■1 群（信号・システム）- 8 編（論理回路）

2 章 組合せ論理回路

（執筆者：高木直史）[2010 年 2 月 受領]

■概要■

組合せ（論理）回路は，NOT や AND，OR などの基本的な論理関数を計算する論理ゲートと，それらを接続する配線からなる，フィードバックループをもたない回路である．いくつかの入力と出力をもち，各入出力は 0 または 1 の値を取る．各出力の値は，入力の値の組合せのみにより一意に定まる．すなわち，組合せ回路は論理関数を計算している．組合せ回路がある論理関数を計算しているとき，その論理関数を実現しているという．

論理関数が与えられたとき，それを実現する組合せ回路を設計することは，ディジタル回路の設計における重要な過程である．任意の論理関数は積和形論理式で表すことができる．したがって，NOT，AND，OR の各論理ゲートを用いて，NOT-AND-OR 形式の組合せ回路で，任意の論理関数を実現することができる．このような回路を慣習上，（AND-OR）二段（組合せ）回路という．n 変数論理関数を 二段回路で実現する場合，NOT ゲートは最大 n 個，AND ゲートは積和形表現における積項の個数，OR ゲートは一つ必要である，各 AND ゲートのファンイン（入次数）は対応する積項のリテラル数に等しい．したがって，ゲート数や配線数の少ない二段回路を得るには，積項数や積項に現れるリテラル数の少ない積和形論理式を求めればよい．与えられた論理関数に対し，最小の積和形論理式，すなわち，積項数が最小，積項数が同じ場合はリテラル総数が最小の積和形論理式を求める問題は，典型的な組合せ最適化問題であり，厳密解法とともに近似解法が提案されている．

用いる論理ゲートのファンインに制限がある場合，論理関数を実現する組合せ回路は，通常，多段になる．用いることのできる論理ゲートは，テクノロジによって異なる．多段（組合せ）回路の設計は，まず，テクノロジーに依存しないレベルで行い，後に，テクノロジーマッピングを行うことが多い．テクノロジーに依存しないレベルでの多段回路の設計では，与えられた論理式をより簡潔な論理式に変換するファクタリング，及び，複数の論理式からの共通式の抽出が重要である．これらの処理では，論理式の除算が基本となる．論理式の除算は解が一意に定まらない．このため，論理的な除算に代わり，代数的な除算，特に，商と剰余が一意に定まる弱い除算が広く用いられている．

【本章の構成】

本章では，2-1 節で，組合せ論理回路とはどのようなものか，組合せ論理回路により論理関数を実現するとはどういうことかを述べる．2-2 節では，最小の二段組合せ回路（最小の積和形論理式）を求める厳密解法，及び，近似解法の一つである ESPRESSO アルゴリズムについて述べる．2-3 節では，多段組合せ回路のテクノロジーに依存しないレベルでの設計手法について述べる．2-4 節では，ディジタル回路でよく用いられる基本的な組合せ回路をいくつか紹介する．

■1 群 - 8 編 - 2 章

2-1 組合せ回路による論理関数の実現

（執筆者：高木直史）[2009 年 3 月 受領]

「組合せ論理回路」（あるいは，単に「組合せ回路」）は，図 2･1 に示すように，いくつかの入力信号（図では，x_1，x_2，x_3）といくつかの出力信号（図では，z_1，z_2）をもち，否定（NOT）や論理積（AND），論理和（OR）などの基本的な論理関数を計算する論理ゲートと，それらを接続する配線（信号線）からなる．入力信号はいくつかの論理ゲートの入力に結ばれ，論理ゲートの出力はほかのいくつかの論理ゲートの入力に結ばれるか，あるいは，回路の出力信号となる．信号は回路の入力から論理ゲートの入力，論理ゲートの出力から他の論理ゲートの入力，回路の出力へと一方向に伝わる．ある論理ゲートから出力された信号が（いくつかの論理ゲートを経て），その論理ゲートに入力されることはない．すなわち，フィードバックループは存在しない．入力信号及び出力信号は，それぞれ論理変数に対応しており，0 か 1 のいずれかの値をとる．したがって，回路内の配線，論理ゲートの入力及び出力は，すべて，0 か 1 のいずれかの値をとる．

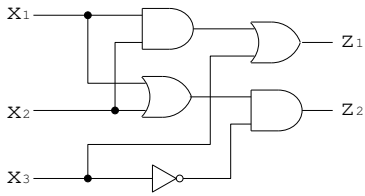

図 2･1　論理回路図

図 2･1 のような図を論理回路図という．論理回路図において，信号線と信号線が交差している場合，「･」があれば接続しており，なければ単に交差しているだけで接続していないと考える．論理回路図では，論理ゲートを表す記号として，図 2･2 に示す記号が広く用いられている．図の (a) は NOT ゲート（インバータ），(b) は AND ゲート，(c) は OR ゲートを表している．図では 2 入力の AND ゲート及び OR ゲートを示しているが，3 入力以上のものも用いられる．論理ゲートの入力数をファンイン（入次数）という．このほか，(d) に示す NAND ゲートや，(e) の NOR ゲート，(f) の EXOR ゲートなどが用いられる，NAND ゲートは否定論理積，NOR ゲートは否定論理和．EXOR ゲートは排他的論理和を計算する．

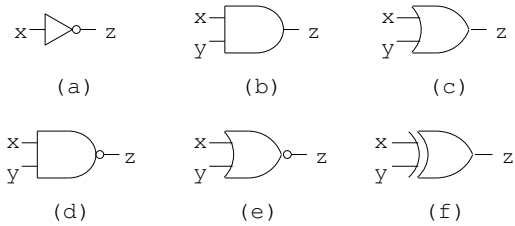

図 2･2　基本論理ゲートの記号

組合せ回路において，各出力信号に対応する論理変数（出力変数）の値（0 か 1）は，入力信号に対応する論理変数（入力変数）の値によって定まる．すなわち，各出力変数は入力変数の関数になっている．この関数は，論理関数である．つまり，n 入力, m 出力の組合せ回路は，m 個の n 変数論理関数を計算している．例えば，図 2･1 の組合せ回路は二つの 3 変数論理関数を計算しており，$z_1 = (x_1 \cdot x_2) + x_3$，$z_2 = (x_1 + x_2) \cdot \overline{x_3}$ である．組合せ回路が，ある論理関数を計算しているとき，その組合せ回路はその論理関数を実現しているという．

論理関数が与えられたときに，これを実現する組合せ回路を設計することを考える．与えられた論理関数が n 変数論理関数である場合，回路は n 入力 1 出力になり，論理関数は回路の入出力関係が満たすべき条件，すなわち，設計仕様を表している．与えられた論理関数が不完全指定ならば，ドントケアな入力の値の組合せに対する出力は 0 であっても 1 であってもよい．複数の論理関数を一つの回路で実現する場合は，回路は出力を複数もち，その各々が各論理関数に対応する．

任意の論理関数は積和形論理式で表すことができる．したがって，NOT，AND，OR の各ゲートを用いて，NOT-AND-OR 形式の組合せ回路で，任意の論理関数を実現することができる．このような回路を慣習上，AND-OR 二段組合せ回路という．n 変数論理関数を AND-OR 二段組合せ回路で実現する場合，NOT ゲートは最大 n 個，AND ゲートは積和形表現における積項の個数，OR ゲートは一つ必要である．AND ゲートのファンインは最大 n，OR ゲートのファンインは AND ゲートの個数（積項の個数）になり，場合によっては，非常に大きくなる．AND-OR 二段組合せ回路を実現するデバイスとして，PLA（programmable logic array）がある．AND-OR 二段組合せ回路の設計法については，2-2 節で述べる．

論理ゲートには，通常，ファンインに制限がある．論理ゲートのファンインに制限がある場合，論理関数を実現する組合せ回路は，通常，多段になる．実際のディジタル回路では，TTL（Transistor Transistor Logic）という回路技術では NAND，ECL（Emitter Coupled Logic）では NOR 及び OR，現在の LSI の主流である CMOS（Complementary Metal-Oxide Semiconductor）では NAND，NOR のほか，OR-AND-NOT，AND-OR-NOT などの複合ゲートが使用可能な論理ゲートとなる．これらの論理ゲートのファンインは通常 4〜6 程度までである．

