ADAPTIVE NOTCH IIR FILTERS. AN IMPLEMENTATION ON MOTOROLA SC140

Silviu Ciochină, Cristina Ciochină, Andrei Roman
"Politehnica" University of Bucharest, Faculty of Electronics and Telecommunications, 1-3 Iuliu Maniu Boulevard, Bucharest, Romania
email: silviu@comm.pub.ro, cristinaciochina@xnet.ro, andreiroman@xnet.ro

Abstract: Notch filters represent a method of estimating and/or tracking sinusoidal frequencies immerged in background noise. We shall refer to adaptive notch IIR filters, which make use of the RLS method. This paper presents an implementation of adaptive notch IIR filters on MOTOROLA StarCore140 DSP. We have used a modified RLS algorithm, more suitable for DSP implementation. The paper presents theoretical aspects but also focuses on implementation issues and experimental results.

Keywords: adaptive filtering, notch IIR filters, RLS algorithm, MOTOROLA StarCore140, DSP implementation.

1. INTRODUCTION

Adaptive notch filters are designed for signal environments consisting of sinusoidal components of unknown frequency immerged in background noise. The objective is either to estimate/track frequencies of sinusoidal components in noise, to obtain an enhanced version of them, or to cancel a periodic interference.

Adaptive notch filters play an important role in telecommunications industry today, having various applications: adaptive cancellation of RF interferences in DS-SS, CDMA, TDMA and GSM systems; applications in QPSK communications and in BPSK demodulation; enhancements for blind equalization algorithms; DTMF detection; applications in speech processing; military applications (radar, sonar, satellites movement control); bio-medical applications; speed control in ultra-precision devices for semiconductor components manufacturing.

For updating the tap coefficients of the adaptive filter we used the RLS (Recursive Least Squares) algorithm. In this case, the RLS algorithm has the advantage of its fast convergence rate, but also several important drawbacks. The arithmetical complexity is high ($O(N^2)$), the dynamic range of the algorithm's variables is large and it also contains two division operations.

These issues raise problems from the point of view of a DSP implementation. In a fixed-point arithmetic context, this might lead to overflow or stalling phenomena.

The paper is structured as follows. Section 2 refers to the RLS algorithm for adaptive notch filtering. Section 3 shortly presents the SC140 architecture and focuses on implementation issues. Section 4 contains experimental results and section 5 is dedicated to conclusions and perspectives.

2. DIRECT FORM RLS ALGORITHM FOR NOTCH FILTERING

Let us consider a signal environment consisting of N sinusoidal components of unknown frequency immerged in background noise. The signal corrupted by noise has the form:

$$x(n) = u(n) + v(n) \tag{1}$$

where

$$u(n) = \sum_{k=1}^{N} A_k \cos(n\omega_k + \varphi_k) \tag{2}$$

$\{u(.)\}$ denotes the useful signal and $\{v(.)\}$ denotes the background noise. The notch filter rejects the sinusoidal components. Therefore, in different filtering configurations, it is possible either to estimate frequencies of sinusoidal signals in noise, to cancel them ("whiten" the noise) or to obtain an enhanced version of them ("line enhancement" structure).

2.1. Second Order Notch Filters

In order to identify or cancel a sinusoidal interference in a signal, a stop band filter with strong rejection of the desired frequency should be used. It would be difficult to obtain a sharp amplitude-frequency characteristic with a FIR filter (it would require a high order). The alternative consists in using an IIR filter.

For a direct form implementation, the transfer function of a second order notch filter (which cancels one frequency) is:

$$\hat{H}(z) = \frac{1 + az^{-1} + z^{-2}}{1 + a\rho z^{-1} + \rho^2 z^{-2}}, 0 < \rho < 1 \tag{3}$$

where the notch frequency is given by:

$$\omega_0 = \arccos(-a/2) \tag{4}$$

and ρ gives the rejection bandwidth.

A notch IIR filter has an amplitude-frequency characteristic that cancels a certain frequency (this frequency is called "the notch frequency") and is almost constant at any other frequency, as shown in **Figure 1**; the zero-pole plot is shown in **Figure 2.** The higher ρ is, the more selective the filter gets.

