Low-Power Split-Radix FFT Processors Using Radix-2 Butterfly Units

Zhuo Qian and Martin Margala

***Abstract*— Split-radix fast Fourier transform (SRFFT) is an ideal candidate for the implementation of a low-power FFT processor, because it has the lowest number of arithmetic operations among all the FFT algorithms. In the design of such processors, an efficient addressing scheme for FFT data as well as twiddle factors is required. The signal flow graph of SRFFT is the same as radix-2 FFT, and therefore, the conventional address generation schemes of FFT data could also be applied to SRFFT. However, SRFFT has irregular locations of twiddle factors and forbids the application of radix-2 address generation methods. This brief presents a shared-memory low-power SRFFT processor architecture. We show that SRFFT can be computed by using a modified radix-2 butterfly unit. The butterfly unit exploits the multiplier-gating technique to save dynamic power at the expense of using more hardware resources. In addition, two novel address generation algorithms for both the trivial and nontrivial twiddle factors are developed. Simulation results show that compared with the conventional radix-2 shared-memory implementations, the proposed design achieves over 20% lower power consumption when computing a 1024-point complex-valued transform.**

***Index Terms*— Address generation, low power, radix-2, split-radix fast Fourier transform (SRFFT), twiddle factors.**

I. Introduction

The fast Fourier transform (FFT) is one of the most important and fundamental algorithms in the digital signal processing area. Since the discovery of FFT, many variants of the FFT algorithm have been developed, such as radix-2 and radix-4 FFT. In 1984, Duhamel and Hollmann [1] proposed a new variant of FFT algorithm called split-radix FFT (SRFFT). Their algorithm requires the least number of multiplications and additions among all the known FFT algorithms. Since arithmetic operations significantly contribute to overall system power consumption, SRFFT is a good candidate for the implementation of a low-power FFT processor.

In general, all the FFT processors can be categorized into two main groups: pipelined processors or shared-memory processors. Examples of pipelined FFT processors can be found in [2] and [3]. A pipelined architecture provides high throughputs, but it requires more hardware resources at the same time. One or multiple pipelines are often implemented, each consisting of butterfly units and control logic. In contrast, the shared-memory-based architecture requires the least amount of hardware resources at the expense of slower throughput. Examples of such processors can be found in [4] and [5]. In the radix-2 shared-memory architecture, the FFT data are organized into two memory banks. At each clock cycle, two FFT data are provided by memory banks and one butterfly unit is used to process the data. At the next clock cycle, the calculation results are written back to the memory banks and replace the old data. The scope of this brief is limited to the shared-memory architecture.

In the shared-memory architecture, an efficient addressing scheme for FFT data as well as coefficients (called twiddle factors) is required. For the fixed-radix FFT, previous works of this topic can be found in [5] and [6]. For split-radix FFT, it conventionally involves an L-shaped butterfly datapath whose irregular shape has uneven latencies and makes scheduling difficult. In this brief, we show that the SRFFT can be computed by using a modified radix-2 butterfly structure. Our contribution consists of mapping the split-radix FFT algorithm to the shared-memory architecture, leveraging the lower multiplicative complexity of the algorithm to reduce the dynamic power and developing two novel twiddle factor addressing schemes for the split-radix FFT.

The rest of this brief is organized as follows. Section II provides a theoretical comparison of the number of complex multiplications between the radix-2 FFT and the SRFFT. Section III discusses the architecture of the proposed design. Section IV provides the implementation results and Section V concludes this brief.

