Hardware Architectures for the Fast Fourier Transform

Mario Garrido, Fahad Qureshi, Jarmo Takala, and Oscar Gustafsson

Abstract The fast Fourier transform (FFT) is a widely used algorithm in signal processing applications. FFT hardware architectures are designed to meet the requirements of the most demanding applications in terms of performance, circuit area, and/or power consumption. This chapter summarizes the research on FFT hardware architectures by presenting the FFT algorithms, the building blocks in FFT hardware architectures, the architectures themselves, and the bit reversal algorithm.

1 Introduction

The *Fourier transform* is one of the most important tools in digital signal processing. It is used to convert continuous signals in time domain into frequency domain. For discrete data, such as that in digital systems, the *discrete Fourier transform* (DFT) is used instead. The DFT transforms a finite sequence of equally spaced samples to a corresponding frequency domain representation as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \ldots, N-1, \tag{1}$$

where N denotes the DFT size, $x[n]$ is the input signal in the time domain, and $X[k]$ is the output signal in the frequency domain, which is defined for the frequencies

M. Garrido · O. Gustafsson
Linköping University, Linköping, Sweden
e-mail: mario.garrido.galvez@liu.se; oscar.gustafsson@liu.se

F. Qureshi (✉)
Tampere University of Technology, Tampere, Finland
e-mail: fahad@tut.fi

J. Takala
Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: jarmo.takala@tut.fi

© Springer International Publishing AG, part of Springer Nature 2019

S. S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
https://doi.org/10.1007/978-3-319-91734-4_17

$k \in [0, N-1]$. Note that both $x[n]$ and $X[k]$ are discrete signals. The coefficients W_N^{nk} are called *twiddle factors* and correspond to rotations in the complex plane defined as

$$W_N^{nk} = e^{-j\frac{2\pi}{N}nk} = \cos\left(\frac{2\pi}{N}nk\right) - j\sin\left(\frac{2\pi}{N}nk\right), \tag{2}$$

where j denotes the imaginary unit.

The original signal $x[n]$ can be recovered from $X[k]$ by calculating the *inverse discrete Fourier transform* (IDFT):

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]W_N^{nk}, \qquad n = 0, 1, \ldots, N-1. \tag{3}$$

The arithmetic complexity of the DFT in (1) is $O(N^2)$. However, the DFT contains redundant operations.

The term *fast Fourier transform* (FFT) refers to various methods that reduce the computational complexity of the DFT. The most popular one is the Cooley-Tukey algorithm [16]. Section 2 discusses FFT algorithms and representations.

For the implementation of FFT hardware architectures, Sect. 3 discusses the building blocks that they consist of, i.e., butterflies, rotators and shuffling circuits. Later, Sect. 4 presents the FFT hardware architectures. They are divided into fully parallel, iterative and pipelined FFTs. The outputs of FFT hardware architectures are generally provided in bit-reversed order. Section 5 explain the bit reversal algorithm used to sort them out. Finally, Sect. 6 summarizes the main conclusions of this chapter.

2 FFT Algorithms

2.1 The Cooley-Tukey Algorithm

The Cooley-Tukey algorithm [16] decomposes the DFT into a set of smaller DFTs, when N is not a prime number. Let us assume that $N = N_2 \cdot N_1$ and consider that n and k are calculated as

$$\begin{aligned} n &= n_1 N_2 + n_2, \quad \text{with} \quad n_1 = 0, \ldots, N_1 - 1 \quad \text{and} \quad n_2 = 0, \ldots, N_2 - 1; \\ k &= k_2 N_1 + k_1, \quad \text{with} \quad k_1 = 0, \ldots, N_1 - 1, \quad \text{and} \quad k_2 = 0, \ldots, N_2 - 1. \end{aligned} \tag{4}$$

Then, (1) can be written as

$$X[k_2N_1 + k_1] = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x[n_1N_2 + n_2] W_N^{(n_1N_2+n_2)(k_2N_1+k_1)}. \quad (5)$$

By exploiting the periodicity of the twiddle factors, i.e., $W_N^{\phi N_2} = W_{N_1}^{\phi}$, $W_N^{\phi N_1} = W_{N_2}^{\phi}$ and $W_N^{\phi N_1 N_2} = 1$, the equation is transformed into

$$X[k_2N_1 + k_1] = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x[n_1N_2 + n_2] W_{N_2}^{n_2k_2} W_{N_1}^{n_1k_1} W_N^{n_2k_1}, \quad (6)$$

which finally results in

$$X[k_2N_1 + k_1] = \underbrace{\sum_{n_2=0}^{N_2-1} \left[\underbrace{\left(\sum_{n_1=0}^{N_1-1} x[n_1N_2 + k_1] W_{N_1}^{n_1k_1} \right)}_{N_1-\text{point DFT}} \underbrace{W_N^{n_2k_1}}_{\text{Twiddlefactor}} \right] W_{N_2}^{n_2k_2}}_{N_2-\text{point DFT}}, \quad (7)$$

where the N_1-point and N_2-point DFTs are referred to as inner and outer DFTs, respectively. As a result, an N-point DFT is broken down into N_2 DFTs of size N_1 and N_1 FFTs of size N_2, with twiddle factor multiplications in the middle. This is illustrated in Fig. 1 for $N = 16$, $N_2 = 8$ and $N_1 = 2$.

In general, N can be the product of several numbers, i.e., $N = N_{m-1} \cdot N_{m-2} \cdot \ldots \cdot N_0$. *Radix-r* refers to the case in which $N_i = r, \forall i = 0 \ldots m-1$. The radix-$r$ FFT is derived by expressing n and k as

$$\begin{aligned} n &= n_{m-1} \cdot r^{m-1} + n_{m-2} \cdot r^{m-2} + n_1 \cdot r + n_0, \\ n_i &\in [0, \ldots, r-1], \forall i = 0, \ldots, m-1; \\ k &= k_{m-1} \cdot r^{m-1} + k_{m-2} \cdot r^{m-2} + k_1 \cdot r + k_0, \\ k_i &\in [0, \ldots, r-1], \forall k = 0, \ldots, m-1. \end{aligned} \quad (8)$$

This results in $m = \log_r N$ nested r-point DFTs with twiddle factors in between. When $r = 2$, each nested 2-point DFT is calculated on $N/2$ pairs of data and requires a total of N additions. This leads to an arithmetic complexity of $O(N \log N)$ for the entire FFT, compared to $O(N^2)$ in the DFT in (1).

Finally, the recursive application of Cooley-Tukey principle can be done by starting from the time domain sequence, which results in a decimation-in-time (DIT) decomposition. In a similar fashion, the decimation-in-frequency (DIF) decomposition is obtained by starting from the frequency sequence.

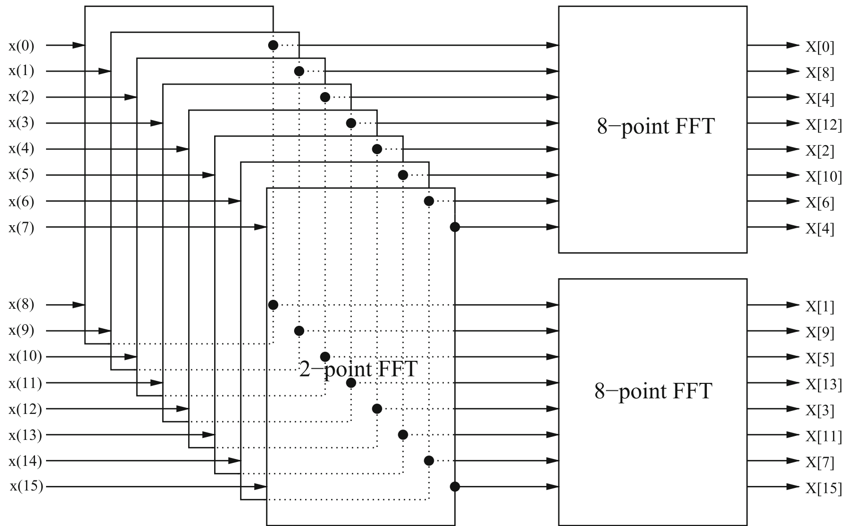


Fig. 1 Graphical representation of the Cooley-Tukey algorithm

2.2 Representation Using Flow Graphs

Figures 2 and 3 show the flow graphs of the radix-2 DIF and DIT FFT algorithms, respectively. The numbers at the input of the graph represent the indexes of the input sequence, $x[n]$, whereas those at the output are the frequencies, k, of the output signal $X[k]$. The flow graphs consist of a series of n stages, $s \in \{1 \dots n\}$. At each stage, additions, subtractions and rotations are calculated. Additions and subtractions come in pairs, forming the so called *butterflies*, which have the shape ∇. The upper output of the butterfly provides the sum of the inputs and the lower Δ
part subtracts the lower input from the upper one.

Each number ϕ in between butterflies represents a rotation, which corresponds to a complex multiplication by

$$W_N^{\phi} = e^{-j\frac{2\pi}{N}\phi}. \tag{9}$$

Rotations by $\phi \in \{0, N/4, N/2, 3N/4\}$ are called *trivial rotations*, due to the fact that they correspond to multiplications by $1, -j, -1$ or j. Trivial rotations can easily be calculated by interchanging the real and imaginary parts of the inputs and/or changing their signs.

Finally, an index I is added to the left of the flow graph, together with its binary representation $b_{n-1}b_{n-2}\dots b_1b_0$. This index together with the stage is used to refer to the rotations in the flow graph. For instance, the rotation with index $I = 14$ at stage $s = 1$ in Fig. 2 is $\phi_s(I) = \phi_1(14) = 6$. Through the chapter, the symbol

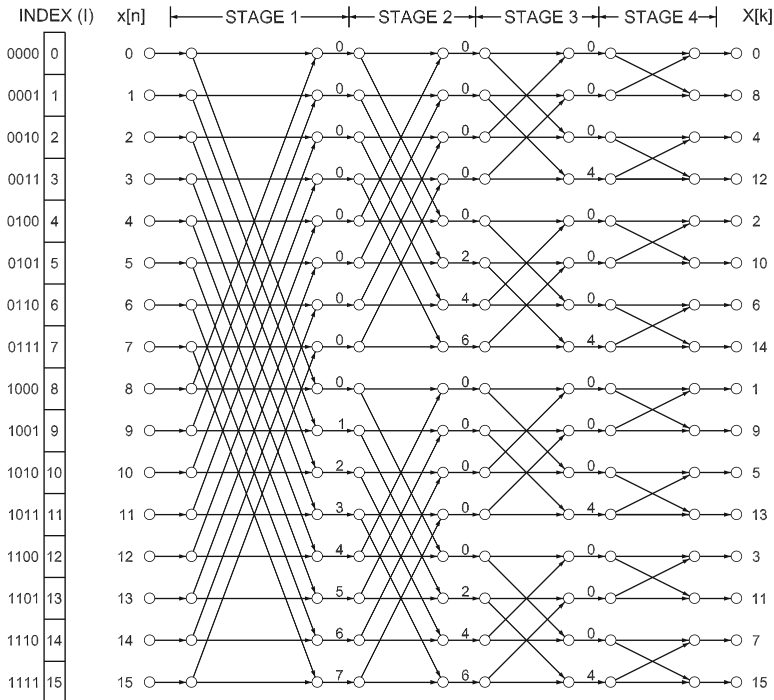


Fig. 2 Flow graph of a 16-point radix-2 DIF FFT

(≡) is used to relate decimal numbers and their binary representation, e.g., $I \equiv b_{n-1}b_{n-2}\ldots b_1b_0$.

When the FFT size N is large, it is not feasible to represent the FFT by a flow graph. In this case, the binary tree and the triangular matrix representations are useful tools to represent the algorithms in a simple manner.

Note that any FFT flow graph of the same radix will look the same. The only difference is where the twiddle factor multiplications are positioned. This can be seen by comparing Figs. 2 and 3. Hence, all the following algorithms only differ in the twiddle factor multiplications, although this may provide significant differences when implemented.

