

## Fast Parallel Implementation of DFT Using Configurable Devices\*

Andreas Dandalis and Viktor K. Prasanna  
Department of Electrical Engineering-Systems  
University of Southern California  
Los Angeles, CA 90089-2562, USA  
{dandalis, prasanna}@usc.edu  
<http://ceng.usc.edu/~prasanna>  
Tel: +1-213-740-4336, Fax:+1-213-740-4418

**Abstract.** In this paper we propose a fast parallel implementation of Discrete Fourier Transform (DFT) using FPGAs. Our design is based on the Arithmetic Fourier Transform (AFT) using zero-order interpolation. For a given problem of size  $N$ , AFT requires only  $O(N^2)$  additions and  $O(N)$  real multiplications with constant factors. Our design employs  $2p + 1$  PEs ( $1 \leq p \leq N$ ),  $O(N)$  memory and fixed I/O with the host. It is scalable over  $p$  ( $1 \leq p \leq N$ ) and can solve larger problems with the same hardware by increasing the memory. All the PEs have fixed architecture. Our implementation is faster than most standard DSP designs for FFT. It also outperforms other FPGA-based implementations for FFT, in terms of speed and adaptability to larger problems.

### 1 Introduction

The Discrete Fourier Transform (DFT) plays a fundamental role in digital signal processing. The complexity and computation time of algorithmic approaches for forward computation of DFT, are essential issues in algorithms where many forward DFTs are required while one inverse Fourier transform must be performed at the end. For a problem of size  $N$ , the sequential computation time of a straightforward approach is  $O(N^2)$  and is characterized mainly by the large number of complex multiplications and additions. This fact limits the computational performance of the approach as well as the algorithmic efficiency of implementations using Field Programmable Gate Arrays (FPGAs).

The FPGA based implementations for computing the DFT, proposed in [12, 8, 11], use the Fast Fourier Transform (FFT) to reduce the computation time and complexity to  $O(N \log_2 N)$ . The basic computation unit is the butterfly. Butterfly is a repetitive structure that has 2 inputs and 2 outputs. It involves

---

\* This research was performed as part of the MAARC project (Models, Algorithms and Architectures for Reconfigurable Computing, <http://maarc.usc.edu>). This work is supported by DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-00049 monitored by Fort Hauchuca.

one complex multiplication, one complex addition and one complex subtraction. For a problem of size  $N$ , the algorithm requires  $\log_2 N$  stages with  $N/2$  butterflies in each stage. Even though these designs optimize the structure of the butterfly, the complexity still remains high. All these designs are solutions optimized for a particular problem size. For larger problems, re-design is required resulting in area penalty. Parallelism is not exploited and the designs are not scalable except the one proposed in [11]. In [8], the idea of FPGAs with an external multiplier is used to overcome the critical issue of complex multiplication. This solution has still problems since it adds extra control/complexity and requires a large number of I/O pins for interfacing the multiplier chip. In spite of this, the computation time is not attractive. The implementation in [11] uses the CORDIC approach for optimizing the butterfly by eliminating multiplications. Again, the resulting performance is not attractive.

In this paper we propose a novel parallel, scalable, partitioned solution for computing the DFT using FPGAs, based on the Arithmetic Fourier Transform (AFT). Using this approach, we can solve larger problems with fixed hardware, simply by increasing the memory size. We can linearly speed-up the computation proportionally to the number of PEs employed and achieve superior performance compared with previous FPGA-based solutions. Also, it offers faster solution compared with most standard DSP designs for computing the DFT. The key idea of our design is the use of an algorithmic approach to the problem. Contrary to traditional approaches, we perform an algorithmic design for reconfigurable devices, based upon the architecture/features of the device. While known techniques map an algorithm for DFT onto the device and perform device dependent optimizations, our methodology employs algorithm synthesis techniques instead of logic synthesis. This alleviates the FPGA's restriction of fast/compact adders vs slow/area-consuming multipliers. Complex multiplication is a critical issue in DSP applications and can lead to poor performance of FPGA-based solutions. AFT turns to be a suitable algorithmic approach for FPGAs since it is less complex than the FFT and performs real multiplications with constant factors instead of complex multiplications.

