# A low logic depth complex multiplier

Berkeman, Anders; Öwall, Viktor; Torkelson, Mats

1998

# A Low Logic Depth Complex Multiplier

Anders Berkeman, Viktor Öwall and Mats Torkelson

*Department of Applied Electronics, Lund University*
*Box 118, SE-221 00 Lund, Sweden*
*Tel. +46 46 2229377, Fax. +46 46 129948*
*E-mail: Anders.Berkeman@tde.lth.se*

## Abstract

*A complex multiplier has been designed for use in a pipelined fast fourier transform processor. The performance in terms of throughput of the processor is limited by the multiplication. Therefore, the multiplier is optimized to make the input to output delay as short as possible. A new architecture based on distributed arithmetic and Wallace-trees has been developed and is compared to a previous multiplier realized as a regular distributed arithmetic array. The simulated gain in speed for the presented multiplier is about 100%. For verification, the multiplier is fabricated in a three metal-layer* $0.5\mu$ *CMOS process using a standard cell library. The fabricated multiplier chip has been functionally verified.*

## 1. Introduction

A pipelined Fast Fourier Transform (FFT) processor has been designed for use in an Orthogonal Frequency Division Multiplex (OFDM) system. Multiplication is often the most time-critical and area consuming operation in a digital signal processor. Therefore, effort has to be made to decrease the number of multipliers and to increase their speed. In the designed FFT processor the critical path consists of a complex multiplier in series with a butterfly unit performing addition and subtraction. A part of the FFT pipeline is shown in figure 1. Since the butterfly processors are much faster than the complex multiplier, the maximum clock frequency of the processor strongly depends of the multiplier delay.

This paper present a novel multiplier architecture based on distributed arithmetic and Wallace trees. The new architecture is by simulation compared to a multiplier realized as a regular array, and the speed improvement is approximately 100%. The multiplier is fully parameterized, so any configuration of input and output wordlengths could be elaborated. The tree multiplier has been fabricated and verified for functionality. However, no performance measurements have been made.

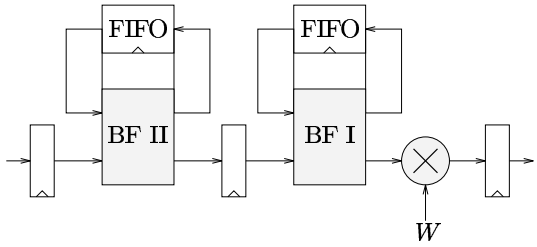


Figure 1: Part of the FFT processor pipeline. The butterfly processors are named "BF I" and "BF II". Shaded boxes are combinatorial blocks without pipeline.

## 2. The FFT processor

In the early versions of the FFT-processor, a complex array multiplier was used [1]. The array multiplier is a highly regular structure resulting in a minimal wire-length, which is important for high-speed design in sub-micron processes where wiring delay gives a significant contribution to the overall delay. However, in a process where cell delay dominates wire delay, the logic depth of the design is more important than regularity. In the complex array multiplier the logic depth is proportional to the input wordlength $N$. In the adder tree multiplier, on the other hand, the depth is proportional to $\log N$ [2]. Even for short worldlengths this leads to a substantial reduction in delay.

A way to decrease the critical path of the FFT processor would be to pipeline the multiplier into two or more stages. However, due to the pipelined structure of the FFT processor, complexity of the controlling hardware would increase [3]. Furthermore, the wordlengths of the data paths are wide, due to the application of the processor, and all operators use complex arithmetic. A multiplier in this application has between 44 and 52 inputs, and a pipeline register inserted somewhere in the middle of the multiplier would need a word length of more than a hundred bits, due to the internal 'carry save' number representation. This would increase area, routing and clock load and is not a preferable solution. Instead, the multiply operation is entirely combinatorial.