2.3 Binary Tree Representation

The binary tree representation [61, 83] is a generalization of the Cooley-Tukey algorithm. In the Cooley-Tukey algorithm, N is decomposed into a product of factors, i.e., $N = N_{m-1} \cdot N_{m-2} \cdot \ldots \cdot N_0$. This results in nested DFTs where each DFT contains the previous one. Conversely, in the binary tree representation, N is only split in two factors, i.e., $N = P \cdot Q$, which is analogous to (7). Then, both P

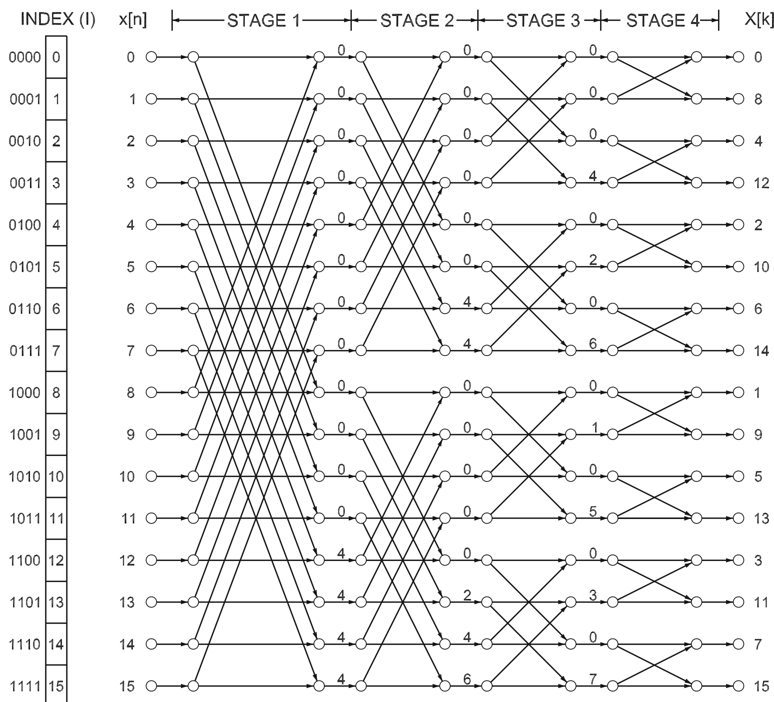


Fig. 3 Flow graph of a 16-point radix-2 DIT FFT

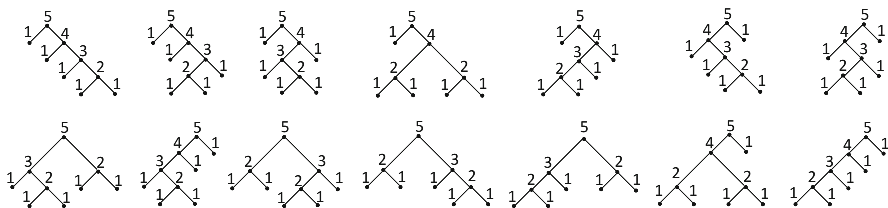


Fig. 4 Binary tree diagram of all possible algorithm for $N = 32$

and Q are again decomposed in two factors each. This process repeats iteratively forming a tree in which each node is split in two unless it is a leaf node.

A binary tree diagram [61] is an effective way of understanding the difference between FFT algorithms. For instance, Fig. 4 shows all the binary tree representations for $N = 32$. Note that each node has at most two branches.

In the binary tree representation, the upper node is assigned a value n to represent the 2^n-point DFT. Then, n is split into p and q, where $n = p + q$, $P = 2^p$ and $Q = 2^q$. After the first iteration, the remaining DFTs are again divided into smaller DFTs using the same procedure, so that the value of a node is the same as the sum of the sub-nodes.

The values of the sub-nodes can be chosen arbitrarily at each iteration. This leads to a large number of alternatives. In general, for $N = 2^n$, the number of N-point

FFT algorithms generated by using binary trees is [83]

$$\frac{(2(n-1))!}{n!(n-1)!}, \tag{10}$$

which comes from all the possible selections at each iteration. For a 32-point FFT, the number of FFT algorithms according to (10) is 14, which corresponds to the cases in Fig. 4.

The twiddle factors at each stages are directly obtained from the binary tree. This is done by going across the binary tree from left to right. In this process, the final leafs of the tree represented by 1 are skipped, as they correspond to radix-2 DFT operations. For example, the sequence of numbers in Fig. 4a is 5, 4, 3, 2. Given this sequence, the number of angles of the corresponding twiddle factors is the power of these numbers. Thus, for this example the twiddle factors are W_{32}, W_{16}, W_8, and W_4. This corresponds to the DIF decomposition, as it is obtained by decimating a 2-point DFT at each iteration from the input samples towards the output frequencies. How to generate the rotation coefficients ϕ for any FFT stage is described in [83].

2.4 Triangular Matrix Representation

The triangular matrix representation [24] is based on the ideas that rotations can be moved among FFT stages. The triangular matrix representation of some typical 16-point FFT algorithms is shown in Fig. 5. Rows are numbered as $x = 1, \ldots, n-1$ from top to bottom and columns as $y = 1, \ldots, n-1$ from left to right. Each element in row x and column y, M_{xy}, corresponds to a set of rotations that can be moved through several stages. Its value is the stage where these rotations are placed, which must be in the range $x \leq M_{xy} \leq y$.

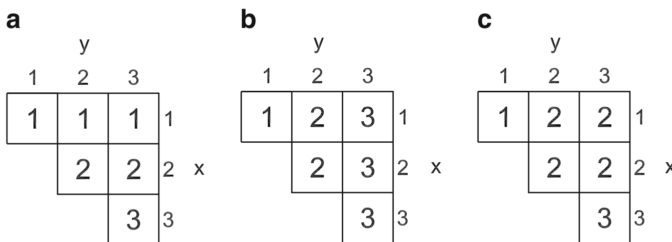


Fig. 5 Triangular matrix representation of typical 16-point FFT algorithms: (**a**) radix-2 DIF, (**b**) radix-2 DIT, (**c**) radix-2^2 DIF

Accordingly, the rotation coefficients $\phi_s(I)$ at any FFT stage s are calculated as

$$\phi_s(I) = \sum_{M_{xy=s}} b_{n-x} \cdot b_{n-y-1} \cdot 2^{n+(x-y)-2}. \tag{11}$$

Note that the numbering of rows and columns differs from the original paper [24], where the variables $i = n - x$ and $j = n - y - 1$ are used instead of x and y.

By applying Eq. (11) to Fig. 5a, the rotation coefficients are obtained as

$$\begin{aligned} \phi_1(I) &= b_3 \cdot b_2 \cdot 2^2 + b_3 \cdot b_1 \cdot 2^1 + b_3 \cdot b_0 \cdot 2^0 = b_3 \cdot [b_2 b_1 b_0]; \\ \phi_2(I) &= b_2 \cdot b_1 \cdot 2^2 + b_2 \cdot b_0 \cdot 2^1 = b_2 \cdot [b_1 b_0 0]; \\ \phi_3(I) &= b_1 \cdot b_0 \cdot 2^2 = b_1 \cdot [b_0 00], \end{aligned} \tag{12}$$

which correspond to the radix-2 DIF algorithm in Fig. 2. For instance, if $s = 1$ and $I = 14 \equiv 1110 = b_3 b_2 b_1 b_0$, then according to (12), $\phi_1(14) = b_3 \cdot [b_2 b_1 b_0] = 1 \cdot [110] = 6$. This corresponds to the rotation by 6 for $s = 1$ and $I = 14$ in Fig. 2. The rotations for the other algorithms in Fig. 5 can be derived in the same way.

In the triangular matrix representation, the radix-2 DIF FFT is the case where all the rotations are in the lowest possible stage, i.e., $\forall x, y, M_{xy} = x$. Analogously, the radix-2 DIT FFT is the case where all the rotations are in the highest possible stage, i.e., $\forall x, y, M_{xy} = y$. Finally, the radix-2^2 DIF FFT is the case where the rotations of the radix-2 DIF algorithm in odd stages are moved to the next even stage, except for those in the main diagonal, which cannot be moved.

Given that $x \leq M_{xy} \leq y$, each element M_{xy} can take $y - x + 1$ different values. By multiplying all the alternatives, the total number of algorithms that are represented by the triangular matrix as a function of n is

$$\prod_{k=1}^{n-1} (n-k)^k. \tag{13}$$

This is a large number of algorithms that includes, among others, all the algorithms representable by a binary tree.

2.5 The Radix in FFTs

The concept of radix has been used since the beginning of the FFT. It serves to distinguish different FFT algorithms. The radix is represented with a base and an exponent, i.e., $r = \rho^\alpha$. The base ρ indicates the size of the butterflies, i.e., the smallest DFT size that is used for the decomposition. Both base and exponent together provide the rotations at the FFT stages.

The first radices to be considered were radix-2, radix-4 and higher powers of two. These algorithms can be derived by the Cooley-Tukey algorithms as in (8).

Split radix [20] was also proposed in the early days. It combines radix-2 and radix-4 into an L-shaped butterfly. Split radix is known for having the least number of non-trivial rotations among FFT algorithms. However, it is seldom used in FFT hardware architectures due to the irregularity in the distribution of rotation.

Some years later, radix-2^2 was introduced for FFT hardware architectures [45]. From the algorithmic point of view, radix-2^2 is the same algorithm as radix-4 as both carry out exactly the same arithmetic operations [24]. However, both radices lead to different FFT architectures, due to the fact that radix-2^2 groups the calculations into 2-point butterflies and radix-4 groups them into 4-point butterflies.

The binary tree decomposition provides radix-2^k, for $k \in \mathbb{N}$. Radix-2^k is obtained when any node v of the binary tree is split into k and $v - k$, being k constant. Notice that the concept of radix is not unique any more, as radix-2^k may refer to different trees. Furthermore, many FFT algorithms based on the binary tree cannot be described by a single radix, but by a *mixed radix*, such as radix-2^4,2^3. As a result, many algorithms are better referred to by their tree than by their radix. The same happens to the triangular matrix representation, where there are even algorithms that cannot be described by a radix.

2.6 Non-power-of-two and Mixed-Radix FFTs

Most of the FFT designs consider FFT sizes that are powers of two and the rest of the chapter focuses on them. However, there are cases when N is a non-power-of-two and/or is a combination of powers of different radices [11, 88, 105, 106, 111, 112].

When N is a power of the radix, i.e., $N = r^k$, the Cooley-Tukey algorithm is the most suitable approach, even when it is a non-power-of-two. When N is a product of powers of coprime numbers, i.e.,

$$N = \prod_i r_i^{k_i}, \tag{14}$$

where r_i and r_j are coprime $\forall i \neq j$, then the Cooley-Tukey algorithm leads to twiddle factors between blocks of different radices. Conversely, the twiddle factors in between those blocks do not appear when using the prime factor algorithm (PFA) [6, 38]. Therefore, the PFA algorithm is recommended under these circumstances.

3 Building Blocks for FFT Hardware Architectures

FFT hardware architectures consist of butterflies, rotators and circuits for data shuffling. In the architectures, butterflies/rotators may be used to calculate one or several of the butterflies/rotations of the flow graph. Circuits for data shuffling are used to generate the data order to the butterflies and rotators in the architecture.

3.1 Butterflies

Butterflies in FFT architectures are characterized by their radix. A radix-ρ butterfly is a circuit with ρ inputs and ρ outputs that calculates an ρ-point DFT. Therefore, it corresponds to a direct implementation of the ρ-point DFT flow graph, where each addition/rotation in the flow graph is translated into an adder/rotator.

Radix-ρ butterflies are used in radix-ρ^α FFTs, $\alpha \geq 0$. The most common radices for butterflies are radix-2 and radix-4, which cover all radix-2^k and radix-4^k FFTs. Radix-2 and radix-4 butterflies have the advantage that they only consist of adders, and no rotator is needed. In case of radix-4, its trivial rotation by $-j$ can be embedded by changing the routing of the signals and the signs in the butterfly operations. Conversely, higher-radix butterflies include non-trivial rotators, which increases their cost.

3.2 Rotators

In a digital system with complex signals, a rotation by an angle α can be described as

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} C & -S \\ S & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \tag{15}$$

where $X + jY$ is the result of the rotation and $C, S \in \mathbb{Z}$ are the real and imaginary part of the rotation coefficient $C + jS$.