II. Comparison of SRFFT and Radix-2 FFT

The N-point discrete Fourier transform is defined by

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \tag{1}$$

where $k = 0, 1, \ldots, N-1$ and $W_N^{nk} = e^{-j2\pi nk/N}$. If we split $X(k)$ into even and odd terms, radix-2 FFT can be derived as

$$X(2k) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)] W_{N/2}^{nk} \tag{2}$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} [x(n) - x(n+N/2)] W_N^n W_{N/2}^{nk}. \tag{3}$$

The basic idea behind the SRFFT is the application of a radix-2 index map to the even-index terms and a radix-4 map to the odd-index terms. For the even-index terms, it can be decomposed as (2). For the odd-index terms, it can be decomposed as

$$X(4k+1) = \sum_{n=0}^{N/4-1} [x(n) - x(n+N/2) - j(x(n+N/4) - x(n+3N/4))] W_N^n W_{N/4}^{nk} \tag{4}$$

$$X(4k+3) = \sum_{n=0}^{N/4-1} [x(n) - x(n+N/2) + j(x(n+N/4) - x(n+3N/4))] W_N^n W_{N/4}^{nk} \tag{5}$$

where $k = 0, 1, \ldots, N/4$. The formulas above result in the L-shaped split-radix butterfly structure, which can be found in [2] and the scheduling of the L-shaped butterfly is irregular.

Assume that we have $N = 2^S$ point FFT, both SRFFT and radix-2 FFT require S passes to finish the computation, as shown in Figs. 1 and 2. For SRFFT, the total number of the L butterflies N_{SR} is given by [2]

$$N_{SR} = [(3S-2)2^{S-1} + (-1)^S]/9. \tag{6}$$

Manuscript received August 23, 2015; revised November 28, 2015 and January 30, 2016; accepted March 4, 2016.

The authors are with the Department of Electrical and Computer Engineering, University of Massachusetts Lowell, Lowell, MA 01851 USA (e-mail: zhuo_qian@student.uml.edu; martin_margala@uml.edu).

Digital Object Identifier 10.1109/TVLSI.2016.2544838

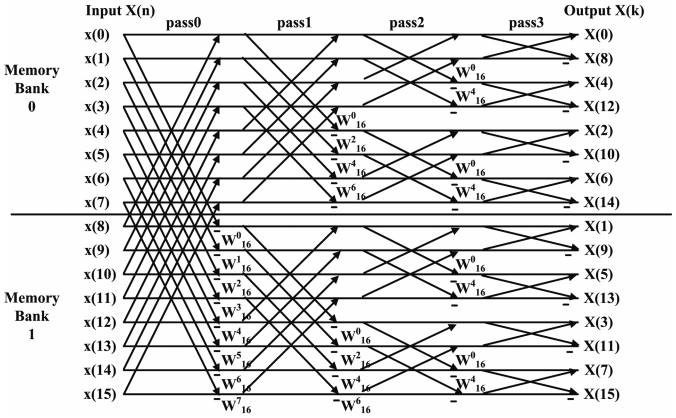

Fig. 1. Signal flow graph for radix-2 FFT.

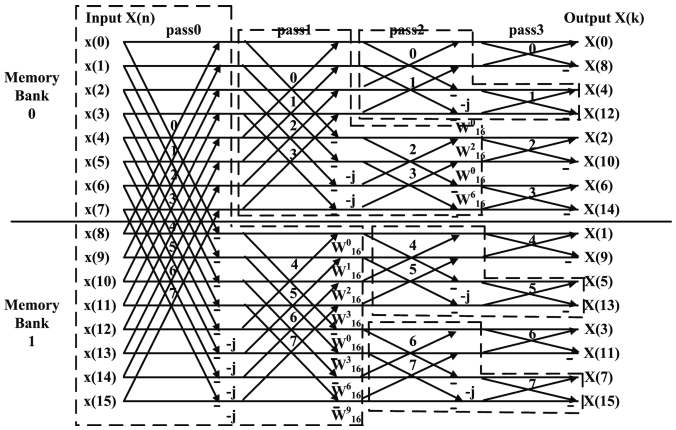

Fig. 2. Signal flow graph for SRFFT.

