Hardware-Efficient Twiddle Factor Generator for Mixed-Radix-2/3/4/5 FFTs

Citation

Year
2016

Version
Peer reviewed version (post-print)

Link to publication
TUTCRIS Portal (http://www.tut.fi/tutcris)

Published in
Proceedings IEEE International Workshop on Signal Processing Systems

DOI
10.1109/SiPS.2016.43

Take down policy
If you believe that this document breaches copyright, please contact tutcris@tut.fi, and we will remove access to the work immediately and investigate your claim.
Hardware-Efficient Twiddle Factor Generator for Mixed Radix-2/3/4/5 FFTs

Tomasz Patyk
Dolby Poland
Wroclaw, Poland
Email: tomasz.patyk@dolby.com

Fahad Qureshi
Tampere University of Technology
Tampere, Finland
Email: fahad@tut.fi

Jarmo Takala
Tampere University of Technology
Tampere, Finland
Email: jarmo.takala@tut.fi

Abstract—Twiddle factors are an integral part of FFT computations. Conventionally, they are either computed in run-time, hence increasing computational complexity, or pre-calculated and stored in RAM memory, which requires large memory footprint and increases power consumption. We created a systematic approach for designing digital circuits that generate twiddle factors based on reduced ROM tables. The approach supports radix-2, radix-3, radix-4, radix-5, and mixed radix-2/3/4/5 algorithms and several transform lengths. Number of complex twiddle factors stored in the memory equals only $\left\lfloor \frac{N_{\text{max}}}{8} \right\rfloor + 1$ for transform lengths up to $N_{\text{max}}$.

I. INTRODUCTION

In the year 1965, Cooley and Tukey presented their seminal paper [1] and brought the fast Fourier transform (FFT) algorithm to scientific and engineering communities’ attention. Their work, laid the groundwork for a fledgling discipline of the digital signal processing (DSP). Since then, the FFT along with digital filters constitute two most important classes of DSP algorithms [2].

The essence of FFT algorithms lies in the computational complexity reduction they offer. Cooley and Tukey [1] proved that for a given radix, complexity is proportional to $N \log_2 N$. Effectively, the quadratic $O(N^2)$ complexity of the discrete Fourier transform (DFT) computed by definition is reduced to the linearithmic complexity. Compared with the direct calculation of the DFT, FFT algorithms exploit two techniques in order to avoid redundant operations. First, by employing divide and conquer approach and reorganise data, a long DFT is recursively divided into shorter DFTs. This topic was extensively studied by many authors, including [1], [3], [4], [5], and generalised as the multidimensional index mapping by Burrus [6]. He divided FFT algorithms into two classes, namely prime (PFA) and common (CFA) factor algorithms. The second technique to exploit is the short length DFT optimisation. Most commonly used short length DFTs of size 2 and 4 (radix-2 and radix-4 butterflies) can be implemented with complex adders only, saving computational resources on not needed complex multipliers. Winograd [4] presented optimised DFTs of non-power-of-two sizes of 3, 5, and 7.

Historic FFT applications include spectral analysis, filter banks, convolutions, and many more [2]. More recently, FFT implementations became a technology of choice for the communication applications. Wideband digital communication networks, including 3G and 4G mobile communication networks, often use orthogonal frequency-division multiplexing (OFDM) as a method of encoding digital data on multiple carrier frequencies. Efficient OFDM modulators and demodulators can be implemented with the IFFT and FFT [7].

Over the years, radix-2 and other power-of-two algorithms were used predominantly, since efficient implementations, both hardware and software [8], existed. Recently, also non-power-of-two transform lengths have got attention. Non-power-of-two transform lengths are used by the Audio Spectral Front-End waveform codec [9], one of the coding tools used in the state-of-the-art audio codec Dolby AC-4 [10]. LTE uplink precoding [7] requires transform sizes of $12 - 1296$. Efficient implementations of mixed radix algorithms, including radices 3 and 5, are desirable. Challenges are yet to be overcome as non-power-of-two computations of odd radices are not trivial to implement on binary logic which we use today.

