論理回路

目次

第 1 章

準備

1.1 n 進数

1.1.1 値

基底が n，各項の値が $a_0, a_1, a_2, \cdots$ となる整数は 10 進数で以下のように表記される．

$$\sum_{k=0}^{l} a_k n^k \tag{1.1}$$

10 進数は $n = 10$ の場合である．小数については計算結果の k のスタートが負の数まで拡張されればよいことがわかる．

1.1.2 表記

n 進数は abc_n のように基底が右下につくことで表される．10 進数は書かないこともある．

1.2 基底の変換

1.2.1 10 進数から

整数部と小数部で分けるとよい．整数部に対しては変更先の基底 n で元の数を割り，出てきた商をさらに n で割り続けるという計算を余りを記録しながら商が 0 になるまで続ける．割り算が終われば記録した余りを出てきた順番に左から並べれば完成する．

小数部に対しては変更先の基底 n をかける．その時の 1 の位の数を記録し，また小数のみに n をかける．これを小数がなくなるまで繰り返し，記録した数を小数点以下に順番に並べれば完成．

1.2.2　10 進数へ

式 (1.1) に基底 n を代入.

1.2.3　2 進数から 8 進数・16 進数

2 進数を 8 進数に変換するとき，2 進数の数字を 1 の位から 3 つずつ区切り，その区切った要素ごとに 10 進数の変換をすればよい．16 進数への変換では，区切る間隔を 4 つずつにすればよい．いかに例を示す.

$$101111101011001_2 \to 101, 111, 101, 011, 001 \to 57531_8$$

$$101111101011001_2 \to 101, 1111, 0101, 1001 \to 5\mathrm{F}59_{16}$$

1.2.4　8 進数・16 進数から 2 進数

8 進数を 2 進数に変換するとき，8 進数 1 桁の数を 2 進数に変換し，それを 2 進数における 3 桁分にすればよい．16 進数からの場合は 4 桁分にすればよい.

$$7323_8 \to 7, 3, 2, 3 \to 111, 011, 010, 011 \to 111011010011$$

$$\mathrm{ACE}_{16} \to \mathrm{A}, \mathrm{C}, \mathrm{E} \to 1010, 1100, 1110 \to 101011001110$$

第 2 章

論理代数と論理関数

2.1　論理代数

詳しくは配布プリント参照

論理関数　n 個の論理変数から 1 つの論理変数を返す関数.

論理代数　$\{0, 1\}$ 上の代数系．OR(論理和)，AND(論理積)，NOT(論理否定) の 3 つの論理演算を持つ．計算の優先度は算術演算と同じ.

表 2.1　論理代数の性質 (一部)

べき等律	$x+x=x$	$x\cdot x=x$
分配律	$x\cdot(y+z)=x\cdot y+x\cdot z$	$x+y\cdot z=(x+y)\cdot(x+z)$
相補律	$x+\bar{x}=1$	$x\cdot\bar{x}=0$
吸収律[*1]	$x+x\cdot y=x$	$x\cdot(x+y)=x$

2.2　論理代数の持つ性質

2.2.1　算術演算からでは想像できない性質

すべては配布プリント参照

2.2.2　双対性・双対原理

論理代数で成立した等式は 1 と 0，+ と · を置き換えても成立する．

2.3　論理式

2.3.1　語句説明

リテラル　論理変数とその否定

積項　¯も含めて各論理変数が多くとも 1 回しか出てこない論理積

ex)$x\bar{y}z, ab$ は積項．$x\bar{x}, yy$ は積項ではない．

積和形論理式　積項の論理和の式[*2]

和項　¯も含めて各論理変数が多くとも 1 回しか出てこない論理和

ex)$x+\bar{y}+z, a+b$ は和項．$x+\bar{x}, y+y$ は和項ではない．

和積形論理式　和項の論理積の式[*3]

2.3.2　ド・モルガンの定理の機能

ド・モルガンの定理では積和形論理式と和積形論理式を行ったり来たりする作用がある．

2.4　論理関数の表現

真理値表は全入力に対する出力を列挙するわけだが n 入力の場合 2^n 列の表を描かなければならず大変．そのため出力 0,1 になる入力の集合に分けて表現するオンセット (オフセット) 表現がある．以下のように定義

[*1] 便利なので覚えよう

[*2] いくつかの項に同じ論理変数のリテラルかあってもよい．

[*3] いくつかの因数に同じ論理変数のリテラルがあってもよい．

されている．

$$\mathrm{onset}(f) := \{(x_1, x_2 \cdots x_n) \in B^n | f(x_1, x_2 \cdots x_n) = 1\} \tag{2.1}$$
$$\mathrm{offset}(f) := \{(x_1, x_2 \cdots x_n) \in B^n | f(x_1, x_2 \cdots x_n) = 0\} \tag{2.2}$$

特にオンセット表現に対してはその入力 $(x_1, x_2 \cdots x_n)$ を 2 進数の整数に見立て，この入力を 10 進数に直したうえで

$$f(x_1, x_2 \cdots x_n) = \sum(a, b, c, \cdots) \tag{2.3}$$

と表現できる．

2.4.1 完全定義・不完全定義

完全定義論理関数

入力値のすべての組み合わせについて関数値が定まっている論理関数．

不完全定義論理関数

入力値のいくつかの組み合わせについて関数値が定まっていない論理関数．

例えば 4 入力の論理関数では 16 の出力があるが，必要な出力が 10 のみの場合があったりする．この場合不要な 6 の出力に対しては任意な結果を選ぶことができる．これらをドントケアセットと呼ぶ．式 (2.3) の表現を不完全定義論理関数に拡張すると，

$$f(x_1, x_2 \cdots x_n) = \sum(a, b, c, \cdots) + \sum{}_{dc}(a', b', c', \cdots) \tag{2.4}$$

2 項目がドントケアセットの入力を表している．

2.5 論理関数を論理式で表現

2.5.1 最小項・最大項

論理関数の全入力に対して最小項，最大項の組が 1 つ与えられる．最小項と最大項は否定の関係があり，一方を求められれば他方は計算できる．ある入力に対する最小項とはその論理関数の各変数のリテラルを 1 つずつ含んだ積項で，かつその入力では 1 を返すものである．最大項はこの否定を考えればよい．

2.5.2 標準形

積和標準形　最小項展開，加法標準形とも．

オンセットに対応した最小項のすべての論理和で求められる．

和積標準形　乗法標準形とも

オフセットに対応した最大項のすべての論理積で求められる．しかし，最大項の導出などに失敗が多く，求める関数の否定に対して積和標準形を求め，再度否定した方が安全に導出できる．