The Arithmetic Fourier Transform is based on the Möbius inversion formula of series and has been shown to be competitive with the conventional FFT in terms of accuracy, complexity and speed [9]. It needs  $O(N^2)$  additions and  $O(N)$  real multiplications by constant coefficients. It reduces the computation time of DFT to  $O(N)$ . In our design, two sets of  $p$  PEs ( $1 \leq p \leq N$ ) and an additional PE are used for computing  $2N + 1$  Fourier coefficients [7]. Our design is scalable over  $p$  ( $1 \leq p \leq N$ ), thus it can achieve  $O(p)$  speed-up. It is also a partitioned solution since it can solve larger problems by increasing the memory size in proportion to the  $N$  the size of the problem. In each set, all PEs have the same architecture and perform additions and zero-order interpolation. The additional PE performs the scaling of the intermediate values by constant factors. All the PEs are cascaded using pipelining. The data as well as the control signals move from left to right. The complete design requires  $O(N)$  memory and has fixed I/O bandwidth. External memory is used for storing the scaling factors as well

as intermediate Fourier coefficients. Constant coefficients multiplier (KCM) [2], is used for performing the scaling operation. KCMs use the Distributed Arithmetic approach (DA) and turn out to be a very efficient choice for digital signal processing in terms of speed and area. The compact size and high performance of the KCMs compared with standard full multipliers, are promising features that make the AFT algorithm an efficient solution for computing the DFT using FPGAs. In addition, the parallel/modular structure, the regular architecture as well as the fixed, independent of the problem size I/O bandwidth, make our approach an attractive solution for implementation in FPGAs.

Preliminary estimations shows that our design achieves speed-up of 2-10 over most standard DSP designs for 256-FFT. Compared with the Fastest FFT in the West [12], the CORDIC approach [11] and the implementation in [8], our design outperforms these solutions in terms of speed and adaptability to larger size problems. Our preliminary implementation reported here using Xilinx devices, can be further optimized resulting in higher speed and less area.

This paper is organized as follows. In Section 2 we describe the Arithmetic Fourier Transform while in Section 3 we introduce our scalable architecture for AFT. In Section 4 the computation time and area estimations are shown. Finally in Section 5 comparisons are discussed and concluding remarks are made.

## 2 Arithmetic Fourier Transform

The Arithmetic Fourier Transform (AFT) is based on the Möbius inversion formula of series. Since it involves only additions and real multiplications by constant factors, it is computationally less complex than FFT while it achieves  $O(\log_2 N)$  speed-up over it. An introduction to AFT is given below and detailed descriptions of it can be found in [7, 9].

Given  $2N$  input samples  $A(m)$ ,  $m = 0, 1, \dots, 2N - 1$ , we compute an average and  $2N$  alternating averages over them. All these averages are scaled by constant factors and then the Möbius inversion formula is applied for the computation of  $2N + 1$  Fourier coefficients. The Möbius inversion formula theorem [5] and the definition of the alternating average, are the key mathematical tools for the AFT algorithm.

**Theorem (The Möbius inversion formula)** Let  $f(n)$  be a non vanishing function in the interval  $1 \leq n \leq N$  and  $f(n) = 0$  for  $n > N$ , where  $n, N$  are positive integers. If

$$g(n) = \sum_{m=1}^{\lfloor N/n \rfloor} f(mn)$$

then

$$f(n) = \sum_{k=1}^{\lfloor N/n \rfloor} \mu(k)g(kn)$$

where  $\lfloor \dots \rfloor$  denotes the integer part of a real number and  $\mu(k)$  is the Möbius function.

**The Möbius function** is defined as

$$\begin{aligned}\mu(n) &= 1 && \text{if } n = 1 \\ \mu(n) &= (-1)^r && \text{if } n = p_1 p_2 \dots p_r \text{ where } p_i (i=1, 2, \dots, r) \text{ distinct primes} \\ \mu(n) &= 0 && \text{if } p^2 \mid n \text{ for some prime } p\end{aligned}$$

**Definition (Alternating Average)** The  $2n$ th alternating average  $B(2n, \alpha)$  of the  $2n$  values  $A(mT/2n + \alpha T)$ ,  $0 \leq m \leq 2n - 1$ , is defined as:

$$B(2n, \alpha) = \frac{1}{2n} \sum_{m=0}^{2n-1} (-1)^m A(mT/2n + \alpha T)$$

where  $\alpha$  is a shifting factor,  $-1 < \alpha < 1$ . Assuming now a finite Fourier series  $A(t)$  with period  $T$ , we can represent it as:

$$A(t) = a_0 + \sum_{n=1}^N a_n \cos 2\pi n f_0 t + \sum_{n=1}^N b_n \sin 2\pi n f_0 t$$

