Journal of Sensor Science and Technology
Vol. 24, No. 3 (2015) pp. 188-193
http://dx.doi.org/10.5369/JSST.2015.24.3.188
pISSN 1225-5475/eISSN 2093-7563

센서네트워크 활용을 위한 경량 병렬 BCH 디코더 설계

최원정 · 이제훈+

Design of Lightweight Parallel BCH Decoder for Sensor Network

Won-Jung Choi and Je-Hoon Lee+

Abstract

This paper presents a new byte-wise BCH (4122, 4096, 2) decoder, which treats byte-wise parallel operations so as to enhance its throughput. In particular, we evaluate the parallel processing technique for the most time-consuming components such as syndrome generator and Chien search owing to the iterative operations. Even though a syndrome generator is based on the conventional LFSR architecture, it allows eight consecutive bit inputs in parallel and it treats them in a cycle. Thus, it can reduce the number of cycles that are needed. In addition, a Chien search eliminates the redundant operations to reduce the hardware complexity. The proposed BCH decoder is implemented with VHDL and it is verified using a Xilinx FPGA. From the simulation results, the proposed BCH decoder can enhance the throughput as 43% and it can reduce the hardware complexity as 67% compared to its counterpart employing parallel processing architecture.

Keywords: BCH code, Parallel BCH decoder, Syndrome generator, Chien search

1. 서 론

최근 다양한 센서들의 네트워크를 이용하여 여러 정보를 추출, 가공 및 종합하여 새로운 어플리케이션을 개발하는 시도가 많아지고 있다. 특히 사물인터넷의 경우 여러 무선 센서 노드들의 데이터 전송이 필수적이다. 데이터 전송 오류가 발생될 경우 이를 검출하여 재전송을 요구하거나, 스스로 오류를 정정할 수 있는 오류 정정 부호를 사용하며, 복잡한 데이터 송수신을 요구하는 센서 네트워크의 경우 이러한 오류 정정 부호 사용이 필수적이다.

BCH (Bose-Chaudhuri-Hochquenghem) 코드는 하나 혹은 그 이상의 랜덤 에러 수정에 널리 사용되는 오류 정정 부호이다. 특히, BCH 코드는 다양한 통신 및 저장 장치의 오류 정정에 주로 사용되며, 전송 코드에 패리티 비트를 삽입하여, 전송시 랜덤 에러가 발생하였을 때 메시지 비트와 패리티 비트를 이용하여 오류 검출 및 수정을 수행한다[1]. BCH는 크게 인코더와 디코더로 나뉘며, 인코더는 LFSR (linear feedback shift register)을 이용하여 메시지비트에 따라 패리티비트를 생성하고, 이를 메시지비트에 추가하여 코드워드를 구성한 후 이를 전송한다. BCH 디코더는 수신된 코드워드로부터 오류 발생을 검사하고, 만일 오류 검출시 이를 정정한다.

최근 저장 장치중 가장 일반적으로 사용되는 것은 NAND 플래쉬 메모리로, 이는 크게 SLC (single- level cell)과 MLC (multi-level cell) 타입의 2가지 방식이 있다[2]. SLC 타입의 경우, 직렬 방식의 BCH 코드를 이용하여 오류 탐색 및 오류 정정이 가능하다. 그러나, 최근 많이 이용되는 MCL 플래쉬 메모리는 하나의 메모리 셀에 2비트 이상이 메모리에 저장되기 때문에 문턱 전압의 마진이 줄어 데이터의 오류 발생 확률이 높아진다. SLC 타입의 경우 오류 정정을 위해 일반적으로 해밍 코드를 이용하였으나, MLC 타입 플래쉬 메모리는 BCH 코드와 같이 좀 더 복잡하고 강력한 오류 정정 부호를 필요로 한다. 특히, 바이트 단위로 오류 정정이 가능한 고속 병렬 BCH 디코더가 필요하다[3].

