初版：2012年8月24日
改訂：2018年7月14日
監修：津田伸生

金沢工業大学

情報工学科

2018 年度版

コンピュータシステム基礎教科書

（前編）

- 現代のコンピュータの構成と動作の基礎を学習する．
- 講義資料のURL(学内専用)
  http://www.ws.kanazawa-it.ac.jp/~joho/
- 毎週，受講前に必要な部分を印刷して持参することを推奨する
- 毎週の授業では各自のノート等に要点のメモをとること
- 毎週の授業の予習・復習を欠かさないこと．特に復習では「何が約束事なのか」を整理して例題と問題を見直すこと．

目次

第１章　コンピュータの基本構成 ・・・ 3

1.1 コンピュータはどのように使われているか ・・・ 3

1.2 デジタル情報とアナログ情報 ・・・ 3

1.3 なぜデジタル情報処理が主流になったのか ・・・ 4

1.4 現代のコンピュータ（von Neumann 型）の特徴 ・・・ 4

1.5 パソコンの中をのぞいてみると ・・・ 4

1.6 コンピュータの基本構成 ・・・ 4

第２章　コンピュータとコード化 ・・・ 7

2.1 コンピュータの機能 ・・・ 7

2.2 一般的な意味での情報の種類 ・・・ 7

2.3 デジタルコンピュータ ・・・ 7

2.4 ビット ・・・ 7

2.5 ビット列 ・・・ 8

2.6 コードとコード化 ・・・ 9

2.7 数のコード化（基本的な考え方） ・・・ 9

2.8　16 進記法 ・・・ 10

2.9 記法の明示 ・・・ 11

2.10 ビット列の長さの単位 ・・・ 11

2.11　10 進数値を表す 2 値コード ・・・ 11

2.12 文字コード ・・・ 12

第３章　ビットの実現と論理回路 ・・・ 15

3.1 ビットの実現 ・・・ 15

3.2 論理関数と真理値表 ・・・ 15

3.3 論理演算と論理回路 ・・・ 17

3.4 算術演算回路 ・・・ 24

3.5 まとめ ・・・ 31

第４章　コンピュータにおける計算処理のしくみ ・・・ 32

4.1 メインメモリの構成 ・・・ 32

4.2 メインメモリの容量と番地 ・・・ 33

4.3 番地指定(アドレス指定) ・・・ 33

4.4 メモリ内容の表示方法 ・・・ 34

4.5 命令と実行 ・・・ 34
4.6 まとめ ・・・ 40

第 5 章　ALU で行う演算の 16 進表記 ・・・ 41
5.1 算術演算 ・・・ 41
5.2 論理演算 ・・・ 44
5.3 ビット列操作 ・・・ 46

以下は後編を参照

第 6 章　16 ビットプロセッサ NT-Processor V1 によるプログラミング
第 7 章　プログラムの処理

2 進数の扱いについては補足編を参照

NT-ProcessorV1 用プログラムエディタ・エミュレータの操作方法は操作編を参照

第１章　コンピュータの基本構成

この章では，現代のコンピュータの特徴と構成概要について学ぶ.

1.1 コンピュータはどのように使われているか

- 汎用コンピュータ（計算機）
  - パソコン（Personal Computer）
  - ワークステーション（Workstation）
- サーバマシン（Server Machine）
- 電卓（Calculator）

  注）コンピュータには分類されない
- 組込みシステム（Computer Embedded System）
  - 携帯電話機
  - ゲーム機
  - 家電製品（マイコン： Microcomputer）

現代のコンピュータはデジタル方式．パソコンなどの汎用コンピュータでも，システムに組み込まれた専用コンピュータでも，基本的な計算処理の仕組みはみな同じ

1.2 デジタル情報とアナログ情報

- 情報には文字コード（番号）や，物の個数といった整数で表現できる（離散的）情報と，長さや電圧の値のように切れ目のない（連続的）情報とがある
- 前者をデジタル情報，後者をアナログ情報という

1.2.1 アナログ情報処理の特徴

- アナログ演算回路は回路規模が小さい

  注）オペアンプ（Operational Amplifier）という回路で四則演算を実行できる
- 加算等の演算を繰り返すと上位 2～3 桁の数値のみが保障される
- 演算の組み合わせを任意に変更できない（回路設計のやり直しが必要）ため用途が限定される（大規模なアナログ演算回路の実現は困難）
- アナログ電気信号はノイズに弱い
- データを電気的または磁気的に記録すると劣化が生じる

1.2.2 デジタル情報処理の特徴

- デジタル演算回路は回路規模が大きい（ただし LSI 技術でカバーできる）
- 加算等の演算を繰り返すと桁数は増えるが最小単位の数値の精度が保障される
- 演算の組み合わせをデータと同じデジタル情報であるプログラムによって任意に

設定できる

- デジタル電気信号はノイズに強い
- データを電気的または磁気的に記録しても劣化が生じにくい．また，データの圧縮，誤り訂正が可能

1.3 なぜデジタル情報処理が主流になったのか

- 自律制御機械（自分で次の状態を決められる）であるコンピュータを実現できる
- 任意の演算をプロセッサとプログラムで実行できる．回路設計をする必要がない
- 汎用性がある（マルチメディアなどいろいろな用途に使える）

