.

(1) The dimensions of $\mathbb{R}$ are the same as the projection order, which is reasonably small. A matrix $\mathbb{R}$ with dimensions of up to $N = 22$ will fit in a single on-chip (Virtex-II) Block RAM, provided that a 32-bit arithmetic is used. Experiments show that using a projection order higher than $N = 20$ does not improve algorithm performance in practice, that is, its convergence and tracking properties.
(2) The update of the matrix $\mathbb{R}$ can be implemented very efficiently using "reindexing" (see the following).
(3) Control logic necessary to calculate addresses of individual elements of the matrix is simpler and faster.

The memory storage for the matrix $\mathbb{R}$ is referred to as `R` Block RAM. Individual elements of $\mathbb{R}$ are stored in `R` in a column-wise manner.

Before we describe the actual hardware structure that can be used to update `R` storage, it is essential to answer the following question: "How can we shift the $N-1$ by $N-1$ left-upper submatrix of $\mathbb{R}_{k-1}$ along the main diagonal to get the $N-1$ by $N-1$ right-lower submatrix of $\mathbb{R}_k$ without superfluous accesses to a memory?" The answer is obvious—a mechanism for rearranging indices of appropriate elements of submatrix $\mathbb{R}_{k-1}$ so that they become elements of submatrix $\mathbb{R}_k$. Our implementation of such procedure is based on a technique similar to using a circular buffer, which we call "reindexing." The principle of updating the storage `R` is depicted in Figure 8. The actual state of the buffer is kept in the register `R_state`, which is decremented by the value $N+1$; and the

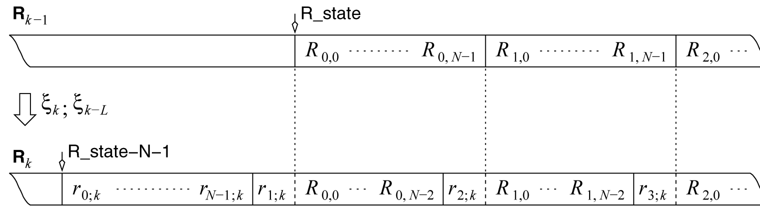

Fig. 8. The principle of "reindexing" using a circular buffer: updating of the matrix $\mathbb{R}_k$ by shifting the original matrix $\mathbb{R}_{k-1}$ diagonally and replacing its first row/column with the updated correlation vector $\mathbf{r}_k$ in the Block RAM R.

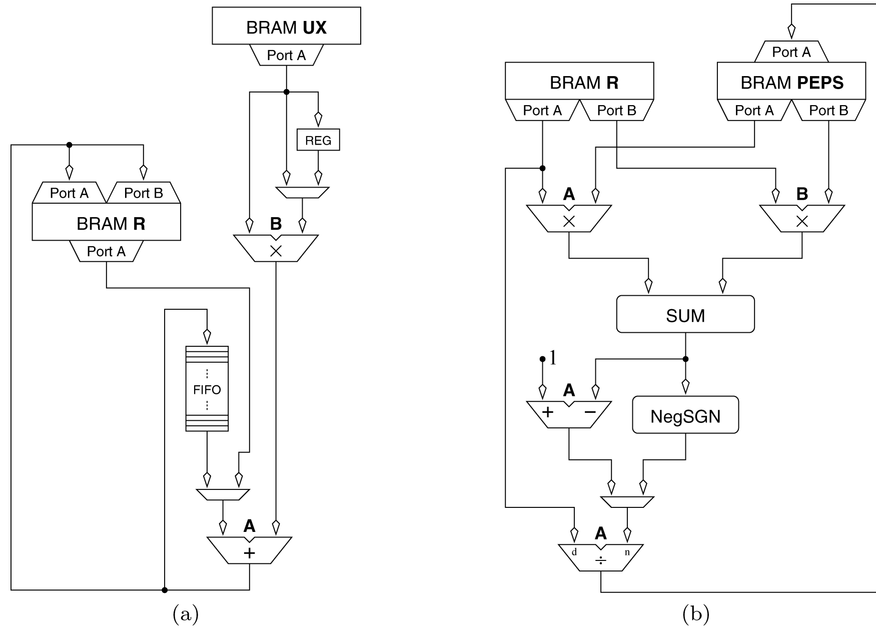

Fig. 9. (a) Architecture schematic of the update of correlation matrix $\mathbb{R}_k$ stored in the Block RAM R; (b) Architecture schematic of the one step—calculation of the i-th element of vector $\mathbf{p}_k$—of the Gauss-Seidel (GS) iteration procedure.

elements of updated correlation vector $\mathbf{r}_k$ are stored into positions corresponding to elements of the first row/column of matrix $\mathbb{R}_k$.

Then, the "R update" module can be implemented very effectively in hardware, the architecture of which is schematically depicted in Figure 9(a). The values of vector $\boldsymbol{\xi}_k$ are read from the Block RAM UX and the multiplication $\xi_{0,k}\boldsymbol{\xi}_k$ using the MUL B unit is invoked. Concurrently, another "process" controls reading values from the Block RAM R to get the values of vector $\mathbf{r}_{k-1}$ (the first column/row of $\mathbb{R}_{k-1}$). The results of multiplication are successively added to the values read from R using the ADD/SUB A unit, which results in implementation of $\mathbf{r}_{k-1} + \xi_{0,k}\boldsymbol{\xi}_k$. After all values of $\mathbf{r}_{k-1}$ have been read from the Block RAM R, the value of the register R_state is updated—decremented by $N+1$—to prepare the buffer R for updating. When the pipeline MUL B $\rightarrow$ ADD/SUB A is freed, the values of $\boldsymbol{\xi}_{k-L}$ are read from the Block RAM UX and the multiplication $\xi_{0,k-L}\boldsymbol{\xi}_{k-L}$

is started. The results of this multiplication are successively subtracted from values $\mathbf{r}_{k-1} + \xi_{0,k}\boldsymbol{\xi}_k$ and propagated through the feedback path implemented using the ADD/SUB pipe and the `FIFO`. The results of subtraction, forming the vector $\mathbf{r}_k$, are written simultaneously to both Block RAM ports of `R` to positions representing the first row/column of the matrix $\mathbb{R}_k$.

