한국산학기술학회논문지
Vol. 12, No. 4 pp. 1867-1875, 2011

Common Sub-expression Sharing을 사용한 저면적 FFT 프로세서 구조

장영범[1*], 이동훈[1]
[1]상명대학교 정보통신공학과

Low-area FFT Processor Structure using Common Sub-expression Sharing

Young-Beom Jang[1*] and Dong-Hoon Lee[1]
[1]**Department of Information Technology and Communications, SANGMYUNG UNIV.**

요 약 이 논문에서는 저면적 256-point FFT 구조를 제안한다. 저면적 구현을 위하여 CSD(Canonic Signed Digit) 곱셈기 방식을 채택하여 구현하였다. CSD 곱셈기 방식을 효율적으로 적용하기 위해서는 곱셈연산의 가지 수가 적어야 하는데, 여러 알고리즘을 조사한 결과 Radix-4^2 알고리즘이 곱셈연산의 가지 수가 적음을 발견하였다. 따라서 제안 구조는 Radix-4^2 DIF 알고리즘과 CSD 곱셈기 방식을 사용하였다. 즉 Radix-4^2 알고리즘을 사용하여 4개의 스테이지에서 사용되는 곱셈연산의 가지 수를 최소화한 후에 각각의 곱셈연산 블록은 CSD 곱셈기를 사용하여 구현하였다. CSD 곱셈기 구현에서 공통패턴을 공유하여 덧셈기의 수를 줄일 수 있는 CSS(Common Sub-expression Sharing) 기술을 사용하여 구현면적을 더욱 감소시켰다. 제안된 FFT 구조를 Verilog-HDL 코딩 후 합성하여 구현한 결과, Radix-4를 사용한 구조와 비교하여 복소 곱셈기 부분의 29.9%의 cell area 감소를 보였고 전체적인 256-point FFT 구조에 대한 비교에서는 12.54% cell area 감소를 보였다.

Abstract In this paper, a low-area 256-point FFT structure is proposed. For low-area implementation CSD(Canonic Signed Digit) multiplier method is chosen. Because multiplication type should be less for efficient CSD multiplier application to the FFT structure, the Radix-4^2 algorithm is chosen for those purposes. After, in the proposed structure, the number of multiplication type is minimized in each multiplication block, the CSD multipliers are applied for implementation of multiplication. Furthermore, in CSD multiplier implementation, cell-area is more reduced through common sub-expression sharing(CSS). The Verilog-HDL coding result shows 29.9% cell area reduction in the complex multiplication part and 12.54% cell area reduction in overall 256-point FFT structure comparison with those of the conventional structure.

Key Words : FFT, Radix-4^2, CSD, Common Sub-expression Sharing(CSS), OFDM

1. 서 론

OFDM(Orthogonal Frequency Division Multi plexing) 방식을 사용하는 통신 단말기가 널리 상용화됨에 따라 MODEM SoC(System on Chip)의 고속 및 저면적 구현에 관한 연구가 활발히 진행되고 있다. 특히 OFDM 블록 중 가장 큰 면적을 차지하는 FFT 블록의 저면적 구현은 반드시 필요한 연구 분야이다.

FFT의 구현에는 파이프라인 방식이 주로 사용되며, 파이프라인 방식에서는 복소 곱셈연산의 구현이 가장 중요하다. 복소 곱셈연산을 구현하는 방법으로는 Modified Booth 곱셈기를 사용하는 방법이 가장 일반적이며, CORDIC(COordinate Rotation DIgital Computer) 곱셈기를 사용하는 방식도 사용되고 있다.[1][2] 이와 더불어 CSD(Canonic Signed Digit) 곱셈기를 쓰는 방식도 연구

*교신저자 : 장영범(ybjang@smu.ac.kr)
접수일 11년 01월 07일 수정일 11년 04월 06일 게재확정일 11년 04월 07일

되었다[3]. CSD 곱셈기는 디지털 필터의 저면적 구현에 도 널리 사용되고 있다[4,5].

FFT의 곱셈 연산을 위한 CSD 곱셈기 방식은 수를 CSD형으로 표기한 후에 이를 덧셈기와 쉬프트를 사용하여 구현하는 저면적 구현방법이다[6-8].

이 논문에서는 CSD 곱셈기 방식의 새로운 고속/저면적 FFT 구조를 제안한다. 고속 FFT 구현을 위하여 Radix-4 계열을 선택하였으며 저면적 구현을 위하여 CSD 곱셈기 방식을 선택하였다. CSD 곱셈기 방식을 사용하여 저면적으로 구현하기 위하서는 복소 곱셈연산의 가지 수가 적어야만 한다. 이 논문에서 Radix-4^2 알고리즘을 사용하여 복소 곱셈연산의 가지 수를 최소화시킴으로써 저면적 구현을 달성할 수 있었다. 이 논문에서 제안한 CSD 곱셈기 방식에서는 CSS 기술을 사용하는데 이 CSS(Common Sub-expression Sharing) 기술은 각 스테이지의 CSD형 계수를 공통패턴을 찾아 공유함으로써 덧셈기를 줄여 기존 CSD 알고리즘만을 사용하여 구현한 방식보다 더욱 구현 면적을 줄일 수 있음을 보인다. 이 논문의 2장에서는 Radix-4^2 알고리즘에 대하여 알아보고 3장에서는 저면적 FFT 구조를 제안한다. 4장에서는 구현을 통하여 제안 구조의 효용성을 입증하며 5장에서 결론을 맺는다.

2. Radix-4^2 FFT 알고리즘

곱셈기를 사용하지 않는 256-point FFT 구조를 만들기 위해서 알고리즘을 먼저 선택해야 한다. 고속 연산을 위하여 Radix-4 계열의 알고리즘을 선택하였고, 그 중에서도 곱셈연산의 수가 적은 Radix-4^2 알고리즘을 사용하여 저 전력 구조를 설계하기로 한다. 먼저 Radix-4^2 알고리즘에 대하여 간단히 소개한다. *N*-point의 DFT의 정의는 다음과 같다.