BCH 디코더는 크게 신드롬 생성부, Key equation solver, Chien search 그리고 오류정정부로 구성된다. 특히, 신드롬 생성부와 Chien search의 경우 반복적인 연산이 요구되기 때문에 병렬 처리가 필수적이다. Y. Chen은 long BCH 코드를 위한 병렬 Chien search 구조를 제안하였다. 특히, 중복되는 연산을 공유하는 그룹 매칭 기법을 이용하여 회로 크기를 줄이는 방법을 제안하였

강원대학교 전자정보통신공학부 (Div. of of Electronics, Information and Communication Engineering, Kangwon National University)
Samcheock Campus, Kangwon National University, 1 Jooangang-ro, Samcheock,Gangwon, 245-711, Korea
+Corresponding author: jehoon.lee@kangwon.ac.kr
(Received:Mar.24,2015, Accepted:May.28, 2015)

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License(http://creativecommons.org/licenses/bync/3.0) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

다[4]. Y. Lee는 병렬 신드롬 생성기 구조를 제안하였고, 특히 하드웨어 복잡도를 줄이기 위한 새로운 최적화 기법을 제안하였다[5].

본 논문에서는 Shortened BCH (4122, 4096, 2) 코드를 위한 병렬 디코더 구조를 제안하였다. 코드 길이는 4,122비트이고 메시지비트 길이는 4,096비트, 패리티비트 길이는 26비트로 구성되어 있고 2개까지 오류수정이 가능하다. 특히, 일반적인 BCH 디코더는 신드롬 생성 및 Chien search의 경우 반복적인 연산을 필요로 하기 때문에 가장 긴 동작 시간이 요구된다. 따라서, 본 논문에서는 신드롬 생성기와 Chien search를 바이트 단위로 입력을 허용하고 이를 한 클럭에 연산하는 Byte-wise 병렬구조를 제안하여 전체 BCH 디코더의 처리 속도를 향상시키는 방안을 제안하였다.

2. 기존의 직렬 BCH 디코더

2.1 BCH 디코더

Fig. 1은 오류 정정 코드가 내장된 일반적인 NAND 플래쉬 메모리 구조를 나타낸다. 오류 정정을 위한 ECC (error correction code) 회로가 호스트 시스템과 NAND 플래쉬 사이에 위치하게 되며, MLC 타입의 경우 오류 정정 코도로 주로 BCH 코드를 이용한다.

Shortened BCH는 기호로 BCH (n,k,t)로 표기되며 코드 길이, n은 N-[k-m*t] (N=Zero sequence), 메시지 길이, k는 n-(m*t), 오류 정정 가능 비트는 t로 나타낸다. 일반적으로, BCH 복호기는 Fig. 2와 같이 구성된다. 송신단으로부터 수신된 데이터는 신드롬 생성기, Key equation solver, 그리고 Chien search를 통해 오류 여부를 검사하고, 만일 오류가 발생되었다면 오류 위치를 계산하게 된다. 그리고 마지막으로 해당 위치의 오류를 정정한다. Fig. 2에 나타낸 것처럼 신드롬 생성기는 1비트씩 입력받아 직렬로 연산할 경우 수신된 코드워드의 길이만큼 반복 연산을 수행해야 하며, Chien Search의 경우에도 반복 연산이 요구된다. 따라서, BCH 디코더의 성능 향상을 위해서는 신드롬 생성기와 Chien search 의 병렬 처리 지원이 필수적이다.

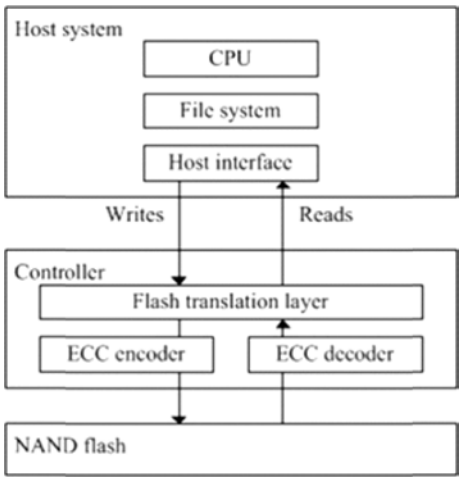

Fig. 1. Block diagram of NAND flash memory.

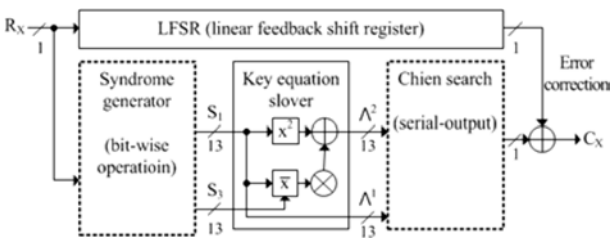

Fig. 2. Architecture of BCH decoder.