Twiddle factors are an integral part of FFT computations. Conventionally, they are either computed in run-time or pre-calculated and stored in RAM memory. Run-time generation methods include CORDIC [11], frequency synthesizers based on polynomials [12], and function generators based on recursive feedback difference equation [13]. These implementations decrease memory requirements at the cost of increased transistor switching activity resulting in higher power consumption. Alternatively, twiddle factors can be stored in a ROM table. Solutions using $\frac{N}{2}$ [14], $\frac{N}{4}$ [15], and $\frac{N}{8} + 1$ [16], [17] have been reported over the years. However, those methods support only radix-2, radix-4 and mixed radix-2/4 algorithms.

A twiddle factor generator unit could be used as a special function unit in an application-specific instruction-set processor (ASIP). However, if several instructions are needed to compute the correct index to the unit expected performance gains can be diminished. Therefore, it is crucial for the unit to perform index modifications to avoid additional instructions.

In this paper, we propose a systematic approach to design low-power twiddle factor units using reduced ROM tables. Proposed approach differs from the related work such that a) supports radix-2, radix-3, radix-4, radix-5, and mixed radix-2/3/4/5 FFT algorithms; b) supports several, both power-of-two and non-power-of-two transform sizes; c) integrates index manipulation. The scheme is generic as it supports any DFT
size which can be factorised to supported radices. The order and number of radices can be chosen arbitrarily since internal computations of the unit are performed in a binary coded mixed radix numeral system. Additionally, we present an implementation of a twiddle factor generator circuit (TFG). The implementation is based on pseudo-linear counter; binary coded mixed radix adder/subtractor, and constant multiplier; and a set of ROM tables to store twiddle factors. Each table combines a superset of twiddle factors common to a certain set of FFTs. The number of ROM tables depends on the variety of FFT sizes the TFG is designed to support. Small design corresponds to less silicon area, shorter critical path, and reduced power consumption. Latter is especially important for battery-powered portable devices where new communication standards are applicable.

The rest of the paper is organised as follows. Section II presents current state of the research in the field. Section III explains the novelty of the proposed twiddle factor index mapping. Finally, section IV concludes the paper.

II. RELATED WORK

Twiddle factors are an integral part of FFT computations. Conventionally, they are either computed in run-time or pre-calculated and stored in RAM memory. A number of algorithms have been used like CORDIC algorithms [11], polynomial-based approach [12], and recursive function generators [13].

Another approach is to reduce the twiddle factor memory. Because the twiddle factors are specific sets of angles generated by dividing the circumference \([0, 2\pi]\) in \(N\) equal parts. This leads to multiple symmetries in the complex plane. As a result, for an \(N\)-point FFT, only \(\frac{N}{2} + 1\) angles in the range of \([0, \frac{\pi}{2}]\), might need to be considered. The rest of the angles can be obtained by optionally swapping outputs and negating one or both outputs [19]. However, an additional hardware is required to generate the exponent \(nk\) of twiddle factor \(\omega_{nk}\). The traditional twiddle factor generators for power-of-two FFT in [20] are difficult to apply to non-power-of-two point FFT design. Because computation of exponent \(nk\) of twiddle factor \(\omega_{nk}\) is not identical to power-of-two FFT.

Few authors [21], [22] have used an \(N\)-size memory to store all the required twiddle factor coefficients which support the 3780-point FFT. However, the decomposition of FFT was carried in in way that one single twiddle factor multiplication appeared.

Chu Yu [23] and Sheng [24] proposed a variable length FFT architecture, which supports transform lengths of 128 – 2048/1536 point but no twiddle factor reduction technique.

The twiddle factor generator proposed in this paper supports all power-of-2/3/4/5 lengths unlike previous methods. The method has the same memory requirements as the previous methods.