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

식 (1)에서 Twiddle Factor는 $W_N^{nk}=e^{-j2\pi nk/N}$이고 여기서 n은 Time Index, k는 Frequency Index이다. n과 k를 CFA(Common Factor Algorithm)에 의하여 다음과 같이 3차원으로 분리하여 전개한다.

$$\begin{aligned} n &= \frac{N}{4}n_1+\frac{N}{16}n_2+n_3 \quad \left\{0 \le n_1,n_2 \le 3,\ 0 \le n_3 \le \frac{N}{4}-1\right\} \\ k &= k_1+4k_2+16k_3 \quad \left\{0 \le k_1,k_2 \le 3,\ 0 \le k_3 \le \frac{N}{4}-1\right\} \end{aligned} \tag{2}$$

식 (1)에 식 (2)를 대입하면 Radix-4^2의 FFT를 다음과 같이 나타낼 수 있다.

$$\begin{aligned} &X(k_1+4k_2+16k_3) \\ &=\sum_{n_3=0}^{\frac{N}{16}-1}\sum_{n_2=0}^{3}\sum_{n_1=0}^{3} x\left(\frac{N}{4}n_1+\frac{N}{16}n_2+n_3\right) W_N^{\left(\frac{N}{4}n_1+\frac{N}{16}n_2+n_3\right)(k_1+4k_2+16k_3)} \\ &=\sum_{n_3=0}^{\frac{N}{16}-1}\sum_{n_2=0}^{3}\left\{\sum_{n_1=0}^{3} x\left(\frac{N}{4}n_1+\frac{N}{16}n_2+n_3\right) W_N^{\frac{N}{4}n_1k_1}\right\} W_N^{\left(\frac{N}{16}n_2+n_3\right)(k_1+4k_2+16k_3)} \\ &=\sum_{n_3=0}^{\frac{N}{16}-1}\sum_{n_2=0}^{3}\left[B\left(\frac{N}{16}n_2+n_3,k_1\right)\right] W_N^{\left(\frac{N}{16}n_2+n_3\right)(k_1+4k_2+16k_3)} \end{aligned} \tag{3}$$

위의 식에서 보듯이 첫 번째 스테이지 버터플라이 $B(\ )$는 4-point FFT로서 다음과 같다.

$$B\left(\frac{N}{16}n_2+n_3,k_1\right)=\sum_{n_1=0}^{3} x\left(\frac{N}{4}n_1+\frac{N}{16}n_2+n_3\right) W_N^{\frac{N}{4}n_1k_1} \tag{4}$$

식 (3)에서 Twiddle Factor는 다음과 같이 간소화 될 수 있다.

$$W_N^{\left(\frac{N}{16}n_2+n_3\right)(k_1+4k_2+16k_3)}= W_N^{\frac{N}{16}n_2k_1} W_N^{\frac{N}{4}n_2k_2} W_N^{n_3(k_1+4k_2)} W_{\frac{N}{16}}^{n_3k_3} \tag{5}$$

간소화 된 Twiddle Factor를 이용하여 식 (3)을 간단히 하면 다음과 같다.

$$\begin{aligned} &X(k_1+4k_2+16k_3)= \\ &\sum_{n_3=0}^{\frac{N}{16}-1}\sum_{n_2=0}^{3}\left[B\left(\frac{N}{16}n_2+n_3,k_1\right)\right] W_N^{\frac{N}{16}n_2k_1} W_N^{\frac{N}{4}n_2k_2} W_N^{n_3(k_1+4k_2)} W_{\frac{N}{16}}^{n_3k_3} \end{aligned} \tag{6}$$

이 식에서 두 번째 stage의 4-point FFT를 버터플라이 $H(\ )$로 다음과 같이 정의한다.

$$H(k_1,k_2,n_3)=\sum_{n_2=0}^{3}\left\{B\left(\frac{N}{16}n_2+n_3,k_1\right) W_N^{\frac{N}{16}n_2k_1}\right\} W_N^{\frac{N}{4}n_2k_2} \tag{7}$$

식 (7)의 정의를 사용하여 식 (6)을 나타내면 다음과 같다.

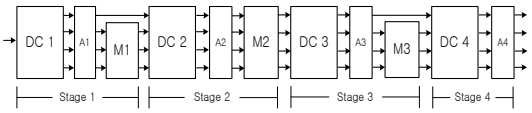
$$X(k_1+4k_2+16k_3)=\sum_{n_3=0}^{\frac{N}{16}-1}\left[H(k_1,k_2,n_3)\, W_N^{n_3(k_1+4k_2)}\right] W_{\frac{N}{16}}^{n_3k_3} \tag{8}$$

식 (8)에서 보듯이 Radix-4^2 알고리즘을 사용하면 *N*-point FFT를 *N*/16-point FFT로 나타낼 수 있다. 위의 과정과 같이 두 번째 스테이지까지 전개 한 후 남은 *N*/16-point 길이의 DFT 연산은 같은 방식으로 반복적으로 전개할 수 있다.

3. 제안된 FFT 구조

3.1 전체 구조

이 절에서는 CSD 곱셈기를 사용한 새로운 고속/저면적 256-point FFT의 구조를 제안한다. 즉 곱셈연산을 위하여 덧셈기와 쉬프트 연산만을 사용하는 CSD 곱셈기 FFT 구조를 제안한다. 우리가 선택한 Radix-4^2의 알고리즘을 사용한 256-point FFT의 전체 구조의 블록 도는 그림 1과 같이 4개의 스테이지로 구성된다.



[그림 1] Radix-4^2 256-point FFT 블록도

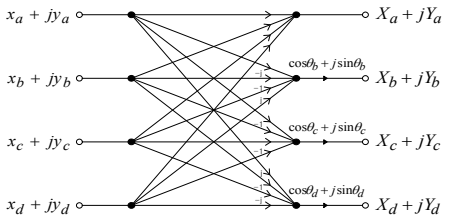
그림 1에서 보듯이 각각의 스테이지는 Delay commutator(DC), 덧셈기(A), 그리고 CSS로 불리는 곱셈연산 블록(M)으로 구성된다. 256-point FFT는 Radix-4^2 알고리즘을 사용하면 4x4x16으로 분해되므로 마지막 16-point FFT는 다시 Radix-4^2 알고리즘을 적용한다. 모든 스테이지에서 사용되는 Delay Commutator는 MDC(Multi-path Delay Commutator) 방식을 사용하였다. 이 논문에서는 각 스테이지에서 사용되는 곱셈연산을 저면적으로 구현하는 CSD 방식을 제안한다. 마지막 스테이지에서는 곱셈연산이 사용되지 않으므로 앞의 3개의 스테이지에 대한 곱셈연산 구조를 다음 절에서 제안한다.

3.2 첫 번째 스테이지 설계

첫 번째 스테이지의 DC1은 MDC 방식을 사용하여 설계하였다. 이제 첫 번째 스테이지의 버터플라이 블록을 설계해보기로 한다. 먼저 구현해야할 식은 다음과 같다.

$$B\left(\frac{N}{16}n_2+n_3,k_1\right)W_N^{\frac{N}{16}n_2k_1} \tag{9}$$

식 (9)를 구현하기 위한 버터플라이 구조는 그림 2와 같다.

