On the Design of an On-line FFT Network for FPGA's

Robert McIlhenny
rmcilhen@cs.ucla.edu

Miloš D. Ercegovac
milos@cs.ucla.edu

Computer Science Department
University of California
Los Angeles, CA 90024

Abstract

In this paper, a novel implementation is presented for an on-line FFT network, using complex on-line arithmetic, based on adopting a redundant complex number system (RCNS) to represent complex operands as a single number. We present cost comparisons with alternative approaches, to demonstrate a significant improvement in design for FPGA's.

1. Introduction

The Discrete Fourier Transform (DFT) is defined by the equation

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \qquad k = 0, \ldots, N-1 \quad (1)$$

where $W_N = e^{-j(2\pi/N)}$. An efficient scheme for computing the DFT is by a Fast Fourier Transform (FFT) network. The basic module of a FFT network is the *butterfly module*. Given inputs $X_{in} = X_r + jX_i$, $Y_{in} = Y_r + jY_i$, and twiddle factor $W^k = W_r^k + jW_i^k$, the butterfly module, or two-point FFT module, evaluates the complex equations:

$$\begin{aligned} X_{out} &= X_{in} + Y_{in} W^k \\ Y_{out} &= X_{in} - Y_{in} W^k \end{aligned} \quad (2)$$

The implementation of each butterfly module requires one complex multiplication and two complex additions/subtractions. The standard method for implementing complex multiplication is a network of 4 real multiplications and 2 additions. Therefore, a butterfly module consists of 4 real multiplications and 6 real additions/subtractions. This method is employed in [5] and [8]. Since the cost of a multiplication is much more than addition, $O(n^2)$ for multiplication vs. $O(n)$ for addition, an alternative approach for complex multiplication, as presented in [9], consists of 3 real multiplications and 5 real additions. Therefore a butterfly module can be constructed by this approach, consisting of 3 real multiplications and 9 real additions/subtractions.

Since a FFT network is multiply-intensive, and multiply operations consume a significant number of CLB's on an FPGA, both of the more standard approaches may not be suitable for efficient mapping onto FPGAs, especially as the data length increases. Therefore, a more efficient design is needed. Since the multiply operation contains one constant operand (the twiddle factor), several authors have proposed a distributed arithmetic (DA) approach, in which appropriate multiples of the twiddle factor are stored, and referenced by the value of the other variable operand [4][7][8]. This paper adopts this idea, but whereas previous papers assumed a radix 2 system, treating the real and imaginary components separately, this paper proposes a design where the real and imaginary components are treated as a unified number.

Also, assuming single-precision operations, only the most significant half of the product will be used. Using a conventional right-to-left multiplier approach requires producing then discarding the least significant half. Therefore, on-line arithmetic, where for single precision, only the most significant half of the product is produced, will be used in the design.

The proposed design is an extension of the designs presented in [6] for efficient implementations of complex on-line operators, utilizing a redundant complex number system (RCNS) to treat the real and imaginary components as a unified number. This approach serves as an alternative to treating complex operations as a network of real operators, making designs more efficient. This paper extends this approach for the implementation of on-line butterfly modules and an on-line FFT network.

2. Redundant complex number system

In this section, the representation of complex numbers using a redundant complex number system is presented, as

well as a method for converting to and from a non-redundant binary representation.

2.1. Representation

A redundant complex number system (RCNS), as proposed in [1], is a radix rj system, in which digits are in the set $\{-\rho, \cdots, 0, \cdots, \rho\}$, where $r \geq 2$ and $\lceil r^2/2 \rceil \leq \rho \leq r^2 - 1$. This allows a unified representation of the real and imaginary components. Redundancy allows carry-free addition, in which result digits can be produced without the need of carry propagation. It is achieved by using a signed-digit number representation. The redundancy factor K in a RCNS is:

$$K = \frac{\rho}{r^2 - 1} \tag{3}$$

A RCNS with $r = 2$, $\rho = 3$, and hence $K = 1$ is adopted, since conversion from a conventional binary representation is easily peformed. The actual representation of digits will be done using an extension of borrow-save encoding. Using this encoding, the value of a digit x_k=$(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0})$ is: $2x^+_{k,1} - 2x^-_{k,1} + x^+_{k,0} - x^-_{k,0}$.