The "`R` update" operation is fully pipelined without any interim pipeline stall, with the result that the ADD/SUB `A` is fully utilized. Due to the use of ADD/SUB pipe as an intermediate storage for partial results, no additional storage (except the `FIFO`) is needed. These factors in particular make the "`R` update" module very efficient. The length of `FIFO` is unambiguously determined by dimensions of the matrix $\mathbb{R}$, that is, by the projection order N. In order to use the feedback path as a temporary storage for the full set of intermediate results $\mathbf{r}_{k-1} + \xi_{0,k}\boldsymbol{\xi}_k$, the length of ADD/SUB pipe plus length of `FIFO` must correspond to N. Then, the length of `FIFO` can be expressed as $N_{fifo} = N - N_A$, where N_A is the length of the ADD/SUB pipeline.

Considering the latencies of the LNS adder and multiplier, this operation takes only 30 clock cycles for a matrix of order $N = 9$. It is apparent that our solution is dramatically faster than the simple approach according to (11), which involves $N(N+1)$ multiply-accumulate operations. In addition there are also savings in the number of memory accesses.

5.3 GS Solver: Update of $\mathbf{p}_k$

One of the key modules of the algorithm is the Gauss-Seidel (GS) solver, which is used to calculate the vector $\mathbf{p}_k$ (Step 2 of the algorithm in Figure 1). Unfortunately, the GS procedure is actually a sequential algorithm. Each element $p_{i,k}$ of the vector $\mathbf{p}_k$ depends on previously computed elements, $p_{0\ldots i-1,k}$ and $p_{i+1\ldots N-1,k-1}$. Hence, the individual elements of $\mathbf{p}_k$ cannot be updated simultaneously, so parallelization is not feasible. Instead we use a pipelined architecture, in which we try to minimize the latency of one step of the algorithm, that is, the latency of the $p_{i,k}$ calculation. The hardware architecture for the calculation of one step, that is, of one element of vector $\mathbf{p}_k$, is depicted in Figure 9(b). This operation has to be performed N times to update the whole vector $\mathbf{p}$. The computation of the next element can start after the previous value has been written to the Block RAM referred to as `PEPS`. We use this Block RAM to store both vector $\mathbf{p}$ and ϵ.

In order to minimize the cycle count, we use both ports of Block RAMs `R` and `PEPS` and two MUL (`A` and `B`) units in parallel. The multiplication results are summed and the result of summation forms the dot product of two $N-1$ length vectors. Now, recall that the vector $\mathbf{b}$ has the value 1 in its first element, and all other elements have the value 0. Because of this, we need to subtract the value from 1 only in the first step, but in all other steps we are subtracting from zero, which we can be implemented by simply negating the sign bit—this manipulation is depicted in Figure 9(b) as the "branch path" consisting of subtracting the `SUM` result from `'1'`, the component `NegSGN` (sign negation) and the multiplexer. Considering the ADD/SUB unit latency and that the negation of the sign bit costs virtually nothing, we can save $9(N-1)$ clock cycles for

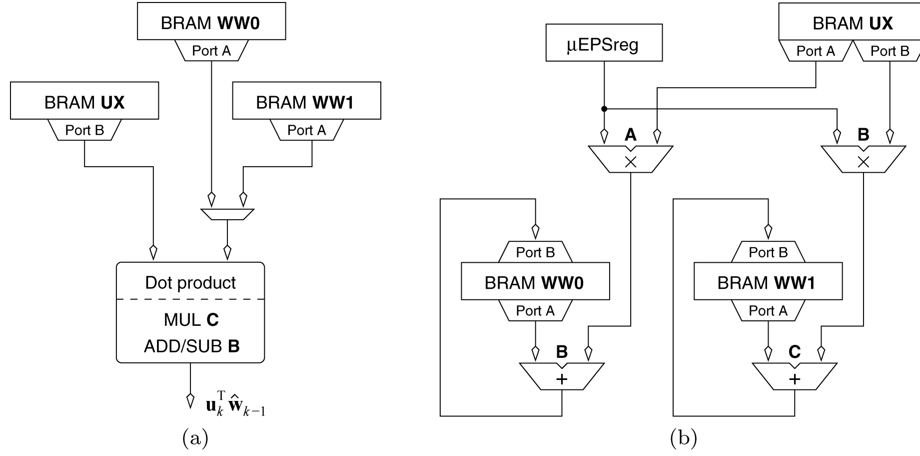

Fig. 10. (a) Organization of the Block RAMs UX and WW[0,1] connections to the dot-product unit, which calculates $\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$; (b) Architecture schematic of the update of alternate coefficient vector $\hat{\mathbf{w}}_k$ stored in Block RAMs WW[0,1].

a complete "P update." The result of this operation (subtraction or negation) is then divided by a corresponding diagonal element of correlation matrix $R_{ii,k}$ and the result $p_{i,k}$—the i-th element of vector $\mathbf{p}_k$—is finally written to the Block RAM PEPS. It is also important to recall that although the GS procedure is a naturally sequential algorithm, the LNS multiplication and division are fast and cheap, so the resulting hardware is highly efficient.

5.4 Filter Output and Estimation Error

The next step of the algorithm is to compute the filter output y_k and the estimation error e_k as shown in Step 3 in Figure 1. The right-hand term of Step 3a—dot product $\bar{\epsilon}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$—is computed just after the vector $\mathbf{p}_k$ has been updated. In Section 5.3, we mentioned that vectors $\mathbf{p}_k$ and ϵ_k are both stored in the single Block RAM denoted PEPS. We do this partly because these vectors are of length N and they can easily fit into a single Block RAM.[2] But this arrangement also allows us to use the pipeline R, PEPS → MUL A, B → SUM (used in the update of $\mathbf{p}_k$—see Figure 9(b)) for calculation of the dot product $\bar{\epsilon}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$, without creating any extra hardware. Both dot-product operations are performed on vectors of length $N-1$, so the only necessary modification is to change the addressing of Block RAMs R and PEPS. The resulting value is then multiplied, using the MUL D unit, by a value of μ in order to get the result $\mu\bar{\epsilon}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$.