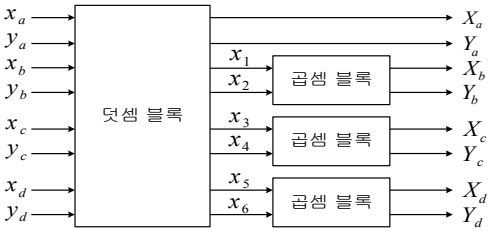


[그림 2] 첫 번째 스테이지의 버터플라이 블록도

그림 2의 버터플라이 출력의 실수부와 허수부는 각각 다음과 같이 나타낼 수 있다.

$$\begin{aligned}
X_a &= x_a+x_b+x_c+x_d, \quad Y_a = y_a+y_b+y_c+y_d \\
X_b &= x_1\cos\theta_b+x_2\sin\theta_b, \quad Y_b = x_2\cos\theta_b-x_1\sin\theta_b \\
X_c &= x_3\cos\theta_c+x_4\sin\theta_c, \quad Y_c = x_4\cos\theta_c-x_3\sin\theta_c \\
X_d &= x_5\cos\theta_d+x_6\sin\theta_d, \quad Y_d = x_6\cos\theta_d-x_5\sin\theta_d \\
x_1 &= x_a+y_b-x_c-y_d, \quad x_2 = y_a-x_b-y_c+x_d \\
x_3 &= x_a-x_b+x_c-x_d, \quad x_4 = y_a-y_b+y_c-y_d \\
x_5 &= x_a-y_b-x_c+y_d, \quad x_6 = y_a+x_b-y_c-x_d
\end{aligned} \tag{10}$$

식 (10)을 구현하기 위한 회로는 그림 3과 같다.



[그림 3] 첫 번째 스테이지 버터플라이 블록도

그림 3의 덧셈 블록은 식 (10)의 덧셈 연산을 수행하도록 설계한다. 첫 번째 스테이지는 총 6개의 Twiddle Factor $W_{16}^1, W_{16}^2, W_{16}^3, W_{16}^4, W_{16}^6, W_{16}^9$ 가 사용된다. 그러나 Twiddle Factor는 주기함수이므로 중복되는 것을 제외하고 N/8 만큼의 각도만 고려하면 W_{16}^1, W_{16}^2 만 남는다. 즉 첫 번째 스테이지에서 사용되는 곱셈의 가지 수는 0.7071, 0.9239, 0.3827의 3개의 수밖에 없다. 오직 3개의 곱셈만 사용되므로 덧셈기와 쉬프트를 사용하여 구현하였다. 곱셈용 계수는 덧셈 연산의 수를 줄이기 위하여 CSD(Canonic Signed Digit)형의 계수를 사용하였다. 계수들의 2진수 표현에서 1의 수가 곧 덧셈의 수가 되므로 1의 빈도가 구현 비용과 비례한다. 따라서 CSD형 계수를 사용하면 2의 보수형 계수를 사용하는 것보다 덧셈연산의 수를 줄여 구현비용을 크게 절감할 수 있다. 즉, CSD형의 계수는 (N+1)/2 이상의 non-zero 비트를 갖지 않는 장점이 있으므로 덧셈의 수를 더욱 감소시킬 수 있다. 표 1은 첫 번째 스테이지에서 사용되는 3개의 계수에 대한 CSD형 계수이다.

[표 1] 첫 번째 스테이지 Twiddle Factor의 CSD형 계수

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.9239	1	0	0	0	N	0	N	0	0	1	0	0	0	0	1	0
0.3827	0	1	0	N	0	0	0	1	0	0	0	0	0	N	0	0
0.7071	1	0	N	0	N	0	1	0	1	0	0	0	0	0	1	0

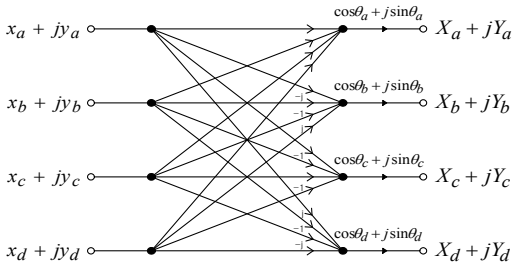
표 1에서 사용되는 0.9239, 0.3827, 0.7071의 CSD형 계수를 구현하는 데에는 각각 4개, 3개, 5개의 덧셈기가 필요하다.

3.3 두 번째 스테이지 설계

두 번째 스테이지의 DC2도 MDC 방식을 사용하여 설계하였다. 이제 두 번째 스테이지의 버터플라이 블록을 설계해보기로 한다. 먼저 구현해야할 식은 다음과 같다.

$$H(k_1,k_2,n_3)\,W_N^{n_3(k_1+4k_2)} \tag{11}$$

식 (11)을 구현하기 위한 버터플라이 구조는 그림 4와 같다.

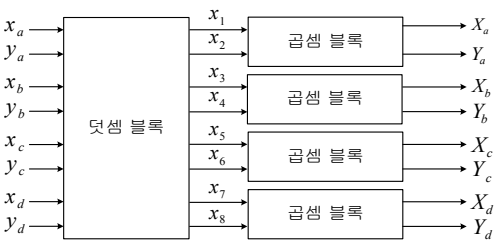


[그림 4] 두 번째 스테이지의 버터플라이 블록도

그림 4의 버터플라이 출력의 실수부와 허수부는 각각 다음과 같이 나타낼 수 있다.

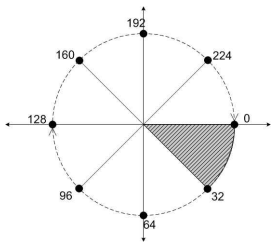
$$\begin{aligned}
&X_a = x_1\cos\theta_a + x_2\sin\theta_a,\quad Y_b = x_2\cos\theta_a - x_1\sin\theta_a\\
&X_b = x_3\cos\theta_b + x_4\sin\theta_b,\quad Y_b = x_4\cos\theta_b - x_3\sin\theta_b\\
&X_c = x_5\cos\theta_c + x_6\sin\theta_c,\quad Y_c = x_6\cos\theta_c - x_5\sin\theta_c\\
&X_d = x_7\cos\theta_d + x_8\sin\theta_d,\quad Y_d = x_8\cos\theta_d - x_7\sin\theta_d\\
&x_1 = x_a + x_b + x_c + x_d,\qquad x_2 = y_a + y_b + y_c + y_d\\
&x_3 = x_a + y_b - x_c - y_d,\qquad x_4 = y_a - x_b - y_c + x_d\\
&x_5 = x_a - x_b + x_c - x_d,\qquad x_6 = y_a - y_b + y_c - y_d\\
&x_7 = x_a - y_b - x_c + y_d,\qquad x_8 = y_a + x_b - y_c - x_d
\end{aligned} \tag{12}$$

식 (12)를 구현하기 위한 회로는 그림 5와 같다.