Due the finite word length effects, the rotation by $C + jS$ provides an approximation of the angle α with a certain error, being

$$\begin{aligned} C &= R(\cos\alpha + \epsilon_c); \\ S &= R(\sin\alpha + \epsilon_s), \end{aligned} \tag{16}$$

where ϵ_c and ϵ_s are the relative approximation errors of the cosine and sine components, respectively, and R is the magnitude. The approximation error for a given rotation can then be calculated as [30]

$$\epsilon = \sqrt{\epsilon_c^2 + \epsilon_s^2}. \tag{17}$$

Here, it should be noted that although it is common that R is a power-of-two, any magnitude can be used for the rotators, as long as the magnitude is the same for all the rotators at the same FFT stage [34].

It is often common to map the angle range into one or two quadrants. In this way, the rotators can often be simplified at the cost of a W_2 or W_4 rotator at the end. Which in turn sometimes may be integrated with the preceding butterfly as discussed above. Hence, in the following, the quadrant discussion is sometimes neglected.

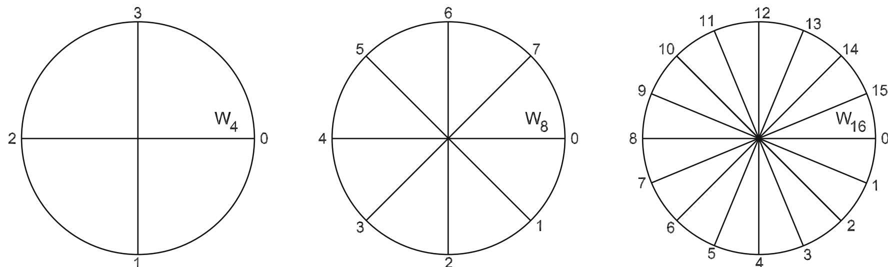


Fig. 6 Twiddle factors W_4, W_8 and W_{16}

The sine and cosine values are typically either stored in a memory or computed using an approximation, e.g., one of the methods discussed in the chapter "Arithmetic" [44]. The size of the sine and cosine memory can be reduced by utilizing octave symmetry, i.e.,

$$\begin{array}{r}\sin(\alpha) = -\cos\left(\alpha + \frac{\pi}{2}\right)\\ \cos(\alpha) = \sin\left(\alpha + \frac{\pi}{2}\right)\end{array}. \tag{18}$$

In this way, only sin and cos values for inputs between 0 and $\frac{\pi}{4}$ must be stored or approximated.

Rotators in FFT hardware architecture usually calculate rotations by different angles at different time instants. These rotation angles are part of a the set of rotations

$$W_L^{\phi} = e^{-j\frac{2\pi}{L}\phi}, \ \phi \in \{0, 1, \ldots, L-1\}, \tag{19}$$

where L is the resolution of the twiddle factor. Note that W_L refers to the entire set of L angles, whereas W_L^{ϕ} refers to a single angle which is the ϕ-th angle of the set. Any twiddle factor W_L divides the circumference in L equal parts and it contains the angles that create these divisions. The twiddle factors W_4, W_8 and W_{16} are shown in Fig. 6.

The number of different rotation angles that a rotator has to compute depends on the selected algorithm and architecture. When this number is large, general rotators are used. When the number of angles is small, the rotators may be simplified. Next sections describe different techniques to implement general and simplified rotators. They are mainly based on the use of multipliers or the CORDIC algorithm. An overview of techniques to implement rotators can be found in [73].

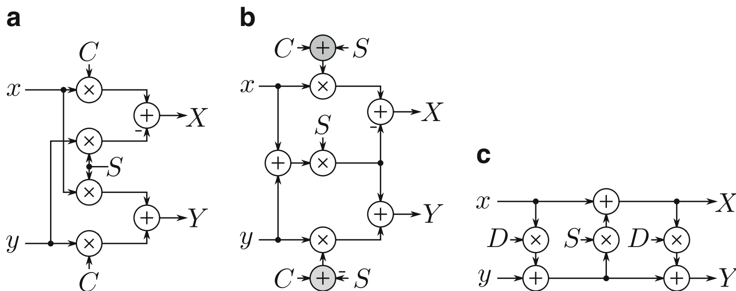


Fig. 7 Multiplier-based rotators. (**a**) Standard complex multiplier using four multiplications and two additions based on (20). (**b**) Complex multiplier using three multiplications and five additions based on (21) (three additions if the gray ones are replaced by memories). (**c**) Lifting-based rotator

3.2.1 Multiplier-Based General Rotators

The most straightforward approach is the direct implementation of Eq. (15), i.e.,

$$\begin{aligned} X &= xC - yS; \\ Y &= xS + yC. \end{aligned} \tag{20}$$

This requires four real-valued multipliers and two real-valued additions, as shown in Fig. 7a.

C and S are generally obtained as $C = \lfloor R\cos\alpha \rceil$ and $S = \lfloor R\sin\alpha \rceil$, where $\lfloor \cdot \rceil$ represents a rounding operation and being R a power of 2. Allowing R to be a non-power-of-two, widens the search for efficient rotation coefficients and the approximation errors can be reduced [30].

Other alternatives are based on rewriting (20) [101] to reduced the number of multipliers from four to three. Among them, the more interesting ones are

$$\begin{aligned} X &= x(C + S) - (x + y)S; \\ Y &= y(C - S) + (x + y)S, \end{aligned} \tag{21}$$

and

$$\begin{aligned} X &= (x + y)C - y(C + S); \\ Y &= (x + y)C - x(C - S). \end{aligned} \tag{22}$$

Both cases include a common term in the equations for X and Y that only need to be calculated once. Thus, when $C + S$ and $C - S$ are precomputed, these cases require three real-valued multiplications and three real-valued additions. Otherwise, they require three real-valued multiplications and five real-valued additions. The structure for (21) is shown in Fig. 7b.

Lifting-based rotators [8, 41, 79] are another way to obtain low-complexity rotations. There exist several different ways to rewrite (15) using lifting, one being

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} 1 & D \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ S & 1 \end{bmatrix} \begin{bmatrix} 1 & D \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \tag{23}$$

where $S = \sin(\alpha)$ and $D = \frac{1-\cos(\alpha)}{\sin(\alpha)} = \tan(\alpha/2)$. This requires three real-valued multiplications and three real-valued additions as shown in Fig. 7c. The other three standard alternatives have a similar form, but differ in how the coefficient corresponding to D is derived. Based on the angle, a structure can be selected to make sure that $|D| \leq \frac{1-\cos(\frac{\pi}{4})}{\sin(\frac{\pi}{4})} \approx 0.414$. This avoids the large magnitudes obtained for certain angles, as in the example structure $D \to \infty$ when $\alpha \to \pi$.

3.2.2 Multi-Stage General Rotators

As opposed to the multiplier-based rotators in the previous section, multi-stage rotators perform only a part of the total rotation in each stage. Step k of such a rotator can rotate with a set of angles, typically $\delta_k \alpha_k$, where $\delta_k \in \{-1, 1\}$ or $\delta_k \in \{-1, 0, 1\}$, i.e., a fixed angle, α_k, can be rotated in either direction or rotated in either direction or no rotation at all. Based on the remaining angle to be rotated, z_k, δ_k is determined and the remaining angle after the rotation, z_{k+1}, is updated.

The general structure of a rotation stage includes the calculation of x_{k+1}, y_{k+1} and z_{k+1} and is shown in Fig. 8. Multi-stage rotators mainly involve the CORDIC algorithm and its variations.

The total angle of rotation is then

$$\alpha = \sum_{k=0}^{M} \delta_k \alpha_k + \epsilon_\phi, \tag{24}$$

where ϵ_ϕ is the phase error of the approximation.

The available techniques optimize for different goals. For example, the CORDIC algorithm initially discussed, selects the angle of rotation such that the rotator becomes simple. A later discussed technique instead selects the angles to be suited for FFT computations with length power-of-two, i.e., the angle resolution is on a grid with power-of-two resolution.

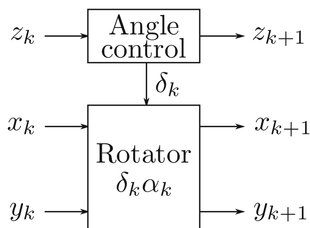


Fig. 8 General rotation stage in multi-stage rotator

In the CORDIC algorithm [97], discussed from a general perspective in the chapter “Arithmetic” [44], each stage multiply with the coefficients $C_k + jS_k = 2^k + j\delta_k$, where $\delta_k \in \{-1, 1\}$. The corresponding angles are then

$$\alpha_k = \tan^{-1}\left(2^{-k}\right). \tag{25}$$

These angles have the property that any angle α can be expressed as a sum of them. This enables the CORDIC algorithm to rotate by any rotation angle. Furthermore, due to the simplicity of the coefficients $C_k + jS_k$, each rotation stage is calculated with only 2 adders/subtracters as

$$\begin{aligned} x_{k+1} &= x_k C_k - y_k S_k = 2^k x_k - \delta_k y_k; \\ y_{k+1} &= x_k S_k + y_k C_k = 2^k y_k + \delta_k x_k. \end{aligned} \tag{26}$$

According to this, the scaling of each rotation stage is

$$R_k = \sqrt{C_k^2 + S_k^2} = C_k\sqrt{1 + (S_k/C_k)^2} = C_k\sqrt{1 + \tan(\alpha_k)^2} = \frac{2^k}{\cos(\alpha_k)}. \tag{27}$$

The term 2^k is generally compensated by removing the k LSBs after each rotation stage. The product of the scalings by $1/\cos(\alpha_k)$ at all the stages produces a total scaling of approximately 1.64, which can be compensated by multiplying the output of the CORDIC rotator by

$$K = \prod_{k=0}^{M} \cos(\alpha_k) = \prod_{k=0}^{M} \cos(\tan^{-1}(2^{-k})) = 0.6073. \tag{28}$$

The CORDIC assumes that the initial angle $z_0 = \alpha$ is in the interval $[-90°, 90°]$. Otherwise, this is easily achieved by a trivial rotation by 180°. Then, direction of the rotations δ_k are calculated for $k = 0, \ldots, M$ according to

$$\begin{aligned} \delta_k &= -\text{sign}(z_k); \\ z_{k+1} &= z_k + \delta_k \alpha_k, \end{aligned} \tag{29}$$

where z_k is the remainder rotation angle at the input of stage k, $z_{M+1} = \epsilon_\phi$, and $\text{sign}(\eta) = 1$ if $\eta \geq 0$ and $\text{sign}(\eta) = -1$ if $\eta < 0$.

There are multiple variations of the CORDIC algorithm. Some of the main modifications to the CORDIC algorithms are introduced next, and surveys on CORDIC techniques can be found in [2, 76]. For some of the mentioned approaches it is not straightforward to determine the rotation parameters at run-time. Hence, for these methods it is required to perform the design offline and store the control signals in memory rather than the angles. This approach is naturally possible for all techniques, and, as the sequence of angles is often known beforehand, most likely advantageous compared to storing the angle values.

The redundant CORDIC [63, 91] considers that $\delta_k \in \{-1, 0, 1\}$ [91] or even $\delta_k \in \{-2, -1, 0, 1, 2\}$ [63]. This enables several rotation angles at each CORDIC stage. However, the scaling for different angles is different, which demands a specific circuit for scaling compensation. The extended elementary angle set (EEAS) CORDIC [104] and the mixed-scaling-rotation (MSR) CORDIC [66, 81] also follow the idea of increasing the number of rotation angles per rotation stage.

The memoryless CORDIC [27] removes the need for rotation memory to store the FFT rotation angles. Instead, the control signals δ_k are generated from a counter. This is advantageous for large FFTs, which have stages with a large number of rotations.

The modified vector rotational (MRV) CORDIC [103] allows for skipping and repeating CORDIC stages, whereas the hybrid CORDIC [47, 89] divides the rotations into a coarse and a fine rotations. These techniques reduce the number of stages and, therefore, the latency of the CORDIC.

The CORDIC II [33] proposes new types of rotation stages: Friend angles, uniformly scaled redundant (USR) CORDIC and nanorotations. They allow for both a low latency and a small number of adders.

Finally, the base-3 rotators [57] consider an elementary angle set that is different to that of the CORDIC. All the rotations are generated by combining a small set of FFT angles. This set fits better the rotation angles of the FFT than that of the CORDIC, which results in a reduction in the rotation error, number of adders and latency of the circuit.

