WAVE 시스템에서 행렬 테이블로 연산하기 위한 알고리즘 설계 및 구현

정회원 이 대 식*°, 유 영 모*, 이 상 윤*, 장 청 룡**

The Algorithm Design and Implemention for Operation using a Matrix Table in the WAVE system

Dae-sik Lee*°, Young-mo-You*, SangYoun Lee*, Chung-ryong Jang** ***Regular Members***

요 약

WAVE(Wireless Access for Vehicular Environment) 시스템은 차량용 통신 기술로서, 차량 운전 중 발생 가능한 사고들을 미연에 방지하기 위한 서비스와 차량기능 관리, 시스템 장애를 모니터링하는 각종 서비스를 제공하기 위해 사용된다. 그러나 WAVE 시스템의 스크램블러 비트 연산은 병렬 처리가 불가능하므로 소프트웨어나 하드웨어 설계의 효율성이 떨어지게 된다. 본 논문에서는 스크램블러의 비트 연산 과정으로 행렬 테이블을 구성하는 알고리즘과 입력 데이터와 행렬 테이블을 병렬 연산하는 알고리즘을 제안한다. 본 논문에서 제안한 스크램블러 알고리즘은 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트냐에 따라 처리 속도가 다르지만 입력 단위에 따라 병렬 처리가 가능하므로 WAVE 시스템의 처리 속도를 더욱 향상시킨다.

Key Words : OFDM(Orthogonal Frequency Division Multiplexing), WAVE System시스템(Wireless Access for Vehicular Environment System), 스크램블러(Scrambler), 행렬 테이블(Matrix Table), IEEE 802.11p

ABSTRACT

A WAVE(Wireless Access for Vehicular Environment) system is a vehicle communication technology. The system provides the services to prevent vehicle accidents that might occur during driving. Also, it is used to provide various services such as monitoring vehicle management and system failure. However, the scrambler bit operation of WAVE system becomes less efficient in the organizations of software and hardware design because the parallel processing is impossible. Although scrambler algorithm proposed in this paper has different processing speed depending on input data 8 bit, 16 bit, 32 bit, and 64 bit. it improves the processing speed of the operation because it can make parallel processing possible depending on the input unit.

Ⅰ. 서 론

WAVE(Wireless Access for Vehicular Environment) 시스템은 차량 단말기를 보유한 이용자에게 교통정보, 위치정보 및 안전에 관한 정보 등 다양한 서비스를 제공하기 위한 시스템으로 단거리 고속무선패킷통신 분야를 담당하는 통신 기술이다[1].

* 트라이콤텍(주) 부설 연구소({daesik, youngmo, sangyoon}@tricomtek.com), (° : 교신저자)
** 경동대학교 IT공학부(crjang@k1.ac.kr)
논문번호 : KICS2012-01-011, 접수일자 : 2012년 1월 10일, 최종논문접수일자 : 2012년 4월 5일

현재 미국을 중심으로 일본, 유럽, 한국 등에서 개발 추진 중이다.

현재 추진하고 있는 WAVE 시스템은 IEEE 802.11p 표준 규격의 무선기술을 기반으로 고속 데이터를 전송하고, 낮은 Latency, 지역적인 서비스 특화 및 고속 이동시 필요한 다양한 정보서비스 제공에 적합하게 설계되었다. 따라서 교통정보 수집 및 활용, 차량운전 중 전방에서의 사고 및 긴급 상황, 교통 흐름 제어, 교통정보 제공 인접지역에 대한 여행자 정보, 자동요금 징수, 대중교통 관리를 위한 운행안내 시스템, 화물차량 관리 및 교차로 충돌회피용, 차량데이터 전송 및 자동고속도로 시스템 구축용 등으로 활용되고 있다. 하지만, WAVE 시스템의 스크램블러 비트 연산은 병렬 처리가 불가능하므로 소프트웨어나 하드웨어 설계의 효율성이 떨어지게 된다.

본 논문에서는 WAVE 시스템을 스크램블러의 비트 연산으로 행렬 테이블을 구성하는 알고리즘과 입력 데이터와 행렬 테이블을 병렬 처리하는 알고리즘을 제안한다. 특히, 스크램블러의 비트 연산으로 행렬 테이블을 구성하여 입력 데이터의 입력 단위와 행렬 테이블의 단위로 병렬 처리하는 알고리즘을 비교 분석해 보고자 한다.

