한국정보전자통신기술학회논문지(jkiiect)'20-2, Vol.13 No.1
논문 20-13-01-35 http://dx.doi.org/10.17661/jkiiect.2020.13.1.35

연산복잡도가 적은 radix-2^6 FFT 프로세서

조경주*

Novel Radix-2^6 DF IFFT Processor with Low Computational Complexity

Kyung-Ju Cho*

요 약 FFT(fast Fourier transform) 프로세서는 통신, 영상, 생체 신호처리와 같은 다양한 응용에 폭 넓게 사용된다. 특히, 고성능 저전력 FFT 연산은 OFDM 전송방식을 사용하는 통신시스템에서는 필수적이다. 본 논문에서는 연산복잡도가 적고 하드웨어 효율이 우수한 새로운 radix-26 FFT 알고리즘을 제안한다. 7차원 인덱스 매핑을 사용하여 회전인자를 분해하고 radix-26 FFT 알고리즘을 유도한다. 제안한 알고리즘은 기존 알고리즘과 비교하여 회전인자가 간단하고 복소 곱셈 수가 적어 회전인자를 저장하는 메모리 크기를 줄일 수 있다. 한 스테이지에서 회전인자의 계수가 적을 때 복소 곱셈기 대신 복소 상수곱셈기를 사용하면 복소곱셈을 효율적으로 처리할 수 있다. 복소 상수곱셈기는 CSD(canonic signed digit)과 CSE(common subexpression elimination) 알고리즘을 사용하여 보다 효율적으로 설계할 수 있다. 제안한 radix-26 알고리즘에서 필요한 복소 상수곱셈기를 CSD와 CSE를 이용하여 효율적으로 설계하는 방법을 제안한다. 제안한 방법의 성능을 평가하기 위해 SDF(single-path delay feedback) 구조를 사용하여 256 포인트 FFT를 설계하고 FPGA로 합성한 결과, 제안한 알고리즘은 기존 알고리즘 보다 약 10% 정도 하드웨어를 적게 사용하였다.

Abstract Fast Fourier transform (FFT) processors have been widely used in various application such as communications, image, and biomedical signal processing. Especially, high-performance and low-power FFT processing is indispensable in OFDM-based communication systems. This paper presents a novel radix-26 FFT algorithm with low computational complexity and high hardware efficiency. Applying a 7-dimensional index mapping, the twiddle factor is decomposed and then radix-26 FFT algorithm is derived. The proposed algorithm has a simple twiddle factor sequence and a small number of complex multiplications, which can reduce the memory size for storing the twiddle factor. When the coefficient of twiddle factor is small, complex constant multipliers can be used efficiently instead of complex multipliers. Complex constant multipliers can be designed more efficiently using canonic signed digit (CSD) and common subexpression elimination (CSE) algorithm. An efficient complex constant multiplier design method for the twiddle factor multiplication used in the proposed radix-26 algorithm is proposed applying CSD and CSE algorithm. To evaluate performance of the previous and the proposed methods, 256-point single-path delay feedback (SDF) FFT is designed and synthesized into FPGA. The proposed algorithm uses about 10% less hardware than the previous algorithm.

Key Words : FFT, Radix-26 algorithm, Twiddle factor, SDF, Complex constant multiplier, CSD

1. 서 론

FFT 프로세서는 통신, 영상, 생체 신호처리와 같은 다양한 응용에 폭 넓게 사용된다. 예를 들면, 고성능 저전력 FFT는 OFDM 전송방식을 사용하는 5G를 포함한 IEEE 802.11a/g/n, IEEE 802 11n, LTE와 같은 통신 시스템에서는 필수적이다[1][2]. 따라서 FFT 프로세서

*Department of Electronic Engineering Wonkwang University
Received February 04, 2020 Revised February 18, 2020 Accepted February 18, 2020

의 성능을 향상시키기 위한 많은 연구가 진행되었으며, 주로 FFT 알고리즘과 구조, 메모리 크기와 연산복잡도 감소에 초점을 맞춰 연구되고 있다.