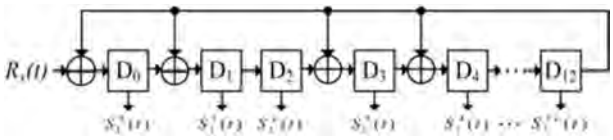

Fig. 3. LFSR based bit-serial syndrome generator, S_1.

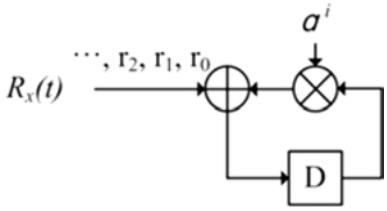

Fig. 4. GFM based bit-serial syndrome generator.

2.2 직렬 신드롬 생성기

신드롬 생성기는 일반적으로 Fig. 3 및 Fig. 4에 나타낸 것처럼 LFSR 기반 구조 혹은 GFM (Galois field multiplier) 구조로 설계된다 [6]. LFSR 구조는 연산 속도가 빠르고 회로 크기가 작다는 장점이 있고, GFM 구조는 병렬화하기 쉬운 구조라는 장점이 있다.

2.3 Key equation solver

Key equation solver는 신드롬 생성부의 두 출력값, S_1과 S_3을 이용하여 오류 위치 방정식, Λ_j을 생성한다. 오류 위치 방정식은 Λ_j (j=0, 1, 2, 3, ... ,t)로 표현되며 식 (1)에 나타낸 것처럼, 두 개의 신드롬, S_1과 S_3을 연산하여 얻어진다 [6]. 그리고 이 연산에 따라 Fig. 5에 나타낸 것처럼 회로를 구성한다. Key equation solver의 출력인 오류 위치 방정식, Λ_j는 Chien search로 전송되며, 이를 통해 오류 위치를 검출 및 오류정정을 수행한다 [7].

$$\Lambda(x) = ([S_1 \times S_2] + S_3)x^2 + S_2 x + S_1 \quad (1)$$

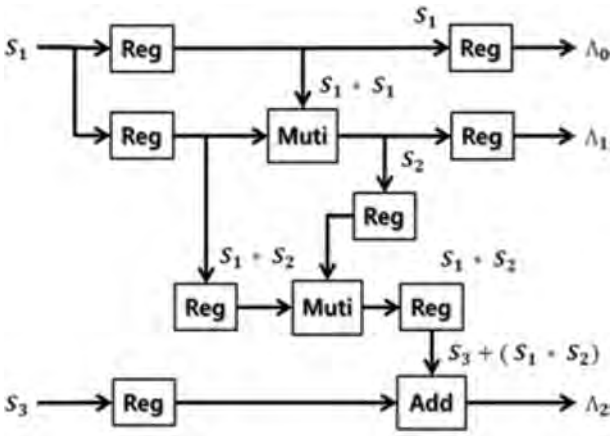

Fig. 5. Block diagram of key equation solver algorithm.

2.4 Serial Chien search

Chien search 연산 과정은 Key equation solver에서 추출된 Λ_j (j=0, 1, 2, 3, .. ,t)와 식 (1)을 이용하여 오류 위치 검출 및 정정을 수행한다. 연산 과정은 GF(2^m)의 유한체인 α^i(i=0, 1, .. ,n-1)을 Key equation solver에서 수신한 오류 위치 방정식인 식 (2)와 식 (3)에 대입하여 방정식의 근을 찾는 과정을 통하여 오류 위치 감지 및 오류를 정정 할 수 있다[8,9].

$$\Lambda(\alpha^i) = \sum_j^t \alpha^{ij} + \Lambda_0 \tag{2}$$

$$\Lambda(\alpha^i) = \Lambda_0 + \Lambda_1\alpha^i + \Lambda_2\alpha^{2i} \tag{3}$$

식 (2)은 일반적인 오류위치 방정식이며 BCH (4122, 4096, 2) 코드의 오류 위치 방정식은 식 (3)과 같이 표현된다. GF(2^{13})의 원소 α^i ($0 \le i \le 4{,}121$)을 식 (3)에 대입하여 근을 통한 오류 위치를 찾으면 Chien search 출력부에서 '1'이 출력이 되고 출력값과 오류 비트의 XOR 연산을 통하여 오류가 수정된다.

