논리 최적화 기법을 이용한 병렬 CRC 회로 설계

*이현빈 *김주섭° *박성주 **박창원
*한양대학교 컴퓨터공학과, **전자부품연구원
{bean, kjs9191, parksj}@mslab.hanyang.ac.kr, parkcw@keti.re.kr

A Design of High Performance Parallel CRC Using A Simple Logic Optimization

*Hyunbean Yi, *Jusub Kim°, *Sungju Park, **Changwon Park
*Dept. of Computer Science and Engineering Hanyang University
**Intelligent IT System Research Center, Korea Electronics Technology Institute

요 약

본 논문은 통신 시스템에서 오류 검출을 위해 널리 사용되고 있는 Cyclic Redundancy Check (CRC)회로의 병렬 구현을 위한 최적화 알고리즘을 제시한다. 논리 단을 최소로 하면서 가능한 많은 공유 텀을 찾아 매핑 함으로써 속도 및 게이트 수를 줄인다. 본 논문에서는 이더넷의 32비트 CRC를 병렬로 구현하여 성능평가를 하였다. FPGA 및 표준 셀 라이브러리를 이용하여 합성하였으며, 기존의 방식에 비해 속도와 면적 모두 향상되었음을 보여준다.

1. 서 론

Cyclic Redundancy Check(CRC)는 통신 시스템에서 가장 널리 사용되는 데이터 오류 검출 방법이다. 여러 시스템에서, 홀수 비트 오류 검출 방법인 패리티 (Parity)나 한 바이트에 3~4개의 비트를 사용하여 오류를 검출하고 복구하는 해밍코드 (Hamming Code)가 많이 사용되어왔다. 그러나, 고속의 중장거리 데이터 송수신시에 발생하는 대부분의 에러는 연속적인 여러 비트에 걸쳐 발생하는 특성이 있으며, 오류 검출을 위한 오버헤드가 너무 커진다. 따라서, 데이터의 크기에 상관 없이 코드 생성 다항식(폴리노미얼: Polynomial)의 차수만큼의 비트열로 여러 비트의 오류를 검출할 수 있는 CRC는 고속 통신 시스템에 매우 적합하다. 더욱이 통신 시스템 기술의 발달로 네트워크 및 데이터 I/O 송수신 표준들은 1 Gb/s를 넘어 10 Gb/s 이상의 속도를 목표로 하고 있으며, 그에 따라 고속으로 동작할 수 있는 CRC 회로 설계 기술이 필요하다.

CRC는 기본적으로, Linear Feedback Shift Register(LFSR)를 이용한 직렬 회로로 구성되어, CRC 코드를 생성하기 위해 직렬 데이터 입출력 속도에 맞는 비트 클럭을 사용해야 하거나, CRC 코드 생성 및 오류 검출에 걸리는 지연시간을 위한 버퍼링이 필요하다. 따라서, 고속으로 갈수록 구현이 어려워지므로, 병렬 CRC 구현에 관한 많은 연구가 이루어지고 있다. 전통적으로 병렬 CRC 회로 구현을 위해, 표 검색(table look-up) 방식[1,2]과 Exclusive-OR (XOR) 조합회로[3,4,5,6,7,8]로 구성하는 두 가지 방식이 채택되고 있다.

본 논문에서는 XOR 조합으로 이루어진 병렬 CRC 회로에서 최소의 논리 단을 유지하면서, 가능한 많은 XOR 텀을 공유하도록 매핑하여 게이트 수를 줄이는 휴리스틱 알고리즘을 제시한다. 2장에서 기본적인 병렬 CRC 회로 구현 방법을 간단하게 설명하고, 3장에서 본 논문에서 제시하는 알고리즘을 설명 및 분석한다. 4 장에서 기존의 방식에 의한 결과와 성능을 비교하고, 5장에서 결론을 맺는다.

2. 병렬 CRC 회로 구현

여러 통신 시스템에서는 몇 가지 폴리노미얼을 주로 사용하며, 이진 모듈로-2 연산을 수행함으로써 나온 결과를 CRC 코드로 사용한다. 이와 같은 연산을 직렬 데이터 송수신 시스템에서는 LFSR과 XOR 게이트를 이용하여 하드웨어로 구현할 수 있다. 병렬 CRC 코드 생성의 기본적인 아이디어는, LFSR에서 여러 번의 쉬프트와 XOR 연산 후에 각 플립플롭에 저장되는 결과를 병렬 데이터 입력의 XOR 조합회로로 한 클럭 사이클에 생성될 수 있도록 하는 것이다[1].