一般に簡単化は積和標準形のことを指す．もちろん和積標準形が必要になることもあるので両方の十分な理解が必要．

2.5.3 シャノン展開

入力数を 1 つ減らすことを目的とした式変形．n 入力の論理関数 f 対し以下の式変化が成立する．

$$f(x_1, \cdots, x_r, \cdots, x_n) = \bar{x}_r f(x_1, \cdots, 0, \cdots, x_n) + x_r f(x_1, \cdots, 1, \cdots, x_n) \tag{2.5}$$

シャノン展開を繰り返すと積和標準形が得られる．当然するわけがない．

第 3 章

論理関数の簡単化

3.1 簡単化とは

任意の完全定義論理関数は積和形論理式を与えることができるがこれは複数存在する．その一つの論理関数を表す積和形論理式のうち，積項数が最も少なくかつ積項に含まれるリテラルが少なくなっているものを最小 (最簡) 積和形表現という．これ自体も複数存在する．論理関数の簡単化とはこの最小積和形論理関数を導出することである．指定がなければ 1 つ求めればよい．

当然この簡単化は和積形論理式にも同様のことが言える．

3.2 簡単化の方法

3.2.1 Boolean Cube

方法に入る前に以下の語句を理解する．

隣接辺　辺の両端の点はその辺に対応する 1 つの変数の値のみ異なる．
隣接面　四角形の 4 頂点はその面に対応する 2 つの変数のみ異なる．
隣接立法　立方体の 8 頂点はその立方体対応する 3 つの変数値のみ異なる．

では早速簡単化にうつる．

1. Boolean Cube を作図し，最小項にあたる頂点を明示する．
2. 明示した頂点に対し隣接辺や，隣接面の関係にある頂点の組を極力高次元のもので探し包含する．この時どうしても 1 頂点のみになるものがあっても構わない．
3. 最後に包含した頂点の組全体を積項で表し，その論理和を求める．

原理としては，2 段階目で作った最小項の組は同じ変数の入力が異なる状態のものを含んでいながらみなオンセットであるから，その入力が異なる変数を不問にできる．するとその最小項の組を表す積項のリテラルが減

ることになる．そして最小項の組の個数自体が少なければ最小積和論理関数の条件に沿う論理式が導ける．

最後にこの一連の操作で現れる語句を示す．論理関数 f，f に現れるリテラルのみの積項 P とする．

包含　P を 1 とするすべての入力で f が 1 ならば P は f に包含されているという．
内項　f に包含されている P は f の内項である．
主項　他の内項に包含されていない内項
必須主項　f のある最小項 m がただ 1 つの主項に含まれている時，その主項を必須主項という．

この語句の定義から最小積和形表現は主項の和であり，かつすべての必須主項を含むものだとわかる．

3.2.2　カルノー図

カルノー図とは n 次元立方体の展開図である．具体的な図は配布資料を参照すればよいが何度か自ら作成してなれる必要がある．

それでは簡単化に移る．

1. カルノー図を作図しオンセットを明確にする．
2. すべてのオンセットの要素に対し隣接辺や隣接項などの最大グループ (主項) を求める．
3. 必須主項を探し，まずその主項を確定させる．
4. 残ったオンセットの要素を包含するできるだけ少ない主項を選択する．すると選んだすべての主項の和は最小積和表現になっている．

主項の見つけ方は配布資料を参照.

カルノー図の作図において厄介なのは排他的論理和であるが，この場合は前段階のカルノー図を複数かいて，そのカルノー図を比べながら排他的論理和の計算を地道に行ってかけば何とかなる．

3.2.3　クワイン・マクラスキー法

これはあくまで計算機に行わせる方法である．方法の理解で十分と考えられる．以下がその手順である．

1. **主項の生成**: 最小項から初めて積項の併合を繰り返す
   まず積項を以下のように”キューブ表現[*1]”で表す．

$$a\bar{b}c\bar{d} \to 1010$$
$$a\bar{c} \to 1-0-$$

   積項の併合は以下のような計算を行い隣接を表す．

$$\begin{array}{cc} abcd & 1111 \\ a\bar{b}cd & 1011 \\ \downarrow & \\ a-cd & 1-11 \end{array}$$

   1 か所のみ異なる場合隣接してるので併合できる．併合の原理を考えるとビットの数が 1 違う積項の間でしか起こらないため，まず最小項をすべて挙げたら，ビット数で分類して差が 1 つのもの同士だけ併

[*1] 積項を 01 のベクトル表現すること．

合が可能か判定する．この時併合されたものは主項になり得ないことに注意する．さらに併合した先の隣接辺に対しても同様の併合を行い，併合できなくなるまで続ける．併合されなかったもの，もう併合ができないものが主項となる．

2. **主項による関数の被覆**:主項表を作って効率的に解く
   主項表は横軸を最小項 (10 進数表現)，縦軸を先の操作で導出した主項として主項に対しその内項となっている場所に印をつけた表のこと．
   この表からはまず容易[*2]に必須主項がわかる．まずそれらを選択し，それらによって被覆されなかった最小項を被覆する主項を抽出する．この後に非支配行を除去，支配列を除去することを繰り返し，主項を減らし選択を続けることで，選択した主項の和が最小積和表現となる．
   しかしこの操作は時として複雑を極めることがあり，このとき分枝限定，ペットリックの方法という 2 つの方法が存在する．ではまず，その複雑を極めるとはどのような状況なのかを述べる．
   端的に説明するとすでに述べた主項を除去する方法では何一つ決められない場合である[*3]．このような場合をサイクリックテーブルと呼ぶがこれに対しては直前に述べた 2 つの方法が有効である．

   分枝限定

   この方法は場合分けの同じような方法で，まずどの主項を選択するのかを任意で 1 つ選択しそこから上記の方法で簡単化を図る方法である．上手に最初選択する主項を決定できれば素早く簡単化できる．

   ペトリックの方法

   この方法は主項表完成時にサイクリックテーブルであると分かったら，主項表それぞれの主項に論理変数を割り当て[*4]どの主項を使用するのかということに対し論理関数を立てる[*5]．この論理関数自体を標準積和形で簡単化する．そこで現れる最もリテラルの少ない積項が選択すべき主項の組である．

3.3　複数の論理関数の簡単化

複数の論理関数の簡単化は何も複数個の論理関数を独立して簡単化しておしまいとしてはならない．ここでの簡単化の条件は，簡単化の後の複数の最小積和形表現全体で積項数と積項各自のリテラル数を最小にすることである．この時複数の最小積和表現の論理関数で被っている積項は 1 つとカウントされるため，ただ簡単化しては正しい結果ではない場合がある．