Assuming fractional operands, the value of a number $X = (x_1, \cdots, x_m)$, is:

$$X = \sum_{k=1}^{m} x_k (2j)^{-k} \tag{4}$$

Alternatively, X can be considered as having radix -4 real and imaginary components, X_r and X_i, respectively, where:

$$X_r = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} x_{2k} (-4)^{-k} \tag{5}$$

$$X_i = (2j) \sum_{k=1}^{\lfloor \frac{n+1}{2} \rfloor} x_{2k-1} (-4)^{-k} \tag{6}$$

Therefore, the complexity of a radix $2j$ operator is equivalent to a radix -4 operator.

2.2. Binary to radix $2j$ conversion

Conversion from a more conventional non-redundant binary representation consisting of a sign bit s and two binary digits: $(y_{k,1}, y_{k,0})$ to a redundant radix $2j$ digit of weight $(2j)^{-k}$ using borrow-save encoding: $(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0})$ has to consider four cases of k:

Case 1: k mod 4=0 (Real, positive weight)

$$(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0}) = (\overline{s}y_{k,1}, s\,y_{k,1}, \overline{s}y_{k,0}, s\,y_{k,0})$$

Case 2: k mod 4=1 (Imaginary, negative weight)

$$(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0}) = (\overline{\overline{s}y_{k-1,0}}, \overline{s\,y_{k-1,0}}, \overline{\overline{s}y_{k,1}}, \overline{s\,y_{k,1}})$$

Case 3: k mod 4=2 (Real, negative weight)

$$(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0}) = (\overline{\overline{s}y_{k,1}}, \overline{s\,y_{k,1}}, \overline{\overline{s}y_{k,0}}, \overline{s\,y_{k,0}})$$

Case 4: k mod 4=3 (Imaginary, positive weight)

$$(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0}) = (\overline{s}y_{k-1,0}, s\,y_{k-1,0}, \overline{s}y_{k,1}, s\,y_{k,1})$$

2.3. Radix $2j$ to binary conversion

Given redundant serial digits x_k of weight $(2j)^{-k}$, conversion to non-redundant real and imgainary vectors, Y_r and Y_i respectively, requires the storage of temporary results, $A_{r,k}$, $A_{i,k}$, $B_{r,k}$ and $B_{i,k}$, where assuming m-bit precision, $Y_r = A_{r,m}$ and $Y_i = A_{i,m}$. The values of $A_{r,k}$, $A_{i,k}$, $B_{r,k}$, and $B_{i,k}$ have to consider four cases of k:

Case 1: k mod 4=0 (Real, positive weight)

$$A_{r,k} = \begin{cases} A_{r,k-1} + x_k (2j)^{-k} & \text{if } x_k \geq 0 \\ B_{r,k-1} + (4 - |x_k|)(2j)^{-k} & \text{if } x_k < 0 \end{cases}$$

$$B_{r,k} = \begin{cases} A_{r,k-1} + (x_k - 1)(2j)^{-k} & \text{if } x_k > 0 \\ B_{r,k-1} + (3 - |x_k|)(2j)^{-k} & \text{if } x_k \leq 0 \end{cases}$$

Case 2: k mod 4=1 (Imaginary, negative weight)

$$A_{i,k} = \begin{cases} A_{i,k-1} + |x_k|(2j)^{-k} & \text{if } x_k \leq 0 \\ B_{i,k-1} + (4 - x_k)(2j)^{-k} & \text{if } x_k > 0 \end{cases}$$

$$B_{i,k} = \begin{cases} A_{i,k-1} + (|x_k| - 1)(2j)^{-k} & \text{if } x_k < 0 \\ B_{i,k-1} + (3 - x_k)(2j)^{-k} & \text{if } x_k \geq 0 \end{cases}$$

Case 3: k mod 4=2 (Real, negative weight)

$$A_{r,k} = \begin{cases} A_{r,k-1} + |x_k|(2j)^{-k} & \text{if } x_k \leq 0 \\ B_{r,k-1} + (4 - x_k)(2j)^{-k} & \text{if } x_k > 0 \end{cases}$$

$$B_{r,k} = \begin{cases} A_{r,k-1} + (|x_k| - 1)(2j)^{-k} & \text{if } x_k < 0 \\ B_{r,k-1} + (3 - x_k)(2j)^{-k} & \text{if } x_k \geq 0 \end{cases}$$

