다치 논리 함수의 ESOP 최소화 알고리즘에 관한 연구

송 홍 복†

요 약

본 논문에서는 몇가지 규칙에 의해 ESOP(Exclusive-OR Sum-Of-Products) 함수를 간단화 하는 알고리즘을 제시하였다. 알고리즘은 두 개의 함수에 대한 곱항 변형 연산을 각항의 상태에 따라 선택적으로 반복수행하여 간단화를 행하였다. 다치 입력 2치 다출력 함수를 최소화함으로써 입력 디코더를 이용하여 EXOR PLA를 입력의 최적화를 하였다. 4치 연산회로 함수에 본 알고리즘을 적용하여 EXOR형 논리회로를 설계하였고, 2bit 입력 디코더를 EXOR-PLA의 설계에 적용하였다. 컴퓨터 시뮬레이션(IBM PC 486 상에서 실행)을 통해 제시된 알고리즘을 여러가지 연산 회로에 적용한 결과, 함수의 입력 변수의 수와 관계없이 최소화가 가능하였고, 출력함수의 곱항수를 줄일 수 있음을 알 수 있었다.

A Study on Minimization Algorithm for ESOP of Multiple-Valued Function

Hong Bok Song†

ABSTRACT

This paper presents an algorithm simplifying the ESOP function by several rules. The algorithm is repeatedly performing operations based on the state of each terms by the product transformation operation of two functions and thus it is simplifying the ESOP function through the reduction of the product terms. Through the minimization of the product terms of the multi-valued input binary multi-output function, an optimization of the input has been done using EXOR PLA with input decoder. The algorithm when applied to four valued arithmetic circuit has been used for a EXOR logic circuit design and the two bits input decoder has been used for a EXOR-PLA design. It has been found from a computer simulation(IBM PC486) that the suggested algorithm can reduce the product terms of the output function remarkably regardless of the number of input variables when the variable AND-EXOR PLA is applied to the poperation circuit.

1. 서 론

반도체 기술의 발달로 집적도가 비약적으로 발달하고 있지만 회로가 복잡해짐에 따라 VLSI 설계에 있어서도 회로의 간단화를 요구되고 있다.

MC68020, Intel80386, IBM Micro/370등과 같은 현대의 32bit 마이크로 프로세서의 제어회로 부분에 PLA(Programmable Logic Array)는 유용하게 이용되고 있으며, 대규모 PLA의 논리 간단화를 위한 설계 자동화용 프로그램이나 그 외의 여러 가지 개발용 툴이 개발되고 있다.

일반적으로, 임의의 논리함수는 AND와 OR게이트

※본 논문은 1996년도 동의대학교 자체 학술연구조성비(자유공모)에 의해 연구되었음.
† 정 회 원:동의대학교 전자공학과
논문접수:1996년 10월 29일, 심사완료:1997년 5월 28일

에 의해 쉽게 표현할 수 있으므로 AND-OR 2단 논리 회로의 설계방법에 대한 많은 연구가 계속되어 왔고, VLSI/ULSI의 기술의 발달로 이를 용이하게 실현할 수 있다. AND-OR형 PLA는 규칙적인 구조와 설계, 테스트, 변형이 비교적 간단하기 때문에 임의의 논리 함수의 설계, 마이크로 프로세서 및 LSI/VLSI 설계에 널리 이용되고 있다. 또 설계 공정, 비용 및 기간을 단축하고, 소자수의 감소에 따른 장치의 소형화 등의 효과로 많이 사용되고 있다.

최근 MVL(Multiple-Valued Logic)함수를 충족시켜 주는 PLA의 개발이 시작되고 있다. MVL-PLA의 합성은 SOP(Sum-Of-Products)형태의 함수로 표현된다. PLA의 곱항은 Column에 의해되고, 이 항들은 함수를 수행하기 위해 여러항이 조합된다. 곱항은 입력 변수의 리터럴(literal)의 OR로 구성되고 SUM 동작의 조합은 SUM 또는 Max이다. SOP-PLA는 곱항의 수가 많고 결선의 수가 많아지는 반면 ESOP(Exclusive-OR Sum-Of-Products) 논리함수를 사용한 EXOR-PLA를 이용하여 ALU(Arithmetic Logic Unit) 등의 연산 회로를 구성할 경우 게이트 수와 곱항이 대폭 감소되므로 EXOR 게이트를 기본 논리 소자로 한 회로설계에 대한 연구가 많이 진행되고 있다. EXOR을 사용한 ESOP형태의 다치논리 함수의 표현은 회로가 간단해지고, 게이트 수와 칩 면적이 감소하고, 약간의 회로를 부가함으로써 만능 검사가 가능해진다는 장점이 있어, 최근 들어 ESOP함수의 literal과 곱항의 수를 줄이는 방법들이 계속 연구되어 왔다. 특히 2 bit 입력 디코더를 갖는 ESOP형 PLA는 다치입력 2치 출력 함수의 곱항의 수를 최소화할 수 있어 많은 연구가 이루어지고 있다[1, 2, 3, 4].

Promper와 Armstrog이 최소항을 불규칙적으로 선택하고, 선택된 최소항을 implicant cover에 의해 주항을 선택하는 MVL 함수의 직접 cover방법을 제안하였고[5], Besslich는 가장 격리된 최소항의 cover를 먼저 찾고 가장 큰 implicant를 선택하는 새로운 직접 cover 방법을 제시하였다[6].

M. Perkowiski는 다치입력 ESOP 함수를 부울함수로 처리하여 2치 출력을 변수 배열에 의해 인접 주항과 각변수의 상태에 따라 주항간의 그룹핑을 행하는 Xlinking방법으로 간략화를 하였고[7], T. Sasao는 다치입력 2치출력의 ESOP형 함수식의 변수를 배열하고 주항의 쌍을 재배열하는 일곱가지 방법을 반복 개선에 의해 곱항수의 최소화를 행하였다[8]. M. Perkowiski는 진리치의 결합이 불가능한 리터럴을 변수의 수를 큐브의 거리로 한 3가지 exorlink방법에 의한 주항 재배열의 반복으로 최소화를 수행하는 알고리즘을 제시하였다[9].

본 논문에서는 곱항의 재배열과 exorlink 수행에 의한 다치입력 다출력 함수의 ESOP 형태의 최소화 기법의 새로운 알고리즘을 제시한다. 이와 같은 최소화 방법의 하나로 본 논문에서는 AND-EXOR 논리함수에 대한 최소논리식의 최소화와 입력 디코더를 갖는 PLA의 최소화 설계 알고리즘을 제시한다. 제안된 방법에 의해서 연산논리 함수의 최소논리식을 구하고 다치 입력 다출력 함수의 간단화 방법인 입력 디코더를 가진 PLA의 설계 방법에 대해서 곱항수를 줄이는 최소화 방법을 논한다.

본 논문에서 제시한 최소화 알고리즘은 LSI, VLSI 논리설계 및 마이크로 프로세서 설계시 회로의 간단화를 통해 칩 면적을 상당히 줄일 수 있을 것으로 기대한다.

2. 수학적 배경과 다치입력 다출력 함수

2.1 다치 입력 2치 출력함수

다치입력, 2치 출력 함수 f는

$$f(X_1, X_2, \cdots, X_n): P_1 \times P_2 \times \cdots \times P_n \rightarrow B \qquad (1)$$

로 나타낼 수 있다.