As depicted in Figure 5, the "long" dot product—of two vectors of length given by the filter order L—$\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$ is calculated in parallel with previously described blocks: "R update," "P update," and the dot product $\mu\bar{\epsilon}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$. The organization

[2]Considering that N is usually less than 20, all three variables $\mathbb{R}_k$, $\mathbf{p}_k$ and ϵ_k could easily fit into a single Block RAM, but this arrangement would be inconvenient with respect to insufficient number of memory access ports when updating $\mathbf{p}_k$ and for calculation of $\bar{\epsilon}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$. This is clearly demonstrated in Figure 9(b).

of Block RAMs connected to the dot-product unit, which employs the MUL C unit and the ADD/SUB B unit, is depicted in Figure 10(a).

Results of two dot-product operations, $\mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$ and $\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$, discussed in previous paragraphs are used to calculate the filter output y_k and the estimation error e_k, but to save a few more clock cycles they are processed in the following way

$$e_k = \left(d_k - \mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}\right) - \mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$$
$$y_k = \mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1} + \mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$$

Two subtractions, $d_k - \mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$ and $(\cdot) - \mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$, are performed using the ADD/SUB A unit, whereas the addition $\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1} + \mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$ is performed on the ADD/SUB B unit. This order of operations causes that the filter output y_k and the estimation error e_k are available at the same clock cycle, which is schematically depicted in Figure 5.

5.5 Alternate Coefficient Vector Update

The last two modules of the GSFAP unit are the "EPS update," which performs the update of normalized estimation error vector $\boldsymbol{\epsilon}_k$, and the "WW[0,1] update," which performs the update of alternate coefficient vector $\hat{\mathbf{w}}_k$. The two modules have a similar structure, with the former simpler than the latter (see Step 4 of the algorithm in Figure 1).

The "EPS update" module performs a simple pipelined multiply-add operation. It reads operands from the Block RAM PEPS. Since both vector $\mathbf{p}_k$ and ϵ_{k-1} are stored in this Block RAM, data are read from both ports. After all operands have been retrieved from the Block RAM PEPS, the updated elements of vector $\boldsymbol{\epsilon}_k$ are successively stored to the Block RAM using its port A. Before the port A of PEPS is freed, the ADD/SUB pipeline and an auxiliary FIFO of length $N - N_A$ are used as a buffer for results to be stored into the Block RAM.

After the vector $\boldsymbol{\epsilon}_k$ has been updated, its last element $\epsilon_{N-1,k}$ is multiplied by μ using the MUL D unit. The result of multiplication is stored in the register μEPSreg, the contents of which is then used for the update of $\hat{\mathbf{w}}_k$.

The last task to complete one iteration of GSFAP is to update the alternate coefficient vector $\hat{\mathbf{w}}_k$, which is performed by the module "WW[0,1] update." Its hardware architecture is schematically shown in Figure 10(b). We decided to split the coefficient vector $\mathbf{w}_k$ into two parts, which are stored in separate Block RAMs denoted WW0 and WW1. The reason is that the data bandwidth to Block RAMs is limited by the number of ports. Splitting the vector $\mathbf{w}$ allows us to use two independent pipelines to update both halves of $\mathbf{w}$ in parallel as depicted in the figure, in order to further reduce execution time, that is, the number of clock cycles, of GSFAP iteration. This stage fully utilizes both pipelines of the ADD/SUB unit and both MUL A and B units.

6. EXPERIMENTAL RESULTS

6.1 Convergence Properties and Stability

In order to investigate the behavior and performance of adaptive filtering algorithms, we implemented several algorithms in a simulation environment. These were LMS, NLMS, RLS, APA, MFAP, CGFAP, and GSFAP. The algorithms were implemented using IEEE 64-bit (double) and 32-bit (single) and LNS 32-bit and 19-bit arithmetics.

In our simulations, the unknown system is modelled by a finite impulse response (FIR) filter of order L and its transfer function is given by the filter coefficients. We refer to these coefficients as the optimal weights or the optimal weight vector, $\mathbf{w}_{opt}$. In most experiments, the individual weights were chosen as random values. The weight vector was, however, normalized so that $\|\mathbf{w}_{opt}\| = 1$, to avoid amplification or attenuation of the input signal. In the rest of the experiments, an impulse response measured in an acoustic studio was used as a model.

We decided to test the algorithms with three different filter lengths, with 50, 250, and 1000 filter coefficients. This provided a realistic simulation environment for applications where a very high order adaptive filter is required, particularly for acoustic applications like adaptive noise cancellation.

Two different measures to compare the convergence of individual algorithms were used, the system error norm and the mean squared error. The system error norm (SEN) can be expressed as

$$SEN_k = 10 \log_{10}(\|\mathbf{w}_k - \mathbf{w}_{opt}\|^2) \quad [\text{dB}], \tag{14}$$

which is the squared norm of the difference between the adaptive filter weights, $\mathbf{w}_k$, and the unknown system model coefficients represented by $\mathbf{w}_{opt}$. The other measure is the mean squared error (MSE) and can be calculated as

$$MSE_k = \frac{1}{K} \sum_{i=0}^{K-1} e_{i,k}^2, \tag{15}$$

which is the average of the square of estimation error over K independent runs of an experiment. The indices i and k represent the index number of experiment and the iterations, respectively. The error $e_{i,k}$ is then the estimation error in the k-th iteration of the i-th experiment. It can also be expressed in decibels (dB), that is, as $10 \log_{10}(MSE_k)$.

To simulate a more realistic environment, additive white Gaussian noise was added to the system output. The white noise, a zero-mean Gaussian process, with the variance $\sigma_n^2 = 10^{-4}$, was used. This noise then determines the theoretical value of the minimum MSE that can be reached if the coefficient vector of the adaptive filter, $\mathbf{w}_k$, is identical to its optimal value, that is to the coefficient vector of the unknown system, $\mathbf{w}_{opt}$.