본 논문의 구성은 2장에서 관련연구인 WAVE 시스템의 물리계층과 IEEE 802.11p 표준 규격의 스크램블러를 살펴보고 3장에서 본 논문에서 제안하는 스크램블러 알고리즘을 설명한다. 그리고 4장에서 실험 결과를 설명하고 5장에서 결론을 맺는다.

Ⅱ. 관련연구

2.1. WAVE 시스템의 물리계층

OFDM(Orthogonal Frequency Division Multiplexing)[2,3] 모듈레이션은 순서대로 스크램블러, 콘볼루션, 인터리버, 주파수 변조 기법에 따른 심볼 값 매핑, 파일롯 추가, IFFT (Inverse Fast Fourier Transform) 연산, GI(Guard Interval) 추가, 심볼 정형, IQ 모드 과정을 거친 후 RF(Radio Frequency)를 통해 데이터가 전달된다.

WAVE 물리계층에서 OFDM 모듈레이션은 그림 1과 같다[4,5].

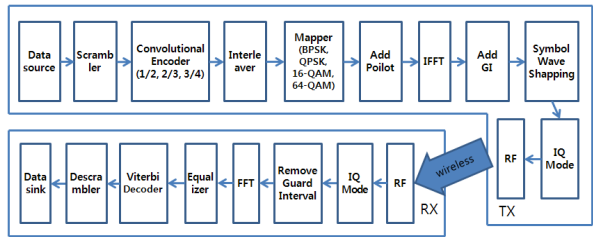


그림 1. OFDM 모듈레이션
Fig. 1. OFDM Modulation

그림 1에서 보면 데이터 소스가 전달되면 데이터 암호화, 데이터의 연속성이나 반복 패턴 등의 현상을 방지하고자 스크램블러 연산을 한다. 다음 콘볼루션 엔코더는 무선상의 산발적인 에러에 강한 코딩을 하고, 모듈레이션에 따라 1/2, 2/3, 3/4 Rate로 구성된다. 콘볼루션 엔코딩이 완료된 데이터는 무선상 블록 에러를 산발 에러로 바꾸어 줄 수 있는 인터리버를 통과하고, 인터리버는 모듈레이션에 따라 레지스터 용량이 다르게 적용되어 데이터 비트가 섞이게 된다. 다음은 각각의 주파수 변조 기법에 따라 비트를 묶어 IQ 데이터로 매핑하고 도플러 현상과 신호의 세기를 통해 실시간으로 채널 조정을 해주기 위해 파일롯 신호를 추가하여 FFT 연산을 한다. FFT 연산을 통해 생성된 각각의 심볼의 앞부분에 심볼간 간섭을 억제하기 위해 FFT 심볼 크기의 1/4에 해당하는 GI를 추가한다. 생성된 OFDM 심볼들은 심볼 값에 따라 주파수 위상이 크게 변하는 것을 막기 위한 정형과정을 거치고, IQ 모드에서 반송파와 곱해진다. 수신 OFDM 디모듈레이션은 송신 OFDM 모듈레이션의 역으로 이루어진다.

2.2. IEEE 802.11p 표준 규격의 스크램블러

스크램블러는 데이터의 암호화, 반복적 패턴을 갖는 비트 블록 방지, 비트 값들 연속성(Null Data) 방지 등을 보장하기 위해 사용된다. WAVE 시스템의 스크램블러는 7개의 쉬프트 레지스터와 XOR 연산을 포함한다[6-8]. 스크램블러의 하드웨어 설계는 그림 2와 같다.