Case 4: k mod 4=3 (Imaginary, positive weight)

$$A_{i,k} = \begin{cases} A_{i,k-1} + x_k (2j)^{-k} & \text{if } x_k \geq 0 \\ B_{i,k-1} + (4 - |x_k|)(2j)^{-k} & \text{if } x_k < 0 \end{cases}$$

$$B_{i,k} = \begin{cases} A_{i,k-1} + (x_k - 1)(2j)^{-k} & \text{if } x_k > 0 \\ B_{i,k-1} + (3 - |x_k|)(2j)^{-k} & \text{if } x_k \leq 0 \end{cases}$$

3. Basic operations

The basic on-line operations include (i) multiply by radix (shifting), (ii) digit by vector multiplication, and (iii) fully parallel addition.

3.1. Multiply by radix

Multiplying a vector by the radix $2j$ consists of shifting each digit one position to the left. In other words, to compute $Z = (2j)X$, for each digit of Z, $z_k = (z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0})$:

$$\begin{array}{ll} z^+_{k,1} = x^+_{k+1,1} & z^-_{k,1} = x^-_{k+1,1} \\ z^+_{k,0} = x^+_{k+1,0} & z^-_{k,0} = x^-_{k+1,0} \end{array} \quad (7)$$

3.2. Digit by vector multiplication

Digit by vector multiplication for radix $2j$ is defined such that, given input vector X=(x_1,x_2,. . .,x_m), where each digit x_k=($x^+_{k,1}$,$x^-_{k,1}$,$x^+_{k,0}$,$x^-_{k,0}$) and digit y_i, we want to compute $Z = Xy_i$, where Z=(z_0,z_1,. . .,z_m). Since the twiddle factor vectors are already known, performing digit by vector multiplication is simply a matter of selecting the appropriate multiple of the twiddle factor, based on the input digit.

3.3. Fully parallel addition

A radix $2j$ borrow save adder can be constructed as a modification of the radix-2 borrow save adder presented in [2] consisting of PPM and MMP adder cells. The details of the adder cells are shown in Figure 1. As a note, Xilinx FPGA's provide a fast carry chain for efficient addition. This alternative was considered and compared to the radix $2j$ borrow save adder, yet yielded a higher cost with the overall design. Thus the fast carry chain was not utilized.

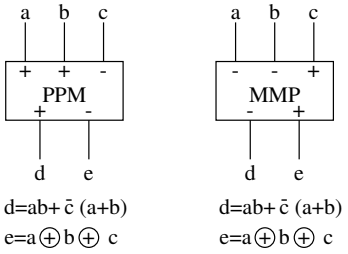

Figure 1. (a) PPM adder cell; (b) MMP adder cell

A radix $2j$ borrow-save adder considers two inputs: x=(x_{k-1}, . . ., x_0), and y=(y_{k-1}, . . ., y_0), and produces sum z=(z_{k-1}, . . ., z_0). A digit slice of the borrow-save adder is shown in Figure 2.

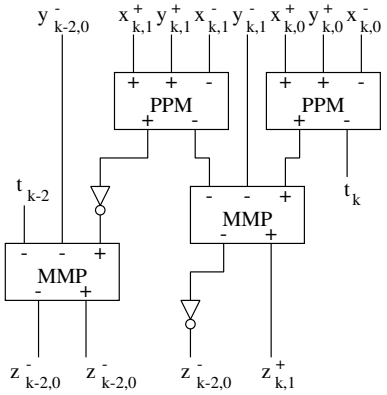

Figure 2. Radix $2j$ borrow-save adder

A radix $2j$ on-line adder can be constructed as a digit slice of the borrow-save adder by introducing appropriate delay registers, as shown in Figure 3.