여기서 X는 다치 입력 변수 $P_i=\{0, 1, \cdots, P_i-1\}$ $(i=0, 1, \cdots, n)$는 변수의 진리치의 집합, $B=\{0, 1\}$, X는 입력변수로 $P=\{0, 1, \cdots, r-1\}$로 4치의 경우 r=4이다.

[정의 1]

임의의 부분집합 $S_i \in P_i$에서 $X_i^{S_i}$를 리터럴 함수로 표현하면 식(2)와 같다.

$$X_i^{S_i} = \begin{cases} 1 : \text{if } X_i \in S_i \\ 0 : \text{if } X_i \notin S_i \end{cases} \qquad (2)$$

곱항은 리터럴의 곱 $X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$로 나타내어지고

AND 연산을 한다. 모든 변수 $X_1^{S_1}$, $X_2^{S_2}$,$\cdots X_n^{S_n}$로 구성되어진 곱항을 전항(full term)이라 한다.

[정의 2]
r치 n변수의 다치논리 함수 $f(X_1, X_2,\cdots, X_n)$을 SOP의 리터럴 함수로 표현하면 식 (3)과 같다.

$$f(X_1, X_2,\cdots, X_n)=\sum(C\prod_{i=1}^{n}(X_n^{S_n})) \tag{3}$$

여기서 $S_i\in P_i$ 및 $0\neq C\in r$ (3)에서 배타적 논리화를 이용한 곱의합 형의 논리식(ESOP : Exclusive-Or Sum-Of-Products)으로 표현할 수 있다.

$$f(X_1, X_2,\cdots, X_n)=\sum\oplus X_1^{S_1}\cdot X_2^{S_2}\cdots X_n^{S_n} \tag{4}$$

A, B, C를 다치 입력 리터럴이라 하면 EXOR 연산에 의해 다음을 만족한다.

$$1.\ A\oplus(B\oplus C)=(A\oplus B)\oplus C \tag{5}$$
$$2.\ A\oplus B=B\oplus A \tag{6}$$
$$3.\ A\oplus A=0 \tag{7}$$
$$X_i^{S_i}\oplus 1=X_i^{P_i-S_i} \tag{8}$$
$$X_i^{S_i}\oplus X_i^{P_i-S_i}=X_i^{P_i}=1 \tag{9}$$
$$X_r^{S_i}\oplus X_i^{S_j}=X_r^{(S_i-S_j)\cup(S_j-S_i)} \tag{10}$$
$$\begin{aligned}&X_i^{S_i}X_j^{S_j}\oplus X_i^{R_i}X_j^{R_j}\\&=X_i^{(S_i-R_j)\cup(R_i-S_j)}X_j^{S_j}\oplus X_i^{R_i}X_j^{(S_j-R_j)\cup(R_j-S_j)}\\&=X_i^{(S_i-R_j)\cup(R_i-S_j)}X_j^{R_j}\oplus X_i^{S_i}X_j^{(S_j-R_j)\cup(R_j-S_j)}\end{aligned} \tag{11}$$

2.2 다치입력 다출력 함수

다출력 함수는 각 출력함수를 독립적으로 최소화하는 방법으로는 전체적인 PLA의 곱항수가 최소화될 수 없으므로 출력 전체에 대한 최소화가 요구되어진다.

[정의 3]
다치 입력 2치 m출력 함수를 $f(X_1, X_2,\cdots, X_{n-1})=(f_0,\cdots, f_{m-1})$라 할 때, 함수 f의 하나의 출력 X_i가 n치 값을 가지고 변수 X_n가 $\{0, 1,\cdots, m-1\}$일 때 함수 f는 출력 측을 하나의 변수로 가지는 함수로 다음과 같이 표현된다.

$$f(X_1, X_2,\cdots, X_{n-1}, X_n=i)=f_i(X_1,\cdots, X_{n-1}) \tag{12}$$

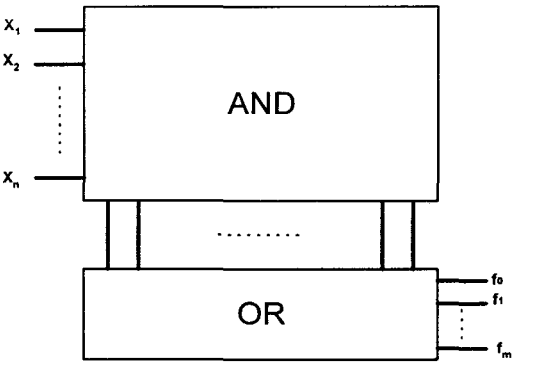
여기서 $f_i(X_1,\cdots, X_{n-1})$를 함수 f의 변수 X_n의 i번째 $i=\{0, 1,\ldots, n-1\}$ 출력으로 하여 $f(X_1, X_2,\cdots, X_{n-1}, X_n)$와 같이 함수 f를 재정의 할 수 있다.

입력중 하나의 출력 switching 함수를 입력의 n번째 변수로 사용함으로 다출력 함수를 표현한다. 3입력 2출력 이치 함수 f(a, b, c)=(f_0, f_1)를 생각할 때 출력 f_0와 f_1를 입력변수 X를 이용하여 나타내면 주어진 함수는 f(a, b, c, X)와 같이 2치 입력 X를 포함한 4입력 함수로 나타내어진다.

3. ESOP-PLA의 구성

3.1 AND-OR PLA의 구성

AND-OR 2단 논리회로로 임의의 논리함수를 표현할 수 있다. 보통 제어 회로 등은 그 기능을 진리표로 표현하는 것 보다 논리식을 이용하면 훨씬 간단하므로, PLA는 논리식을 이용하여 회로를 구성한다. (그림 1)은 일반적인 AND-OR 2단 PLA를 나타내었다.



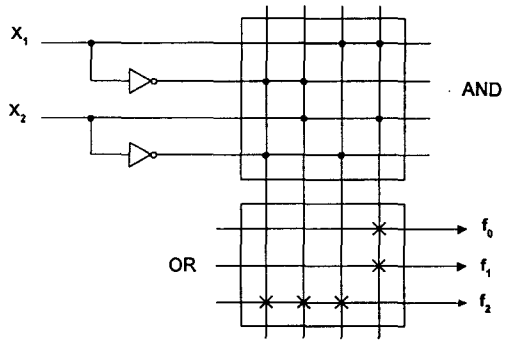
(그림 1) 일반적인 AND-OR PLA 구성도
(Fig. 1) Blockdiagram of standard AND-OR PLA

AND-OR 2단 PLA를 이용하여 〈표 1〉의 진리표를 실현하려면 f_0와 f_1를 나타내는 곱항의 논리합형 $f_0=X_1X_2$, $f_1=X_1X_2$과 f_2를 나타내는 $f_2=\overline{X_1}X_2+X_1\overline{X_2}$를 구하면 된다. (그림 2)는 〈표 1〉을 AND-OR PLA로 실현한 것이며 AND 어레이의 각 곱항선의 왼쪽부터 X_1X_2, $\overline{X_1}X_2$, $X_1\overline{X_2}$, $\overline{X_1}\,\overline{X_2}$를 나타내고 있다. 또한 OR어레이는 각 곱항의 논리합을 나타내며

이 PLA에서 첫 번째 곱항은 f_0와 f_1 양쪽의 출력에 접속되어 있다. 주어진 진리표를 구현하기 위해서는 5개의 곱항이 필요하고, 이 PLA를 최소화 하는데는 모든 출력을 동시에 고려한 다출력 함수의 간단화가 필요하다.