[그림 5] 두 번째 스테이지 버터플라이 블록도

그림 5의 덧셈 블록은 식 (12)의 덧셈연산을 수행한다. 그림 5의 곱셈 연산 블록은 많은 종류의 곱셈을 수행하므로 구현면적을 줄여야 하는 것이 이 논문의 핵심 목표이다. 두 번째 스테이지에서는 256개의 Twiddle Factor 중에서 그림 6과 같이 1/8에 해당하는 32개의 Twiddle Factor 만을 고려하면 된다.



[그림 6] 256-point FFT에 사용되는 Twiddle Factor 영역

두 번째 스테이지에서는 그림 6의 회색부분에 분포한 Twiddle Factor가 이용된다. 이 회색부분에 분포한 32개의 Twiddle Factor에 대한 실수부 32개, 허수부 32개의 총 64개의 계수들이 사용되는데 45° 부분의 값은 (0.7071, 0.7071)로 같으므로 63개의 계수를 사용하게 된다.

N-point Radix 알고리즘에서 point의 값이 증가할수록 이를 저장하는 ROM의 크기가 커지게 되고, 복소 곱셈 연산의 수도 증가하게 되어 전력 소모 및 면적이 증가하게 된다. 이를 해결하기 위해 두 번째 스테이지에서는 CSD형의 계수와 CSS(Common Sub- expression Sharing) 방식을 사용하여 구현하였다. CSS 방식은 곱셈 연산을 덧셈연산과 쉬프트를 이용하여 구현하는 방법으로서 공통패턴을 정의하여 서로 공유하는 방식이다. CSS 기술을 사용하기 위해서는 먼저 사용되는 상수 계수들을 CSD형으로 나타내어야 한다.

두 번째 스테이지에서 사용되는 63개의 계수 중에서 실수에 해당하는 32개의 계수를 16비트 정세도 CSD형으로 나타내면 표 2와 같다. 표 2에서 N은 -1을 나타낸 것이고 이와 같은 CSD형으로 나타내면 2의 보수형보다

non-zero 비트의 수가 더 적게 쓰이는 것을 알 수 있다.

[표 2] 두 번째 스테이지의 실수부 CSD형 계수

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.9997	1	0	0	0	0	0	0	0	0	0	0	0	N	0	N	0
0.9988	1	0	0	0	0	0	0	0	0	0	N	0	N	0	0	0
0.9973	1	0	0	0	0	0	0	0	N	0	1	0	1	0	0	N
0.9952	1	0	0	0	0	0	0	0	N	0	N	0	0	0	1	0
0.9925	1	0	0	0	0	0	0	N	0	0	0	0	1	0	1	0
0.9892	1	0	0	0	0	0	N	0	1	0	1	0	0	0	N	0
0.9853	1	0	0	0	0	0	N	0	0	0	1	0	0	0	N	0
0.9809	1	0	0	0	0	0	N	0	N	0	0	1	0	0	N	0
0.9757	1	0	0	0	0	N	0	1	0	0	N	0	0	1	0	N
0.9700	1	0	0	0	0	N	0	0	0	0	1	0	1	0	0	0
0.9638	1	0	0	0	0	N	0	0	N	0	N	0	0	N	0	1
0.9569	1	0	0	0	N	0	1	0	1	0	0	0	0	N	0	N
0.9495	1	0	0	0	N	0	1	0	N	0	0	0	1	0	0	1
0.9415	1	0	0	0	N	0	0	0	1	0	0	0	0	1	0	N
0.9330	1	0	0	0	N	0	0	0	N	0	0	N	0	N	0	0
0.9239	1	0	0	0	N	0	N	0	0	1	0	0	0	0	1	0
0.9142	1	0	0	N	0	1	0	1	0	0	0	0	0	1	0	0
0.9040	1	0	0	N	0	1	0	0	0	N	0	0	N	0	N	0
0.8932	1	0	0	N	0	0	1	0	0	1	0	1	0	1	0	0
0.8819	1	0	0	N	0	0	0	1	0	0	N	0	0	0	1	0
0.8701	1	0	0	N	0	0	0	0	N	0	N	0	0	0	0	N
0.8577	1	0	0	N	0	0	N	0	0	N	0	0	1	0	0	1
0.8449	1	0	0	N	0	N	0	0	0	0	1	0	0	1	0	1
0.8315	1	0	N	0	1	0	1	0	1	0	0	N	0	0	N	0
0.8176	1	0	N	0	1	0	0	0	1	0	1	0	1	0	0	N
0.8032	1	0	N	0	1	0	0	N	0	N	0	1	0	0	0	N
0.7883	1	0	N	0	0	1	0	1	0	0	N	0	1	0	0	N
0.7730	1	0	N	0	0	1	0	N	0	0	0	N	0	0	0	1
0.7572	1	0	N	0	0	0	0	1	0	0	0	N	0	N	0	N
0.7410	1	0	N	0	0	0	0	N	0	0	N	0	N	0	0	1
0.7242	1	0	N	0	0	N	0	1	0	N	0	N	0	0	1	0
0.7071	1	0	N	0	N	0	1	0	1	0	0	0	0	0	1	0