Each L butterfly contains two nontrivial complex multiplications, and therefore, the total number of nontrivial complex multiplications M_{SR} in SRFFT is

$$M_{\mathrm{SR}} = [(3S-2)2^{S-1} + (-1)^S]2/9. \tag{7}$$

In the $(S-1)$th pass, the number of SR butterfly N_{S-1} is

$$N_{S-1} = [2 + (-1/2)^{S-2}]N/12\,. \tag{8}$$

However, in the $(S-1)$th pass, each L butterfly does not contain any nontrivial twiddle factors and hence, the total number of nontrivial multiplications M'_{SR} in SRFFT is

$$M'_{\mathrm{SR}} = M_{\mathrm{SR}} - 2N_{S-1}. \tag{9}$$

For the conventional radix-2 FFT, the total number of complex multiplications M_{R2} is

$$M_{R2} = 2^{S-1}(S-1). \tag{10}$$

III. Hardware Implementation

A. Shared-Memory Architecture

The architecture of shared-memory processor is shown in Fig. 3. The FFT data and the twiddle factors are stored in the RAM and ROM banks, respectively. We observed that the flow graph of split-radix algorithm is the same as radix-2 FFT except for the locations and values of the twiddle factors and therefore, the conventional radix-2 FFT data address generation schemes could also be applied to SRFFT (RAM address generator). However, the mixed-radix property

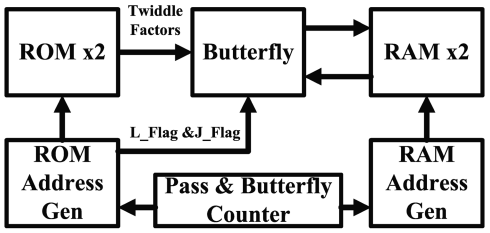

Fig. 3. Shared-memory architecture.

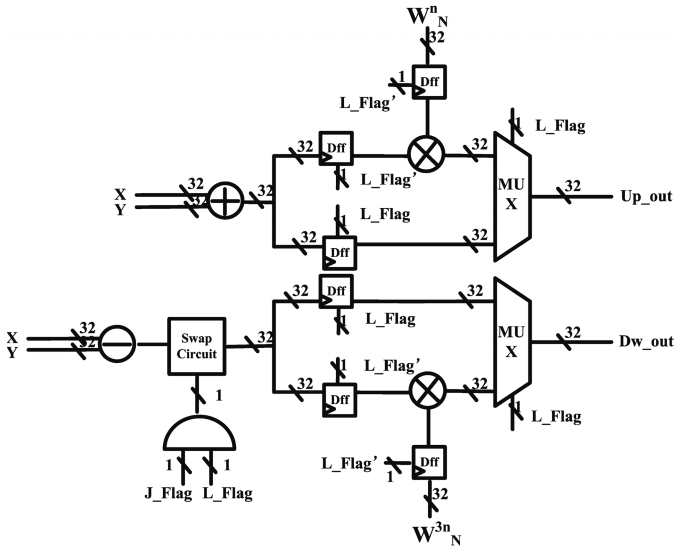

Fig. 4. Modified butterfly structure.

of SRFFT algorithm leads to the irregular locations of twiddle factors and forbids any conventional address generation algorithm (ROM address generator).

B. Modified Radix-2 Butterfly Unit

In our previous work [7], we proposed a modified butterfly unit which is shown in Fig. 4. The structure of this butterfly unit is determined by the fact that the SRFFT has multiplications of both the upper and lower legs. To prevent unnecessary switching activity, we put the clock gating registers in the multiplier path and a few registers are placed at the address port of memory banks to synchronize the whole design. The key to use this architecture is the knowing about which butterflies require no multiplications (the complex multipliers are then skipped), trivial multiplications (swapping), and nontrivial multiplications (using complex multipliers). In Section III-C, we present an efficient algorithm to solve this problem.

