J. Software Engineering & Applications, 2010, 3: 391-403
doi:10.4236/jsea.2010.34044 Published Online April 2010 (http://www.SciRP.org/journal/jsea)

DSPs/FPGAs Comparative Study for Power Consumption, Noise Cancellation, and Real Time High Speed Applications

Alon Hayim, Michael Knieser, Maher Rizkalla

Department of Electrical and Computer Engineering, Indiana University Purdue University Indianapolis, Indianapolis, USA.
Email: mrizkall@iupui.edu, mrizkall@yahoo.com

Received December 24th, 2009; revised January 6th, 2010; accepted February 3rd, 2010.

ABSTRACT

*Adaptive noise data filtering in real-time requires dedicated hardware to meet demanding time requirements. Both DSP processors and FPGAs were studied with respect to their performance in power consumption, hardware architecture, and speed for real time applications. For testing purposes, real time adaptive noise filters have been implemented and simulated on two different platforms, Motorola DSP*56303 *EVM and Xilinx Spartan III boards. This study has shown that in high speed applications, FPGAs are advantageous over DSPs with respect of their speed and noise reduction because of their parallel architecture. FPGAs can handle more processes at the same time when compared to DSPs, while the later can only handle a limited number of parallel instructions at a time. The speed in both processors impacts the noise reduction in real time. As the DSP core gets slower, the noise removal in real time gets harder to achieve. With respect to power, DSPs are advantageous over FPGAs. FPGAs have reconfigurable gate structure which consumes more power. In case of DSPs, the hardware has been already configured, which requires less power consumption? FPGAs are built for general purposes, and their silicon area in the core is bigger than that of DSPs. This is another factor that affects power consumption. As a result, in high frequency applications, FPGAs are advantageous as compared to DSPs. In low frequency applications, DSPs and FPGAs both satisfy the requirements for noise cancelling. For low frequency applications, DSPs are advantageous in their power consumption and applications for the battery power devices. Software utilizing Matlab, VHDL code run on Xilinix system, and assembly running on Motorola development systems, have been used for the demonstration of this study.*

***Keywords**: Four Quadrant* (4*Q*) *Converter, Interlacing, Traction Systems, Power Quality Analysis*

1. Introduction

The performance of real-time data processing is often limited to the processing capability of the system. Therefore, evaluation of different digital signal processing platforms to determine the most efficient platform is an important task. There have been many discussions regarding the preference of Digital Signal processors (DSPs) or Field Programmable Gate Arrays (FPGA) in real time noise cancellation. The purpose of this work is to study features of DSPs and FPGAs with respect to their power consumption, speed, architecture and cost. DSP is found in a wide variety of applications, such as filtering, speech recognition, image enhancement and data compression, neural networks, as well as analog linear-phase filters. Signals from the real world received in analog form, then discretely sampled for a digital computer to understand and manipulate. There are many advantages of hardware that can be reconfigured with different programming. Reconfigurable hardware devices offer both the flexibility of computer software, and the ability to construct custom high performance computing circuits. In space applications, it may be necessary to install new functionality into a system, which may have been unforeseen. For example, satellite applications need to adjust to changing operation requirements. With a reconfigurable chip, functionality that is not normally predicted at the outset can be uploaded to the satellite when needed. To test the adaptive noise cancelling, the least mean square (LMS) approach has been used. Besides the standard LMS algorithm, the modified algorithms that are proposed by Stefano [1] and by Das [2] have been implemented for the noise cancellation approach, giving the opportunity of comparing both platforms with respect to their speed, noise, architecture, cost, and power.

2. Adaptive Filter Design on Motorola DSP56300

Adaptive filters have the ability to adjust their own parameters and coefficients automatically. Hence, their design requires little or no prior knowledge of the input signal or noise characteristics of the system. Adaptive filters have two inputs, x(n) and d(n), which are usually correlated in some manner. **Figure 1** gives the basic concept of the adaptive filter.

The filter's output y(n), which is computed with the parameter estimates, is compared with the input signal d(n). The resulting prediction error e(n) is fed back through a parameter adaption algorithm that produces a new estimate for the parameters and as the next input sample is received, a new prediction error can be generated. The adaptive filter features minimum prediction error. Two aspects of the adaptive filter are its internal structure and adaptation algorithm. Its internal structure can be either that of a nonrecursive (FIR) filter or that of a recursive (IIR) filter. An adaptation algorithm can be divided into two major classes; gradient algorithms and nongradient algorithms. A gradient algorithm is used to adjust the parameters of the FIR filter. The least mean square (LMS) algorithm is the most widely applied gradient algorithm. This adjusts the filter's parameters to minimize the mean-square error between the filter's output y(n) and the desired response input d(n) [3]. When an adaptive filter is implemented on the DSP56300 processer, address pointer to mimic FIFO (First-In-First-Out)-like shifting of the RAM data, modulo addressing capability to provide wrap around data buffers, multiply/accumulate (MAC) instruction top both multiply two operands and add the product to a third operand in a single instruction cycle, data move in parallel with the MAC instructions to keep the multiplier running at 100% capacity and Repeat Next Instruction (REP) to provide compact filter code are being used by the processor. The processor's capability to perform modulo addressing allows an address register (Rn) value to be incremented (or decremented) and yet remain within an address range of size L, where L is defined by a lower and an upper address boundary. For the adaptive FIR filter, L is the number of coefficients (taps). The value L-1 is stored in the processor's Modifier Register (Mn). The upper address boundary is calculated by the processor and is not stored in a register. When modulo addressing is used, the Address Register (Rn) points to a modulo data buffer located in X-Memory and/or Y-Memory. The address pointer (Rn) is not required to point at the lower address boundary; it can point anywhere within the defined modulo address range L. If the address pointer increments past the upper address boundary (base address plus L-1 plus 1), it will wrap around to the base address. Modulo Register M1 is programmed to the value NTAPS-1 (modulo NTAPS). Address Register R1 is programmed to point to the state variable modulo buffer located in X-Memory. Modulo Register M4 is programmed to the value NTAPS-1. Address Register R4 is programmed to point to the coefficient buffer located in Y-Memory. Given that the FIR filter algorithm has been executing for some time and is ready to process the input sample x(n) in the Data ALU input Register X0, the address in R4 is the base address (lower boundary) of the coefficient buffer. The address in R1 is M, where M is greater than or equal to the lower boundary of X-Memory address and less than or equal to the upper boundary of X-Memory address. The X-Memory map for the filter states, the Y-Memory map for the coefficients, and the contents of the processor's A and B Accumulators and Data ALU Input Registers X0, X1, Y0 and Y1 are shown in the **Figure 2**. The CLR instruction clears the A-Accu-