組合せ回路は，使用可能な論理ゲートを配線でつなぎ合わせて構成する．したがって，組合せ回路の設計は，使用可能な論理ゲートが実現する論理関数（基本論理関数）の集合から，与えられた論理関数を合成することであると考えることができる．基本論理関数の集合が万能であれば，任意の論理関数を合成できる．一般に，ある論理関数を実現する組合せ回路は多数存在する．回路を設計する場合，与えられた論理関数を実現する，できる限り「よい」回路が得られるようにすることが重要である．どのような回路が最も「よい」かは，何を求められているかによる．高速計算が求められることもあれば，ハードウェア量を少なくすることを求められる場合もある．更には，ある与えられた計算時間以内で動作し，できる限りハードウェア量を少なくすることを求められる場合もある．複数の論理関数を実現する場合は，部分回路が複数の論理関数に共有されるようにすることも重要である．多段組合せ回路の設計法については，2-3 節で述べる．

■1 群 - 8 編 - 2 章

2-2 二段組合せ回路の設計

（執筆者：松永裕介）[2008 年 11 月 受領]

一般に一段目に AND ゲート，二段目に OR ゲートを用いた回路を二段（組合せ）回路と呼ぶ．二段回路は PLA（Programmable Logic Array）で容易に実現することができ，論理関数を表現する積和形論理式から直接その構造をつくり出すことが可能であるため，その設計法に関しては古くから様々な研究がなされてきた．ここでは最小*な二段回路を求める厳密手法と近似手法について紹介する．

n 入力 1 出力の論理関数 $f : B^n \to B$ $(B = \{0, 1\})$ で，すべての入力 $m \in B^n$ に対してその出力値が定められているものを**完全指定論理関数（completely specified logic function）**と呼ぶ．完全指定論理関数はまた，その出力を 1 とするような入力値の集合と考えることもできる．二つの完全指定論理関数 f^m, f^M（ただし，二つの関数を集合とみなしたときに $f^m \subseteq f^M$ が成り立つ）に対して，完全指定論理関数の集合 $\mathcal{F} = \{f | f^m \subseteq f \subseteq f^M\}$ が定義される．この集合を**不完全指定論理関数（incompletely specified logic function）**と呼ぶ．不完全指定論理関数は n 入力ブール空間 B^n を次のような三つの部分空間 f^{ON}, f^{DC}, f^{OFF} に分割していると考えることもできる．ここで，$f^{ON} = f^m, f^{DC} = \overline{f^m} \cdot f^M, f^{OFF} = \overline{f^M}$ であり，f^{ON}, f^{DC}, f^{OFF} をそれぞれ**オン・セット**，**ドントケア・セット**，**オフ・セット**と呼ぶ．図 2･3 に不完全指定論理関数の例を示す．

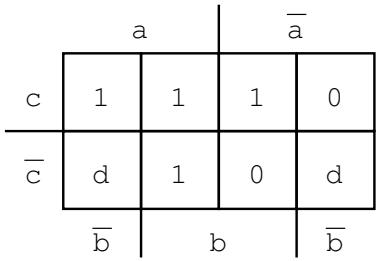

図 2･3　不完全指定論理関数を表すカルノー図

ここで，

$$f^{ON} = ab + ac + bc,\ f^{OFF} = \bar{a}b\bar{c} + \bar{a}\bar{b}c,\ f^{DC} = \bar{b}\bar{c}$$

である．

変数または変数の否定を**リテラル**と呼ぶ．本節及び次節では，**積項**をリテラルの集合として扱う．また，同様に，**積和形論理式**を積項の集合として扱う†．積項 C_1 の表す論理関数 f_{C_1} が積項 C_2 の表す論理関数 f_{C_2} を包含しているとき（すなわち，$f_{C_1} \supseteq f_{C_2}$），積項 C_1 は積項 C_2 を**論理的に包含する**という．

---

* 厳密な定義は後述

† 意味的には，積項は，その要素のリテラルが表すリテラル関数の論理積を表し，積和形論理式は，その要素の積項が表す関数の論理和を表している．

積項 C に対応した論理関数 f_C が不完全指定論理関数 $\mathcal{F} = (f^{ON}, f^{DC}, f^{OFF})$ に対して，$f_C \subseteq f^{ON} + f^{DC}$ を満たすとき，積項 C は $\mathcal{F}$ の**内項（implicant）**であるという．また，内項 C からいかなるリテラルを取り除いてもその結果の積項 C' が $f^{ON} + f^{DC}$ に含まれなくなるときに，C を**主項（prime implicant）**という．つまり，主項は自分以外のほかの内項には論理的に包含されない内項である．

図 2･3 の例で，積項 $C_1 = a\bar{c}$ は内項であるが，積項 $C_2 = \bar{b}c$ はオフ・セットの最小項 $\bar{a}\bar{b}c$ を含むので内項ではない．また，C_1 はほかの内項 $C_3 = a$ に論理的に包含されるので主項ではない．この場合，C_3 を論理的に包含する内項は存在しないので C_3 は主項となる．ほかの主項としては $C_4 = bc$ と $C_5 = \bar{b}\bar{c}$ がある．まず，C_4 が内項であることは明らかである．C_4 から b，または c のどちらかのリテラルを取り除いてつくった積項 c と b は内項でないので，C_4 を論理的に包含する内項は存在しないことが分かる．つまり，C_4 は主項となる．C_5 も同様である．ただし，C_5 はドントケア・セットのみを含む特殊な（と同時に役に立たない）主項である．

積項の集合 $\mathcal{C} = \{C_1, \ldots, C_p\}$ が $\mathcal{F} = (f^{ON}, f^{DC}, f^{OFF})$ に対して，$f^{ON} \subseteq \bigcup C_i \subseteq f^{ON} + f^{DC}$ を満たすとき，$\mathcal{C}$ は $\mathcal{F}$ の**被覆**であるという．この被覆を，元の論理関数に対する**積和形論理式表現**（または単に積和形論理式）と呼ぶ．

与えられた不完全指定論理関数 $\mathcal{F} = (f^{ON}, f^{DC}, f^{OFF})$ に対して，積項数最小（積項数が同じ時はリテラル数が最小）の被覆を求める処理を，論理関数に対する積和形論理式の簡単化と呼ぶ．また，そのときの被覆を最小被覆と呼ぶ．例えば，図 2･3 の例に対しては，$a + bc$ が（唯一の）最小被覆となる．ただし，一般に最小被覆は唯一とは限らない．

2-2-1 厳密解法

積和形論理式の簡単化に対する厳密解法としては Quine-McCluskey 法[1)]が知られている．このための重要な定理が以下に示される．

定理 1 任意の不完全指定論理関数 $\mathcal{F}$ に対して，最小被覆は主項のみから構成される．

つまり，まず，与えられた論理関数に対して，すべての主項を列挙し，その主項を組み合わせて最もコストの低い被覆をつくればそれが最小被覆となることが分かる．

(1) 主項の列挙

主項の列挙法としては，与えられた論理関数の最小項をすべて列挙することを出発点とするアルゴリズムがよく知られているが，入力変数が増えると現実的ではない．ここでは，論理関数が任意の積和形論理式で与えられたときに，すべての主項を列挙する実用的なアルゴリズムについて述べる[2)]．そのためにいくつかの用語の定義を行う．

二つの積項 C_1 及び C_2 に対して，ある変数 x の肯定及び否定のリテラルがそれぞれの積項に含まれているとき，C_1 及び C_2 は変数 x に関して相反しているという[3)]．例えば，xyz と $\bar{x}\bar{y}z$ の場合，変数 x 及び y に関して相反している．

ただ一つの変数に関してのみ相反している二つの積項 C_1 及び C_2 から，その相反しているリテラルを取り除いたリテラル集合からつくられる積項を**コンセンサス（consensus）**と呼ぶ．例えば $x\bar{y}$ と yz のコンセンサスは xz となる．相反している変数の数が 1 以外の場

合にはコンセンサスは定義されない．

論理関数 $\mathcal{F}$ のすべての主項からなる積和形論理式を $\mathcal{F}$ **の完全な積和形論理式（complete sum of** $\mathcal{F}$**）**と呼ぶ*．この完全な積和形論理式は与えられた論理関数に対して一意に定まるいわゆる標準形（cannonical form）となっている†．

定理 2 積和形論理式が完全な積和形論理式であるための必要十分条件は以下のとおりである．

1. いかなる積項もほかの積項に論理的に包含されない．
2. いかなる二つの積項のコンセンサスも，
   - 定義されないか，
   - 既にある積項に含まれているか，