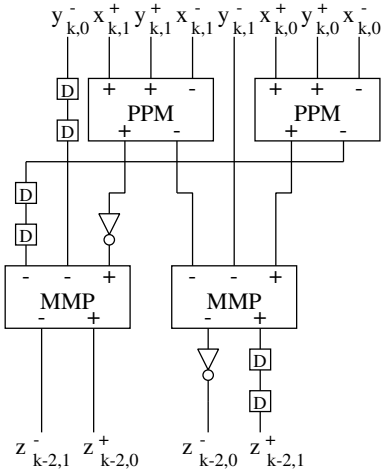

Figure 3. Complex on-line adder

4. Complex on-line butterfly

Four versions of the on-line butterfly module exist, due to the differing weights of the twiddle factor. Each twiddle factor W^k ($k = 0, 1, 2, 3$) can be evaluated as:

$$e^{-j(2\pi/N)} = \cos(\Theta_k) - j\sin(\Theta_k) \quad (8)$$

where $\Theta_k = 2\pi k/N$.

Since $W^0 = 1$, the butterfly equations for a module of type 0 simplify to:

$$\begin{aligned} X_{out} &= (X_r + Y_r) + j(X_i + Y_i) \\ Y_{out} &= (X_r - Y_r) + j(X_i - Y_i) \end{aligned} \quad (9)$$

which can be implemented as two complex on-line adders.

Similarly, since $W^2 = -j$, the butterfly equations for a module of type 2 simplify to:

$$\begin{aligned} X_{out} &= (X_r + Y_i) + j(X_i - Y_r) \\ Y_{out} &= (X_r - Y_i) + j(X_i + Y_r) \end{aligned} \quad (10)$$

which can also be implemented as two complex on-line adders,

Since $W^1 = \frac{\sqrt{2}}{2} - j\frac{\sqrt{2}}{2}$ and $W^3 = -\frac{\sqrt{2}}{2} - j\frac{\sqrt{2}}{2}$, butterfly modules type 1 and type 3 require an actual constant by variable complex multiplication, and two complex additions, with constant $\pm\frac{\sqrt{2}}{2}$, as shown in Figure 4.

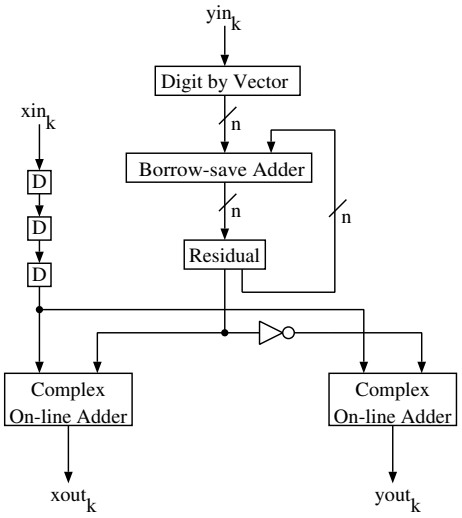

Figure 4. Complex on-line butterfly module

As mentioned before, constant by variable complex multiplication consists of referencing the appropriate multiple of the twiddle factor. Multiples of W^1 and W^3 are shown in Tables 1 and 2, respectively.

The cost, maximum clock frequency, and on-line delay, of the different types of complex on-line butterfly modules, when mapped onto a Xilinx 4000 series FPGA, assuming 8-bit data for real and imaginary components individually, are shown in Table 3.

5. Complex on-line FFT network

A complex on-line FFT network (COLFFT) can be implemented as a network of complex on-line butterfly modules.

Digit	Product vector
$\overline{3}$	$1\overline{2}.000\overline{1}33$
$\overline{2}$	$0\overline{1}.\overline{2}13\overline{2}12$
$\overline{1}$	$00.\overline{1}21\overline{3}21$
0	00.000000
1	$00.1\overline{2}1\overline{3}2\overline{1}$
2	$01.2\overline{1}3\overline{2}1\overline{2}$
3	$\overline{1}2.000\overline{1}3\overline{3}$

Table 1. Multiples of W^1

Digit	Product vector
$\overline{3}$	$11.0\overline{3}0\overline{1}3\overline{3}$
$\overline{2}$	$01.\overline{2}1\overline{3}2\overline{1}\overline{2}$
$\overline{1}$	$00.\overline{1}2\overline{1}3\overline{2}\overline{1}$
0	00.000000
1	$00.12\overline{1}3\overline{2}1$
2	$0\overline{1}.21\overline{3}212$
3	$\overline{1}1.030\overline{1}33$

Table 2. Multiples of W^3

For an 8-point FFT, seven type 0, one type 1, three type 2, and one type 3 butterfly modules are required, as shown in Figure 5. The design occupied 203 CLB's, and ran at a clock frequency of 42 MHz, with an on-line delay of 9.