簡単化を通してわかるあれこれ

基本的にこの章で簡単化をした論理関数は AND，OR，NOT 論理演算子を用いた式がほとんどであった．このようなタイプの論理式は最小積和表現で簡単な式が導出できる．これに対して排他的論理和をもつ論理関数はあまり簡単にはならない．これは演算子の相性というものがあるからでこのため排他的論理和などの演算

[*2] 表の縦の列は最小項を包含している主項を表すものになっているので，印が 1 つの時に必須主項という判断ができる．
[*3] 教科書 P.53 からの例 3.17
[*4] その主項を使用するなら 1，使用しないなら 0.
[*5] 最小項は必ず包含しなければならないことを考えるとある最小項を包含する複数の主項から少なくとも 1 つは選ばなけらばならない．このことから各最小項を包含す主項の和項の積で式が立つ．

子による考え方もある．

第 4 章

論理関数の諸性質

4.1　論理演算と完全系

4.1.1　2 変数論理演算

2 変数の入力 2^2 通りに対し出力を 2 通りずつ与えるとその総数は 16 通り．この中にはすでに知っているものもあるので初めてのもの，記号を知らなかったものを以下の表 4.1 に示す．

表 4.1　新しく見る演算

演算名	記号	基本演算
恒偽	0	
抑止	$x - y$	$\bar{x} \cdot y$
NOR	$x \downarrow y$	$\overline{x + y}$
XNOR(同値)	$x \equiv y$	$\bar{x} \cdot \bar{y} + x \cdot y$
含意	$y \to x$	$x + \bar{y}$
恒真	1	

4.1.2　完全系

難しい定義は配布資料を読めばいいわけだが最も簡単に述べると，数個の論理演算だけですべての論理関数が表現できたらその論理演算の組は完全系であるという．組をなす論理演算はすでに学んでいるものや直前の表 4.1 の演算のことである．

すでに持っている知識として，{AND,OR,NOT} の組が完全系であることは知っている．これによってほかの演算の組み合わせが完全系か否かは {AND,OR,NOT} を表せるのかということに置き換わり，これなら十分に理解できる．

では以下に完全系の論理演算の組を紹介し，{AND,OR,NOT} をいかに実現しているのかを示す．

AND,NOT　以下に OR 演算を示す．

$$x + y = \overline{\overline{x + y}} = \overline{\bar{x} \cdot \bar{y}}$$

OR,NOT　以下に AND 演算を示す．

$$x \cdot y = \overline{\overline{x \cdot y}} = \overline{\bar{x} + \bar{y}}$$

NAND　以下に AND，OR，NOT 演算を示す．

$$\begin{aligned}\bar{x} &= \overline{x \cdot x} \\ x \cdot y &= \overline{\overline{x \cdot y}} = \overline{\overline{x \cdot y} \cdot \overline{x \cdot y}} \\ x + y &= \overline{\overline{x + y}} = \overline{\bar{x} \cdot \bar{y}} = \overline{\overline{x \cdot x} \cdot \overline{y \cdot y}}\end{aligned}$$

NOR　以下に AND，OR，NOT 演算を示す．

$$\begin{aligned}\bar{x} &= \overline{x + x} \\ x \cdot y &= \overline{\overline{x \cdot y}} = \overline{\bar{x} + \bar{y}} = \overline{\overline{x + x} \cdot \overline{y + y}} \\ x + y &= \overline{\overline{x \cdot y}} = \overline{\overline{x + y} + \overline{x + y}}\end{aligned}$$

1,AND,EXOR　以下に NOT，OR 演算を示す．

$$\begin{aligned}\bar{x} &= 1 \oplus x \\ x + y &= (x \oplus y) \oplus (x \cdot y)\end{aligned}$$

この演算系を用いた論理関数の表現法として，リード-マラー標準形というものがある．

4.2　n 項演算への拡張

1 つの論理変数に対する演算を単項演算，2 つの論理変数に対する演算は 2 項演算といいこれまでよく扱ってきたが，これらを n 項に対する演算に拡張する．

- n 項 OR 演算
  n 変数のうち少なくとも 1 つが 1 であれば，演算結果は 1 となる．
- n 項 AND 演算
  n 変数のうちすべてが 1 であるならば，演算結果は 1 となる．
- n 項 EXOR 演算
  n 変数のうち奇数個が 1 であるならば，演算結果は 1 となる．
- n 項 NOR,n 項 NAND 演算
  n 項 OR,n 項 AND 演算を論理否定すればよい．

4.3　論理関数の変数依存性

ある論理変数がどれほど出力に影響しているのかを確認したい．

4.3.1　論理関数の微分 (ブール微分)

さっそくではあるが以下の定義を覚えよう．

ブール微分

n 入力論理関数 $f(x_1, \cdots, x_n)$ に対して x_iによる微分を$\dfrac{\partial f}{\partial x_i}$とすると

$$\frac{\partial f}{\partial x_i} = f(x_1, \cdots, 0, \cdots, x_n) \oplus f(x_1, \cdots, 1, \cdots, x_n)$$

さらにこの微分の結果の解釈は

$$\frac{\partial f}{\partial x_i} = 0 \Rightarrow f \text{ は } x_i\text{に依存しない}$$
$$\frac{\partial f}{\partial x_i} = 1 \Rightarrow f \text{ は } x_i\text{に依存する}$$

ここで排他的論理和が

$$x \oplus y = x\bar{y} + \bar{x}y$$

であることを改めて示す.

4.3.2 論理関数における単調性とユネイト関数

微分が計算できたので次は単調性について述べる．単調性についての定義は以下のとおりである.

論理関数における単調性

n 入力論理関数 f に対し

単調増大

$$f(x_1, \cdots, 0, \cdots, x_n) \leq f(x_1, \cdots, 1, \cdots, x_n)$$

単調減少

$$f(x_1, \cdots, 0, \cdots, x_n) \geq f(x_1, \cdots, 1, \cdots, x_n)$$

さらにこのような単調性のある論理関数をユネイト関数という.

ここでさらっと論理関数の大小を表す式が現れているが，これは数学的な大小を表すのではなく，論理関数の包含関係を表している．すなわち，論理関数 f の任意のオンセットが別の論理関数 g のオンセットであるとき g は f を包含していて，$g \geq f$ と表すのである.

単調性の入り口が分かったところでその性質を考える．まずは単調増大関数についてである．単調増大関数であるということは上記の定義以外に論理関数 f が正のリテラルの積項のみで表すことができるということと同値である．いきなりそのように言われてもなんとも言えないので以下で証明する.