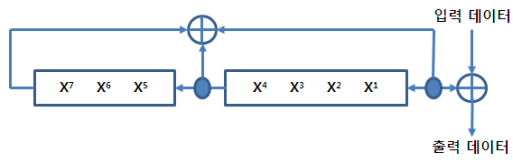


그림 2. 스크램블러의 하드웨어 설계
Fig. 2. Hardware Design of Scrambler

그림 2를 구현하는 알고리즘을 제시하면 알고리즘 1과 같다.

```
Void Gen_scrambler_calculation( ) {
/* 스크램블러의 비트 연산 함수 */
Uint8 seed_src = seed, seed_val = 0;
/* 초기값 설정 */
int i, ii = 0;
Uint8* src, val;
src = input;
val = output;
for(ii = 0 ; ii < input_size ; ii++)  {
    for(i = 0 ; i < 8 ; i++)  {
    seed_val = ((seed_src & 0x08)/0x08) ^
    ((seed_src & 0x40)/0x40);
    /* x4와 x7을 XOR 연산 */
    seed_src = (seed_src * 0x02) + seed_val;
    /* 시프트 연산과 x1 추가 */
    *val += (*src<<(i % 8)) ^
    (seed_val<<(i % 8));  } /* 입력 데이터와
    seed_val을 XOR 연산하여 출력 */
    val++;    src++;  }  }
```

알고리즘 1 스크램블러의 비트 연산 알고리즘

알고리즘 1은 Gen_scrambler_calculation 함수로 스크램블러의 비트 연산을 한다. 스크램블러의 비트 연산에서 7개의 레지스터에 입력되는 초기값 seed를 seed_src로 할당하고, 변수 seed_val, i, ii의 초기값은 0으로 할당한다. 또한 입력 데이터는 src, 출력 데이터는 val에 할당한다.

알고리즘 1을 적용하여 초기값 seed "1111111"을 7개의 레지스터에 입력하면 그림 3과 같다.

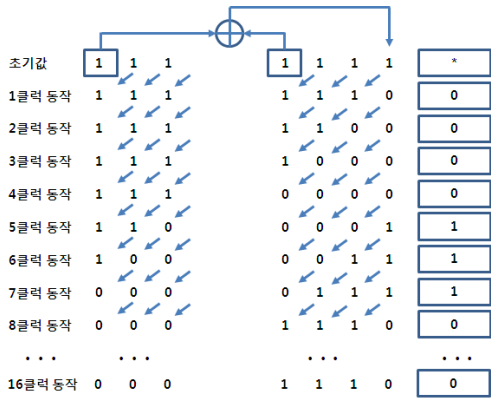


그림 3. 스크램블러의 비트 연산
Fig. 3. Bit Operation of Scrambler

그림 3의 스크램블러의 비트 연산에서 입력 데이터는 "10000010 01000010" 16비트로 예를 들어 설명하면 다음과 같다.

초기단계는 초기값 seed_src "1111111"을 7개의 레지스터에 입력한다.

1클럭 동작으로 7개의 레지스터에 입력된 초기값 "1111111"에서 x4와 x7을 XOR 연산하여 생성된 스크램블러의 비트 "0"을 seed_val에 할당하고, 시프트 연산으로 스크램블러의 비트 "0"을 X1에 추가하여 2클럭 동작을 위해 seed_ src에 할당한다. 입력 데이터의 첫 번째 비트 "1"과 스크램블러의 비트 "0"을 XOR 연산하여 출력 데이터 비트 "1"을 val에 저장한다.

2클럭 동작으로 seed_src "1111110"을 7개의 레지스터에 입력하면 x^4와 x^7을 XOR 연산하여 생성된 스크램블러의 비트 "0"을 seed_ val에 할당하고, 시프트 연산으로 스크램블러의 비트 "0"을 X1에 추가하여 3클럭 동작을 위해 seed_src에 할당한다. 입력 데이터의 두 번째 비트 "0"과 스크램블러의 비트 "0"을 XOR 연산하여 출력 데이터 비트 "0"을 val에 저장한다.

3클럭 동작으로 seed_src "1111100"을 7개의 레지스터에 입력하면 x^4와 x^7을 XOR 연산하여 생성된 스크램블러의 비트 "0"을 seed_ val에 할당하고, 시프트 연산으로 스크램블러의 비트 "0"을 X^1에 추가하여 4클럭 동작을 위해 seed_src에 할당한다. 입력 데이터의 세 번째 비트 "0"과 스크램블러의 비트 "0"을 XOR 연산하여 출력 데이터 비트 "0"을 val에 저장한다.

반복 동작으로 입력 데이터 비트 수만큼 16클럭까지 동작한다.

따라서 입력 데이터 "10000010 01000010"은 16비트이므로 16클럭까지 동작하여 각 클럭마다 스크램블러의 비트를 생성하여 16개의 입력 데이터 비트와 한비트씩 XOR 연산을 하여 출력 데이터 16비트 "10001100 1011000 0"을 생성한다.