Chien search 연산기 구현을 위해서는 곱셈 연산을 위한 FFM (finite field multiplier)과 덧셈 연산을 위한 FFA (finite field adder)가 필요하다. Chien search 연산에는 n 클럭의 연산 시간이 필요하기 때문에 신드롬 생성기와 함께 BCH 디코더에서 연산 시간이 가장 긴 블록이다. 그러므로 신드롬 생성기와 Chien search를 병렬 연산을 통하여 BCH 디코더의 연산 속도를 향상 시킬 수 있다.

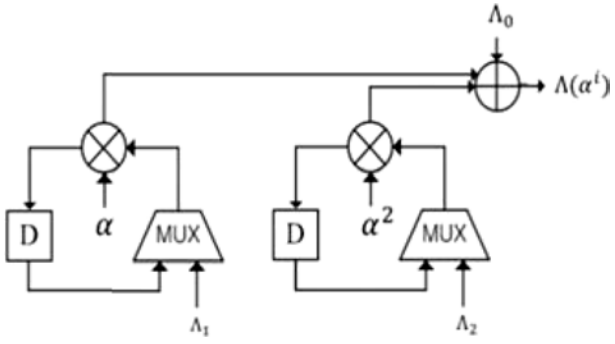

Fig. 6. Typical GFM-based serial Chien search.

본 논문에서는 그룹 매칭 (group matching) 방식을 사용하여 바이트 단위로 병렬화 구현뿐만 아니라, 더 적은 크기로 회로를 구현하여 Chien search을 최적화 하였으며 결과적으로 제안한 방식은 Fig. 6에 나타낸 기존 GFM 방식의 Chien search와 비교하여 속도 및 회로의 크기가 최적화되었다.

3. 제안된 바이트 단위의 병렬 BCH 디코더

3.1 병렬 신드롬 생성기

Fig. 7에 나타낸 GFM 기반의 병렬 신드롬 생성기의 경우 병렬 계수가 높을수록. 임계 경로 지연이 급격하게 증가하여, 회로의 연산 속도가 느려진다. 본 논문에서는 이를 해결하기 위하여 LFSR 구조를 사용한 병렬 연산 방법을 제안한다. 제안된 구조에서는, 병렬계수가 높아져도 임계 경로 지연이 크

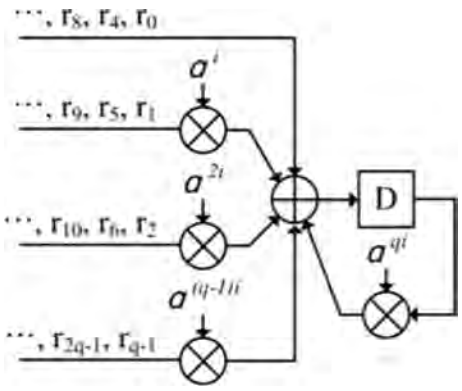

Fig. 7. GFM based syndrome generator.

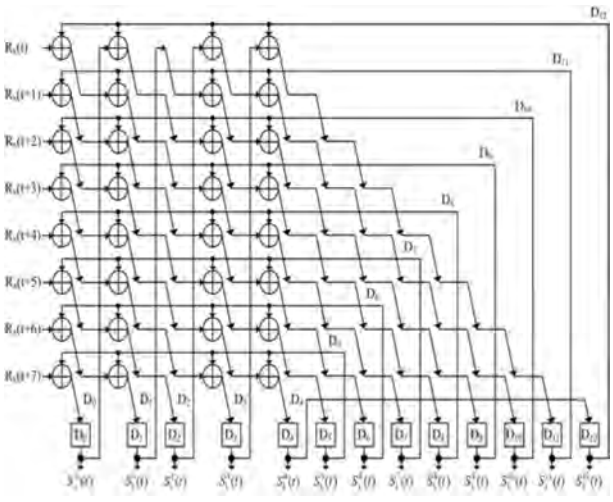

Fig. 8. The proposed 8-parallel LFSR-based syndrome generator for S_1.

게 증가하지 않고 고속으로 신드롬 값을 추출할 수 있다. LFSR 방식의 신드롬 S_i의 연산식은 식(4)로 계산된다. 임의 시간 t 에서, BCH 인코더에서 수신된 부호어 $R_x(t)$값과 LFSR의 각 레지스터에 저장된13비트의 $D_0(t)$ 부터 $D_{12}(t)$의 부호어를 n 시간 만큼의 XOR 연산을 통하여 신드롬 값을 계산한다. 4,121 클럭의 연산이 완료되면 레지스터의 13 비트 출력을 통해 신드롬 생성을 완료한다.