〈표 1〉 진리표
〈Table 1〉 Truth Table

X_1	X_2	f_0	f_1	f_2
0	0	0	0	1
0	1	0	0	1
1	0	0	0	1
1	1	1	1	0



(그림 2) 〈표 1〉의 함수를 실현하는 AND-OR형 PLA
(Fig. 2) AND-OR PLA for 〈Table 1〉

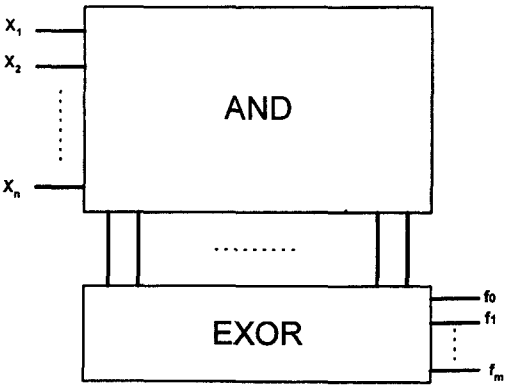
3.2 AND-EXOR PLA의 구성

AND-EXOR형 PLA는 AND-OR형 PLA의 OR 어레이 부분을 EXOR로 대치하여 구성되어 있다. AND-EXOR PLA는 AND-OR형 PLA에 비해 AND 어레이가 줄어들고, 회로의 검사가 용이한 장점을 가지고 있다.

(그림 3)은 AND-EXOR PLA의 구조도를 나타낸 것이다.

(그림 4)는 〈표 1〉의 진리표를 AND-EXOR형 PLA를 이용하여 설계한 구조도이며 변수는 정 및 부정변수를 모두 사용하고 있다.

이것은 만능 검사를 갖는 AND-EXOR형 PLA의 기본적인 방법이다. AND-EXOR형 PLA는 고장검사 시간이 짧으며 표준 PLA보다 곱항수로 논리 함수를 실현 할 수 있다.



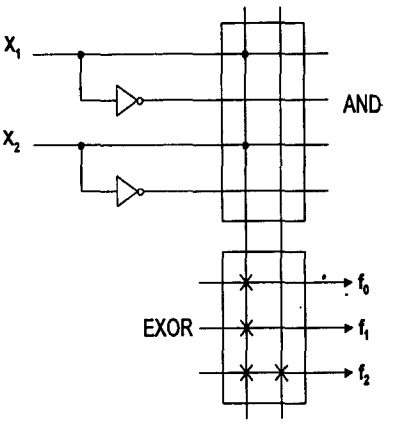
(그림 3) 입력 디코더를 갖지 않는 AND-EXOR형 PLA
(Fig. 3) AND-EXOR PLA without input decoder

예)

〈표 1〉의 2입력 출력 함수 $F=(f_1, f_1, f_2)$를 고려하면 각 출력 f_0, f_1, f_2를 X_3를 이용하여 F를 나타내면 ESOP 최소화 함수로 식 (13)과 같이 된다.

$$F=X_1^1 \cdot X_2^1 \oplus X_3^2 \tag{13}$$

여기서 X_3는 3개의 출력을 가지는 3치변수 $X_3=0$, $X_3=1$과 $X_3=2$로 나타내면 f_0, f_1, f_2는 $f_0=f_1=X_1^1 \cdot X_2^1$, $f_2=X_1^1 \cdot X_2^1 \oplus 1$로 나타낼수 있다.



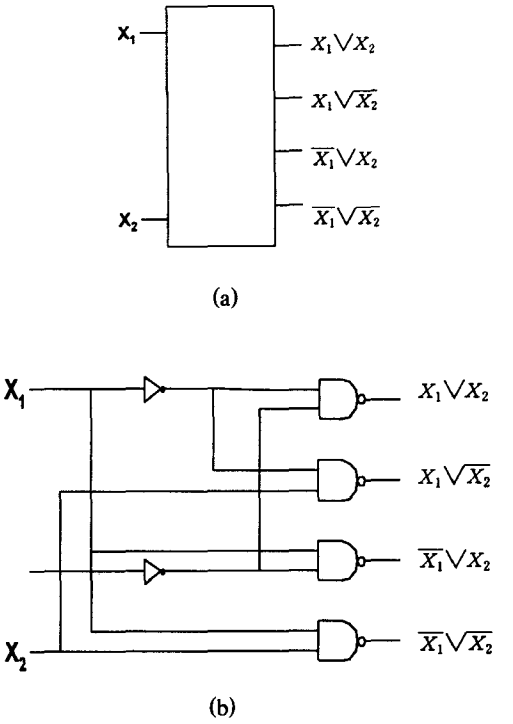
$F=(f_0, f_1, f_2)$

(그림 4) 〈표 1〉의 함수를 실현하는 AND-EXOR형 PLA
(Fig. 4) AND-EXOR PLA for 〈Table 1〉

여기서 각출력 f_0, f_1, f_2는 그림 4와 같이 AND-EXOR 형 PLA으로 실현 할 수 있다.

4.2 bit입력 디코더를 가진 AND-EXOR PLA

2 bit 디코더를 부가한 회로는 입력 변수를 디코더를 이용하여 두 변수의 논리합으로 변형한다. (그림 5)는 2비트 입력 디코더를 나타내었다.



(그림 5) 2 bit 입력 디코더
(Fig. 5) 2 bit input decoder

2 bit 입력 디코더는 2변수의 논리합의 최대항으로 구성되어지고, 디코더의 출력은 $X_1 \vee X_2$, $X_1 \vee \overline{X_2}$, $\overline{X_1} \vee X_2$, $\overline{X_1} \vee X_2$를 생성한다. 임의의 논리 함수는 이 최대항의 논리적 형태로 표현되어지고 2변수의 경우 다음과 같은 형식으로 나타낼 수 있다.

$$f(X_1, X_2) = (c_0 \vee X_1 \vee X_2)\,(c_1 \vee X_1 \vee \overline{X_2})\,(c_2 \vee \overline{X_1} \vee X_2)\,(c_0 \vee \overline{X_1} \vee \overline{X_2}) \quad (14)$$

여기서 $c_i (i = 0, 1, 2, 3)$는 0 또는 1의 값을 가진다.

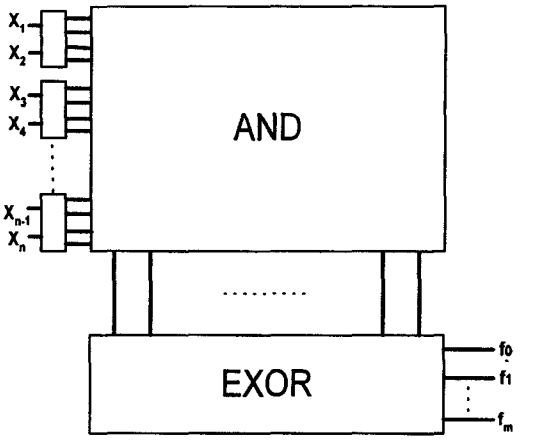
예를 들어 $(c_0 c_1 c_2 c_3) = (1, 0, 0, 1)$라 할 때 $c_i = 0$인 항의 논리적 형태로 함수 f를 식 (15)와 같이 나타낼 수 있다.

$$f(X_1, X_2) = (X_1 \vee \overline{X_2})\,(\overline{X_1} \vee X_2) = X_1 X_2 \vee \overline{X_1 X_2} \quad (15)$$

c_i가 0과 1의 값을 가지므로 $(c_0 c_1 c_2 c_3)$의 조합은 표 2과 같이 $2^4 = 16$개를 가진다.