하드웨어나 소프트웨어 측면에서 스크램블러의 비트 연산은 입력 데이터의 비트 수만큼 반복문의 Loop 횟수가 비정상적으로 많아지고, 프로세서가 32비트 이상 지원되어도 8비트 프로세서와 처리속도는 같다. 결과적으로 스크램블러 처리속도를 높이기 위해 본 논문에서는 스크램블러의 비트 연산으로 행렬 테이블을 구성하는 알고리즘과 입력 데이터와 행렬 테이블을 병렬 처리하는 알고리즘을 제안한다.

Ⅲ. 제안한 스크램블러 알고리즘

IEEE 802.11p 표준 규격 스크램블러의 비트 연산은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하므로 효율성이 떨어진다.

본 논문에서는 스크램블러의 비트 연산으로 행렬 테이블을 구성하여 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 병렬처리 할 수 있는 알고리즘을 제안한다.

제안한 스크램블러의 하드웨어 설계는 그림 4와 같으며 입력 데이터의 입력 단위를 8비트 기준으로 설계한 것이다.

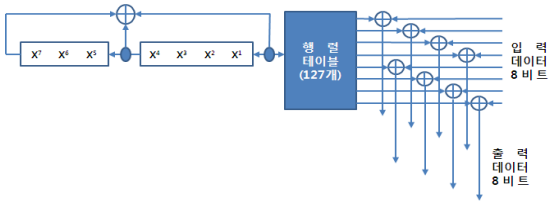


그림 4. 제안한 스크램블러의 하드웨어 설계
Fig. 4. Hardware design of the proposed scrambler

그림 4는 초기값 "x^7 x^6 x^5 x^4 x^3 x^2 x^1"이 7개 레지스터에 입력되어 스크램블러의 비트 연산으로 메모리에 127개의 서로 다른 비트열로 행렬 테이블을 구성한다. 127개의 서로 다른 비트열로 구성된 행렬 테이블의 8비트 단위가 입력 데이터의 입력 단위 8비트와 병렬처리 할 수 있다. 따라서 그림 4와 같이 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 병렬처리 할 수 있는 알고리즘을 제안한다.

그림 2의 하드웨어 설계는 입력 데이터가 16비트이면 각 클럭의 동작으로 스크램블러의 비트를 생성하여 한비트씩 입력 데이터와 XOR 연산을 16번 처리한다. 하지만 그림 4에서 제안한 하드웨어 설계는 입력 데이터의 입력 단위가 8비트이므로 행렬 테이블의 8비트 단위와 병렬로 XOR 연산을 2번 처리한다.

입력 데이터의 입력 단위가 8비트인 그림 4를 구현하는 알고리즘을 제시하면 알고리즘 2, 알고리즘 3과 같다.

알고리즘 2는 스크램블러의 비트 연산으로 행렬 테이블을 구성하는 알고리즘이다.

```
void Gen_scrambler_table(void)  {
/* 행렬 테이블 구성하는 함수 */
Uint8 seed_src = seed, seed_val=0;
Uint8* ran_com = randomizer_table;
int i = 0, ii = 0;
memset(ran_com,0x00,1000);
/* 행렬 테이블 초기화하는 함수 */
for(ii = 0 ; ii < 127 ; ii++)  {
/* 127개의 서로 다른 비트열 계산 */
    for(i = 0 ; i < 8 ; i++)  {
    seed_val = ((seed_src & 0x08)/0x08) ^
    ((seed_src & 0x40)/0x40);
    /* x4와 x7을 XOR 연산 */
    seed_src = (seed_src * 0x02) + seed_val;
    /* 시프트 연산과 x1 추가 */
    *ran_com += seed_val<<(i % 8);  }
    /* 행렬 테이블에 비트열 저장 */
ran_com++;  }  }
```

알고리즘 2 행렬 테이블 구성 알고리즘

알고리즘 2를 보면 Gen_scrambler_table 함수로 127개의 서로 다른 비트열로 행렬 테이블을 구성한다. 스크램블러의 비트 연산으로 행렬 테이블을 구성하기 위해 7개의 레지스터에 입력되는 초기값 seed를 seed_src로 할당하고, 변수 seed_val, i, ii의 초기값은 0으로 할당한다. memset 함수는 행렬 테이블을 초기화하는 함수이고, 행렬 테이블에 비트열을 저장하기 위해 randomizer_table을 포인터 ran_com에 할당한다.