For the filter input, we use two types of signal: white and colored noise. The white noise used is zero-mean Gaussian process with the variance $\sigma_u^2 =$ 1. To generate the colored noise, we use first-order Markov process which is generated by applying white Gaussian noise, with variance σ_{inp}^2, to a first-order

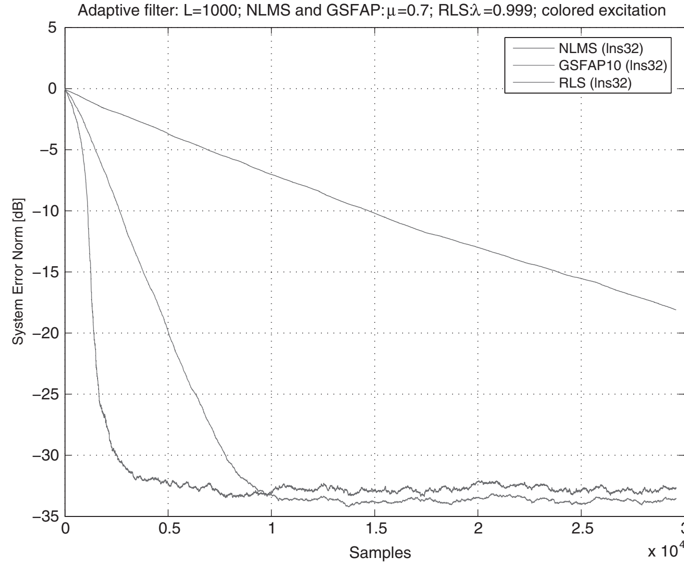

Fig. 11. Convergence rates of the NLMS, GSFAP, and RLS algorithms using the 32-bit LNS arithmetic.

auto-regressive filter with transfer function

$$H(z) = \frac{1}{1 - az^{-1}}, \tag{16}$$

where a is a fixed parameter. The colored noise, that is, the adaptive filter input, is chosen to have the variance $\sigma_u^2 = 1$. In order to generate the output of an auto-regressive filter with this variance, the variance of the input white Gaussian noise is given as $\sigma_{inp}^2 = 1 - a^2$.

In Figure 11, we present the convergence rate of three algorithms: NLMS, GSFAP with the projection order $N = 10$ denoted as GSFAP10, and RLS with parameters depicted in the title of the figure. In this experiment we used highly colored noise (with $a = 0.9$) as the excitation signal u. It can clearly be seen that even if the GSFAP algorithm is not as good as the RLS algorithm, its convergence rate is far superior to the convergence rate of NLMS with the corresponding algorithm parameters. In addition the complexity of GSFAP is significantly lower compared to RLS.

Besides the SEN and MSE, we also measured the signal-to-noise (SNR) ratios for the 32-bit floating-point and the 32-bit LNS implementations of these algorithms (using various parameters), in order to compare the numerical stability. As mentioned in Section 4.4.3, we obtained comparable results for both FLP and LNS implementations. We also found that all 32-bit implementations were numerically stable for both highly colored and nonstationary input signals. For example, the GSFAP implementation using FLP showed a SNR of 108.4 dB, whereas the corresponding LNS figure was 108.6 dB for the error signal. There were two deviations from this rule.

(1) The level of precision of GSFAP implemented using the 19-bit LNS arithmetic was not sufficient for some filtering applications. In particular, we found that the algorithm became numerically unstable for some nonstationary excitation signals such as speech. However, the precision was found to be sufficient for most other filtering applications, for example canceling random noise in speech.
(2) The results of experiments with the RLS algorithm when excited with both white and colored noise showed that for the 32-bit FLP implementation the RLS algorithm becomes unstable very quickly—after about 3500 and 3000 input samples for white and colored noise excitation, respectively. To the contrary, the RLS run under exactly the same conditions but implemented using the 32-bit LNS with the same precision remains stable.

To conclude this discussion, we can state that both 32-bit floating-point and 32-bit LNS implementations of GSFAP algorithm are numerically stable and show very similar results. Another conclusion is that implementation using the 32-bit LNS compared to 32-bit FLP arithmetic can give considerable advantage under some specific conditions, while further reducing of precision (saving hardware resources) by using the 19-bit LNS arithmetic can lead in stability problems in applications where non-stationary signals are involved.

6.2 Hardware Implementation

We developed separate GSFAP cores for the LNS 32-bit and 19-bit precisions. The parameters of both cores are also fully configurable. It is possible to vary filter order L for values $20 \leq L \leq 1000$, projection order N for values $2 \leq L \leq 20$, step-size parameter μ for values $0 < \mu < 2$ and regularization parameter δ for values $10^{-3} \leq \delta \leq 1$. However, modifying the value of N requires minor architectural changes.

For presenting cores in this section, we fixed the length of the filter as $L = 1000$ and the projection order as $N = 9$. In this configuration, a full iteration of the GSFAP algorithm takes 1597 cycles and performs 4227 log arithmic operations, which represents 2.64 operations per cycle.

Table III shows area and performance parameters for the 32-bit and 19-bit LNS implementations of GSFAP algorithm on the two FPGA Xilinx devices: one-million gate Virtex-II XC2V1000-4 and six-million gate Virtex-II XC2V6000-4. On both devices, the designs can be clocked at a little over 80MHz. This is very close to the maximum clock rate of LNS cores, indicating that our architecture is not the limiting factor on clock speed. At this clock speed the design is performing over 210 million log arithmic operations per second, which is equivalent to 210MFLOPS. The cores can filter signals at a sampling rate of more than 50kHz.