   のどちらかである．

この定理から，与えられた積和形論理式の積項の対からコンセンサスを計算し，その項を元の式に追加して，包含される積項を削除すれば，完全な積和形論理式となることが分かる．例として次の式の表す論理関数の完全な積和形論理式を求めることを考える．

$$x_1x_2 + \overline{x_2}x_3 + x_2x_3x_4$$

まず，1 番目の項と 2 番目の項のコンセンサス x_1x_3 を生成し，追加する．次に，2 番目の項と 3 番目の項のコンセンサス x_3x_4 を生成し，追加する．ここまでの中間結果は以下のようになる．

$$x_1x_2 + \overline{x_2}x_3 + x_2x_3x_4 + x_1x_3 + x_3x_4$$

これ以上はコンセンサスの定義できる積項の対がないので，今度は論理的に包含されている積項を探す．この例では $x_2x_3x_4$ が x_3x_4 に論理的に包含されているので削除する．結果として，

$$x_1x_2 + \overline{x_2}x_3 + x_1x_3 + x_3x_4$$

が得られる．この式は完全な積和形論理式であり，すべての主項からなっている．

完全な積和形論理式に関する性質を述べる前に，積項の積及び，積和形論理式の積について定義しておく（定義 1）．

定義 1 二つの積項 C と D の積（$C \cdot D$ と表記）とは，以下のように定義される積項のことである．

$$C \cdot D = \begin{cases} \phi & \textit{if } \exists x\ (x \in C \cup D \text{ かつ } \bar{x} \in C \cup D) \\ C \cup D & \text{それ以外} \end{cases}$$

* 完全指定論理関数の「完全」とは意味が異なる．

† Blake cannonical form （BCF）とも呼ばれる[3)]．

二つの積和形論理式 F と G の積（$F \cdot G$ と表記）とは，以下のように定義される積和形論理式のことである．

$$F \cdot G = \{C \cdot D | C \in F, D \in G, C \cdot D \neq \phi\}$$

□

定理 3 二つの論理関数 $\mathcal{F}_1$ 及び $\mathcal{F}_2$ とその完全な積和形論理式 F_1 及び F_2 が与えられたとき，以下の手続きで得られる論理式は，論理関数 $\mathcal{F}_1 \cdot \mathcal{F}_2$ の完全な積和形論理式である．

1. F_1 と F_2 の積を計算する．
2. ほかの積項に論理的に包含される積項を削除する．

また，以下のシャノン展開（ブール展開）の公式を用いると，関数 f を二つの関数の積のかたちで表すことができる．

$$f(x_1, x_2, \ldots, x_n) = (\overline{x_1} + f(1, x_2, \ldots, x_n)) \cdot (x_1 + f(0, x_2, \ldots, x_n))$$

このシャノン展開を 1 回行うごとに結果の論理関数に関係する変数は最低一つずつ減ってゆく．この展開を再帰的に行えば，最終的には展開された部分関数は定数 0 または 1 になる．そこから完全な積和形論理式を構築し，定理 3 にしたがって二つの完全な積和形論理式の積を計算すれば，与えられた論理式 F に対する完全な積和形論理式を計算するアルゴリズム（図 2･4）を構築することができる．

```
COMPLETE_SUM(F) {
    if (F が定数であるか一つの積項からなる) {
        return F
    }
    x ← F に現れる任意の変数
    F_0 ← F|_{x=0}
    F_1 ← F|_{x=1}
    F_tmp = (x + COMPLETE_SUM(F_0)) · (x̄ + COMPLETE_SUM(F_1))
    /* F_tmp は積和形論理式の積を定義 1 に従って積和形論理式に展開したもの */
    foreach C ∈ F_tmp {
        if (C が F_tmp の他の積項に論理的に包含されている) {
            C を F_tmp から取り除く．
        }
    }
    return F_tmp
}
```

図 2･4 COMPLETE_SUM アルゴリズム

（2）最小被覆の選択

ある主項の集合(内項でもよい)$F = \{C_1, C_2, \ldots C_n\}$が論理関数$\mathcal{F} = (f^{ON}, f^{DC}, f^{OFF})$の被覆となるための必要十分条件は，オン・セットに含まれるすべての最小項に対して，その最小項を論理的に包含する積項 C_i が必ず一つ以上，存在することである*．つまり，

$$\forall m \in f^{ON}, \exists C_i \in F, m \subseteq f_{C_i}$$

である．この条件を被覆条件と呼び，この問題の被覆解の中で最小のものを求める問題を最小被覆（minimum covering）問題と呼ぶ．積和形論理式の簡単化問題としては，1) 積項数が最小，2) 積項数が同じときにはリテラル数が最小，である解を最小解と呼んでいるので，ここでは，（積項数, リテラル数）の二つ組を解のコストと定義し，二つのコスト $C_1 = (P_1, L_1)$ と $C_2 = (P_2, L_2)$ の間の大小関係を以下のように定義することとする．

$$C_1 < C_2 \iff (P_1 < P_2) \vee ((P_1 = P_2) \wedge (L_1 < L_2))$$

図 2･5 に最小被覆問題を行列のかたちで表現したものを示す．ここで，各列 C_i は主項に対応しており，各行 m_j は被覆すべきオン・セットの最小項に対応している．主項 C_i が最小項 m_j を被覆しているとき，要素 B_{ij} は 1 となる．この行列を用いると最小被覆問題は，各行に対して最低一つの列が被覆しているような列集合の中の最小コスト集合を求める問題とみなすことができる．この例で，もしも各列のコストがすべて同一であれば，最小被覆問題の解は $\{C_2, C_3\}$ となる．

	C_1	C_2	C_3	C_4
m_1	1	0	1	0
m_2	1	1	0	1
m_3	0	1	0	0
m_4	0	0	1	1

図 2･5　最小被覆問題

積和形論理式の簡単化問題を解くためのコストを考えると，積項数はすなわち解に含まれる列の数であり，リテラル数は解の各列に対応する主項のリテラル数の和となるので，各々の列のコストとして，対応する主項のリテラル数を設定しておけば，解全体のコストは（列数, $\sum$ 列のコスト）で表される．もしくは，すべての主項のリテラル数の和よりも大きな定数を K として，各列に “K + 対応する主項のリテラル数” というコストを設定しておけば，“$\sum$ 列のコスト” を最小化する解が積和形論理式の簡単化問題の解となる．

最小被覆問題を解く汎用的なアルゴリズムは，一つの列を選び，a) その列の主項を解として含めたときの部分問題の最適解と，b) その列の主項を解として含めなかったときの部分問題の最適解を求め，その良い方の解を返す，というものである．ただし，そのままでは常に列数（＝主項の数）の指数乗の手間がかかってしまい効率的ではない．そこでいくつかの工夫が提案されている．

* 主項（内項）であるのでオフ・セットに含まれる最小項を論理的に包含しないことは保証されている．

必須列: ある行の 1 となる要素が一つしかない場合，その行を被覆するためには唯一 1 である列を選択しなければならない．そのような列を必須列と呼ぶ．必須列の選択により，元の行と同様に必須列によって被覆されているほかの行も削除できる．

行支配: 二つの行 i_1 と i_2 に対して，i_1 上で 1 となっているすべての列において i_2 上の要素も 1 である場合，行 i_1 は行 i_2 を支配するという．行 i_1 を被覆するためにいかなる列を選択しても同時に行 i_2 が被覆されることは明らかなので，行 i_2 は削除できる．

列支配: 二つの列 j_1 と j_2 に対して，j_2 上で 1 となっているすべての行において j_1 上の要素も 1 である場合，列 j_1 は列 j_2 を支配するという．列 j_2 を選択するときに被覆されるすべての行は列 j_1 を選択しても被覆されるので，列 j_2 のコストが列 j_1 のコストと同じか大きい場合には j_2 を選ぶ代わりに列 j_1 を選ぶことでコストが同じかより良い解が得られるので列 j_2 は削除できる．

分岐限定法: 例えば，選択された列の主項を解に含めたときの最適解のコスト ϕ_1 が実際に求まっていて，選択された列の主項を解に含めないときの最適解はまだ求まっていないとする．このいまだ解いていない部分問題 S に対する下限値 $L(S)$ を計算した結果，$L(S) \geq \phi_1$ であることが分かったら，実際に S に対する最適解を求めてもその解のコストが ϕ_1 を下回ることはない．そこで，そのような場合には部分問題 S を解くことをしないで ϕ_1 を全体の最適解として返す．このように下限を用いて分岐した子問題の探索を中止する手法を分岐限定法という．

論理関数の変数の数を n としたときには最悪 3^n に比例した数の主項が存在し，更にその主項を列要素とした集合被覆問題を解かなければならない．そのため，積和形論理式の簡単化問題は，変数の数が大きくなると現実的な時間内で解くことは難しくなる．

2-2-2 近似解法

ある程度規模が大きくなっても扱うことができる近似解法もいくつか提案されている (MINI[4], ESPRESSO[5]など)．特に文献 5) には ESPRESSO アルゴリズムの詳細が述べられている．ここでは ESPRESSO アルゴリズムについて簡単に説明する．