The FFT processor is implemented using the R2$^2$DIF FFT-algorithm [3]. In this algorithm, every second multiplication can be exchanged to a multiply by $-j$, which for an 8192-point FFT leaves only six complex multipliers. This is to be compared to thirteen using a straightforward implementation. The multiplication by $-j$ is realized without a multiply by real-imaginary swap and negation of the imaginary part. This is the reason for the two different butterfly processors, "BF I" and "BF II", in figure 1. By using this algorithm, the number of instanciated multipliers is minimized compared to an ordinary radix-2 FFT without any loss in throughput.

## 3. Multiplier algorithm

A complex multiplier calculates two inner products,

$$\begin{cases} Z_R = A_R W_R - A_I W_I \\ Z_I = A_R W_I + A_I W_R. \end{cases} \quad (1)$$

In the case of the FFT-processor, $W = W_R + jW_I$ are the twiddle-factors stored in a ROM. The wordlength of $W_R$ and $W_I$ is denoted $M$. According to equation (1), four real multiplications and two additions are required.

With the exception of logic minimization, there are two methods to decrease multiplication delay if it is assumed that multiplication is performed by summation of partial products. The first is to reduce the number of partial products, and the second is to use a faster adder strategy to sum all the partial products together [2]. Both methods have been combined in the presented architecture.

Distributed arithmetic [4] was chosen as a means to reduce the number of partial products, and a Wallace tree adder was selected for adding the partial products together. By using distributed arithmetic, the complex multiplication is treated as two independent inner products $Z_R$ and $Z_I$. Each of the inner products will be calculated using one distributed arithmetic multiplier, as explained in section 3. This should be compared to a multiplier realized using equation (1), in which case four real multiplications are required.

As an alternative to distributed arithmetic, modified Booth-encoding was considered. However, as the number of partial products are about the same for both methods, modified Booth-encoding requires more logic gates to implement. This is due to that in the modified Booth algorithm, three variables have to be decoded to select the proper partial product. In a complex multiplier based on distributed arithmetic, a simple two-input xor-gate does the selection.

When using distributed arithmetic, the twiddle-factors have to be transformed from $W_R$ and $W_I$ to $W_S$ and $W_D$, where

$$\begin{cases} W_S = W_R + W_I \\ W_D = W_R - W_I. \end{cases} \quad (2)$$

This transformation does not cause any problems in the implementation, since the twiddle-factors are pre-calculated in the $W_S$ and $W_D$ format before realization. However, it is important that $W_S$ and $W_D$ are calculated using floating-point arithmetic before they are converted to fixed point. Otherwise, accuracy is reduced.

## 4. Mathematical background

This section gives a mathematical background to the operation of the multiplier. In the equations that follow, a bit-variable is treated as a variable holding the arithmetic value of 0 or 1. In this way, bits can be used together with arithmetic variables and operators. If A is an $N$-bit fractional number in two's complement, the value of A is calculated according to

$$A = -a_0 + \sum_{i=1}^{N-1} a_i 2^{-i}. \quad (3)$$

By using the identity

$$A = \frac{1}{2}[A - (-A)] \quad (4)$$

and the rule for negating a two's complement number

$$-A = \overline{A} + 2^{-(N-1)}, \quad (5)$$

equation (3) can be written as

$$A = -(a_0 - \overline{a_0})\, 2^{-1} + \sum_{i=1}^{N-1} (a_i - \overline{a_i})\, 2^{-i-1} - 2^{-N}. \quad (6)$$

Introduce $\alpha_0 = (\overline{a_0} - a_0)$, and for $k \neq 0$, $\alpha_k = (a_k - \overline{a_k})$, note that all $\alpha_k \in \{-1, +1\}$. Using this notation, $A$ can be written as

$$A = A' - 2^{-N}, \quad (7)$$

where

$$A' = \sum_{i=0}^{N-1} \alpha_i 2^{-i-1}. \quad (8)$$

The relationship between $a_i$ and $\alpha_i$ is

$$\alpha_i = \begin{cases} +1 \text{ , if } a_{i\neq 0} = 1 \text{ or } a_0 = 0 \\ -1 \text{ , if } a_{i\neq 0} = 0 \text{ or } a_0 = 1 \end{cases} \quad (9)$$