〈표 2〉 2bit 입력 디코더 출력
〈Table 2〉 Output of 2bit decoder

c_0	c_1	c_2	c_3	논리식	c_0	c_1	c_2	c_3	논리식
0	0	0	0	0	1	0	0	0	$\overline{X_1 X_2}$
				$X_1 X_2$					$X_1 \oplus X_2$
0	0	0	1	$X_1 X_2$	1	0	0	1	X_2
0	0	1	0	X_1	1	0	1	0	$\overline{X_1} \vee X_2$
0	0	1	1	$\overline{X_1} X_2$	1	0	1	1	$\overline{X_1} \vee X_2$
0	1	0	0	$X_1 \oplus X_2$	1	1	0	0	$X_1 \vee X_2$
0	1	0	1	$X_1 \vee X_2$	1	1	0	1	1



(그림 6) 2 bit입력 디코더를 가진 AND-EXOR형 PLA
(Fig. 6) AND-EXOR PLA with 2 bit input decoder

5. 다처 입력 다출력 함수의 최소화 논리

5.1 ESOP 함수의 간단화 방법

제안하는 알고리즘은 다음 7종류의 곱항 변형 규칙을 반복 적용 하여 ESOP 함수의 간단화를 행한다. 곱

항 변형 규칙은 각 곱항이 상호 최소항을 공유하지 않는 DSOP(Disjoint Sum-Of-Product) 형태에서 OR 연산을 EXOR연산으로 치환하여 동일한 함수를 표시하는 ESOP로 변환하고, X-MERGE와 RESHAPE를 반복하여 곱항수가 적은 ESOP를 얻어 최소화를 수행한다.

(1) X-MERGE

$$X^a \oplus X^b = X^{(a \oplus b)} \tag{16}$$

(2) RESHAPE

$$X^a Y^b \oplus X^c Y^d = X^a Y^{(b \cap \bar{d})} \oplus X^{(a \cup c)} Y^a \tag{17}$$

if $(a \cap c = \phi,\ b \supset d)$

(3) DUAL-COMPLEMENT

$$X^a Y^b \oplus X^c Y^d = X^c Y^{(b \cap \bar{d})} \oplus X^{(\bar{a} \cup c)} Y^b \tag{18}$$

if $(a \subset c,\ b \supset d)$

(4) X-EXPAND-1

$$X^a Y^b \oplus X^c Y^d = X^a Y^{(b \cup d)} \oplus X^{(a \cup c)} Y^a$$
$$= X^{(a \cup c)} Y^b \oplus X^c Y^{(b \cup d)} \tag{19}$$

if $(a \cap c = \phi,\ b \cap d = \phi)$

(5) X-EXPAND-2

$$X^a Y^b \oplus X^c Y^d = X^{(a \cup c)} X^b \oplus X^c Y^{(b \cup \bar{d})} \tag{20}$$

if $(a \cap c = \phi,\ b \supset d)$

(6) X-REDUCE-1

$$X^a Y^b \oplus X^c Y^d = X^{(a \cap \bar{c})} Y^b \oplus X^c Y^{(\bar{d} \cup \bar{b})} \tag{21}$$

if $(a \supset c,\ b \subset d)$

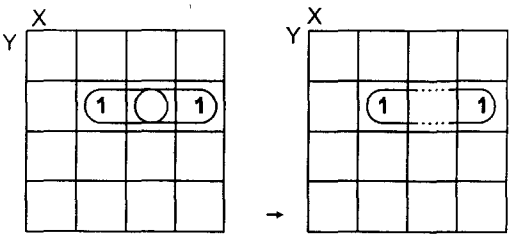
(7) X-REDUCE-2

$$X^a Y^b \oplus X^c Y^d = X^{(a \cup \bar{c})} Y^b \oplus X^c Y^{(b \cup \bar{d})}$$
$$= X^a Y^{(b \cup \bar{d})} \oplus X^{(a \cup \bar{c})} Y^a \tag{22}$$

if $(a \supset c,\ b \supset d)$

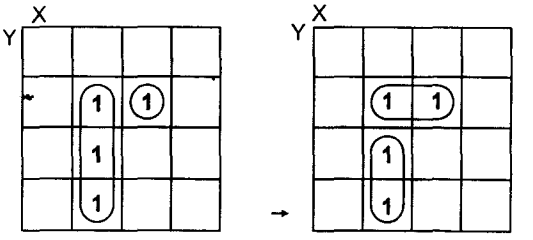
(그림 7.1~7.7)은 ESOP 함수의 곱항 변형 규칙을 4치(four-valued) 입력의 경우를 예로 나타낸 것이다.

여기서 곱항 변형 규칙 중 곱항의 수를 줄일수 있는 방법은 X-MERGE뿐이다. 나머지 변형규칙은 X-MERGE를 이용하여 곱항을 감소시키기 위해 곱항을 변형하는 방법이다. (2)의 RESHAPE는 SOP의 간단화에도 적용되는 규칙이며, (3)~(7)의 규칙은 ESOP 고유의 규칙이다. 변형 규칙에서 RESHAPE와 X-EXPAND-2, DUAL-COMPLEMENT와 X-REDUCE-1는 동일 입력이 변형의 방법에 따라 서로 다른 출력을 보여주고 있다. 본 논문에서는 이러한 규칙을 입력변수의 조건에 따라 선택적으로 연산을 하여 AND 게이트수를 최소화한다.



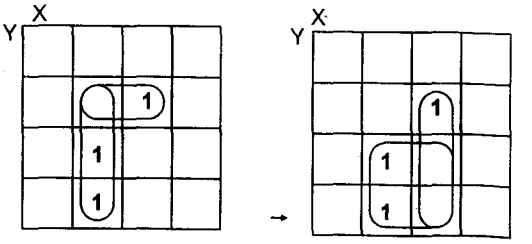
$X^{12}Y^1 \oplus X^{23}Y^1 = X^{13}Y^1$

(그림 7.1) X-MERGE
(Fig. 7.1) X-MERGE



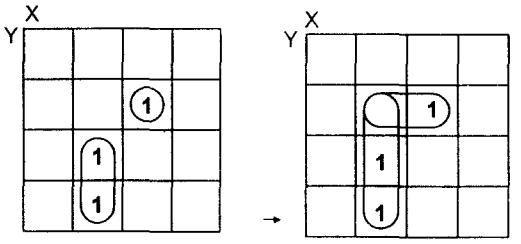
$X^1Y^{123} \oplus X^2Y^1 = X^{12}Y^1 \oplus X^1Y^{23}$

(그림 7.2) RESHAPE
(Fig. 7.2) RESHAPE



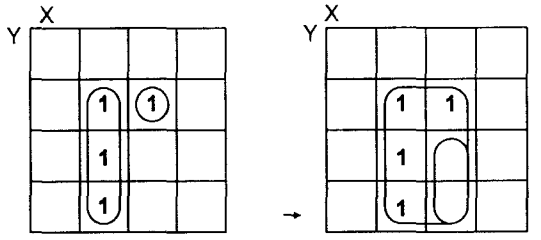
$X^1Y^{123} \oplus X^{12}Y^1 = X^{12}Y^{23} \oplus X^2Y^{123}$