多くの近似解法では，与えられた初期解を変形して良い解を求めるという逐次改善法を用いている．逐次改善法では常に現在の解の近傍しか探索しないので計算時間の指数的爆発を抑えることはできるが，逆に局所最適解（local minimum solution）に陥ると真の最適解に到達しにくい，という欠点をもつ．そこで，ESPRESSO ではいったん，解を悪い方向に戻してから異なる方向に対して改善を行う戦略を用いている．ESPRESSO アルゴリズムの概略を図 2･6 に示す．

ここで，引数の F，D，R はそれぞれオン・セット，ドントケア・セット，及びオフ・セットを表す積和形論理式である．ただし，F と D は重なる場合があり，その場合には実際のオン・セットは $F \setminus D$ と定めるものとする．つまり，このアルゴリズム中では F はオン・セットを表すと同時に現在の解である被覆を表しているとみなすこともできる．ESPRESSO アルゴリズムは三つのサブアルゴリズムから構成される．

```
ESPRESSO(F, D, R) {
    F ← EXPAND(F, R)
    F ← IRREDUNDANT(F, D)
    do {
        F ← REDUCE(F, D)
        F ← EXPAND(F, R)
        F ← IRREDUNDANT(F, D)
        最良解を更新する．
    } while(解が減少している)
}
```

図 2·6 ESPRESSO **アルゴリズム**

以下では，ESPRESSO の各処理を説明するために，処理途中の積和形論理式を表すためにカルノー図を用いる．カルノー図中の一つのループで囲まれた部分が一つの積項に対応している．

(1) EXPAND 処理

EXPAND は項の拡大（expand）を行うアルゴリズムで，与えられた被覆の項を一つずつ取り出してその項を含む主項と置き換える．図 2·7 の例で，元の論理式は $ac + ab\bar{c} + \bar{a}\bar{b}\bar{c}$ であるが（同図 (a)），例えば ac は $\bar{c}$ の方向に拡大することが可能であり，より大きな項（この場合は主項）a に置き換えることができる．すると，新たな項 a はほかの項 $ab\bar{c}$ をも含むのでこの項を削除することができる．残りの項 $\bar{a}\bar{b}\bar{c}$ も拡大することができて $\bar{c}$ に置き換え可能である．結果としてより簡単な論理式 $a + \bar{c}$ を得る（図 2·7(b)）．このように項の拡大はまず第一にリテラル数の減少を伴う．また，場合によってはほかの項を完全に含むことによって積項数を減少させることもある．

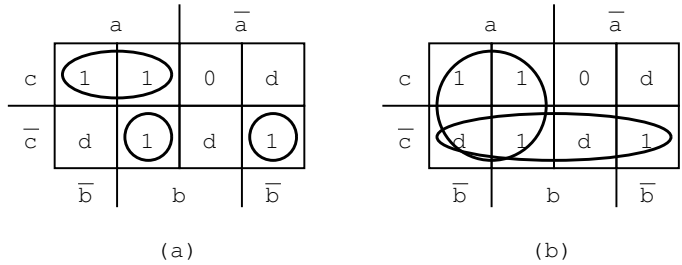

図 2·7 EXPAND **処理**

一つの項を含む主項はただ一つとは限らないのでどの主項と置き換えるのかが EXPAND 処理の一つの問題となる．k 個の変数リテラルを含む項を拡大するには $l \leq k$ 個のリテラルを選んで取り除けばよい．最悪の場合，このようなリテラルの選び方（拡大の仕方）は $O(2^k)$ 通り存在するので，それらをすべて列挙して最も良いものを選び出す方法は効率が悪い．そ

こで，ESPRESSO ではほかの項を包含する可能性のある方向へ拡大を行い，包含する項がない場合には最も多くのリテラルが削除できる（＝大きくできる）方向に拡大を行う．このように探索空間を狭めることで，最適解を見逃す可能性はあるが，与えられた項に対して常に多項式時間で処理を完了させることを保証している．CAD のアルゴリズムに対してはこのような計算時間（や使用メモリ量）に対する堅牢さ（robustness）は重要な用件である．

どの順番で項を拡大してゆくかも結果に影響を与える問題である．例えばある項 C_1 を先に拡大するとほかの項 C_2 を包含できるが，先に C_2 を拡大してしまうと C_1 を拡大しても包含できない場合がある．ESPRESSO ではまず，ほかの項から包含される機会の少ない項から先に拡大するヒューリスティックを用いている．

(2) IRREDUNDANT 処理

IRREDUNDANT は与えられた項の中から冗長な（redundant）項を取り除く．例えば，図 2･8 の例で，元の論理式 $ab + ac + b\bar{c} + \bar{a}\bar{c} + \bar{a}\bar{b}$ 中の項 ab と $\bar{a}\bar{c}$ は冗長であり，それらを取り除いた論理式 $ac + b\bar{c} + \bar{a}\bar{b}$ も正しい被覆となっている．

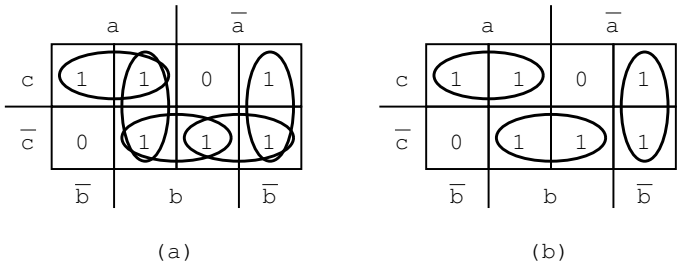

図 2･8 IRREDUNDANT **処理**

注意しなければならないのは冗長な項というのが相対的な関係で定められるということである．例えば，同じ例で，項 ab と $b\bar{c}$ はそれぞれ単独で取り除くことは可能であるが，この二つを同時に取り除くことはできない．このように削除可能かどうかは同時に削除する項の組合せで決まるので，最悪の場合，項数の指数に比例した組合せを考えなければならない．実はこの問題は厳密最小化アルゴリズムで用いられている最小被覆問題そのものである．異なるのは対象となるのがすべての主項ではなく，現在の解に含まれている項のみということである．いずれにせよ，最小被覆問題を厳密に解くためには最悪，指数時間を必要とするため，ESPRESSO では極小解を多項式時間で求める近似解法を用いている．また，図 2･8 の例で，ac や $\bar{a}\bar{b}$ のように，必ず残しておかなければならない項を特定し（必須項と呼ぶ），解空間を狭める工夫もある．これは厳密解法でも有効な手段である．

(3) REDUCE 処理

REDUCE は項を順に選んで，被覆条件を変えない範囲で最も小さな項に置き換えるものである（縮小処理）．REDUCE は EXPAND の逆の処理とみなすこともできるが，EXPAND と異なり，処理する項を決めた時点で縮小結果は一意に求まる．REDUCE の結果は処理

する項の順番のみに依存する．直感的に言うと先に縮小する項ほど小さくなる傾向があるので，ほかの項に包含されやすい項を先に処理すると良い．REDUCE はほかの二つの処理と異なり解のコストを増大させるが，そのために局所最適解を抜け出す可能性もある．例えば図 2･9 の例 (a) は図 2･8 の例から冗長な項を一つ抜いたものであるが，もうこれ以上の項の拡大は行えず，また，項を削除することもできない局所最適解となっている．しかし，この例に対しては図 2･8(b) のような真の最適解が存在する．そのような最適解へは EXPAND と IRREDUNDANT だけでは到達することが不可能なので，いったん局所最適解でなくす REDUCE 処理を行う．この例では項 ab と $\bar{a}\bar{c}$ を縮小して，$ab\bar{c}$ と $\bar{a}b\bar{c}$ に置き換えている（図 2･9(b)）．するとこれらの縮小された項は，その縮小された方向以外に拡大可能となり，ほかの項を包含する方向を優先する EXPAND 処理によって一つの項で包含することができる（図 2･9(c)）．このように REDUCE は次の EXPAND と IRREDUNDANT のために解を動かす働きをもつ．

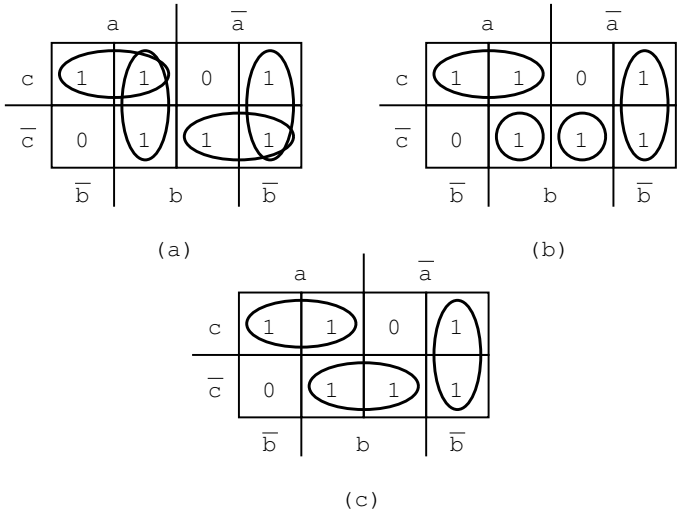

図 2･9 REDUCE **処理**

■参考文献

1) E.J. McCluskey, “Minimization of Boolean functions,” Bell System Technical Journal, vol.35, pp.1417-1444, Nov. 1956.
2) G.D. Hactel, and Fabio Somenzi, “Logic Synthesis and Verification Algorithms,” Kluwer Academic Publishers, 1996.
3) F.M. Brown, “Boolean Reasoning,” Kluwer Academic Publishers, 1990.
4) S.J. Hong, R.G. Cain and D.L. Ostapko, “MINI: A Heuristic Approach for Logic Minimization,” IBM J. of Res. and Dev., vol.18, pp.443-453, Sep. 1974.
5) R.K. Brayton, C. McMullen, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis,” Kluwer Academic Publishers, 1984.