Using this encoding the complex product can be written as

$$Z_R = \sum_{i=0}^{N-1} (W_R \alpha_{Ri} - W_I \alpha_{Ii})\, 2^{-i-1} - (W_R - W_I)\, 2^{-N} \quad (10)$$

$$Z_I = \sum_{i=0}^{N-1} (W_I \alpha_{Ri} + W_R \alpha_{Ii})\, 2^{-i-1} - (W_I + W_R)\, 2^{-N}. \quad (11)$$

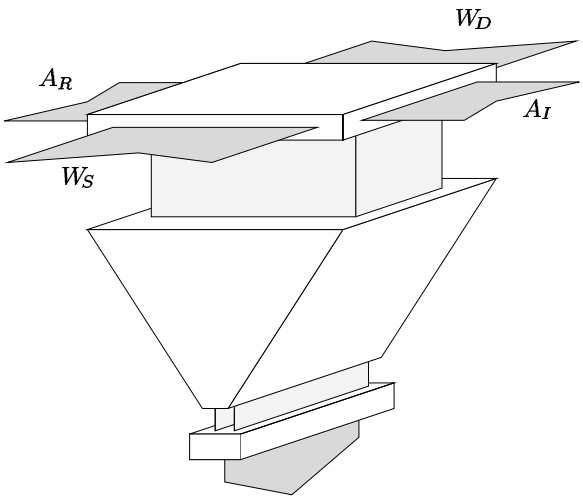


Figure 2: The multiplier for $Z_R$ or $Z_I$, the complete complex multiplier consists of two of these. Partial inner product generator at the top, adder tree in the middle, and fast carry-lookahead adder at the bottom.

The expression $W_I\alpha_{Ri} + W_R\alpha_{Ii}$ is, for $i \neq 0$, examined in the following table,

| $\alpha_{Ri}$ | $\alpha_{Ii}$ | $a_{Ri}$ | $a_{Ii}$ | $W_I\alpha_{Ri} + W_R\alpha_{Ii}$ |
|---|---|---|---|---|
| $-1$ | $-1$ | 0 | 0 | $-W_S$ |
| $-1$ | 1 | 0 | 1 | $W_D$ |
| 1 | $-1$ | 1 | 0 | $-W_D$ |
| 1 | 1 | 1 | 1 | $W_S$ |

where $W_S$ and $W_D$ were introduced in equation (2). From the table it is clear that $p = (a_{Ri} \oplus a_{Ii})$ can be used to select $W_S$ or $W_D$. Using $p$, $W_S$ and $W_D$, equation (10) and (11) can be written as

$$Z_R = \sum_{i=0}^{N-1} (-1)^{\overline{a_{Ii}}} [pW_S \vee \overline{p}W_D]\, 2^{-i-1} - W_D\, 2^{-N} =$$

$$\sum_{i=0}^{N-1} (\overline{a_{Ii}} \oplus [pW_S \vee \overline{p}W_D] + \overline{a_{Ii}})\, 2^{-i-1} - W_D\, 2^{-N} \quad (12)$$

$$Z_I = \sum_{i=0}^{N-1} (-1)^{\overline{a_{Ri}}} [pW_D \vee \overline{p}W_S]\, 2^{-i-1} - W_S\, 2^{-N} =$$

$$\sum_{i=0}^{N-1} (\overline{a_{Ri}} \oplus [pW_D \vee \overline{p}W_S] + \overline{a_{Ri}})\, 2^{-i-1} - W_S\, 2^{-N}. \quad (13)$$