FFT의 설계에는 radix-2^k(k=1~5)과 radix-4, mixed-radix 알고리즘 등이 사용된다[3]-[7]. 이를 이용하여 다양한 구조가 개발되었으며, 면적이 작고 전력 소모가 적으면서 데이터 처리량이 높은 파이프라인 구조가 구현시 선호된다[6]. 파이프라인 구조에서 메모리는 입력신호의 지연과 회전인자의 저장을 위해 필요하다. 효율적인 알고리즘을 채택하여 회전인자의 저장을 위한 메모리의 크기를 감소시킬 수 있다.

FFT는 매 스테이지 마다 버터플이 연산과 복소곱셈이 수행되는데, radix-2^k(k=2~5) 알고리즘을 개발하여 복소 곱셈의 수와 복소 곱셈기의 크기를 감소시키는 방법이 제안되었다[3]-[5].

고성능 FFT 설계를 위해서는 상위레벨에서 효율적인 알고리즘과 하드웨어 구조를 선택하고, 하위레벨에서 메모리 크기와 연산 복잡도 감소를 고려해야 한다. 본 논문에서는 회전인자 분해를 통해 새로운 radix-2^6 알고리즘을 제안하고, 회전인자와의 곱셈을 위한 복소 상수곱셈기의 효율적인 설계방법을 제안한다.

2. Radix-2 기반 FFT 알고리즘과 구조

신호 x(n)에 대한 N 포인트 DFT는 다음과 같다.

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

여기서 $W_N^{kn}=e^{-j2\pi nk/N}$은 회전인자이다. 위 식을 divide-and-conquer 기법을 사용하여 고속 계산할 수 있는 radix-2 FFT 알고리즘이 개발되었다.

Radix-2^2 FFT는 n과 k에 대해 3차원 인덱스 맵으로 분해하면 유도할 수 있다[3].

$$n=\frac{N}{2}n_1+\frac{N}{4}n_2+n_3,(n_1,n_2=0,1,\ n_3=0,1,\cdots\frac{N}{2}-1)$$

$$k=k_1+2k_2+4k_3,\ \ (k_1,k_2=0,1,\ \ k_3=0,1,\cdots\frac{N}{2}-1)$$

위의 인덱스 맵으로 회전인자를 분해하고, 버터플라이(BF)와 회전인자(TW)를 구분하면 다음과 같다.

$$W_N^{(\frac{N}{2}n_1+\frac{N}{4}n_2+n_3)(k_1+2k_2+4k_3)} \tag{2}$$
$$=\underbrace{(-1)^{n_1k_1}}_{\text{stg1 BF}}\overbrace{(-j)^{n_2k_1}}^{\text{stg1 TW}}\underbrace{(-1)^{n_2k_2}}_{\text{stg2 BF}}\overbrace{W_N^{n_3(k_1+2k_2)}}^{\text{stg2 TW}}W_{\frac{N}{4}}^{n_3k_3}$$

따라서 식 (1)은 다음과 같이 표현할 수 있다.

$$X(k_1+2k_2+4k_3) \tag{3}$$
$$=\sum_{n=0}^{N/4-1}\left[B_{\frac{N}{4}}(k_1,k_2,n_3)\,W_N^{n_3(k_1+2k_2)}\right]W_{\frac{N}{4}}^{n_3k_3}$$

여기서,

$$B_{\frac{N}{4}}(k_1,k_2,n_3)=x(n_3)+(-1)^{k_1}x(\frac{N}{2}+n_3)$$
$$+(-j)^{(k_1+2k_2)}x(\frac{N}{4}+n_3)+(-1)^{k_1}x(\frac{3N}{4}+n_3).$$

유사하게 n과 k에 대해 4차, 5차, 6차 인덱스 맵 분해를 통해 각각 radix-2^3과 radix-2^4, radix-2^5 알고리즘이 제안되었다[3]-[5].

그림 1은 8 포인트 radix-2와 radix-2^2 FFT의 신호흐름선도를 나타낸다. Radix-2 FFT는 매 스테이지 마다 복소 곱셈이 존재하지만, radix-2^2는 두 스테이지마다 복소 곱셈이 존재하므로 회전인자 저장을 위한 메모리의 크기를 감소시킬 수 있다.