■1 群 - 8 編 - 2 章

2-3 多段組合せ回路の設計

（執筆者：松永裕介）[2008 年 11 月 受領]

組合せ論理回路をスタンダードセルや FPGA（Field Programmable Gate Array）で実現する場合，特定の機能をもつ論理素子を自由に組み合わせて使えるため，その構造に二段回路のような規則性はない．そのため，面積や遅延時間，消費電力など様々な制約条件や目的関数を考慮した自由度の高い設計が可能となる．しかし，このことは逆に「厳密に最適な」多段回路を設計することが困難であることも示している．実際，面積やゲート数といった単純な尺度に対しても多段回路の厳密最小解を求めることは非常に難しい．また，構成要素として用いることのできる論理素子の種類が可変であるという問題もある．例えば，あるテクノロジーライブラリでは $\overline{(a_1b_1 + a_2b_2 + a_3b_3)}$ というかたちの AND-OR-INVERTOR が用意されているが，別のテクノロジーライブラリでは存在しないということがあり得る．また，FPGA を用いて実現する場合には，一定の入力数以下の論理関数であれば一つの論理素子（Logic Element）で実現可能という場合もある．このように多段組合せ回路の設計においては回路実現に用いるテクノロジーに対する様々な考慮が必要となるが，紙面の都合上，テクノロジーに依存した設計手法は割愛し，ここでは，テクノロジーに依存しないレベルで多段回路を設計するための手法について紹介する．テクノロジーマッピングに関しては 10 群 1 編 3 章 3-3 節を参照のこと．

2-3-1 多段論理回路のモデル

テクノロジー非依存のレベルで多段回路をモデル化するためにブーリアンネットワーク[1)]と呼ばれるデータ構造が用いられる．以下にブーリアンネットワークに関する定義を述べる．

定義 1 ブーリアンネットワーク は非巡回有向グラフで，その節点は，**入力節点**，**論理節点**，及び**出力節点**と呼ばれる 3 種類の節点から構成される．**入力節点**は回路の外部入力を表し，入り枝をもたない．**論理節点**は回路内部の論理機能を表す論理関数をもち，その論理関数の入力変数と同数の入り枝をもつ*．**出力節点**は回路の外部出力を表し，ただ一つの入り枝をもつ．入り枝に接続している論理節点の出力が外部出力となっていることを表している．節点 j の機能を実現するために節点 i が直接用いられている場合にのみ節点 i から節点 j への枝が存在する ． □

図 2･10 にブーリアンネットワークの例を示す．また，このブーリアンネットワークは次のような論理式として表すこともできる．

$$
\begin{aligned}
d &= ab \\
e &= b + c \\
f &= ad + e \\
g &= f
\end{aligned}
$$

* 定数関数の場合，入力変数はないので，そのような論理節点は入り枝をもたない

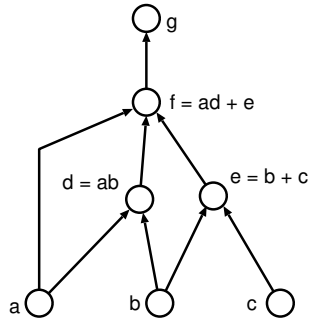

図 2·10　ブーリアンネットワークの例

この例では，節点 a,b，及び c が入力節点，節点 d,e，及び f が論理節点，節点 g が出力節点である．このようなモデル化を行うことで，実際に使うセルライブラリとは無関係に大まかな回路の構造を決めることができる．

ブーリアンネットワークの各論理節点はその機能を表す論理関数をもつが，それをどのようなかたちで保持するか，という問題がある．二段論理回路の場合には，

- 対象回路の機能（論理関数）をいかに表現するか？
- 最終的な実装をいかに表現するか？

の二つは同一のものであったので，選択の余地はなく積和形論理式が用いられていたが，多段論理回路の場合にはそうではない．カリフォルニア大学バークレイ校で開発された論理合成システム MIS[1)]では，積和形論理式とファクタードフォームの二つの表現を用いてそれぞれの特長を活かした処理を行っていた．例えば，

$$ace + ade + bce + bde + \bar{e}$$

という積和形論理式は次のような論理式（ファクタードフォーム）でも表せる．

$$e(a + b)(c + d) + \bar{e}$$

こちらのかたちの方がより簡潔であり，また実際の回路の実現形式にも近いので，このようなかたちで論理関数を表現した方が論理合成には都合が良い．

定義 2 ファクタードフォームは以下の規則により再帰的に定義される．

1. **積**とは単一のリテラルかファクタードフォームの積である．
2. **和**とは単一のリテラルかファクタードフォームの和である．
3. ファクタードフォームとは積か和である．

□

一般に，ファクタードフォームはそのリテラル数を評価指標とすることが多い．以下では，よりリテラル数の少ないファクタードフォームを生成することを目的とする．

2-3-2 代数的な除算

良いファクタードフォームを得るために重要な操作は，論理関数または論理式に対する「除算（division）」である．ただし，論理関数に対する除算は解が一意に定まらない点と，処理が効率的でないという問題点がある．更に，論理関数の入力変数の数を N とした場合，その論理関数を「割る」ことのできる除数は $O(2^{2^N})$ 存在する．つまり，論理関数に対して除数を列挙し，その除数を用いた除算を行って解の候補を列挙する，というアルゴリズムでは厳密解を効率的に求めることはできない．そこで，実用的な手法として（積和形）論理式に対する除算を定義し，その除算に従った除数の列挙を行うというヒューリスティックを用いた近似アルゴリズムが用いられている．

定義 3 積項 C のサポート $supp(C)$ とは，C のリテラルに関係している変数の集合である．すなわち，$supp(C) = \{x | x \in C \ or \ \bar{x} \in C\}$

積和形論理式 F のサポート $supp(F)$ とは，F が直接，関係している変数の集合である．つまり，$supp(F) = \bigcup_{C \in F} supp(C)$ □

定義 4 積和形論理式 F と G のサポートが互いに素なとき，その積 $F \cdot G$ を**代数的な積（*algebraic product*）**という．そうでなければ**論理的な積（*Boolean product*）**という．□

積和形論理式を積項の集合としてとらえた場合, 代数的な積は F と G の直積集合（Cartesian product）とみなすことができる．

定義 5 与えられた二つの積和形論理式 F と P に対して，次のような二つの積和形論理式 Q と R をつくり出す演算子 $\circ$ を**除算（*division*）**と呼ぶ．

$$\begin{aligned} Q, R &= F \circ P \\ F &= P \cdot Q + R \end{aligned}$$

結果として得られる積和形論理式 Q を**商**，R を**余り**と呼ぶ．

$P \cdot Q$ が代数的な積であるとき，演算 $\circ$ を**代数的な除算（*algebraic division*）**と呼ぶ．そうでなければ**論理的な除算（*Boolean division*）**と呼ぶ．□

一般に，除算の問題は，

- 除数の候補を選ぶ．
- その除数を用いて除算を行う．

という二つの段階に分けて考えることができる．ここでは，代数的な除算の一つである **弱い除算（weak division）**について説明する．

定義 6 次のような除算を**弱い除算（*weak division*）**と呼ぶ．ただし，F は入力となる元の積和形論理式，P は除数となる積和形論理式，Q は結果として得られる商，R は結果として得られる余りである．

- $P \cdot Q$ が代数的な積である．
- R は積項数が最小の積和形論理式である．
- $P \cdot Q + R$ と F は同一の積和形論理式である（積項集合として等価）．