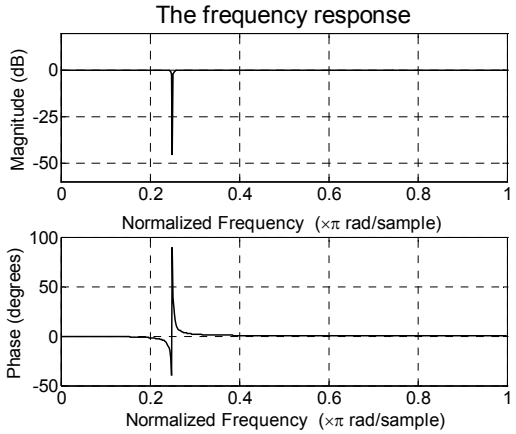

Figure 1 Amplitude-frequency and phase-frequency characteristics of a second order notch filter

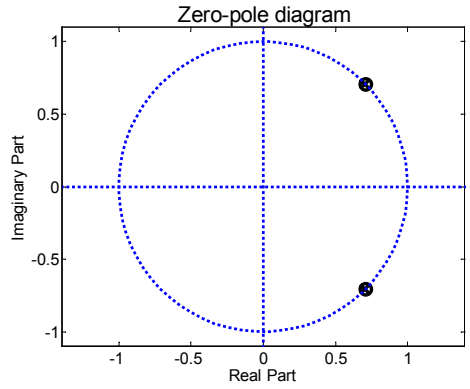

Figure 2 - Zero-pole plot for a second order notch filter

In the case of multiple frequency tracking, we have two options: one is to use one direct form filter of order 2N, the other one is to use N cascade cells of order 2. Cascade form proved to be more suitable for DSP implementation. The use of an adaptive filter is imposed by the fact that the sinusoid has an unknown frequency, which must be estimated.

2.2. RLS Adaptive Algorithm

Following the theoretical considerations in (Ciochină, Negrescu, 1999), we may state that the key problem for the RLS algorithm is minimizing the cost function

$$J(n)=\sum_{i=1}^{n}\lambda^{n-i}\left|e(i)\right|^2=\lambda J(n-1)+|e(n)|^2 \qquad (5)$$

where λ is a positive sub unitary weighting factor also called "forgetting factor" and $e(i)$ is the error signal at time i :

$$e(i)=d(i)-y(i) \qquad (6)$$

We denoted by $d(i)$ the desired response and by $y(i)$ the output produced by the filter. We shall further denote by $\mathbf{\Phi}(n),\boldsymbol{\theta}(n)$ and $\mathbf{w}(n)$ the correlation matrix, the cross-correlation vector and the tap-weight vector. Difference equations similar to (5) can be written:

$$\mathbf{\Phi}(n)=\lambda\mathbf{\Phi}(n-1)+\mathbf{x}(n)\mathbf{x}^H(n) \qquad (7)$$

$$\boldsymbol{\theta}(n)=\lambda\boldsymbol{\theta}(n-1)+\mathbf{x}(n)d^*(n) \qquad (8)$$

The optimum value for $\mathbf{w}(n)$ results from the normal equation:

$$\mathbf{\Phi}(n)\mathbf{w}(n)=\boldsymbol{\theta}(n) \qquad (9)$$

Denoting

$$\mathbf{P}(n)=\mathbf{\Phi}^{-1}(n) \qquad (10)$$

the Kalman vector may be defined as follows:

$$\mathbf{k}(n)=\frac{1}{\lambda}\frac{\mathbf{P}(n-1)\mathbf{x}(n)}{1+\frac{1}{\lambda}\mathbf{x}^H(n)\mathbf{P}(n-1)\mathbf{x}(n)} \qquad (11)$$

Performing the regular steps, it follows:

$$\mathbf{P}(n)=\frac{1}{\lambda}[\mathbf{P}(n-1)-\mathbf{k}(n)\mathbf{x}^H(n)\mathbf{P}(n-1)] \qquad (12)$$

$$\mathbf{w}(n)=\mathbf{w}(n-1)+\mathbf{k}(n)\alpha^*(n) \qquad (13)$$

$$\alpha(n)=d(n)-\mathbf{w}^H(n-1)\mathbf{x}(n) \qquad (14)$$

The initialization value for $\mathbf{P}(n)$ is usually chosen

$$\mathbf{P}(0)=\delta^{-1}\mathbf{I} \qquad (15)$$

where δ is a positive constant and $\mathbf{I}$ is the identity matrix. A λ close to unity assures a higher stability after the algorithm converges, but it leads to very small (close to zero) values of the elements in $\mathbf{P}(n)$. In fixed-point context, this might cause algorithm stalling.

