筆 者 紹 介

金 錫 源
1964年 5月 4日生
1987年 2月 서울대학교 수학과 학사
1991年 2月 서울대학교 수학과 석사

1991年 2月 ~ 현재 삼성전자(주) ASIC 사업부

주관심 분야 : fault simulation, test generation scan design 등 design for testability

趙 敬 淳
1959年 10月 11日生
1982年 2月 서울대학교 전자공학과 학사
1984年 2月 서울대학교 전자공학과 석사
1988年 8月 Carnegie Mellon University, Electrical and Computer Engineering 박사

1988年 11月 ~ 현재 삼성전자(주) ASIC 사업부

주관심 분야 : ASIC Cell Characterization and Automation,
ASIC Design Automation, Test.

金 光 鉉
1955年 8月 22日生
1978年 2月 서강대학교 전자공학과 학사
1985年 5月 Virginia Tech 전기과 석사
1989年 5月 Virginia Tech 전기과 박사

1978年 3月 ~ 1983年 4月 국방과학연구소 연구원
1989年 6月 ~ 현재 삼성전자(주) ASIC 사업부

주관심 분야 : VLSI CAD, ASIC Design & Test.

特輯

논리 합성 기술

洪 性 濟
浦項工科大學 電子計算學科

Ⅰ. 서론

NAND나 NOR 게이트로 이루어지는 논리 회로의 설계는 매우 중요하다. 왜냐하면 이런 게이트들이 IC 구현에 매우 자주 쓰이기 때문이다.

논리함수를 설계하는데 변수 $x_1 \cdots x_n$과 그의 보수 $\overline{x_1} \cdots \overline{x_n}$을 입력으로 동시에 사용할 수 있으면 doubly-rail 논리라 하며 보수는 입력으로 사용할 수 없으면 single-rail 논리라고 한다. double-rail의 경우, 2-레벨 논리 회로는 minimal sum(또는 minimal product)에 기초해서 쉽게 설계된다. 그러나 흔히, 실제 경우에 있어서는, IC 패키지의 핀 수나 IC 칩 또는 PC 보드의 연결수(결과적으로 면적)를 감소시킬 수 있기 때문에 single-rail이 많이 쓰인다. 이 경우 필요한 게이트 수(결과적으로 전력 소비)는 최대 n 게이트 증가하게 된다(여기서 n은 입력 변수의 갯수를 말함). 불행히도, single-rail 논리 회로에서 최소 게이트를 갖는 회로를 구하는 간단한 설계 방법은 아직 알려져 있지 않다. single-rail의 경우 게이트 수를 최소로하는 설계방법으로는 integer programming 논리 설계방법[1]이 유일한 방법으로 알려져 있다. 이 방법은 복잡하여 수작업(hand-processing)으로는 적합치 않고 컴퓨터를 사용하여야 하는데 계산시간이 너무 많이 걸려 게이트수가 10개를 넘으면 현실적으로 사용할 수 없는 단점이 있다. 그러므로, 본 원고 Ⅱ장에서는 single-rail 논리에서 NAND(또는 NOR) 회로 설계를 위해, 손쉽게 설계할 수있는 map-factoring 방법[2]을 소개한다. 이 방법을 쓰면 대체로 만족할 만한 회로를 얻을 수있다. 이 방법은 double-rail 논리의 다단계 NAND 회로 설계로도 쉽게 확장될 수있다.

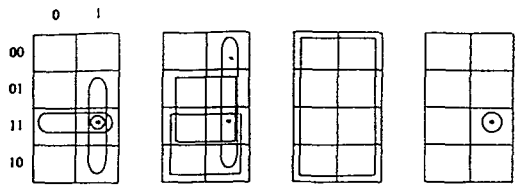
3-레벨 회로는 single-rail 논리에서 최단의 게이트 지연 시간을 갖는다. 그러나, minimal sum(또는 minimal product)에 기초한, double-rail 논리에서 2-레벨의 최소의 AND/OR나 NAND 회로를 얻는 Karnaugh map방법 같은 간단한 방법이 3-레벨의 경우에는 없다. 3-레벨 single-rail의 경우 게이트수를 최소로 하는 설계 방법으로는 Gimpel방법[3]이 있으나 변수가 많아지면 역시 계산시간이 많이 걸려 비현실적인 단점이 있다. 본 원고 Ⅲ장에서는 게이트 수는 많아지지만 주어진 함수로부터 간단히 3-레벨 single-rail을 설계하는 Universal Nor회로 방법을 소개하고 Ⅳ장에서는 게이트수의 최소화는 보장하지 못하지만 때때로 좋은 결과를 내는 선험적(heuristic) 설계 방법을 소개한다.

Ⅱ. Map-factoring 방법

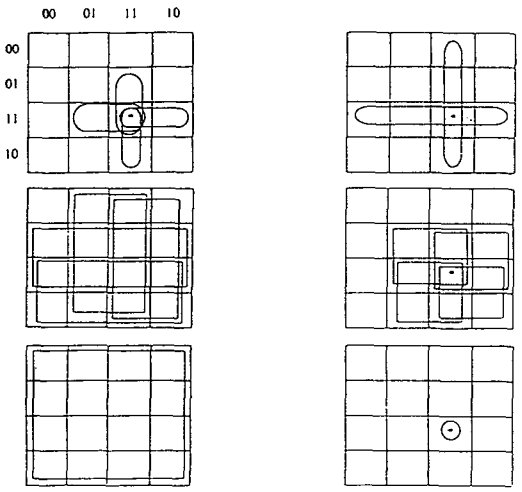
기존의 tabular[4], algebraic[5,6,7], 또는 graphical[8,9] 등, single-rail 논리의 NAND(NOR) 회로를 설계하는 여러 가지 방법들이 있으나, 회로의 minimality(즉 게이트 수 또는 연결선을 최소로 하는)를 보장하는 간단한 설계 방법은 없다. 그러나, map-factoring 방법을 쓰면 손쉽게 어느 정도 만족할 만한 회로를 구할 수있다. 이것은 Karnaugh map에 기초하는데, 2-레벨에서의 AND/OR 회로 설계 경우처럼 minimality를 보장하지는 못한다. Karnaugh map의 그림의 특성에 기초한 설계자의 직관에 의존하여, 여러 번의 시행착오를 거쳐, 최소

한의 상당히 좋은 회로를 얻을 수 있다. 사실 때때로 최적의 회로가 구해지기도 한다. 단점은 이 방법이 배우기 힘들다는 것이다. 해결책은 많은 예제를 통한 연습으로 감(feeling)을 얻는 것이다.

먼저, Karnaugh map에서 permissible-loop를 정의하자. permissible-loop란 prime-implicant와 마찬가지로 개의 cell로 이루어진 직사각형으로, 특정 cell 즉 모든 입력 값이 '1'인 cell을 포함한다. 다시 말해, permissible-loop는 그림 1의 *로 표시된 특정 cell을 반드시 포함하여야 한다. 그러나 prime-implicant loop와 다른 점은 loop의 구성이 0-cell, 1-cell, 또는 이 둘의 혼합일 수있다.