where  $f_0 = 1/T$ ,  $a_n$  and  $b_n$  are the real and imaginary parts of the Fourier coefficients of the non vanishing function in the interval  $-N \leq n \leq N$  and  $a_0$  is the mean of  $A(t)$ . Applying the Möbius inversion formula to  $A(t)$ , we can compute the  $2N + 1$  Fourier coefficients in terms of the alternating averages as follows:

$$\begin{aligned}a_0 &= \frac{1}{2N} \sum_{m=1}^{2N} A(m), & a_n &= \sum_{l=1, 3, \dots}^{\lfloor N/n \rfloor} \mu(l) B(2n, 0) \\ b_n &= \sum_{l=1, 3, \dots}^{\lfloor N/n \rfloor} \mu(l) (-1)^{\frac{l-1}{2}} B(2n, \frac{1}{4nl})\end{aligned}$$

where  $n = 1, 2, \dots, N$  and  $m = 0, 1, \dots, 2N - 1$ .

For computing the alternating averages  $B(2n, 0)$ ,  $B(2n, \frac{1}{4nl})$  from the input samples  $A(m)$ , we use zero-order interpolation for computational efficiency [9]. In this method we interpolate an unknown value  $A(mT/2n + \alpha T)$  to a known input sample  $A(i)$ , where  $i$  is the integer part of  $mT/2n + \alpha T$ . The resulting error due to this approximation is shown to be tolerable [9, 10]. The AFT computation method presented above, requires  $(2N + 1)^2$  additions and  $(2N + 1)$  multiplications with constant factors, for computing  $2N + 1$  Fourier coefficients. The reduced complexity, the use of scaling by constant factors instead of complex multiplications and the amenability to parallel processing makes AFT more desirable computationally than FFT [10].

In Figure 1 we can show the structure of the AFT algorithm. In Part I the non-scaled alternating averages are computed while in Part II the scaling operation and the computation of  $a_0$  take place. Finally, in Part III the  $2N$  Fourier coefficients are computed. Multiplications with constant factors are performed only in Part II while in the other parts additions and zero-order interpolation are performed. The Möbius values in the last part define the sign of the alternating averages.

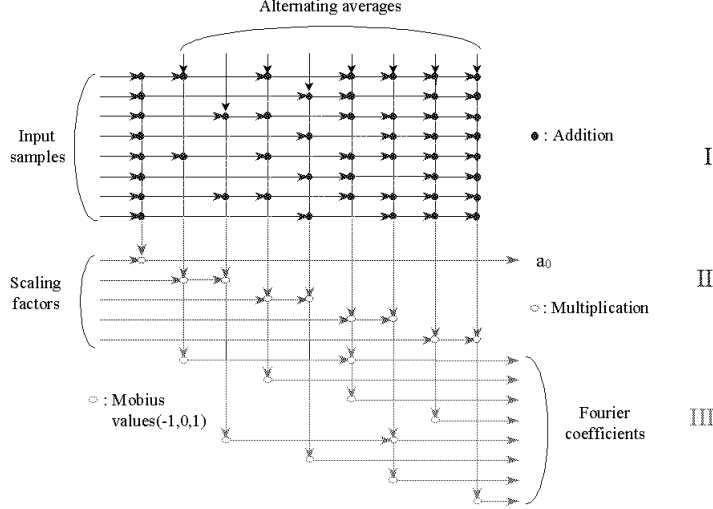


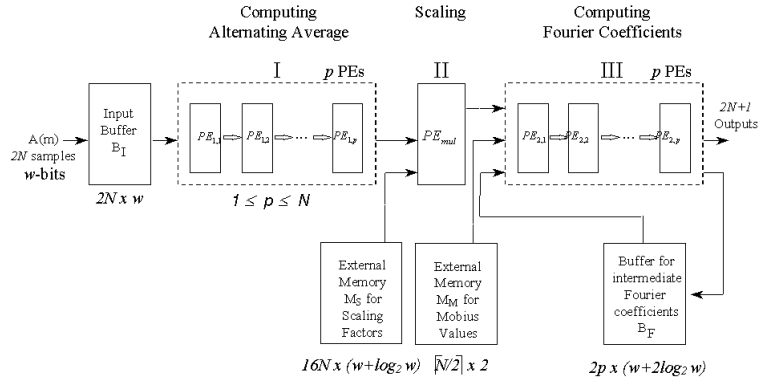
Fig. 1. Structure of AFT

### 3 A Scalable Architecture for AFT

In this section we show a scalable architecture to map the AFT algorithm (see Figure 2). Each part of the architecture corresponds to a part of the AFT structure in Figure 1. Data and control signals flow from left to right.