III. PROPOSED TWIDDLE FACTOR GENERATOR

The DFT of an input vector \(X = [x_0, x_1, ..., x_{N-1}]^T\) is defined as a vector \(Y = [y_0, y_1, ..., y_{N-1}]^T\) such that:

\[
y_m = \sum_{n=0}^{N-1} \omega_{nk}^n x_n, \tag{1}
\]

or equivalently \(Y = F_N X\), where \(F_N\) is the \((N \times N)\)-matrix of DFT with entries:

\[
\omega_{nk}^n = \exp(-i2\pi nm/N). \tag{2}
\]

Many FFT algorithms have been derived for efficient computation of the DFT. In this paper, we use the in-place, decimation-in-time (DIT), mixed radix algorithm with permuted input, and in-order output. The formula for this FFT algorithm is given by (3):

\[
F_{p_i}^m = \prod_{s=m}^{1} \left( I_{p_i}^{m,s} \otimes (F_r \otimes I_{p_i}^{s-1}) \right) R_{p_i}^m, \tag{3}
\]

where \(I_N\) denotes an identity matrix of order \(N\), \(\otimes\) is a tensor product, \(m\) is the number of stages, \(s\) is the stage index in \(s \in (1, m)\), \(r_s\) is the radix of the \(s^{th}\) stage, and \(p_m^n\) is a product of radices of stages \(n\) to \(m\) given by:

\[
p_m^n = \begin{cases} 
1 & \text{if } n \leq m, \\
\prod_{i=n}^{m} r_i & \text{if } n > m.
\end{cases} \tag{4}
\]
TABLE I
MULTIPLICATION BY ROTATION OF THE EXPONENT OF W_60 TO OBTAIN W_480. DIGITS IN BOLD REPRESENT THE EXponent VALUE FOR A REPRESENTATION.

<table>
<thead>
<tr>
<th>60-point FFT</th>
<th>480-point FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-10</td>
<td>R-4/2/5/4/3</td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
</tr>
<tr>
<td>2</td>
<td>00002</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
</tr>
<tr>
<td>4</td>
<td>00000</td>
</tr>
<tr>
<td>5</td>
<td>00010</td>
</tr>
<tr>
<td>6</td>
<td>00020</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>19</td>
<td>00121</td>
</tr>
<tr>
<td>38</td>
<td>00302</td>
</tr>
</tbody>
</table>

The size of transform(s) calculated in the stage is equal to N = r * p_k. Given the sequence of indices and the size of transform, the required twiddle factors can be obtained with (2). Direct implementation of (10) requires 3 loops (modulo operation), multiplication, and cosine and sine functions.

In this paper, we present a hardware-efficient, customisable implementation of (9) designed to serve as a twiddle factor generation unit for an application-specific instruction-set processor (ASIP).

A. Binary-Coded Mixed Radix Iteration Counter

Proposed twiddle factor generator takes as input an iteration (linear) counter. The value of counter is used to extract index of the butterfly k and index of the input of butterfly i. The counter uses binary-coded mixed radix (BCMR) representation. BCMR encoding, unlike binary-coded decimal (BCD), uses n = ⌊log_2(r_s)⌋-bits to represent a numerical digit. Contrary to BCD, radix, and therefore number of bits representing digits in the BCMR, might be different for each numerical position.

For the linear index, radices assigned according to the order of radices in the FFT computations. For stage s ∈ (1, m), least significant digit of the counter indicates index i. Following s−1 digits represent index k.

When used together, BCMR countered is shared between the twiddle factor generator described in this paper, and operand access generator presented in [27], to compute mixed radix-2/3/4/5 FFTs.

B. Lookup Table Index

The twiddle factor generator stores required twiddle factors in one or more lookup tables (LUT), indexed with the exponent n of W_N. The exponent is given by (10) as a product of k and i, both obtained from the iteration counter. Since we know that i represents the index of the input of butterfly it must be smaller than the highest radix that our design supports. If the largest supported radix is 5, multiplication by

\[ T_{r_k} = \bigoplus_{i=0}^{k-1} D_N^{ki}, \]

\[ D_N^{ki} = \text{diag}(\omega_N^{0i}, \omega_N^{1i}, ..., \omega_N^{(k-1)i}), \]

where \( N = rk \) equals DFT size and \( \bigoplus \) denotes the matrix direct sum. Finally, \( T_{p_{i+1}m} = \bigoplus_{i=0}^{k-1} P_{p_{i+1}^{-1}, i} \).