(a) 변수가 3개인 경우



(b) 변수가 4개인 경우

그림 1. Permissible loops

기술하는 다음의 방법에서 회로의 입력으로는 x_i형태만이 가능하다고 가정한다. ($\bar{x}_i$는 불가능)

Map-Factoring 방법: Single-Rail의 회로 설계

Step 1 : 먼저, 첫번째 permissible-loop를 구한다. 이 loop에 해당하는 NAND 게이트를 그리고, 이 게이트에 loop가 나타내는 product에 있는 모든 literal을 연결한다. 이후, loop를 검게 칠한다.

Step 2 : permissible-loop를 구한 후, 이 loop에 해당하는 NAND 게이트를 그린다. 이 새로운 게이트에 loop가 나타내는 product의 literal을 연결한다. (이제까지는 Step 1과 동일하다.)
다음 필요하면, 새로이 만들어진 NAND 게이트에 이미 구해진 게이트의 출력을 연결한다. (이미 구한 게이트의 출력을 새로 구한 게이트의 입력으로 해야할 지, 말아야할 지를 지시해주는 원칙은 없다.)

1. 이미 구한 게이트를 새로운 게이트에 연결치 않겠다면, 새 permissible-loop는 전부 칠해진다.
2. 이미 구한 게이트 중 일부를 새로운 게이트에 연결하겠다만, 단지, 이전 게이트의 칠해진 loop가 새 게이트의 permissible-loop를 가로막는(겹치거나, 포함되는) 것들만이 새 게이트에 연결될 수있다. 새 permissible-loop에서, 새 게이트에 연결된 이전 게이트의 칠해진 loop를 제외한 부분을 한데 묶고 이를 칠한다. (칠해지는 loop를 새로운 게이트의 출력과 '연관'되었다고 한다.)

Step 3 : 다음 조건이 만족될 때까지 Step 2를 반복한다.

종결 조건 : 새 permissible-loop와 이에 해당되는 게이트가 구해지면, 전체 map의 모든 0-cell이 이 게이트의 출력과 연관된 검게 칠해진 loop를 구성한다. (즉, map 전체의 모든 0-cell이 새로운 permissible-loop에 모두 포함되고, 새 게이트의 출력과 연관된 칠해진 loop는 오직 이런 0-cell들만을 포함한다.)

map이 d-cell(don't care 조건에 관계된 cell)을 포함하더라도 map-factoring 방법은 Step 3의 종결 조건에서 필요에 따라 d-cell을 0-cell로 또는 1-cell로 적절히 해석함으로써 쉽게 사용할 수 있다.

예 2.1 : $f(x,y,z) = \Sigma(0,2,3,4,5,6,7)$을 map-factoring 방법으로 설계해 보자. 첫번째 permissible-loop를 선택한다. (그림 2(a)) 이를 선택한 특별한 이유는 없다. 다른 것을 선택할 수도 있다. 이 선택된 loop를 1번으로 번호를 매긴 후, 검게 칠한다. 다음, NAND 게이트 1이 그려진다. loop가 x를 나타내므로 x를 이 게이트에 연결한다.

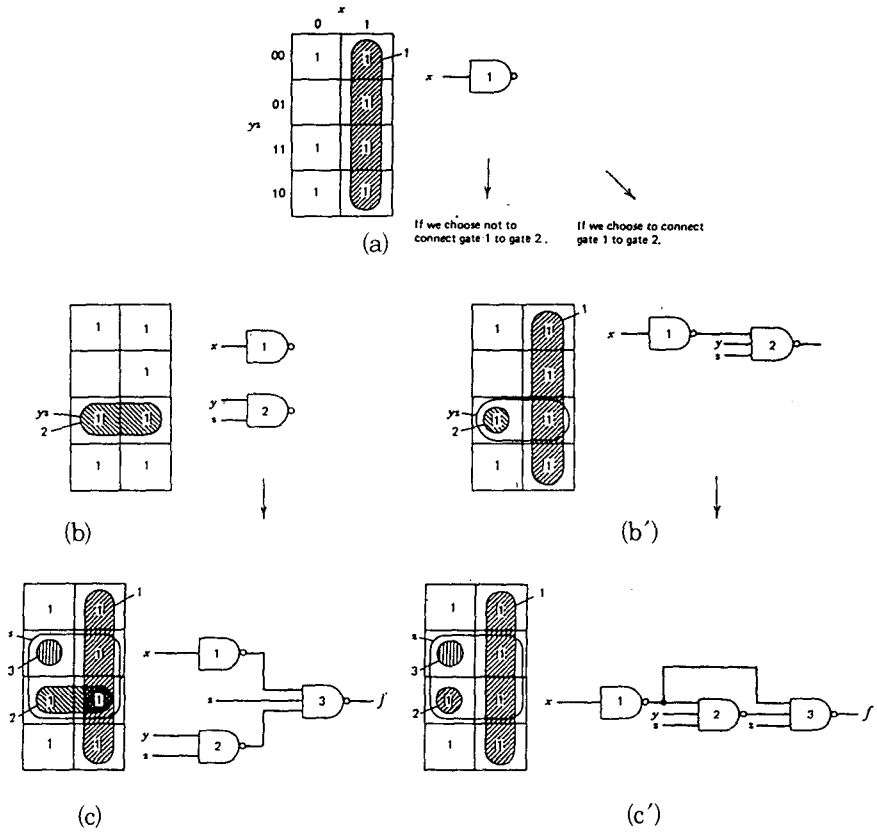


그림 2. $f(x,y,z) = \Sigma(0,2,3,4,5,6,7)$의 설계

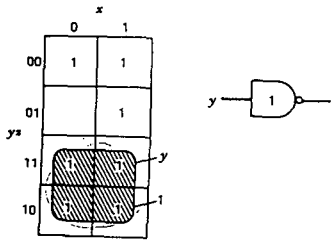
그림 2(b)같이, 새 permissible-loop를 선택한다. (이를 선택한 특별한 이유는 없다.) 이 loop가 product yz를 나타내므로 y와 z를 새 게이트에 연결하고 이를 2번으로 한다. 만일 게이트 1을 게이트 2에 연결시키지 않겠다면, 그림 2(b)의 회로를 얻는다. 게이트 1의 검게 칠한 loop는 새 loop와는 상관없다. 새로운 permissible-loop는 전부 검게 칠해지고, 2번으로 매겨진다. (그림 2(b))

만일 게이트 1을 게이트 2에 연결하면 그림 2(b')의 회로를 얻게 된다. 게이트 1의 칠해진 loop에 의해 중복되지 않은 새로운 permissible-loop의 부분이 검게 칠해지고 이를 2번으로 한다.