□

与えられた積和形論理式 F と P に対して，弱い除算によって求められる Q と R は一意に定まる．商 Q を表すのに F/P がよく用いられる．

弱い除算を行うには，F を P の各積項 p_i で割り，その共通部分を集めればよい．アルゴリズムを図 2･11 に示す．

```
WEAK_DIV(F, P) {
  foreach p_i ∈ P {
    V^{p_i} = φ
    foreach f_j ∈ F {
      if (f_j が p_i のすべてのリテラルを含んでいる) {
        v_ij = f_j から p_i のリテラルを取り除いたもの
        V^{p_i} = V^{p_i} ∪ v_ij
      }
    }
  }
  Q = ⋂_i V^{p_i}
  R = F \ PQ
  return(Q, R)
}
```

図 2･11　弱い除算アルゴリズム

2-3-3 カーネルとコカーネル

代数的な除算（弱い除算）を定義することによって，積和形論理式に対する**カーネル（kernel）**という概念を定義することができる．カーネルは積和形論理式中に現われる共通な部分論理式であり，以下のように定義される．

定義 7 積和形論理式中の全ての積項を余りなく割ることができる共通の積項がない積和形論理式 F を**キューブ・フリー（*cube-free*）**という*．□

$ab + c$ はキューブ・フリーであるが，$ab + ac$ はキューブ・フリーではない．また，一つの積項だけからなる積和形論理式はキューブ・フリーとはいわない．

* 積項（product）を別名でキューブ（cube）と呼ぶことに起因する

定義 8　積和形論理式 F をある積項 C で割った結果の式を**主除数（*primary divisor*）**と呼ぶ．つまり，主除数の集合 $D(F)$ は以下のように表される．

$$D(F) = \{F/C \mid C \text{ は積項 }\}$$

積和形論理式 F の主除数のうち，キューブフリーである式を F の**カーネル**と呼ぶ．F のカーネルの集合 $K(F)$ は以下のように表される．

$$K(F) = \{G \mid G \in D(F) \text{ かつ } G \text{ はキューブ・フリー }\}$$

□

カーネル $K = F/C$ を得るのに用いられる積項 C は K の**コカーネル（co-kernel）**と呼ばれる．また，F に対するコカーネルの集合を表すのに $C(F)$ が用いられる．

定義 9 自分自身以外にカーネルを含まないカーネルをレベル 0 のカーネルと呼ぶ．自分自身以外にはたかだかレベル $L-1$ のカーネルしか含まないカーネルをレベル L のカーネルと呼ぶ．□

コカーネルのレベルはそれの導出するカーネルのレベルによって定められる．つまり，レベル L のカーネルを導出するコカーネルのレベルは L となる．

$$\begin{aligned} F &= adf + aef + bdf + bef + cdf + cef + bfg + h \\ &= (a+b+c)(d+e)f + bfg + h \\ &= ((a+c)(d+e) + b(d+e+g))f + h \end{aligned}$$

に対するカーネル，コカーネルは表 2･1 のようになる．

表 2･1　カーネルとコカーネルの例

kernel	co-kernel	level
d + e	af, cf	0
d + e + g	bf	0
a + b + c	df, ef	0
(a + b + c)(d + e) + bg	f	1
((a + b + c)(d + e) + bg)f + h	1	2

$F/a = df + ef$ は主除数であるがキューブ・フリーではない．一つのカーネルを導出するコカーネルが二つ以上ある場合や，空集合の積項（全入力空間を論理的に包含する）がコカーネルになる場合もあることに注意（この場合，元の式がキューブ・フリーである必要がある）．

2-3-4 カーネルとコカーネルの計算

$\mathcal{C}^*(F)$ を積和形論理式 F 中の二つ以上の積項の共通部分を集めた積項集合とする．$\mathcal{C}^*(F)$ の各々の積項 C_i に対して，$K_i = F/C_i = \text{WEAK\_DIV}(F, C_i)$ を定義する．すると，C_i は F のコカーネルであり，K_i は対応するカーネルとなる．$\mathcal{C}^*(F)$ は F のレベル 0 のカーネルをすべて含んでいる（より高位のレベルのカーネルも含む可能性がある）．

例えば，

$$F = abcd + abce + efg = abc(d + e) + efg = abcd + e(abc + fg)$$

なら

$$\mathcal{C}^*(F) = \{abc, e\}$$

となる．対応するカーネルは $d + e$ と $abc + fg$ となる．この例ではレベル 0 のカーネルしかないが，$F = abcd + abce + abde$ の場合，二つの積項の共通部分を考えるとレベル 0 のコカーネル $\{abc, abd, abe\}$ を得ることができ，三つの積項の共通部分を考えるとレベル 1 のコカーネル $\{ab\}$ を得ることができる．

カーネル及びコカーネルを機械的に列挙するのに積項交差表（cube intersection table）が用いられる．これは与えられた積和形論理式中の積項どうしの共通部分（intersection）を計算するもので，最初に二つの積項の共通部分を求め，それらを新たに表に追加し，それらとほかの積項との共通部分を求める．このような処理を変化がなくなるまで繰り返す．

2-3-5 ファクタリングアルゴリズム

与えられた論理式から，それと同一の論理関数を表すより簡潔な論理式（ファクタードフォーム）をつくり出す処理をファクタリングと呼ぶ．ここまでに述べてきた除算処理と除数を求める処理を組み合わせて，ファクタリングアルゴリズムの構築を行う．

FACTOR(F, DIVISOR, DIVIDE) **{**
  if (F has no factor) **return**(F)
  D = DIVISOR(F)
  (Q, R) = DIVIDE(F, D)
  return (FACTOR(Q)FACTOR(D) + FACTOR(R))
}

図 2·12　ファクタリングアルゴリズム

DIVISOR 及び DIVIDE は変数ではなく，プロシージャ（関数）を指していることに注意．与えられた積和形論理式 F に対して DIVSOR(F) は除数の候補を見つける．次に，DIVIDE(F, D) を用いて商 Q(及び R) が求められる．この時点で，論理式 F は $F' = Q \cdot D + R$ のかたちにファクタリングされる．あとはこの部分論理式 Q，D，R に対して再帰的に FACTOR を適用してゆく．ここで紹介するアルゴリズムはすべてこのようなトップダウン手法を用いている．

与えられた積和形論理式に対して，そのカーネル，コカーネル，単一のリテラルを返すような関数を DIVISOR とし，WEAK_DIV を DIVIDE とすることもできるが，FACTOR は汎用なので論理的な除数を求める DIVSOR 関数や論理的な除算を行う DIVIDE 関数を用いることも可能である．この図 2·12 で示したファクタリングアルゴリズムは単純なため，明らかに「悪い」ファクタリングを行ってしまう例が多く存在する．そのため，改良版のアルゴリズムが提案されている．詳細は文献 2) を参照のこと．

2-3-6 共通式の抽出

ファクタリングは一つの節点内部の論理関数を効率良く表す論理式を求めるものである（いわゆる局所的最適化処理）が，抽出（extraction）はブーリアンネットワーク全体の構造を変更する大局的な最適化処理の一つである．

定理 1（基本定理）．二つの積和形論理式 F と G に対して，そのいかなるカーネル $K_F \in K(F)$，$K_G \in K(G)$ も，二つ以上の積項からなる共通な部分論理式をもたないとき，すなわち，$|K_F \cap K_G| \leq 1$ のとき，F と G は（二つ以上の積項からなる）共通部分論理式をもたない．

つまり，二つの積和形論理式の共通な部分論理式を求めるためには，その二つの積和形論理式のカーネルの共通な部分論理式を求めればよいことが分かる．また，カーネルに共通な部分論理式がなければ，一つの積項からなる除数の候補集合を求め，同様に共通な積項の集合を求めればよい．この定理を用いた処理がカーネルの共通部分を用いた抽出（kernel intersection extraction）で，矩形被覆問題として定式化されている[3, 4]．

例を用いて説明を行う．

$$\begin{aligned} F &= af + bf + ag + cg + ade + bde + cde \\ G &= af + bf + ace + bce \\ H &= ade + cde \end{aligned}$$

という論理式（ブーリアンネットワーク）に対するカーネル／コカーネルを求めることにしよう．まず，F に対するカーネル／コカーネルは，$(de + f + g)/a$，$(de + f)/b$，$(a + b + c)/de$，$(a + b)/f$，$(de + g)/c$，$(a + c)/g$ となる．同様に，G に対しては，$(ce + f)/(a, b)$，$(a + b)/(f, ce)$，H に対しては $(a + c)/de$ を得る．これを元に，表 2･2 に示すような，コカーネルを行に，カーネルの各キューブを列に対応させた行列をつくる．このような行列をコカーネル・キューブ行列（co-kernel cube matrix）と呼ぶ．