\[ R_{p_{i+1}m} = \prod_{s=m}^1 (I_{p_{i+1}^s} \otimes P_{p_{i+1}^{-1}, i}^{s-1}), \]

based on the stride-by-K permutation matrix [25] of order of \( N \) defined as follows:

\[ [P_{N,K}]_{m,k} = \begin{cases} 1 & \text{if } k = (mK \mod N) + \lfloor mK/N \rfloor, \\ 0 & \text{otherwise.} \end{cases} \]

The signal flow graph of a 32-point mixed radix-4/4/2 FFT is illustrated in Figure 1. The algorithm uses in-place processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processed in stages, and overwritten by the result data. Thus, processing; input data is read through a stride permutation, processes...
a shift operation. This is not the case for non-power-of-two transform lengths. Factor \( a \) is not likely to be a power-of-two value. In binary representation a non-trivial multiplication would be required. In the BCRM numeral system, which we proposed earlier, this multiplication can be implemented efficiently as a rotation. Tables I and II give examples of index conversion from \( W_{60} \) to \( W_{480} \) and from \( W_{192} \) to \( W_{1152} \) respectively.

Table III presents OFDM related transform lengths organised into 7 LUTs. Each LUT contains FFTs that share common twiddle factors. A set corresponds to a ROM table, which stores \( \lceil \frac{N_{\max}}{8} \rceil + 1 \) twiddle factors, where \( N_{\max} \) is the longest transform of the set.

### D. Memory Reduction

The maximum number of unique twiddle factors, needed for an \( N \)-point FFT, depends on the highest radix \( r \) used in the computations, and can be as high as \( \lceil \frac{N}{r-1} \rceil N \). It was shown before for radix-2 [16] and mixed radix-4/2 [17] that the total number of twiddle factors to be stored to a ROM table can be reduced to \( tf_{\text{min}} = \lceil \frac{N_{\max}}{8} \rceil + 1 \). In this paper, we extend this approach to mixed radix-2/3/4/5 FFT. The minimum number of unique twiddle factors for a \( N_{\max} \)-point transform can be obtained with:

\[
\text{if } N_{\max} \text{ mod } 2 = 1 \text{ then } \quad \text{tf}_{\text{min}} \leftarrow \left\lceil \frac{N_{\max}}{8} \right\rceil + 1 \quad \text{▷ Group 1} \\
\text{else} \quad \text{tf}_{\text{min}} \leftarrow \left\lceil \frac{N_{\max}}{8} \right\rceil + 1 \quad \text{▷ Group 2} \\
\text{tf}_{\text{min}} \leftarrow \left\lfloor \frac{N_{\max}}{4} \right\rfloor + 1 \quad \text{▷ Group 3} \\
\text{end if}
\]

Listed in Table III transform sizes, commonly used in OFDM applications, belong to Group 2. Therefore, significant memory savings can be achieved. It can be observed that by reordering the coefficients into 8 sets, \( S_0, S_1, \ldots, S_7 \), all the twiddle factors can be obtained from the coefficients of the set \( S_0 \). Since we need to support transform sizes up to \( N_{\max} \), we need to store only \( \left\lfloor \frac{N_{\max}}{8} \right\rfloor + 1 \) complex coefficients into the remaining twiddle factors can be obtained from the table \( C = (C_0, C_1, \ldots, C_{\frac{N_{\max}}{8}}) \) by \( C_n = W_n^N \).
operations including negation, and real and imaginary part swap. This operation is given by the formula:

