Bulletin of the University of Electro-Communications, 18-1·2, pp. 39-45 (2006) [Regular Paper]

An Optimum Design of FFT Multi-Digit Multiplier and Its VLSI Implementation

Syunji Yazaki, Kôki Abe

Abstract

We designed a VLSI chip of FFT multiplier based on simple Cooly-Tukey FFT using a floating-point representation with optimal data length based on an experimental error analysis. The VLSI implementation using HITACHI CMOS 0.18 μm technology can perform multiplication of 2^5 to 2^{13} digit hexadecimal numbers 19.7 to 34.3 times (25.7 times in average) faster than software FFT multiplier at an area cost of 9.05mm^2 . The hardware FFT multiplier is 35.7 times faster than the software FFT multiplier for multiplication of 2^{21} digit hexadecimal numbers. Advantage of hardware FFT multiplier over software will increase when more sophisticated FFT architectures are applied to the multiplier.

Keyword: FFT, multi-digit multiplier, VLSI, error analysis

1 Introduction

Multi-digit arithmetic is used by various applications in recent years, including numerical calculation [1], high-performance primality testing [2], chaos arithmetic [3], and cryptography. Several methods for fast multi-digit multiplication have been proposed, such as Karatsuba method [4], Toom-Cook method, and FFT multiplication, the computational complexities of which are given by $O(m^{1.585})$, $O(m^{1.465})$, and $O(m \cdot \log m \cdot \log\log m)$,respectively, where m stands for number of digits. Here we focus our attention to the FFT multiplication.

Hardware implementation of multi-digit multiplication algorithms is effective for realizing high performance arithmetic. However, VLSI implementation of Karatsuba method, for example, could only be used for multi-digit multiplication. On the other hand, VLSI implementation of FFT multiplication can be realized by using FFT hardware [5, 6] as its main component which is used in many application areas as DSP. This implies that hardware FFT multiplier can easily be retargeted to many applications.

In this paper we describe an optimum VLSI design of FFT multiplier with high performance at minimum cost. For this purpose we perform an experimental error analysis to obtain a data representation with minimum bit length. We evaluate the design in terms of performance and area cost by comparing with software implementations of multi-digit multiplication.

This paper differs from previous paper [7] in several points: details of hardware modules, discussions on using existing floating-point multipliers, and considerations of applying FFT processors to multiplication are presented. In addition, comparisons with software were made by using faster processor.

Section 2 briefly explains FFT multiplication algorithms. Section 3 presents a hardware structure for FFT multiplication. Section 4 describes an optimization of the hardware FFT multiplier based on an experimental error analysis. In Section 5 results are presented and discussions are given. We present some concluding remarks in Section 6.

2 FFT Multiplication Algorithm

Let r be the radix, and m be the number of digits such that $2m = 2^k$ for some nonnegative integer k. We represent two integers $a = (a_{m-1} \cdots a_0)_r$ and $b = (b_{m-1} \cdots b_0)_r$ by $2m$ dimension complex vectors

Received on October 19, 2005
Department of Computer Science

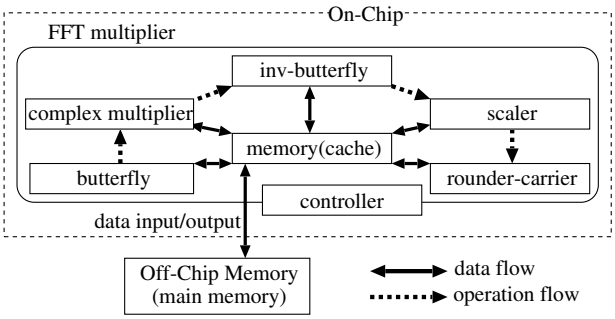

Figure 1: Structure of FFT multiplier.

with the imaginary parts of all zeros, $u = (0 \ldots 0a_{m-1} \ldots a_0)_r$ and $v = (0 \ldots 0b_{m-1} \ldots b_0)_r$. Let the product of u and v be $h = u \cdot v = (c_{2m-1} \ldots c_0)_r$, $F(\cdot)$ be an FFT, and $F^{-1}(\cdot)$ be an inverse FFT. Then we have

$$h = F^{-1}(F(u) \otimes F(v))/2m, \qquad (1)$$

where $\otimes$ denotes multiplying two arrays to obtain an array each element of which is the product of the elements with the same index. In addition, we need to round each digit to the nearest integer, adjust it to be within the range of radix r, determine the carry received from the lower digit, and send the carry to the upper digit if necessary.