2.3. A Modified RLS Version

In the previous discussion, $J(n)$ is the estimate of the expectation $E\{|e(n)|^2\}$ and $\mathbf{\Phi}(n)$ is the estimate of the correlation matrix of the input data, $\mathbf{R}$. Obviously, both estimates are biased. It has been proven (Ciochină, Paleologu., Enescu., 2003) that several modifications would improve the algorithm's behavior from the point of view of a DSP fixed-point implementation. With this goal, the cost function from equation (5) can be modified as follows:

$$\begin{aligned}J(n)&=(1-\lambda)\sum_{i=1}^{n}\lambda^{n-i}\,|e(i)|^2=\\&=\lambda J(n-1)+(1-\lambda)\,|e(n)|^2\end{aligned} \qquad (16)$$

In this case $J(n)$ is an asymptotically unbiased estimate of the mean square error:

$$E\{J(n)\}=(1-\lambda^n)E\{|e(n)|^2\}\xrightarrow[n\to\infty]{}E\{|e(n)|^2\} \qquad (17)$$

Similarly, equations (7) and (8) become:

$$\mathbf{\Phi}(n)=\lambda\mathbf{\Phi}(n-1)+(1-\lambda)\mathbf{x}(n)\mathbf{x}^H(n) \qquad (18)$$

$$\boldsymbol{\theta}(n)=\lambda\boldsymbol{\theta}(n-1)+(1-\lambda)\mathbf{x}(n)d^*(n) \qquad (19)$$

$\mathbf{\Phi}(n)$ is an asymptotically unbiased estimate of the correlation matrix of the input data, $\mathbf{R}$:

$$E\{\mathbf{\Phi}(n)\}=(1-\lambda^n)\mathbf{R}\xrightarrow[n\to\infty]{}\mathbf{R} \qquad (20)$$

Following the same computational steps, equation (11) can be rewritten:

$$\mathbf{k}(n)=\frac{\mathbf{P}(n-1)\mathbf{x}(n)}{\frac{\lambda}{1-\lambda}+\mathbf{x}^H(n)\mathbf{P}(n-1)\mathbf{x}(n)} \qquad (21)$$

Moreover, since the convergence rate of the algorithm is higher at low values of λ it is natural to speed up the initial convergence; at a later time, λ should be increased. A reasonable compromise between convergence rate and dynamic range of the elements in

P(n) can be achieved by using a variable forgetting factor:

$$\lambda(n)=\begin{cases}\dfrac{n-1}{n} & \text{for} \quad 1<n\le\dfrac{1}{1-\lambda}\\ \lambda & \text{for} \quad n>\dfrac{1}{1-\lambda}\end{cases} \tag{22}$$

Equation (22) is equivalent with a more natural way of initializing the cost function $J(n)$ by using the averaging algorithm:

$$J(n)=\begin{cases}\dfrac{n-1}{n}J(n-1)+\dfrac{1}{n}\left|e(n)\right|^2, & 1\le n\le N\\ \dfrac{N-1}{N}J(n-1)+\dfrac{1}{N}\left|e(n)\right|^2, & n>N\end{cases} \tag{23}$$

The relation between $J(n)$ and λ remains the same as in equation (16):

$$J(n)=\lambda(n)J(n-1)+\left(1-\lambda(n)\right)\left|e(n)\right|^2 \tag{24}$$

Since for $n=1$ $\lambda(1)=0$ is not a permitted value, iterations will start at $n=2$ with a initialization value

$$\mathbf{P}(1)=x(0)^{-2} \tag{25}$$

The algorithm starts only when there is a valid (non-zero) input signal, i.e. $x(0)$ exceeds an imposed value.

All other equations remain unchanged.

3. REAL-TIME IMPLEMENTATION ON MOTOROLA SC140 DSP

MATLAB tests have proven that the last discussed version – the modified RLS algorithm in subsection **(2.3)** - is more suitable for a fixed-point implementation than the classical version.

3.1. Motorola StarCore140 Architecture

Motorola StarCore140 core is presented in **Figure 3**. It mainly consists of a data arithmetic logic unit (DALU) including four ALU, address generation unit (AGU), program sequencer and control unit (PSEQ) and phase locked loop (PLL) clock generator.

The core provides 32-bit data and program address space and hardware support for fractional and integer data types. It has a very rich 16-bit wide orthogonal instruction set.

Key features of the SC140 core include sixteen 40-bit data registers for fractional and integer data operand storage, sixteen 32-bit address registers (eight of them can be used as 32-bit base address registers), four address offset registers and four modulo address registers. Each ALU comprises a true (16*16)+40→40-bit MAC unit.