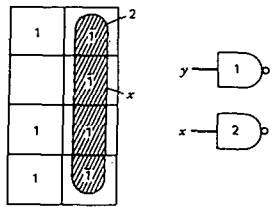
새 permissible-loop를 그림 2(c)와 같이 취하고, 게이트 1, 게이트 2를 이 새로운 게이트 3에 연결코자 한다면(이전의 칠해진 loop 1, 2는 이 새로운 permissible-loop를 가로막는 것으로 해석할 수있다.) 종결 조건이 만족된다. 다시 말해, 전체 map의 모든 0-cell이 게이트 3의 출력과 연관되어 칠해진 loop안에 들어 있게 된다.

그림 2(c)와 다른 예로서, 그림 2(b')부터 시작해 보자. 그림 2(c')처럼 permissible-loop를 취하고, 이전 게이트 1, 2의 loop를 이 새 permissible-loop를 가로막는 것으로 하면 Step 3의 종결 조건이 만족되고, 그림 2(c)와 다른 그림 2(c')의 회로를 얻는다.

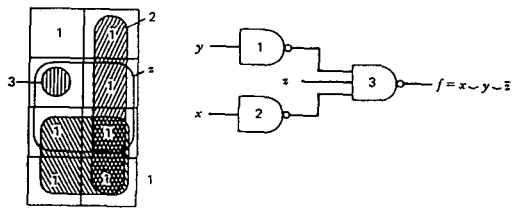
첫번째 permissible-loop를 그림 2(a)와 달리 그림 3(a)처럼 취했다면, 그림 3(b)로 진행됐을 것이고 그림 3(c)의 회로를 얻었을 것이다. 물론 그림 3(b), 그림 3(a)와는 또 달리 취할 수도 있다. 허나, 이처럼 모든 경우를 다 조사해보는 것은 많은 시간이 걸리므로, 몇 번의 시도로써 만족해야 한다. 적당한 것을 선택할 수있는 감을 얻기 위해 여러 번의 시도를 필요로 한다. 이후 적당한 것을 얻게 된다. 위의 예의 경우인 $f=x\vee y\vee z$의 논리회로인 그림 3의 회로는 최적 회로(integer programming에 의해 증명된 게이트 수와, 부목표인 게이트간의 연결 수를 최소로 하는 회로)로 알려져 있다.



(a)



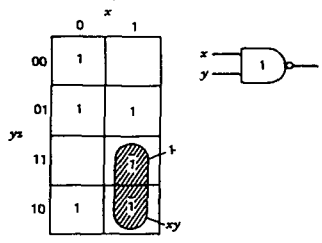
(b)



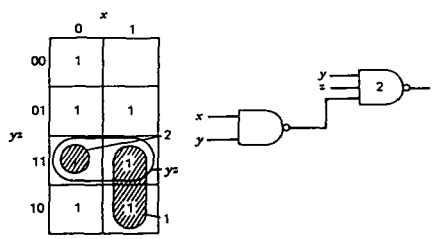
(c)

그림 3. 그림 2의 변형

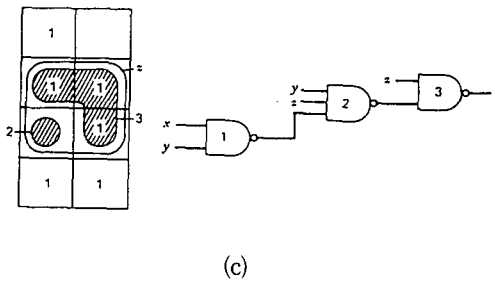
예 2.2 : $f(x,y,z) = \Sigma(0,1,2,5,6,7)$을 설계해 보자.



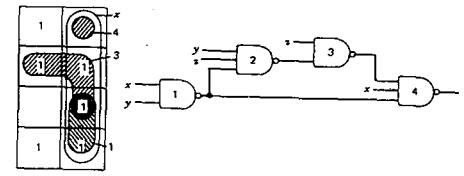
(a)



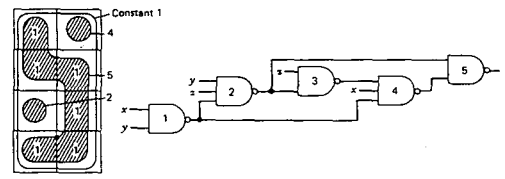
(b)



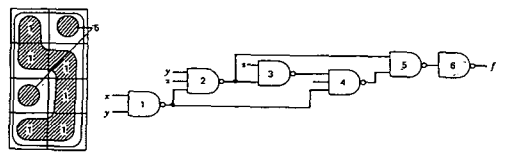
(c)



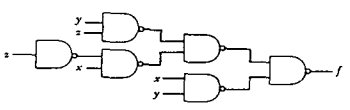
(d)



(e)



(f)



(g)

그림 4. $f = \Sigma(0,1,2,5,6,7)$의 설계

첫번째 permissible-loop로써 그림 4(a)처럼 취한다. 이 loop를 1번으로 하고 칠한다. 이 loop는 xy product를 나타내므로, x와 y를 이 게이트에 연결시킨다.

두번째 permissible-loop는 그림 4(b)처럼 취한

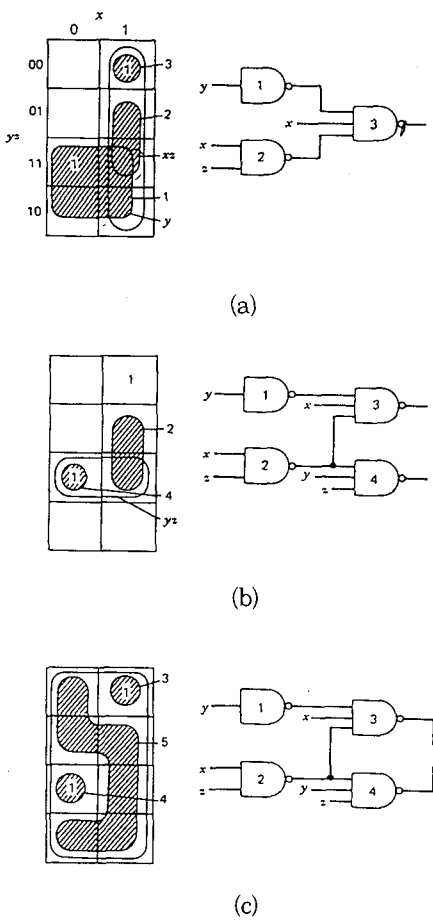
다. 이 loop는 product yz를 나타내므로 y와 z가 이 새로운 게이트에 연결되고, 2번으로 매긴다. 이 permissible-loop는 칠해진 loop 1과 겹치는 부분이 있으므로 게이트 1의 출력을 게이트 2에 연결한다. (연결 안하는 쪽을 택할 수도 있다.)