3 Design of FFT Multiplier

Fig. 1 outlines the structure of the FFT multiplier we designed. The solid and broken lines mean the data and operation flows, respectively. We employ floating-point as data representation. All data-path modules share a memory. Each module repeats fetching data from the memory, calculating them, and writing back the results. The following subsections explain the circuit construction of each module.

3.1 complex multiplier

When two complex numbers, $s_r + s_i i$ and $t_r + t_i i$ are given, this module performs their multiplication based on Eq. (2).

$$\begin{aligned} p_r + p_i i &= (s_r + s_i i)(t_r + t_i i) \\ &= s_r t_r \,.\, s_i t_i + ((s_r + s_i)(t_r + t_i) - (s_r t_r + s_i t_i))i. \end{aligned} \qquad (2)$$

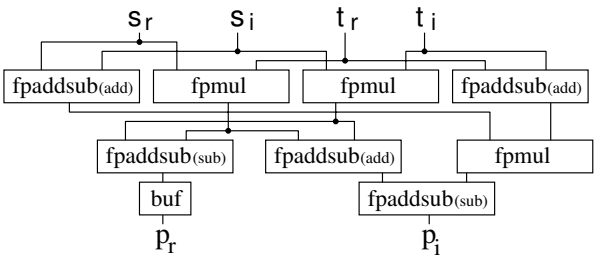

Figure 2: Structure of complex multiplier.

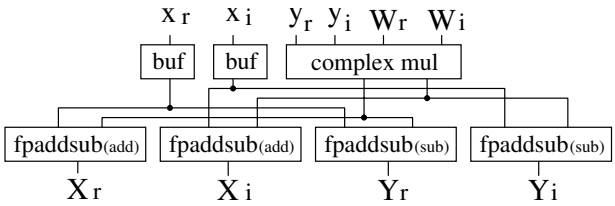

Figure 3: Structure of butterfly module.

Because the cost of adder is lower than that of multiplier, the number of multiplications is decreased at a sacriflce of increasing the number of add-subtract operations according to Eq. (2). The resulting module structure is shown in Fig. 2, where fpmul and fpaddsub submodules perform floating-point multiplication and add-subtract, respectively. The buffer module is used for synchronization.

3.2 butterfly and inv-butterfly modules

The butterfly and inv-butterfly modules are used for FFT forward and inverse transforms, respectively. Their structures depend on the type of FFT algorithm being used. Here we employ the Cooley-Tukey algorithm [8] for simplicity.

The butterfly operation for Cooley-Tukey FFT is given by $X = x + yW$, $Y = x - yW$, where W represents one of the primitive $2m$-th roots of unity. The module structure is shown in Fig. 3.

The inverse butterfly operation is given by $X' = x' + y'$, $Y' = (x' - y')W$. The module is structured similarly and is shown in Fig. 4.

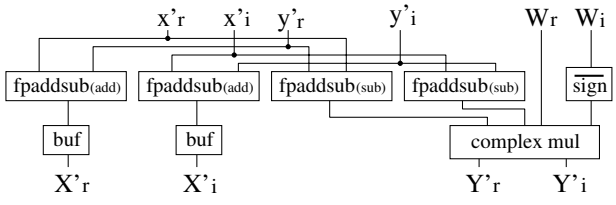

Figure 4: Structure of inv-butterfly module.

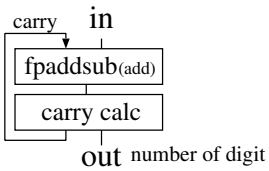

Figure 5: Structure of rounder-carrier module.

3.3 scaler module

After the inverse butterfly, each digit is divided by $2m$. For this operation only the exponent part of the result needs to be subtracted by k because $2m = 2^k$, where k is an integer. This operation is called scaling. Due to the symmetric characteristics of FFT, the values fed into the scaler should be real numbers, although finite small errors would appear in their imaginary parts which are allowed to be neglected. The scaler module is thus comprised by a subtracter of the exponent bit length.

3.4 rounder-carrier module

This module rounds each digit of a real number to the nearest integer and adjusts it within the range of radix r. The rounding is correct only if the absolute error is within ± 0.5.

Fig. 5 shows the structure of the rounder-carrier module. The module consists of fpaddsub and carry calc submodules. The fpaddsub adds the digit with a carry from the lower digit, rounds the results, and determines the carry to the upper digit. The carry calc submodule consists of a multiplexer and an adder for rounding.