When evaluating the sums, the powers $\overline{a_{Ri}}$ and $\overline{a_{Ii}}$ should be replaced with $a_{Ri}$ and $a_{Ii}$ for the case $i = 0$, since these bits represent the sign in two's complement representation. The partial inner product

$$\overline{a_{Ri}} \oplus [pW_D \vee \overline{p}W_S] + \overline{a_{Ri}} \quad (14)$$

is suitable for hardware mapping. It is realized as a multiplexer selecting $\pm W_S$ or $\pm W_D$, depending on the value of $p = (a_{Ri} \oplus a_{Ii})$. If $a_{Ri\neq 0} = 0$ (or $a_{R0} = 1$), an inverted version of the coefficients is chosen,

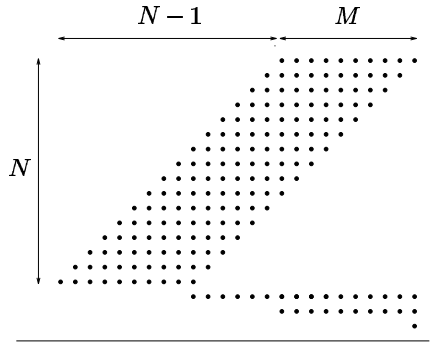


Figure 3: All partial product bits by significance for $Z_R$ or $Z_I$. Input wordlength is $N$ and coefficient wordlength is $M$.

and a '1' in the least significant position is added, corresponding to a two's complement negation. The expression

$$\overline{a_{Ii}} \oplus [pW_S \vee \overline{p}W_D] + \overline{a_{Ii}} \quad (15)$$

is treated similarly. Figure 3 shows all the partial product bits that has to be added to generate $Z_R$ or $Z_I$. The wordlength for the twiddle factor, $W$, is $M$ bits and for the data, $A$, it is $N$ bits, in this case 10 and 16 bits respectively. The top sixteen lines in the figure is the partial products generated inside the sum of equation (12) or (13), and the third line from bottom is the ones that form the corresponding two's complement of these products. The last two lines is the $-W_{S|D}2^{-N}$ term.

## 5. Implementation

The proposed multiplier consists of two distributed arithmetic blocks, one calculating $Z_R$, and the other $Z_I$. The two blocks are similar and the difference is basically the sign in equation (1). Each block is divided into three parts, partial inner product generator, adder tree and carry lookahead adder, see figure 2.

The multiplier is synthesized to a $0.5\mu$ cell library that does not contain any dedicated half or full adder cells. Estimated delay for a 10+10 by 16+16 multiplier using a worst case industrial environment is about 16 nanoseconds, compared to 34 nanoseconds for the array multiplier using the same cell libraries and comparable design methodology. About 55% of the total delay is due to the adder tree. The partial inner-product generator contributes with 20% and the carry-lookahead adder 25% of the total delay. Most of the delay is spent in the adder tree, and by using dedicated adder cells this delay can be decreased. However, the target cell library does not contain any such cells and such improvements have not been implemented, which is the case for both designs.

When designing the adder tree, a generic tree generator was used. This generator produces a tree with $y$ inputs of wordlength $x$, that is a rectangle of $x$ by $y$ input bits. This rectangle has to be large enough to cover all the partial product bits of figure 3, i.e. $x = M+N-1$ and $y = N+3$. For certain sizes of $N$ and $M$, the two last lines in figure 3 can be joined with two of the $N$ first lines, minimizing $y$ to $N+1$. Unfortunately, almost 50% of the inputs to the adder tree are unused or used for sign extension, and extra logic will be generated. Therefore, the area for the tree multiplier is approximately 75% larger than for the array multiplier. The number of gates for the array multiplier is 3000, while the tree multiplier uses 6200 gates, of which 4400 belongs to the two adder trees. Theoretically, the area for a dedicated tree generator should be only slightly larger than for the array multiplier. Both multipliers have been fabricated and the die photo for the tree multiplier is shown in figure 4.