The 32-bit LNS version of GSFAP core occupies only a small fraction of the six-million gate Xilinx XC2V6000-4. On the smaller XC2V1000-4 device, it uses a very large percentage of available resources. In particular, 99% of slices on the FPGA contain some logic. However, our design demonstrates that FPGAs are very suitable for implementing complex adaptive filters. It is possible to

Table III. Resource Utilization and Parameters of the 32-bit and 19-bit LNS GSFAP Units When Used in Xilinx Virtex-II XC2V1000-4 and XC2V6000-4 FPGAs

	LNS 32-bit				LNS 19-bit			
	XC2V1000-4		XC2V6000-4		XC2V1000-4		XC2V6000-4	
Slice Flip Flops	4,835	47%	4,496	6%	3,414	33%	3,234	4%
4 input LUTs	6,049	59%	6,058	8%	4,245	41%	4,294	6%
Occupied Slices	5,118	99%	4,833	14%	3,538	69%	3,353	9%
Tbufs	1,280	50%	1,280	7%	192	7%	192	1%
Block RAMs	34	85%	34	23%	12	30%	12	8%
MULT18X18s	8	20%	8	5%	8	20%	8	5%
Clock rate	80.006MHz		80.051MHz		80.502MHz		80.160MHz	
XPower estimate								
Vccint Dynamic	401 mW		455 mW		188 mW		219 mW	
Quiescent	18 mW		68 mW		18 mW		68 mW	
Vccaux Dynamic	0 mW		0 mW		0 mW		0 mW	
Quiescent	330 mW		330 mW		330 mW		330 mW	
Vcco33 Dynamic	0 mW		0 mW		0 mW		0 mW	
Quiescent	3 mW		7 mW		3 mW		7 mW	
Total Power	752 mW		860 mW		539 mW		624 mW	

implement such adaptive filters with 32-bit precision arithmetic on small FPGAs, suitable for resource-constrained embedded systems.

The 19-bit LNS version of the GSFAP core can operate at a clock speed similar to the 32-bit version, but resource requirements are substantially lower. In particular, on the XC2V1000-4 only 30% of Block RAMs and 69% of slices are used, compared to 85% and 99% for the 32-bit version, respectively. There is clearly potential for either implementing other logic on the same chip using free resources, or placing the 19-bit GSFAP core on a smaller device.

For comparison, we created similar cores that implement the NLMS algorithm in the LNS 32-bit and 19-bit precisions. The parameters of all cores are also fully configurable: $20 \leq L \leq 1022$; $0 < \mu < 2$; and $0 \leq \delta \leq 1$. The corresponding NLMS filter of order $L = 1000$ is used to compare performance. In this configuration, a full iteration of the NLMS algorithm takes 1088 clock cycles, and performs 4008 log arithmic operations, which represents 3.68 operations per cycle.

Table IV shows the corresponding figures for our NLMS cores. The clock rates are again around 80MHz. The design performs 295 million log arithmic operations per second (equivalent to MFLOPS), and can operate on signals at a sampling rate of more than 73kHz.

The 32-bit LNS NLMS core occupies only a small fraction (12%) of the six-million gate XC2V6000-4 device. On the one-million gate XC2V1000-4 FPGA, it uses quite a large percentage of available chip resources; in particular, 87% of slices and 80% of Block RAMs. For the 19-bit LNS implementation, the figures show that the core occupies a little over a half of the available slices and 25% of Block RAMs on the one-million gate chip. Although the NLMS cores are smaller than the GSFAP cores, the NLMS ones allow higher sampling rates.

The FPGAs we used to test the implementations are speed-grade four devices. More expensive speed-grade six devices would allow even faster clock

Table IV. Resource Utilization and Parameters of the 32-bit and 19-bit LNS NLMS Units When Used in Xilinx Virtex-II XC2V1000-4 and XC2V6000-4 FPGAs

	LNS 32-bit				LNS 19-bit			
	XC2V1000-4		XC2V6000-4		XC2V1000-4		XC2V6000-4	
Slice Flip Flops	4,408	43%	4,069	6%	3,026	29%	2,846	4%
4 input LUTs	4,834	47%	4,831	7%	3,301	32%	3,369	4%
Occupied Slices	4,473	87%	4,160	12%	2,973	58%	2,820	8%
Tbufs	1,280	50%	1,280	7%	192	7%	192	1%
Block RAMs	32	80%	32	22%	10	25%	10	6%
MULT18X18s	8	20%	8	5%	8	20%	8	5%
Clock rate	80.051MHz		80.058MHz		80.652MHz		80.483MHz	
XPower estimate								
Vccint Dynamic	381 mW		434 mW		161 mW		189 mW	
Quiescent	18 mW		68 mW		18 mW		68 mW	
Vccaux Dynamic	0 mW		0 mW		0 mW		0 mW	
Quiescent	330 mW		330 mW		330 mW		330 mW	
Vcco33 Dynamic	0 mW		0 mW		0 mW		0 mW	
Quiescent	3 mW		7 mW		3 mW		7 mW	
Total Power	732 mW		839 mW		512 mW		594 mW	

speeds, in the region of 100MHz. Virtex-4 devices would be even faster. However, our cores are designed for resource-constrained embedded systems, where cost is an important factor, and they achieve more than adequate results on these slower devices. But even on our slower devices, we tested the cores at 100MHz and found them to be fully functional at room temperatures. For non-safety-critical applications, it may be possible to run the cores above the clock speeds reported by the design tools.

6.3 Practical Applications of GSFAP Core

To demonstrate performance of the GSFAP algorithm in practical application, we developed the adaptive *noise cancellation* example. In this case, the adaptive filter is used to cancel an unknown interference contained in a primary signal. The primary signal serves as the desired response for the adaptive filter. A reference signal, which is correlated with the interference, is used as the input to the adaptive filter.

For our demonstration we used the LNS 32-bit implementation of a digital adaptive filter of length $L = 250$ based on the GSFAP algorithm with the projection order $N = 9$ and the step size parameter $\mu = 0.7$. The result of the noise cancellation process is depicted in Figure 12. The figure shows the clear (original) signal, the corrupted signal, and the filtered signal, which consists of the original signal and an uncancelled (residual) noise. The impact of filtering process can be easily recognized from the figure. The best demonstration of results is, however, subjective listening which unfortunately cannot be presented in the text.