(그림 7.3) DUAL-COMPLEMENT
(Fig. 7.3) DUAL-COMPLEMENT



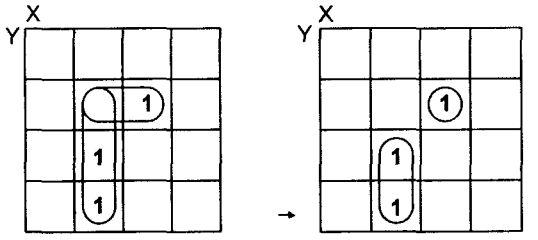
$X^1Y^{23} \oplus X^2Y^1 = X^1Y^{123} \oplus X^{12}Y^1$

(그림 7.4) X-EXPAND-1
(Fig. 7.4) X-EXPAND-1



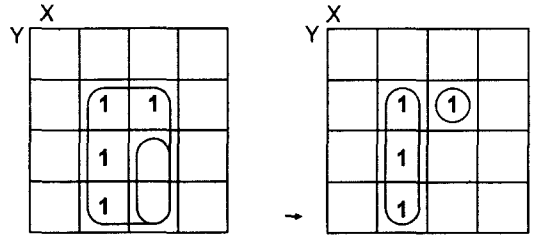
$X^2Y^1 \oplus X^1Y^{123} = X^{12}Y^{123} \oplus X^2Y^{23}$

(그림 7.5) X-EXPAND-2
(Fig. 7.5) X-EXPAND-2



$X^{12}Y^1 \oplus X^1Y^{123} = X^1Y^{23} \oplus X^2Y^1$

(그림 7.6) X-REDUCE-1
(Fig. 7.6) X-REDUCE-1



$X^2Y^{23} \oplus X^{12}Y^{123} = X^1Y^{123} \oplus X^2Y^1$

(그림 7.7) X-REDUCE-2
(Fig. 7.7) X-REDUCE-2

5.2 exorlink를 이용한 다치입력 출력 함수의 간단화

제안된 알고리즘은 exorlink연산을 기본적인 연산으로 하고 있다. 이 연산은 간단화 규칙의 7가지 방법을 모두 사용하는데 몇가지 방법에 따른 기본적인 조건을 필요로 한다.

두 개의 cube를 $C_s = X_1^{S_1} \cdots X_n^{S_n}$, $C_R = X_1^{R_1} \cdots X_n^{R_n} \neq C_s$ 라 하면, 큐브 C_s와 C_R의 exorlink연산은 다음과 같이 정의된다.

$$C_S \otimes C_R = \bigcup_{i=1}^{r}$$

$$\left[(X_{j_1}^{S_j} \cdots X_{j_{i-1}}^{S_{j_{i-1}}} X_{j_i}^{(S_{j_{i-1}} - R_{j_i}) \cup (R_{j_i} - S_{j_i})} X_{j_{i+1}}^{R_{j_{i+1}}} \cdots X_{j_r}^{R_{j_r}}) (\prod_{k=1}^{n-r} X_{j_k}^{S_{j_k}}) \right] \quad (23)$$

두 개의 항의 쌍을 exorlink연산을 통해 변형된 곱항을 형성하는 예로 $A^{01}B^{01}C^{012}D^2E^{13}$와 $A^{12}B^{12}C^2D^2E^{13}$로 주어진 4치 함수를 exorlink의 예로 나타내었다.

주어진 함수의 각 변수의 쌍을 비교하여 변수의 리터럴에 대해 연산을 수행하여 결과항을 얻는다. 함수에서 A, B, C가 다른 리터럴을 가지고 있으므로 각 변수에 대해 연산을 수행하여 3개의 곱항을 구할 수 있다.

변수 A에 대하여 $A^{02}B^{12}C^2D^2E^{13}$를 다음과 같이 첫 번째 항으로 얻을 수 있다.

$A^{01}B^{02}C^{012}D^2E^{13}$
$A^{12}B^{12}C^2D^2E^{13}$

$A^{02}B^{12}C^2D^2E^{13}$

변수 B에 대하여 $A^{01}B^{01}C^2D^2E^{13}$가 두 번째 곱항으로 나타나고,

$A^{01}B^{02}C^{012}D^2E^{13}$
$A^{12}B^{12}C^2D^2E^{13}$

$A^{01}B^{01}C^2D^2E^{13}$

변수 C에 대하여 $A^{02}B^{12}C^2D^2E^{13}$를 세 번째 결과로 얻는다.

$A^{01}B^{02}C^{012}D^2E^{13}$

$A^{12}B^{12}C^{2}D^{2}E^{13}$

$A^{01}B^{02}C^{01}D^{2}E^{13}$

각각의 결과를 EXOR연산에 의해 ESOP 형태로 나타내면 exorlink의 결과를 다음과 같이 얻을 수 있다. 주어진 변수의 다른 리터럴 수에 따라 결과의 항수가 달라지게 된다. 예에서는 3개의 다른 리터럴을 가진 함수이므로 곱항의 수가 3개가 된다.

$$A^{01}B^{02}C^{012}D^{2}E^{13} \oplus A^{12}B^{12}C^{2}D^{2}E^{13} = A^{02}B^{02}C^{2}D^{2}E^{13} \oplus A^{01}B^{01}C^{2}D^{2}E^{13} \oplus A^{01}B^{02}C^{01}D^{2}E^{13} \quad (24)$$

알고리즘에서는 주어진 함수를 먼저 리터럴의 수에 따라 sorting하고 각 큐브의 쌍을 연산하여 새로운 큐브를 생성하고 곱항수를 비교하여 다음 연산을 행한다.

본 논문의 간단화 알고리즘의 목적은 곱항수를 줄여 PLA의 면적을 줄이는데 있다. 그리고 AND와 EXOR게이트의 총 입력선 최소화로 회로의 항수를 줄여 함수를 간단화하고 있다.

알고리즘의 반복 수행을 위해 Cost함수 C를 다음과 같이 정의한다.

$$C = NT + \frac{NI}{NI_i} \quad (25)$$

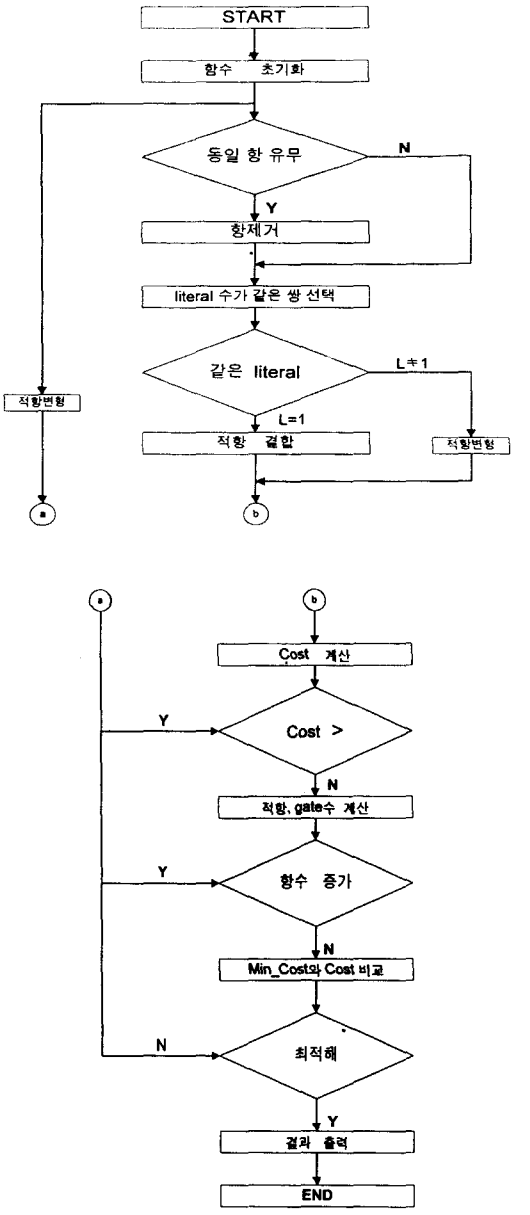
NT : 결과 해의 항의 총수
NI : 결과 해의 AND와 EXOR 게이트의 입력의 총수
NIi : 초기함수의 AND와 EXOR 게이트의 입력 총수