必要性

まず，単調増大であるので

$$
\begin{aligned}
& f(x_1,\cdots,0,\cdots,x_n) \leq f(x_1,\cdots,1,\cdots,x_n) \\
\Rightarrow\ & f(x_1,\cdots,1,\cdots,x_n) = f(x_1,\cdots,0,\cdots,x_n) + f(x_1,\cdots,1,\cdots,x_n)
\end{aligned}
$$

シャノン展開をすると

$$
\begin{aligned}
f(x_1,\cdots,x_r,\cdots,x_n) &= \bar{x}_r f(x_1,\cdots,0,\cdots,x_n) + x_r f(x_1,\cdots,1,\cdots,x_n) \\
&= (x_r + \bar{x}_r) f(x_1,\cdots,0,\cdots,x_n) + x_r f(x_1,\cdots,1,\cdots,x_n) \\
&= f(x_1,\cdots,0,\cdots,x_n) + x_r f(x_1,\cdots,1,\cdots,x_n)
\end{aligned}
$$

このように $\bar{x}_r$ のない論理式になる．この計算を全てのリテラルに対して行えば正のリテラルのみの積和標準形が導出できる．

十分性　n 変数論理関数 f に対し x_i のリテラル以外はすべて含む n-1 変数論理関数 g, h を用意すると f は以下のようにあらわすことができる．

$$
\begin{aligned}
f &= x_i g + h \\
&\text{ここで} \\
f|_{x_i=0} &= h \\
f|_{x_i=1} &= g + h \\
\therefore f|_{x_i=0} &\leq f|_{x_i=1}
\end{aligned}
$$

つまり f は単調増大である．またこれは f のすべてのリテラルに対しても同様であるので f に含まれるすべてのリテラルは正である．

では，単調減少論理関数であればどうなるのかであるが，それは当然論理関数 f が負のリテラルの積項のみで表すことができるというのが同値な内容となる．

4.4　論理関数の除算

論理関数において除算から導かれる商と剰余に一意性はない．しかし，剰余が恒偽ならば商を不完全定義論理関数として定めることができる．

実際の計算の実行であるがこれは剰余が恒偽である場合のみが出題されると考えよい．その場合以下の式をそのままカルノー図に書き換え，オンセットの位置を注意深く確認しながら q を決定すれば良い．

$$f = g \cdot q$$

第 5 章

組み合わせ論理回路

5.1　論理ゲートを実際の回路で表現

基本的な論理回路記号はすでにどこかで見たことがあるようなものばかりであり詳しくは扱わない．

5.1.1　CMOS トランジスタ

論理回路が実際の回路ではどのように実現されているのかを明らかにするべく，まずは論理回路の実現に使用される MOS トランジスタについて述べる．MOS トランジスタには図 5.1，図 5.2 で以降示される PMOS と NMOS の 2 種がある．

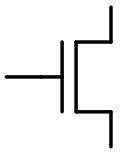

図 5.1　PMOS トランジスタ

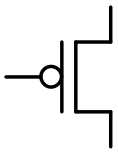

図 5.2　NMOS トランジスタ

PMOS トランジスタは入力電圧があるときに通電し，NMOS トランジスタは入力電圧がないときに通電する．これらを相補的に組み合わせて作る回路を CMOS という．では次は最も基本的な論理ゲートである AND，OR，NOT 回路を図 5.3 から図 5.5 で示す．示した回路から分かるのは NAND，NOR 回路が AND，OR 回路から NOT 回路の部分を消したものであるということである．つまり，NAND，NOR 回路の方が簡単な回路で実現できるということである．

5.1.2　ド・モルガンの定理を論理ゲートで表現

ド・モルガンの定理の最も基本的な 2 つの式を以下に挙げる．

$$\begin{aligned} a + b &= \overline{\overline{a + b}} \\ &= \overline{\bar{a} \cdot \bar{b}} \\ a \cdot b &= \overline{\overline{a \cdot b}} \\ &= \overline{\bar{a} + \bar{b}} \end{aligned}$$