To demonstrate another practical application of adaptive filters, we developed an *echo cancellation* example. In this case the adaptive filter is used to suppress echo generated by an unknown system, typically a room or a car cabin. The real-world room impulse responses and speech signals have been used in our

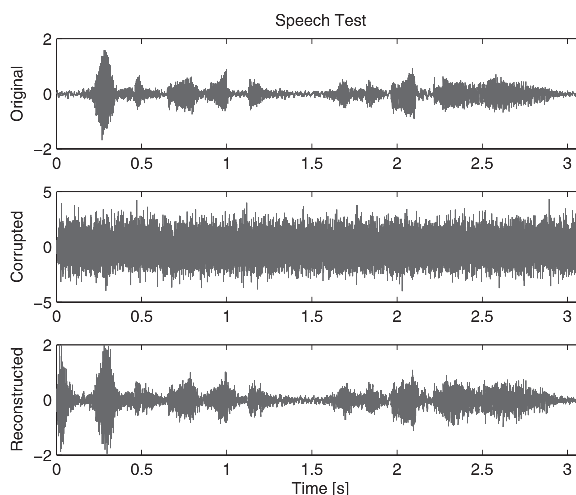

Fig. 12. Canceling random noise in a speech signal using the 32-bit LNS GSFAP core.

experiments. We used the input and echo signals sampled at the rate of 16kHz, the adaptive filters of length $L = 500$ and the GSFAP projection order $N = 9$. The step-size parameter was chosen as $\mu = 1$ for both NLMS and GSFAP.

The results of using the LNS 32-bit implementation of NLMS and GSFAP algorithms for this task are shown in Figure 13. The upper-left picture of the figure shows the input (far-end) `U` and echo `D` signals. The upper-right picture shows the echo signal to be suppressed and the convergence rates of NLMS and GSFAP adaptive filter. It can clearly be seen that the GSFAP converges much faster than the NLMS algorithm. The lower-left picture represents residual echo for both algorithms. It should be noted that the variance (var) of residual echo `E`—which can also be used as a measure of quality of the adaptive algorithm—“left” by the NLMS adaptive filter is 5.79×10^{-4} while for GSFAP it is 6.32×10^{-5}. The lower-right picture shows the squared value of `E`.

7. RELATED WORK

Chew and Farhang-Boroujeny [1999] present the fixed-point implementation of the adaptive filter using the LMS-Newton algorithm and use it for acoustic echo cancellation. They report the sampling rate 29.4kHz for the 578-tap adaptive filter on a Xilinx XC4042XL chip.

Jang et al. [2002] describe FPGA implementation of the NLMS using fixed-point arithmetic within the context of acoustic echo cancellation. They use FPGA only as a prototyping platform for implementation in ASIC and report a sampling rate of 8kHz for the 256-tap filter on an Altera FLEX 10K50RC240 FPGA.

The GSFAP algorithm was developed by Albu et al. [2002a] to reduce the time complexity of FAP based adaptive algorithms. The original description of the algorithm contained a proposal that the algorithm be implemented on a general purpose processor using LNS arithmetic. Simulations suggested that a sampling rate of 16kHz could be achieved on a 200MHz processor.

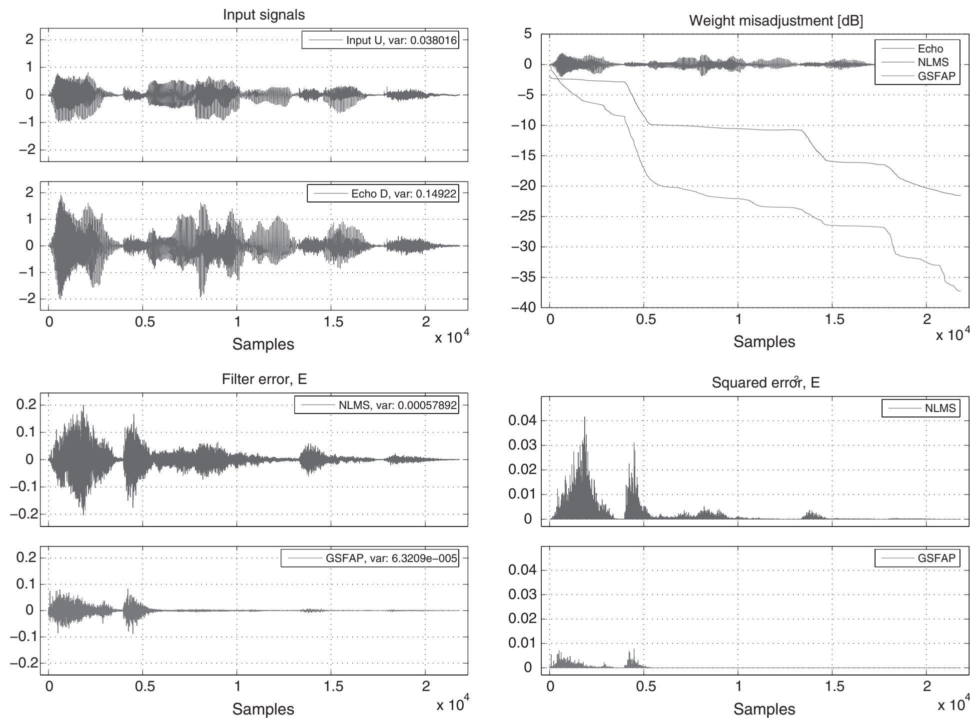

Fig. 13. Comparison of the NLMS and GSFAP within the echo cancellation application using speech signals and real impulse response.

An RLS lattice core based on the LNS architecture for embedded systems is described by Kadlec et al. [2002]. Significant speed-ups using single-, dual-, and quad-pipeline LNS architectures are reported. Lattice core designs achieve clock speeds of 35 to 50MHz and the peak performance 168MFLOPS for the 20-bit LNS implementation on a Virtex XCV2000E-6 FPGA device.

Similar work was published in Albu et al. [2001], where an LNS implementation of the normalized RLS lattice algorithm in a Virtex FPGA is presented. They implemented the 8-th order filter capable of processing signals at a sampling rate of 47kHz for standard and 36.7kHz for a normalized version of RLS lattice. Their designs can be clocked at 45MHz and the latency of one iteration of the algorithm is 950 and 1224 clock cycles for standard and normalized RLS lattice, respectively.