예를 들어 X_{012}는 2-by-4 디코더를 사용한 경우 1개의 AND 게이트 입력이 요구된다. $X_0 = X_{012}X_{013}X_{023}$는 3개의 입력선을 요구하고 $X_0Y_1 \oplus X_1Y_2$는 두 개의 합을 가지므로 14개의 입력선, 즉 12개의 AND게이트와 2개의 EXOR 게이트 입력이 요구된다.

5.3 최소화 알고리즘

최소화 알고리즘은 그림 8과 같으며, 주어진 SOP 형태의 함수를 ESOP형태로 변환하고 각 큐브의 리터럴 수에 따라 X-Merge연산을 통해 큐브를 확대하여 곱항의 수를 감소시키고, 큐브쌍의 조건에 따라 exorlink를 수행하여 변형된 큐브를 만들어 다시 X-Merge를 수행하여 같은 방법으로 간략화를 행한다.

항의 수가 최소화 될 경우 새로운 변형을 시도하여 앞의 방법을 Cost함수가 최소가 될 때까지 반복 수행



(그림 8) 최소화 알고리즘의 순서도
(Fig. 8) Flowchart for minimization algorithm

한다. exorlink수행시 큐브쌍의 조건에 따라 여러가지 간단화 연산을 하게 되는데 이때 리터럴의 형태에 따라 수행되는 결과가 주어진 간략화 방법 중 하나를 선택하게 된다.

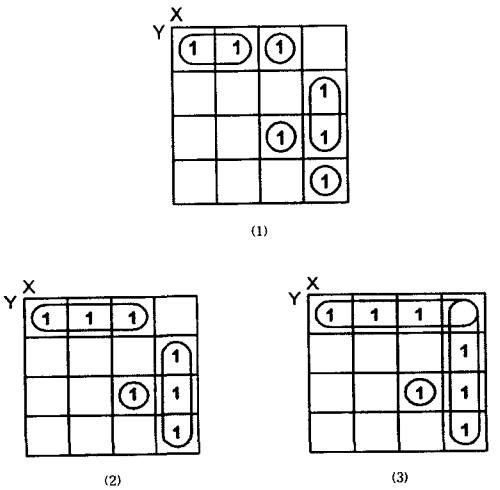
6. 실험 결과 및 고찰

6.1 간단화 알고리즘의 적용 예

(그림 9)는 카르노도(Karnaugh map)로 표시한 2치 4입력 함수에 알고리즘을 적용한 예이다. 그림에서 나타나는 SOP는 이미 DSOP형태로 되어있으므로 바로 ESOP형태로 바꾸어 알고리즘을 적용하면 된다. 적용 예에서는 곱항 변형규칙 중 X-Merge와 X-expand-1만을 이용하여 처리하면 된다. X-Merge와 X-expand-1를 적용함에 있어 exorlink연산을 이용하여 연산을 수행하였다.

적용 예)

(그림 9)의 (1)에서 두 개의 항에 X-Merge를 적용하여 곱항을 확대하면 새로운 항을 얻어 (그림 9)의 (2)같이 변형된다. (그림 9)의 (2)에서 X-expand-1을 적용하면 (그림 9)의 (3)과 같은 결과를 얻는다. 이상과 같이 규칙을 적용하여 곱항의 수를 줄이고 최종결과를 얻으면 알고리즘은 정지한다. 알고리즘에서는 X-expand-1등의 최소화 방법을 exorlink를 이용하여 수행하였으며, 결과는 곱항수 3개의 ESOP가 된다. 적용 예의 경우에서 최소화는 2가지 수행으로 이루어지므로 제안된 알고리즘은 간단한 방법을 사용하여 고속으로 처리된다.



(그림 9) 최소화의 예
(Fig. 9) Example of Minimization

6.2 알고리즘 수행에 의한 최소화

입력 함수가 식 26과 같이 주어질 때 f를 간단화 하면

$$f = A^0B^1Y^0 \oplus A^0B^2Y^1 \oplus A^0B^3Y^{01} \oplus A^1B^0Y^0 \oplus A^1B^1Y^1 \oplus A^1B^2Y^{01} \quad (26)$$

먼저 6개의 곱항중 동일항이 있으면 소거하고 X-Merge를 실행하여 항을 결합한다. 주어진 함수에서 동일항과 X-Merge를 실행할 곱항이 없으므로 먼저 ①항과 ②항을 곱항 변형하여 $A^0B^{12}Y^1$와 $A^0B^1Y^{01}$를 생성하고 같은 방법으로 항과 항, 항과 항을 변형하여 중간 함수를 생성한다.

함수의 변형을 동일한 곱항이 발생하거나 X-Merge를 실시할 수 있는 조건이 될 때까지 반복 수행한다.

함수 f를 2번 exorlink를 수행하면 5개의 곱항으로 간단화 할 수 있다.

이와 같이 주어진 함수의 곱항 변형과 동일항 소거를 통해 간단화를 행한다.

$$f = A^0B^1Y^0 \oplus A^0B^2Y^1 \oplus A^0B^3Y^{01} \oplus A^1B^0Y^0 \oplus A^1B^1Y^1 \oplus A^1B^2Y^{01}$$

① $A^0B^1Y^0$	①-② $A^0B^{12}Y^1$	$A^0B^{123}Y^{01}$
② $A^0B^2Y^1$	$A^0B^1Y^{01}$	$A^0B^{12}Y^1$
③ $A^0B^3Y^{01}$	③-⑥ $A^{01}B^2Y^{01}$	$A^{01}B^2Y^{01}$
④ $A^1B^0Y^0$	$A^0B^{23}Y^{01}$	$A^1B^{01}Y^1$
⑤ $A^1B^1Y^1$	④-⑤ $A^1B^{01}Y^1$	$A^1B^0Y^{01}$
⑥ $A^1B^2Y^{01}$	$A^1B^0Y^{01}$	

$$A^0B^1Y^0 \oplus A^0B^2Y^1 \oplus A^0B^3Y^{01} \oplus A^1B^0Y^0 \oplus A^1B^1Y^1 \oplus A^1B^2Y^{01} =$$
$$A^0B^{123}Y^{01} \oplus A^0B^{12}Y^1 \oplus A^{01}B^2Y^{01} \oplus A^1B^{01}Y^1 \oplus A^1B^0Y^{01}$$

6.3 알고리즘 수행 결과

제안된 알고리즘에 의해 수행된 연산회로의 최소화 수행 결과는 표 3과 같다. 먼저 가산회로에 대한 수행 결과는 4bit 2입력 가산회로의 경우 항의 수가

31개로 간단화 할수 있었고, 이때 2bit 입력 디코더를 사용하여 입력변수를 최적화 할 경우 항의 수를 11개로 최소화 할 수 있었다.