식(5)는 연속된 8개의 부호어 비트 입력 대신 한 번에 바이트 단위 입력을 허용하고, 이들의 연산을 병렬화한 S_i 신드롬 생성기의 연산 수식이다. 병렬계수가 증가함에 따라서 XOR의 수가 증가한다. 그러나, 기존 직렬 방식의 신드롬 생성기 보다 n/p 만큼의 (p=병렬계수) 연산 시간이 감소하며 또한 GFM 병렬 방식보다 임계 경로 지연이 적어 처리 속도가 향상된다.

Fig. 8은 바이트 단위의 병렬 S_i 신드롬 생성기 회로이다. ($0 \le t \le (4121/8)$) 클럭 동안 연산을 수행하며 516 클럭의 연산이 완료되면 레지스터, $D_0(t+8)$에서 $D_{12}(t+8)$까지 저장된 13비트 신드롬 값을 추출하여 신드롬 생성을 완료한다.

$$
\begin{aligned}
D_0(t+1) &= R_x(t) + D_{10}(t)\\
D_1(t+1) &= D_{10}(t) + D_{11}(t)\\
D_2(t+1) &= D_{11}(t) + D_{12}(t)\\
D_3(t+1) &= D_0(t) + D_{10}(t) + D_{12}(t)\\
D_4(t+1) &= D_1(t) + D_{10}(t) + D_{11}(t)\\
D_5(t+1) &= D_2(t) + D_{11}(t) + D_{12}(t)\\
D_6(t+1) &= D_3(t) + D_{12}(t)\\
D_7(t+1) &= D_4(t)\\
D_8(t+1) &= D_5(t)\\
D_9(t+1) &= D_6(t)\\
D_{10}(t+1) &= D_7(t)\\
D_{11}(t+1) &= D_8(t)\\
D_{12}(t+1) &= D_9(t)
\end{aligned}
\tag{4}
$$

$$
\begin{aligned}
D_0(t+8) &= R_x(t+7) + D_5(t)\\
D_1(t+8) &= R_x(t+6) + D_5(t) + D_6(t)\\
D_2(t+8) &= R_x(t+5) + D_6(t) + D_7(t)\\
D_3(t+8) &= R_x(t+4) + D_5(t) + D_7(t) + D_8(t)\\
D_4(t+8) &= R_x(t+3) + D_5(t) + D_7(t) + D_8(t) + D_9(t)\\
D_5(t+8) &= R_x(t+2) + D_6(t) + D_7(t) + D_8(t) + D_{10}(t)\\
D_6(t+8) &= R_x(t+1) + D_7(t) + D_8(t) + D_{10}(t) + D_{11}(t)\\
D_7(t+8) &= R_x(t+1) + D_8(t) + D_9(t) + D_{11}(t) + D_{12}(t)\\
D_8(t+8) &= D_0(t) + D_9(t) + D_{10}(t) + D_{12}(t)\\
D_9(t+8) &= D_1(t) + D_{10}(t) + D_{11}(t)\\
D_{10}(t+8) &= D_2(t) + D_{11}(t) + D_{12}(t)\\
D_{11}(t+8) &= D_3(t) + D_{12}(t)\\
D_{12}(t+8) &= D_4(t)
\end{aligned}
\tag{5}
$$

3.2 병렬 Chien search

일반적인 병렬 GFM 기반의 Chien Search 회로는 병렬 계수가 높아질수록 회로의 크기와 임계 경로 지연시간이 급격하게 증가한다. 본 논문에서는 이를 해결하기 위하여 중복 연산을 제거하여 회로의 크기와 임계 경로 지연시간을 감소시키는 그룹 매칭 기법을 사용하였다. 이를 통해, 회로 크기가 작고 동작 속도가 빠른 바이트 단위 병렬 Chien search 회로를 설계하였다.

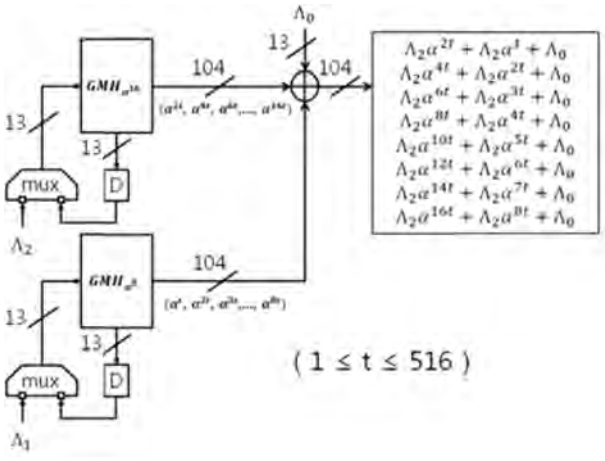

Fig. 9. The architecture of the proposed 8-parallel Chien search block.