다양한 FFT 구조 중 저면적, 저전력을 제공하는 SDF(single-path delay feedback) 구조가 많이 채택된다. 그림 2는 8 포인트 radix-2와 radix-2^2 FFT의 SDF 구조를 나타낸다. 이 구조는 적절한 버터플라이 연산을 위해 피드백 경로에 입력신호를 지연시키는 메모리가 필요하다. 박스 안에 표시된 숫자는 메모리의 크기를 나타낸다. 버터플라이 연산 값 중 덧셈결과는 다음 스테이지로 입력되고, 뺄셈결과는 피드백 메모리에 저장된다. 버터플라이 BF2는 BF1과 달리 -j의 곱셈을 포함하고 있다.

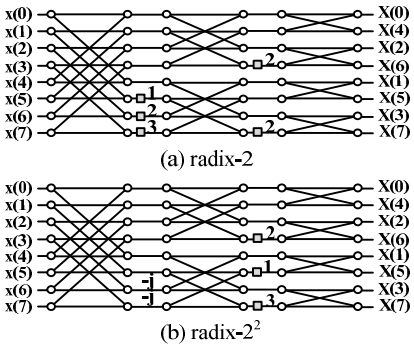

그림 1. Radix-2와 radix-2^2의 FFT 신호 흐름선도.
Fig. 1. Radix-2 and radix-2^2 FFT signal flow graph.

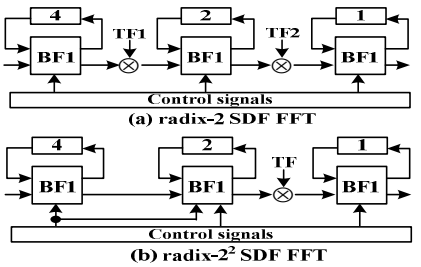

그림 2. Radix-2와 radix-2^2의 SDF FFT 구조.
Fig. 2. Radix-2 and radix-2^2 SDF FFT architectures.

3. 제안한 radix-2^6 FFT 알고리즘

3.1 기본 radix-2^6 알고리즘

Radix-2^6 알고리즘을 유도하기 위해 다음과 같은 7차원 인덱스 맵을 고려한다.

$$n=\frac{N}{2}n_1+\frac{N}{4}n_2+\frac{N}{8}n_3+\frac{N}{16}n_4+\frac{N}{32}n_5+\frac{N}{64}n_6+n_7,$$
$$(n_1,n_2,n_3,n_4,n_5,n_6=0,1,\ n_7=0,1,\cdots\frac{N}{64}-1)$$
$$k=k_1+2k_2+4k_3+8k_4+16k_5+32k_6+64k_7,$$
$$(k_1,k_2,k_3,k_4,k_5,k_6=0,1,\ k_7=0,1,\cdots\frac{N}{64}-1)$$

위의 인덱스 맵을 사용하여 회전인자를 분해하고, 정리하면 식 (4)와 같고, 버터플라이 연산과 회전인자를 고려하여 표현하면 다음과 같이 표현할 수 있다.

$$W_N^{kn}=\underbrace{(-1)^{n_1k_1}}_{\text{stg1 BF}}\overbrace{(-j)^{n_2k_1}}^{\text{stg1 TW}}\underbrace{(-1)^{n_2k_2}}_{\text{stg2 BF}}\overbrace{W_8^{n_3(k_1+2k_2)}}^{\text{stg2 TW}}\underbrace{(-1)^{n_3k_3}}_{\text{stg3 BF}} \quad (5)$$
$$\cdot\ \overbrace{W_N^{\frac{N}{16}n_4(k_1+2k_2+4k_3)}}^{\text{stg3 TW}}\underbrace{(-1)^{n_4k_4}}_{\text{stg4 BF}}\overbrace{W_N^{\frac{N}{32}n_5(k_1+2k_2+4k_3+8k_4)}}^{\text{stg4 TW}}$$
$$\cdot\ \underbrace{(-1)^{n_5k_5}}_{\text{stg5 BF}}\overbrace{W_N^{\frac{N}{64}n_6(k_1+2k_2+4k_3+8k_4+16k_5)}}^{\text{stg5 TW}}$$
$$\cdot\ \underbrace{(-1)^{n_6k_6}}_{\text{stg6 BF}}\overbrace{W_N^{n_7(k_1+2k_2+4k_3+8k_4+16k_5+32k_6)}}^{\text{stg6 TW}}\ W_{\frac{N}{64}}^{n_7k_7}$$

식 (5)는 스테이지 1을 제외한 나머지 스테이지에서 복소 곱셈이 필요하므로 기존 radix-2^k 알고리즘과 비교하여 장점이 없다. 본 논문에서는 식 (4)를 변형하여 개선한 새로운 알고리즘을 제안한다.