3.2.3 Simplified Multiplier-Based Rotators

Naturally, the real-valued multipliers in Sect. 3.2.1 can be implemented using shift-and-add multiplication, as discussed in the chapter "Arithmetic" [44]. This is specially useful when the rotator only needs to rotate by a single angle.

Initially, consider a rotation by $\frac{\pi}{4}$. As $\sin\left(\frac{\pi}{4}\right) = \cos\left(\frac{\pi}{4}\right)$ only one multiplication coefficient is required for each input (or output). Hence, either each input is multiplied by $\sin\left(\frac{\pi}{4}\right)$ and then the results are added and subtracted, as shown in Fig. 9a, or the inputs are first added and subtracted and then multiplied by $\sin\left(\frac{\pi}{4}\right)$, as shown in shown in Fig. 9b. The constant multiplication can be implemented using a optimal single constant multiplier from [42], while the best shift-and-add approximation with a given number of additions for the exact coefficient $\sin\left(\frac{\pi}{4}\right)$ can be found using the approach in [43].

For a general angle, the rotator shown in Fig. 7a can be used as a starting point. Now, both the multipliers sharing the same input can be simultaneously realized using multiple constant multiplication (MCM) techniques. This is illustrated in Fig. 10a, where the dashed box indicates the two multipliers realized using MCM. An identical MCM block is used for the other input. General MCM algorithms include [40, 98], while the algorithm in [18] is specifically tailored for two constants. An alternative view of the problem is to implement a sum-of-product block for the two multipliers going to the same output, as illustrated in Fig. 10b. However,

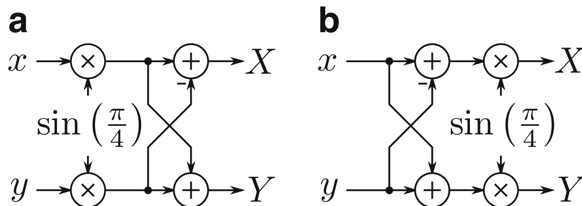


Fig. 9 Single constant rotations by $\pi/4$. (**a**) Multiplication followed by addition. (**b**) Addition followed by multiplication

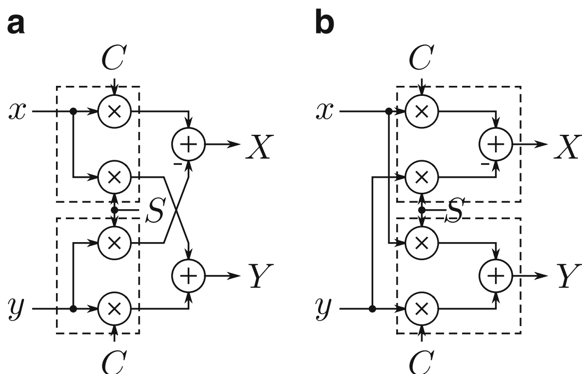


Fig. 10 Single constant rotations by a general angle. (**a**) Rotation implemented by using two MCM blocks as indicated by the dashed box. (**b**) Rotation implemented using two sum-of-product blocks as indicated by the dashed box

as the MCM and the sum-of-product problems are dual to each other, exactly the same number of adders are expected. A third option is to consider the problem as a constant matrix multiplication problem [5, 60].

For the rotators in Fig. 7b, c, no sharing can be done between the multipliers. However, due to the initially reduced complexity of the rotator, it may still happen that the total complexity is reduced. It should also be noted that the computations of the lifting-based structure in (23) can be merged to one matrix.

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} E & F \\ S & E \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \tag{30}$$

where $E = 1 + DS$ and $F = D(2 + SD)$. This forms another option to realize in any of the ways mentioned above.

When more than one angle is considered it is still possible to use shift-and-add techniques. Consider a W_8 rotator with indices $\phi \in \{0, 1\}$. For $\phi = 0$, the inputs are simply bypassed to the outputs. For $\phi = 1$ one of the approaches in Fig. 9 can be used. Then, the correct rotation result is selected by using a multiplexer, as shown in Fig. 11, where the $\frac{\pi}{4}$-rotator in Fig. 9a is used.

For several non-trivial angles, the naïve approach is to implement all different coefficients by using shift-and-add as an MCM problem and then select the correct angle by a multiplexer. However, the multiplexers can be merged with the shift-and-add network to significantly reduce the complexity [77, 96].

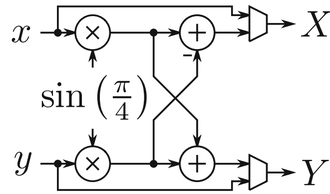


Fig. 11 W_8 rotator based on Fig. 9a

Of special interest is to note that it may be possible to find coefficients with longer word length, but with the same or smaller number of adders [34, 43]. Also, as earlier discussed, the magnitude may be selected as a non-power-of-two to further simplify the computations [34].

3.2.4 Simplified Multi-Stage Rotators

All the techniques mentioned in Sect. 3.2.2 are based on simple rotator stages. Clearly, for a constant coefficient the best selection of stages can be found. This has been explicitly proposed for CORDIC-based rotators [48, 75], although it can be easily generalized to arbitrary types of stages. If several angles are to be realized at different time instances it is of benefit having similar structure for the stages such that multiplexers can be easily introduced.

3.2.5 Rotators Based on Trigonometric Identities

Approaches based on trigonometrical identities [78, 84] search for expressions that are shared among different rotation angles. As a result, a simplified version of the rotator is obtained, which includes a reduced number of adders, multiplexers and multiplications by real constants. For instance, the twiddle factor W_{16} can be reduced to constant multiplications by $\cos(\pi/8)$ and/or $\sin(\pi/8)$ [84].

3.3 Shuffling Circuits

The purpose of the shuffling circuits in FFT architectures is to provide the data in the correct order needed for FFT stages. At each FFT stage, butterflies operate on pairs of data whose index I differ in b_{n-s} [29]. This can be observed in Fig. 2. For instance, the index of pairs of inputs to butterflies in stage 1 differs in $b_{n-s} = b_{4-1} = b_3$. As $I \equiv b_3b_2b_1b_0$, $I = 0 \equiv 0000$ and $I = 8 \equiv 1000$ differ in b_3 and are processed together in a butterfly at the first stage.

As different FFT stages demand different data orders, circuits for data permutation need to be included in between stages. These circuits have been studied by using life-time analysis and register allocation [74, 80], Kronecker products [39, 52–54, 82, 87, 92, 93] and bit-dimension permutations [21–23]. The following explanation is based on the latter, and follows the theory in [23].

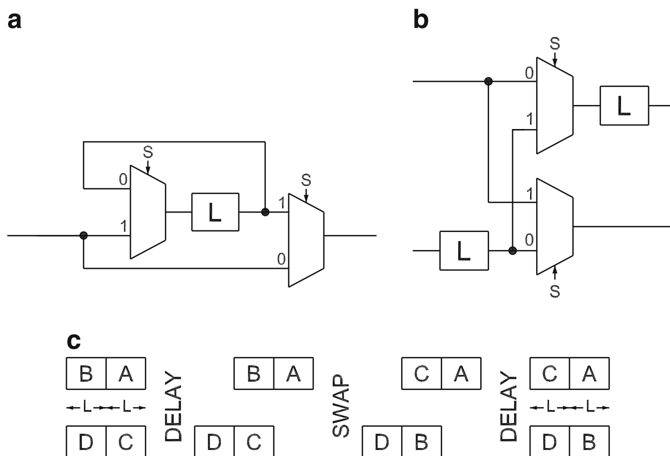


Fig. 12 Shuffling circuits. (**a**) Serial-serial permutation. (**b**) Serial-parallel permutation. (**c**) Dataflow of the serial-parallel permutation

Bit-dimension permutations are permutations on a group of 2^n data defined by a permutation of the n bits that represent the index of those data in binary. In FFTs, each stage processes $N = 2^n$ data that we index with $I = b_{n-1} \dots b_1 b_0$ as in Fig. 2. For these data, we can define their position [23]. For instance, $\text{P} \equiv b_0 b_3 b_2 b_1$ defines a specific data order, where each index $I \equiv b_3 b_2 b_1 b_0$ is in position P. Thus, $I = 8 \equiv 1000$ is in $\text{P} \equiv 0100$. For parallel data, a vertical bar separates serial and parallel data, e.g., $\text{P} \equiv b_0 b_3 b_2 | b_1$. The first part of the position until the bar indicates the time of arrival, which is defined as the relative time to the arrival of the first sample to a given point of the circuit. The second part after the bar indicates the terminal of arrival. The number of parallel dimensions p corresponds to the number of bits after the bar. Thus, for $\text{P} \equiv b_0 b_3 b_2 | b_1$, the number of parallel dimensions is $p = 1$ and $I = 8 \equiv 1000$ is in $\text{P} = 010|0$, i.e., it arrives at terminal $T(\text{P}) = 0$ at time $t(\text{P}) = 2 \equiv 010$.

Different positions define different orders, and the data order is changed when permuting the bits of the position. This is achieved by the shuffling circuits in FFT architectures. Figure 12 shows the main shuffling circuits used in FFT hardware architectures.

The circuit in Fig. 12a is used to calculate a serial-serial bit-dimension permutation. It consists of a buffer of length L and two multiplexers. This circuit is used to change the position of pairs of data separated by L clock cycles. This is done when one of the data is at the input of the circuit and the other one is at the output of the buffer. Then $S = 0$ is selected so that the position change. Otherwise, when $S = 1$ data passes through the buffer maintaining the order.

The length of the buffer defines the bit-dimension permutation that is carried out. If x_{n-1} is the first bit from the left and x_0 is the last one, then a serial-serial permutation that interchanges x_j and x_k, $j > k \geq p$ has a buffer length [23]

$$L = (2^j - 2^k)/2^p. \tag{31}$$

The circuit in Fig. 12b carries out a serial-parallel permutation. Figure 12c shows how it works: The groups of data A, B, C and D have L data in series each. First, the L data A and C arrive to the circuit at the upper and lower inputs, respectively. Then the L data B and D arrive. The circuit first delays the lower data, then it swaps the groups B and C, and finally it delays the upper part. The result is that data in groups B and C are interchanged.

The length of the buffers for interchanging x_j and x_k, $j \geq p > k$ is [23]

$$L = 2^{j-p}. \tag{32}$$

Finally, parallel-parallel permutations interchange parallel data flows. This does not require any hardware, as it can be hard wired.

4 FFT Hardware Architectures

There are three main types of FFT hardware architectures: Pipelined, iterative and fully parallel. Next section discusses when to choose each type and the following sections describe the different types.

4.1 Architecture Selection

Table 1 classifies the types of FFT hardware architectures in terms of the input data flow and the number of parallel samples, P. The higher P, the higher the throughput and also the larger the area of the circuit.

Iterative FFT hardware architectures loads data into a memory, then processes them and finally outputs them. During the processing, new data cannot be loaded. Therefore, iterative FFT architectures are suitable for processing data bursts.

Table 1 FFT architecture selection

FFT architecture type	Data flow	P
Iterative	Burst	≥ 1
Serial pipelined	Continuous flow	1
Parallel pipelined	Continuous flow	>1
Fully parallel	Continuous flow	N

The other architectures process data in a continuous flow. The difference among them is the parallelization P. The selection of the architecture depends on the expected performance. The throughput is calculated as $\text{Th} = P \cdot f_{\text{clk}}$ where f_{clk} is the clock frequency of the system and P is generally a power of 2. Thus, if the minimum throughput that the system must achieve is Th_{min}, then P is obtained as

$$P = 2^{\left\lceil \log_2 (\text{Th}_{\text{min}}/f_{\text{clk}}) \right\rceil}. \tag{33}$$

4.2 Fully Parallel FFT

Fully parallel FFT architectures correspond to the direct implementation of the FFT flow graph, i.e., each multiplication/addition in the flow graph is implemented by a separated multiplier/adder. Therefore, the number of hardware components is of order $O(N \log N)$. Fully parallel FFT architectures offer the maximum parallelization of the FFT algorithm. As a consequence, they provide the maximum throughput and the minimum latency among FFT architectures.

The implementation of rotators as shift-and-add and the selection of the FFT algorithm play an important role in the design of fully parallel FFTs. As each rotator calculates a rotation by a single angle, it is beneficial to use simplified rotators. Complementary to this, the selection of the FFT algorithms determines the number of non-trivial rotations in the fully parallel FFT. Therefore, algorithms with a small number of non-trivial rotations lead to more hardware-efficient implementations.