3.5 memory module

The data storage area required for multi-digit arithmetics is very large. If the whole storage were implemented on a single chip, the area would become huge. As a solution to this, we store the whole data in an external large-scale off-chip memory device and carry out the arithmetic operations in a cache. The memory module in Fig. 1 behaves as a cache which stores only a part of the data. To avoid structural hazard caused by pipelining, the cache should have two data ports each for reading and writing the data. The values of W are calculated and written in a look up table in advance, which is also included in this memory module.

3.6 controller

This module controls the data-path of the multiplier by sending control signals to submodules at appropriate times.

4 Optimum Data Representation

We employ floating-point as the data representation. Because floating-point arithmetic circuits tend to be very large we minimize the bit length of floating-point data representation required for performing correct FFT multiplication.

The minimum length of the exponent part can simply be determined by making sure that overflow does not occur during the operation. On the other hand, the minimum length of the fraction part must be determined based on the required minimum precision which is obtained through error analysis. An error analysis of FFT multiplication was performed by Henrici [9] in the past. He showed that to obtain correct products through FFT multiplication, the following inequality must be satisfled:

$$\varepsilon_M \leq 1/192m^{2}(2\log_2 m + 7)r^{2}, \qquad (3)$$

where r, m, and ε_M are the radix, the number of digits, and the machine epsilon, respectively.

On the other hand, an observation indicated that Henrici's analysis was based on a worst case scenario and the condition could be more relaxed in actual situation [10]. The observation implies that the maximum error occurs when multiplying the maximum values represented by m digits. In the following an experiment to verify the observation is described. The radix $r = 16$ is used hereafter.

We measured errors produced by FFT multiplication $u = (\overbrace{0 \ldots 0}^{m}\ \overbrace{a \ldots a}^{m})_{16}$ and $v = (\overbrace{0 \ldots 0}^{m}\ \overbrace{b \ldots b}^{m})_{16}$, when varying a and b from 0 to F. The Cooley-Tukey algorithm and 64-bit floating-point data representation such as IEEE754 were used in the experiment. The result is shown in Fig. 6 for $m = 1024$ and 2048, where the vertical axis represents absolute errors associated with $u \cdot v$. It was thus verified that the maximum error occurs when multiplying the maximum values represented by m digits. A similar result was obtained for $m = 4096$ and 8192.

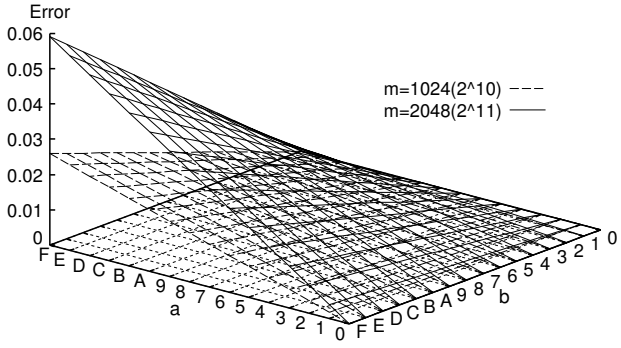

Figure 6: Errors in multiplying $(0^m a^m)_{16}$ and $(0^m b^m)_{16}$ using FFT.

Based on the above experimental error analysis, we defined a floating-point representation with variable-length exponent and fraction parts, and carried out a numerical calculation to obtain the least bit lengths required for performing correct FFT multiplication. The result is shown in Fig. 7, where horizontal axis represents the base 2 logarithm of number of digits m, and the points marked by + and × represent the least length of the exponent and fraction parts, respectively. The solid line represents a linear function best fit to the experimental data by the least-mean square method. Note that the fraction length (the broken line in the figure) obtained from the inequality (3) is much longer than the experimentally obtained one. Also note Fig. 7 implies that if 64-bit floating-point data representation were used, FFT multiplication of up to 2^{30} digits could be executed.

5 Implementation Results and Discussions

We designed an FFT multiplier using an optimal data length based on the results obtained in the previous section, and evaluated the area and performance of the VLSI implementation. We described the design in Verilog-HDL and performed a logic synthesis of the descriptions using Design Compiler (Synopsis Inc.,ver. 2003.6-SP1). For the synthesis we used the CMOS 0.18 μm technology cell library developed by VDEC (VLSI Design and Education Center) [11] based on Hitachi's specifIcations. We gave a delay-optimization restriction to the Design Compiler.