병렬 데이터의 크기와 CRC 레지스터의 길이가 모두 양의 정수 n인, n-비트 CRC 생성기 구현을 위한 알고리즘을 설명하기 위하여, i를 1이상 n이하인 양의 정수로 놓고, F_i 는 LFSR의 i 번째 레지스터, C_i 는 쉬프트를 시작하기 이전에 F_i 에 저장되어 있는 초기값, D_i는 i번째 입력데이터라고 각각 정의한다. 데이터는 최하위 비트인 D_1부터 입력되고, 쉬프트는 번호가 낮은 레지스터 방향으로 진행된다고 하자. 그러면, 쉬프트를 수행하기 전의 레지스터의 초기값은 C_i 가 되고, 매 클럭마다 한 비트씩 쉬프트를 하여, F_i 에는 쉬프트와 XOR 게이트에 의해 C 와 D 의 조합으로 이루어진 새로운 값이 저장된다. 이와 같이 하면, n번 쉬프트 후에 레지스터에 저장되는 값이 n 비트 데이터에 대한 CRC 코드가 된다. 게다가, X_i를 (D_i xor C_i)라고 하면, n 번 쉬프트 후, 레지스터의 결과값이 모두 X_i만의 조합으로 이루어진다는 특징이 있다. 따라서 X_i하나가 2입력 XOR 게이트가 되고 X_i의 조합으로부터 한 클럭에 CRC 코드를 생성하는 병렬 CRC 회로를 구성할 수 있다. n번의 쉬프트 후 각 F에 저장되는 결과를, 계산된 숫자가 아닌 X_i로 저장하여 나타냄으로써 원하는 X_i의 조합을 구할 수 있다. 그림 1이 n번의 쉬프트 후, 각 레지스터에 저장되는 X_i의 조합을 이끌어 내기 위한 알고리즘의 유사코드이다. 루프를 실행할 때마다, 각 F에 저장되는 결과를 X_i의 조합을 구할 수 있다.

```
CRC Gen:
    for (i = 1; i = n; i++) {
        X[i] = D[i] ^ C[i]
        for (j = 1; j = n; j++) {
            If (P[j] == 1)
                F[j] = X[i] ^ F[j+1];
            else
                F[j] = F[j+1];
        }
    }
```

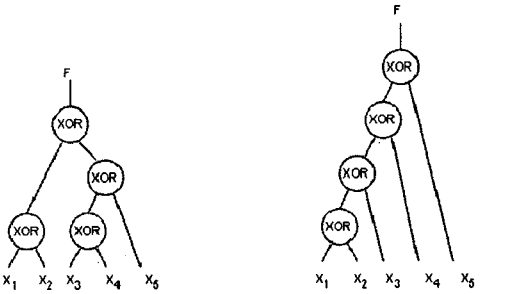
(P : n bit parameter such that if XOR tap is needed, P[j]='1', otherwise, P[j]='0'. It depends on the polynomial used.)

그림 1. CRC 코드 생성 알고리즘

3. 휴리스틱 CRC XOR 로직 최적화 알고리즘

그림 1의 CRC 코드 생성 알고리즘을 통해 얻어진 결과를 바로 회로로 구현할 수 있다. 그러나 체계적인 매핑과정을 거

치지 않으면 좋은 성능을 기대하기 어렵다. 그림 2를 보자.



(a) F = (((X1^X2)^X3)^X4)^X5 (b) F = (X1^X2)^((X3^X4)^X5)

그림 2. 로직 매핑 예

이와 같은 이진 트리구조에서 최소의 논리 단은, 입력의 개수를 n이라고 하면 "$\log_2 n$의 올림" 으로 결정된다. 그림 2의 (a)와 (b)는, 게이트 수가 같고, 함수적으로도 같은기능을 수행하지만, (a)는 3 논리 단으로, (b)는 4 논리 단으로 매핑이 되어 속도차이가 발생한다. 이러한 문제를 해결하기 위한 최적화가 필요하다. 그러나, 최적화 알고리즘 또한 NP-Hard 문제로써 회로의 특성에 따라 경험적인 방법(Heuristic)을 적용할 필요가 있다[8,9]. 일반적으로, 최적화에 있어서, 면적과 속도는 서로 반비례 관계에 있으므로, 최적화의 목적에 맞는 알고리즘을 구상해야 한다[9]. 본 논문에서는, 속도 최적화에 초점을 두고, 논리 단을 최소로 유지하면서 가능한 한 많은 XOR 텀을 공유하도록 매핑하여 면적을 줄이는 것을 목표로 한다.