これらの式を以下の 2 つの見方で見ると論理ゲートでの理解に近づく．

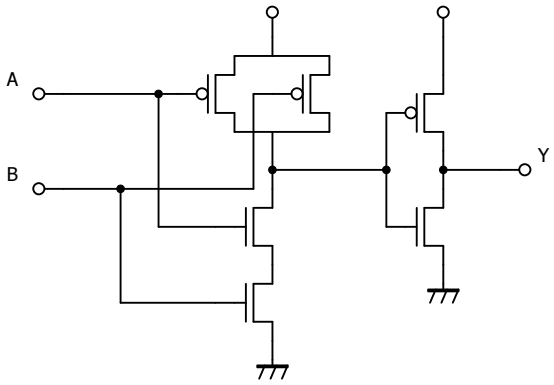

図 5.3　AND 回路

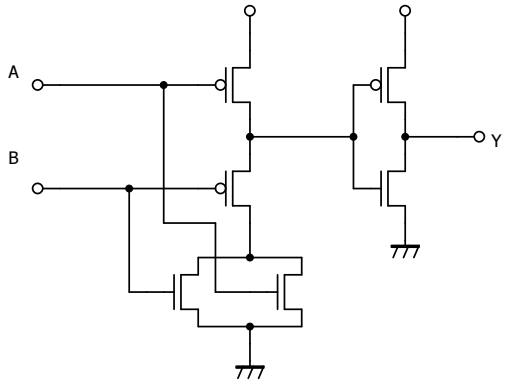

図 5.4　OR 回路

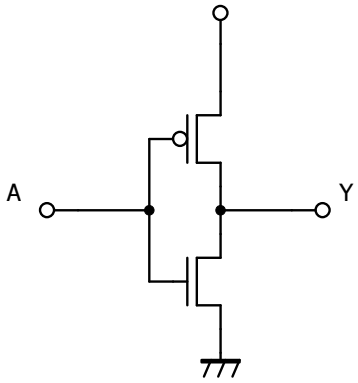

図 5.5　NOT 回路

- リテラル直上のバーは入力の否定と考える
- 式全体に作用するバーは出力の否定と考える

これらから AND 演算の二重否定では図 5.6 の対応が，OR 演算の二重否定では図 5.7 の対応が導かれる．この関係は直ちに思い出せるように覚えるとよい．

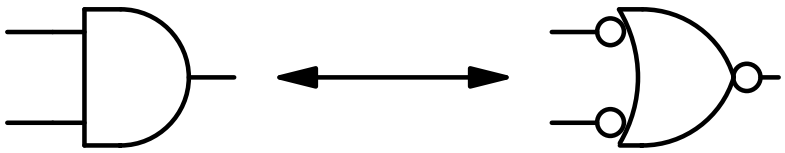

図 5.6　AND 演算の二重否定

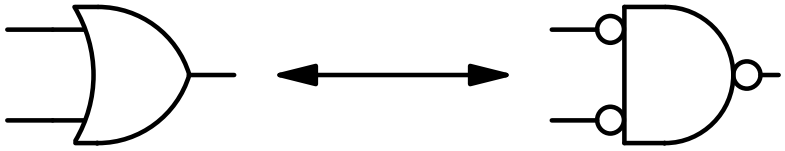

図 5.7　OR 演算の二重否定

5.2　組み合わせ論理回路とは

- n 入力，m 出力でそれぞれ 0 または 1 をとる．
- m 個の論理関数を計算する回路
- PO[*1]は PI[*2]の値に依存し一意
- 回路中に帰還路を含まない

5.3　論理式を組み合わせ論理回路で表現

標準形は 2 段論理で示すことができる．しかし，n 項の演算が多く出て実際の回路での実現は困難な論理回路が出るため，通常は簡単化したのち多段論理で表現する．

5.3.1　積和標準形・和積標準形

積和標準形を論理回路で表すとき，各積項を AND で表したのちに OR でまとめて出力とする．和積標準形なら各和項を OR で表したのち AND でまとめ出力する．

[*1] Priminary output
[*2] Priminary input

5.3.2　簡単化からの多段論理

積和か和積で簡単化し論理ゲートで表現．一体どちらの簡単化を使うのかは当然場合によるが，その判断基準は後に述べる．

5.4　NAND-NAND 回路・NOR-NOR 回路

ここでは主に論理回路の書き換えについて述べる．書き換えの基本はまず適当な位置に二重否定を意味する二連の NOT ゲートを挿入する．そこからド・モルガンの定理による論理ゲートの書き換えを行い所望の論理ゲートを作る．

5.4.1　AND-OR 回路から NAND-NAND 回路への書き換え

AND-OR 回路とは積和標準形を論理ゲートで表現したときに現れる 1 段目が AND 回路，2 段目が OR 回路となっている論理回路のことである．積和標準形だけでなく多段回路に関しても図 5.8 のような部分的な論理回路もここでは AND-OR 回路と呼ぶ．

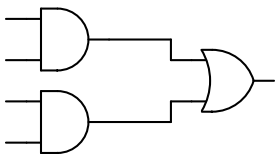

図 5.8　多段理論に見られる AND-OR 回路

この節の初めで適当な位置に二重否定を挿入しド・モルガンの定理を使うと述べたが，面倒である．またすでに前もって NAND ゲートに関しては図 5.7 の関係を示しているのだから図 5.9 のような過程で NAND-NAND 回路への書き換えが完了する．

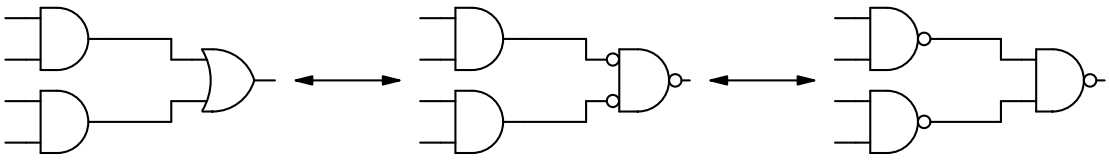

図 5.9　AND-OR 回路から NAND-NAND 回路へ

5.4.2　OR-AND 回路から NOR-NOR 回路への書き換え

OR-AND 回路とは和積標準形を論理ゲートで表現したときに現れる 1 段目が OR 回路，2 段目が AND 回路となっている論理回路のことである．和積標準形だけでなく多段回路に関しても図 5.10 のような部分的な論理回路もここでは OR-AND 回路と呼ぶ．

この回路の NOR-NOR 回路への書き換えは先の NAND-NAND 回路のときと同様で図 5.11 のようになる．

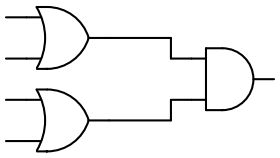

図 5.10　多段理論に見られる OR-AND 回路

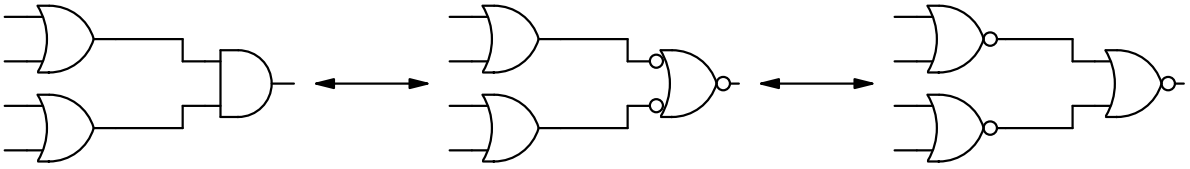

図 5.11　OR-AND 回路から NOR-NOR 回路へ

5.4.3　NAND-NAND 回路・NOR-NOR 回路の作図について

自ら与えられた論理関数を NAND-NAND 回路または NOR-NOR 回路で表現するとき，以上の話から以下の方法をとらなければならないことが分かる．

NAND-NAND 回路　積和標準形で与えられた論理関数を表し，そこから AND-OR の 2 段理論で標準形を表す．その後，NAND-NAND 回路への書き換えをする．

NOR-NOR 回路　和積標準形で与えられた論理関数を表し，そこから OR-AND の 2 段理論で標準形を表す．その後，NOR-NOR 回路への書き換えをする．

つまり，標準形は積和も和積も出来なければならない．

5.4.4　多段理論の書き換えについて

多段理論の書き換えは基本的にド・モルガンの定理の関係を使った書き換えの図 5.6，図 5.7 と，直前で扱った図 5.9，図 5.11 の書き換えを多段の回路に対して部分的に使うことで進める．また個の書き換えは n 入力に対しても同様に行うことができる．

さらに，AND と OR の回路を NAND と NOR の回路にすべて書き換えする場合は最も出力に近い論理ゲートから順に書き換えるとよい[*3]．

5.5　組み合わせ論理回路の解析

図を示すのは面倒なので軽く言及するのみ．

[*3] 今扱っている論理関数は複数入力，1 出力であるためこのように言える．

5.5.1　AND と OR の回路

組み合わせ論理回路の解析でよく取られる方法は入力から出力まで順に回路を追ってゆき，論理ゲートに差し掛かった時にその段階で 1 を出力する入力の論理式を記してゆき，最終的に出力の式まで持ってゆくという方法である．AND と OR のみの回路の場合，各論理ゲートでの立式は容易であるためそれほど困難な方法ではない．