When data flows through the pipeline of the FFT processor, the wordlength has to increase to keep accuracy in the calculations. For the current application the input wordlength is 12+12 bits (real + imaginary) and the output wordlength is 16+16 bits. The twiddle-factors are kept constant at 10+10 bits at all stages of the pipeline. Different wordlengths in the datapath means that a set of multipliers of different wordlengths have to be instantiated if the longest wordlength is not to be used for all multipliers with a corresponding increase in area. Also, as FFT processors will be built for different applications the wordlength is subject to change. Therefore, the multiplier is fully parameterized and a multiplier of specific wordlength can be elaborated when needed.

For our application, the output wordlength should equal the input wordlength, i.e. some of the least significant bits of the result are cut away. A simple rounding scheme is applied to lower the distorsion when the output is truncated. A rounding bit is added to the right of the rightmost bit to be kept after truncation, causing a carry to propagate when the most significant position of the bits cut away is a one. A feature of the adder tree is that this bit can be inserted together with the partial inner products at the top of the tree, see figure 3. In the array multiplier, an additional row of half-adders had to be included to handle rounding. As rounding includes addition of a one with the product, arithmetic overflow at the output is possible. Therefore, a saturation unit is placed at the output of the carry-lookahead adder. This unit checks the most significant bits of the result and modifies the output if an overflow has occurred.

## 6. Conclusion

A Wallace-tree based complex multiplier has been designed and simulated with a speed improvement of approximately 100% compared to a previously designed array multiplier. In a worst case industrial environment, the delay of a 10+10 by 16+16 multiplier is about 16 ns. This is when synthesized to a three metal-layer 0.5$\mu$ process with a standard cell library (Mietec MTC35000) that does not contain any dedicated half- or full-adder cells. The figure is an estimation without post-layout delay information. Under equal conditions the complex array multiplier currently being used has a delay of 34 ns.

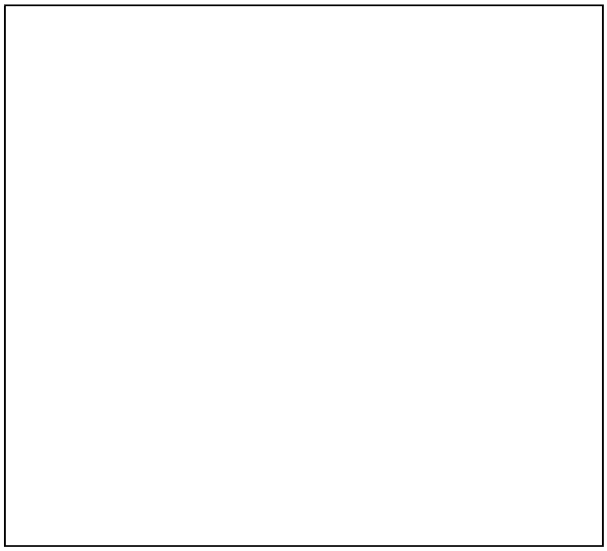

Figure 4: Plots of the tree multiplier. The array multiplier is similar. The pad-frame is 3.2x2.9 mm$^2$ and equal for both designs.

Since the multiplier, together with an adder/subtractor, is located in the critical path of the FFT-processor, throughput is expected to increase with approximately 80%. The multiplier is fully parameterized so any configuration of input and output wordlengths can be elaborated and synthesized. Both the array and the tree multiplier have been fabricated on the same die.

## References


[1] S. He and M. Torkelson. "A Complex Array Multiplier Using Distributed Arithmetic". In *Proc. of IEEE Custom Integrated Circuits Conference*, 1991.

[2] C.S. Wallace. "A Suggestion for a Fast Multiplier". *IEEE Transactions on Electronic Components*, Vol. EC-13, Feb 1964.

[3] S. He and M. Torkelson. "A New Approach to Pipeline FFT Processor". In *Proc. of IEEE International Parallel Processing Symposium*, 1996.

[4] S.G. Smith and P.B. Denyer. "Efficient Bit-Serial Complex Multiplication and Sum-Of Products Computation Using Distributed Arithmetic". In *Proc. of IEEE ICASSP*, 1986.