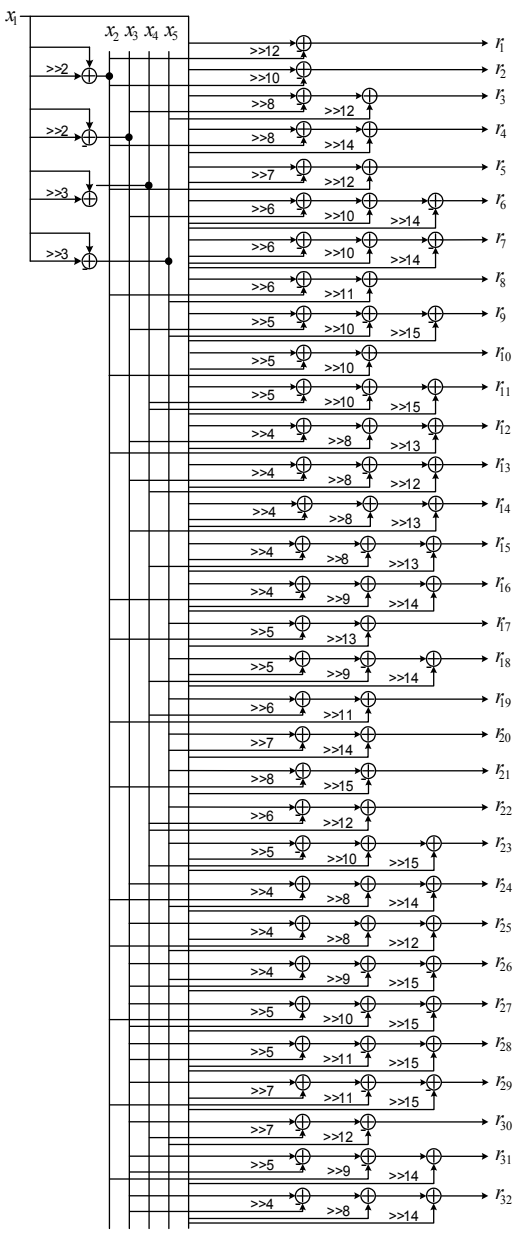
표 2의 CSD형 계수를 덧셈기를 사용하여 구현하면 142개의 덧셈기가 필요하게 된다.

이제 덧셈 연산의 수를 더욱 줄이기 위해 CSS 방식을 실수부에 적용하기로 한다. 공통 패턴은 표 2에서 2중 실선으로 표현하였다. 이 표에서 101, 10N, 1001, 100N, N01, N0N, N001, N00N의 8개의 공통패턴을 정의하였다. 여기서 N01, N0N, N001, N00N은 101, 10N, 1001, 100N의 부호만 바꾸면 되므로 따로 공통패턴으로 정의할 필요는 없다. 따라서 공통 패턴과 실수 계수는 다음 식과 같다.

$$
\begin{aligned}
&101 \quad \text{패턴} && x_2 = x_1 + x_1 \gg 2 \\
&10N \quad \text{패턴} && x_3 = x_1 - x_1 \gg 2 \\
&1001 \quad \text{패턴} && x_4 = x_1 + x_1 \gg 3 \\
&100N \text{ 패턴} && x_5 = x_1 - x_1 \gg 3
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
r_1 &= x_1 - x_2 \gg 12 \\
r_2 &= x_1 - x_2 \gg 10 \\
r_3 &= x_1 - x_3 \gg 8 + x_5 \gg 12 \\
r_4 &= x_1 - x_2 \gg 8 + x_1 \gg 14 \\
r_5 &= x_1 - x_1 \gg 7 + x_2 \gg 12 \\
r_6 &= x_1 - x_3 \gg 6 + x_1 \gg 10 - x_1 \gg 14 \\
r_7 &= x_1 - x_1 \gg 6 + x_1 \gg 10 - x_1 \gg 14 \\
r_8 &= x_1 - x_2 \gg 6 + x_5 \gg 11 \\
r_9 &= x_1 - x_3 \gg 5 - x_5 \gg 10 - x_1 \gg 15 \\
r_{10} &= x_1 - x_1 \gg 5 + x_2 \gg 10 \\
r_{11} &= x_1 - x_4 \gg 5 - x_4 \gg 10 + x_1 \gg 15 \\
r_{12} &= x_1 - x_3 \gg 4 + x_1 \gg 8 - x_2 \gg 13 \\
r_{13} &= x_1 - x_3 \gg 4 - x_1 \gg 8 + x_4 \gg 12 \\
r_{14} &= x_1 - x_1 \gg 4 + x_1 \gg 8 + x_3 \gg 13 \\
r_{15} &= x_1 - x_1 \gg 4 - x_4 \gg 8 - x_1 \gg 13 \\
r_{16} &= x_1 - x_2 \gg 4 + x_1 \gg 9 + x_1 \gg 14 \\
r_{17} &= x_5 + x_2 \gg 5 + x_1 \gg 13 \\
r_{18} &= x_5 + x_1 \gg 5 - x_4 \gg 9 - x_1 \gg 14 \\
r_{19} &= x_5 + x_4 \gg 6 + x_2 \gg 11 \\
r_{20} &= x_5 + x_5 \gg 7 + x_1 \gg 14 \\
r_{21} &= x_5 - x_2 \gg 8 - x_1 \gg 15 \\
r_{22} &= x_5 - x_4 \gg 6 + x_4 \gg 12 \\
r_{23} &= x_5 - x_1 \gg 5 + x_4 \gg 10 + x_1 \gg 15 \\
r_{24} &= x_3 + x_2 \gg 4 + x_5 \gg 8 - x_1 \gg 14 \\
r_{25} &= x_3 + x_1 \gg 4 + x_2 \gg 8 + x_5 \gg 12 \\
r_{26} &= x_3 + x_5 \gg 4 - x_3 \gg 9 - x_1 \gg 15 \\
r_{27} &= x_3 + x_2 \gg 5 - x_3 \gg 10 - x_1 \gg 15 \\
r_{28} &= x_3 + x_3 \gg 5 - x_1 \gg 11 + x_1 \gg 15 \\
r_{29} &= x_3 + x_1 \gg 7 - x_2 \gg 11 - x_1 \gg 15 \\
r_{30} &= x_3 - x_4 \gg 7 - x_5 \gg 12 \\
r_{31} &= x_3 - x_3 \gg 5 - x_2 \gg 9 + x_1 \gg 14 \\
r_{32} &= x_3 - x_3 \gg 4 + x_1 \gg 8 + x_1 \gg 14
\end{aligned} \tag{14}
$$

이와 같이 얻어진 식을 사용하여 두 번째 스테이지 실수부에 대한 회로를 그림 7과 같이 설계하였다.



[그림 7] 제안된 두 번째 스테이지 실수 곱셈블록 회로도

그림 7의 실수부 곱셈기에서 사용되는 덧셈기는 85이다. 즉 CSD형 계수를 그대로 사용할 때보다 57개를 줄일 수 있었다.

두 번째 스테이지에서 사용되는 63개의 계수 중에서 허수를 16비트 정세도 CSD형으로 나타내면 표 3과 같다. 표 3에서 0.7071은 그림 6의 Twiddle Factor영역 중 45°의 값으로 (0.7071, 0.7071)로 계수가 같으므로 실수부의 표 2에서 구현하였으므로 허수부 구현에서는 구현하지 않는다.