5.5.2　NAND と NOR の回路

方法は AND と OR の回路の時と変わらないのだが，実際に行ってみると NAND，NOR 演算は毎回ド・モルガンの定理を使い論理式を立てなればならず，かなり面倒な作業になる．これを回避するため，NAND，NOR の論理ゲートはすべて AND，OR に前もって書き換えをしてから解析をする．

5.6　代表的な組み合わせ論理回路

この節では論理回路のうち極めて代表的なものをその真理値表などとともに紹介する．

5.6.1　半加算回路 (half adder)

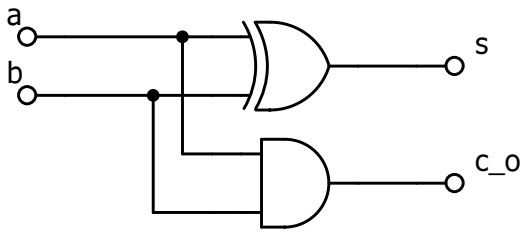

図 5.12　半加算回路の論理回路

表 5.1　半加算回路の真理値表

a	b	s	c_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

半加算回路は表 5.1 の真理値表となる図 5.12 の回路である．論理式では

$$s = a \oplus b$$
$$c_o = a \cdot b$$

となり，2 進数 1 バイトの足し算と繰り上がりを示している．

5.6.2　全加算回路 (full adder)

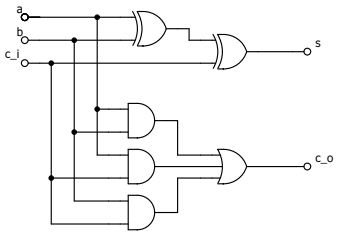

図 5.13　全加算回路の論理回路

表 5.2　全加算回路の真理値表

a	b	c_i	s	c_0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加算器回路とは表 5.2 の真理値表となる図 5.13 の回路である．論理式では

$$s = a \oplus b \oplus c_i$$
$$c_o = a \cdot b + b \cdot c_i + c_i \cdot$$

となり，2 進数 1 バイトの足し算に c_i がさらに足された格好になる．これを半加算回路になかった繰り上がった数と考えることで全加算回路は 1 つ小さな位からの繰り上がりを考慮した足し算を示していることになる．

5.6.3　n ビットの比較回路

n ビットの二進数の正の数の大小比較を実行する論理回路を考える．単純な発想としては大きいくらいのビットで大小比較をすればよいように思えるが論理回路での実現は困難である[*4]．

ではその大小比較のアルゴリズムを考える．基本的な発想は上位桁の大小比較結果は下位桁の結果に対し優

[*4] 判定終了の条件はまず思いつかない．一連の動作ですべての位を必ず見るほうが簡単な論理でできると考えられる．

先され，上位桁に大小がない場合下位桁の結果が採用される．

$$A_2 = a_{n-1}a_{n-2}\cdots a_1a_0$$
$$B_2 = b_{n-1}b_{n-2}\cdots b_1b_0$$
$$A \leq B \Rightarrow c_n = 0$$
$$A > B \Rightarrow c_n = 1$$

このような条件を満たす c_n は以下の計算で達成される．

$$c_0 = \begin{cases} 0 & (a_0 \leq b_0) \\ 1 & (a_0 > b_0) \end{cases}$$

ここから i が 1 から n まで

$$c_i = \begin{cases} 0 & (a_i \leq b_i) \\ 1 & (a_i > b_i) \\ c_{i-1} & (a_i = b_i) \end{cases}$$

この一連の計算を論理関数で示すと

$$c_0 = a_0\bar{b_0}$$
$$c_i = a_1\bar{b_i} + a_ic_{i-1} + \bar{b_1}c_{i-1}$$

回路図は示さない．

5.6.4 デコーダ

参考資料にあったデコーダの説明はこうなっている．

> n ビットの 2 進符合に対応した n 個の入力から，最大 2^n 個の最小項 (minterm) を出力する．

よくわからない．もっとかみ砕くとこのデコーダは n 入力 2^n 出力の論理回路でいかなる入力に対してもただ 1 つの出力のみ 1 で残りは 0 となる．また複数種類の入力で同じ出力が 1 になることはない回路のことである．ここまで説明すると何となく理解ができてくるのではないか．もっと言えばデコーダとは全入力に対する最小値をそれぞれ出力として用意した回路だといえる．これでかなり初めに挙げた参考資料の説明も理解できる．2 入力のデコーダの論理回路を図に示す．

5.6.5 エンコーダ

これはデコーダと全くの正反対の機能を持つ論理回路である．すなわち，2^n 個でかつ 1 つを除きすべて 0 の入力を受け取り n 個の出力をする回路である．詳しくは参考資料を参照してもらうことにしてこの説明は終わる．

5.6.6 プライオリティエンコーダ

これは先に説明したエンコーダとは少し異なり，入力に対し優先順位を与えることで，複数個の 1 の入力があってもエンコーダの働きができる論理回路である．以下の図と表で 4 入力のプライオリティエンコーダの一例を挙げる．

5.6.7　マルチプレクサ

複数の入力から 1 つの入力を選択し，その入力をそのまま出力する論理回路である．例えば入力 I_1, I_2 を入力 S によって選択するという 2:1 マルチプレクサを考えると $S = 0 \to I_1, S = 1 \to I_2$ の対応で論理式を立てると

$$f(S) = \bar{S}I_1 + SI_2$$

となり，この論理式を実現する論理回路は図のとおりである．

さらに 4 つの入力を選択する 4:1 マルチプレクサの真理値表，論理回路を表と図で示す．

5.6.8　マルチプレクサによる任意の論理回路の実現

マルチプレクサは入力によって出力を選択する論理回路であるため，選択される側の入力を恒真や恒偽などをうまく使いながら回路の作成をするをいかなる n 入力論理ゲートも実現することができるのである．具体的な例は挙げないがこの事実を知ってほしい．

第 6 章

順序回路

6.1　順序回路とは

この章で新しく出てきた順序回路というのは出力がその瞬間の入力に加え，過去の入力に依存する回路のことである．組み合わせ回路と記憶回路で構成される．

6.1.1　同期式順序回路

組み合わせ回路ごとに入力が与えられてから出力が決定するまでの時間は違う．それでは入力で与えられる入力変数と，記憶回路から与えられる状態変数が意図したタイミングからずれる状態が想像できる．それを解消するのが変数の伝達のタイミングを必ず一致させるという方法である．クロックと一定間隔で発する信号に合わせて変数の伝達を行う手法[*1]を同期式という．