이를 위해 본 논문에서는 이진 트리구조의 논리 단을 행렬로 모델링 하였고, 효과적인 경험적 방법(Heuristic) - 공통 XOR 텀을 최대한 빠른 단계내에 체계적으로 검색하고, 적절한 논리 단으로 매핑하는 방법 - 을 연구하였다.

이 알고리즘을 설명하기위해 그림 3의 테이블 T를 정의한다.

$$T = \begin{array}{|c|c|c|c|c|c|} \hline & X_1 & X_2 & X_3 & \cdots & X_n \\ \hline F_1 & e_{11} & e_{12} & e_{13} & \cdots & e_{1n} \\ \hline F_2 & e_{21} & e_{22} & e_{23} & \cdots & e_{2n} \\ \hline F_3 & e_{31} & e_{32} & e_{33} & \cdots & e_{3n} \\ \hline \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hline F_n & e_{n1} & e_{n2} & e_{n3} & \cdots & e_{nn} \\ \hline \end{array} \qquad e_{ij} \begin{cases} \text{'1', if } F_i \text{ has } X_j \\ \text{'0', otherwise.} \end{cases}$$

그림 3. F 대 X 테이블

여기에서 F_i가 XOR게이트가 포함되어 있으면 '1', 그렇지 않으면 '0'으로 표시한다. 표현 방법은 그림 4와 같다.

F_1	X_1 X_3 X_4 X_5
F_2	X_2 X_3 X_5
F_3	X_1 X_2 X_5
F_4	X_1 X_2 X_4 X_5
F_5	X_2 X_3

⇒

	X_1	X_2	X_3	X_4	X_5
F_1	1	0	1	1	1
F_2	0	1	1	0	1
F_3	1	1	0	0	1
F_4	1	1	0	1	1
F_5	0	1	1	0	0

(a) (b)

그림 4. 5 x 5 테이블 T 생성 예

이 테이블 T를 바탕으로 다음과 같이 알고리즘을 수행한다.

Step 1 : T의 열 중, '1'의 개수가 가장 많은 열을 찾고, 그 열을 "Column A"라고 하자.

Step 2 : T의 열 내에서 "Column A" 자신을 제외한 다른 모든 열과 비교하여, 가장 많은 행에 공통으로 '1'을 포함하고 있는 열을 찾고, 그 열을 "Column B"라고 하자. 만약 해당하는 "Column B"가 없다면 다음 '1'의 개수가 가장 많은 열을 찾고, Step 1의 과정을 다시 밟는다.

Step 3 : Column A, B에서, 같은 행에 공통으로 포함되어 있는 '1'을 두 열에서 모두 삭제하되, 남은 '1'이 하나도 없으면 그 열 전체를 삭제한다. [찾은 공유 텀을 삭제하여 알고리즘 수행 횟수와 생성되는 열의 수를 감소시킨다.]

Step 4 : Step 3에서 삭제된 '1'에 해당되는 행은 '1'로, 나머지는 '0'로 구성된 새로운 열을 "Column AB"라고 한다. [매핑되지 않은 X_s에 대해서 먼저 매핑될 기회를 제공함으로써 논리 단을 최소로 유지시킨다.]

Step 5 : T의 열이 모두 삭제되거나, 그 중 남아있는 열의 각 행에 '1'이 하나 이하가 될 때까지 Step 1~ Step 4를 반복하고 Step 6으로 넘어간다.

Step 6: Step 1 ~ Step 5의 결과 남은 T와 새로 생성된 열을 합친 새로운 테이블을 T로 놓는다. T의 각 행에 '1'이 하나 이하이면, 알고리즘을 종료하고, 아니면, Step 1 ~ Step 5를 수행한다.

알고리즘 수행 중, 아래와 같은 선택의 문제가 발생한다.

Case 1 : Step 1에서 찾은 열이 두 개 이상인 경우.

Case 2 : Step 2에서 Column B로 선택될 수 있는 열이 두개 이상인 경우.

이 목표에 부합하는 열 선택 기준을 정하기 위해, 다음과 같은 가설을 세우고, 그 타당성을 설명한다.