알고리즘을 이용하여 몇 가지 산술회로에 대해서 간단화를 행하여 산술회로에 대한 항의 수를 비교하였고, 2비트 디코더를 갖는 AND-EXOR형 PLA에 대해서도 같은 방법을 사용하였다.

〈표 3〉 알고리즘 적용 결과
〈Table 1〉 Algorithm Application result

	EXMIN[5]	제안 알고리즘
ADR2	7	7
ADR4	32	31
2bit 디코더 사용	12	11
MLP3	18	18
MLP4	66	63
2bit 디코더 사용	56	51

7. 결 론

본 논문에서는 exorlink를 선택적으로 사용한 다치 입력 2치 다출력 함수의 최소화 알고리즘을 제시하였으며, AND-EXOR형 PLA의 설계에 이를 적용시켰다. 주어진 함수를 간단화하기 위해 7종류의 곱항 변형 규칙을 사용하였으며, 이를 알고리즘에 적용하기 위해 exorlink를 사용하여 곱항수를 줄여 최소화를 행하였다. 또한 2치 입력을 확장한 다치 입력다출력 함수의 간단화 방법인 입력 디코더를 사용한 ESOP의 PLA 설계방법과 제안한 알고리즘의 적용을 보였다. 4변수 이하의 연산회로 함수에 알고리즘을 적용하여 AND-EXOR 형 논리회로에 대해 비교하였고, 2치 입력을 최적화한 입력 디코더를 ESOP의 PLA의 설계에 적용하였다. 컴퓨터 시뮬레이션(IBM PC 486상에서 실행)하여 다음과 같은 결론을 얻을수 있었다.

- AND-EXOR형 논리 회로에서 함수의 간단화를 입력 변수의 수와 관계없이 최소화가 가능 하였다.
- EXMIN과 알고리즘을 비교하였을 때 6종의 연산회로에서 총 94.7%의 항의 최소화를 얻을수 있었다.

2bit 디코더를 사용할 경우 91.2%의 항수로 연산회로를 구성할수 있어 AND array의 면적을 감소시켜 PLA의 총면적을 줄일수 있을 것으로 기대된다. 따라서 본 논문의 알고리즘은 PLA 설계시 PLA의 칩면적을 보다 효과적으로 줄일수 있고 설계비용과 시간을 절감할수 있어 CPU 등의 회로 설계에 이용될수 있을 것으로 예상된다. 제안된 알고리즘은 회로의 최적화에 관점을 두었으므로 최소화 실행시 최소 논리식을 적용하여 단시간 내에 최소형을 얻는 방법과 칩으로 구현시 발생되는 문제에 대한 연구가 계속되어야 할 것이다.

참 고 문 헌

[1] Sasao T., "On the Optimal Design of Multiple-valued PLA's", in Proc. 16th ISMVL, IEEE Computer Society, pp. 214-223, May. 1986.

[2] Sasao T., "A transformation of multiple-valued input two-valued output functions and its application to simplification of exclusive-or sum-of-products expressions", 21th ISMVL, IEEE Computer Society, pp. 270-279, May. 1991.

[3] Parthasarathy P. Tirumalai and Varadaraian G. Cadakkencherry, "Parallel Algorithms for Minimizing Multiple-Valued Programmable Logic Arrays", 21th ISMVL, IEEE Computer Society, pp. 287-292, May. 1991.

[4] Parthasarathy P. Tirumalai and Jon T. Bulter, "Minimization Algorithms for Multiple-Valued Programmable Logic Arrays", IEEE trans. Comp. pp. 167-177, Feb. 1991.

[5] Promper G. and Armstrong J. A., "Representation of multivalued function using the direct cover method" IEEE trans. Comp. pp. 674-679, Sept. 1981.

[6] Besslich P. W., "Heuristic minimization of MVL function:a direct cover approach", IEEE trans. Computer Society, pp. 134-144. Feb. 1986.

[7] Perkowiski M. A., Helliwell M., and Wu P., "Minimization of Multiple-Valued Input Muti-Output Mixed-Radix Exclusive Sum of Products

for incompletely Specified Boolean function", 19th ISMVL, IEEE Computer Society, pp. 256-263, May. 1989.

[8] Sasao T., "EXMIN:A Simplification Algorithm for Exclusive-or-Sum-Products Expressions for Multiple-Valued Input Two-Valued Output Functions", 20th ISMVL, IEEE Computer Society, pp. 128-135, May. 1990.

[9] Ning Song and Perkowiski M. A. "EXORCISM-MV-2:Minimization of Exclusive Sum of Products Expressions for Multiple-Valued input incompletely Specified Functions", 23th IS MVL, IEEE Computer Society, pp. 132-137, May. 1993.

[10] Dueck G. W. and Miller D. M., "A direct cover MVL minimization using truncated sum", 17th ISMVL, pp. 221-227. 1987.

[11] 笹尾勤, "論理設計 スイッチング回路理論", 近代科學社, 1995.

[12] Sasao T. and Terada H., "Multiple-Valued logic and design of programmable logic array with decoders", Proc. 9th ISMVL, pp. 27-37. 1991.

[13] Dueck G. W. and Miller D. M., "A 4-Valued PLA using the Modsum", Proc. 16th ISMV L, pp. 232-240, May. 1986.

[14] Kerkhoff G. W. and Butler J. T., "Design of a High-Radix Programmable Logic Array using profiled peristatlic charge-Coupled device", Proc. 16th ISMVL, pp. 100-103, May. 1986.