Step 2를 반복함으로써 다음의 permissible-loop z가 그림 4(c)처럼 구해진다. 이에 해당하는 게이트 3이 그려진다. 단지 게이트 2만을 게이트 3에 연결시켜 보자. (이 결정이 좋은 건지, 나쁜 건지 모른다. 게이트 1도 연결하든가, 게이트 1은 연결하고 게이트 2는 연결치 않던가, 또는 게이트 1, 2 둘다 연결치 않던가 한다면, 각각 서로 다른 회로가 구해진다.) 다음의 Step 2를 반복할 때, x를 나타내는 새로운 permissible-loop를 그린다. (그림 4(d)) 게이트 1의 loop이 새로운 loop에 포함된다. 게이트 3의 loop는 이 새 loop와 겹치는 부분이 있다. 그러나 게이트 2를 게이트 4에 연결하지 않는다. 왜냐하면 게이트 2의 loop는 이 새 loop와 겹치지도 포함되지도 않기 때문이다. 게이트 1, 3을 게이트 4에 연결한다. Step 2의 반복 때, 새 permissible-loop이 그려진다. 상수 1의 입력이 그림 4(e)에 나타나 있다. 해당되는 게이트 5가 그려진다. 상수 1의 입력은 게이트 5에 연결되지 않는다. 이유는 상수 1의 입력이 되건 안되건 상관없이 게이트 5의 입-출력 관계는 변하지 않기 때문이다. 고로, 게이트 2와 4만이 연결된다. 아직 종결 조건이 만족되지 않았다. (만일, 그림 4(e)에서, 모든 0과 1이 서로 뒤바뀐다면 종결 조건이 만족된다.) 또 다른 상수 1이 그림 4(f)에 나타나 있다. 만일 게이트 5를 6에 연결하고자 한다면, 종결 조건이 만족되게 된다. 이로써 주어진 함수 f의 회로를 그림 4(f)처럼 얻었다.

그림 4(g)는 최소의 회로를 나타낸다. 이는 주목표인 게이트 수, 부목표인 연결 수에 있어 최소치이다. (integer programming에 의해 증명됨) 이 회로는 연결 수에 있어서는 그림 4(f)보다 2개 적고, 게이트 수는 똑같다.

Map-factoring 방법으로 설계된 다른 예들이 그림 5에 나타나 있다. 그림 5(a)는 Step 2가 몇 번 반복된 후의 결과를 나타낸다.