We evaluated the FFT multiplier with respect to the total area and critical path delay of the synthesized logic for $m = 2^5$ to 2^{13} . The controller's area was excluded in the evaluation because it did not depend on bit length of the data representation and in fact was very small compared to other modules. For the memory module (cache) we allocated 8 entries each of which can hold data used by one butterfly operation. The result is shown in Fig. 8, where the right and left vertical axes represent the total area and critical path delay of the synthesized logic, respectively. For $m = 2^{13}$, the multiplier with the optimum data representation (sign bit, 7-bit exponent, and 27-bit fraction) was found to be realized with the total area of 6.55mm^2 and critical path delay of 7.63ns. We also evaluated FFT multiplier with 64-bit data representation (sign bit, exponent:11-bit, fraction:52-bit), and found that total area and critical path delay were 16.0mm^2 and 10.3ns, respectively. Thus for $m = 2^{13}$ we can reduce the total

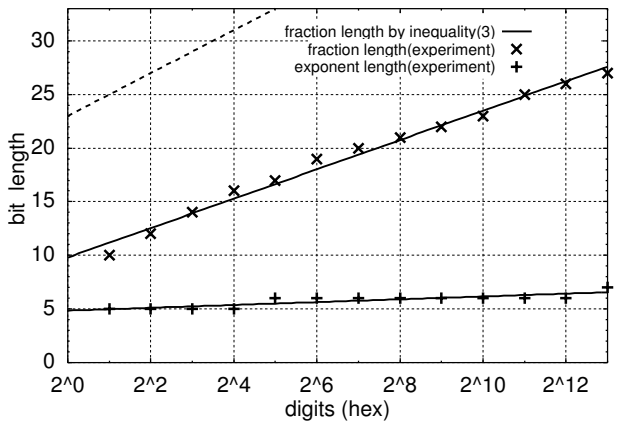

Figure 7: Fraction and exponent bit lengths required for multiplying two multi-digit numbers vs. the number of digits (r = 16).

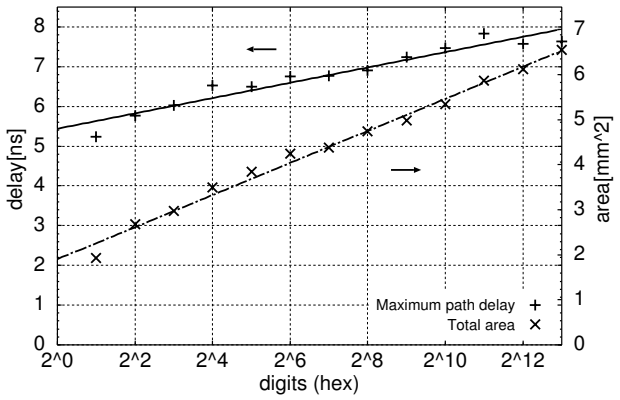

Figure 8: Critical path delay and total area of FFT multiplier vs. number of digits (r = 16).

Table 1: Area and critical path delay of the optimally pipelined FFT multiplier ($m = 2^{13}$).

module	area[mm^2]	delay[ns]
complex mul	2.01	1.89
butterfly	2.98	1.81
inv-butterfly	3.06	1.81
scaler	0.02	1.26
rounder-carrier	0.38	1.74
memory	0.60	1.60
total	9.05	1.89

area by 59% and critical path delay by 26% using the optimum data representation.

We can replace floating-point multipliers used in the described FFT multiplier by alternatives [12, 13, 14, 15]. In that case, using an optimum data length based on the proposed error analysis will also lead to reducing the total area and critical path delay.

The FFT multiplier described so far was not optimally pipelined. An optimum version of pipelined FFT multiplier for $m = 2^{13}$ was designed as follows. Firstly, the critical path which exists in Wallace tree multiplier [16] was divided into several stages considering the area and performance trade-off. After that, the amounts of critical paths among pipeline stages was made as even as possible. The implementation result is summarized in Table 1. The maximum path delay was thus reduced to 1.89ns with the area cost of 9.05mm^2 .