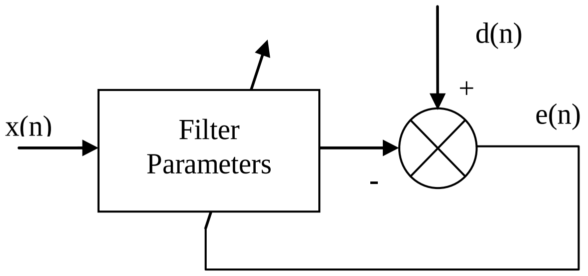

Figure 1. Basic concept of the adaptive filter

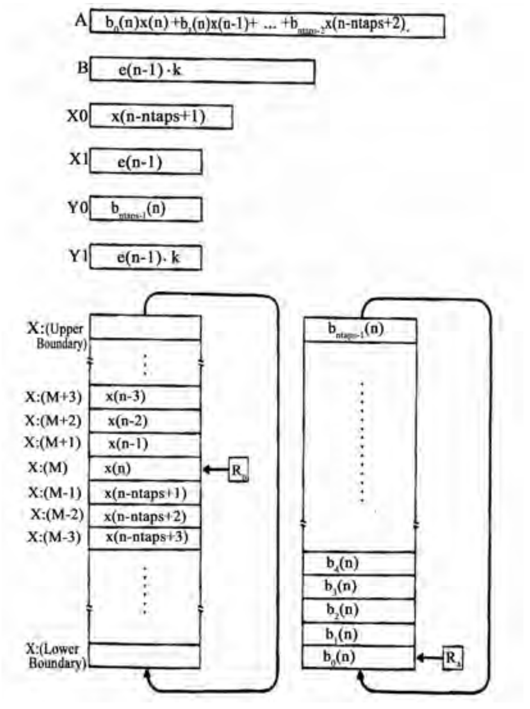

Figure 2. Memory map and data registers after last MAC instruction

mulator and simultaneously moves the input sample x(n) from the Data ALU's Input Register X0 to the X-Memory location pointed to by address register R1, and moves the first coefficient from the Y-Memory location pointed to by address register R4 to the Data ALU's Input Register Y0. Both Address Registers R1 and R4 are automatically incremented by one at the end of the CLR instruction (post-incremented). The REP instruction regulates execution of NTAPS-1 iteration of the MAC instruction. The MAC instruction multiplies the filter state variable X0 by the coefficient in Y0, adds the product to the A-Accumulator and simultaneously moves the next state variable from the X-Memory location pointed to by the Address Register R1 to the Input Register X0, and moves the next coefficient from the Y-Memory location pointed to by Address Register R4 to Input Register Y0. Both Address Registers R1 and R4 are automatically incremented by one at the end of the MAC instruction (post-incremented).

During the execution of the filter algorithm, Address Register R4 is post incremented to a total of NTAPS times; once in conjunction with the CLR instruction and NTAPS-1 times (due to the REP instruction) in conjunction with the MAC instruction. Since the modulus for R4 is NTAPS and R4 is incremented NTAPS times, the address value in R4 wraps around and points to the coefficient buffer's lower boundary location [3]. Also Address Register R1 is post incremented to a total NTAPS times; once in conjunction with the CLR instruction and NTAPS-1 times (due to the REP instruction) in conjunction with the MAC instruction. Also at the beginning of the algorithm, the input sample x(n) is moved from the Data ALU Input Register X0 to the X-Memory location pointed to by R1. Since the modulus for R1 is NTAPS and R1is incremented NTAPS times, the address value in R1 wraps around and points to the state variable buffer's X-Memory location M. The MACR instruction calculates the final tap of the filter algorithm and performs convergent rounding of the result. The data move portion of this instruction loads the input sample x(n) into the B-Accumulator. At the end of the MACR instruction, the accumulator contains the filter output sample y(n) as shown in **Figure 3**.

The two Move instructions transfers the loop gain K to the data register Y1 and the error sample e(n) to the Data Input Register X1. The first MOVE instruction in the "do loop" transfers the parameter $b_i(n)$ to the A-Accumulator and the filter state x(n-i) to the Data Input Register X0. Address Register R1 is incremented by one to point to the next filter state. The MAC instruction multiplies the filter state, in X0, by the product of the loop gain and the error sample, in Y1, and adds the product to the A-Accumulator. The result in the A-Accumulator is the updated parameter $b_i(n+1)$. The second Move instruction in the "do loop" transfers the parameter $b_i(n+1)$ to the Y-Memory location pointed to by the Address Register R4. R4 is incremented by one to point to the next filter parameter as shown in **Figure 4**. The LUA instruction decrements R1 by one, and R1 then points to the state variable buffer's X-Memory location M-1. When the algorithm is executed, a new (next) input sample x(n+1) will overwrite the value in X-Memory location M-1. Thus FIFO-like shifting of the filter state variables is accomplished by adjusting the R1 address pointer as shown in **Figure 5**.

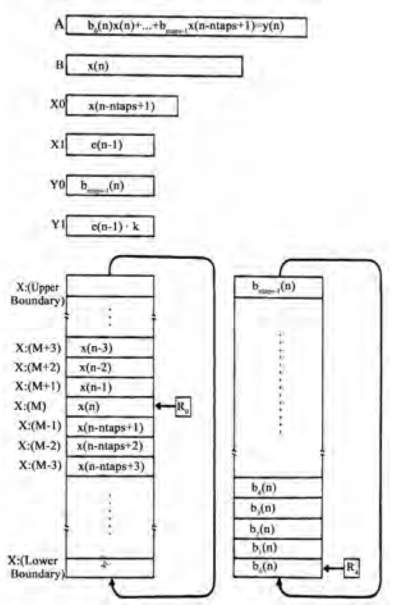

Figure 3. Memory map and data registers after MACR instruction

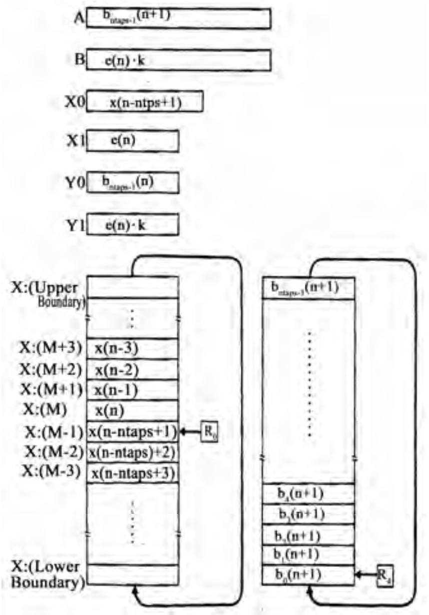

Figure 4. Memory map and data registers after last pass of do loop

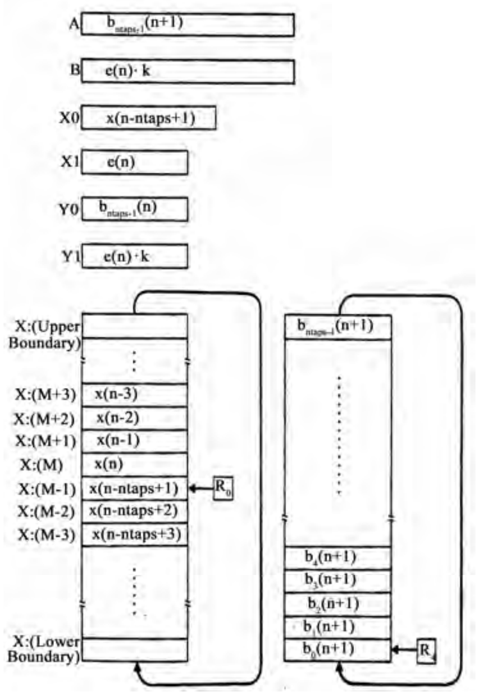

Figure 5. Memory map and data registers after LUA instruction

Consider the problem of finding the linear minimum mean square estimate (LMMSE) of a zero-mean signal vector, S, from a noisy zero-mean data vector, $X = S + N$, where N denotes the additive noise vector. A LMMSE of S is given in Equation (1), where A denotes a matrix of filter coefficients as given in Equation (2).