4-변수일 경우의 예가 그림 6에 있다. 그림 6(d)에



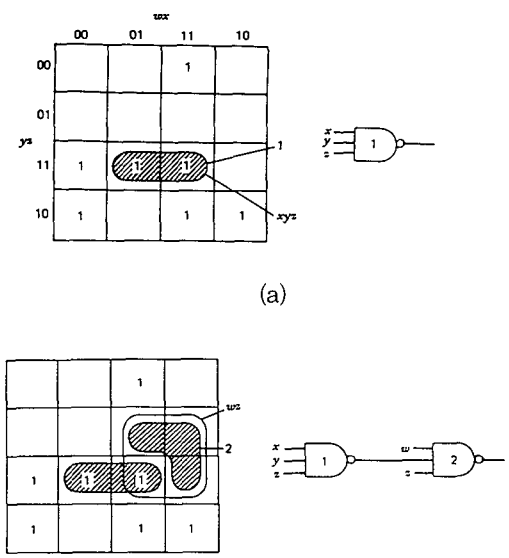


그림 5. $f=\Sigma(3,4)$의 설계

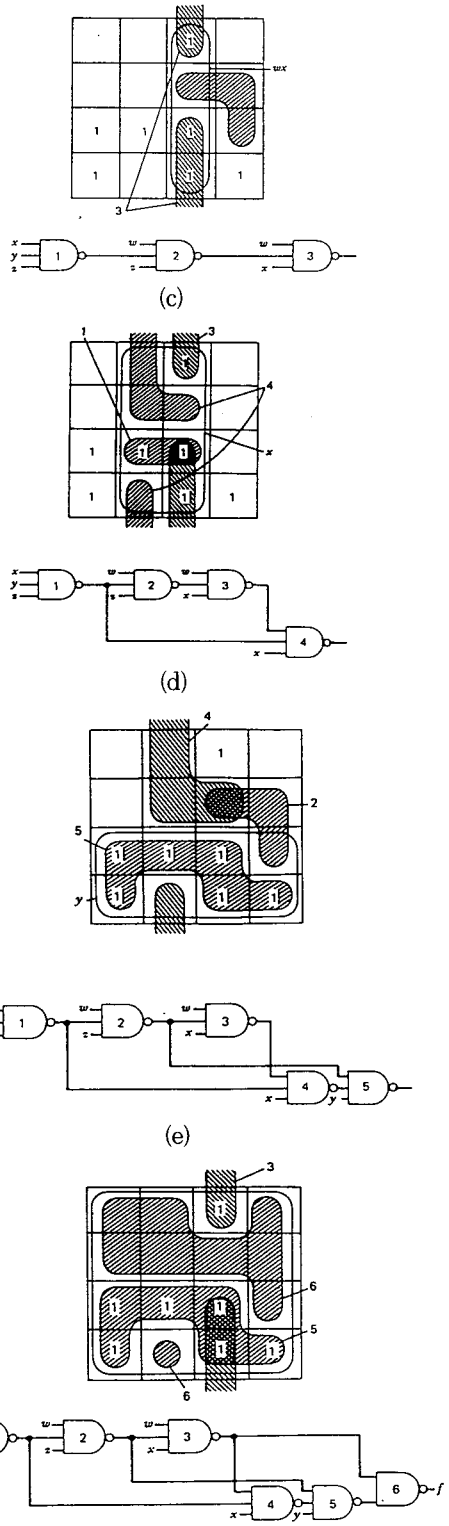


그림 6. $f=\Sigma(2,3,7,10,12,14,15)$의 설계

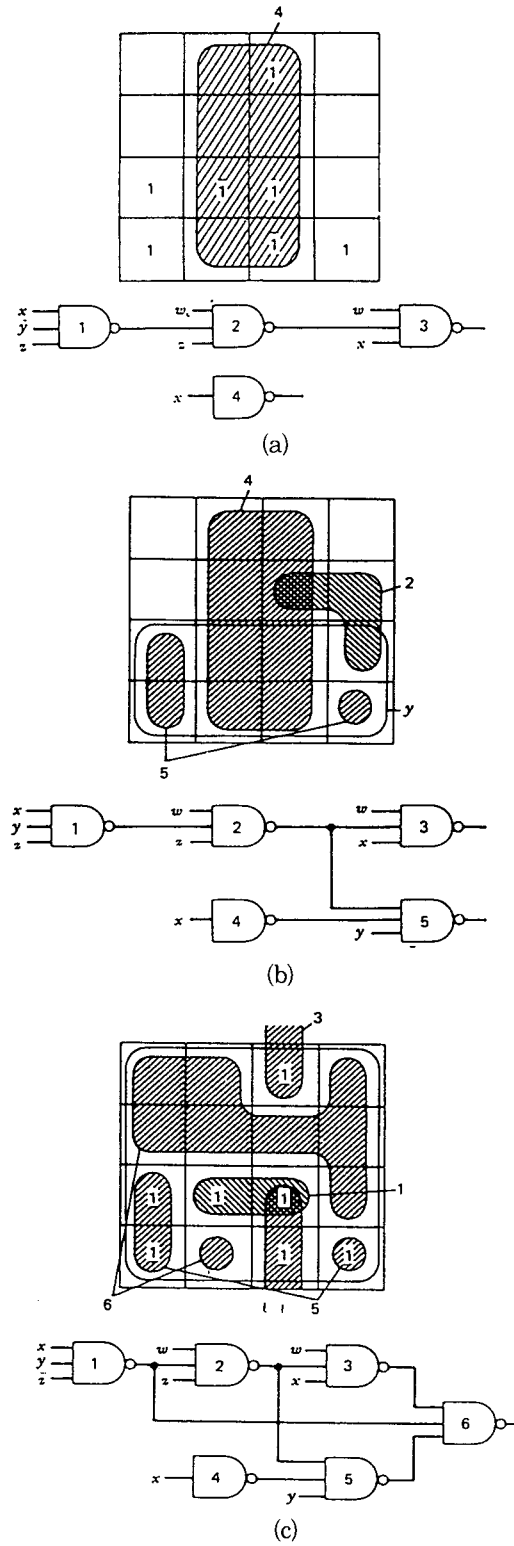


그림 7. 그림 6의 변형

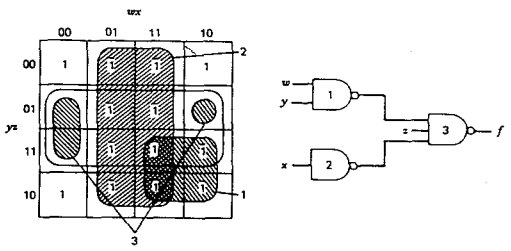
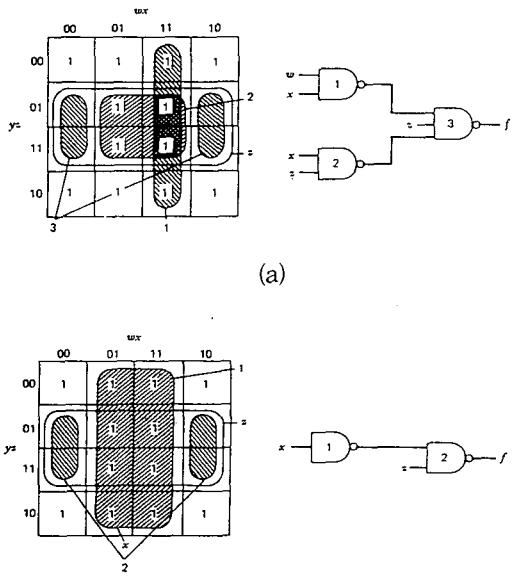


그림 8. $f=\Sigma(0,2,4,5,6,7,8,10,11,12,13,14,15)$의 설계



(a)

(b)

그림 9. $f=\Sigma(0,2,4,5,6,7,8,10,11,12,13,14,15)$의 설계

서 게이트 1, 3을 게이트 4에 연결치 않겠다면 그림 7로 진행하여 다른 회로가 구해진다. 그림 8, 그림 9에 또다른 예가 있다. 그림 9(a)와 (b)는 같은 함수를 나타내지만 서로 다른 loop들로 이루어진 경우이다.

Ⅲ. Universal NOR 회로 방법

n-변수 함수, f가 주어질 때 S, M을 다음과 같이 정하면

$S=\{m_i \mid m_i$는 $minterm$, $i=0,1,2,\cdots,2^n-1\}$

$M=\{m_i \mid m_i \in S,\ f(m_i)=1\}$

함수 f는 $\sum_{m_i \in M} m_i$ 또는 $\overline{\sum_{m_i \in S-M} m_i}$로 나타낼 수 있다. 만일 NOR 회로가 존재해서 회로 내의 각 게이트가 S-M의 minterm을 나타내는 함수라면, 이 게이트들의 출력을 다른 NOR 게이트의 입력으로 하여 이 새로운 NOR 게이트의 출력이 함수 f를 나타내게 할 수 있다. 이 때 함수를 구현하는데 불필요한 게이트들은 제거될 수있다.

universal NOR 회로 방법은, 회로의 입력으로 uncomplement된 외부 변수를 갖는, 2^n개의 게이트로 이루어진 NOR 회로를 만들 수있다. 회로 내의 각 게이트는 S의 원소인 한 함수(minterm)을 나타낸다. 고로 임의의 함수 f는 위의 방식으로 최대 2^n개의 게이트를 써서 나타낼 수있다. 그림 10은 3-변수의 universal-NOR 회로와 각 게이트로 구현되는 함수가 나타나 있다. 여기서 f_{gi}는 게이트 i의 출력으로 구현되는 논리함수를 나타낸다.

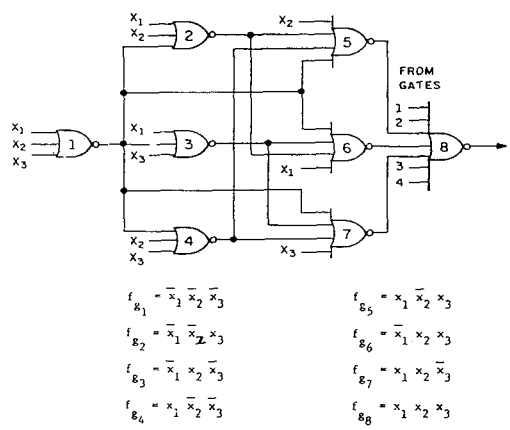


그림 10. 3-변수 universal NOR 회로

universal-NOR 회로를 생성해내는 알고리즘을 아래에 소개한다. 여기서 n을 외부 변수의 갯수, 게이트 I의 레벨 값 l을 게이트 I와 출력 게이트(게이트 레벨 1)를 연결하는 모든 경로상의 게이트의 레벨 중 최대라 하자. n-변수일 경우 $1 \le l \le n+1$이다.

Universal-NOR 회로 생성 알고리듬

Step 1 : 모든 외부 변수를 가장 높은 레벨(n+1째 레벨)의 게이트에 연결한다.

Step 2 : 각 레벨 l에 대해($1<l<n+1$), l-1개 외부 변수의 모든 가능한 조합을 구한다. 각 l-1개 외부변수들 각각의 l 레벨의 게이트에 연결한다.

Step 3 : 각 l($1<l<n+1$)레벨 게이트 I에 게이트 I

의 외부 변수를 포함하는 모든 상위 레벨 게이트의 출력을 연결한다.

Step 4 : 출력 게이트에 대해 모든 상위 레벨 게이트의 출력을 입력으로 연결한다.