4.3 Iterative FFT Architectures

Iterative FFT architectures [15, 35, 46, 49, 55, 71, 72, 85, 93, 95] are also called memory-based or in-place FFT architectures. We suggest to call them iterative, as this is the term that reflects their nature better. Although the term memory-based is widely used, we prefer to avoid it due to the facts that non-iterative FFTs may also use memories, and iterative FFTs may use delays instead of memories.

Iterative FFT architectures generally consist of a memory or bank of data memories. Data are read from memory, processed by butterflies and rotators, and stored again in memory. This process repeats iteratively until all the stages of the FFT algorithm are calculated. The advantage of iterative FFTs is the reduction in the number of butterflies and rotators, as they are reused for different stages of the FFT.

A simple iterative FFT architecture is shown in Fig. 13. It consists of a memory and a processing element (PE), which computes the butterfly and rotation. As seen in Fig. 13, after every iteration data are stored in memory so it is necessary to compute whole FFT before it receives new samples. Thus, the memory-based architecture is unable to compute the FFT when data arrives continuously at the input. For a

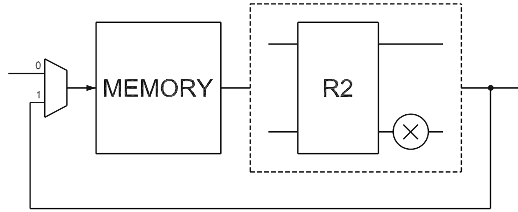


Fig. 13 Iterative (memory-based) FFT architecture

continuous flow, the iterative FFT requires additional memory to store the incoming data while the FFT is being calculated. If the processing time is longer than the time between FFTs, a larger processing element is also needed in order to handle the data flow.

Considering that the processing element is a radix-r butterfly and a set of rotators, the number of iterations of an iterative FFT architecture is calculated as

$$\text{It} = \frac{\log_2 N}{\log_2 r} = \frac{n}{\log_2 r}, \tag{34}$$

and the number of clock cycles to process each iteration is N/r, which leads to a total processing time of

$$T_{\text{proc}} = \frac{N \log_2 N}{r \log_2 r}. \tag{35}$$

The loading time depends on the FFT size and on the number of input samples that are loaded in parallel to the memory bank:

$$T_{\text{load}} = \frac{N}{P}. \tag{36}$$

When reading output data from the memories and writing the new input data are done simultaneously, the latency of the iterative FFT in clock cycles is

$$\text{Lat} = T_{\text{load}} + T_{\text{proc}} = \frac{N}{P} + \frac{N \log_2 N}{r \log_2 r}, \tag{37}$$

and, as N samples are processed every Lat clock cycles, the throughput in samples per clock cycle is

$$\text{Th} = \frac{N}{\text{Lat}} = \frac{rP}{r + P\frac{\log_2 N}{\log_2 r}}. \tag{38}$$

To increase the throughput and decrease the latency in iterative FFT architectures, high-radix processing elements are used. For instance, radix-16 is used in [49]. However, this also increases the amount of hardware of the FFT. Therefore, there is a trade-off between performance and hardware complexity.

The highest degree of parallelization for iterative FFT architectures consists in calculating simultaneously all the operations of the same stage of the FFT. This is done by the so called *column FFT* [3, 90], which computes the FFT by means of a column of processing elements composed of butterflies and rotators.. This architecture allows fast computation of the FFT, but requires a large number of hardware components, being the number of butterflies and rotators of order $O(N)$.

Finally, the radix-r butterfly in the PE processes r samples in parallel. Thus, r samples must be read from and written to memory each clock cycle. This requires to divide the memory into r memory banks that are accessed simultaneously. Furthermore, data must be written in memory in the same addresses that are read. This demands a conflict-free access strategy. Such memory organization can be studied from [15, 35, 46, 49, 55, 71, 72, 85, 93, 95].

4.4 Pipelined FFT Architectures

Pipelined FFT architectures [1, 13, 14, 17, 23, 25, 26, 29, 31, 32, 36, 45, 51, 58, 62, 64, 65, 67–69, 86, 94, 99, 107–109] consist of a set of $n = \log_\rho N$ stages connected in series, where ρ is the base of the radix $r = \rho^\alpha$. In a pipelined FFT, each stage of the architecture computes one stage of the FFT algorithm. The main advantage of pipelined architectures is that they process a continuous flow of data, with a good trade-off between performance and resources.

There are three main types of pipelined FFT architectures: feedback (FB), feedforward (FF) and serial commutator (SC). First, feedback architectures [13, 14, 17, 26, 45, 62, 64, 68, 69, 86, 94, 99, 107, 109] are characterized by their feedback loops, i.e., some outputs of the butterflies are fed back to the memories at the same stage. Feedback architectures are divided into single-path delay feedback (SDF) [17, 26, 45, 86, 109], which process a continuous flow of one sample per clock cycle, and multi-path delay feedback (MDF) [13, 14, 62, 64, 68, 69, 94, 99, 107], which process several samples in parallel. Second, feedforward architectures [1, 9, 10, 17, 25, 29, 31, 36, 45, 51, 58, 70, 86, 108] do not have feedback loops and each stage passes the processed data to the next stage. Single-delay commutator (SDC) FFTs are used for serial data [9, 10, 17, 70] and, multi-path delay commutator (MDC) FFTs [1, 25, 29, 31, 36, 45, 51, 58, 86, 108] are used to process several data in parallel. Finally, SC FFT architectures [32] are characterized by the use of circuits for bit-dimension permutation of serial data.

Pipelined FFT architectures can also be classified into serial pipelined and parallel pipelined FFT architectures. Next sections use this classification, which allows for comparing the hardware resources of FFTs with the same performance.

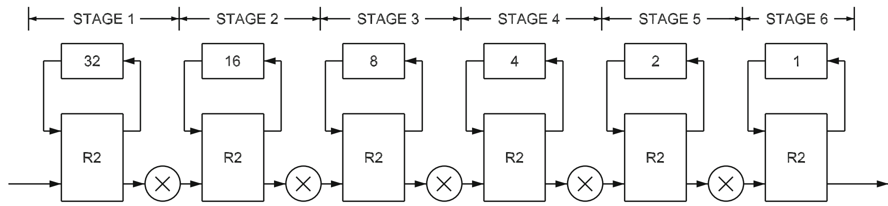
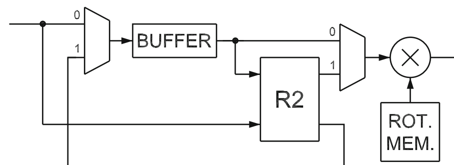


Fig. 14 64-Point SDF FFT architecture

Table 2 Twiddle factors of a 64-point DIF FFT for different radices

FFT algorithm	FFT stage				
	1	2	3	4	5
Radix-2 DIF	W_{64}	W_{32}	W_{16}	W_8	W_4
Radix-2^2 DIF	W_4	W_{64}	W_4	W_{16}	W_4
Radix-2^3 DIF	W_4	W_8	W_{64}	W_4	W_8
Radix-2^4 DIF	W_4	W_{64}	W_4	W_{16}	W_4

Fig. 15 Internal structure of a SDF stage



4.4.1 Serial Pipelined FFT Architectures

Serial pipelined FFT architectures consists of SDF, SDC and SC FFT architectures. These architectures are characterized by their relatively low number of components (adders, rotators and memory) and a throughput of 1 sample per clock cycle, which allows for high data rates of MSamples/s.

An example of SDF FFT architecture is shown in Fig. 14 for $N = 64$. It consists of $n = \log_2 N$ stages with radix-2 butterflies (R2), rotators and buffers. This architecture can implement radix-2^k FFT algorithms, including radix-2. The difference among FFT algorithms is reflected in the rotations calculated at each stage. Table 2 shows the twiddle factors for typical DIF algorithms. Note that W_4 corresponds to trivial rotations, which leads to simple rotators. Thus, the complexity of radices 2^2, 2^3 and 2^4 is smaller than that of radix-2.

The twiddle factors for DIT algorithms are the same as DIF ones, where the twiddle factor at stage s in the DIT case corresponds to that one in stage $n - s$ in the DIF case.

The internal structure of a SDF stage is shown in Fig. 15. It consists of a buffer, a radix-2 butterfly, two multiplexers, a rotator and eventually its rotation memory.

The buffer at stage s has length

$$L_s = 2^{n-s}. \tag{39}$$

The reason is that L_s input data are loaded to the buffer, causing a delay of those data. After L_s clock cycles, the output of the buffer is processed in the butterfly together with the input data for L_s clock cycles. During these clock cycles, one output of the butterfly is sent to the rotator and the other output of the butterfly is fed back to the buffer. Later, the values that go through the buffer are sent to the rotator, while a new sequence is loaded to the buffer. Therefore, the buffer is reused for inputting and outputting data.

This data management can be related to the data flow of the FFT algorithm: If we consider data arriving in series from top to bottom in the data flow of Fig. 2 then, at each stage, groups of L_s data must be delayed L_s clock cycles. This makes them arrive to the input of the butterfly at the same time as the data that they have to be operated with. For instance, if sample $x[0]$ at stage $s = 1$ is delayed $L_s = 2^{n-s} = 2^{4-1} = 8$ clock cycles, then it may be input to the butterfly together with $x[8]$. After the butterfly calculation, data are ordered in series again by delaying the lowest output of the butterfly L_s clock cycles.

In terms of hardware components, the radix-2^k SDF FFT requires one butterfly per stage, which results in $2 \log_2 N$ complex adders, a total memory of $N - 1$, which is the sum of all the buffer lengths, and a number of rotators that depends on the algorithm itself.

Figure 16 shows a 64-point radix-4 SDF FFT architecture. In this case, the number of stages is $\log_\rho N = \log_4 64 = 3$. The data management is analogous to the radix-2^k SDF FFT: Data are delayed so that all 4 data into a butterfly arrive at the same clock cycle.

At each stage, the radix-4 butterfly requires 8 complex adders, leading to a total of $8(\log_4 N)$ complex adders for the entire FFT. For the data, radix-4 uses 3 memories of size 4^{n-s} at stage s, which leads to a total FFT memory of $N - 1$.

The second type of serial FFT architectures is SDC [9, 10, 17, 70]. It is based on separating the data stream in two parallel data streams with the real and imaginary parts of the samples, respectively. An explanation of radix-2^k SDC and SDF architectures can be found in [17]. Generally, SDF FFTs are preferred to SDC ones, due to the larger data memory in SDC FFT architectures.

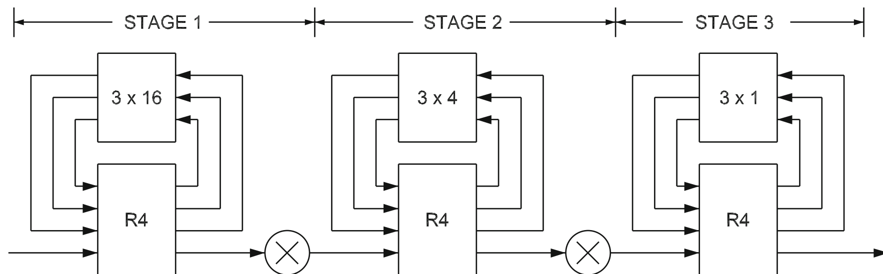


Fig. 16 64-Point radix-4 SDF FFT architecture

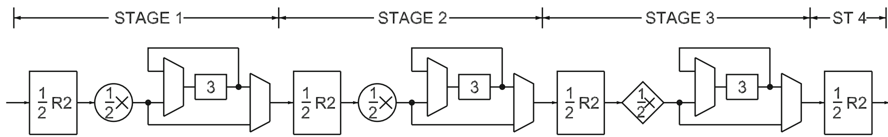


Fig. 17 16-Point radix-2 DIF SC FFT architecture

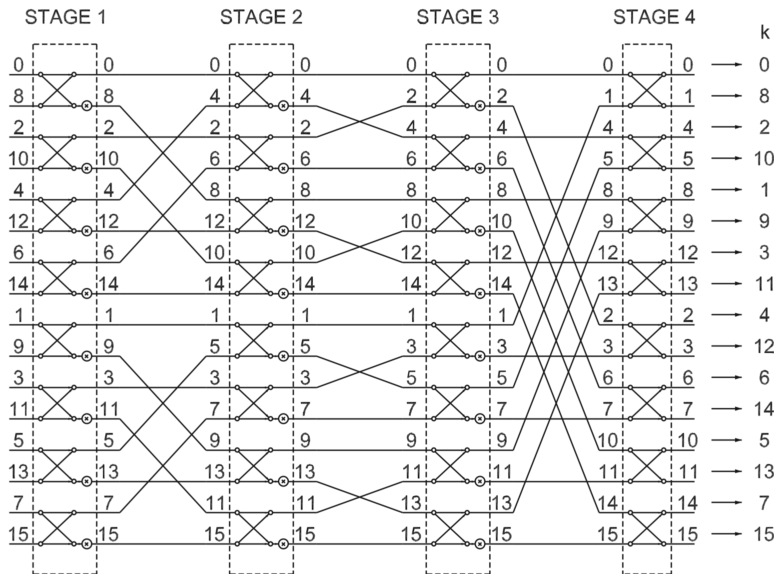


Fig. 18 Data management of the 16-point radix-2 DIF SC FFT

The third type of serial FFT architectures is SC [32]. Figure 17 shows a 16-point DIF serial commutator FFT. It uses circuits for bit-dimension permutation of serial data, which were described in Sect. 3.3.