GFM 기반의 p-병렬 Chien search 는 key equation solver 에서 생성된 오류 위치 방정식에 α. …. α^p 에서 α^2. …. α^{2p} 까지 순차적으로 곱셈연산을 수행한다. 이 연산 과정 중 중복되는 연산이 존재하며, 식(6)에 나타낸 것처럼, 중복되는 연산의 예로 α^2 과 α^4 곱셈기 간의 중복 연산이 존재한다. 회로 크기를 줄이기 위해서는 중복 연산을 갖는 곱셈기의 연산을 여러 입력간에 상호 공유함으로써 연산기의 회로 크기를 줄일 수 있다. 이를 일반적으로 그룹 매칭 기법이라 하며, 본 논문에서는 GMH_i로 표기한다. 그룹 매칭 기법을 α^4 곱셈기에 적용하면 식(7)과 같이 표현되어 식이 간략화 된다. 만일 중복 입력을 제거하지 않는다면 식(8)과 같이 표현될 수 있으며, 연산의 필요한 XOR 개수가 크게 늘어남을 확인할 수 있다.

그룹 매칭 기법을 적용한 바이트 단위 병렬 Chien search 블록은 Fig. 9에 나타내었다. 그룹 매칭 기법을 이용하여 중복 입력을 갖는 곱셈기들을 제거할 경우병렬 계수를 증가 할수록 중복되는 연산이 많아지기 때문에 기존의 병렬 방식 보다 Chien search 회로 크기를 큰 폭으로 줄일 수 있다.

$$
\begin{aligned}
GMH_0 &= D_{11}(t) + D_{12}(t)\\
GMH_1 &= D_0(t) + D_{12}(t)\\
GMH_2 &= D_1(t) + D_{11}(t)\\
GMH_3 &= D_2(t) + D_{11}(t) + D_{12}(t)\\
GMH_4 &= D_3(t) + D_{12}(t)
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
D_0(t+2) &= D_9(t)\\
D_1(t+2) &= D_9(t) + D_{10}(t)\\
D_2(t+2) &= D_{11}(t) + D_{10}(t)\\
D_3(t+2) &= GMH_0 + D_9(t)\\
D_4(t+2) &= GMH_1 + D_9(t) + D_{10}(t)\\
D_5(t+2) &= GMH_2 + D_{10}(t)\\
D_6(t+2) &= GMH_3\\
D_7(t+2) &= GMH_4\\
D_8(t+2) &= D_4(t)\\
D_9(t+2) &= D_5(t)\\
D_{10}(t+2) &= D_6(t)\\
D_{11}(t+2) &= D_7(t)\\
D_{12}(t+2) &= D_8(t)
\end{aligned}
\tag{7}
$$

$$
\begin{aligned}
D_0(t+2) &= D_9(t) \\
D_1(t+2) &= D_9(t) + D_{10}(t) \\
D_2(t+2) &= D_{11}(t) + D_{10}(t) \\
D_3(t+2) &= D_{11}(t) + D_{12}(t) + D_9(t) \\
D_4(t+2) &= D_0(t) + D_{12}(t) + D_9(t) + D_{10}(t) \\
D_5(t+2) &= D_1(t) + D_{11}(t) + D_{10}(t) \\
D_6(t+2) &= D_2(t) + D_{11}(t) + D_{12}(t) \\
D_7(t+2) &= D_3(t) + D_{12}(t) \\
D_8(t+2) &= D_4(t) \\
D_9(t+2) &= D_5(t) \\
D_{10}(t+2) &= D_6(t) \\
D_{11}(t+2) &= D_7(t) \\
D_{12}(t+2) &= D_8(t)
\end{aligned}
\tag{8}
$$

4. 시뮬레이션 결과 및 분석

제안한 BCH 디코더는 Xilinx 설계 환경에서 VHDL 코딩을 통하여 검증하였다. BCH 디코더의 신드롬 생성기 및 Chien search는 반복적인 연산을 수행하기 때문에 직렬 방식으로 연산할 경우 긴 연산시간이 소요된다. 하지만, 병렬화 설계를 하면 병렬계수 p에 비례하여 연산속도가 증가시킬 수 있다. 하지만 병렬화를 할 경우 병렬계수 p에 비례하여 회로 크기는 증가하며 임계 경로 지연시간의 증가로 회로의 연산 속도도 느려지는 단점이 있다. 본 논문에서는 LFSR 구조 및 그룹 매칭 기법의 적용을 통하여 병렬 BCH 디코더의 속도를 향상 시켰다.