To provide data exchange between the core and the other on-chip blocks, the following buses are implemented: two data memory buses (address and data pairs: XABA and XDABA, XABB and XDBB); program data and address buses (PDB and PAB); special buses.

SC140 performs up to 4MMACS (four million multiply-accumulate operations per second) for each megahertz of clock frequency and up to 10 RISC MIPS. Up to six instructions can be executed in parallel in a single clock cycle.

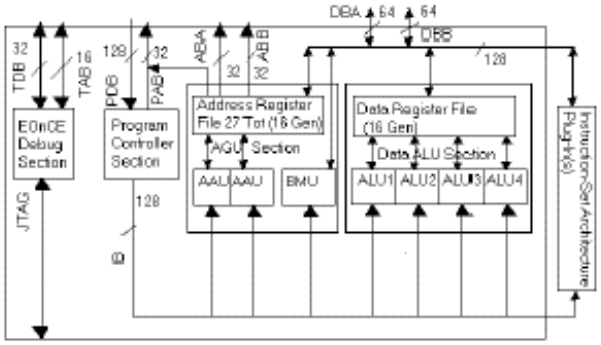

Figure 3 – Block diagram of the SC140 core

StarCore also offers a set of software development tools and utilities. The SC 100 C Compiler converts C source code into highly optimized assembly code, which can then be assembled and linked with other assembly files and library object modules in order to form an executable program. The SC 100 Simulator is an interactive simulation tool that provides cycle-accurate measurements of code execution time. It duplicates the functions of the SC140 DSP core, including memory and register updates associated with code execution and exception processing activity.

3.2. Implementation Issues

With regard to the implementation, several aspects must be underlined. The RLS algorithm contains two division operations. The parameters of the RLS algorithm have a wide dynamic range. A precision of 16 bits proves to be insufficient, even if we tried to use 32 bits representations for the sensitive parameters. In this case, when the parameter **P** becomes 0 or overflows the algorithm freezes.

Preliminary Results

A first practical DSP implementation of the adaptive notch RLS algorithm was performed on another MOTOROLA platform, DSP56311. The DSP 56311 core is based on a 24-bit architecture. Internally, the processor supports 48-bit data representations in order to store multiplication results and to allow for extended precision. The implementation made use of 24-bit data representation for most of the parameters but also 48-bit representations for sensitive parameters. Scaling was employed. The two divisions in the algorithm were performed using 24 bits of precision. This implementation was able to track normalized frequencies ranging from 0.1 to 0.9, at SNR = 10 dB. The total number of instructions per iteration is about 1200. Still, this version is that the algorithm often locked and the algorithm had to be restarted.

The StarCore DSP has a different, more advanced architecture. The core is designed to work with fixed-point data types of 16 and 32 bits. Because the robustness of the fixed-point implementation on DSP 56311 was not satisfactory, we did not attempt another fixed-point implementation.

First we attempted a floating-point implementation. The SC140 compiler, through a user library, supports single precision floating-point format; however it is not supported directly in hardware. The single precision floating-point implementation, as expected, performs well. It is able to track normalized frequencies from 0.05 to 0.95, at SNR = 0 dB. The high number of instructions per iteration is the major drawback of this implementation. It consumes about 10000 processor instructions per sample, and we noted that it is not possible for two cascaded filters to work at the same time, in real-time.

The next attempt was to implement mantissa and exponent data representations. The distinctions from floating-point are: we can use processor specific data formats (16-bit and 32-bit in this case), we can use fixed-point operations to compute mantissas, we can use integer operations to compute exponents, and we can also skip the checking for many of the IEEE exceptions. We call this approach *dynamic scaling*. We use 16 bits for exponent and 16 bits for mantissa in a first attempt. Results show that it is a fast implementation; only 600 instructions per sample are performed. However, the range of frequencies is limited from 0.2 to 0.8, because the precision is limited to 16 bits.

Implementation Details

We have opted for using data representation with 32-bit mantissa and 16-bit exponent, in order to avoid the drawbacks of the 16-bit mantissa & 16-bit exponent implementation. However, there are several issues to be discussed.

A first problem is that, in this case, each arithmetic function must return two variables, the 32-bit mantissa and the 16-bit exponent. The mantissa and the exponent must be though passed by reference and stored in intermediary variables. This requires memory space and needs more time for execution. This was not an issue for the 16-bit mantissa implementation because we could *pack* both 16-bit mantissa and 16-bit exponent into a 32-bit data type that could be returned directly.