$$S = A \cdot X \quad (1)$$

$$S = C_{SS}\left(C_{SS} + C_{nn}\right)^{-1} X \quad (2)$$

Here, C_{SS} and C_{nn} denote the covariance matrices of signal and noise, respectively. Notice that if X has a non-zero mean vector, μ, Equation e becomes:

$$S = \mu + C_{SS}\left(C_{SS} + C_{nn}\right)^{-1}\left(X - \mu\right) \quad (3)$$

For point-wise processing of a non-stationary signal of a local mean, μ_S, and local variance, σ_S^2, and the noise to be zero-mean, white with a local variance, σ_n^2, the point-wise LMMSE will be given by:

$$S = \mu_S + \left[\frac{\sigma_S^2}{\sigma_S^2 + \sigma_n^2}\right]\left[x - \mu_S\right] \quad (4)$$

σ_n^2 is constant, while σ_S^2 and μ_S vary with the time index, k. Thus the filtered estimate at time, k can be written as:

$$S = \mu_S(k) + \left[\frac{\sigma_S^2(k)}{\sigma_S^2(k) + \sigma_n^2}\right]\left[x(k) - \mu_S(k)\right] \quad (5)$$

where $\mu_S(k)$ and $\sigma_S^2(k)$ denote the time varying estimates of local mean and local variance of $S(k)$. An improved version of Lee's adaptive wiener filter has been proposed by Das [4]. The main contributions of this algorithm include a better technique for estimation of noise variance, and incorporation of a data window for adaptive filtering. Lee's adaptive wiener filter suffers from two major drawbacks. First, it requires prior knowledge of noise power and second, its performance deteriorates when the signal-to-noise ratio (SNR) is low and noise power is imprecisely known. The improved wiener filter incorporates two modifications. First, the de-noising performance of the filter is improved by introducing a non-rectangular window to process weighted data samples and second, a scheme for online estimation of noise power is incorporated which is based on analyzing the power spectral density, $S(\omega)$, of the data. Assuming that the observed data consists of predominantly low-frequency signal components and additive white noise, then $S(\omega)$ can be modeled as a sum of the spectral density of the signal and a constant, σ_n^2, which represents the variance of noise. The estimated σ_n^2 is the average value of the high-frequency section of $S(\omega)$ [2]. The improved wiener filter can be done in a fashion similar to that of Lee's wiener filter, but Equation (2) now takes the form $S = AWX$, where A denotes a matrix of filter coefficients, and W is a (diagonal) data weighting matrix. The LMMSE of S is now given by Equation (6), where $X_W = WX$, and similarly, the point-wise LMMSE is given by

$$S = C_{SS}\left(C_{SS} + C_{nn}\right)^{-1} X_W \quad (6)$$

$$S = \mu_S + \left[\frac{\sigma_S^2}{\sigma_S^2 + \sigma_n^2}\right]\left[X_W - \mu_S\right] \quad (7)$$

3. FPGAs Adaptive Filter Design

The efficient realization of complex algorithms on FPGAs requires a familiarity with their specific architectures. The modifications needed to implement an algorithm on an FPGA and also the specific architectures for adaptive filtering and their advantages are given below.

3.1 FPGA Realization Issues

FPGAs are ideally suited for the implementation of adaptive filters. However, there are several issues that need to be addressed. When performing software simulations of adaptive filters, calculations are normally carried out with floating point precision. Unfortunately, the resources required of an FPGA to perform floating point arithmetic are normally too large to be justified. Another concern is the filter tap itself. Numerous techniques have been devised to efficiently calculate the convolution operation when the filter's coefficients are fixed in advance. For an adaptive filter whose coefficients change over time, these methods will not work or need to be modified significantly [5]. The reconfigurable filter tap is the most important issue for high performance adaptive filter architecture, and as such it will be discussed at length.

3.2 Finite Precision Effects

Although computing floating point arithmetic in FPGA is possible, it is usually accomplished with the inclusion of custom floating point units, which are costly in terms of logic resources. Therefore, a small number of floating point units can be used in the entire design, and must be shared between processes. This does not take full advantage of the parallelization that is possible with FPGAs and is therefore not the most efficient method. All calculation should therefore be mapped into fixed point only, but this can introduce some errors. The main errors in DSP include ADC quantization error, coefficient quantization error, overflow error caused impermissible word length, and round off error. The other three issues will be addressed later.

3.2.1 Scale Factor Adjustment

A suitable compromise for dealing with the loss of precision when transitioning from a floating point to a fixed-point representation is to keep a limited number of decimal digits. Normally, two to three decimal places is adequate, but the number required for a given algorithm to converge must be found through experimentation. When performing software simulations of a digital filter for example, it is determined that two decimal places is sufficient for accurate data processing. This can easily be obtained by multiplying the filter's coefficients by 100 and truncating to an integer value. Dividing the output by 100 recovers the anticipated value. Since multiplying and dividing be powers of two can be done easily in hardware by shifting bits, a power of two can be used to simplify the process. In this case, one would multiply by 128, which would require seven extra bits in hardware. If it is determined that three decimal digits are needed, then ten extra bits would be needed in hardware, while one decimal digit requires only four bits. For simple convolution, multiplying by a preset scale and then dividing the output by the same scale has no effect on the calculation. For a more complex algorithm, there are several modifications that are required for this scheme to work [6]. The first change needed to maintain the original algorithm's consistency requires dividing by a scale constant any time and previously scaled values are multiplied together. Consider, for example, the values *a* and *b* and the scale constant *s*, the scaled integer values are represented by $s \cdot a$ and $s \cdot b$. To multiply theses values requires dividing by *s* to correct for the s^2 term that would be introduced and recover the scaled product $a \cdot b$.

$$\left(s \cdot a \times s \cdot b\right) \big/ s = s \cdot ab \tag{8}$$

Likewise, division must be corrected with a subsequent multiplication. It should now be evident why a power of two is chosen for the scale constant, since multiplication and division by power of two results in simple bit shifting. Addition and subtraction require no additional adjustment. The aforementioned procedure must be applied with caution, however, and does not work in all circumstances. While it is perfectly legal to apply to the convolution operation of a filter, it may need to be tailored for certain aspects of a given algorithm. Consider the tap-weight adaptation equation for the LMS algorithm in Equation (9).

$$\hat{w}(n+1) = \hat{w}(n) + \mu \cdot u(n)\dot{e}(n) \tag{9}$$

where μ is the learning rate parameter; its purpose is to control the speed of the adaptation process. The LMS algorithm is convergent in the mean square provided in Equation (10).

$$0 < \mu < \frac{2}{\lambda_{MAX}\Pi} \tag{10}$$

where $\lambda_{MAX}\Pi$ is the largest eigenvalue of the correlation matrix R_x of the filter's input. Typically this is a fraction value and its product with the error term has the effect of keeping the algorithm from diverging. If μ is blindly multiplied by some scale factor and truncated to a fixed-point integer, it will take on a value greater than one. The affect will be to make the LMS algorithm diverge, as its inclusion will now amplify the added error term. The heuristic adopted in this case is to divide by the inverse value, which will be greater than one. Similarly, division by values smaller than one should be replaced by multiplication with its inverse. The outputs of the algorithm will then need to be divided by the scale to obtain the true output. The following algorithm describes the fixed point conversion:

Determine Scale

Through simulations, find the needed accuracy (# decimal places).