Fig. 10은 일반적인 직렬 방식으로 구현된 신드롬 생성기와, 병렬 계수가 2, 4, 그리고 8인 경우의 신드롬 생성기의 동작 실험을 나타낸다. Fig. 10에 나타낸 것처럼, 병렬 계수 p가 증가할수록 기존 직렬 방식에 비해 연산에 필요한 클럭의 수가 1/p배로 감소함을 확인할 수 있다. 따라서, 기존 직렬 방식으로 구동되는 신드롬 생성기의 소요 클럭 수에 비해, 바이트 단위로 병렬화된 신드롬 생성기는 1/8배의 클럭만을 필요로 한다.

신드롬 생성기뿐만 아니라 Chien search 블록의 경우에도 병렬 계수를 증가시킬수록 필요로 하는 클럭의 수가 1/p배로 요구되는 클럭 사이클의 수가 감소된다.

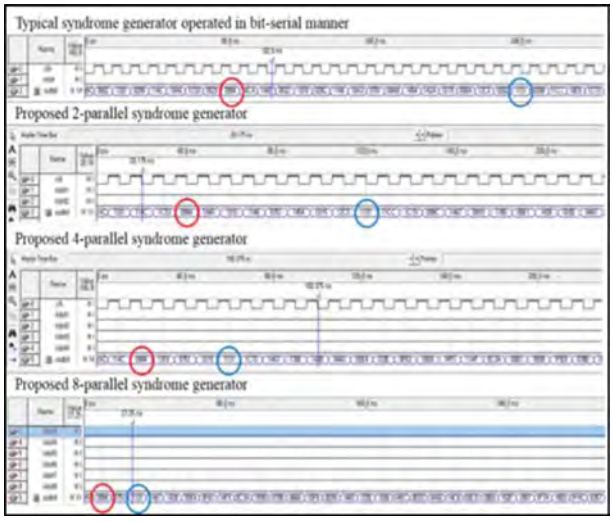

Fig. 10. VHDL simulation waveform results.

Table 1은 병렬 계수, p의 증가에 따라 변화하는 동작 속도 및 회로 크기의 비교 결과를 나타낸다. 첫번째로 제안된 신드롬 생성기의 경우 직렬 방식으로 동작할 때 최대 클럭 주파수는 1.24 GHz가 된다. 병렬 계수를 증가시킬수록 임계 경로의 길이가 길어지며, 병렬 계수가 8인 신드롬 생성기의 경우 클럭 주파수의는 575 MHz가 된다. 따라서, 직렬 방식에 비해 연산에 필요한 클럭 수는 1/8배로 감소되나, 실제 동작 속도는 3.71배 향상된다. 병렬 계수가 8인 LFSR 구조의 신드롬 생성기와 동일한 병렬 계수를 갖는 GFM 구조의 신드롬 생성기의 동작 속도를 비교하면, 약 30%의 성능 향상을 얻을 수 있다.

Table 2는 Chien search 블록의 회로 크기 비교를 나타낸다. 제안된 Chien search 회로의 경우 직렬로 구동하는 경우 XOR 게이트의 수는 9개로 구성할 수 있다. 그러나, 병렬 계수를 2, 4, 그리고 8로 증가시킬수록, 회로에 포함된 XOR 게이트의 수는 18, 36, 그리고 95로 증가한다. 일반적인 GFM 기반의 Chien search 블럭과 비교할 경우 제안된 Chien search 블록의 경우 하드웨어 복잡도를 최대 67%까지 감소시킬 수 있음을 확인하였다.