The pipeline latencies of the resulted modules are summarized in Table 2, where $T_{\text{butterfly}}$, T_{cmul}, T_{scl}, and T_{rc} represent pipeline latencies of (inv-) butterfly, complex multiplier, scaler, and rounder-carrier modules, respectively. Number of clocks required for an FFT multiplication is given by

$$T_{\text{fftmul}} = 3(T_{\text{butterfly}} + m - 1)\log_2 m + (2T_{\text{rc}} + 7)m + (3T_{\text{butterfly}} + T_{\text{cmul}} + T_{\text{scl}} - T_{\text{rc}} - 3). \quad (4)$$

Requiring 541,563 clocks for an FFT multiplication of 2^{13} digit numbers, the execution time is estimated to be 1.02ms (541,563×1.89ns).

Table 2: Latency of FFT multiplier modules.

	$T_{\text{butterfly}}$	T_{cmul}	T_{scl}	T_{rc}
Latency	22	17	2	10

Table 3: Comparison of FFT multiplication performance of software and hardware implementations.

Digits	Soft[ms]	Hard[ms]	Soft/Hard
2^5	0.0917	0.0033	27.8
2^6	0.1242	0.0063	19.7
2^7	0.3924	0.0126	31.1
2^8	0.8854	0.0258	34.3
2^9	1.4930	0.0535	27.9
2^{10}	2.6840	0.1116	24.1
2^{11}	5.2400	0.2337	22.4
2^{12}	10.8000	0.4893	22.1
2^{13}	22.6200	1.0236	22.1

We compared the performance of hardware and software implementations of FFT multiplier for $m \leq 2^{13}$. We used FFTW 3.0.1 for software FFT, Pentium 4 1.7GHz CPU with the same design rule as our hardware implementation, FreeBSD 5.4 OS, and gcc 3.4.2 compiler. (We refer to the software implementation as FFTW multiplier hereafter.) The maximum path delay employed for the comparison was 1.89ns for all digits. The result is shown in Table 3.

The performance of the hardware FFT multiplier was found to be 19.7 to 34.3 (25.7 in average) times better than FFTW multiplier. The reason that the performance ratio varies depending on digits is that FFTW is tuned to operate best for 1,000 to 10,000 samples [17].

We also compared the performance of FFTW multiplier and a software implementation of Karatsuba method. We used a multi-digit arithmetic software library called exflib [18] which implements Karatsuba method. The result is that FFTW multiplier was faster than exflib for $m \leq 2^{21}$. At $m = 2^{21}$ the execution time of FFTW multiplier was 13.9s.

Here we consider hardware FFT multiplier at $m = 2^{21}$. By extrapolating the linear functions in Fig. 7 to $m = 2^{21}$ we have 9 and 40 bits for the minimum exponent and fraction parts, respectively, taking the deviation into account. The pipelined version of FFT multiplier with the optimum data representation obtained as above yields a critical path delay of 1.96ns, leading to 0.39s FFT multiplication at an area cost of 16.1mm^2. From the result, it is found that hardware FFT multiplier is 35.7 times faster than FFTW multiplier at $m = 2^{21}$.

Hardware implementation of Karatsuba method would be an alternative for $m \geq 2^{21}$. However,

Karatsuba method applies only to multi-digit multiplication. A multi-digit multiplier using FFT hardware has an advantage that the FFT module can be utilized for various applications.

In this paper, we designed a simple FFT hardware of Cooly-Tukey FFT. However, there are many FFT processors implemented so far which have high performance at low cost by applying following features: a memory architecture to avoid memory conflict in mixed-radix (MR) algorithm [5], multipath pipeline architecture to achieve high-throughput rate [6], and so on. By applying these features, we could design an FFT multiplier with high performance. However, these processors employ data representations for DSP use. For example, FFT processors described in [5] and [6] employ 16-bit block floating-point and 10bit flxed-point data representations, respectively. The data representations limit FFT multiplication to small digits. Thus, these FFT processors in the original can not be applied to multiplying large-digit integers. They could be applied to multiplication of large-digit integers if the accuracy required by FFT multiplication is guaranteed by using the proposed error analysis.

6 Conclusion

We described a VLSI design of FFT multiplier with floating-point number representation the bit length of which was minimized based on an experimental error analysis. Using HITACHI CMOS 0.18 μm technology cell library for logic synthesis the performance of the hardware FFT multiplier based on simple Cooly-Tukey FFT was found to be 19.7 to 34.3 (25.7 in average) times better than software FFTW multiplier when multiplying m-digit hexadecimal numbers, $m \leq 2^{13}$, with an area cost of 9.05mm^2 . The hardware FFT multiplier was 35.7 times faster than software FFTW multiplier of 2^{21} digit hexadecimal numbers where FFTW multiplier exceeds Karatsuba method in performance, with an area cost of 16.1mm^2 . A multi-digit multiplier using FFT hardware has an advantage over hardware Karatsuba multiplier in that the FFT component can be utilized for various applications. Advantage of hardware FFT multiplier over software will increase when more sophisticated FFT architectures are applied to the multiplier. The quantitative evaluation belongs to future work.