Scale = accuracy rounded up to a power of two.

Multiply all constants by scale

- Divide by scale when two scaled values are multiplied.

- Multiply by scale when two scaled values are divided.

Replace

For multiplication by values less than 1

- Replace with division by the reciprocal value.

Likewise, for division by values less than 1

Replace with multiplication by the reciprocal value.

3.2.2 Training Algorithm Modification

The training algorithms for the adaptive filter need some minor modifications in order to converge for a fixed-point implementation. Changes to the LMS weight update equation were discussed in the previous section.

Specifically, the learning rate μ and all other constants should be multiplied by the scale factor. When μ is adjusted it takes the form in Equation (11). With μ modification weight update Equation (11) can be modified as in Equation (12).

$$\hat{\mu} = \frac{1}{\mu} \cdot scale \tag{11}$$

$$\hat{w}(n+1) = \hat{w}(n) + \frac{u(n)\dot{e}(n)}{\hat{\mu}} \tag{12}$$

The direct form FIR structure has a delay that is determined by the depth of the output adder tree, which is dependent on the filter's order. The transposed FIR, on the other hand, has a delay of only one multiplier and one adder, regardless of the filter length. It is therefore advantageous to use the transposed form for FPGA implementation to achieve maximum bandwidth. **Figure 6** shows the direct and **Figure 7** shows the transposed FIR structures for a three tap filter. The relevant nodes have been labeled A, B and C for a data flow analysis. Each filter has three coefficients, and are labeled h_0[n], h_1[n] and h_2[n]. The coefficients' subscript denotes the relevant filter tap, and the n subscript represents the time index, which is required since adaptive filters adjust their coefficients at every time instance.

The direct FIR structure is shown in **Figure 6** and the output y at any time n is given by Equation (13), where nodes B and C are described in Equations (14) and (15) respectively.

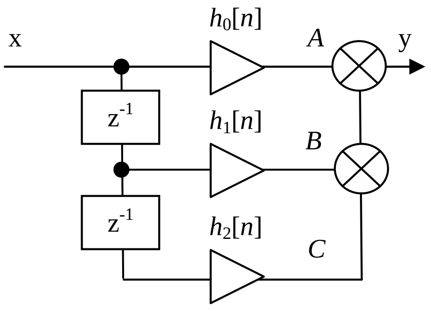

Figure 6. Direct form FIR structure

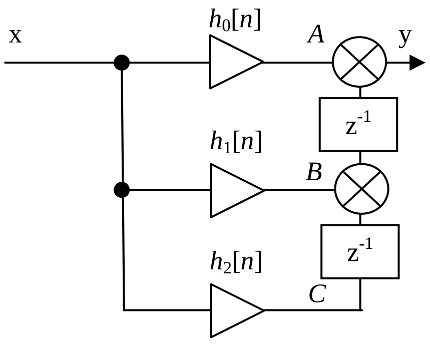

Figure 7. Transposed form FIR structure

$$y[n] = A[n] = x[n]h_0[n] + B[n] \tag{13}$$

$$B[n] = x[n-1]h_1[n] + C[n] \tag{14}$$

$$C[n] = x[n-2]h_2[n] \tag{15}$$

$$y[n] = x[n]h_0[n] + x[n-1]h_1[n] + x[n-2]h_2[n] \tag{16}$$

$$y[n] = \sum_{k=0}^{N=2} x[n-k]h_k[n] \tag{17}$$

The transposed FIR structure is shown in **Figure 7** and the output y at any time n is given below.

$$y[n] = x[n]h_0[n] + B[n-1] \tag{18}$$

$$B[n] = x[n]h_1[n] + C[n-1] \tag{19}$$

$$C[n] = x[n]h_2[n] \tag{20}$$

$$y[n] = x[n]h_0[n] + x[n-1]h_1[n-1] + x[n-2]h_2[n-2] \tag{21}$$

$$y[n] = \sum_{k=0}^{N=2} x[n-k]h_k[n-k] \tag{22}$$

Compared with the direct FIR output, the difference in the [n-k] index of the coefficient indicates that the filters produce equivalent output only when the coefficients don't change with time. This means if the transposed FIR architecture is used, the LMS algorithm will not converge differently from the direct implementation is used [7]. The change needed was to account for the weights as shown in Equation (23). A suitable approximation is to update the weights at every N input, where N is the length of the filter. This obviously will converge N times slower. Though simulations show that it never actually converges with as good results as the traditional LMS algorithm. It may be acceptable still though, due to the increased bandwidth of the transposed form FIR, when high convergence rates are not required.

$$\hat{w}(M-n+1) = \hat{w}(M-n) + \frac{u(n)\dot{e}(n)}{\mu \cdot scale} \tag{23}$$

3.3 Implementing Adaptive Noise Filter with FPGAs

Adaptive noise filtering techniques are applied to low frequency like voice signals, and high frequency signals such as video streams, modulated data, and multiplexed data coming from an array of sensors. Unfortunately in all high frequency and high speed applications, a software implementation of the adaptive noise filtering usually doesn't meet the required processing speed, unless a high end DSP processor is used. A convenient solution can be represented by a dedicated hardware implementation using a Field Programmable Gate Array (FPGA). In this case the limiting factor is represented by a number of multiplications required by the adaptive noise cancellation algorithm. By using a novel modified version of the LMS algorithm, the proposed implementation allows the use of a reduced number of hardware multipliers. Moreover experimental data showed that the modified algorithm achieves the same or even better performances than the standard LMS version. There are many possible implementations for an adaptive noise filter, but the most widely used employs a Finite Impulse Response (FIR) digital filter, whose coefficients are iteratively updated using the LMS algorithm. The algorithm is described in Equations (24) to (26), leading to the evaluation of the FIR output, the error, and the weights update.

$$Y_i = X_i^T W_i \tag{24}$$

$$e_i = D_i - Y_i \tag{25}$$

$$W_{i+1} = W_i + 2\mu e_i X_i \tag{26}$$

In the above equations, X_i is a vector containing the reference noise samples, D_i is the primary input signal, W_i is the filter weights vector at the i^{th} iteration, and e_i is the error signal. The μ coefficient is often empirically chosen to optimize the learning rate of the LMS algorithm. The hardware implementation of the algorithm in an FPGA device is not trivial, since the FIR filter has not constant coefficients, so multipliers cannot be synthesized by using a look-up table (LUT) based approach. This however, should be straightforward in FPGA architecture. Multipliers with changing inputs instead need to be built by using a significantly greater number of internal logic resources (either elementary logic blocks or embedded multipliers). In an Nth order filter the algorithm requires at least 2N multiplications and 2N additions. Note the factor 2μ that is usually chosen to be a power of two in order to be executed by shifting. This makes it impractical for fully parallel hardware implementation of the algorithm as the value of N grows. This is due to the huge number of multipliers required. In order to reduce the complexity of the algorithm, the weights update expression (Equation (26)) is simplified as in Equation (27).

$$W_{i+1} = W_i + \alpha e_i \operatorname{sgn}(X_i) = W_i + \Delta_i \tag{27}$$

As a consequence the weights are updated using a factor proportional to the error and the sign of the current reference noise sample, instead of its value. This implies that weights can be updated by using an addition (or subtraction) instead of a multiplication. This simplified algorithm requires only N multiplications and 2N additions. However the simplification of the weights update rule usually results in worse learning performances, *i.e.* in a slower adaptation capability of the filter. To overcome this weakness, and significantly improve the learning characteristics, a dynamic learning rate coefficient α has been used. Generally this can be done by updating it with an adaptive rule, or, by using a heuristic function. Simulations of the above mentioned method shows that a dynamic learning rate gives an advantage not only in the learning characteristics, but also in the accuracy of the final solution (in term of improvement of the signal to noise ratio of the steady state solution). The product αe_i is used to update all weights; only one additional multiplication is required.