C. Address Generation of Twiddle Factors

The flow graph for the 16-point SRFFT is shown in Fig. 2. In Fig. 2, there are two kinds of twiddle factors: j and W_n. For those multiplications involving j is called trivial multiplications, because these operations are essentially the swapping of the real and imaginary part of the multiplier, hence no multiplication is involved. For those multiplications involving W_n are called nontrivial multiplications, because complex multipliers are used to complete these operations. In Fig. 2, each area surrounded by the dashed lines is called one L block which is formed by L butterflies in each pass [8] and there are totally five L blocks for a 16-point SRFFT.

```
Initialization: set all the L_Flag to 1
for P = 0 to S − 1 do
  for B = 0 to 2^(S−1) do
    J_Flag ← b_(S−2−P)
    if L_Flag_b = 1 then
      if J_Flag = 1 then
        Multiply trivial twiddle factor j (swapping)
        ROM_Address ← don'tcare
        L_Flag_b ← 0
      else
        No Multiplication Required
        ROM_Address ← don'tcare
        L_Flag_b ← 1
      end if
    else
      Multiply non-trivial twiddle factor W_n
      L_Flag_b ← 1
      ROM_Address
      ← b_(S−2−P) b_(S−3−P) ... b_0 0 ... 0
    end if
  end for
end for
```

Fig. 5. Pseudocode for tracking trivial and nontrivial twiddle factors.

Given $N = 2^S$ point FFT data, we first have the following definitions.

1) *Butterfly Counter B:* $(S-1)$-bit counter that indicates, in each pass, which butterfly is currently under operation.
2) *Pass Counter P:* $(\lceil \log_2 S \rceil)$-bit counter that indicates which pass is currently under operation.
3) *L_Flag:* A set of variables indicate if the butterfly under operation is in the L-shaped block. Each variable corresponds to one butterfly in each pass and the number of such variables is the same as the number of radix-2 butterflies in each pass. For example, in Fig. 2, each pass contains eight butterflies so eight L_Flags are required.
4) *J_Flag:* A variable indicates if the butterfly under operation should multiply the trivial twiddle factor j (swapping).

In Fig. 2, we have made two observations. First, in the current pass, if the ith butterfly is not within the L block (L_Flag $= 0$), in the next pass, the same i'th butterfly will be definitely within the L block. For example, butterflies 100, 101, 110, and 111 are not within the L block in pass 1 (because they belong to the L block in pass 0) and butterflies 100, 101, 110, and 111 are within the L block in pass 2.

The second observation is that if the butterfly is within the L block in this pass (L_Flag $= 1$), in the next pass, whether it will be within the L block is determined by J_Flag. If, in the current pass, the ith butterfly needs to multiply j (J_Flag $= 1$), then in the next pass, the same butterfly needs to multiply W_n. For example, butterflies 100, 101, 110, and 111 are within the L block in pass 0, and they need to multiply by j; in pass 1, the butterflies 100, 101, 110, and 111 need to multiply W_n. On the other hand, if, in the current pass, the ith butterfly does not need to multiply j (J_Flag $= 0$), then in the next pass, the same butterfly still belongs to L block and does not need to multiply W_n. For example, butterflies 000, 001, 010, and 011 are within the L block in pass 0 and they do not need to multiply by j (J_Flag $= 0$), in pass 1, the butterflies 000, 001, 010, and 011 still belong to the L block and do not need to multiply W_n.

The high level structure of the proposed algorithm is shown in Fig. 5. All L_Flag are set to one before the program starts, because all the butterflies in pass 0 are contained in the first L block.

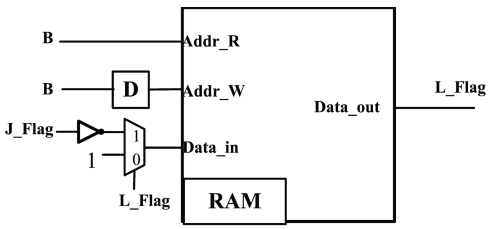

Fig. 6. L_Flag structure.