3.2 개선된 radix-2^6 알고리즘

식 (4)에서 A와 B, C와 D를 결합하여 정리한 후 각 스테이지에서 버터플라이 연산과 회전인자를 구분하면 다음과 같고, 방법1이라 한다.

$$W_N^{kn}=\underbrace{(-1)^{n_1k_1}}_{\text{stg1 BF}}\overbrace{(-j)^{n_2k_1}}^{\text{stg1 TW}}\underbrace{(-1)^{n_2k_2}}_{\text{stg2 BF}}\overbrace{W_{16}^{(2n_3+n_4)(k_1+2k_2)}}^{\text{stg2 TW}} \quad (6)$$
$$\cdot\ \underbrace{(-1)^{n_3k_3}}_{\text{stg3 BF}}\overbrace{(-j)^{n_4n_3}}^{\text{stg3 TW}}\underbrace{(-1)^{n_4k_4}}_{\text{stg4 BF}}\overbrace{W_{64}^{(2n_5+n_6)(k_1+2k_2+4k_3+8k_4)}}^{\text{stg4 TW}}$$
$$\cdot\ \underbrace{(-1)^{n_5k_5}}_{\text{stg5 BF}}\overbrace{(-j)^{n_6k_5}}^{\text{stg5 TW}}$$
$$\cdot\ \underbrace{(-1)^{n_6k_6}}_{\text{stg6 BF}}\overbrace{W_N^{n_7(k_1+2k_2+4k_3+8k_4+16k_5+32k_6)}}^{\text{stg6 TW}}\ W_{\frac{N}{64}}^{n_7k_7}$$

이 경우, 스테이지 1과 3, 5에서 회전인자는 -j이므로 회전인자 저장을 위한 메모리가 필요 없다.

식 (4)에서 B와 C, D와 E를 결합하고 정리한 후 각 스테이지에서 버터플라이 연산과 회전인자를 구분하면

$$W_N^{\left(\frac{N}{2}n_1+\frac{N}{4}n_2+\frac{N}{8}n_3+\frac{N}{16}n_4+\frac{N}{32}n_5+\frac{N}{64}n_6+n_7\right)(k_1+2k_2+4k_3+8k_4+16k_5+32k_6+64k_7)}$$
$$=(-1)^{n_1k_1}(-j)^{n_2(k_1+2k_2)}\underbrace{W_N^{\frac{N}{8}n_3(k_1+2k_2)}}_{\text{A}}(-1)^{n_3k_3}\underbrace{W_N^{\frac{N}{16}n_4(k_1+2k_2+4k_3)}}_{\text{B}}(-1)^{n_4k_4}\underbrace{W_N^{\frac{N}{32}n_5(k_1+2k_2+4k_3+8k_4)}}_{\text{C}} \quad (4)$$
$$(-1)^{n_5k_5}\underbrace{W_N^{\frac{N}{64}n_6(k_1+2k_2+4k_3+8k_4+16k_5)}}_{\text{D}}(-1)^{n_6k_6}\underbrace{W_N^{n_7(k_1+2k_2+4k_3+8k_4+16k_5+32k_6)}}_{\text{E}}W_{\frac{N}{64}}^{n_7k_7}$$

다음과 같고, 방법2이라 한다. 이 경우, 스테이지 1과 4의 회전인자는 -j이다.