알고리즘 2를 적용하여 초기값 seed "1111111"을 7개 레지스터에 입력하면 그림 3과 같이 스크램블러의 비트 연산으로 입력 데이터의 입력 단위가 8비트이면 8클럭씩 동작하여 16개의 서로 다른 비트열이 생성된다. 입력 데이터의 입력 단위가 8비트로 구성된 16개의 서로 다른 비트열은 표 1과 같다.

표 1. 16개 비트열
Table 1. 16 Bitstring

1111111	00001110	11110010	11001001
00000010	00100110	00101110	10110110
00001100	11010100	11100111	10110100
00101010	11111010	01010001	10111000

표 1과 같이 초기값 seed_src "1111111"을 7개 레지스터에 입력하여 그림 3과 같이 스크램블러의 비트 연산으로 16개의 서로 다른 비트열을 구성하고, 16개의 서로 다른 비트열은 1바이트 메모리 8비트 만큼 순환루프를 반복적으로 실시하면 128(16*8=128)번째 7개 레지스터에 입력되는 비트열은 초기값 "1111111"이 된다. 따라서 입력 데이터의 입력 단위와 같은 크기의 비트열로 127개의 서로 다른 비트열이 행렬 테이블을 구성되어 입력 데이터의 입력 단위와 병렬 처리한다.

알고리즘 2에서 처럼 for 문을 127번의 수행하고, 1번의 for 문으로 8클럭 동작하여 127개의 서로 다

른 비트열이 행렬 테이블을 구성되어 입력 데이터의 전체 크기가 127 바이트 이상이 되어도 스크램블러의 비트 연산을 하지 않고 행렬 테이블과 계속하여 병렬처리 할 수 있다.

입력 데이터의 입력 단위가 8비트로 구성된 127개의 서로 다른 비트열은 표 2와 같다.

표 2. 127개의 서로 다른 비트열로 구성된 행렬 테이블
Table 2. Matrix Table Consisting of 127 Different Bitstring

1111111	00001110	11110010	11001001
00000010	00100110	00101110	10110110
00001100	11010100	11100111	10110100
00101010	11111010	01010001	10111000
11111110	00011101	11100101	10010010
00000100	01001100	01011101	01101100
00011001	10101001	11001111	01101000
01010101	11110100	10100011	01110001
11111100	00111011	11001011	00100100
00001000	10011000	10111010	11011000
00110011	01010011	10011110	11010000
10101011	11101001	01000110	11100011
11111000	01110111	10010110	01001000
00010001	00110001	01110101	10110000
01100110	10100111	00111101	10100001
01010111	11010010	10001101	11000111
11110000	11101111	00101100	10010000
00100010	01100010	11101011	01100000
11001101	01001110	01111011	01000010
10101111	10100101	00011011	10001111
11100001	11011110	01011001	00100000
01000100	11000101	11010110	11000001
10011010	10011100	11110110	10000101
01011111	01001010	00110111	00011111
11000011	10111100	10110010	01000000
10001001	10001011	10101101	10000011
00110101	00111001	11101101	00001010
10111110	10010100	01101110	00111111
10000111	01111001	01100100	10000001
00010011	00010111	01011011	00000110
01101010	01110011	11011010	00010101
01111101	00101000	11011100	01111111

표 2와 같이 행렬 테이블의 8비트 단위와 입력 데이터의 입력 단위 8비트씩 행렬 테이블과 병렬로 XOR 연산을 한다.

알고리즘 2에서 127개의 서로 다른 비트열로 구성된 행렬 테이블을 입력 데이터의 입력 단위 8비트와 병렬 처리하는 알고리즘은 알고리즘 3과 같다.

```
void randomizer( ) {
/* 병렬 처리하는 함수 */
Uint8* ran_com, com;
int i = 0;
Uint8* src, val;
Uint8 counter = 0;
ran_com = randomizer_table;
com = ran_com;
src = input;
val = output;
for(i = 0 ; i < input_size ; i++)  {
/* 행렬 테이블을 병렬로 XOR 연산 */
    *val = *src ^ *com;
    val++;  src++;  com++;  counter++;
    if(counter == 127)  {
        counter = 0;
        com = ran_com;  }  }  }
```