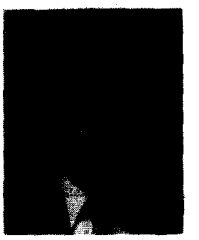
프로그램 소스

```
#include <initval.h>
#include <stdio.h>
#define In  4
#define In_Bit        4
#define Out           5
#define Max   256
#define Bit_All 15

void Init_Function();

void Remove_pair();
void Rep_Link(int Difference_No);
int Difference_Check(int F1_No, int F2_No);
int Cube_C();
int Or_C();
int And_C();
int Cube_Count(int F_No);
int Difference_Count(int F1_No, int F2_No);

float Cost();

struct      In_Variable {

            unsigned A : In_Bit;

};

typedef struct {
              struct In_Variable X[In];
                        unsigned f : Out;
                        unsigned Active : 1;
                     }Cube_Array;

Cube_Array PLA_F[Max], Solution[Max], Temp[In+Out], Temp_F[Max];
Cube_Array *PLA_F_p, *Solution_p, *Temp_p;
int In_And;

main()
{
  int Q_Cost;
  int Or_Cnt, i,j;
  Init_Function();
  In_And=And_C();

for(j=0;j<5;j++){
for(i=0;i<30;i++){

  Move_FtoT();
In_And=And_C();
Rep_Link(2);
  if(In_And>And_C())Move_TtoF();
  if(Check_Difference(1)) Rep_Link(1);
}

  Rep_Link(3);

for(i=0;i<30;i++){

  Move_FtoT();
  In_And=And_C();
  Rep_Link(2);
  if(In_And>And_C())Move_TtoF();
  if(Check_Difference(1)) Rep_Link(1);
}
}

}

float Cost()
{
 float Cost_Val;
```

```
Cost_Val=(float)Cube_C0+((float)And_C0)/(float)In_And0;
return Cost_Val;
}

int Check_Difference(int Difference_No)
{
        int i, j,Diff_Check=0;
        for(i=0;i<Max;i++)
        {
                for(j=i+1;j<Max;j++)
                {
                        if(Cube_Count(i)==Cube_Count(j))
                        {
                                if(Difference_Count(i, j)==Difference_No)
                                {
                                        return Difference_No;
                                }
                        }
                }
        }
        return Diff_Check;
}

void Remove_pair()
{
        int Count, Count1,i,comp;

        for(Count=0;Count<Max;Count++)
        {

                for(Count1=Count+1;Count1<Max;Count1++)
                {

                        if(PLA_F[Count].Active==1)
                        {
                          comp=1;
                          for(i=0;i<In;i++)
                          {
                    if(PLA_F[Count].X[i].A!=PLA_F[Count1].X[i].A) comp=0;
                          }

                    if(comp==1 && PLA_F[Count].f==PLA_F[Count1].f)
                          {
                                PLA_F[Count1].Active=0;
                          }
                        }
                }


        }

}

int Difference_Count(int F1_No, int F2_No)
{
        int Difference=0, i;
        for(i=0;i<In;i++)
        {
                if(PLA_F[F1_No].X[i].A!=PLA_F[F2_No].X[i].A) Difference++;
        }
        if(PLA_F[F1_No].f!=PLA_F[F2_No].f) Difference++;
        return Difference;
}

int Cube_Count(int F_No)
{
        int Literal=0, i;
        for(i=0;i<In;i++)
        {
                if(PLA_F[F_No].X[i].A!=Bit_All) Literal++;
        }
        if(PLA_F[F_No].f!=0) Literal++;
        return Literal;
}

int Cube_C()
{
        int i;
        int C_Term=0;

        for(i=0;i<Max;i++){

        if(PLA_F[i].Active==1) C_Term++;

        }

        return C_Term;
}

int Or_C()
{
        int i, k, Mask_Or, val;
        int C_Or=0;

        for(i=0;i<Max;i++){

                Mask_Or=1;

      if(PLA_F[i].Active==0) continue;

                for(k=0;k<Out;k++){

                        val=PLA_F[i].f & Mask_Or;

                        if(val!=0){
                                C_Or++;
                        }

                        Mask_Or=Mask_Or<<1;
                }

        }

        return C_Or;
}
```

```
int And_C()
{
        int i, k, j;
        int C_And=0;
        int Test_Bit;

        for(i=0;i<Max;i++){
                for(k=0;k<In;k++){

                        Test_Bit=1;

                        if(FLA_F[i].X[k].A!=Bit_All && FLA_F[i].Active==1){

                                for(j=0;j<In_Bit;j++){

                                        if(FLA_F[i].X[k].A & Test_Bit){
                                                C_And++;
                                        }

                                        Test_Bit=Test_Bit<<1;
                                }

                        }

                }

        }

        return C_And;
}

void Rep_Link(int Difference_No) /* Ex-Or Link를 반복하여 수행 */
{
        int i, j ,k,l,n,m,diff=0 ;
    int Solution_No, diff_no, Rep_Ok;
        int Or_Count,Or_Count1,Or_Count2,Or_Count3, Xlink_Ok=0;
        float Old_Cost, Old_Cost1, Old_Cost2, Old_Cost3;
    Old_Cost1=Old_Cost2=Old_Cost3=Old_Cost=Cost();

      Or_Count=Or_Count1=Or_Count2=Or_Count3=Or_C();
Rep_Ok=0;

do
{
    Solution_No=0;
    Sorting_Cube();
    Old_Cost=Cost();
    Or_Count=Or_C();

    if(Or_Count>Or_C()) Or_Count=Or_C();
    for(i=0;i<Max;i++)
    {
      if(FLA_F[i].Active==0) continue;
      for(j=i+1;j<Max;j++)
      {
                if(FLA_F[i].Active==0) continue;
                if(FLA_F[j].Active==0) continue;

                if(Cube_Count(i)==Cube_Count(j) && Difference_Check(i, j)==Difference_No)
                {
                  diff=0;

                  for(n=0;n<Difference_No;n++)
                  {
                        diff_no=0;
                        Xlink_Ok=0;
                        for(m=0;m<In;m++)
                        {
                                if(FLA_F[i].X[m].A!=FLA_F[j].X[m].A)
                                {
                                        if(diff_no==diff)
                                        {
                                        Solution[Solution_No].X[m].A=FLA_F[i].X[m].A|FLA_F[j].X[m].A
                                         Xlink_Ok=1;
                                        }else
                                        {
                                        if(Xlink_Ok==0)
                                        {
                                          Solution[Solution_No].X[m].A=FLA_F[i].X[m].A;
                                        }else
                                        {
                                          Solution[Solution_No].X[m].A=FLA_F[j].X[m].A;
                                        }
                                      }
                                      diff_no++;
                                }else
                                {
                                        if(FLA_F[i].X[m].A == FLA_F[j].X[m].A && diff<=diff_no)
                                        {
                                                Solution[Solution_No].X[m].A=FLA_F[i].X[m].A;
                                        }else
                                        {
                                                Solution[Solution_No].X[m].A=FLA_F[j].X[m].A;
                                        }
                                }
                        }
                        diff++;
                        if(Xlink_Ok==1)
                        {
                                Solution[Solution_No].f=FLA_F[i].f;
                        }else
                        {
                                if(FLA_F[i].f!=FLA_F[j].f)
                                {
                                        Solution[Solution_No].f=FLA_F[i].f|FLA_F[j].f;
                                }else
                                {
                                        Solution[Solution_No].f=FLA_F[i].f;
                                }
                        }
                          Solution[Solution_No].Active=1;
                          Solution_No++;
                  }
                  FLA_F[i].Active=0;
                  FLA_F[j].Active=0;
                  break;
                }
      }
    }


    for(i=0;i<Max;i++)
    {
            if(FLA_F[i].Active==1)
            {
             Solution[Solution_No]=FLA_F[i];
             Solution_No++;
            }
    }
```

```
        Move_F();
        Remove_pair();
    Rtn_Cx++;

}while(Check_Difference(Difference_No)==Difference_No && Old_Cost>Cube_C());

}

int Difference_Check(int F1_No, int F2_No)
{
    int Difference=In+1, i;
        for(i=0;i<In;i++)
        {
        if(FLA_F[F1_No][X[i].A=FLA_F[F2_No][X[i].A)
        {
          Difference--;
        }else
        {
            if(FLA_F[F1_No][X[i].A=Bit_All || FLA_F[F2_No][X[i].A=Bit_All) Difference=0;
        }
        }
    if(FLA_F[F1_No][f==FLA_F[F2_No][f) Difference--;
        return Difference;
```



송 홍 복

1983년 광운대학교 전자통신공학과 졸업(학사)

1985년 인하대학교 대학원 전자공학과(공학석사)

1985년~1991년 2월 동의공업전문대학 전자통신공학과 조교수

1989년~1990년 日本 工大 객원 연구원

1990년 8월 동아대학교 대학원 전자공학과(공학박사)

1991년~현재 동의대학교 전자공학과 조교수

1994년~1995년 POST-DOC (日本 宮崎 大學)

관심분야 : 논리회로설계, 컴퓨터구조(ESOP, 多値論理, MVL-PLA)