이 알고리듬의 정당함은 map-factoring 방법을 통해 쉽게 증명된다. 3-변수 Karnaugh map과 map-factoring 방법에 의해 유도된, 대응되는 loop를 그림 11에 나타냈다. 그림 11의 칠해진 원들은 그림 10의 각 minterm에 해당된다.

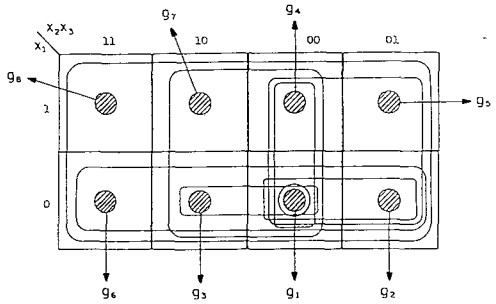


그림 11. 3-변수 Karnaugh map

위의 알고리듬에서 다음의 성질을 발견할 수있다.

(a) 레벨 $l(1\le l\le n+1)$에서의 게이트 수 I_l^n

$$I_l^n=\binom{n}{l-1}=\frac{n!}{(l-1)!(n-l+1)!}$$

(b) universal 회로의 총 게이트 수

$$\sum_{l=1}^{n+1} I_l^n=\sum_{l=1}^{n+1}\binom{n}{l-1}=\sum_{l=1}^{n+1}\frac{n!}{(l-1)!(n-l+1)!}=2^n$$

다음 예는 universal 회로를 이용하여 어떻게 함수를 구현하는가를 보여준다.

예 3.1 : $f(x_1, x_2, x_3)=\Sigma(0,2,3,4,6,7)$를 고려

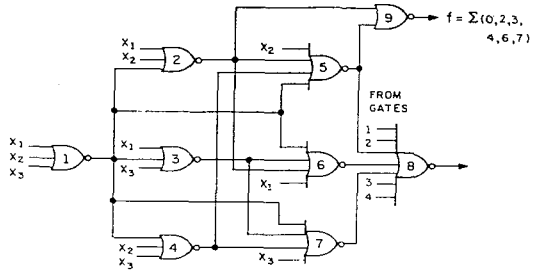


그림 12. $f=\Sigma(0,2,3,4,6,7)$의 설계

하자. S-M={1,5}이므로 게이트 2, 5의 출력을 다른 게이트(게이트 9)의 입력으로 할 수 있다. 그 결과가 그림 12에 있다. 그림 12에서 많은 불필요한 게이트들의 연결이 있음을 쉽게 알 수 있다. 그 예로써 게이트 3, 6, 7, 8이 제거될 수 있다.

Ⅳ. 3-레벨 회로 방법

Ⅲ장에서 언급했듯이, 주어진 논리함수 f는 $\overline{\sum_{m_i\in S-M} m_i}$로 나타낼 수 있다. S는 모든 minterm의 집합이며, M은 1-벡터인 모든 minterm의 집합이다. (벡터 $\vec{x}$는 $f(\vec{x})=1$일 때 1-벡터, $f(\vec{x})=0$ 일 때 0-벡터라 한다.) 3-레벨 회로 방법은 위의 방법에 기초한다. NOR 회로를 설계하는데는, uncomplement된 외부 변수만을 그 입력으로 하며 이 방법은 최대 3-레벨로 제한한다. 알고리듬을 소개하기 앞서, 다음을 정의하자.

V_n: n차원 이진 벡터들의 집합(n: 외부 변수의 갯수)

정의 4.1 : 두 벡터 $\vec{x}_1=(x_{11},\cdots,x_{1n}), \vec{x}_2=(x_{21},\cdots,x_{2n})$ 가 주어지면,

1. 모든 i에서 $x_{1i}\le x_{2i}$이고, 최소 1개 이상의 i에 대해 $x_{1i}<x_{2i}$일 때, $\vec{x}_1<\vec{x}_2$.
2. 모든 i에서 $x_{1i}=x_{2i}$일 때, $\vec{x}_1=\vec{x}_2$.
3. $\vec{x}_1<\vec{x}_2$, $\vec{x}_1>\vec{x}_2$, $\vec{x}_1=\vec{x}_2$중 어느 것에도 속하지 않을 때, $\vec{x}_1$와 $\vec{x}_2$는 '비교 불능'하다고 한다.

정의 4.2 : 입력 벡터 $\vec{x}$의 무게$W(\vec{x})$: $\vec{x}$안의 '1'의 갯수.

두 벡터간의 거리: $d(\vec{x}_1,\vec{x}_2)=W(\vec{x}_1\oplus\vec{x}_2)$,단, $\vec{x}_1\oplus\vec{x}_2=(\vec{x}_{11}\oplus\vec{x}_{21},\cdots,\vec{x}_{1n}\oplus\vec{x}_{2n})$.

정의 4.3 : Z-집합은 다음 조건을 만족하는 입력 벡터들의 최대 집합이다.

1. 각 Z-집합에서, 모든 벡터는 0-벡터여야 하며, 다른 0-벡터보다 큰 단 하나의0-벡터가 존재한다.
2. 0-벡터 $\vec{x}_1$, $\vec{x}_2$가 $\vec{x}_1<\vec{x}_2$이고, $\vec{x}_1<\vec{x}_3<\vec{x}_2$인 1-벡터 $\vec{x}_3$가 존재한다면, $\vec{x}_1$, $\vec{x}_2$ 가 같은 Z-집합에 포함될 수 없다. 이 조건에 맞지 않으면 $\vec{x}_1$와 $\vec{x}_2$는 동일 Z-집합에 포함된다.
3. 각 Z-집합에서, 다른 Z-집합에는 포함되지 않는 벡터 $\vec{x}$가 최소한 1개씩은 있다.

Z-집합 내에서, $\vec{x}$가 가장 큰 벡터일 때, $\vec{x}^*$을 Z-집합 $Z(\vec{x}^*)$의 'root'라 한다.

정의 4.4: Y-집합은 다음 조건을 만족하는 입력 벡터들의 최대 집합이다.

1. 각 Y-집합에서, 모든 벡터는 1-벡터여야 하며, 다른 1-벡터보다 큰 단 하나의 1-벡터가 존재한다.

2. 1-벡터 $\vec{x_1}$, $\vec{x_2}$가 $\vec{x_1} < \vec{x_2}$이고, $\vec{x_1} < \vec{x_3} < \vec{x_2}$인 0-벡터 $\vec{x_3}$가 존재한다면, $\vec{x_1}$, $\vec{x_2}$가 같은 Y-집합에 포함될 수 없다. 이 조건에 맞지 않으면 $\vec{x_1}$와 $\vec{x_2}$는 동일 Y-집합에 포함된다.

3. 각 Y-집합에서, 다른 Y-집합에는 포함되지 않는 벡터 $\vec{x}$가 최소한 1개씩은 있다.

Y-집합 내에서, 가 가장 큰 벡터일 때, $\vec{x^*}$을 Y-집합 $Y(\vec{x^*})$의 'root'라 한다.