7 Acknowledgements

The authors would like to thank Profs. T. Kako and N.Yamamoto of University of Electro-Communications for their valuable suggestions and discussions, and Dr. N. Tsukie of Tokyo University of Technology for his helpful suggestions. They would also like to thank the anonymous reviewer for his/her helpful comments to improve the quality of the manuscript. This research was conducted in collaboration with Synopsys Inc. and Cadence Design System Inc. through the VLSI Design and Education Center, University of Tokyo. It was partially supported by JSPS Grant-in-Aid for Scientiflc Research (C)(2) 1650026.

References

[1] H.Fujiwara, “High-Accurate Numerical Method for Integral Equations of the First Kind under Multiple-Precision Arithmetic,” *Theoretical and Applied Mechanics Japan*,Vol. 52, pp.193-203, 2003.

[2] M. Agrawal, N. Kayal, and N. Saxena, “PRIMES Is in P,” http://www.cse.iitk.ac.in/, 2002.

[3] J. C. Sprott, “Chaos and Time-series Analysis,” *Oxford University Press*, 2003.

[4] A. Karatsuba and Y. Ofman “Multiplication of Multidigit Numbers on Automata,” *Sov. Phys. Dokl.*, Vol.7, pp.595-596, 1963.

[5] B. G. Jo and M. H. Sunwoo, “New Continuous-Flow Mixed-Radix (CFMR) FFT Processor Using Novel In-Place Strategy,” *IEEE Trans. on Circuits and Systems-I: Regular Papers*,Vol.52, No.5, pp.911-919, May 2005.

[6] Y. Lin, H. Liu, and C. Lee, “A 1-GS/s FFT/IFFT Processor for UWB Applications,” *IEEE Journal of Solid-State Circuits*, Vol.40, No.8, pp.1726-1735, Aug. 2005.

[7] S. Yazaki and K. Abe, “VLSI Design of FFT Multi-digit Multiplier,” *Transactions of the Japan Society for Industrial and Applied Mathematics*, Vol.15, No.3, pp.385-401, Sep. 2005. (in Japanese)

[8] J. W. Cooley and J. W. Tukey, “An Algorithm for Machine Computation of the Complex Fourier Series,” *Mathematics of Computation*, Vol.19, pp.297-301, Apr. 1965.

[9] P. Henrici, *Applied and Computational Complex Analysis*, Vol.3, Chap.13, John Wiley &Sons, NY, 1986.

[10] H. Hirayama, “Multiplication of High Precision Numbers by FFT,” *IPSJ SIG Technical Reports (High Performance Computing)*, Vol.65, No.7, pp.27-32, 1997. (in Japanese)

[11] VLSI Design and Education Center Homepage,

http://www.vdec.u-tokyo.ac.jp

[12] H. Kim and T. Strong, "A Complementary GaAs 53-bit Parallel Array Floating Point Multiplier," University of Michigan, Dec. 1995.

[13] J. E. Stine and M. J. Schulte, "A Combined Interval and Floating Point Multiplier," *Proc. of 8th Great Lakes Symposium on VLSI*, Lafayette, LA, pp.208-213, Feb. 1998.

[14] K. C. Bickersta., E. E. Swartzlander Jr., and M. J. Schulte, "Analysis of Column Compression Multipliers," *Proc. of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, IEEE Computer Society Press, pp.33-39, Jun. 2001.

[15] K. E. Wires, M. J. Schulte, and J. E. Stine, "Combined IEEE Compliant and Truncated Floating Point Multipliers for Reduced Power Dissipation," *Proc. of the International Conference on Computer Design*, Austin, TX, IEEE Computer Society Press, pp.497-500,Sep. 2001.

[16] C. S. Wallace, "A Suggestion for a Fast Multipliers," *IEEE Trans. Electronic Computers*, Vol.EC-13, No.2, pp.14-17, Feb. 1964.

[17] benchFFT–FFT Benchmark Results, http://www.fftw.org/speed/

[18] exflib–Extended Precision Float-Point Arithmetic Library, http://www-an.acs.i.kyotou.ac.jp/fujiwara/exflib/exflib-index.html