(가설 1) Case 1에 대해서, 가능한 행렬의 왼쪽에 있는 열을 선택함으로써 논리 단 증가를 막을 수 있다.

(증명) 하나의 F에 대하여, 새로 생성된 열의 선택 횟수가 증가하면 할수록, 그 F에 영향을 주는 X_s의 XOR 논리 단 또한 증가할 가능성이 높아진다.

(가설 2) Case 2에 대해서, '1'의 총 개수가 적은 열을 선택함으로써 새로 생성되는 열의 개수를 줄여 게이트 수 증가를 막을 수 있다.

(증명) 새로 생성되는 열의 개수가 적을수록 많은 XOR 텀을 공유하도록 매핑 시켰음을 의미한다. 따라서, 가능한 한 빨리 각 열의 '1'의 개수를 줄이고, 열을 삭제 함으로써 알고리즘 수행시간을 단축시켜 게이트 수를 줄일 수 있다.

따라서, Case 1, 2에 대해 다음의 열 선택 규칙을 정한다.

(열 선택 규칙 1 (for Case 1))

: j가 가장낮은 열을 선택한다.

(열 선택 규칙 2 (for Case 2))

: '1'이 가장적은 열 중, j가 가장 낮은 열을 선택한다.

제시한 알고리즘을 앞서 생성한 테이블 T에 열 선택 규칙을 적용하여 결과를 구하면 총 7번의 루프가 실행되며 그림 5와 같은 결과를 얻을 수 있다.

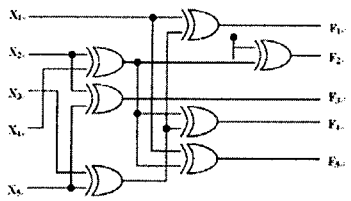
	X_2,X_5	$X_1,(X_2,X_4)$	$X_1,(X_5,X_3)$	$(X_5,X_3),(X_2,X_4)$	$(X_5,X_3),X_1,(X_2,X_4)$
F1	0	0	1	0	0
F2	0	0	0	0	1
F3	1	0	0	0	0
F4	0	0	0	1	0
F5	0	1	0	0	0

그림 5. 최종 결과 테이블

그림 5의 최종 결과 테이블을 부울함수 및 논리회로로 표현하면 그림 6과 같은 결과를 얻을 수 있다.

F1 = X1 ^ (X5 ^ X3)
F2 = (X5 ^ X3) ^ X1 ^ (X2 ^ X4)
F3 = X2 ^ X5
F4 = (X5 ^ X3) ^ (X2 ^ X4)
F5 = X1 ^ (X2 ^ X4)

(a) 최적화 결과 부울함수 식



(b) 최적화 결과 논리회로

그림 6. 최적화 알고리즘 결과 XOR 부울함수 및 논리회로

즉, 제시한 알고리즘을 통해 12개의 XOR 게이트, 4 논리 단으로 구현된 회로가 7개의 XOR 게이트, 3 논리 단으로 최적화되어 구현됨을 확인할 수 있다.

4. 구현 및 성능평가

성능평가를 위해 이더넷에서 사용하는 32 비트 CRC 회로를 설계하였다. 표 1이 이더넷 32비트 CRC 코드생성 다항식이다.

표 1. 32 비트 CRC 코드 생성 다항식

CRC-32 (Ethernet)	$P(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X^{1} + 1$

이 다항식으로부터 그림 1의 알고리즘을 사용하여 32비트 LFSR의 32번 쉬프트 후에 F에 누적되는 X_8의 표를 구성하였다. 이 표로부터 테이블 T를 구성하여 본 논문에서 제안한 알고리즘을 수행하였다. 이더넷 32비트 CRC 회로에 대해 알고리즘 수행 전 초기 테이블 T로부터 최적화를 고려하지 않고 합성될 때에 논리 단은 최소 6에서 최대 16이며, XOR 게이트 수는 총 452라는 계산이 나온다. 표 2는 기존 논문과 SIS [10], 그리고 본 논문에서 제시한 알고리즘에 의한 최적화 결과 비교표이다.