行列中の要素の数字は元の積和形論理式中の積項（キューブ）に対応している．ここでは F の最初の積項 af を 1 番，次の積項 bf を 2 番，というふうに番号付けをしており，その積項番号が行列の要素の値として書き込まれている．同一の積項が異なるカーネル／コカーネルに用いられていることもあるので，同じ番号が異なる要素に現われている場合もあることに注意．ここで，i 行 j 列の要素の値を B_{ij} で表すものとする．表中の'.' は値 0 を表している．

この行列における矩形（rectangle）とは，次のような行と列の集合 (R, C) のことである．

$$\forall i \in R, \forall j \in C, B_{ij} \neq 0$$

明らかに，コカーネル・キューブ行列上の矩形とはカーネルの共通部分にほかならない．そこで，このような行列上の矩形の中で「良い」ものを選び出して，その矩形に対応する共通部分式を用いてブーリアンネットワークを簡単化すればよい．例えば，表 2･2 の例で $(\{3, 4, 9, 10\}, \{1, 2\})$ という矩形を選んだとすると，対応する共通部分式は $a + b$ となり，この矩形の被覆している要素に対応した積項がそれぞれ簡単化され次のようになる．

表 2·2　コカーネル・キューブ行列

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	.	.	.	.	5	1	3
F	b	2	.	.	.	.	6	2	.
F	de	3	5	6	7	.	.	.	.
F	f	4	1	2	.	.	.	.	.
F	c	5	.	.	.	.	7	.	4
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	10	.	8	.
G	b	8	.	.	.	11	.	9	.
G	ce	9	10	11	.	.	.	.	.
G	f	10	8	9	.	.	.	.	.
H	de	11	12	.	13	.	.	.	.

$$
\begin{aligned}
F &= fX + ag + cg + deX + cde \\
G &= fX + ceX \\
H &= ade + cde \\
X &= a + b
\end{aligned}
$$

X が新たに追加された共通部分式である．この後，処理を続けるためにはコカーネル・キューブ行列にこの変更を反映する必要がある．具体的には新しい論理式 X に関する行の追加と既に被覆された要素の削除が行われる．

どのような矩形を選べば良いかの評価尺度は次のように与えられる．まず，行列の非ゼロの各要素は前述のように元のブーリアンネットワークの節点の積和形論理式中の積項に対応しているので，その積項のリテラル数を価値（value）としてもつ．要素 B_{ij} の価値を V_{ij} で表す．また，各々の行及び列もそれぞれ積項に対応しているので，そのリテラル数をコストとしてもつ．i 行目のコストを w_i^r で，j 列目のコストを w_j^c で表す．矩形にもコストが定義される．矩形のコストとはその矩形に対応する共通部分式を新たに追加したときに増えるリテラル数である．具体的には矩形 (R, C) のコスト $w(R, C)$ は，

$$
w(R, C) = \sum_{i \in R} w_i^r + \sum_{j \in C} w_j^c
$$

で与えられる．そこで，最終的に矩形 (R, C) の価値 $v(R, C)$ は，

$$
v(R, C) = \sum_{i \in R, j \in C} V_{ij} - w(R, C)
$$

となる．つまり，矩形がカバーする要素に対応する積項が無くなるのでその要素の価値分のメリットから新たに追加されるリテラル数分のデメリットを引いたものがその矩形の価値となる．

■参考文献

1) R.K. Brayton, R.L. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, “MIS: A Multiple-Level Logic Optimization System,” IEEE transactions of CAD, vol.CAD-6, no.6, pp.1062-1081, Nov. 1987.
2) G.D. Hactel, and Fabio Somenzi, “Logic Synthesis and Verification Algorithms,” Kluwer Academic Publishers, 1996.
3) R. Rudell, “Logic synthesis for VLSI design,” Ph.D. thesis, University of California, Berkeley, 1989.
4) R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, “Multi-level logic optimization and the rectangle covering problem,” In Proceedings of ICCAD-87, pp.66-69, Nov. 1987.

■1 群 - 8 編 - 2 章

2-4 基本的な組合せ回路

（執筆者：高木直史）[2009 年 4 月 受領]

いくつかの基本的な組合せ回路の構成例を示す．

2-4-1 デコーダとエンコーダ

m ビットデコーダは m 個の入力と 2^m 個の出力をもち，入力が 2 進数とみなして k であれば，k 番目の出力のみを 1 とし，ほかを 0 とする．図 2･13 に 2 ビットデコーダの回路例を示す．

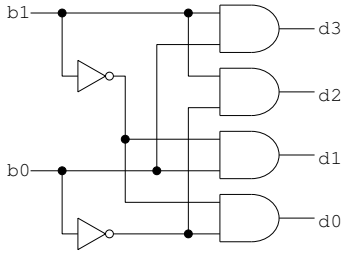

図 2･13　2 ビットデコーダの回路例

エンコーダはデコーダと反対の機能をもつ．2^m-m エンコーダは 2^m 個の入力と m 個の出力をもち，通常，入力のうち一つだけが 1 でほかは 0 であり，k 番目の入力が 1 のとき，k の 2 進数表現を出力する．図 2･14 に 4-2 エンコーダの回路例を示す．

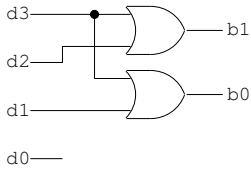

図 2･14　4-2 エンコーダの回路例

入力のうち二つ以上が 1 である場合に，1 である入力のうち，例えば最も小さな入力番号を出力するプライオリティエンコーダもある．

2-4-2 マルチプレクサ（セレクタ）とデマルチプレクサ

マルチプレクサ（セレクタ）は，二つのデータ入力と一つの制御入力，一つの出力をもち，制御入力の値により，二つのデータ入力のいずれかの値を出力する．図 2･15 にマルチプレクサの回路例を示す．図の回路では，出力 z は，制御入力 s が 0 のとき x，1 のとき y の値となる．すなわち，$z = \bar{s}x + sy$ である．

図 2･16 に示すように，マルチプレクサ（MUX）を n 個並列に並べて制御入力を共有させることにより，制御入力信号の値により二つの n ビットデータの一方を選択する，n ビットマルチプレクサを構成できる．

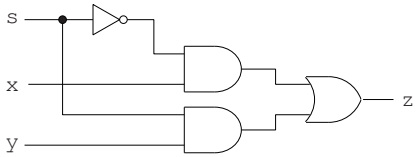

図 2·15　マルチプレクサの回路例

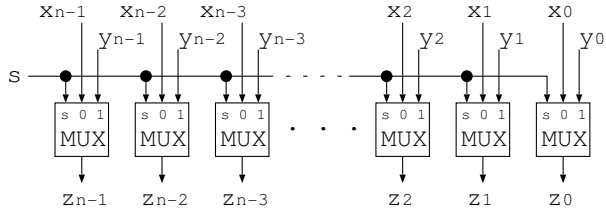

図 2·16　n ビットマルチプレクサの構成

図 2·17 に示すように，マルチプレクサを二分木状に接続することにより，2^m-1 マルチプレクサを構成できる．2^m-1 マルチプレクサは，2^m 個のデータ入力と m 個の制御入力，一つの出力をもち，制御入力が 2 進数とみなして k であれば，k 番目のデータ入力の値を出力する．図は，8-1 マルチプレクサである．

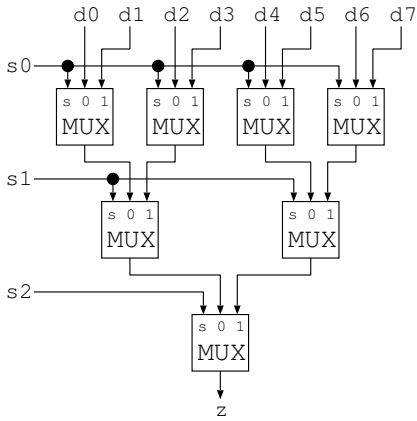

図 2·17　8-1 マルチプレクサの構成

デマルチプレクサは，マルチプレクサと反対の機能をもつ回路である．一つのデータ入力と一つの制御入力，二つのデータ出力をもち，制御入力の値により，二つのデータ出力のいずれかに，データ入力の値を出力する（ほかの出力は 0 である）．図 2·18 にデマルチプレクサの回路例を示す．図の回路では，制御入力 s が 0 のとき出力 y に，1 のとき z に，データ入力 x の値が出力される．すなわち，$y = \bar{s}x$，$z = sx$ である．