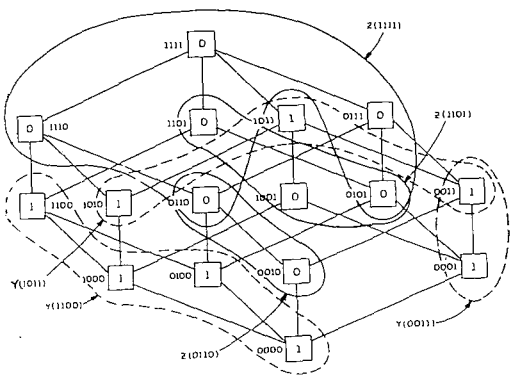


그림 13. $f=\Sigma(0,1,3,4,8,10,11,12)$의 설계

예 4.1 : 그림 13에서, 4-변수 함수 $f=\sum(2,5,6,7,9,13,14,15)$가 주어졌을 때, 정의 4.3, 4.4에 의해 다음처럼 Z-집합과 Y-집합이 각각 3개씩 3개씩 구해진다.

Z(1111) = {(1111), (1110), (0110), (1101), (0101), (0111)} Z(1101) = {(1101), (1001), (0101)}

Z(0110) = {(0110), (0010)}

Y(1011) = {(1011), (1010), (0011), (0111)}

Y(1100) = {(1100), (1000), (0100), (0000)}

Y(0011) = {(0011), (0001)}

위의 Y-집합의 어떤 부분 집합도 Y-집합이 아니다. 또한 위의 Z-집합의 어떤 부분 집합도 Z-집합이 아니다. {(0000), (0001)}은 Y-집합이 아니다. 왜냐하면 (0000)∈Y(1100)이며 (0001)∈Y(0011)이므로 정의 4.4의 조건 3을 위반한다.

3-레벨 초기 회로 방법에서(출력 게이트는 첫째 레벨의 게이트), 각 Y-집합은 셋째 레벨의 게이트에 대응될 수있다. 어떤 셋째 레벨의 게이트를 둘째 레벨의 게이트에 연결해야 할 지를 알기 위해, Y-집합과 Z-집합간의 인접 관계를 아래에 소개한다.

정의 4.5 : 다음 조건 1과 2를 만족하면 $Y(\vec{x_1})$는 $Z(\vec{x_2})$에 '인접'한다고 한다.

1. $d(\vec{x_3}, \vec{x_4})=1$이고, $\vec{x_3} < \vec{x_4}$이고, $\vec{x_3} \in Y(\vec{x_1})$, $\vec{x_4} \in Y(\vec{x_2})$인 $\vec{x_3}$, $\vec{x_4}$가 존재한다.

2. $\vec{x_3} \in Y(\vec{x_1})$, $\vec{x_4} \in Y(\vec{x_2})$, 인 모든 $\vec{x_3}$와 $\vec{x_4}$에 대하여 $\vec{x_3} > \vec{x_4}$ 가 아니다.

예 4.2 : 그림 13에서, 먼저 Z-집합 Z(1111)과 Y-집합 Y(1100)을 고려하자. 1-벡터 (1100)∈Y(1100), 0-벡터 (1110)∈Z(1111)이고, (1100)<(1110), d(1100, 1110)=1 이므로 정의 4.5의 조건 1을 만족한다. 또 Z(1111)의 어느 벡터도 Y(1100)의 어느 벡터보다 작지 않으므로 조건 2도 만족한다. 그러므로 Y-집합 Y(1100)은 Z집합 Z(1111)에 인접한다. 같은 식으로 Y(1011), Y(0011)도 Z(1111)에 인접함을 알 수있다. 다음은 이 예의 인접 관계를 요약한 것이다.

Y(0011)은 Z(1111), Z(1101)에 인접한다.

Y(1100)은 Z(1111), Z(1101), Z(0110)에 인접한다.

Y(1011)은 Z(1111)에 인접한다.

각 Y-집합(또는 Z-집합)이 어떤 Z-집합(또는 Y-집합)엔가 꼭 인접할 필요는 없음을 주목하라. 다음이 그 예이다.

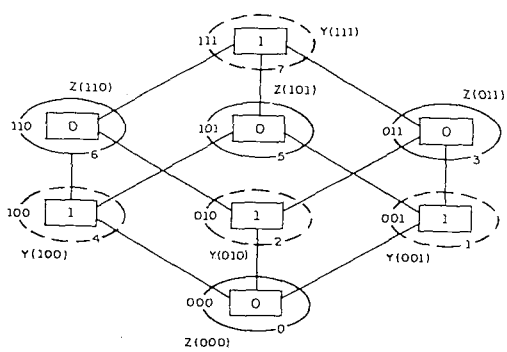


그림 14. 3-변수 parity함수

예 4.3 : 그림 14에서, 주어진 함수는 3-변수 parity 함수이다. 여기엔 4개의 Y-집합과 4개의 Z-집합이 있다. 분명히, Z(000)은 어느 Y-집합에도 인접하지 않으며, Y(111)도 어느 Z-집합에도 인접하지 않는다.

정의 4.6 : 입력 벡터 $\vec{x}$ 의 β -집합은 $\vec{x}$ 에 나타나는 uncomplement된 외부 변수의 집합이다.

예 4.4 : 1100의 β -집합은 $\{x_3, x_4\}$ 이고, 1111의 β -집합은 공집합이다.

3-레벨 NOR 회로 생성 알고리듬

Step 1 : 주어진 함수의 Z, Y-집합을 찾고 인접 관계를 계산한다.

Step 2 : Z-집합에 인접한 각 $Y(\vec{x})$에 대해 셋째 레벨의 게이트를 구성한다. 이 게이트의 입력은 $\vec{x}$의 β -집합의 원소들(외부 변수들)이다.

Step 3 : 각 $Z(\vec{x})$에 대해 둘째 레벨의 게이트를 구성한다. 이 게이트의 입력은 $\vec{x}$의 β -집합의 원소들(외부 변수들)과, $Z(\vec{x})$와 인접한 Y-집합에 해당하는 셋째 레벨 게이트들의 출력이다. 만일 $Y(\vec{x_1})$과 $Y(\vec{x_2})$가 $Z(\vec{x})$에 인접하고, $\vec{x_1} > \vec{x_2}$이면 $Y(\vec{x_2})$에 해당하는 셋째 레벨 게이트는 $Z(\vec{x})$에 해당하는 둘째 레벨의 게이트에 연결될 필요가 없다.

Step 4 : 둘째 레벨의 모든 게이트들의 출력을 첫째 레벨의 게이트(출력 게이트)의 입력으로 한다.

예 4.5 : 위의 알고리듬에 따라, 예 4.1의 3-레벨 회로가 그림 15처럼 구해진다.