표 2. 최적화 결과 비교표

	논리 단	XOR 게이트 수
논문 [6]	7	408
논문 [8]	6	370
SIS [10]	8	275
Proposed	6	258

논문 [8]에서 최소의 논리 단이 나왔지만 단지 "manual trick"에 의한 결과이기 때문에 활용성은 없으며, 이 논문에서는 FPGA 구현 최적화를 위한 VHDL 코드에 초점을 두었다. SIS는 다중 레벨 합성툴로써, XOR 게이트 수에 있어서 기존 논문에 비해 훨씬 좋은 결과를 보이지만, 논리 단까지 동시에 최적화하지는 못했다. 하지만, 본 논문에서 제시한 알고리즘(Proposed)에 의한 결과는 최소한의 논리 단을 유지하면서도 XOR 게이트 수가 가장 적게 나왔다.

표 2의 결과를 바탕으로, Verilog HDL로 코딩을 하여 구현 및 성능을 평가하였는데, 논문 [6]은 논리 단과 XOR 게이트 수 모든 면에 있어 두드러진 결과를 보이지 않아서 성능 비교에서 제외하였다. 논문 [8]의 VHDL 코드는 표준 셀 합성이 되지 않아 FPGA 설계툴인 "Synplify Pro 7.0"에서 LookUpTable (LUT) 방식인 "Xilinx Virtex2 XC2v40"을 선택하여 FPGA 합성을 하였으며, SIS는 Synopsys Design Analyzer에서 Synopsys에서 제공하는 LSI_10K표준 셀로 합성을 하여, Proposed와 각각 따로 비교하였다.

표 3은 논문[8]와 Proposed, SIS와 Proposed의 성능 비교표이다. 각각 면적과 임계경로 (Critical Path)의 지연시간 (Arrival Time)을 측정하였다.

표 3. 타이밍 비교

Synthesis Tool (Device or Library)	Synplify (Xilinx Virtex2 XC2V40)		Synopsys (LSI_10K)	
Measure	LUTs	Arrival Time	TCA	Arrival Time
논문[8]	175	5.067ns	X	X
SIS[10]	X	X	1451	14.36ns
Proposed	175	3.650ns	1234	11.56ns

- LUTs: Number of Look Up Table
- TCA: Total Cell Area (Number of NAND gates)

결과적으로 본 논문에서 제시한 알고리즘을 통해 XOR 게이트 수와 지연시간이 각각 42.9%~6.2%, 27.9%~19.7% 감소했음을 알 수 있다.

5. 결론 및 향후계획

본 논문에서는 속도 최적화에 초점을 두고 논리 단을 최소로 유지하면서 가능한 많은 XOR 텀을 공유하도록 매핑하여 면적을 줄일 수 있는 알고리즘을 제시하였다. 또한, FPGA와 표준 셀 합성을 통해 면적과 지연에 있어 최적화된 이더넷 32비트 CRC 회로를 구현하였다. 앞으로 다양한 최적화 알고리즘의 장단점을 분석하여, CRC 이외의 다양한 회로에 적용할 수 있도록 개선 및 일반화하는 연구를 계속할 것이다.

References

[1] D. V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up," Comm. ACM, Aug. 1988.

[2] S. M. Joshi, P. K. Dubey and M. A. Kaplan, "A New Parallel Algorithm for CRC Generation,"IEEE International Conference on Communications, Vol. 3, pp. 18-22, Jun. 2000.

[3] Cypress Semiconductor Corporation, "Parallel Cyclic Redundancy Check (CRC) for HOTLINKTM," Application note, Mar. 1999.

[4] T. B. Pei and C. Zukowski, "High-Speed Parallel CRC Circuits in VLSI,"IEEE Transaction on Communications, Vol. 40, no. 4, pp. 653-657, 1992.

[5] R. F. Hobson and K. L. Cheng, "A High-Performance CMOS 32-Bit Parallel CRC Engine," IEEE Journal of Solid-State Circuits, Vol. 34, No. 2, pp. 233-235, Feb. 1999.

[6] M. D. Shieh et al., "A Systematic Approach for Parallel CRC Computations," J. Information Science & Engineering, May 2001.

[7] M. Spachmann, "Automatic Generation of Parallel CRC Circuits," IEEE Design & Test of Computers, Vol.18, pp.108-114, May. 2001.

[8] G. Campobello, G. Patane and M. Russo, "Parallel CRC Realization," IEEE Transactions on Computers, Vol. 52, pp. 63-71, Oct. 2003.

[9] G. D. Micheli, Synthesis and Optimization of Digital Circuits. McGRAW-HILL INTERNATIONAL EDITIONS, 1994.

[10] http://www-cad.eecs.berkeley.edu/Software/software.html