$$W_N^{kn} = \underbrace{(-1)^{n_1k_1}}_{\text{stg1 BF}} \overbrace{(-j)^{n_2k_1}}^{\text{stg1 TW}} \underbrace{(-1)^{n_2k_2}}_{\text{stg2 BF}} \overbrace{W_8^{n_3(k_1+2k_2)}}^{\text{stg2 TW}} \cdot \underbrace{(-1)^{n_3k_3}}_{\text{stg3 BF}} \overbrace{W_{32}^{(2n_4+n_5)(k_1+2k_2+4k_3)}}^{\text{stg3 BF}} \underbrace{(-1)^{n_4k_4}}_{\text{stg4 BF}} \overbrace{(-j)^{n_5n_4}}^{\text{stg4 BF}} \cdot \underbrace{(-1)^{n_5k_5}}_{\text{stg5 BF}} \overbrace{W_N^{(\frac{N}{64}n_6+n_7)(k_1+2k_2+4k_3+8k_4+16k_5)}}^{\text{stg5 BF}} \cdot \underbrace{-1)^{n_6k_6}}_{\text{stg6 BF}} \overbrace{W_{\frac{N}{32}}^{n_7k_6}}^{\text{stg6 BF}} \; W_{\frac{N}{64}}^{n_7k_7} \tag{7}$$

방법1의 스테이지 4 회전인자 $W_{64}^{(2n_5+n_6)(k_1+2k_2+4k_3+8k_4)}$에서 가능한 서로 다른 회전인자는 31개(지수: 0~16, 18, 20, 21, 22, 24, 26, 27, 28, 30, 33, 36, 39, 42, 45)이며, 정현파 주기함수의 대칭성을 이용하면 $Re(W_{64}^i)$ (i=0,1,…15)만을 사용하여 구할 수 있다. 예를 들면, $W_{64}^{36} = -Re(W_{64}^4)$ $-j(W_{64}^{12})$이다. 즉, 16개의 회전인자 값의 부호를 바꾸거나, 코사인 값과 사인 값을 스위칭하면 가능한 모든 회전인자를 구할 수 있다.

일반적으로 회전인자 곱셈에 프로그래머블 복소 곱셈기가 사용된다. 만약 회전인자의 계수의 수가 적으면 회전인자 곱셈에 프로그래머블 복소 곱셈기 대신에 복소 상수곱셈기를 사용할 수 있다. 상수곱셈기는 0이 아닌 디지트의 수를 최소로 포함하는 CSD(canonic signed digit) 곱셈기를 이용하여 효율적으로 구현할 수 있다[8].

표 1은 회전인자 W_{64}^i의 CSD 계수를 나타낸다. 표에서 '10-1'(또는 -101), '101'(또는 -10-1), '100-1' 항이 반복되어 나타나므로 공통항을 한번만 구현하고 공유하면 효율적으로 CSD 곱셈기를 구현할 수 있다. 식 (8)은 W_{64}^i의 CSD 곱셈으로 쉬프트와 덧셈으로만 계산된다. 그림 3은 W_{64}^i에 대한 복소 상수곱셈기의 구조를 나타내며, 그림 4는 CSD 곱셈기의 세부회로를 나타낸다.

방법2에서 스테이지 3에서 나타나는 회전인자 W_{32}^n는 표 1의 $Re(W_{64}^{2i})$(i=0,1,…7) 값을 이용하여 W_{64}^n의 곱셈과 동일한 방법으로 구현할 수 있다.

$$\begin{aligned}
&d_1 = d - d \gg 2 \\
&d_2 = d + d \gg 2 \\
&d_3 = d - d \gg 3 \\
&d \times Re\{W_{64}^1\} = d - d_1 \gg 7 \\
&d \times Re\{W_{64}^2\} = d - d_2 \gg 6 \\
&d \times Re\{W_{64}^3\} = d - d_1 \gg 4 + d \gg 9 \\
&d \times Re\{W_{64}^4\} = d - d_2 \gg 4 + d \gg 9 \\
&d \times Re\{W_{64}^5\} = d_3 + d \gg 7 \\
&d \times Re\{W_{64}^6\} = d_1 + d_1 \gg 4 + d \gg 9 \\
&d \times Re\{W_{64}^7\} = d_1 + d_1 \gg 5 - d \gg 9 \\
&d \times Re\{W_{64}^8\} = d_1 - d_1 \gg 4 + d \gg 8 \\
&d \times Re\{W_{64}^9\} = d_2 \gg 1 + d \gg 7
\end{aligned} \tag{8}$$