In Part I,  $p$  PEs are employed to compute the alternating averages. An input buffer  $B_I$  of size  $2N \times w$  is used for storing the input window of  $2N$  samples, where  $w$  denotes the number of bits in each input sample. Assuming that  $p$  divides  $N$ , each window is fed  $N/p$  times into the pipe. Let  $PE_{1,i}$  denote the  $i$ th PE in Part I,  $1 \leq p \leq N$  and  $n = 1, 2, \dots, N$ . In  $PE_{1,n \bmod p}$ , the alternating averages  $B(2n, 0)$  and  $B(2n, \frac{1}{4nl})$  are computed during the  $[n/p]$ th feeding of the input data window into the pipe. Each PE checks if the received data is needed for its computation based on the zero-order interpolation. The interpolation is implemented using local registers and a comparator for checking the index of the received sample. Every  $2N$  units,  $p$  alternating averages are computed. Thus, the total computation time for  $2N$  alternating averages is  $\frac{2N^2}{p} + 2p - 1$  units. All the PEs in this part have the same architecture and consist of one adder, one comparator and local registers. The local registers are used for performing the interpolation as well as for interconnection with other PEs.

In Part II, one PE is employed for scaling the averages computed in the previous part and for computing the mean  $a_0$ . This PE is denoted as  $PE_{mul}$ . Totally  $2p$  scaled averages are computed every  $2N$  time units.  $PE_{mul}$  employs



**Fig. 2.** Overall Architecture

one Constant Coefficient Multiplier (KCM) [2] for computing  $a_0$  as well as for scaling  $B(2n, 0)$ ,  $B(2n, \frac{1}{4nI})$  by constant factors during each step. Using the hybrid technique described in [2], we have to precalculate 16 values for each scaling factor. The hybrid technique of multiplication is a hexademical equivalent of the long hand method. Since a single hex digit represents four bits, the look-up table for each constant factor has entries for 0 to 15 ( $F$ ). Thus, the size of the external memory  $M_S$  would be  $16N \times (w + \log_2 w)$ . The set of precalculated values of constant factor  $\frac{1}{2i}$  is stored in 16 consecutive locations of the memory, starting from the  $i$ th memory location where  $1 \leq i \leq N$ .  $PE_{mul}$  also employs local registers for interconnection with other stages.

In Part III,  $p$  PEs are employed to compute the Fourier coefficients. Similar to Part I, the PEs in this part are denoted as  $PE_{2,i}$ . All the PEs have the same architecture. Each of them consists of one adder, one comparator and local registers. As in the first group, each processing element checks if the received average is needed in its partial sum. This checking is performed using the index of the incoming average. The Möbius values required for the computation of the partial sums are provided by an external memory  $M_M$ . The memory size is  $\lceil \frac{N}{2} \rceil$  and the stored values are  $\{-1, 0, 1\}$ . Few local registers are employed for controlling the data flow between consecutive PEs. A buffer  $B_f$  is employed for storing the intermediate results of the computation. When a new set of  $2p$  alternating averages are available to Part III, the intermediate results of the computation and the Möbius function values are fed back from the rightmost

to the leftmost PE. The size of  $B_f$  is  $2p \times (w + 2 \log_2 w)$  where  $w$  denotes the number of bits of each input sample. In this part, only the Fourier coefficients  $a_1, b_1, a_2, b_2, \dots, a_{N/3}, b_{N/3}$  are computed since for  $n > N/3$ ,  $a_n$  and  $b_n$  are equal to alternating averages  $B(2n, \alpha)$ .

Since the computation time of  $2N$  alternating averages is  $\frac{2N^2}{p} + 2p - 1$  units, the total computation time for  $2N+1$  Fourier coefficients becomes  $\frac{2N^2}{p} + 3p + \lceil \frac{p}{2} \rceil$  units. The throughput rate of KCM critically affects the overall performance since it determines the minimum time unit. The architecture employs  $2p$  adders, 1 KCM, few local registers and external memories. The total size of the external memories is  $O(N)$ . A key advantage of the design is that it is scalable over  $p$ ,  $1 \leq p \leq N$ , thus it can linearly speed-up the computation by increasing  $p$ . It can also solve larger size problems (at a lower throughput rate) by simply increasing the memory but still using the same number and structure of PEs.