The main problem though was that StarCore DSP does not provide divide iteration for 32-bit operands. As a result we had to implement *a fixed-point division algorithm.*

We employed a complex algorithm for division called *SRT division*, which is a form of non-restoring division. This algorithm was developed by three researchers: Sweeney, Robertson, and Tocher, independently from each other. This algorithm was the main cause for the Pentium FDIV bug that was well publicized by the media around 1995. SRT dividers are the most common implementation of digit recurrence division in modern floating-point units. A key feature of this algorithm is that by using a redundant number representation, each quotient digit can be computed using only estimates of the partial remainder and divisor. The algorithm works with a partial remainder, initialized to the dividend, and a partial quotient, initialized to 0. The following recurrence is used in each iteration of the SRT algorithm:

$$P_{j+1} = 2 \cdot P_j - q_{j+1} \cdot divisor \tag{26}$$

where P_j is the partial reminder at iteration j. At each iteration, one digit of the quotient is determined by a quotient-digit selection function:

$$q_{j+1} = SEL\left(P_j, divisor\right) \tag{27}$$

The final quotient after k iterations is then:

$$q = \sum_{j=1}^{k} q_j \cdot 2^{-j} \tag{28}$$

More information about SRT division can be found in (Harris, Oberman, Horowitz 1997). Specifically, for our radix 2 implementation, the digit set for the quotient is $\{-1 \quad 0 \quad 1\}$. We use two registers in order to represent the quotient internally, and the final value for the quotient is computed by subtracting the two registers.

4. EXPERIMENTAL RESULTS

Experimental results were obtained in two ways. The first way is to use the SC 100 Simulator for SC140 DSP; test vectors generated in MATLAB were used. The second one is using the SC140 DSP board; the input signal was generated by specific software and transmitted to the DSP board through the soundboard. We used a fast I/O library working over Ethernet link in order to write the result files. The results obtained were identical.

In order to underline the properties of a 2nd order notch cell, we show in **Figure 4** experimental results obtained in the case of detecting the normalized frequency f=0.4 in high-level background noise (SNR=0dB). FFT was computed in 1024 points.

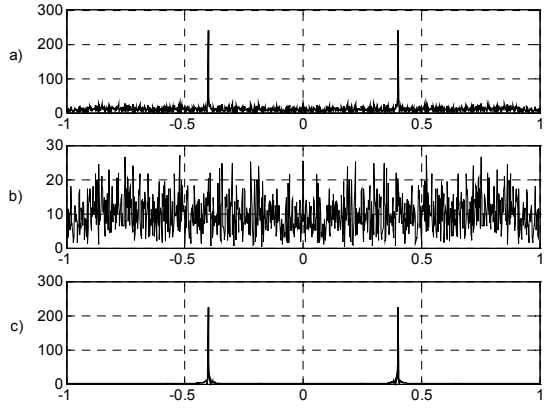

Figure 4 Spectrum of the a) input signal; b) output signal; c) recovered signal for f=0.4, SNR=0dB

Tests show that the accuracy of detecting the notch frequency is higher for normalized frequencies close to

0.25. The accuracy also increases when SNR increases. At low SNR, very low frequencies are detected with difficulty, while at SNR above 5dB this problem does not occur.

Figure 5 presents the variation of the normalized frequency against the sample number at semi logarithmic scale. The studied case is that of the detection of $f = 0.3$ at SNR = 10dB.

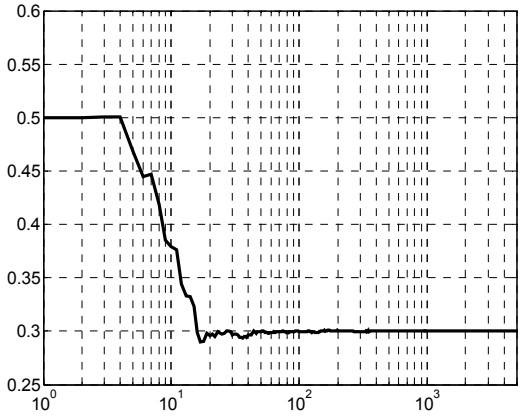

Figure 5 Normalized frequency variation against the sample number

The algorithm "searches" the frequency beginning with the default frequency value $f = 0.5$. It can be seen that the algorithm converges after approximately 150 samples.

Other studied case is that of two cascaded 2nd order filters.

We took into account the case of relatively different normalized frequencies (**Figure 6**) and the case of two close normalized frequencies (**Figure 7**) without background noise. It can be seen that the frequencies in the input signal are detected and cancelled.