알고리즘 3 입력 데이터와 행렬 테이블을 병렬 연산하는 알고리즘

알고리즘 3을 보면 randomizer 함수로 127개의 서로 다른 비트열로 구성된 행렬 테이블을 입력 데이터의 비트와 병렬 처리한다. ran_com과 com으로 행렬 테이블의 포인터를 지정하고, 변수 i, count, src, val을 지정한다. 127개의 서로 다른 비트열로 구성된 행렬 테이블 randomizer_table을 ran_com에 할당하고, 행렬 테이블의 초기 포인터 위치 ran_com을 com에 할당한다. 또한 입력 데이터는 src, 출력 데이터는 val에 할당한다.

입력 데이터 “10000010 01000010” 16비트와 행렬 테이블 randomizer_table은 초기값 seed “1111111”인 표 2로 예를 들어 설명하면 다음과 같다.

알고리즘 3을 적용하면 입력 데이터의 입력 단위 8비트와 행렬 테이블의 단위 8비트가 병렬로 XOR 연산을 한다. 입력 데이터의 입력 단위 src는 8비트 “10000010”과 행렬 테이블의 8비트 “00001110”을 병렬로 XOR 연산하여 출력 데이터 8비트 “10001100”을 val에 저장하고, src, com, val, count는 1씩 증가한다. 또한 행렬 테이블의 count가 127과 같으면 행렬 테이블의 마지막이므로 count = 0, 행렬 테이블의 초기 포인터 ran_com의 위치를 com으로 할당한다.

다음 입력 데이터의 입력 단위 src는 8비트 “01000010”과 행렬 테이블의 다음 단위 8비트 “11110010”을 병렬로 XOR 연산하여 출력 데이터

8비트 "10110000"을 val에 추가하여 저장하고, src, com, val, count는 1씩 증가한다. 또한 행렬 테이블의 count가 127과 같으면 행렬 테이블의 마지막이므로 count = 0, 행렬 테이블의 초기 포인터 ran_com의 위치를 com으로 할당한다.

알고리즘 3을 적용한 결과 입력 데이터의 전체 비트가 16비트이므로 행렬 테이블의 8비트 단위와 병렬로 XOR 연산하여 출력 데이터 val에 "10001100 10110000"을 생성하고 프로그램을 종료한다.

따라서 IEEE 802.11p 표준 규격 스크램블러의 비트 연산 알고리즘에서 16번 실행하였고, 제안한 스크램블러 알고리즘에서는 2번 실행하였다.

본 논문에서는 127개의 서로 다른 비트열로 구성된 행렬 테이블로 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 병렬로 XOR 연산을 할 수 있으므로 WAVE 시스템의 처리 속도를 더욱 향상시킨다.

Ⅳ. 실험 결과

그림 3의 표준 규격 스크램블러 비트 연산과 표 2와 같이 127개의 서로 다른 비트열로 구성된 행렬 테이블에서 입력 데이터는 "10000010 01000010" 16비트를 적용하여 실험한 결과에서 보듯이 표준 규격 스크램블러의 비트 연산 알고리즘에서 16번 실행하였고, 제안한 스크램블러 알고리즘에서는 2번 실행하였다.

또한 표준 규격 스크램블러의 비트 연산 알고리즘과 제안한 스크램블러 알고리즘을 입력 데이터의 입력 단위로 비교 분석하고, 실제 WAVE DSP(Digital Signal Processor) 보드로 실험하였고, 실험사양은 표 3과 같다.

표 3. 실험 사양
Table 3. Test specifications

시스템	WAVE DSP 보드
운영체제	DSP/BIOS
CPU	TMS320C6455
CPU 속도	1.2GHz
메모리	256MB
언어	C언어

본 논문에서 표준 규격 스크램블러의 비트 연산 알고리즘과 제안한 스크램블러 알고리즘의 타당성을 검증하기 위해 입력 데이터의 입력 단위를 8비트, 16비트, 32비트로 처리하여 실험해 본 결과 표 4, 5, 6에서 알 수 있듯이 제안한 스크램블러 알고리즘이 표준 규격 스크램블러의 비트 연산 알고리즘보다 처리 속도는 감소하고, 처리횟수는 증가하는 것을 알 수 있다. 입력 데이터의 크기는 1080Byte로 하고, 처리 속도나 처리횟수는 10회 측정한 것을 평균값으로 표현하였다.