## 4 Performance Estimates

Table 1 lists the estimated area for various components of our architecture. We estimated the area of each of the functional blocks and the total area for each PE was then derived. All the PEs in a group (I or II) have the same architecture. Thus, each of them occupies the same constant area. We have assumed 16-bit input data and that our design is mapped onto Xilinx XC4000 series of FPGAs.

CLBs per Function	$PE_{1,i}$	$PE_{2,i}$	$PE_{mul}$
Registers	110	140	50
16-bit comparator	2	2	-
16-bit adder	16	16	-
16-bit KCM	-	-	230
Control	2	2	2
<b>Total CLBs</b>	130	160	282

**Table 1.** PE area requirements in terms of number of CLBs to realize the function

Our design consists of  $p$   $PE_{1,i}$ ,  $p$   $PE_{2,i}$  and one  $PE_{mul}$ . Assuming  $2N$  input samples, the total time for computing the  $2N+1$  Fourier coefficients is  $\frac{2N^2}{p} + 3p + \lceil \frac{p}{2} \rceil$  time units. The time unit is determined by the performance of the KCM since Part II is the most time-consuming stage of our design. The pipelined performance of a 16-bit operand KCM (after place and route) is  $50MHz$  [2] using the  $-3$  speed grade components. The performance of a 16-bit adder is in the range of  $100MHz$  using  $-3$  speed grade components [6]. Thus, there is enough time for the PEs in parts I and III to complete their operations using  $50MHz$

clock rate. Currently,  $-2$  speed grade devices are available. Thus,  $50\text{MHz}$  system clock rate is an achievable goal for the entire design. Table 2 shows the area requirements and estimate of the computation time for two designs. We assume 256 input samples and  $50\text{MHz}$  system clock rate.

Hardware	Area Requirements	Computation Time
$p = 8$	2602 CLBs, 3 XC4025	4124 time units (82.48 $\mu\text{sec}$ )
$p = 16$	4922 CLBs, 5 XC4025	2104 time units (42.08 $\mu\text{sec}$ )

**Table 2.** Area and performance estimates

## 5 Comparisons and Conclusions

In this paper, we have proposed a novel parallel, scalable, partitioned solution for computing the DFT using FPGAs. Our solution based on the AFT turns out to be more efficient than the FFT based approach in terms of area and speed. Our design is scalable over  $1 \leq p \leq N$ , where  $p$  is the number of PEs employed. We can also linearly speed-up the computation proportionally to  $p$ . The architecture of the PEs and the I/O bandwidth are fixed and independent of the problem size. The required memory is  $O(N)$ . Our design can solve larger problems (with reduced throughput) with fixed hardware.

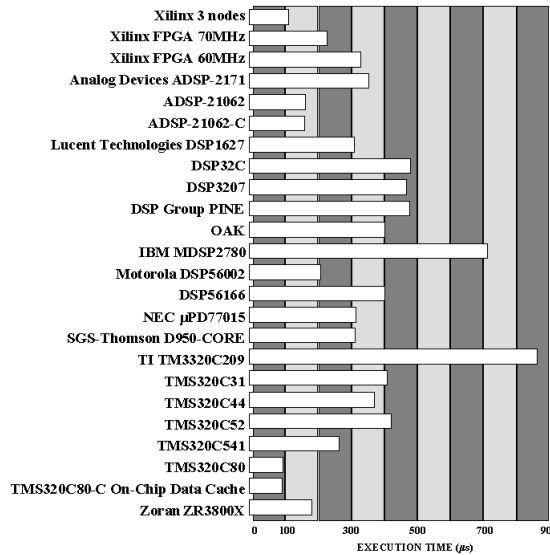
In Figure 3 and in Table 3 the execution times of various designs for 256-FFT are shown. The input samples are 16-bit data for all the designs. Figure 3 shows the results of a benchmark evaluation [12] of DSP-based and Xilinx FPGA-based designs. Our design achieves speed-up of 2 – 10 over most single chip DSP designs for 256-FFT.