The same group continued their research on implementing RLS using FPGAs. They developed a core based on the modified *a priori* error-feedback least square lattice (EF-LSL) algorithm, which was described in Albu et al. [2002b]. The authors report a maximum performance of 8kHz when using a 175-tap filter implemented with 32-bit LNS on a Virtex device.

Pohl et al. [2003] implemented a similar RLS lattice core using LNS arithmetic on a Virtex-II XC2V6000-6 (six-million gate, speed grade 6) device. The design achieves sampling rates of 31.6kHz, 15.8kHz, and 7.9kHz for filters of order 84, 172, and 252, respectively. In contrast, our GSFAP cores achieve a

sampling rate of more than 50 kHz for filters of order 1000. Although the RLS algorithm may converge more quickly than GSFAP, it is at the cost of a greatly reduced sampling rate.

Sucha et al. [2004] present an implementation of a 129-tap QR-RLS-based adaptive filter capable of operating on signals sampled at rate of 44kHz. This is a significant achievement on a resource-constrained Xilinx XC2V1000-4 device. They use the 19-bit (rather than 32-bit) versions of the LNS arithmetic units that we use in this article, in order to reduce resource usage.

Boppana et al. [2004] also implement a QR-RLS adaptive filter. They use CORDIC operators and implement the algorithm using both Altera's Nios and custom logic. They present filter orders 4 to 32, where for a 32-tap filter they need > 100000 (120μs) or > 1000 (3μs) clock cycles when Nios or custom logic is used, respectively.

Most recently, the dichotomous coordinate descent (DCD) FAP [Zakharov and Albu 2005] algorithm has been proposed. In this algorithm, the DCD method is employed to solve the problem of matrix inversion. The DCD algorithm [Zakharov et al. 2004; Liu et al. 2006] is a numerically stable algorithm for solving systems of linear equations with relatively low computational complexity (lower than the Gauss-Seidel algorithm). Since the DCD algorithm is free of multiplication and division, the DCD-FAP algorithm could also be a convenient candidate for FPGA implementation, but we leave it for future work.

8. CONCLUSIONS

Adaptive filters are widely used in digital signal processing (DSP) for countless applications in telecommunications, digital broadcasting, etc. Traditionally, small resource-constrained embedded systems have used the least computationally intensive filter adaptive algorithms, such as NLMS. In this article we have shown that FPGAs are a highly suitable platform for more complex algorithms with better adaptation properties.

We have designed a core that implements the GSFAP adaptive filtering algorithm. GSFAP combines relatively low computational complexity with excellent adaptation properties. In order to reduce resource requirements, we have used log arithmic arithmetic, rather than traditional floating point.

Although it is efficient, GSFAP is a complicated algorithm which presented several implementation challenges. Efficient design of the data structures using our reindexing mechanism allowed matrices to be shifted diagonally without copying. The GS stage is particularly difficult to parallelize efficiently because of cyclic data dependencies between iterations of the inner loop. We have presented efficient solutions to these problems, making the best use of the pipelined log arithmic addition units, and taking advantage of the very low cost of log arithmic multiplication and division.

The resulting GSFAP core can be clocked at more than 80MHz on a one million-gate Xilinx XC2V1000-4 device. At this clock speed the design performs over 210 million log arithmic operations per second (equivalent to MFLOPS). We used it to implement adaptive filters of orders 20 to 1000 performing echo cancellation on speech signals at a sampling rate exceeding 50kHz. A similar

NLMS core is around 15% smaller and around 50% faster, allowing it to filter signals at a sampling rate of around 73kHz. However, experiments show that GSFAP has adaptation properties that are much superior to NLMS, and that our core can provide very sophisticated adaptive filtering capabilities for resource-constrained embedded systems.

REFERENCES

ALBU, F., FAGAN, A., KADLEC, J., AND COLEMAN, N. 2002a. The Gauss-Seidel fast affine projection algorithm. In *Proceedings of the Workshop on Signal Processing Systems (SIPS'02)*. IEEE, 109–114.

ALBU, F., KADLEC, J., COLEMAN, N., AND FAGAN, A. 2002b. Pipelined implementations of the a priori error-feedback LSL algorithm using logarithmic arithmetic. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'02)*. IEEE, III–2681–III–2684.

ALBU, F., KADLEC, J., SOFTLEY, C., MATOUSEK, R., HERMANEK, A., FAGAN, A., AND COLEMAN, N. 2001. Implementation of (normalized) RLS lattice on Virtex. In *Field-Programmable Logic and Applications*. G. J. Brebner and R. Woods, Eds. Lecture Notes in Computer Science. vol. 2147, Springer-Verlag, Berlin.

BOPPANA, D., DHANOA, K., AND KEMPA, J. 2004. FPGA based embedded processing architecture for the QRD-RLS algorithm. In *Proceedings of the 12th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*. IEEE, 330–331.

CHEW, W. C. AND FARHANG-BOROUJENY, B. 1999. FPGA implementation of acoustic echo cancelling. In *Proceedings of the Region 10 Conference TENCON99*. Vol. 1. IEEE, 263–266.

CIOFFI, J. AND KAILATH, T. 1983. Fast, fixed-order, least-squares algorithms for adaptive filtering. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, (ICASSP'83)*. Vol. 8. IEEE, 679–682.

COLEMAN, J. N. 1995. Simplification of table structure in logarithmic arithmetic. *Electro. Lett. 31,* 22, 1905–1906.

COLEMAN, J. N. AND CHESTER, E. I. 1999. A 32-bit logarithmic arithmetic unit and its performance compared to floating-point. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*. IEEE, 142–151.

COLEMAN, J. N., CHESTER, E. I., SOFTLEY, C. I., AND KADLEC, J. 2000. Arithmetic on the European Logarithmic Microprocessor. *IEEE Trans. Comput. 49,* 7, 702–715.

COLEMAN, J. N., SOFTLEY, C. I., KADLEC, J., MATOUSEK, R., TICHY, M., POHL, Z., HERMANEK, A., AND BENSCHOP, N. F. 2008. The European logarithmic microprocesor. *IEEE Trans. Comput. 57*, 4, 532.