TABLE I
ROM CONFIGURATION FOR A 16-POINT SRFFT

ROM0		ROM1	
Address	Twiddle Factors	Address	Twiddle Factors
00	W_{16}^{0}	00	W_{16}^{0}
01	W_{16}^{1}	01	W_{16}^{3}
10	W_{16}^{2}	10	W_{16}^{6}
11	W_{16}^{3}	11	W_{16}^{9}

TABLE II
ADDRESS GENERATION TABLE OF THE PROPOSED ALGORITHM 1 FOR A 16-POINT SRFFT

Pass 00				
Butterfly	L_Flag	J_Flag	ROM0 Addr	ROM1 Addr
000	1	0	XX	XX
001	1	0	XX	XX
010	1	0	XX	XX
011	1	0	XX	XX
100	1	1	XX	XX
101	1	1	XX	XX
110	1	1	XX	XX
111	1	1	XX	XX
Pass 01				
Butterfly	L_Flag	J_Flag	ROM0 Addr	ROM1 Addr
000	1	0	XX	XX
001	1	0	XX	XX
010	1	1	XX	XX
011	1	1	XX	XX
100	0	0	00	00
101	0	0	01	01
110	0	1	10	10
111	0	1	11	11
Pass 10				
Butterfly	L_Flag	J_Flag	ROM0 Addr	ROM1 Addr
000	1	0	XX	XX
001	1	1	XX	XX
010	0	0	00	00
011	0	1	10	10
100	1	0	XX	XX
101	1	1	XX	XX
110	1	0	XX	XX
111	1	1	XX	XX
Pass 11				
Butterfly	L_Flag	J_Flag	ROM0 Addr	ROM1 Addr
000	1	0	XX	XX
001	1	0	XX	XX
010	1	0	XX	XX
011	1	0	XX	XX
100	1	0	XX	XX
101	1	0	XX	XX
110	1	0	XX	XX
111	1	0	XX	XX

L_Flags could be efficiently implemented as an RAM block, as shown in Fig. 6. Butterfly counter B is served as the read address of RAM and at each clock cycle, the corresponding L_Flag value for the current butterfly is provided. The updated L_Flag value for the next pass is written to this memory at next clock cycle. The size of this memory is 2^{S-1} bits, which equals to the number of butterflies in each pass. Such a size is trivial on the modern field-programmable gate array (FPGA). For example, for a 2048-point FFT only 1 kbit is required.

J_Flag is a combinational signal. The value of this variable depends on the butterfly counter B. In the P_{th} pass, J_Flag equals to

$$b_{S-2-P}. \tag{11}$$

TABLE III
IMPLEMENTATION RESULTS ON SPARTAN-6 FPGA

FFT Size	Design	FFs	LUTs	DRAMs	BRAMs	DSPs	Max Freq. (MHz)	Static Pow.(mW)	Dynamic Pow.(mW)
256	Cohen[5]	26	220	-	3	4	101.17	13.89	25.54
	Xiao[6]	29	224	-	3	4	102.09	13.90	24.10
	Proposed	160	466	4	3	6	101.37	13.95	16.86
512	Cohen[5]	30	235	-	3	4	101.17	13.91	27.03
	Xiao[6]	32	229	-	3	4	101.57	13.91	26.86
	Proposed	164	462	8	3	6	100.17	13.95	17.44
1024	Cohen[5]	33	233	-	3	4	100.10	13.91	29.55
	Xiao[6]	34	234	-	3	4	101.22	13.91	28.08
	Proposed	166	468	16	3	6	100.33	13.95	20.01

TABLE IV
COMPARISON OF EACH COMPONENT FOR A 1024-POINT FFT ON SPARTAN-6 FPGA

Components	Xiao[6]						Proposed					
	FFs	LUTs	DRAMs	BRAMs	DSPs	Power(mW)	FFs	LUTs	DRAMs	BRAMs	DSPs	Power(mW)
RAM	-	94	-	2	-	10.34	-	128	-	2	-	10.91
Butterfly	-	96	-	-	4	11.77	126	288	-	-	6	4.71
ROM	-	-	-	1	-	4.50	-	-	-	1	-	1.88
Counter	13	14	-	-	-	0.60	13	14	-	-	-	1.27
RAM Addr Gen	21	12	-	-	-	0.42	21	12	-	-	-	0.76
ROM Addr Gen	-	18	-	-	-	0.45	6	26	16	-	-	0.49

In the last pass, L_Flag is set to one and J_Flag is set to zero.