표 1. W_{64}^i의 10비트 CSD 표현

Table 1 CSD representation for W_{64}^i with 10 bits

$Re(W_{64}^0)$	1	0	0	0	0	0	0	0	0	0
$Re(W_{64}^1)$	1	0	0	0	0	0	0	-1	0	1
$Re(W_{64}^2)$	1	0	0	0	0	0	-1	0	-1	0
$Re(W_{64}^3)$	1	0	0	0	-1	0	1	0	0	1
$Re(W_{64}^4)$	1	0	0	0	-1	0	-1	0	0	1
$Re(W_{64}^5)$	1	0	0	-1	0	0	0	1	0	-1
$Re(W_{64}^6)$	1	0	-1	0	1	0	1	0	0	1
$Re(W_{64}^7)$	1	0	-1	0	0	1	0	-1	0	-1
$Re(W_{64}^8)$	1	0	-1	0	-1	0	1	0	1	0
$Re(W_{64}^9)$	0	1	0	1	0	0	0	1	0	0
$Re(W_{64}^{10})$	0	1	0	0	1	0	0	-1	0	0
$Re(W_{64}^{11})$	0	1	0	0	0	-1	0	0	0	1
$Re(W_{64}^{12})$	0	1	0	-1	0	0	0	1	0	0
$Re(W1_{64}^3)$	0	0	1	0	0	1	0	1	0	0
$Re(W_{64}^{14})$	0	0	1	0	-1	0	0	1	0	-1
$Re(W_{64}^{15})$	0	0	0	1	0	-1	0	0	1	0

$$\begin{aligned}
&d \times Re\{W_{64}^{10}\} = d \gg 1 + d_3 \gg 4 \\
&d \times Re\{W_{64}^{11}\} = d \gg 1 - d \gg 5 + d \gg 9 \\
&d \times Re\{W_{64}^{12}\} = d_1 \gg 1 + d \gg 7 \\
&d \times Re\{W_{64}^{13}\} = d \gg 2 + d_2 \gg 5 \\
&d \times Re\{W_{64}^{14}\} = d_1 \gg 2 + d_1 \gg 7 \\
&d \times Re\{W_{64}^{15}\} = d_1 \gg 3 + d \gg 8
\end{aligned}$$

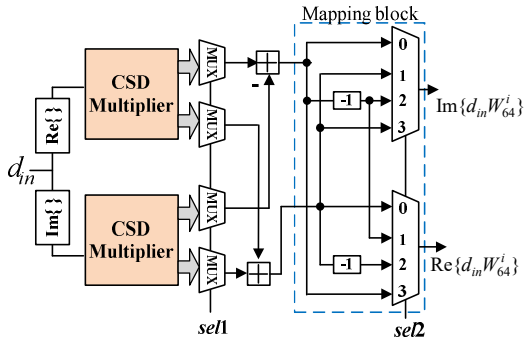

그림 3. W_{64}^{i}의 복소 CSD 곱셈기의 구조.
Fig. 3. Complex CSD multiplier structure of W_{64}^{i}.

제안한 radix-2^6 알고리즘을 SDF 구조에 적용한 256 포인트 FFT의 구조는 그림 5와 같다. 그림 5(a)의 방법1은 스테이지 2, 4, 6에만 복소 곱셈이 필요하며, 나머지 스테이지에서는 -j만 곱하면 된다. 그림 5(b)의 방법2는 스테이지 2, 3, 5, 6에서 복소곱셈이 필요하므로 방법1보다 연산복잡도가 높다.

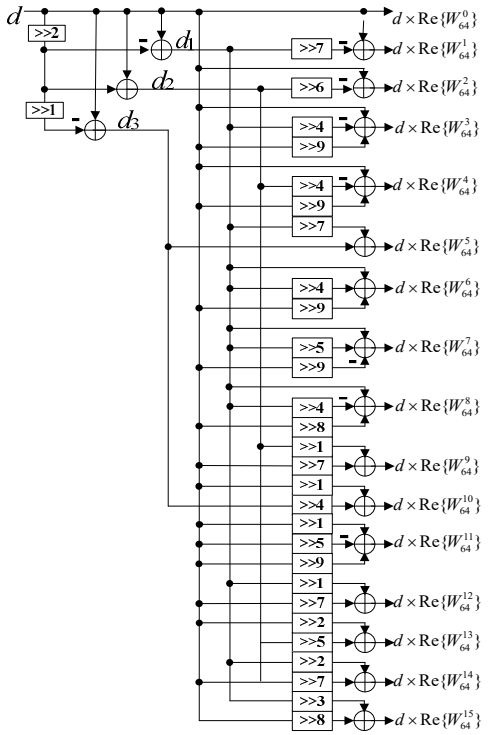

그림 4. 그림 3의 CSD 곱셈기의 세부구조
Fig. 4. Detailed CSD multiplier in Fig. 3.