Table 1. Comparison results of the operation speed between the proposed LFSR-based syndrome generators and the conventional GFM-based one

Approach		Maximum Clock Frequency	Speed Saving
Proposed Syndrome generator	P=1	1.24Ghz	-
	P=2	893Mhz	-
	P=4	699Mhz	43%
	P=8	575Mhz	30%
GFM Syndrome generator	P=1	-	-
	P=2	-	-
	P=4	400Mhz	-
	P=8	400Mhz	-

Table 2. Comparison results of hardware complexity speed between the proposed LFSR-based syndrome generators and the conventional GFM-based one

Approach		Number of XORs	Size Saving
Proposed Chien search	P=1	9	0%
	P=2	18	33%
	P=4	36	60%
	P=8	95	67%
GFM Chien search	P=1	9	-
	P=2	27	-
	P=4	90	-
	P=8	290	-

5. 결 론

본 논문은 BCH 디코더의 동작 속도 향상을 위하여 바이트 단위의 병렬 구조 설계 방법을 제안하였다. BCH 디코더의 동작 속도에 가장 큰 영향을 미치는 블록은 반복적인 연산이 필요한 신드롬 생성기와 Chien search이며, 본 논문은 두 블록에 바이트 단위로 연산할 수 있는 병렬 구조를 적용시켜 기존 직렬 방식에 비해 동작 속도를 크게 향상시켰다. 또한, 병렬 계수를 증가함에 따라 회로 크기 증가는 필연적이나 본 논문은 GFM 구조가 아닌 LFSR 구조의 병렬화 회로를 구성함으로써, 회로 크기 증가율을 크게 감소시켰다. GFM 구조의 병렬 구조와 비교하여 클럭 사이클의 수는 동일하게 감소시켰으나, 오히려 블록 내의 임계 경로의 크기를 줄임으로써 고속의 클럭을 사용하여, 동작 속도를 최대 43% 향상시켰다. 또한, Chien search 블록의 경우 병렬화를 수행한 후, 중복된 입력을 갖는 곱셈기를 공유하는 그룹 매칭 기법을 적용하여 이를 적용하지 않은 회로에 비해 회로 크기를 최대 67% 감소시켰다. 또한, 병렬 계수로 8을 선택하여 바이트 단위의 데이터 전송 구조를 갖는 메모리에 효율적으로 적용할 수 있다. 따라서, 제안한 구조는 고속으로 오류를 정정하고 또한 회로의 크기를 최적화시켜 경량 회로 구현이 가능하며, 사물인터넷과 같은 무선 센서 네트워크에 효율적으로 사용 할 수 있다.

Acknowledgement

본 연구는 교육부와 한국연구재단의 지역혁신창의 인력양성 사업으로 수행된 연구결과임 (No. NRF-2012H1B8A2026055).

REFERENCES

[1] S. Lin and D. J. Costello, *Error Control Coding*, Upper Saddle River, NJ: Prentice Hall, 2004.
[2] R. Martin, et al., “Fault detection and diagnosis for multi-level cell flash memories,” *Proc. of IMT 2006*, pp.1896-1901, Apr. 2006
[3] S. C. Jang, et. Al., “Design of a parallel BCH decoder for MLC memory,” *Proc. ISOCC ’08,* Vol. 3, pp. 46-47, Nov. 2008.
[4] Chen, Yanni and Parhi, K.K. “Small Area Parallel Chien Search Architectures for Long BCH Codes,” *IEEE Trans. Inform. Theory*, Vol. 12, No. 5, pp. 545-549, May. 2004.
[5] Y. J. Lee and H. Y. Yoo and I. C. Park, “Small-area parallel syndrome calculation for strong BCH decoding,” *Proc. of ICASSP 2012*, pp. 1609-1612, March. 2012.
[6] Massey, J, L., “Shift-register synthesis and BCH decoding,” *IEEE Trans. Inform. Theory*, Vol. 15, No. 1, pp. 122-127, Jan. 1969.
[7] J. H. Jeng and T. Y. Truong, “On decoding of both errors and erasures of a Reed-Solomon code using an inverse-free Berlekamp-Massey algorithm,” *IEEE Trans. Commun.,* Vol. 47, No. 10, pp. 1488-1494*, Oct. 1999.*
[8] R. T. Chien, “Cyclic decoding procedure for the Bose-Chaudguri-Hocquenhem Code,” *IEEE Trans. Inform. Theory*, Vol. 10, No. 10, pp.357-363, Oct. 1964.
[9] S. V. Fedorenko and P. V. Trifonov, “Finding roots of polynomials over finite fields,” *IEEE Trans. Commun.*, Vol. 50, No. 11, Nov. 2002.