6.1.2　非同期式順序回路

端的にはクロックを使わない順序回路のことである．当然順序回路には先に述べた入力のタイミングに関する問題があるが非同期式では様々な方法でこれを解消しようとしている．しかし，今の大勢は同期式である．

[*1] 一番遅いものを待つという発想である．従ってクロックの間隔はすべての組み合わせ回路の演算時間よりも長い．

6.2 順序回路の表現

6.2.1 有限状態機械 (FSM)

順序回路の動作を数学的に表現したものであるが，2 つの型を説明する．議論の簡単化のため以下の文字を置く．

$\boldsymbol{x}$: 入力変数
$\boldsymbol{y}$: 状態変数
$\boldsymbol{z}$: 出力変数

ミーリ型機械　このタイプの機械では状態と入力の両方が関与し次の状態と出力を決定する．つまり以下のような式によって表すことのできる場合である．

$$\boldsymbol{y}_{n+1} = \delta(\boldsymbol{x}_n, \boldsymbol{y}_n)$$
$$\boldsymbol{z}_{n+1} = \lambda(\boldsymbol{x}_n, \boldsymbol{y}_n)$$

ムーア型機械　このタイプの機械は状態と入力の両方が関与し次の状態は決定するが，出力は前の状態にのみ依存する．つまり以下の式である．

$$\boldsymbol{y}_{n+1} = \delta(\boldsymbol{x}_n, \boldsymbol{y}_n)$$
$$\boldsymbol{z}_{n+1} = \lambda(\boldsymbol{y}_n)$$

6.2.2 状態遷移表・状態遷移表

有限状態回路の挙動を考えるうえで効果的なのは状態遷移図や状態遷移表を利用することである．正確には状態遷移図の作成を通し対象とする論理回路の動きを体感し，状態遷移表を作成するという流れである．具体的な例は参考資料に任せる．

6.2.3 状態の記憶

話は急に実際の論理回路の作成の際のものに移る．ここでは状態変数 $\boldsymbol{y}$ が全状態の集合 S の元であるとしそれらを番号付けした場合，いちいち複数の状態変数の記憶を行わなくとも，付けた番号さえ覚えておけば十分であろうという発想から始まる．

最短符号　集合 S の元が n 個であった場合 $\log_2 n$ ビットの符号を与え記憶する方法である．できる限り状態が遷移する可能性のある状態間で与えられる符号のハミング距離[*2]は小さいほうが良いとされる．

1 ホット符号　n 個の状態を記憶するのに n ビット用意し，一か所のみが 1 の符号を各状態に与え記憶する．この方法だと状態の遷移の際，常に 2 ビットの変換で済む．

では，なぜこの話をしたのかであるが，実は今まで状態変数，状態変数といってはきたが実際にどのようなものかは説明できてはいなかった．すなわち，状態変数と対応する状態は自ら定めるということである．

[*2] 変換するビット数との理解で十分である．

6.2.4　組み合わせ回路のカルノー図の作成

組み合わせ回路のカルノー図の作成ができれば組み合わせ回路の設計に大きく近づく．順を追って説明する[*3]．

1. 入力，状態，出力を決定し状態遷移図を作成する．このとき状態や入力に漏れがないようにする．
2. 状態遷移表を作成する．このとき次状態と出力は分けるとよい．
3. 入力，出力に対してはそのそれぞれに対応するリテラルを与え[*4]，状態に対しては最短符号を与える．
4. 状態割り当て，符号化をした結果をそのまま状態遷移表で書き換える．

6.3　フリップフロップ

順序回路における記憶回路に使用される回路である．4 種のフリップフロップ (FF) 回路があり，それぞれで動きが違うのでしっかりと覚えることが必要．まずは，FF の前身であるラッチ回路から話を始める．

6.3.1　ラッチ回路

NOR や NAND などの否定ゲートをたすき掛けにした回路でこれが 1 ビットの記憶をする．入力 S, R がセットとリセットの意味を成し，出力が $Q, \bar{Q}$ である．出力 Q がセットにより 1 にリセットにより 0 になり，入力がなければ値は保存される．正確には SR ラッチ回路と呼ぶ．

このラッチ回路の前に別の回路をつけることで様々なフリップフロップを作る．

6.3.2　SR フリップフロップ

直前で説明したラッチ回路を縦続させることでクロックに同期して動くようにした回路である．入力 S, R がセット，リセットであるので真理値表がなくともどのように値が変化し保存されるかは理解できる．

6.3.3　D フリップフロップ

SRFF における入力 R を $\bar{S}$ としたフリップフロップである．そのため記憶されるのは 1 つ前のクロックにおける入力 S であり，理解が最も容易である．

6.3.4　JK フリップフロップ

SRFF において入力 S, R がともに 1 となる入力は挙動が予想できないため入力禁止とされている．それに対しこの JK フリップフロップは入力 J が入力 S に，入力 K が入力 R に対応しながら，もし J, K がともに 1 の時，トグル[*5]を行う記憶回路になっている．

[*3] ミーリ型機械に対して，状態変数には最短符号を与える場合を説明する．

[*4] 符号化と呼ぶ．

[*5] ビットの反転である．

6.3.5　T フリップフロップ

JKFF における入力を $J = K$ としたものである．つまり JKFF における入力がともに 0，またはともに 1 の場合のみになり，値の維持かトグルかの 2 つの動きになる．

6.3.6　フリップフロップまとめ

フリップフロップは 1 つ前のクロックにおける入力によって何らかの記憶を行い，次のクロックでその記憶の結果を出力する回路である．つまり理解しておきたい点は記憶する値がどのような場合に変化し，または維持されるかである以下の表 6.1 に示す．

表 6.1　各種フリップフロップ

$Q(t) \to Q(t+1)$	S	R	D	J	K	T
$0 \to 0$	0	*	0	0	*	0
$0 \to 1$	1	0	1	1	*	1
$1 \to 0$	0	1	0	*	1	1
$1 \to 1$	*	0	1	*	0	0

6.4　同期式順序回路の設計

実際にフリップフロップを用いて記憶回路を作れば順序回路は作成できる段階になっている．実際にどのようにするのか順を追う．

1. 状態の設定・状態遷移図の作成
   この時に出力も完成させておく
2. 状態数の最小化
   状態を設定するとき必ずしも最小の状態数となることはない．多くの場合，同義の状態を定義しているためこれを除かなければならない．
3. 状態の符号化
   状態数に応じてビットを用意して状態を符号化する．この符号化が記憶回路で実際に記憶させるビットとなる．
4. 符号化された状態を使って出力表，状態遷移表を作成
   これでほぼ順序回路の設計は完了である．
5. 出力表，状態遷移表から論理式を立てる
   これで本当に順序回路の設計は完了である．当然，論理式は簡単化するが，この時の簡単化の方法はその時の条件に従う．

以上の手順をもって順序回路の設計はなされる．使用するフリップフロップによって記憶回路に対する論理式

に変化は生まれるが表 6.1 を理解さえしていれば大きな問題はない.