The SC FFT is based on the idea of placing in consecutive clock cycles pairs of data that must be processed in the butterflies. Figure 18 shows the data management of the SC FFT in Fig. 17. The last column in Fig. 18 shows the frequencies k of $X[k]$. The rest of the numbers represent the data index I according to the definition in Fig. 2. The order of arrival to each FFT stage is from top to bottom. Therefore, $x[0]$ and $x[8]$ are the first and second inputs to the first stage, respectively. Butterflies operate on consecutive data. This allows for calculating the butterflies in two clock cycles, which halves the hardware complexity with respect to SDF FFTs. Note that the architecture in Fig. 17 uses half-butterflies (1/2 R2) instead of (R2). Rotators are also calculated in two clock cycles and its hardware is halved. The shuffling circuits in Fig. 17 delay three, one, and seven clock cycles, respectively. This can be observed in Fig. 18, where data that are exchanged are separated these numbers of clock cycles at the corresponding stages. Further details are shown in [32].

Table 3 Comparison of pipelined serial FFT architectures

Pipelined architecture	Area			Performance	
	Complex rotators	Complex adders	Complex sample memory	Latency (cycles)	Throughput (samples/cycle)
Radix-2 SDF [45]	$2(\log_4 N - 1)$	$4(\log_4 N)$	N	N	1
SDF Radix-2 [109]	$\log_4 N - 1$	$2(\log_4 N)$	$4N/3$	$4N/3$	1
SDF Radix-4 [19, 86]	$\log_4 N - 1$	$8(\log_4 N)$	N	N	1
SDF Radix-2^2 [45]	$\log_4 N - 1$	$4(\log_4 N)$	N	N	1
SDF Split-radix [110]	$\log_4 N - 1$	$4(\log_4 N)$	N	N	1
SDC Radix-2 [9, 10]	$2(\log_4 N - 1)$	$2(\log_4 N)$	$3N/2$	$3N/2$	1
SDC Radix-2 [70]	$2(\log_4 N - 1)$	$2(\log_4 N)$	$3N/2$	$3N/2$	1
SDC Radix-4 [4]	$\log_4 N - 1$	$3(\log_4 N)$	$2N$	N	1
SDC-SDF Radix-2 [100]	$\log_4 N - 1$	$2(\log_4 N) + 1$	$3N/2$	$3N/2$	1
SC Radix-2 [32]	$\log_4 N - 1$	$2(\log_4 N)$	N	N	1

Table 3 compares serial pipelined N-point FFT architectures. The table shows the trade-off between area and performance. Area is measured in terms of the number of complex rotators, adders and memory addresses, whereas performance is represented by throughput and latency. The throughput is 1 sample per clock cycle for all the architectures, which makes them comparable in terms of hardware resources.

As can be observed in the table, the order of magnitude of all parameters is the same for all architectures. The number of rotators and adders has order $O(\log N)$ and the memory has order $O(N)$. For large FFTs, the data memory takes up most of the area of the circuit, so it is preferable to use an architecture with a small memory. For small N, most of the FFT area is due to rotators, whose area is always larger than the area of the adders.

4.4.2 Parallel Pipelined FFT Architectures

This section discusses parallel FFT architectures, i.e., MDF and MDC. These architectures are characterized by their high throughputs. They can process P parallel samples in continuous flow and achieve a throughput of $\mathrm{Th} = P \cdot f_{\mathrm{clk}}$, reaching rates of GSamples/s.

MDF FFT architectures [13, 14, 62, 64, 68, 69, 94, 99, 102, 107] consists of multiple SDF paths in parallel. Thus, they work in a similar way as SDF FFT

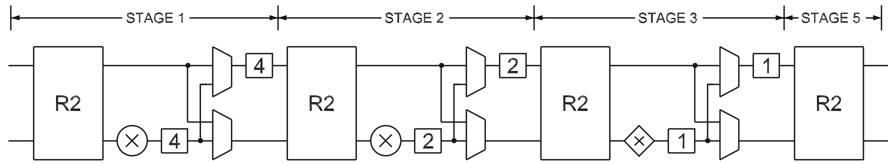


Fig. 19 16-Point 2-parallel radix-2 DIF MDC FFT architecture

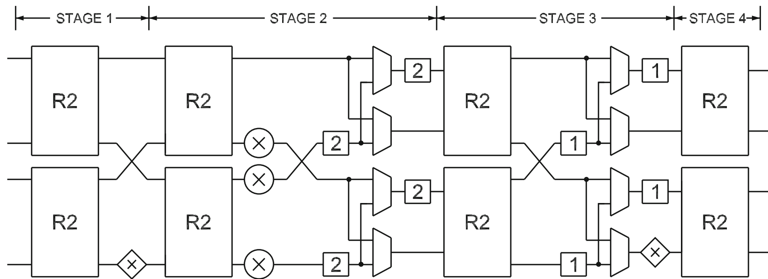


Fig. 20 16-Point 4-parallel radix-2^2 MDC FFT

architectures. The only difference is that the parallel SDF paths are interconnected either at some intermediate stages [102] or in the last stages.

MDC FFT architectures [1, 25, 29, 31, 36, 45, 51, 58, 86, 108] forward data to the next stage instead of feeding it back to the buffer of the same stage. Figure 19 shows a 16-point 2-parallel radix-2 DIF MDC FFT architecture. It consists of radix-2 butterflies, rotators and shuffling circuits for serial-parallel permutations. This architecture processes 2 samples per clock cycle in a continuous flow.

Higher throughput is achieved by increasing the parallelization. Figure 20 shows a 16-point 4-parallel radix-2^2 feedforward architecture. This architecture processes 4 samples per clock cycle in a continuous flow.

Table 4 compares parallel pipelined FFT hardware architectures. The architectures are classified into 4-parallel and 8-parallel ones. The table includes the number of adders and rotators. W_{16} and W_8 rotators are separated from general rotators due to its lower complexity. The number of complex adders is related to the architecture type: MDC FFTs have 100% utilization of butterflies and usually require less adders than MDF FFTs. The amount and complexity of the rotators depend on the radix and on the FFT size. Radices-2^4 and 2^3 are usually the best options. They achieve the least number of general rotators with some overhead of W_{16} and W_8 rotators. New approaches focus on reducing the amount rotators in parallel pipelined FFT architectures even further [31]. Finally, for most parallel pipelined FFT architectures the total data memory size is $N - P$.

Table 4 Comparison of parallel pipelined FFT architectures

Pipelined architecture	Area			
	Complex adders	Rotators		
		General	W_{16}	W_8
4-Parallel architectures				
R2 MDC, [36]	$4(\log_2 N)+4$	$2(\log_2 N)-4$	0	0
R4 MDC, [86]	$4(\log_2 N)$	$3\lceil(\log_2 N)/2\rceil-3$	0	0
R4 MDC, [108]	$4(\log_2 N)$	$3\lceil(\log_2 N)/2\rceil-3$	0	0
$R2^2$ MDC, [29]	$4(\log_2 N)$	$3\lceil(\log_2 N)/2\rceil-3$	0	0
$R2^3$ MDC, [29]	$4(\log_2 N)$	$4\lceil(\log_2 N)/3\rceil-4$	0	$2\lfloor(\log_2 N)/3\rfloor$
$R2^4$ MDF, [99]	$4(\log_2 N)$	$4\lceil((\log_2 N)-2)/4\rceil-1$	$4\lfloor((\log_2 N)-2)/4\rfloor$	(2)[a]
$R2^4$ MDF, [68]	$8(\log_2 N)$	$4\lceil(\log_2 N)/4\rceil-4$	$4\lfloor(\log_2 N)/4\rfloor$	(1)[b]
$R2^4$ MDF, [14]	$8(\log_2 N)$	$4\lceil(\log_2 N)/4\rceil-4$	$4\lfloor(\log_2 N)/4\rfloor$	(1)[b]
$R2^4$ MDC, [29]	$4(\log_2 N)$	$4\lceil(\log_2 N)/4\rceil-4$	$3\lfloor(\log_2 N)/4\rfloor$	(2)[b]
8-Parallel architectures				
R2 MDC, [56]	$8(\log_2 N)$	$4(\log_2 N)-8$	0	0
R2 MDF, [102]	$16(\log_2 N)$	$4(\log_2 N)-8$	0	0
$R2^2$ MDC, [29]	$8(\log_2 N)$	$6\lceil(\log_2 N)/2\rceil-6$	0	0
R8 MDC, [86]	$8(\log_2 N)$	$7\lceil(\log_2 N)/3\rceil-7$	0	$2\lfloor(\log_2 N)/3\rfloor$
$R2^3$ MDC, [29]	$8(\log_2 N)$	$7\lceil(\log_2 N)/3\rceil-7$	0	$2\lfloor(\log_2 N)/3\rfloor$
$R2^4$ MDF, [99]	$8(\log_2 N)$	$8\lceil((\log_2 N)-3)/4\rceil-1$	$8\lfloor((\log_2 N)-3)/4\rfloor$	2+(4)[c]
$R2^4$ MDF, [94]	$16(\log_2 N)$	$8\lceil(\log_2 N)/4\rceil-8$	$8\lfloor(\log_2 N)/4\rfloor$	(2)[b]
$R2^4$ MDC, [29]	$8(\log_2 N)$	$8\lceil(\log_2 N)/4\rceil-8$	$6\lfloor(\log_2 N)/4\rfloor$	(2)[b]

[a] Additional W_8 rotators required only when $\mathrm{mod}((\log_2 N)-2,4)=3$
[b] Additional W_8 rotators required only when $\mathrm{mod}((\log_2 N),4)=3$
[c] Additional W_8 rotators required only when $\mathrm{mod}((\log_2 N)-3,4)=3$

5 Bit Reversal for FFT Architectures

The outputs of FFT hardware architectures are generally provided in bit-reversed order. The bit reversal algorithm [37] is used to sort them out.

5.1 The Bit Reversal Algorithm

The bit reversal of $N = 2^n$ indexed data is an algorithm that reorders the data according to a reversing of the bits of the index [37]. This means that any sample with index $I \equiv b_{n-1} \ldots b_1 b_0$ moves to the place $\text{BR}(I) \equiv b_0 b_1 \ldots b_{n-1}$. Note that the bit reversal is an inversion operation, i.e., $\text{BR}(x) = \text{BR}^{-1}(x)$. Therefore, if data are in natural order, the bit reversal algorithm obtains them in bit-reversed order and vice versa. For instance, the bit reversal of $(0, 1, 2, 3, 4, 5, 6, 7)$ is $(0, 4, 2, 6, 1, 5, 3, 7)$ and the bit reversal of the latter set is the former.