Implementation	Area Requirements	Computation Time
<b>Xilinx 3 nodes</b> [12]	1 XC4025	102.4 $\mu\text{sec}$
<b>Xilinx 70MHz</b> [12]	1 XC4025	223 $\mu\text{sec}$
<b>Xilinx 60MHz</b> [12]	1 XC4025	312.5 $\mu\text{sec}$
<b>PDSP16116/A</b> [8]	2 Chips	61.4 $\mu\text{sec}$
<b>CORDIC</b> <sup>1</sup> [11]	10 XC4010	5000 $\mu\text{sec}$

**Table 3.** Performance of FPGA-based designs for 256-FFT

---

<sup>1</sup> 1000-FFT



**Fig. 3.** Performance of DSP-based and FPGA-based designs for 256-FFT (from[12])

In Table 3 the performance of five FPGA-based implementations are shown. Three of them are from “The Fastest FFT in the West” [12]. In that work a radix-2 butterfly FFT design was used. The implementation in [8] makes use of one Altera FPGA and a PDS P16116/A 16-bit complex multiplier to overcome the critical problem of performing complex multiplication in FPGAs. A radix-4 design and necessary control were mapped onto the Altera FPGA. The implementation in [11] uses the CORDIC approach to eliminate the complex multiplications. Even though it computes a 1000-point FFT, the performance is not attractive compared with our approach. Table 3 also shows the area requirements of these implementations.

Our design is faster than the earlier FPGA-based implementations. The implementations in Table 3 are designs optimized for a particular problem size and device features and need to be redesigned for larger problems. Our design can also handle larger problems with the same fixed hardware by increasing the memory. Known implementations exploit the power of a single chip while we have developed a scalable and partitioned solution with high performance and adaptability to larger problems. Our performance estimation has been based on a preliminary implementation and no optimizations have been performed. Further improvement of the performance of our design is possible. Parallelism can be exploited; for example parallel I/O can improve the performance significantly. Many registers can be eliminated by efficiently performing zero-order

interpolation. Other interpolation approaches (such as first-order) can also be exploited.

The work reported here is part of the USC MAARC project. This project is developing algorithmic techniques for realising scalable and portable applications using configurable computing devices and architectures. Contrary to traditional approaches to configurable computing, in our approach the user “sees” the architecture/device features and uses algorithm synthesis techniques instead of logic synthesis. We are developing computational models and algorithmic techniques based on these models to exploit dynamic reconfiguration. In addition, compilation onto reconfigurable hardware is also addressed. Some related results can be found in [1], [3], [4].

## References

1. K. Bondalapati and V. K. Prasanna, “Reconfigurable Meshes: Theory and Practice”, *Reconfigurable Architectures Workshop, Int. Parallel Processing Symposium (IPPS)*, April 1997.
2. K. Chapman, “Constant Coefficient Multipliers for the XC4000C”, XILINX Application Note 054, Dec. 1996.
3. S. Choi, Y. Chung and V. K. Prasanna, “Configurable Hardware for Symbolic Search Operations”, submitted to *Int. Conf. Parallel and Distributed Systems*, Dec. 1997.
4. Y. Chung S. Choi and V. K. Prasanna, “Parallel Object Recognition on an FPGA-based Configurable Computing Platform”, submitted to *Int. Workshop on Computer Architecture for Machine Perception*, Oct. 1997.
5. L. K. Hua, “Introduction to Number Theory”, New York: Springer-Verlag, 1982.
6. B. New, “Estimating the Performance of XC4000E Adders and Counters”, XILINX Application Note 018, July 1996.
7. H. Park and V. K. Prasanna, “Modular VLSI Architectures for Computing the Arithmetic Fourier Transform”, *IEEE Trans. Signal Processing*, vol. 41, no. 6, pp. 2236-2246, June 1993.
8. R. J. Petersen and B. L. Hutchings, “An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing”, in *Proc. Int. Workshop on Field-Programmable Logic and Applications*, Aug. 1995.
9. I. S. Reed, D. W. Tufts, X. Yu, T. K. Truong, M. Shih and X. Yin, “Fourier analysis and signal processing by use of the Möbius inversion formula”, *IEEE Trans. Acoust., Signal Processing*, vol.38, no. 3, pp. 458-470, Mar. 1990.
10. I. S. Reed, M. T. Shih, T. K. Truong, E. Hendon and D. W. Tufts, “A VLSI architecture for simplified arithmetic Fourier transform algorithm”, in *Proc. Int. Conf. Application Specific Array Processor*, 1990.
11. R. Wilson, “Reprogrammable FPGA-based techniques provide prototyping aid”, *Electronic Engineering Times*, Mar. 11, 1996.
12. XILINX DSP Application notes, “The Fastest FFT in the West”, <http://www.xilinx.com/apps/displit.htm>