3.4 Architecture for Implementation on FPGA

The architecture of the adaptive noise filtering based on the modified LMS algorithm is shown in **Figure 8**. It was designed to implement 32 tap adaptive noise filter in a medium density FPGA device. It has a modular and scalable structure composed by 8 parallel stages, each one capable of executing 1 to 4 multiply and accumulate (MAC) operations and weights update. By controlling the number of operation performed by each block it is possible to implement an adaptive filter whose order can range from 8 to 32. In the first case, by exploiting maximum parallelism, the filter is capable of processing a data sample per clock cycle. In the other cases 2 to 4 clock cycles are requested. Some FPGA's internal RAM blocks were used to implement the tap delays and to store weights coefficients. Each weights update block is mainly composed by an adder/subtractor accumulator. The weights update coefficients Δ_i are computed by a separated block, which also handles the learning rate update function, following the above mentioned heuristic algorithm, and implements its multiplication with the error signal. By slightly modifying this unit, a more sophisticated adaptive function, can be easily obtained, thus enhancing the performances of the adaptive noise filtering for non stationary signals.

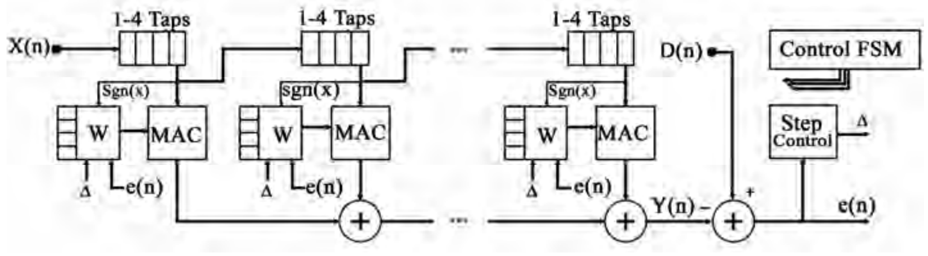

Figure 8. Architecture of the modified LMS filter

4. Simulations and Results

Adaptive noise filters have been implemented on DSPs and FPGAs. Motorola DSP56303 has been used for DSP platform, while Xilinx Spartan III boards are used to implement FPGA adaptive noise filtering. Matlab Simulink has been used to test the effectiveness and correctness of the adaptive filters before hardware implementation.

4.1 Matlab Simulink Simulations and Results

To test the theory and see the improvements visually that is proposed by Das, the adaptive filter that is proposed by Lee and Das has been compares through Matlab Simulink. (see **Figure 9**)

The target simulink model is responsible for code generation where as the host simulink model is responsible for testing. The host drives the target model with heavy wavelet noisy test data consisting of 4096 samples generated from wnoise function in Matlab. Matlab's fdatool is used for designing the bandpass filter to color the noise source. A colored Gaussian noise is then added to the input test signal. This noisy signal and the reference noise are inputs to the terminal of the LMS filter Simulink block. **Figure 10** Desired Signal (top), received signal (middle), output (bottom) This code has been implemented in C programming language. The LMS filter is placed in the virtual internal ram of the simulink model. In the code, breakpoints are placed in the corresponding section of the code where FIR filtering takes place. It takes 46, 213 and 266 clock cycles to run the filtering section. The time computation would be the clock cycles measured, divided by 225 MHz, which is the virtual clock speed. The execution time is 205 ms. The implementation of LMS filter takes worst case time of 38.95 ms to compute the filtering of heavy sine noisy signal consisting of 4096 samples per frame. **Figure 11** shows the comparison between the Das proposal of the wiener filter and the Lee's wiener filter proposal in the signal to noise ratio aspect. As it can be seen from the **Figure 11** the performance for the Das proposal is higher than the Lee's wiener filter. The improved adaptive wiener filter provides SNR improvement from 2.5 to 4 dB as compared to Lee's adaptive wiener filter.

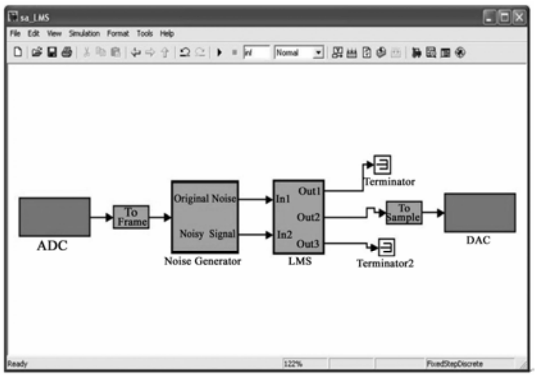

Figure 9. Block diagram of Matlab Simulink

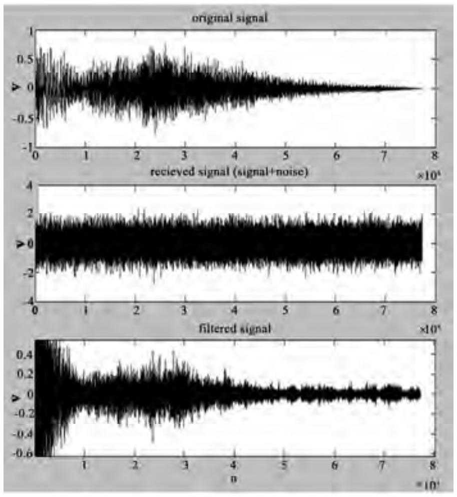

Figure 10. Desired signal

4.2 Motorola DSP56300 Results

The DSP system consists of two analog-to-digital (A/D) converters, and two digital-to-analog converters (D/A) converters. The DSP56303EVM evolution module is used to provide and control the DSP56300 processor, the two A/D converters, and the two D/A converters. The left analog input signal $x(t)$ consists of the desired input signal $s(n)$ plus a white noise signal $w(n)$. The left analog input signal $x(t)$ is first digitized using the A/D converter on the evaluation board. DSP Processor executes the adaptive filter algorithm to process the left digitized input signal $x(n)$, the left and right output signals y_1(n) and $y_2(n)$ will be generated. The left output signal $y_1(n)$ is the error signal. The right output signal $y_2(n)$ is the filtered version of the left digitized input signal $x(n)$, which is an estimate of the desired input signal $s(n)$. The two D/A converters on the evaluation board are then used to convert the left and right digital output signals $y_1(n)$ and $y_2(n)$ to the left and right analog output signals $y_1(t)$ and $y_2(t)$.