5.2 Bit Reversal for Serial Data

For a hardware circuit that receives a series of N data in bit-reversed order, the bit reversal of the data is calculated by the permutation:

$$\sigma(u_{n-1} \ldots u_1 u_0) = u_0 u_1 \ldots u_{n-1}. \tag{40}$$

A first option to calculate the bit reversal of a series of data is to use a double buffering strategy [9, 59]. This consists of 2 memories of size N where even and odd FFT output sequences are written alternatively in the memories. The bit reversal can also be calculated using a single memory of size N. This is achieved by generating the memory address in natural and bit-reversed order, alternatively for even an odd sequences [7]. For SDC FFT architectures, the output reordering can be calculated by using two memories of $N/2$ addresses [9, 70]. Alternatively, the output reordering circuit can be integrated with the last stage of the FFT architecture [9, 10].

The optimum solution in terms of memory/delays for the bit reversal of serial data [28] consists of using a series of $j = \lfloor n/2 \rfloor - 1$ circuits for serial-serial bit-dimension permutations. Each of them carries out a permutation of the bits x_j and x_{n-1-j}, which requires a buffer of length

$$L = 2^{n-1-i} - 2^i. \tag{41}$$

By adding the buffer lengths, the total number of delays for even n is

$$(N - 1)^2, \tag{42}$$

and for odd n it is

$$\left(\sqrt{2N}-1\right)\left(\sqrt{\frac{N}{2}}-1\right). \tag{43}$$

5.3 Bit Reversal for Parallel Data

For parallel data, the bit reversal permutation is the same as for serial data, with the difference that the p less significant bits are parallel dimensions. As for serial data, some solutions are based on using memories [50, 108] and other ones are based on using buffers [12].

The optimum solution in terms of memory/delays for the bit reversal of parallel data [12] uses circuits for serial-serial permutations and parallel-parallel permutation. As in the bit reversal of serial data, these circuits are used to interchange bits x_j and x_{n-1-j} for $j = \lfloor n/2 \rfloor - 1$. Assuming that $p < n/2$, a serial-serial permutation is carried out when $0 \leq j < p$ and a serial-parallel one when $p \leq j \leq \lfloor n/2 \rfloor - 1$. As a result, the total numbers of delays for even n is

$$D(\sigma) = N - 2\sqrt{N} + P, \tag{44}$$

and for odd n it is

$$D(\sigma) = N - \sqrt{2N} - \sqrt{\frac{N}{2}} + P. \tag{45}$$

6 Conclusions

More than 50 years after the first FFT algorithms were proposed, the design of FFT hardware architectures is still an active research field that involves multiple research topics. They include the study and selection of FFT algorithms, the design of rotators in hardware, the design of new FFT architectures and the data shuffling, including the bit reversal algorithm. Nowadays, new FFT algorithms as well as new representations for these algorithms are explored. The most common FFT algorithms are radix-2^k, and there is an increasing interest in non-power-of-two FFTs. The area in FFT architectures is reduced by implementing rotators as shift-and-add operations. Most approaches are based on simplifying a complex multiplier or on the CORDIC algorithm. New FFT hardware architectures that achieve fully utilization of butterflies and reduction of the number of rotators and their complexity have been proposed during the last years. Likewise, the optimum circuits for bit reversal have been proposed recently, and the research on shuffling circuits for the FFT is still an open research field.

References

1. Ahmed, T., Garrido, M., Gustafsson, O.: A 512-point 8-parallel pipelined feedforward FFT for WPAN. In: Proc. Asilomar Conf. Signals Syst. Comput., pp. 981–984 (2011)
2. Andraka, R.: A survey of CORDIC algorithms for FPGA based computers. In: Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, pp. 191–200. ACM (1998)
3. Argüello, F., Bruguera, J., Doallo, R., Zapata, E.: Parallel architecture for fast transforms with trigonometric kernel. IEEE Trans. Parallel Distrib. Syst. **5**(10), 1091–1099 (1994)
4. Bi, G., Jones, E.: A pipelined FFT processor for word-sequential data. IEEE Trans. Acoust., Speech, Signal Process. **37**(12), 1982–1985 (1989)
5. Boullis, N., Tisserand, A.: Some optimizations of hardware multiplication by constant matrices. IEEE Trans. Comput. **54**(10), 1271–1282 (2005)
6. Burrus, C., Eschenbacher, P.: An in-place, in-order prime factor FFT algorithm. Proc. IEEE Int. Symp. Circuits Syst. **29**(4), 806–817 (1981)
7. Chakraborty, T.S., Chakrabarti, S.: On output reorder buffer design of bit reversed pipelined continuous data FFT architecture. In: Proc. IEEE Asia-Pacific Conf. Circuits Syst., pp. 1132–1135. IEEE (2008)
8. Chan, S.C., Yiu, P.M.: An efficient multiplierless approximation of the fast Fourier transform using sum-of-powers-of-two (SOPOT) coefficients. IEEE Signal Process. Lett. **9**(10), 322–325 (2002)
9. Chang, Y.N.: An efficient VLSI architecture for normal I/O order pipeline FFT design. IEEE Trans. Circuits Syst. II **55**(12), 1234–1238 (2008)
10. Chang, Y.N.: Design of an 8192-point sequential I/O FFT chip. In: Proc. World Congress Eng. Comp. Science, vol. II (2012)
11. Chen, J., Hu, J., Lee, S., Sobelman, G.E.: Hardware efficient mixed radix-25/16/9 FFT for LTE systems. IEEE Trans. VLSI Syst. **23**(2), 221–229 (2015)
12. Cheng, C., Yu, F.: An optimum architecture for continuous-flow parallel bit reversal. IEEE Signal Process. Lett. **22**(12), 2334–2338 (2015)
13. Cho, S.I., Kang, K.M.: A low-complexity 128-point mixed-radix FFT processor for MB-OFDM UWB systems. ETRI J. **32**(1), 1–10 (2010)
14. Cho, S.I., Kang, K.M., Choi, S.S.: Implementation of 128-point fast Fourier transform processor for UWB systems. In: Proc. Int. Wireless Comm. Mobile Comp. Conf., pp. 210–213 (2008)
15. Cohen, D.: Simplified control of FFT hardware. IEEE Trans. Acoust., Speech, Signal Process. **24**(6), 577–579 (1976)
16. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)
17. Cortés, A., Vélez, I., Sevillano, J.F.: Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation. IEEE Trans. Signal Process. **57**(7), 2824–2839 (2009)
18. Dempster, A.G., Macleod, M.D.: Multiplication by two integers using the minimum number of adders. In: Proc. IEEE Int. Symp. Circuits Syst., vol. 2, pp. 1814–1817 (2005)
19. Despain, A.M.: Fourier transform computers using CORDIC iterations. IEEE Trans. Comput. **C-23**, 993–1001 (1974)
20. Duhamel, P., Hollmann, H.: 'Split radix' FFT algorithm. Electron. Lett. **20**(1), 14–16 (1984)
21. Edelman, A., Heller, S., Johnsson, L.: Index transformation algorithms in a linear algebra framework. IEEE Trans. Parallel Distrib. Syst. **5**(12), 1302–1309 (1994)
22. Fraser, D.: Array permutation by index-digit permutation. J. Assoc. Comp. Machinery (ACM) **23**(2), 298–309 (1976)
23. Garrido, M.: Efficient hardware architectures for the computation of the FFT and other related signal processing algorithms in real time. Ph.D. thesis, Universidad Politécnica de Madrid (2009)
24. Garrido, M.: A new representation of FFT algorithms using triangular matrices. IEEE Trans. Circuits Syst. I **63**(10), 1737–1745 (2016)

25. Garrido, M., Acevedo, M., Ehliar, A., Gustafsson, O.: Challenging the limits of FFT performance on FPGAs. In: Int. Symp. Integrated Circuits, pp. 172–175 (2014)
26. Garrido, M., Andersson, R., Qureshi, F., Gustafsson, O.: Multiplierless unity-gain SDF FFTs. IEEE Trans. VLSI Syst. **24**(9), 3003–3007 (2016)
27. Garrido, M., Grajal, J.: Efficient memoryless CORDIC for FFT computation. In: Proc. IEEE Int. Conf. Acoust. Speech Signal Process., vol. 2, pp. 113–116 (2007)
28. Garrido, M., Grajal, J., Gustafsson, O.: Optimum circuits for bit reversal. IEEE Trans. Circuits Syst. II **58**(10), 657–661 (2011)
29. Garrido, M., Grajal, J., Sánchez, M.A., Gustafsson, O.: Pipelined radix-2^k feedforward FFT architectures. IEEE Trans. VLSI Syst. **21**(1), 23–32 (2013)
30. Garrido, M., Gustafsson, O., Grajal, J.: Accurate rotations based on coefficient scaling. IEEE Trans. Circuits Syst. II **58**(10), 662–666 (2011)
31. Garrido, M., Huang, S.J., Chen, S.G.: Feedforward FFT hardware architectures based on rotator allocation. IEEE Trans. Circuits Syst. I **65**(2), 581–592 (2018)
32. Garrido, M., Huang, S.J., Chen, S.G., Gustafsson, O.: The serial commutator (SC) FFT. IEEE Trans. Circuits Syst. II **63**(10), 974–978 (2016)
33. Garrido, M., Källström, P., Kumm, M., Gustafsson, O.: CORDIC II: A new improved CORDIC algorithm. IEEE Trans. Circuits Syst. II **63**(2), 186–190 (2016)
34. Garrido, M., Qureshi, F., Gustafsson, O.: Low-complexity multiplierless constant rotators based on combined coefficient selection and shift-and-add implementation (CCSSI). IEEE Trans. Circuits Syst. I **61**(7), 2002–2012 (2014)
35. Garrido, M., Sánchez, M., López-Vallejo, M., Grajal, J.: A 4096-point radix-4 memory-based FFT using DSP slices. IEEE Trans. VLSI Syst. **25**(1), 375–379 (2017)
36. Glittas, A.X., Sellathurai, M., Lakshminarayanan, G.: A normal I/O order radix-2 FFT architecture to process twin data streams for MIMO. IEEE Trans. VLSI Syst. **24**(6), 2402–2406 (2016)
37. Gold, B., Rader, C.M.: Digital Processing of Signals. New York: McGraw Hill (1969)
38. Good, I.J.: The interaction algorithm and practical Fourier analysis. J. Royal Statistical Society B **20**(2), 361–372 (1958)
39. Granata, J., Conner, M., Tolimieri, R.: Recursive fast algorithm and the role of the tensor product. IEEE Trans. Signal Process. **40**(12), 2921–2930 (1992)
40. Gustafsson, O.: A difference based adder graph heuristic for multiple constant multiplication problems. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 1097–1100. IEEE (2007)
41. Gustafsson, O.: On lifting-based fixed-point complex multiplications and rotations. In: Proc. IEEE Symp. Comput. Arithmetic (2017)
42. Gustafsson, O., Dempster, A.G., Johansson, K., Macleod, M.D., Wanhammar, L.: Simplified design of constant coefficient multipliers. Circuits Syst. Signal Process. **25**(2), 225–251 (2006)
43. Gustafsson, O., Qureshi, F.: Addition aware quantization for low complexity and high precision constant multiplication. IEEE Signal Processing Letters **17**(2), 173–176 (2010)
44. Gustafsson, O., Wanhammar, L.: Arithmetic. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
45. He, S., Torkelson, M.: Design and implementation of a 1024-point pipeline FFT processor. pp. 131–134 (1998)
46. Hsiao, C.F., Chen, Y., Lee, C.Y.: A generalized mixed-radix algorithm for memory-based FFT processors. IEEE Trans. Circuits Syst. II **57**(1), 26–30 (2010)
47. Hsiao, S.F., Lee, C.H., Cheng, Y.C., Lee, A.: Designs of angle-rotation in digital frequency synthesizer/mixer using multi-stage architectures. In: Proc. Asilomar Conf. Signals Syst. Comput., pp. 2181–2185 (2011)
48. Hu, Y., Naganathan, S.: An angle recoding method for CORDIC algorithm implementation. IEEE Trans. Comput. **42**(1), 99–102 (1993)
49. Huang, S.J., Chen, S.G.: A high-throughput radix-16 FFT processor with parallel and normal input/output ordering for IEEE 802.15.3c systems. IEEE Trans. Circuits Syst. I **59**(8), 1752–1765 (2012)