The proposed design is compared to other approaches for the design of an 8-point FFT network with 8-bit precision, mapped onto a Xilinx 4000 series FPGA. In [7], radix 2 twiddle factors for a butterfly are precomputed and contained within lookup tables. In [3], FFT sub-equations are precomputed and contained within lookup tables. In [4], a DA approach using 4-operand multiply-accumulate modules is presented. The results are shown in Table 4.

As is shown, the proposed approach has the lowest cost in terms of CLB's, more than half of that of the alternative approaches. Frequency measurements were not stated in [3] or [7], yet [4] claimed a maximum frequency of 118 MHz and an on-line delay of 8. Therefore the design in [4]

Module	CLB's	MHz	On-line delay
0	10	121	2
1	49	79	5
2	10	103	2
3	51	79	5

Table 3. Complex On-line Butterfly Results

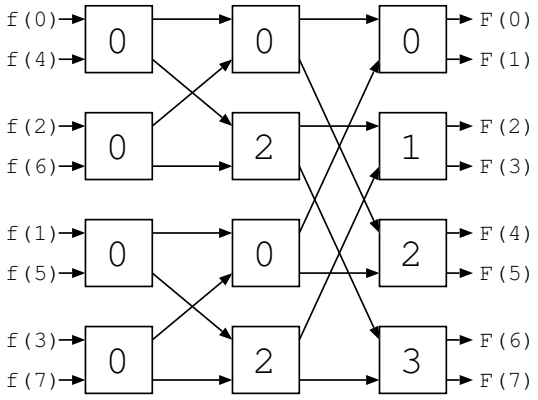

Figure 5. 8-point FFT network

Method	CLB's
[7]	684
[4]	596
[3]	432
COLFFT	206

Table 4. Comparison of FFT implementations

achieved much better timing performance than the proposed design. Despite the slower speed, the proposed design is well suited for mapping FFT networks of more (16,32, etc.) samples and it scales well for larger precision.

6. Summary

An approach was presented for the design of an on-line FFT network, using a redundant complex number system, for a more efficient representation and manipulation of complex operands for operations. Other applications demanding complex operations are also targets of interest and will be investigated in further research.

Acknowledgments. This research has been supported in part by the MICRO Grant 98037 "Reconfigurable Hardware for Numerically Intensive Computations", Raytheon Systems Company and Xilinx

References

[1] T. Aoki, Y. Ohi, and T. Higuchi. Redundant complex number arithmetic for high-speed signal processing. *IEEE Workshop on VLSI Signal Processing*, pages 523–532, October 1995.

[2] A. Guyot, B. Hochet, and J.-M. Muller. JANUS, an on-line multiplier/divider for manipulating large numbers. *9th IEEE Symposium on Computer Arithmetic*, pages 106–111, September 1989.

[3] H. Helal, S. Mashali, and A. Salama. A new implementation of FFT on FPGA using distributed arithmetic. *Proceedings of the 4th IEEE International Conference on Electronics, Circuits and Systems, Cairo, Egypt*, pages 1437–1443, December 1997.

[4] D. Lau, A. Schneider, M. Ercegovac, and J. Villasenor. FPGA-based structures for on-line signal processing. *Journal of VLSI signal processing*, 1999.

[5] S.-K. Lu, C.-W. Wu, and S.-Y. Kuo. On fault-tolerant FFT butterfly network design. *IEEE International Symposium on Circuits and Systems*, pages 69–72, May 1996.

[6] R. McIlhenny and M. Ercegovac. On-line algorithms for complex number arithmetic. *Proceedings of the 32nd Asilomar conference on signals, systems and computers*, pages 172–176, November 1998.

[7] L. Mintzer. The Xilinx FPGA as an FFT processor. *Electronic Engineering*, 69(845):81–84, May 1997.

[8] A. Vacher, M. Benkhebbab, A. Guyot, T. Rousseau, and A. Skaf. A VLSI implementation of parallel Fast Fourier Transform. *Proceedings of the European Design and Test Conference*, pages 250–255, March 1994.

[9] B. Wei, H. Du, and H. Chen. A complex-number multiplier using radix-4 digits. *Proceedings of the 12th symposium on computer arithmetic*, pages 84–90, 1995.