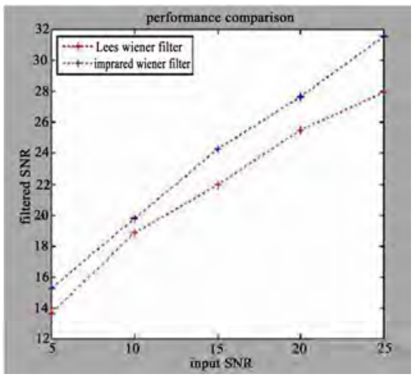

Figure 11. SNR performance comparison between Lee and Das proposals

The continuous analog signal was sampled at a rate of twice the highest frequency present in the spectrum of the sampled analog signal in order to accurately recreate the analog audio signal from the discrete samples. The analog audio signal was mixed with noise using a sum block which is bound to occur when the audio signal passes through the channel. The noise however, first low pass passed filter using a finite impulse response filter to make it finite in bandwidth. FIR noise filter was observed to have little or no significant effect on the signal with noise. The information bearing signal is a sine wave of $0.055\frac{cycles}{sample}$ is shown in **Figure 12**. The noise picked up by the secondary microphone is the input for the adaptive filter as shown in **Figure 13**. The noise that corrupts the sine wave is a low pass filtered version of the noise. The sum of the filtered noise and the information bearing signal is the desired signal for the adaptive filter. The noise corrupting the information bearing signal is a filtered version of noise as shown in the **Figure 14**. **Figure 15** shows that the adaptive filter converges and follows the desired filter response. The filtered noise should be completely subtracted from the signal noise combination and the error signal should only have the original signal. The results can be seen in **Figures 12** to **16**.

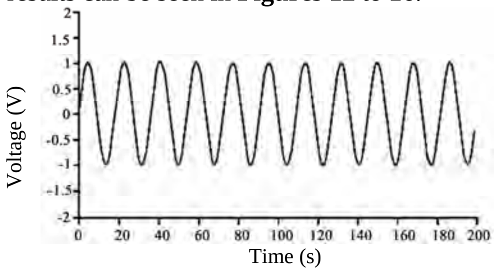

Figure 12. Plot showing the input signal

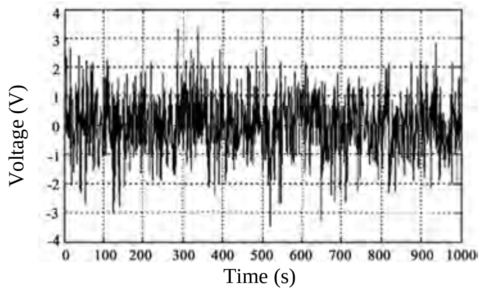

Figure 13. Plot of the noise signal

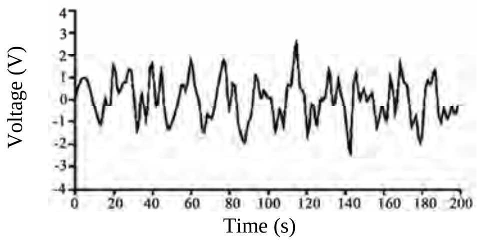

Figure 14. Noise corrupting the original signal

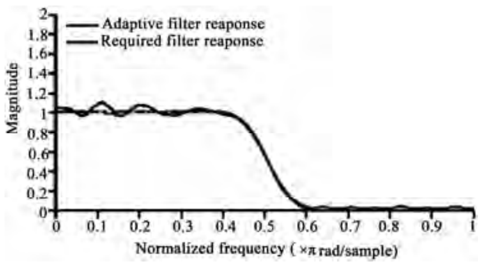

Figure 15. Convergence of the adaptive filter response to the response of the FIR filter

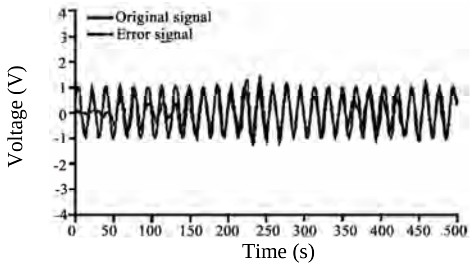

Figure 16. Plot of the error and the original signal

4.3 Xilinx Spartan III Results

The algorithm for adaptive filtering were coded in Matlab and experimented to determine optimal parameters such as the learning rate for the LMS algorithm. After the parameters have been determined, algorithms were coded for Xilinx in VHDL language.

4.3.1 Standard LMS Algorithm Results

The desired signal output was a sine wave, and it was corrupted by a higher frequency sinusoid and random Gaussian noise with a signal to noise ratio of 5.86 dB. The input signal can be seen in **Figure 17**. A direct form FIR filter of length 32 is used to filter the input signal. The adaptive is trained with the LMS algorithm with a learning rate $\mu = 0.05$. It appears that the filter with the standard LMS algorithm has learned the signal statistics and is filtering within 200-250 iterations. Since the results have shown that the standard LMS algorithm removes the noise from the signal, the next section. The timing analyzer has showed that the clock for standard LMS algorithm is 25 MHz. The input and output signals for the standard LMS algorithms are given in **Figures 17** and **18**.

4.3.2 Modified LMS Algorithm Results

The noise reduction obtained by both the standard LMS algorithm and the modified algorithm as applied to a sta-

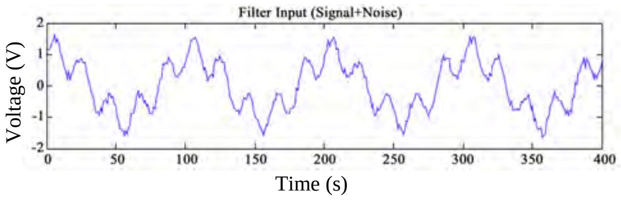

Figure 17. Input signal for standard LMS algorithm

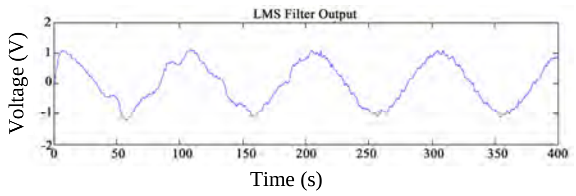

Figure 18. Output signal for standard LMS algorithm

tionary signal composed by 3 frequencies, corrupted by a random Gaussian noise, with signal to noise ratio of 5.86 dB were studied. Both algorithms used 16 bit fixed point representation for data and filter coefficients [14]. The frequency spectrum of the original signal, standard LMS, and modified LMS filter are given in **Figure 19**. The modified LMS used a dynamic learning rate coefficient α based on a heuristic function formerly proposed by Widrow [8], and consisted of 1/n decaying function, coefficients were approximated by a piecewise linear curve, starting from the value 0.1 down to 0.001 (in about 1000 steps). This heuristic function achieved a faster convergence, and les gradient noise. It has proved to be effective when applied to stationary signals. On the other hand the standard LMS used a static learning rate with the best performances obtained by setting the μ parameter equal to 0.05. The two algorithms reported noise attenuation greater than 40 dB and 36 dB respectively. As can be seen from the two learning characteristics in **Figure 20**