\[ W_{N_{\text{max}}}^{n} = \begin{cases} 
C_n & n \leq \left\lfloor \frac{N_{\text{max}}}{8} \right\rfloor, \\
-jC_n^{\ast} & \left\lfloor \frac{2N_{\text{max}}}{8} \right\rfloor < n \leq \left\lfloor \frac{N_{\text{max}}}{8} \right\rfloor, \\
-jC_{n-2N_{\text{max}}}^{\ast} & \left\lfloor \frac{3N_{\text{max}}}{8} \right\rfloor < n \leq \left\lfloor \frac{2N_{\text{max}}}{8} \right\rfloor, \\
-jC_{n-4N_{\text{max}}}^{\ast} & \left\lfloor \frac{4N_{\text{max}}}{8} \right\rfloor < n \leq \left\lfloor \frac{3N_{\text{max}}}{8} \right\rfloor, \\
-jC_{n-6N_{\text{max}}}^{\ast} & \left\lfloor \frac{5N_{\text{max}}}{8} \right\rfloor < n \leq \left\lfloor \frac{4N_{\text{max}}}{8} \right\rfloor, \\
-jC_{n-8N_{\text{max}}}^{\ast} & \left\lfloor \frac{6N_{\text{max}}}{8} \right\rfloor < n \leq \left\lfloor \frac{5N_{\text{max}}}{8} \right\rfloor, \\
-jC_{n-10N_{\text{max}}}^{\ast} & \left\lfloor \frac{7N_{\text{max}}}{8} \right\rfloor < n, \\
\end{cases} \]

(11)

where \( C_n^{\ast} \) is the complex conjugate of \( C_n \).

E. Example

The final algorithm of the twiddle factor generator looks as follows:

1) Obtain index \( k \) of the butterfly;
2) Obtain index \( i \) of the input of the butterfly;
3) Multiply \( n_s = k i \) as for \( \omega_{N_{s}}^{n} \);
4) Select LUT based on \( N_{\text{max}} \) to which given transform length belongs;
5) Multiply \( n_{\text{max}} = n_s N_{\text{max}} \);
6) Map \( n_{\text{max}} \) to \( N_{\text{max}} \) of the unity circle to get \( n_{\text{min}} \);
7) Add \( n_{\text{min}} \) to the base address of ROM table and fetch coefficient from the memory;
8) Reconstruct twiddle factor from the fetched coefficient.

The example of 30-point mixed radix-5/3/2 FFT is given in Figure 2. Table IV presents selected steps of the twiddle factor generation for 30-point FFT with ROM table of coefficients for 90-point FFT (R-10 values are only for illustration purposes and do not correspond to hardware operations).
generation algorithm. The external BCMR iteration counter is generating index value. This liner index value is used by the twiddle factor generator and operand address generator [27] to generate control signals. Twiddle factor generator extracts $i$ and $k$. The latter, is converted with an internal BCMR counter to a different radix representation. This is needed to substitute in the following step multiplication with a rotation. Converted $k$ is multiplied by $i$ by a mixed radix constant multiplier to obtain $n_s$. $n_s$ is rotated to obtain $n_{max}$, which is converted with the help of comparator and subtractor to $n_{min}$. $n_{min}$ indicates a coefficient in a ROM table.

Those twiddle factors are presented in column $C_0$ of Table V. Coefficients from columns $C_1$ and $C_2$ are reconstructed from the coefficient of column $C_0$ by using following formula:

$$W_{N_{max}}^n = \begin{cases} 
C_n, & n \leq \frac{2N_{max}}{8}, \\
C_{n-4N_{max}}^{\ast}, & \frac{2N_{max}}{8} < n \leq \frac{6N_{max}}{8}, \\
C_{n-4N_{max}}^{\ast}, & \frac{6N_{max}}{8} < n, 
\end{cases}$$

(12)

where $C_n^{\ast}$ is the complex conjugate of $C_n$.

IV. CONCLUSION

In this paper, we present a hardware-efficient, customisable implementation designed to serve as a twiddle factor generation unit for an application-specific instruction-set processor (ASIP). The generator is a stand alone design with clearly defined interfaces, where all the required index modifications are performed inside the unit. The unit generates twiddle factors based on 3 input parameters: index from an iteration (linear) counter, array of radices this counter uses, and maximum supported transform size $N_{max}$. The proposed approach supports several mixed radix transform sizes and uses precomputed twiddle factors stored in reduced ROM table(s). Memory reduction scheme is used to reduce number of stored twiddle factors to $\left\lfloor \frac{N_{max}}{8} \right\rfloor + 1$ and reconstruct remaining ones from those stored.

ACKNOWLEDGMENT

This work is partially supported by Tekes - the Finnish Funding Agency for Innovation under funding decision 40142/14 (StreamPro) in the FiDiPro program.

REFERENCES