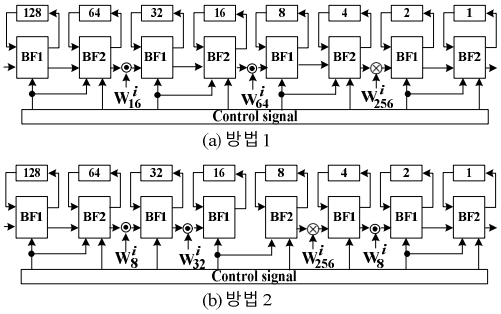

그림 5. 제안한 radix-2^6 SDF FFT 구조 (N=256).
Fig. 5. Proposed radix-2^6 SDF FFT architectures.

4. 시뮬레이션 및 성능평가

제안한 radix-2^6 알고리즘에서 사용되는 복소 곱셈기의 하드웨어 복잡도를 평가하기 위해 modified Booth 곱셈기 4개를 사용한 프로그래머블 복소 곱셈기와 제안한 W_{32}^{i}와 W_{64}^{i}에 대한 복소 CSD 곱셈기(CCM)를 Verilog를 사용하여 구현하고 Intel FPGA로 합성하였다. 표 2는 합성결과를 나타내며, 프로그래머블 복소 곱셈기를 기준으로 복소 CSD 곱셈기의 크기를 정규화하였다. W_{32}^{i}와 W_{64}^{i}에 대한 복소 CSD 곱셈기는 프로그래머블 복소곱셈기 보다 각각 약 55%, 29%가 적다.

표 3은 radix-2^5[5] 알고리즘과 제안한 radix-2^6 알고리즘에 대한 256 포인트 FFT의 스테이지별 회전인자를 나타낸다. 알고리즘에 따라 스테이지별로 회전인자가 다르다. Radix-2^6의 방법1은 복소 곱셈이 3개(프로그램버블 곱셈: 1개, 상수곱셈: 2개)의 스테이지에서 필요하지만, 다른 방법들은 4개(프로그램버블 곱셈: 1개, 상수곱셈: 3개)의 스테이지에서 복소 곱셈이 필요하다. 프로그래머블 복소 곱셈기를 사용하는 경우, 회전인자를 저장한 메모리에서 적절한 회전인자 계수를 가져오기 위해 별도의 회로가 필요하지 않다. 그러나 복소 상수곱셈기를 사용할 경우, 회전인자를 저장할 메모리는 필요 없지만, 그림 3처럼 적절한 회전인자와의 곱셈 결과값을 선택하기 위한 신호들을 계산하거나 저장하기 위한 별도의 메모리와 회로가 필요하다. 표 3에서 알 수 있듯이 radix-2^6의 방법1이 -j 곱셈이 규칙적으로 가장 많이 나타나므로 효율적이다.

제안한 알고리즘의 성능을 보다 정확히 평가하기 위

해 FPGA를 사용하여 256 포인트 SDF FFT를 구현하였다. 입력신호와 회전인자의 워드길이는 각각 12비트와 10비트로 하였다. 표 4는 합성결과에 대한 비교를 나타낸다. Radix-2^6 알고리즘의 방법1이 하드웨어를 가장 적게 사용하여 복잡도가 가장 적은 것을 알 수 있다. 네 방법 모두 회전인자와 피드백 값을 저장하는 메모리의 크기는 동일하나, 알고리즘에 따라 복소 상수곱셈기의 수와 회전인자의 계수가 다르므로 회전인자의 계수를 적절히 선택하기 위한 별도의 메모리와 회로의 복잡도가 달라 로직수와 메모리 차이가 발생하였다.