DING, H. 2000. A stable fast affine projection adaptation algorithm suitable for low-cost processors. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'00)*. Vol. 1. IEEE, 360–363.

GAY, S. L. 1993. A fast converging, low complexity adaptive filtering algorithm. In *Proceedings of the Workshop on Applications of Signal Processing to Audio and Acoustics*. IEEE, 4–7.

GAY, S. L. AND TAVATHIA, S. 1995. The fast affine projection algorithm. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'95)*. Vol. 5. IEEE, 3023–3026.

HAGEMAN, L. A. AND YOUNG, D. M. 1981. *Applied Iterative Methods*. Academic Press, New York.

HASELMAN, M., BEAUCHAMP, M., WOOD, A., HAUCK, S., UNDERWOOD, K. D., AND HEMMERT, K. S. 2005. A comparison of floating point and logarithmic number systems for FPGAs. In *Proceedings of the 13th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE, 181–190.

HAYKIN, S. 2002. *Adaptive Filter Theory* 4th Ed. Prentice Hall, Upper Saddle River, NJ.

Institute of Electrical and Electronics Engineers, Inc. 1985. *An American National Standard: IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, Inc., New York. ANSI/IEEE Std 754-1985.

JANG, S. A., LEE, Y. J., AND MOON, D. T. 2002. Design and implementation of an acoustic echo canceller. In *Proceedings of the Asia-Pacific Conference on ASIC*. IEEE, 299–302.

KADLEC, J., MATOUSEK, R., HERMANEK, A., LICKO, M., AND TICHY, M. 2002. Lattice for FPGAs using logarithmic arithmetic. *Electro. Engin. Des. 74,* 906, 53–56.

KALOUPTSIDIS, N. AND THEODORIDIS, S. 1993. *Adaptative System Identification and Signal Processing Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

KANEDA, Y., TANAKA, M., AND KOJIMA, J. 1995. An adaptive algorithm with fast convergence for multi-point sound control. In *Proceedings of the Active'95*. 993–1004.

KIKKERT, C. 1974. Digital companding techniques. *IEEE Trans. Comm. 22,* 1, 75–78.

KOREN, I. 2002. *Computer Arithmetic Algorithms* 2nd Ed. A. K. Peters, Ltd., Natick, MA.

LIU, J., WEAVER, B., AND WHITE, G. 2006. FPGA implementation of the DCD algorithm. In *Proceedings of the London Communications Symposium*. University College London, London, UK.

LIU, Q. G., CHAMPAGNE, B., AND HO, K. C. 1996. On the use of a modified fast affine projection algorithm in subbands for acoustic echo cancelation. In *Proceedings of the 7th Digital Signal Processing Workshop*. IEEE, 354–357.

LUENBERGER, D. G. 1984. *Linear and Nonlinear Programming*, 2nd ed. Addison-Wesley, Reading,. MA.

MATOUSEK, R., TICHY, M., POHL, Z., KADLEC, J., SOFTLEY, C., AND COLEMAN, N. 2002. Logarithmic number system and floating-point arithmetics on FPGA. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, M. Glesner, P. Zipf, and M. Renovell, Ed. Lecture Notes in Computer Science. vol. 2438. Springer-Verlag, Berlin.

OZEKI, K. AND UMEDA, T. 1984. An adaptive filtering algorithm using an orthogonal projection to an affine subspace and its properties. *Electron. Comm. Japan 67-A,* 5, 126–132.

POHL, Z., MATOUSEK, R., KADLEC, J., TICHY, M., AND LICKO, M. 2003. Lattice adaptive filter implementation for FPGA. In *Proceedings of the 11th Iternational Symposium on Field Programming Gate Arrays (FPGA'03)*. ACM/SIGDA, 246. Abstract.

SLOCK, D. T. M. AND KAILATH, T. 1991. Numerically stable fast transversal filters for recursive least squares adaptive filtering. *IEEE Trans. Sign. Proces. 39,* 1, 92–114.

SUCHA, P., POHL, Z., AND HANZALEK, Z. 2004. Scheduling of iterative algorithms on FPGA with pipelined arithmetic unit. In *Proceedings of the 10th Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*. IEEE, 404–412.

SWARTZLANDER, E. E. AND ALEXOPOULOS, A. G. 1975. The sign/logarithm number system. *IEEE Trans. Comput. C-24,* 12, 1238–1242.

TICHY, M. 2006. Fast adaptive filtering algorithms and their implementation using reconfigurable hardware and log arithmetic. Ph.D. thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic.

TSIVIDIS, Y. P., GOPINATHAN, V., AND TOTH, L. 1990. Companding in signal processing. *Electron. Lett. 26*, 1331–1332.

UNDERWOOD, K. D. 2004. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the 12th International Symposium on Field Programming Gate Arrays (FPGA'04)*. ACM/SIGDA, 171–179.

WHITEHOUSE, H. J. 2006. Implicit sampling analog-to-digital converter. In *Proceedings of the 4th Digital Signal Processing Workshop*. IEEE, 19–22.

WIDROW, B. AND STEARNS, S. D. 1985. *Adaptive Signal Processing*. Prentice Hall, Englewood Cliffs, NJ.

Xilinx, Inc. 2005. *Virtex-II Platform FPGAs: Complete Data Sheet*, v3.4 Ed. Xilinx, Inc. Product Specification.

YU, L. K. AND LEWIS, D. M. 1991. A 30-b integrated logarithmic number system processor. *IEEE J. Solid-State Circu. 26,* 10, 1433–1440.

ZAKHAROV, Y. AND ALBU, F. 2005. Coordinate descent iterations in fast affine projection algorithm. *IEEE Sign. Process. Lett. 12,* 5, 353–356.

ZAKHAROV, Y. V., WEAVER, B., AND TOZER, T. C. 2004. Novel signal processing technique for real-time solution of the least squares problem. In *Proceedings of the 2nd International Workshop on Signal Processing for Wireless Communications*. 155–159.

Received January 2007; revised July 2007, August 2007; accepted December 2007