[표 3] 두 번째 스테이지의 허수부 CSD형 계수

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.0245	0	0	0	0	0	1	0	N	0	0	1	0	0	0	1	0
0.0491	0	0	0	0	1	0	N	0	0	1	0	0	1	0	0	0
0.0736	0	0	0	0	1	0	1	0	N	0	0	N	0	N	0	N
0.0980	0	0	0	1	0	N	0	0	1	0	0	1	0	N	0	N
0.1224	0	0	0	1	0	0	0	0	N	0	1	0	1	0	1	0
0.1467	0	0	0	1	0	1	0	N	0	N	0	0	1	0	0	N
0.1710	0	0	1	0	N	0	N	0	0	0	N	0	0	1	0	N
0.1951	0	0	1	0	N	0	0	1	0	0	0	0	N	0	0	1
0.2191	0	0	1	0	0	N	0	0	0	0	0	1	0	N	0	N
0.2430	0	0	1	0	0	0	0	N	0	0	1	0	N	0	1	0
0.2667	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	N
0.2903	0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	0
0.3137	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	N
0.3369	0	1	0	N	0	N	0	N	0	0	1	0	0	0	0	N
0.3599	0	1	0	N	0	0	N	0	0	0	0	1	0	0	0	1
0.3827	0	1	0	N	0	0	0	1	0	0	0	0	0	N	0	0
0.4052	0	1	0	N	0	1	0	0	0	0	N	0	0	0	N	0
0.4276	0	1	0	0	N	0	0	N	0	N	0	0	0	N	0	N
0.4496	0	1	0	0	N	0	1	0	N	0	0	1	0	N	0	0
0.4714	0	1	0	0	0	N	0	0	1	0	N	0	N	0	N	0
0.4929	0	1	0	0	0	0	0	N	0	0	1	0	N	0	0	N
0.5141	0	1	0	0	0	0	1	0	0	N	0	1	0	0	N	0
0.5350	0	1	0	0	0	1	0	0	1	0	0	0	N	0	1	0
0.5556	0	1	0	0	1	0	0	N	0	0	1	0	0	N	0	1
0.5758	0	1	0	0	1	0	1	0	0	N	0	N	0	1	0	N
0.5957	0	1	0	1	0	N	0	0	0	1	0	0	0	0	0	N
0.6152	0	1	0	1	0	0	0	N	0	N	0	0	0	0	N	0
0.6344	0	1	0	1	0	0	0	1	0	1	0	N	0	1	0	0
0.6532	0	1	0	1	0	1	0	0	N	0	1	0	0	N	0	0
0.6716	1	0	N	0	N	0	N	0	0	0	0	0	N	0	N	0
0.6895	1	0	N	0	N	0	0	0	0	1	0	0	0	0	0	1

표 3의 CSD형 계수를 덧셈기를 사용하여 구현하면 135개의 덧셈기가 필요하게 된다. 이미 정의된 4개의 공통패턴으로 31개의 Twiddle Factor 허수부를 식 (15)와 같이 나타낼 수 있다.

$$
\begin{aligned}
i_1 &= x_3 \gg 5 + x_1 \gg 10 + x_1 \gg 14 \\
i_2 &= x_3 \gg 4 + x_4 \gg 9 \\
i_3 &= x_2 \gg 4 - x_4 \gg 8 - x_2 \gg 13 \\
i_4 &= x_3 \gg 3 + x_4 \gg 8 - x_2 \gg 13 \\
i_5 &= x_1 \gg 3 - x_3 \gg 8 + x_2 \gg 12 \\
i_6 &= x_2 \gg 3 - x_2 \gg 7 + x_5 \gg 12 \\
i_7 &= x_3 \gg 2 - x_1 \gg 6 - x_5 \gg 10 - x_1 \gg 15 \\
i_8 &= x_3 \gg 2 + x_1 \gg 7 - x_5 \gg 12 \\
i_9 &= x_5 \gg 2 + x_3 \gg 11 - x_1 \gg 15 \\
i_{10} &= x_1 \gg 2 - x_5 \gg 7 - x_3 \gg 12 \\
i_{11} &= x_1 \gg 2 + x_1 \gg 6 + x_4 \gg 10 - x_1 \gg 15 \\
i_{12} &= x_4 \gg 2 + x_4 \gg 7 + x_1 \gg 12 \\
i_{13} &= x_2 \gg 2 + x_2 \gg 10 - x_1 \gg 15 \\
i_{14} &= x_3 \gg 1 - x_2 \gg 5 + x_1 \gg 10 - x_1 \gg 15
\end{aligned}
$$

$$
\begin{aligned}
i_{15} &= x_3 \gg 1 - x_1 \gg 6 + x_1 \gg 11 + x_1 \gg 15 \\
i_{16} &= x_3 \gg 1 + x_1 \gg 7 - x_1 \gg 13 \\
i_{17} &= x_3 \gg 1 + x_1 \gg 5 - x_1 \gg 10 - x_1 \gg 14 \\
i_{18} &= x_5 \gg 1 - x_2 \gg 7 - x_2 \gg 13 \\
i_{19} &= x_5 \gg 1 + x_3 \gg 6 + x_3 \gg 11 \\
i_{20} &= x_1 \gg 1 - x_5 \gg 5 - x_2 \gg 10 - x_1 \gg 14 \\
i_{21} &= x_1 \gg 1 - x_5 \gg 7 - x_4 \gg 12 \\
i_{22} &= x_1 \gg 1 + x_5 \gg 6 + x_5 \gg 11 \\
i_{23} &= x_1 \gg 1 + x_4 \gg 5 - x_3 \gg 12 \\
i_{24} &= x_4 \gg 1 - x_5 \gg 7 - x_3 \gg 13 \\
i_{25} &= x_4 \gg 1 + x_5 \gg 6 - x_3 \gg 11 - x_1 \gg 15 \\
i_{26} &= x_2 \gg 1 - x_1 \gg 5 + x_1 \gg 9 - x_1 \gg 15 \\
i_{27} &= x_2 \gg 1 - x_2 \gg 7 - x_1 \gg 14 \\
i_{28} &= x_2 \gg 1 + x_2 \gg 7 - x_3 \gg 11 \\
i_{29} &= x_2 \gg 1 - x_5 \gg 5 - x_5 \gg 10 \\
i_{30} &= x_3 - x_2 \gg 4 - x_2 \gg 12 \\
i_{31} &= x_3 - x_1 \gg 4 + x_1 \gg 9 + x_1 \gg 15
\end{aligned}
\tag{15}
$$

식 (15)의 31개의 Twiddle Factor 허수부를 덧셈기와 쉬프트를 사용하여 설계한 CSS 구조는 그림 8과 같다. 그림 8의 허수부 곱셈기에서 사용되는 덧셈기는 74이다. 즉 CSD형 계수를 그대로 사용할 때보다 61개를 줄일 수 있었다. 그림 7과 그림 8에서 보듯이 초기 입력 값을 사용하여 공통 패턴을 만들고 이 공통패턴 x_2, x_3, x_4, x_5 과 초기입력 값을 이용하여 쉬프트와 덧셈기를 통해 63개의 출력이 나오도록 설계하였다.