50. Huang, S.J., Chen, S.G., Garrido, M., Jou, S.J.: Continuous-flow parallel bit-reversal circuit for MDF and MDC FFT architectures. IEEE Trans. Circuits Syst. I **61**(10), 2869–2877 (2014)
51. Jang, J.K., Kim, M.G., Sunwoo, M.H.: Efficient scheduling scheme for eight-parallel MDC FFT processor. In: Proc. Int. SoC Design Conf., pp. 277–278 (2015)
52. Järvinen, T.: Systematic methods for designing stride permutation interconnections. Ph.D. thesis, Tampere Univ. of Technology (2004)
53. Järvinen, T., Salmela, P., Sorokin, H., Takala, J.: Stride permutation networks for array processors. In: Proc. IEEE Int. Applicat.-Specific Syst. Arch. Processors Conf., pp. 376–386 (2004)
54. Järvinen, T., Salmela, P., Sorokin, H., Takala, J.: Stride permutation networks for array processors. J. VLSI Signal Process. Syst. **49**(1), 51–71 (2007)
55. Jo, B.G., Sunwoo, M.H.: New continuous-flow mixed-radix (CFMR) FFT processor using novel in-place strategy. IEEE Trans. Circuits Syst. I **52**(5), 911–919 (2005)
56. Johnston, J.A.: Parallel pipeline fast Fourier transformer. In: IEE Proc. F Comm. Radar Signal Process., vol. 130, pp. 564–572 (1983)
57. Källström, P., Garrido, M., Gustafsson, O.: Low-complexity rotators for the FFT using base-3 signed stages. In: Proc. IEEE Asia-Pacific Conf. Circuits Syst., pp. 519–522 (2012)
58. Kim, M.G., Shin, S.K., Sunwoo, M.H.: New parallel MDC FFT processor with efficient scheduling scheme. In: Proc. IEEE Asia-Pacific Conf. Circuits Syst., pp. 667–670 (2014)
59. Kristensen, F., Nilsson, P., Olsson, A.: Flexible baseband transmitter for OFDM. In: Proc. IASTED Conf. Circuits Signals Syst., pp. 356–361 (2003)
60. Kumm, M., Hardieck, M., Zipf, P.: Optimization of constant matrix multiplication with low power and high throughput. IEEE Transactions on Computers **PP**(99), 1–1 (2017)
61. Lee, H.Y., Park, I.C.: Balanced binary-tree decomposition for area-efficient pipelined FFT processing. IEEE Trans. Circuits Syst. I **54**(4), 889–900 (2007)
62. Lee, J., Lee, H., in Cho, S., Choi, S.S.: A high-speed, low-complexity radix-2^4 FFT processor for MB-OFDM UWB systems. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 210–213 (2006)
63. Li, C.C., Chen, S.G.: A radix-4 redundant CORDIC algorithm with fast on-line variable scale factor compensation. In: Proc. IEEE Int. Conf. Acoust. Speech Signal Process., vol. 1, pp. 639–642 (1997)
64. Li, N., van der Meijs, N.: A radix 2^2 based parallel pipeline FFT processor for MB-OFDM UWB system. In: Proc. IEEE Int. SOC Conf., pp. 383–386 (2009)
65. Li, S., Xu, H., Fan, W., Chen, Y., Zeng, X.: A 128/256-point pipeline FFT/IFFT processor for MIMO OFDM system IEEE 802.16e. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 1488–1491 (2010)
66. Lin, C.H., Wu, A.Y.: Mixed-scaling-rotation CORDIC (MSR-CORDIC) algorithm and architecture for high-performance vector rotational DSP applications. IEEE Trans. Circuits Syst. I **52**(11), 2385–2396 (2005)
67. Lin, Y.W., Lee, C.Y.: Design of an FFT/IFFT processor for MIMO OFDM systems. IEEE Trans. Circuits Syst. I **54**(4), 807–815 (2007)
68. Liu, H., Lee, H.: A high performance four-parallel 128/64-point radix-2^4 FFT/IFFT processor for MIMO-OFDM systems. In: Proc. IEEE Asia Pacific Conf. Circuits Syst., pp. 834–837 (2008)
69. Liu, L., Ren, J., Wang, X., Ye, F.: Design of low-power, 1GS/s throughput FFT processor for MIMO-OFDM UWB communication system. In: Proc. IEEE Int. Symp. Circuits Syst., pp. 2594–2597 (2007)
70. Liu, X., Yu, F., Wang, Z.: A pipelined architecture for normal I/O order FFT. Journal of Zhejiang University - Science C **12**(1), 76–82 (2011)
71. Ma, Y., Wanhammar, L.: A hardware efficient control of memory addressing for high-performance FFT processors. IEEE Trans. Signal Process. **48**(3), 917–921 (2000)
72. Ma, Z.G., Yin, X.B., Yu, F.: A novel memory-based FFT architecture for real-valued signals based on a radix-2 decimation-in-frequency algorithm. IEEE Trans. Circuits Syst. II **62**(9), 876–880 (2015)

73. Macleod, M.D.: Multiplierless implementation of rotators and FFTs. EURASIP J. Appl. Signal Process. **2005**(17), 2903–2910 (2005)
74. Majumdar, M., Parhi, K.K.: Design of data format converters using two-dimensional register allocation. IEEE Trans. Circuits Syst. II **45**(4), 504–508 (1998)
75. Meher, P.K., Park, S.Y.: CORDIC designs for fixed angle of rotation. IEEE Trans. VLSI Syst. **21**(2), 217–228 (2013)
76. Meher, P.K., Valls, J., Juang, T.B., Sridharan, K., Maharatna, K.: 50 years of CORDIC: Algorithms, architectures, and applications. IEEE Trans. Circuits Syst. I **56**(9), 1893–1907 (2009)
77. Möller, K., Kumm, M., Garrido, M., Zipf, P.: Optimal shift reassignment in reconfigurable constant multiplication circuits. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (2017). Accepted for publication
78. Oh, J.Y., Lim, M.S.: New radix-2 to the 4th power pipeline FFT processor. IEICE Trans. Electron. **E88-C**(8), 1740–1746 (2005)
79. Paeth, A.W.: A fast algorithm for general raster rotation. In: Proc. Graphics Interface, pp. 77–81 (1986)
80. Parhi, K.K.: Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation. IEEE Trans. Circuits Syst. II **39**(7), 423–440 (1992)
81. Park, S.Y., Yu, Y.J.: Fixed-point analysis and parameter selections of MSR-CORDIC with applications to FFT designs. IEEE Trans. Signal Process. **60**(12), 6245–6256 (2012)
82. Püschel, M., Milder, P.A., Hoe, J.C.: Permuting streaming data using RAMs. J. ACM **56**(2), 10:1–10:34 (2009)
83. Qureshi, F., Gustafsson, O.: Generation of all radix-2 fast Fourier transform algorithms using binary trees. In: Proc. Europ. Conf. Circuit Theory Design, pp. 677–680 (2011)
84. Qureshi, F., Gustafsson, O.: Low-complexity constant multiplication based on trigonometric identities with applications to FFTs. IEICE Trans. Fundamentals **E94-A**(11), 324–326 (2011)
85. Reisis, D., Vlassopoulos, N.: Conflict-free parallel memory accessing techniques for FFT architectures. IEEE Trans. Circuits Syst. I **55**(11), 3438–3447 (2008)
86. Sánchez, M., Garrido, M., López, M., Grajal, J.: Implementing FFT-based digital channelized receivers on FPGA platforms. IEEE Trans. Aerosp. Electron. Syst. **44**(4), 1567–1585 (2008)
87. Serre, F., Holenstein, T., Püschel, M.: Optimal circuits for streamed linear permutations using RAM. In: Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, pp. 215–223. ACM (2016)
88. Shih, X.Y., Liu, Y.Q., Chou, H.R.: 48-mode reconfigurable design of SDF FFT hardware architecture using radix-3^2 and radix-2^3 design approaches. IEEE Trans. Circuits Syst. I **64**(6), 1456–1467 (2017)
89. Shukla, R., Ray, K.: Low latency hybrid CORDIC algorithm. IEEE Trans. Comput. **63**(12), 3066–3078 (2014)
90. Stone, H.: Parallel processing with the perfect shuffle. IEEE Trans. Comput. **C-20**(2), 153–161 (1971)
91. Takagi, N., Asada, T., Yajima, S.: Redundant CORDIC methods with a constant scale factor for sine and cosine computation. IEEE Trans. Comput. **40**(9), 989–995 (1991)
92. Takala, J., Järvinen, T.: Stride Permutation Access In Interleaved Memory Systems. Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation, S. Bhattacharyya, E. Deprettere and J. Teich. CRC Press (2003)
93. Takala, J., Jarvinen, T., Sorokin, H.: Conflict-free parallel memory access scheme for FFT processors. In: Proc. IEEE Int. Symp. Circuits Syst., vol. 4, pp. 524–527 (2003)
94. Tang, S.N., Tsai, J.W., Chang, T.Y.: A 2.4-GS/s FFT processor for OFDM-based WPAN applications. IEEE Trans. Circuits Syst. II **57**(6), 451–455 (2010)
95. Tsai, P.Y., Lin, C.Y.: A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling. IEEE Trans. VLSI Syst. **19**(12), 2290–2302 (2011)
96. Tummeltshammer, P., Hoe, J.C., Püschel, M.: Time-multiplexed multiple-constant multiplication. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **26**(9), 1551–1563 (2007)

97. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Trans. Electronic Computing **EC-8**, 330–334 (1959)
98. Voronenko, Y., Püschel, M.: Multiplierless multiple constant multiplication. ACM Trans. Algorithms **3**, 1–39 (2007)
99. Wang, J., Xiong, C., Zhang, K., Wei, J.: A mixed-decimation MDF architecture for radix- 2^k parallel FFT. IEEE Trans. VLSI Syst. **24**(1), 67–78 (2016)
100. Wang, Z., Liu, X., He, B., Yu, F.: A combined SDC-SDF architecture for normal I/O pipelined radix-2 FFT. IEEE Trans. VLSI Syst. **23**(5), 973–977 (2015)
101. Wenzler, A., Luder, E.: New structures for complex multipliers and their noise analysis. In: Proc. IEEE Int. Symp. Circuits Syst., vol. 2, pp. 1432–1435 (1995)
102. Wold, E., Despain, A.: Pipeline and parallel-pipeline FFT processors for VLSI implementations. IEEE Trans. Comput. **C-33**(5), 414–426 (1984)
103. Wu, C.S., Wu, A.Y.: Modified vector rotational CORDIC (MVR-CORDIC) algorithm and architecture. IEEE Trans. Circuits Syst. II **48**(6), 548–561 (2001)
104. Wu, C.S., Wu, A.Y., Lin, C.H.: A high-performance/low-latency vector rotational CORDIC architecture based on extended elementary angle set and trellis-based searching schemes. IEEE Trans. Circuits Syst. II **50**(9), 589–601 (2003)
105. Xia, K.F., Wu, B., Xiong, T., Ye, T.C.: A memory-based FFT processor design with generalized efficient conflict-free address schemes. IEEE Trans. VLSI Syst. **25**(6), 1919–1929 (2017)
106. Xing, Q., Ma, Z., Xu, Y.: A novel conflict-free parallel memory access scheme for FFT processors. IEEE Trans. Circuits Syst. II (2017)
107. Xudong, W., Yu, L.: Special-purpose computer for 64-point FFT based on FPGA. In: Proc. Int. Conf. Wireless Comm. Signal Process., pp. 1–3 (2009)
108. Yang, K.J., Tsai, S.H., Chuang, G.: MDC FFT/IFFT processor with variable length for MIMO-OFDM systems. IEEE Trans. VLSI Syst. **21**(4), 720–731 (2013)
109. Yang, L., Zhang, K., Liu, H., Huang, J., Huang, S.: An efficient locally pipelined FFT processor. IEEE Trans. Circuits Syst. II **53**(7), 585–589 (2006)
110. Yeh, W.C., Jen, C.W.: High-speed and low-power split-radix FFT. IEEE Trans. Signal Process. **51**(3), 864–874 (2003)
111. Yu, C., Yen, M.H.: Area-efficient 128- to 2048/1536-point pipeline FFT processor for LTE and mobile WiMAX systems. IEEE Trans. VLSI Syst. **23**(9), 1793–1800 (2015)
112. Zheng, W., Li, K.: Split radix algorithm for length 6^m DFT. IEEE Signal Process. Lett. **20**(7), 713–716 (2013)