When nontrivial multiplication is required, twiddle factors need to be retrieved from the ROM banks. Unlike conventional method that stores all the W_n in one ROM bank, we organize W_n in two ROM banks: one stores W_n for the upper leg of the butterfly unit and the other stores those for the lower leg of the butterfly unit. Table I shows the content of the two ROM banks for a 16-point SRFFT. Started in pass 1, in the P_{th} pass the address of each ROM bank is given by

$$b_{S-2-P}b_{S-3-P}\ldots b_0 0\ldots 0 \text{ (following } (P-1) \text{ '0' s).} \quad (12)$$

It is worth mentioning that in conventional implementations, the twiddle factors are required for each butterfly so ROM banks are always enabled, and in our implementation, the L_Flag signal can be used as the enable signal for the ROM banks, since that if the butterfly belongs to the L block, no multiplication is required. This could lead to further power reduction. Table II shows an example of the proposed algorithm for the 16-point SRFFT.

Both the address generation of FFT data and twiddle factors depends on certain butterfly processing order in each pass. Other than Xiao's [6] method (Fig. 2), there are other methods of ordering the butterfly sequence, such as [5]. We have also developed the address generation methods for this kind of butterfly sequence using the similar ideas stated above. The details are not discussed here and we only give the conclusions. In each pass except for the last one, J_Flag equals to

$$b_{S-2}. \quad (13)$$

The address for each ROM bank is given by

$$b_{S-2}b_{S-3}\ldots b_1. \quad (14)$$

IV. IMPLEMENTATION AND RESULTS

The proposed design is compared with the two conventional shared-memory architectures. Our two proposed address generation algorithms of the twiddle factors are similar and therefore, we only implement algorithm 1, which is within the ROM address generator. The address generation of RAM is based on [6] and datapath width is 32 b. The three FFTs are synthesized under the constraint of 100 MHz in Xilinx ISE 14.7 targeting for Spartan-6 XC6SLX4 device. The simulation results are shown in Table III. Power is measured by Xilinx XPower analyzer using the switching activity interchange format file recorded in a sufficient long simulation time. For a 1024-point FFT, the proposed algorithm could achieve over 20% lower power but almost maintains the static power consumption. In the given architecture, when the FFT size increases, a larger RAM and ROM size is required, but the butterfly unit does not change. The limitation of the proposed design is the large resources used in the butterfly unit. This limitation could possibly be removed by using different butterfly structures for additions and multiplications. Table IV shows the power consumption of each component for a 1024-point FFT. The reduction of dynamic power consumption is due to the fact that multipliers and ROM banks are enabled only when necessary.

We have also synthesized the design using OSU gscl45 nm library in Cadence RTL compiler. All the three FFTs are able to run above 200 MHz. The library does not have a memory intellectual property, and the memories are constructed using basic cells and flip-flops. The implementation results of a 1024-point FFT are shown in Table V. A large number of cells are used to implement the memory banks, which become the most power hungry component in the design.

Compared with the radix-2 addressing schemes in [5] and [6], our addressing method requires additional 2^{S-1}-bit memory. However, the SRFFT algorithm has the irregular signal flow graph and makes the control of such processors more difficult than the fixed-radix ones. Although a software solution for the indexing problem has been given in [9], the indexing scheme is designed for the L butterfly structure, which is not suitable for the hardware implementation due to its uneven latencies. Some previous works such as [10] use lookup tables to solve the indexing problem. It is obvious that the proposed algorithm requires significantly less memory than the lookup table approach.