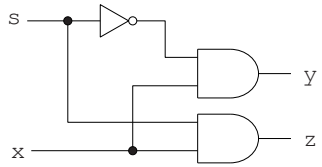

図 2·18　デマルチプレクサの回路例

2-4-3 加算器

2 進数 1 桁の加算について考える．入力 x，y から，和 z と桁上げ c_{out} を求める回路を半加算器（half adder：HA）という．$z = x \oplus y$，$c_{out} = xy$ である．図 2·19(a) に半加算器の回路例を示す．

下位からの桁上げ入力 c_{in} が加わり，入力 x，y，c_{in} から，和 z と桁上げ c_{out} を求める回路を全加算器（full adder：FA）という．$z = x \oplus y \oplus c_{in}$，$c_{out} = xy + xc_{in} + yc_{in}$ である．図 2·19(b) に全加算器の回路例を示す．図中で破線で囲った部分（二つ）は半加算器である．

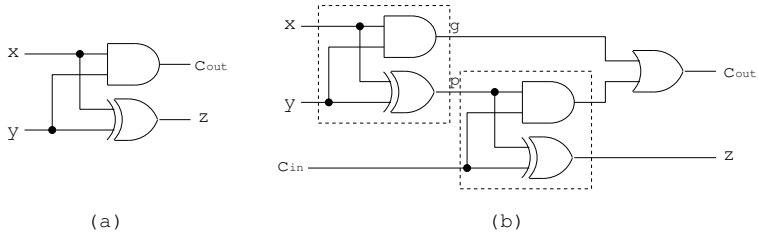

図 2·19　半加算器と全加算器の回路例　(a) 半加算器　(b) 全加算器

図 2·20 に示すように，一つの半加算器と $n-1$ 個の全加算器を直列に接続することにより，n ビット加算器を構成できる．半加算器及び各全加算器の桁上げ出力が，一つ上位の桁の全加算器の桁上げ入力になる．このような加算器を順次桁上げ加算器という．

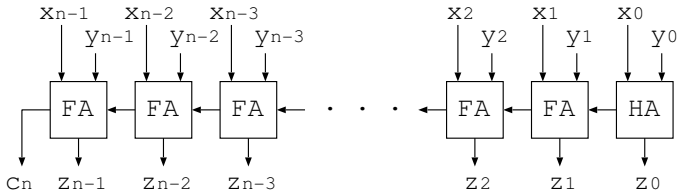

図 2·20　順次桁上げ加算器の構成

加算を二つのブロックに分け，上位のブロックで下位のブロックからの桁上げが 0 である場合と 1 である場合の両方について和を計算しておき，下位ブロックからの桁上げが確定した時点でいずれかを選択するようにすれば，計算を高速化できる．選択にはマルチプレクサ

を用い，下位ブロックからの桁上げを制御入力とすればよい．この考えに基づくのが，桁上げ選択加算器である[1)]．より多くのブロックに分けることにより，さらに高速化が可能である．

加算において，ある桁を下位からの桁上げが通過して上位に伝搬するのは，その桁の二つの演算数 x と y の一方が 0 で他方が 1 のとき，すなわち，$p = x \oplus y$ が 1 のときである．連続する桁のブロックを考えると，そのブロック内のすべての桁で p が 1 であれば，下位からの桁上げがそのブロックを通過して上位に伝搬する．そこで，計算をいくつかのブロックに分け，各ブロックでこの桁上げ伝搬条件が成立するかどうかを求めておき，条件が成立する場合，下位からの桁上げがそのブロックを飛び越して上位に伝搬するようにするのが，桁上げ飛び越し加算器である[2)]．桁上げ飛び越し加算器は，順次桁上げ加算器に桁上げ飛び越し回路を付加した構成になる．図 2･21 に，4 ビット桁上げ飛び越し加算ブロックの構成を示す（図では，各桁で p は全加算器の中で計算されているものを利用するものとしている）．

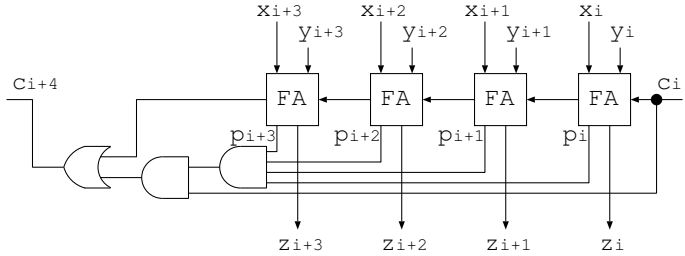

図 2･21　4 ビット桁上げ飛び越し加算ブロックの構成

加算の第 i 桁での計算について考える．演算数入力を x_i，y_i，下位からの桁上げを c_i，上位への桁上げを c_{i+1} とする．$g_i = x_i y_i$ が 1 のとき桁上げが生成され，$p_i = x_i \oplus y_i$ が 1 のとき下位からの桁上げが伝搬する．したがって，$c_{i+1} = g_i + p_i c_i$ である．これに $c_i = g_{i-1} + p_{i-1} c_{i-1}$ を代入すると，$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1}$ となり，更に代入を繰り返すと，

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_1 g_0 \qquad (2\text{･}1)$$

となる．すなわち，各桁の桁上げは，その桁と下位の桁の演算数入力から計算できる．この考えに基づき，桁上げを高速に計算するのが桁上げ先見加算器である．図 2･22 に 4 ビット桁上げ先見加算器の回路例を示す．

式 (2･1) の桁上げの計算をすべての桁で個別に行う純粋な桁上げ先見加算器は，ハードウェア量の観点から，数桁程度のものしか実現が困難である．そこで，計算を数桁ずつのブロックに分けて行うことを考える．各ブロックで，ブロックへの桁上げ入力がないものとして加算を行うと，最上位からの桁上げ出力は，そのブロックで生成される桁上げを示している．j 番目のブロックの桁上げ生成条件を G_j，桁上げ伝搬条件（桁上げ飛び越し加算器で考えたもの）を P_j，上位への桁上げを C_{j+1} とすると，

$$C_{j+1} = G_j + P_j G_{j-1} + P_j P_{j-1} G_{j-2} + \cdots + P_j P_{j-1} \cdots P_1 G_0 \qquad (2\text{･}2)$$

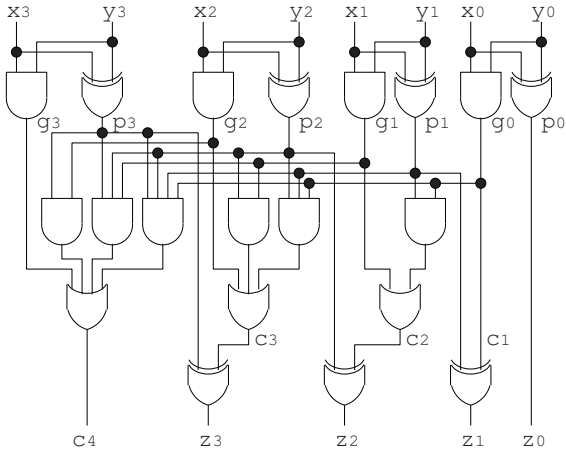

図 2·22　4 ビット桁上げ先見加算器の回路例

となる．これは，式 (2·1) と全く同じである．したがって，数桁の桁上げ先見加算ブロックを木状に接続することにより，多ビットの桁上げ先見加算器を構成できる．

2-4-4 比較器

符号なし 2 進整数 X と Y の大小比較を行い，$X > Y$ のときに，出力 c を 1 とする比較器について考える．最下位桁から順次，各桁で x_i，y_i と下位での比較の結果 c_{in} から，比較結果 c_{out} を計算する．最上位での比較結果が比較器の出力となる．各桁では，$x > y$ ならば c_{out} を 1，$x < y$ ならば 0，$x = y$ ならば，c_{in} の値とすればよい．すなわち，$c_{out} = x\bar{y} + xyc_{in} + \bar{x}\bar{y}c_{in} = x\bar{y} + xc_{in} + \bar{y}c_{in}$ である．図 2·23 に 1 ビット比較器の回路例を示す．順次桁上げ加算器と同様，この回路を n 個直列に接続することにより，n ビット比較器を構成できる．ただし，最下位への c_{in} 入力を 0 とする．

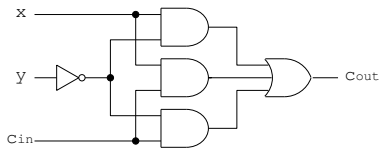

図 2·23　1 ビット比較器の回路例

■参考文献

1) O.J. Bedrij, “Carry-select adder,” IRE Trans. Elec. Comput., vol.EC-11, pp.340-346, Jun. 1962.

2) M. Lehman and N. Burla, “Skip techniques for high-speed carry propagation in binary arithmetic units,” IRE Trans. Elec. Comput., vol.EC-10, pp.691-698, Dec. 1961.