第 7 章

フリップフロップの活用例

7.1　レジスタとカウンタ

7.1.1　レジスタ

レジスタとはフリップフロップを並べることで同じ同期信号でデータの記憶・読み出しを可能にした回路. 難しいことは上回生の授業で扱う模様.

レジスタのうちでもシフト操作を実現したものをシフトレジスタという．当然シフトには左右の区別がある.

7.1.2　カウンタ

カウンタとはレジスタのうち，与えられたパルスによって予め決定していた状態を遷移してゆくものをいう．名前の通り数を数えてゆくのもカウンタであるし，その他奇数で動いたり，とにかく決められている順に遷移するものをカウンタと呼ぶ.

7.2　ハザード

当然のことかもしれないが，実際の論理ゲートは入力のよって出力が決定するのに有限の時間を要する．この出力が決定するまでの時間に出力はどのような状態となっているのかが気になることである．もちろん一発で次の出力に遷移するのが理想ではあるが，実際には出力決定の前に不要な遷移が発生する．これをハザードという．ハザードには出力が実際には変化していないのに発生する静的ハザードと出力が変化するときに 0 と 1 を往復するような動的ハザードがある，以下の表はハザードの種類を示したものである.

表 7.1　ハザードの種類

ハザードの種類	前の定在値	後の定在値	不要遷移の様子
静的ハザード	0	0	$0 \to 1 \to 0$
	1	1	$1 \to 0 \to 1$
動的ハザード	0	1	$0 \to 1 \to 0 \to 1$
	1	0	$1 \to 0 \to 1 \to 0$

単一入力変化の場合には，AND-OR 二段組合わせ回路では 1 ハザードのみが存在し，0 ハザードや動的ハザードは生じない．OR-AND 二段組合わせ回路では，0 ハザードのみが 存在し，1 ハザードや動的ハザードは生じない．

AND-OR あるいは OR-AND 二段回路の場合，単一入力変化で生じるハザードはすべて除去できる．AND-OR 二段回路において，隣接する二つの最小項の間の遷移で 1 ハザード が生じる場合にはそれらの最小項を同時に被覆する積項を追加すればそのハザードを除去できる．OR-AND 二段回路においても同様の操作で 0 ハザードを除去できる．

第 8 章

演算回路

8.1　2 進数の表現と加減

8.1.1　負の数

ビットを用いた 2 進数表現では負の表現に困る．符号のためにビットを割り当てたからといっても計算ができないのである．これはビット計算では本質的に減算ができない[*1]ことが理由である．よって減算は負の値の加算という方法をとるがこの時，適当な形で負の値を表現しなければならない．ここで現れるのが補数である．

補数に 2 種類存在する．ここでは話の一般化のため r 進数で n 桁の整数 N における補数を考える．

$r-1$ の補数

$$(r^n - 1) - N$$

この式によって与えられる．具体的には各桁ごとにその桁の数を $r-1$ から引けばよい．この計算を 2 進数において行うと全ビット反転したことと同じである．

r の補数

$$r^n - N$$

この式によって与えられる．$r-1$ の補数に 1 を足したものであるとも言うことができきる．よって具体的な例は不要であろう．

8.1.2　符号のルール

符号を使えば負の数の表し方は 3 種類想定できる．

[*1] 本当に字面の上での加算しかできない．つまり-1 を足そうとしてもそのまま (マイナス符号)1 というような数の表現でも 1 を足すことしかできないのである．

1. 符号-絶対値表現
2. 符号-1 の補数表現
3. 符号-2 の補数表現

これらのどの場合でも MSB が符号を表すビットとなり，正であれば 0，負であれば 1 である．

8.1.3 桁移動

2 進数表示では左右のシフト計算で値を 2 倍，1/2 倍とすることができる．この時桁移動は符号を表していないビットでのみ起こる．されてこの時，空白となるビットを何で埋めるのかが問題である．その埋め方は以下のようである．

1. 絶対値の場合：常に 0
2. 1 の補数の場合：常に符号と同じ値
3. 2 の補数の場合：左シフトであれば常に 0，右シフトであれば符号と同じ値

8.1.4 2 進数の減算

準備が整ったところでようやく 2 進数における減算である．先に呼べたように減算は負の数を加算するという方法で計算するがこの時の負の数が補数で表現されているのである．計算は $M - N$ (n 桁) を行うものとする．

1 の補数を使う

まず 1 の補数を足すことから計算が始まる．

$$M + (2^n - N - 1) = \begin{cases} M - N + 2^n - 1 & (M > N) \\ 2^n - 1 - (N - M) & (N > M) \end{cases}$$

これより $M > N$ ならば LSB に 1 を足したうえで繰り上がりの分の 2^n を無視することで答えが得られ，$M < N$ ならばそのまま 1 の補数で表現した答えになっている．しかし，これはあまりにもわかりにくい．

2 の補数を使う

ここも 1 の補数の時の同様に 2 の補数を足すことから始まる．

$$M + (2^n - N) = \begin{cases} M - N + 2^n & (M > N) \\ 2^n - (N - M) & (N > M) \end{cases}$$

これより $M > N$ ならば繰り上がりの分である 2^n を無視することで答えが得られ，$M < N$ ならばそのまま 2 の補数として答えが現れている．当然 2 の補数を利用した方法の方が容易である．

8.1.5 符号付き 2 進数の加減

ここでは 2 の補数を用いているものとするが，この計算においては MSB，すなわち符号を表現しているビットのさらに左側にもう 1 ビット加えて[*2]通常の計算を行う．この左から 2 つ分をまとめて符号のビットと

[*2] 符号と同じ値を加えればよい．

考え，計算後の符号を確認する．すると，
00：結果は正の数であふれなし
01：結果は正の数であふれあり
10：結果は負の数であふれあり
11：結果は負の数であふれなし
となる．