표 4. 8비트 처리속도와 처리횟수
Table 4. Processing Speed and Counter of 8Bit CPU

8비트 처리속도	비트연산 알고리즘	제안한 알고리즘
1회	약 1.53ms	약 0.95ms
50000회	약 76.77s	약 47.67s
초당 처리횟수	약 653.6회	약 1052.6회

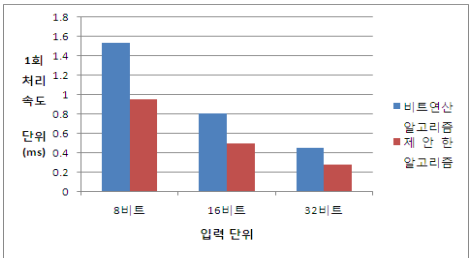
표 5. 16비트 처리속도와 처리횟수
Table 5. Processing Speed and Counter of 16Bit CPU

16비트 처리속도	비트연산 알고리즘	제안한 알고리즘
1회	약 0.81ms	약 0.50ms
50000회	약 40.64s	약 25.24s
초당 처리횟수	약 1234.6회	약 2000.1회

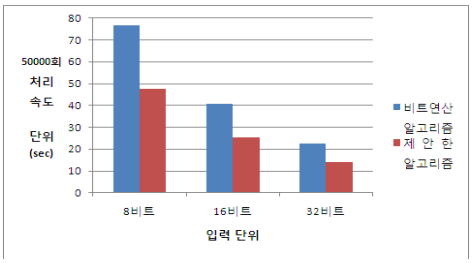
표 6. 32비트 처리속도와 처리횟수
Table 6. Processing Speed and Counter of 32Bit CPU

32비트 처리속도	비트 연산 알고리즘	제안한 알고리즘
1회	약 0.45ms	약 0.28ms
50000회	약 22.58s	약 14.02s
초당 처리횟수	약 2222.2회	약 3571.4회

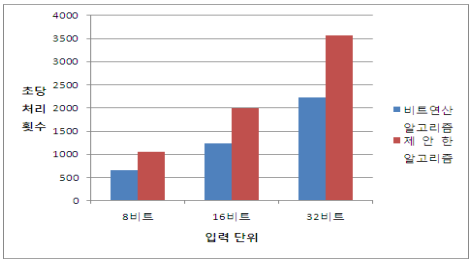
본 논문에서 제안한 스크램블러 알고리즘이 행렬 테이블을 구성하기 위하여 8비트는 127Byte의 스크램블러 비트 연산을 수행하고, 16비트는 254Byte의 스크램블러 비트 연산을 수행하고, 32비트는 508 Byte의 스크램블러 비트 연산을 수행하므로 입력 데이터의 입력 단위를 8비트, 16비트, 32비트로 1회와 50000회 실행한 결과 처리 속도는 약 38%가 향상되고, 초당 처리 횟수는 8비트는 399회, 16비트는 765.5회, 32비트는 1349.2회 더 수행할 수 있다. 따라서 표준 규격 스크램블러의 비트 연산 알고리즘과 비교해 볼 때 제안한 스크램블러 알고리즘의 성능이 향상된 것을 알 수 있다. 그림 5는 표준 규격 스크램블러의 비트 연산 알고리즘과 본 논문에서 제안한 스크램블러 알고리즘의 성능 분석 결과를 나타낸 것이다.



(a) 1회 처리



(b) 50000회 처리



(c) 초당 처리횟수

그림 5. 제안한 알고리즘의 성능분석
Fig. 5. Performance Analysis of the Proposed Algorithm

V. 결 론

WAVE 시스템은 차량단말기를 보유한 이용자에게 교통정보, 위치정보 및 안전에 관한 정보 등 다양한 서비스를 제공하기 위한 시스템으로 다수의 노변기지국, 차량단말기 및 관련 어플리케이션으로 구성된 시스템이다.

WAVE 시스템에서 IEEE 802.11p 표준 규격 스크램블러의 비트 연산 알고리즘은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하므로 효율성이 떨어지게 된다.