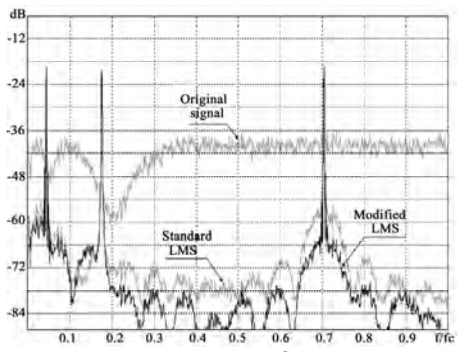

Figure 19. Frequency Spectrum of a signal processed with the standard and modified LMS

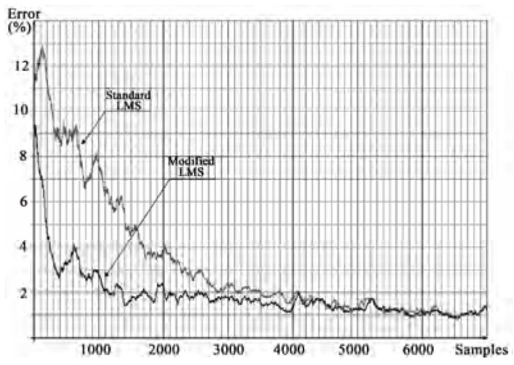

Figure 20. Learning Characteristics of both LMS algorithms

the modified LMS offered a faster convergence. A large class of signals (either stationary or short term stationary) and noises showed similar simulation results. The adaptive noise filtering was implemented using a 16 bit 2's complement fixed point representation for samples and weights. As it can be seen in **Figure 5**, the floor planned design required 1776 slices (logic blocks) of 3072 available (about 57%), and allowed a running clock frequency of 50 MHz (with a non optimized, fully automatic place & route process). It would require 2750 slices (89%) and would run at less than 25 MHz (due mainly to routing congestion). The Assembly file used for the simulation is given in Appendix A. The assembly code is provided elsewhere [26].

5. Conclusions

As discussed in the previous chapters, the concept of the adaptive noise filtering applications can be implemented in both DSP processors like Motorola DSP56300 series and also in the Field Programmable Gate Array such as Xilinx Spartan III boards. In high performance signal processing applications, FPGAs have several advantages over high end DSP processors. Literature survey has showed that high-end FPGAs have a huge throughput advantage over high performance DSP processors for certain types of signal processing applications. FPGAs use highly flexible architectures that can be greatest advantage over regular DSP processors. However, FPGAs come with a hardware cost. The flexibility comes with a great number of gates, which means more silicon area, more routing and higher power consumption. DSP processors are highly efficient for common DSP tasks, but the DSP typically takes only a tiny fraction of the silicon area, which is dedicated for computation purposes. Most of the area is designated for instruction codes and data moving. In high performance signal processing applications like video processing, FPGAs can take highly parallel architectures and offer much higher throughput as compared to DSP processors. As a result FPGA's overall energy consumption may be significantly lower than DSP processors, in spite of the fact that their chip level power consumption is often higher. DSP processors can consume 2-3 watts, while the FPGAs can consume in the order of 10 watts. The pipeline technique, more computation area and with more gates FPGAs can process more channels at the same time. Thus power consumption per channel is significantly less in the FPGA's [15]. DSPs are specialized forms of microprocessor, while the FPGA's are form of highly configurable hardware. In the past, the usage of DSPs has been nearly ubiquitous, but with the needs of many applications outstripping the processing capabilities (MIPS) of DSPs, the use of FPGAs has become very prevalent. It has generally come to be expected that all software, (DSP code is considered a type of software) will contain some bugs and that the best can be done is to minimize them. Common DSP software bugs are caused because of, failure of interrupts to completely restore processor state upon completion, non-uniform assumptions regarding processor resources by multiple engineers simultaneously developing and integrating disparate functions, blocking of critical interrupt by another interrupt or by an uninterruptible process, undetected corruption or non-initialization of pointers, failing to properly initialize or disable circular buffering addressing modes, memory leaks, the gradual consumption of available volatile memory due to failure of a thread to release all memory when finished, dependency of DSP routines on specific memory arrangements of variables, use of special DSP "core mode" instruction options in core, conflict or excessive latency between peripheral accesses, such as DMA, serial ports, L1, L2, and external SDRAM memories, corrupted stack or semaphores, subroutine execution times dependent on input data or configuration, mixture of "C" or high-level language subroutines with assembly language subroutines, and pipeline restrictions of some assembly instructions [15]. Both FPGA and DSP implementation routes offer the option of using third party implementation for common signal processing algorithms, interfaces and protocols. Each offers the ability to reuse existing IP in the future designs. FPGA's are more native implementation for more DSP algorithms. **Figures 21** and **22** give the block diagrams of the DSP and FPGA respectively.

Motorola DSP56300 series can only do one arithmetic

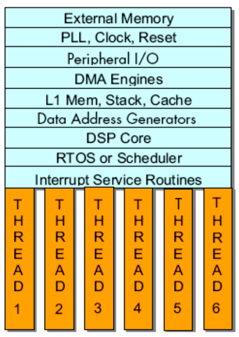

Figure 21. Digital signal processor block diagram

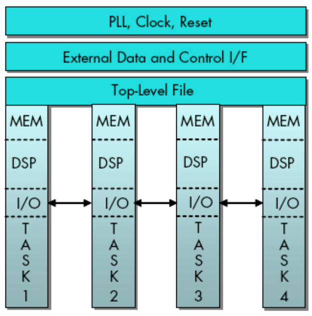

Figure 22. FPGA's block diagram

computation and two move instructions at a time. However, in the case of FPGAs, each task can be computed by its own configurable core and designated input and output interface.

5.1 Speed Comparison

Speed is one of the most important concepts that determine the computation time and also it is one of the most important concepts in the market. In the adaptive filters the parameters are updated with the each iteration and after the each iteration the error between the input and the desired signal get smaller. After some number of iterations the error becomes zero and the desired signal is achieved. According to the specifications from the manufacturer manuals, Motorola DSP56300 series has a CPU clock of 100 MHz, but this speed depend on the instruction fetch, computation speed and also the speed of the peripherals. On the DSP56303EVM board the audio codec runs on 24.57 MHz, this clock speed is determined by an external crystal. In the other hand Xilinx Spartan 3 has the maximum clock frequency of 125 MHz, but this speed can be reduced because of the number of instruction ns, gates and the congestion on the routing of the signals. Both of the modified adaptive noise filtering applications take about 200-250 iterations to cancel the noise and achieve the desired signal. In the Motorola DSP processor case because of the actual clock speed being lower, causality conditions and the speed limitation that is coming from the audio codec part of the board, the running time of the modified LMS algorithm is 20 MHz. in the case of the FPGA's the running speed is around 50 MHz. This due to discussions from the previous section, which is FPGA's flexibility and reconfigurable gates allows for the clock to be faster.