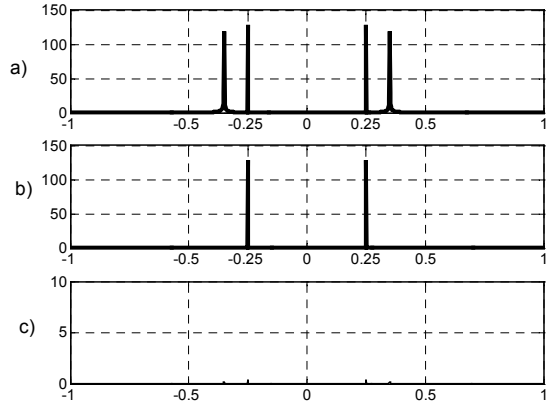

Figure 6 Spectrum of the a) input signal; b) after the first filtration; c) output signal for f_1=0.25, f_2=0.35, no noise

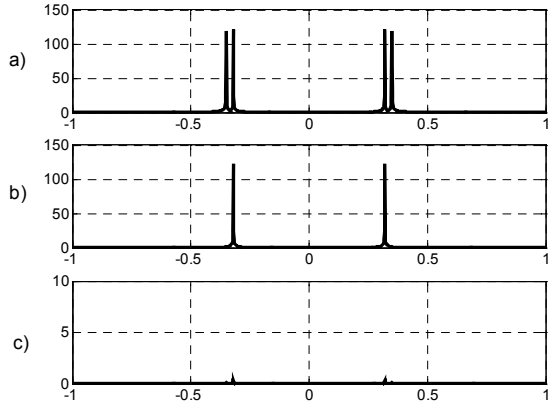

Figure 7 Spectrum of the a) input signal; b) after the first filtration; c) output signal for f_1=0.32, f_2=0.35, no noise

The behavior of the first cells of the cascade form filter should also be discussed. In this case, the first cells always work in conditions of underestimation: every cell is designed to cancel a single sinusoid, but the input signal is multi-sinusoidal. In the case of a second order cell and an input signal containing two sinusoidal components tests show that the presence of the second component leads to a certain bias of the detected frequency. The closer these frequencies are, the higher the bias is.

5. CONCLUSIONS AND PERSPECTIVES

The paper proposes an implementation version on MOTOROLA SC140 DSP of adaptive notch IIR filters. Tap coefficients are updated making use of the modified RLS algorithm proposed in (Ciochină, Paleologu, Enescu, 2003).

Several attempts of implementation have proven that data representation with 32-bit mantissa and 16-bit exponent is the most adequate. A comparison between various implementations that we attempted is presented in the following table:

Table 1

Algorithm Version	Instructions per Sample	Performance, Robustness
Floating point	10000	Very good
Fixed point (on **56311**)	1200	Several problems
16 mantissa 16 exp.	600	Low frequency range
32 mantissa 16 exp.	2500	Very good

This final version of implementation exhibits performances and robustness similar to a floating-point version. The algorithm is able to track a wide range of frequencies (0.05 to 0.95) at SNR = 0 dB and it consumes only 2500 instructions per sample. To all appearance, our last version makes the best tradeoff between numerical accuracy, robustness of the algorithm, and speed of execution.

It must be noted that the number of instructions per sample was obtained by using only the optimization provided by the SC100 C Compiler. Further optimization might be attained using assembly language.

As a perspective, this variant of implementation can be used in order to develop some of the multiple applications involving notch filters.

ACKNOWLEDGEMENT

The authors would like to thank MOTOROLA DSP Center Romania who has provided all necessary hardware and software support for obtaining the results presented in this paper.

REFERENCES

Ciochină S., Paleologu C., Enescu A., 2003, ”On the Behaviour of RLS Adaptive Algorithm in Fixed-Point Implementation”, Proceedings of SCS 2003, pp.57-60

Ciochină S., Slavnicu S., 2001, “Comparative Evaluation of some Least Square Multifrequency Tracking Methods”, Proceedings of SCS 2001, pp. 265-268

Ciochină S., Negrescu C., 1999, “Adaptive Systems”, Editura Tehnica, Bucharest

Harris D. L., Oberman S. F., Horowitz M. A, 1997, “SRT Division Architectures and Implementations”, Computer Systems Laboratory, Stanford University

MOTOROLA 11/2001, SC140 DSP Core Reference Manual

MOTOROLA 06/2000, SC100 C Compiler User’s Manual