본 논문에서는 스크램블러의 비트 연산으로 행렬 테이블을 구성하는 알고리즘과 입력 데이터와 행렬 테이블을 병렬 연산하는 알고리즘을 제안하였다. 제안한 알고리즘은 입력 데이터가 8비트, 16비트, 32비트, 64비트냐에 따라 연산 속도가 다르지만 병렬 처리하므로 WAVE 시스템의 처리 속도를 더욱 향상시킨다.

따라서 본 논문에서 제안한 행렬 테이블을 구성하는 알고리즘과 입력 데이터와 행렬 테이블을 병렬 연산하는 알고리즘은 WAVE 무선통신 시스템 성능을 더욱 업그레이드시켜 시간과 장소에 구애받지 않고 차내 운전자에게 ITS(Intelligent Transport Systems) 서비스의 활용가치를 높이고 교통상황 등 각종 정보수집의 속도와 정밀도를 향상시킬 수 있다.

참 고 문 헌

[1] Huynh Tronganh, Kim Jin Sang, Cho Won Kyung, "Hardware Design for Timing Synchronization of OFDM-Based WAVE Systems," *Korea Information and Communications Society*, Vol.33, No.4, pp.335-486, 2008.

[2] T. M. Schmid l, D. C. Cox, "Robust frequency and timing synchronization for OFDM," *IEEE Trans. Commun*, Vol.45, No.12, pp.1613-1621, 1997.

[3] J. Chuang, N. Sollenberger, "Beyond 3G : Wideband Wireless Data Access Based on OFDM and Dynamic Packet Assignment," *IEEE Comm. Mag*, Vol. 38, No 7, 2000.

[4] Kwak Jae Min, "Implementation of Embedded System for IEEE802.11p based OFDM-DSRC Communications," *The Korean Institute of Information and Commucation Engineering*, Vol.10, No.11, pp.2062-2068, 2006.

[5] Jeongwook Seo, Jae-Min Kwak, Dong Ku Kim, "Simulation Performance of WAVE System with Combined DD-CE and LMMSE Smoothing Scheme in Small-Scale Fading Models," *INTERNATIONAL JOURNAL OF KIMICS*, Vol.8, No.3, pp.281-288, 2010.

[6] ASTM Designation : E 2213-02e13, " Standard Specification for Telecommunications and Information Exchange Between Roadside and Vehicle Systems 5 GHz Band Dedicated Short Range Communications (DSRC) Medium Access Control (MAC) and Physical Layer (PHY) Specifications1," 2003.

[7] IEEE std 802.11, "Wireless LAN Medium Access Control(MAC) and Physical Layer (PHY) specifications," 2007.
[8] IEEE std 802.11pTM/D11.0, "Wireless LAN Medium Access Control (MAC) and Physical Layer(PHY) specifications Amendment 6: Wireless Access in Vehicular Environments," 2010.

이 대 식 (Dae-sik Lee) 정회원

1995년 2월 관동대학교 전자계산공학과 졸업
1999년 8월 관동대학교 전자계산공학과 석사
2004년 2월 관동대학교 전자계산공학과 박사
2005년~2007년 안동과학대학 사이버테러대응과 전임강사
2011년~현재 (주)트라이콤텍 연구소장
<관심분야> 무선/이동 전송, 데이터 통신, 차세대 이동통신, 네트워크 보안

유 영 모 (Young-mo You) 정회원

2010년 2월 경동대학교 멀티미디어 통신학과 졸업
2009년~현재 (주)트라이콤텍 연구원
<관심분야> 차세대 이동통신, 데이터 통신, 유비쿼터스, 네트워크 보안

이 상 윤 (SangYoon Lee) 정회원

1995년 2월 충북대학교 통계학과 졸업
1996년~1999년 텔넷코리아 연구원
1999년~현재 (주)트라이콤텍 엔지니어 총괄 팀장
<관심분야> 무선/이동 전송, 차세대 이동통신, 데이터 통신, 유비쿼터스, 네트워크 보안

장 청 룡 (Chung-ryong Jang) 정회원

1980년 2월 성균관대학교 전자 공학과 졸업
1986년 8월 성균관대학교 전자 공학 석사
1995년 2월 성균관대학교 정보공학 박사
1979년~1983년 한국전자통신 연구원
1984년~1997년 한국통신 연구개발본부 선임연구원
1997년~현재 경동대학교 IT 공학부 부교수
<관심분야> 통신망 보안, 암호/인증 기법, 보안 기술 표준화