5.2 General Conclusions

As discussed in the previous sections, we have shown the differences between the DSP processors and FPGAs. As far as power and cost are considered, DSP processors in general have lower power consumption, which makes them suitable for battery powered applications. These applications can be done on audio applications. These voice applications are very straight forward and do not require sophisticated pipeline and parallel moves. Audio applications can be different filter applications. These are used especially in the voice transmission lines and cell phones. When it comes to the high frequency applications, DSP processors have some restrictions on their part when they are compared to the FPGAs. In high speed applications, FPGA's are much faster than the DSP processors. When it comes to high speed applications, the DSP boards have some limitations when compared to the FPGAs. FPGAs can offer more channels, and thus when cost per channel is considered because FPGAs can offer more channels, the cost per channel is lower than the DSP's. Also the partitioning of the FPGA's can offer more throughputs as compared to DSP processors. Thus FPGAs can handle multiple tasks when their controls and finite state machines are configured correctly.

According to our study, the final conclusion is that for simple audio applications like adaptive noise cancelling, Motorola DSP56300 is more beneficial, because the requirements for audio applications are met with DSP processors. Also they are more power efficient and can be used for battery powered devices. But when adaptive noise filtering is considered in high speed applications like video streaming and multiplexed array signals, FPGA's are offering a faster approach and thus they are more suitable for high frequency applications.

5.3 Future Work

In the future, the adaptive noise filtering can be implemented on high frequency applications, such as noise removal from video streaming and noise removal from multiplexed data arrays. These applications may be applied first to FPGAs with Verilog HDL or VHDL. After application has been verified, hardware code can be converted to a net list and thru Synopsys a custom ASIC design can created. The ASIC design and FPGA design may be compared in the aspect of cost, power, architecture, noise removal and speed. These comparisons would be helping us to provide us a more educated choice for future applications.

REFERENCES

[1] A. Di Stefano, A. Scaglione and C. Giaconia, "Efficient FPGA Implementation of an Adaptive Noise Canceller," *Proceedings Seventh International Workshop on Computer Architecture for Machine Perception,* Palermo, 2005, pp. 87-89.

[2] M. El-Sharkawy, "Digital Signal Processing Applications with Motorola's DSP56002 Processor," Prentice Hall, Upper Saddle River, 1996.

[3] K. Joonwan and A. D. Poularikas, "Performance of Noise Canceller Using Adjusted Step Size LMS Algorithm," *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, Huntsville, 2002, pp. 248-250.

[4] R. M. Mersereau and M. J. T. Smith, "Digital Filtering A Computer Laboratory Textbook," John Wiley & Sons, Inc., New York, 1994.

[5] J. Proakis and D. Manolakis, "Digital Signal Processing Principles, Algorithms, and Applications," 4th Edition, Pearson Prentice Hall, Upper Saddle River, 2007.

[6] G. Saxena, S. Ganesan and M. Das, "Real Time Implementation of Adaptive Noise Cancellation," *EIT* 2008 *IEEE International Conference on Electro/Information Technology,* Ames, 2008, pp. 431-436.

[7] K. L. Su, "Analog Filters," Chapman & Hall, London, 1996.

[8] B. Widrow, J. R. Glover, Jr., J. M. McCool, J. Kaunitz, C. S. Williams, R. H. Hearn, J. R. Zeidler, Eugene Dong, Jr., and R. C. Goodlin, "Adaptive Noise Cancelling: Principles and Applications," *Proceedings of the IEEE*, Vol. 63, 1975, pp. 1692-1716.

[9] S. M. Kuo and D. R. Morgan, "Active Noise Control: A Tutorial Review," *Proceedings of the IEEE*, Vol. 87, No. 6, June 1999, pp. 943-973.

[10] K. C. Zangi, "A New Two-Sensor Active Noise Cancellation Algorithm," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, Vol. 2, 1993, pp. 351-354.

[11] A. V. Oppenheim, E. Weinstein, K. C. Zangi, M. Feder, and D. Gauger, "Single-Sensor Active Noise Cancellation," *IEEE Transactions on Speech and Audio Processing,* Vol. 2, 1994, pp. 285-290.

[12] T. H. Yeap, D. K. Fenton and P. D. Lefebvre, "Novel Common Mode Noise Cancellation Techniques for xDSL Applications," *Proceedings of the* 19*th IEEE Instrumentation and Measurement Technology Conference*, Anchorage, Vol. 2, 2002, pp. 1125-1128.

[13] Xilinx Corp., "Spartan-IIIE 1.8V FPGA Family: Functional Description," November 2002.

[14] B. Dukel, M. E. Rizkalla and P. Salama, "Implementation of Pipelined LMS Adaptive Filter for Low-Power VLSI Applications," *The* 45*th Midwest Symposium on Circuits and Systems,* Tulsa, Vol. 2, 2002, pp. II-533- II-536.

[15] M. Das, "An Improved Adaptive Wiener Filter for De-noising and Signal Detection," *International Association of Science and Technology for Development, International Conference on Signal and Image Processing,* Honolulu, 2005, p. 258.

[16] K. Schutz, "Code Verification using RTDX," MathWorks Matlab Central File Exchange.

[17] S. Haykin, "Adaptive Filter Theory," Englewood Cliffs, Prentice Hall, Upper Saddle River, 1991.

[18] D. L. Donoho and J. M. Johnstone, "Ideal Spatial Adaptation by Wavelet Shrinkage," *Biometrika,* Vol. 81, 1 September 1994, pp. 425-455.

[19] J. Petrone, "Adaptive Filter Architectures for FPGA Implementation," Master's Thesis, Department of Electrical and Computer Engineering, Florida State University, Tallahassee, 2004.

[20] S. Manikandan and M. Madheswaran, "A New Design of Adaptive Noise Cancellation for Speech Signals Using Grazing Estimation of Signal Method," *International Conference on Intelligent and Advanced Systems,* Kuala Lumpur, 2007, pp. 1265-1269.

[21] K. Chang-Min, P. Hyung-Min, K. Taesu, C. Yoon-Kyung and L. Soo-Young, "FPGA Implementation of ICA Algorithm for Blind Signal Separation and Adaptive Noise Canceling," *IEEE Transactions on Neural Networks*, Vol. 14, 2003, pp. 1038-1046.

[22] S. M. Kay, "Fundamentals of Statistical Signal Process-

ing," Prentice Hall, Upper Saddle River, 1996.

[23] J.-S. Lee, "Digital Image Enhancement and Noise Filtering by Use of Local Statistics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, 1980, pp. 165-168.

[24] Alon Halim, "Real Time Noise Cancellation Field Programmable Gate Arrays," MSECE Thesis, Purdue University, Lafayette, 2009.

Appendix A

```
init_filter macro
        move      #states,r1
        move      #ntaps-1,m1
        move      #coef,r4
        move      #ntaps-1,m4
        endm
stafirmacro       ntaps,lg,foutput,ferror
clr   a       x0,x:(r1)+ y:(r4)+,y0
rep   #ntaps-1
mac   x0,y0,a         x:(r1)+,x0 y:(r4)+,y0
macr  x0,y0,a     x:(r1),b
move          a,x:foutput
sub   a,b
nop
move          b,x:ferror
move          #lg,y1
move          b,x1
mpy   x1,y1,b
move          b,y1
do    #ntaps,_update
move          y:(r4),a          x:(r1)+,x0
mac   x0,y1,a
move          a,y:(r4)+
_update
lua   (r1)-,r1
nop
move          x:foutput,b
move          x:ferror,a
endm
```