표 2. 곱셈기별 소요 로직 수
Table 2. Gate count of each multiplier

구분	복소 곱셈	CCM_ W8	CCM_ W16	CCM_ W32	CCM_ W64
로직 수	1,520	205	416	681	1,072
정규화지수	1	0.13	0.27	0.45	0.71

표 3. 256-포인트 FFT의 스테이지별 회전인자
Table 3. Twiddle factors for 256-point FFT

구분	stg1	stg2	stg3	stg4	stg5	stg6	stg7
$R2^5$(M1)	-j	W_8^n	W_{32}^n	-j	W_{256}^{n*}	-j	W_8^n
$R2^5$(M2)	-j	W_{16}^n	-j	W_{256}^{n*}	W_{32}^n	-j	W_{16}^n
$R2^6$(M1)	-j	W_{16}^n	-j	W_{64}^n	-j	W_{256}^{n*}	-j
$R2^6$(M2)	-j	W_8^n	W_{32}^n	-j	W_{256}^{n*}	W_{16}^n	-j

M: Method, W_{256}^{n*}: 프로그래머블 복소 곱셈 사용

표 4. 256-포인트 FFT의 하드웨어 비교
Table 4. Hardware comparison of 256-point FFT

구분	로직 수	메모리 비트 수
$R2^5$(M1)	4,782 (0.97)	14,312 (1)
$R2^5$(M2)	4,912 (1)	14,312 (1)
$R2^6$(M1)	4,446 (0.90)	13,800 (0.95)
$R2^6$(M2)	4,733 (0.96)	14,312 (1)

5. 결 론

본 논문에서는 7차원 인덱스 맵핑을 사용하여 FFT 회전인자를 분해한 후 연산 복잡도가 적고, 하드웨어 효율이 우수한 새로운 radix-2^6 FFT 알고리즘을 제안하였다. 제안한 알고리즘의 방법1은 복소 곱셈의 수와 회전인자를 저장하기 위한 메모리의 크기를 줄일 수 있다. 복소 곱셈에 대한 하드웨어 복잡도를 줄이기 위해 복소 곱셈기 대신 복소 CSD 곱셈기를 설계하는 방법을 제시하였다. 기존 알고리즘과 제안한 알고리즘의 성능을 평가하기 위해 SDF 구조를 사용하여 256 포인트 FFT를 설계하고 FPGA로 합성하여 기존 알고리즘 보다 10% 정도 하드웨어가 적게 사용됨을 보였다.

REFERENCES

[1] J. Yu and K. J. Cho, "A low-area and low-power 512-point pipelined FFT design using raidx-24-23 for OFDM applications", *JKIIECT*, vol. 11. no. 5, pp. 475-480, 2018.

[2] C. Yu and M. H. Yen, "Area-efficient 128- to 2048/1536-point pipeline FFT processor for LTE and mobile WiMAX systems", *IEEE VLSI Syst.*, vol. 23, pp. 1793-1800, 2015.

[3] S. He and M. Torkelson, "A new approach to pipelined FFT processor", *IPPS'96*, pp. 766-770, 1996.

[4] J. Y. Oh and M. S. Lim, "New radix-2 to the 4th power pipeline FFT processor", *IEICE trans. Electron.*, vol. E88-C, no. 8, pp.1740-1746, 2005.

[5] T. S. Cho and H. H. Lee, "A high-speed low-complexity modified radix-2^5 FFT processor for high rate WPAN applications", *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 21, no. 1, pp. 187-191, Feb, 2013.

[6] C. Yang, C. Wei, Y. Xie, H. Chen and C. Ma, "Area-efficient mixed-radix variable -length FFT processor", *IEICE Electron. Expr.*, vol. 14, no. 10, pp. 1-10, 2017.

[7] S. J. Huang, S. G. Chen, "A high-throughput radix-16 FFT processor with parallel and normal input/output ordering for IEEE 802.15. 3c systems", *IEEE Tran. Circuits Sys. I: Reg. Papers*, vol. 59, no. 8, pp.1752-1765, 2012.

[8] G. K. Ganjikunta and S. K. Sahoo, "An area-efficient and low-power 64-point pipeline Fast Fourier Transform for OFDM applications", *Integration, the VLSI Journal*, vol. 57, pp. 125-131, 2017.

저자약력

조 경 주 (Kyung-Ju Cho) **[정회원]**

- 2006년 8월: 전북대학교 정보통신공학과 박사
- 2012년 3월 ~ 현재: 원광대학교 전자공학과 교수

〈관심분야〉 VLSI설계, SOC