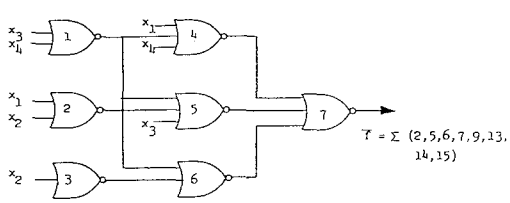
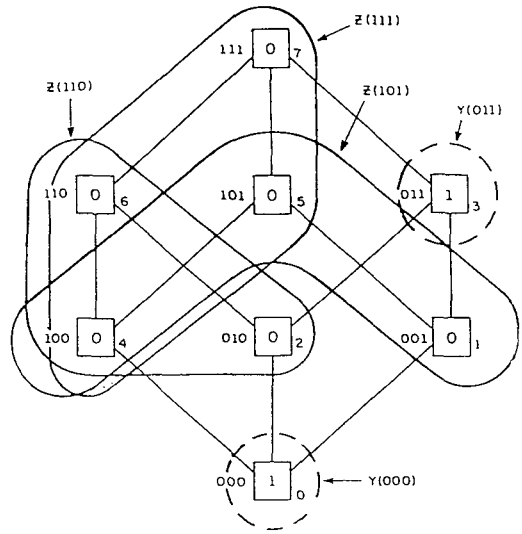


그림 15. $f=\Sigma(0,1,3,4,8,10,11,12)$의 설계

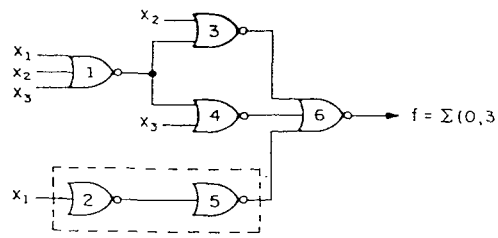
그림 15에서, 게이트 1, 2, 3은 각각 Y-집합 Y(1100), Y(0011), Y(1011)에 대응된다. 그리고 게이트 4, 5, 6은 각각 Z-집합 Z(0110), Z(1101), Z(1111)에 대응된다. Y(1011)과 Y(0011)이 Z(1111)에 인접하고 (0011)<(1011)이므로, Step 3에 따라 게이트 2(즉, Y(0011)에 대응하는)의 출력이 게이트 6(즉, Z(1111)에 대응하는)에 연결되지 않는다.

위의 알고리듬에서, 둘째 레벨의 각 게이트의 출력은 대응되는 $Z(\vec{x})$의 0-벡터에 대응하는 minterm이다. 고로, 첫째 레벨의 입력은 f의 모든 0-벡터에 대응하는 minterm들의 disjunction이다.

위의 알고리듬에서 나타나는 재미있는 현상은, 출력 게이트의 입력으로 외부변수를 취하는 회로를 구할 수 없다는 점이다. 다음은 그 예이다.



(a)



(b)

그림 16. $f=(x_1, x_2, x_3)=\Sigma(0,3)$의 설계

예 4.6 : 3-변수 함수 $f=(x_1, x_2, x_3)=\Sigma(0,3)$가 그림 16(a)와 (b)처럼 주어질 때, 알고리듬을 적용하여 3-레벨 회로가 구해진다. 보다시피, 게이트 2, 5는 불필요하며 제거될 수있다. 즉 x_1을 직접 게이트 6에 연결할 수있다.

3-레벨 회로를 생성하는 위의 알고리듬은 게이트 수와 연결 수가 최소임을 보장하지는 못한다. 그러나 알고리듬이 간단하기 때문에, 초기 회로를 (이후, triangular 조건 같은 것을 이용하는 여러 변환 테크닉을 사용하여 더욱 간단히 할 수있는) 찾는데 있어 유용하다.

V. 결 론

본 원고에서는 게이트의 수의 최소화는 보장하지 못하지만 손으로 쉽게 논리 설계를 할 수 있는 몇 개의 방법을 소개하였다. 2-레벨의 경우는 게이트 수를 최소화하는 알고리듬이 존재하나 3-레벨 이상에서는 최적 알고리듬이 integer programming 방법외에는 현재까지 알려진 것이 없다. integer programming은 계산시간이 너무 많이 걸려 gate수가 10개를 넘는 회로의 설계에는 사용할 수 없는 비현실적인 알고리듬이다. 3-레벨 이상의 논리회로 설계의 좋은 알고리듬이 부재한 이유는 좋은 factoring 알고리듬이 개발되지 못한데 기인한다. 본 원고에서 소개한 map-factoring 방법은 Karnaugh map을 사용하여 factoring하는데 설계 알고리듬이라기 보다는 설계지침에 가까와 설계자의 경험에 크게 의존하고 있다. 대수적인 방법을 사용하여 factoring하는 알고리듬을 최근 Synopsis사에서 상용으로 개발하였으나 아직 숙련된 설계자의 직관에 의한 설계에는 못 미치는 것 같다. 3-레벨 이상의 논리 설계는 앞으로 실제로 유용한 알고리듬의 개발이 시급한 실정이다.

參 考 文 獻

[1] S. Muroga, "Logic Design of Optimal digital networks by integer programming", Capter 5 in Advanced in Information System Science, vol. 3, edited by J. T. tou, Plenum Press, pp. 283-348, 1970.

[2] G. A. Maley and J. Earle, The Logic Design of Transistor Digital Computers, Prentice-Hall, 1963.

[3] J. F. Gimpel, "The minimization of TANT networks", *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 18-38, Feb. 1967.

[4] E. M. McClusky, Jr., "Minimization of Boolean funcions", *Bell sys. Tech. J.*, vol. 35, no. 6, pp. 1417-1444, Nov. 1957.

[5] S. Muroga, Logic Design and switching Theory, New York: Wiley Interscience, 1979.

[6] P. Tison, "Generalization of consensus theory and application to the minimization of Boolean functions", *IEEE Transactions on Computers*, vol. EC-16, pp. 446-456, Aug. 1967.

[7] S. J. Hong and S. Muroga, "Absolute Minimization of Completely Specified Switching Functions", *IEEE Transactions on Computers*, vol. 40, pp. 53-65, Jan. 1991.

[8] M. Karnaugh, "The map method for synthesis of combinational logic circuits", Communication Electronics, pp. 539-599, Nov. 1953.

[9] A. Svoboda and D. white, Advanced logic circuit Design Techniques, New York: Garland STPM press, 1979.

筆 者 紹 介

洪 性 濟

1951年 3月 30日生
1973年 2月 서울대학교 전자공학과 (학사)
1979年 5月 Iowa State University 전자계산학과 (석사)
1983年 5月 University of Illinois at Urbana-Champaign 전자계산학과 (박사)

1973年 5月 ~ 1975年 12月 중앙경리단 프로그래머
1976年 1月 ~ 1977年 7月 동양전산(주) 엔지니어
1983年 6月 ~ 1989年 6月 General Electric 중앙연구소 Staff
1989年 7月 ~ 현재 포항공대 전자계산학과 및 전자전기공학과 부교수

주관심 분야 : VLSI CAD, VLSI 설계, 테스팅, 결함포용시스템