Compared with a pipelined SRFFT architecture such as split-radix single-path delay feedback (SRSDF) given in [11], the shared-memory architecture offers significantly reduced hardware cost and power consumption at the expense of slower throughput. For an N-point FFT, SRSDF requires $\log_4 N - 1$ multipliers and $4\log_4 N$ adders. In contrast, only two multipliers and two adders are used in

TABLE V
COMPARISON OF EACH COMPONENT FOR A 1024-POINT FFT USING gscl45-nm TECHNOLOGY AT 100 MHz AND 1.1 V

Components	Xiao[6]		Cohen[5]		Proposed	
	Cell Count	Power(mW)	Cell Count	Power(mW)	Cell Count	Power(mW)
RAM	187938	76.14	187682	74.80	187938	73.51
Butterfly	3398	8.52	3391	8.09	6041	1.34
ROM	93841	31.12	93841	30.85	93812	26.01
Counter	58	0.05	58	0.05	60	0.05
RAM Addr Gen	100	0.08	176	0.09	98	0.08
ROM Addr Gen	69	0.01	69	0.01	2584	1.20

the proposed architecture. In addition, in order to arrange the different butterfly structures for different operations, SRSDF still needs to track the trivial and nontrivial multiplications, and the indexing scheme is much more complicated than the proposed one, since an additional encoding step (bit-inverse and bit-reverse) is applied to the butterfly sequences.

V. CONCLUSION

In this brief, a shared-memory-based SRFFT processor is proposed. The proposed method reduces the dynamic power consumption at the expense of more hardware resources. We also present two addressing schemes for both the trivial and nontrivial twiddle factors. Since SRFFT has the minimum number of multiplications compared with other types of FFT, the results could be more optimal in the sense of floating point operations.

REFERENCES

[1] P. Duhamel and H. Hollmann, "'Split radix' FFT algorithm," *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, Jan. 1984.

[2] M. A. Richards, "On hardware implementation of the split-radix FFT," *IEEE Trans. Acoust., Speech Signal Process.*, vol. 36, no. 10, pp. 1575–1581, Oct. 1988.

[3] J. Chen, J. Hu, S. Lee, and G. E. Sobelman, "Hardware efficient mixed radix-25/16/9 FFT for LTE systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 2, pp. 221–229, Feb. 2015.

[4] L. G. Johnson, "Conflict free memory addressing for dedicated FFT hardware," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 39, no. 5, pp. 312–316, May 1992.

[5] D. Cohen, "Simplified control of FFT hardware," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 24, no. 6, pp. 577–579, Dec. 1976.

[6] X. Xiao, E. Oruklu, and J. Saniie, "An efficient FFT engine with reduced addressing logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 55, no. 11, pp. 1149–1153, Nov. 2008.

[7] Z. Qian, N. Nasiri, O. Segal, and M. Margala, "FPGA implementation of low-power split-radix FFT processors," in *Proc. 24th Int. Conf. Field Program. Logic Appl.*, Munich, Germany, Sep. 2014, pp. 1–2.

[8] A. N. Skodras and A. G. Constantinides, "Efficient computation of the split-radix FFT," *IEE Proc. F-Radar Signal Process.*, vol. 139, no. 1, pp. 56–60, Feb. 1992.

[9] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, "On computing the split-radix FFT," *IEEE Trans. Acoust., Speech Signal Process.*, vol. 34, no. 1, pp. 152–156, Feb. 1986.

[10] J. Kwong and M. Goel, "A high performance split-radix FFT with constant geometry architecture," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Dresden, Germany, Mar. 2012, pp. 1537–1542.

[11] W.-C. Yeh and C.-W. Jen, "High-speed and low-power split-radix FFT," *IEEE Trans. Signal Process.*, vol. 51, no. 3, pp. 864–874, Mar. 2003.