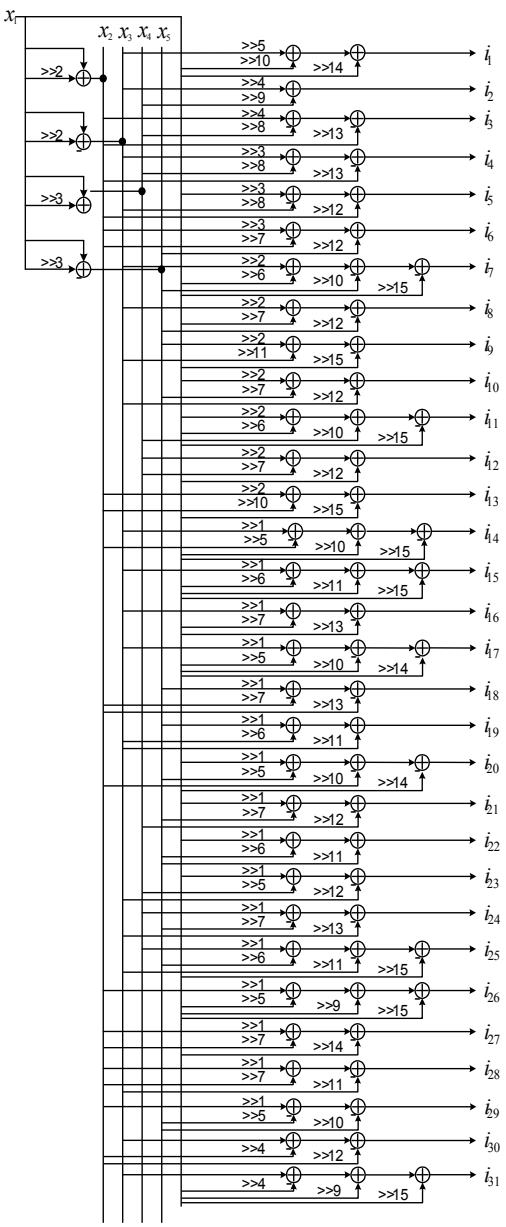
3.4 세 번째 스테이지 설계

Radix-4^2 알고리즘을 사용하여 두 스테이지를 설계하였으므로 이제 16-point FFT가 남았다. 이에 대하여 다시 Radix-4^2 알고리즘을 적용하기로 한다.

세 번째 스테이지의 버터플라이에서 계산해야 할 연산은 첫 번째 스테이지의 연산인 식 (9)와 정확히 같다. 따라서 버터플라이의 구조는 그림 2와 같게 된다. 이 스테이지에서 구현해야 하는 곱셈의 가지 수도 역시 첫 번째 스테이지와 같이 0.7071, 0.9239, 0.3827의 3개의 수밖에 없다. 오직 3개의 곱셈만 사용되므로 덧셈기와 쉬프트를 사용하여 구현하였다. 곱셈용 계수는 CSD형의 계수를 사용하였다.

3.5 네 번째 스테이지 설계

네 번째 스테이지는 마지막 스테이지이므로 곱셈 연산 블록이 없다. 따라서 DC4의 출력 값이 버터플라이로 들어와 덧셈 연산을 수행하고 최종 출력이 된다. 이 스테이지의 덧셈 블록은 세 번째 스테이지의 덧셈 블록과 같으므로 여기에서는 생략한다.



[그림 8] 제안된 Twiddle Factor 허수부 곱셈 구조

4. 구현

이 절에서는 제안된 FFT 프로세서의 효율성을 기존 구조와 비교하여 살펴보기로 한다. 먼저 제안구조의 알고리즘을 통하여 이론적으로 곱셈연산과 덧셈연산의 수가 얼마나 감소 할 수 있는지를 알아보고 다음으로 합성을 통하여 면적이 실제 얼마나 감소하였는지 실험을 통하여

알아보기로 한다.

4.1 곱셈연산의 가지 수 및 덧셈연산의 수 비교

덧셈기를 사용한 효율적인 FFT 구조를 설계하기 위하여 곱셈연산의 가지 수가 적어야하므로 우리는 Radix-4^2 알고리즘을 선택하였다. 256-point FFT에 대한 곱셈 연산의 가지 수 비교는 표 4와 같다.

[표 4] 256-point FFT에 대한 곱셈연산의 가지 수 비교

	radix-4	Radix-4^2 (proposed)
1st stage	63	3
2nd stage	15	63
3rd stage	3	3
total	81(100%)	69(85.19%)

표 4에서 보듯이 Radix-4^2 제안구조의 첫 번째와 세 번째 스테이지는 각각 3가지의 계수를 필요로 하며 두 번째 스테이지는 63가지 계수가 필요하여 총 69가지 계수를 필요로 한다. 이와 비교하여 Radix-4의 256-point FFT는 81가지의 계수를 필요하므로 이 논문에서 Radix-4^2 알고리즘을 선택하였다.

이제 곱셈연산을 덧셈기와 쉬프트 연산기를 사용하여 구현해보기로 한다. Radix-4^2 구조에서 사용되는 69개의 곱셈기를 CSD형으로 나타낸 후에 CSS 기법을 사용해 설계한 결과는 표 5와 같다.

[표 5] 곱셈연산 구현을 위한 덧셈기의 수 (256-point Radix-4^2 FFT 구조)

	2's complement	CSD type coefficient	CSS (proposed)
1st stage	18	15	8
2nd stage	456	277	159
3rd stage	18	15	8
total	492	307	175

표 5에서 보듯이 덧셈기를 사용하여 곱셈연산을 효율적으로 구현하기 위해서 CSS 방식을 사용한 결과 175개의 덧셈기를 사용하여 구현할 수 있었다. 즉 CSD형 계수를 사용하는 구조보다 132개의 덧셈기를 감소시킬 수 있었다.

4.2 Verilog-HDL 시뮬레이션과 합성

제안한 FFT 프로세서의 구현면적 감소에 관한 효율성을 시뮬레이션 하기위해 그림 1과 같은 파이프라인 256-point Radix-4^2 MDC 구조를 구현하였다. 제안구조의 효율성을 비교하기 위해 256-point Radix-4 MDC 구조를 구현하여 비교하였다.

Matlab을 사용하여 제안된 256-point Radix-4^2 MDC 구조의 function을 검증하였다. 즉 각각의 스테이지 출력과 최종 256개 출력이 정확한지 검증하였다. 여기에서 만들어진 각 스테이지 출력 값은 Verilog-HDL 코딩의 검증을 위한 테스트벡터로 사용하였다.

이후 Xilinx ISE 7.1i로 Verilog-HDL 코딩을 하고 Modelsim을 이용하여 나온 출력 값을 테스트벡터와 비교하여 HDL 코딩을 검증하였다.

Verilog HDL로 설계한 FFT 프로세서를 Chartered $0.18\mu m$ CMOS 셀 라이브러리를 이용하여 합성하였고 두 구조 모두 동일한 constraints를 적용하였다. 기존 구조와 제안 구조의 복소 곱셈 블록에 대한 합성 결과는 표 6과 같다.

[표 6] 복소 곱셈블록의 cell area 비교(mm^2)

	Radix-4	Radix-4^2 (proposed)
1st stage	0.721	0.030
2nd stage	0.199	0.606
3rd stage	0.031	0.030
total	0.952 (100%)	0.667 (70.1%)

표 6에서 보듯이 기존 256-point Radix-4 구조의 곱셈연산 블록은 $0.952mm^2$로 합성되었으며 제안된 256-point Radix-4^2 구조의 곱셈연산 블록은 $0.667mm^2$로 합성되었다. 제안된 곱셈연산 블록의 구현 면적은 기존 블록과 비교하여 약 29.9% 감소됨을 알 수 있다.

이제 256-point FFT 구조 전체에 대한 면적을 비교해보기로 한다. 곱셈연산 블록, DC 블록, 덧셈기 블록 등 전체 256-point FFT 구조에 대한 합성한 결과는 표 7과 같다.

[표 7] 제안된 256-point FFT 구조의 면적 비교(mm^2)

	Radix-4			Radix-4^2 (proposed)		
	DC	A	M	DC	A	M
1 stage	0.596	0.028	0.721	0.571	0.027	0.030
2 stage	0.781	0.028	0.199	0.762	0.027	0.606
3 stage	0.235	0.028	0.031	0.224	0.027	0.030
4 stage	0.064	0.028		0.061	0.027	
Total	2.746 (100%)			2.402 (87.46%)		

표 7에서 보듯이 기존의 구조는 2.746mm^2로 합성되었으며 제안구조는 2.402mm^2로 합성되었다. 제안된 구조가 기존 구조와 비교하여 면적이 12.54% 감소된 것을 확인 할 수 있다.

V. 결 론

이 논문에서 OFDM 시스템에서 큰 면적을 차지하고 있는 FFT 블록에 대하여 CSD 곱셈기 방식의 새로운 FFT 구조를 제안하였다. 덧셈기 만을 사용하여 구현하기 위해서는 곱셈연산의 가지 수가 가장 적은 Radix-4^2 알고리즘을 선택함으로써 곱셈연산의 가지 수를 14.81% 줄일 수 있었다.

또한 덧셈 연산의 효율적인 구현을 위하여 CSD 방식과 CSS 방식을 사용하여 구현하였다. HDL 코딩과 Chartered 0.18μm 공정으로 논리합성하여 cell area를 비교한 결과 12.54%의 면적 감소 효과를 입증하였다.

참고문헌

[1] R. Sarmiento, V. D. Armas, J. F. Lopez, J. A. Montiel-Nelson, and A. Nunez, “A CORDIC processor for FFT computation and its implementation using gallium arsenide technology", IEEE Trans. on VLSI Systems, vol. 6, No. 1, pp. 18-30, Mar. 1998.

[2] M. Bekooij, J. Huisken, and K. Nowak, “Numerical accuracy of Fast Fourier Transforms with CORDIC arithmetic", Journal of VLSI Signal Processing 25, pp. 187-193, 2000.

[3] S. M Kim, J. G. Chung, and K. K. Parhi, "Low error Fixed-width CSD Multiplier with Efficient Sign Extension", IEEE Trans. Circuits and Systems-II, vol. 50, No. 12, Dec. 2003.

[4] R. I. Hartley, "Sub-expression sharing in filters using canonic signed digit multipliers", IEEE Trans. Circuits and Systems-II: Analog and Digital Signal Processing, vol. 43, No.10, pp. 677-688, Oct. 1996.

[5] Y. Jang, and S. Yang, "Low-power CSD linear phase FIR filter structure using vertical common sub-expression" IEE Electronics Letters, vol. 38, No. 15, pp. 777-779, Jul. 2002.

[6] J. Lee and H. Lee, "High-Speed 2-Parallel Radix-24 FFT/IFFT Processor for MB-OFDM UWB Systems", IEICE Trans. on Fundamentals of Electronics, communications, and Computer Science, vol. E91-A, No. 4, pp. 1206-1211, April, 2008.

[7] J. Y. Oh, J. S. Cha, S. K. Kim, and M. S. Lim, "Implementation of Orthogonal Frequency Division Multiplexing using radix-N Pipeline Fast Fourier Transform(FFT) Processor", Jpn. J. Appl. Phys., vol. 42, No. 4B, pp. 1-6, April, 2003.

[8] 최동규, 장영범, “Common sub-expression sharing과 CORDIC을 이용한 OFDM 시스템의 저면적 파이프라인 FFT 구조”, 전자공학회논문지, 제46권, SP편 제 4호, pp. 157-164, 2009.

장 영 범(Young-Beom Jang) [정회원]

- 1981년 2월 : 연세대학교 전기공학과(공학사)
- 1990년 1월 : Polytechnic University, EE(공학석사)
- 1994년 1월 : Polytechnic University, EE(공학박사)
- 1983년 ~ 1999년 : 삼성전자 System LSI 사업부 수석연구원.
- 2000년 ~ 2002년 : 이화여자대학교 정보통신학과 연구교수.
- 2002년 ~ 현재 : 상명대학교 정보통신공학과 교수.

<관심분야>
통신신호처리, 비디오신호처리, SoC 설계

이 동 훈(Dong-Hoon Lee) [준회원]

- 2002년 ~ 2009년 2월 : 상명대학교 정보통신공학과 (공학사).
- 2009년 3월 ~ 현재 : 상명대학교 일반대학원 컴퓨터정보통신공학과 석사과정

<관심분야>
통신신호처리, SoC 설계