1.4 現代のコンピュータ（von Neumann 型）の特徴

- **デジタル方式**
- **2 進数演算**（データはすべて‘0’と‘1’の組み合わせで表現されている）
- **プログラム内蔵**（プログラムは機械命令の組み合わせで表現されている．機械命令も‘0’と‘1’の組み合わせで表現されている）
- **逐次制御**（順序制御，シーケンス制御ともいう：プログラムの機械命令を次々に読み出して計算処理を進める）

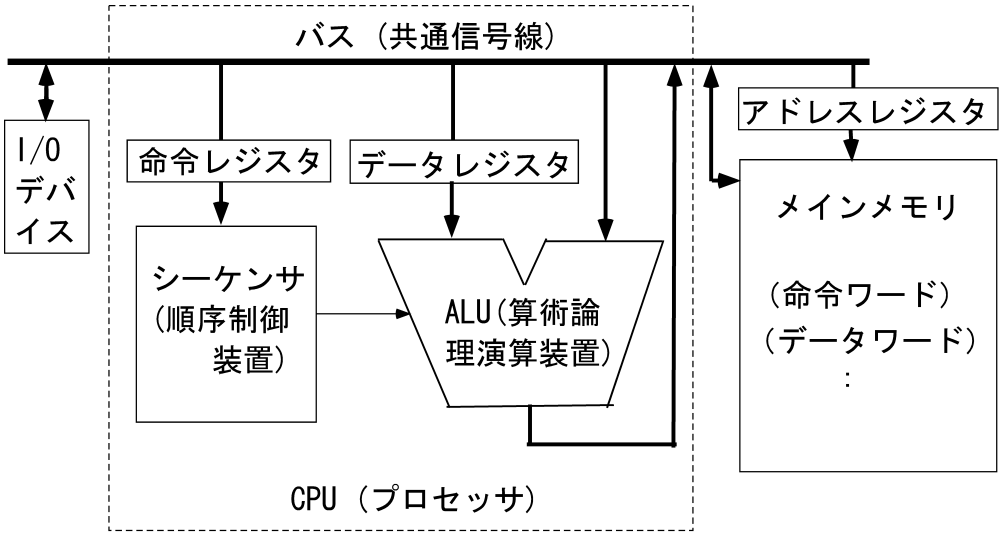
1.5 パソコンの中をのぞいてみると

- **マザーボード**：LSI を搭載するプリント基板
- **プロセッサ**（LSI）：CPU，MPU，中央処理装置ともいう
- **メモリ**（LSI）モジュール：メインメモリ，主記憶装置ともいう
- **I/O インタフェース**：入出力装置，I/O デバイスともいう．ハードディスク，ネットワークカード，キーボード，マウス等の周辺装置を接続するための装置および端子

1.6 コンピュータの基本構成

プログラム内蔵型コンピュータはおおまかには次の２つの部分からなる

- **プロセッサ**（CPU: Central Processing Unit，中央処理装置）：計算処理を実行する部分
- **メインメモリ**（主記憶装置）：機械命令の組み合わせに変換されたプログラムと処理対象のデータを記憶しておく部分



＊コンピュータ内部では命令，データともに'0'または'1'の２値で扱われる．ALUの基本機能は２進数加減算，２値論理演算，ビット列操作．

構成要素の説明

- コンピュータ内部では，個々の機械命令やデータは**命令ワード**，**データワード**という単位で扱われる．**メインメモリ（主記憶）**は，一般にはバイトという単位で仕切られていて，各仕切りには**アドレス（番地）**が付けられている
- **バス（BUS）**：CPU 内部とメインメモリとを結ぶ共通信号線．図では１本に簡略化してあるが，一般には複数本備えられている
- **アドレスレジスタ**：メインメモリ上の読み書きしたいアドレス（番地）を一時記憶しておく高速メモリ．命令ワードのアドレスのみを一時記憶するものは**命令アドレスレジスタ**という
- **命令レジスタ**：メインメモリから読み出した１個の命令ワードを一時的に記憶しておく高速メモリ
- **データレジスタ**：**汎用レジスタ**ともいう．メインメモリから読み出したデータワードや計算（演算）結果のデータワードを一時的に記憶しておく高速メモリ．一般には，複数個のデータワードを記憶できるように仕切られていて，各仕切りには番号が付けられている
- **ALU（Arithmetic & Logic Unit：算術論理演算装置）**：２つまたは１つのデータワードを入力として計算処理（**加減算，論理演算，ビット列操作**（桁ずらし：シフト）など）を行い，１つのデータワードを出力する装置
- **シーケンサ（Sequencer：順序制御装置）**：単純に制御装置ともいう．命令レジスタに一時記憶されている命令ワードを解読し，その命令で指定されている処理（演算）を行うための制御信号を CPU の構成回路へ伝達する．一般には次の機能

をもつ

- ALU でどのような演算を行うのかを解読する（**演算指定**）
- ALU の入力データワードの読み出し先と出力データワードの書き込み先（データレジスタの番号やメインメモリのアドレス：これらをオペランドという）を解読する（**オペランド指定**）
- 次に実行する命令ワードのアドレスを決めてアドレスレジスタに書き込む（**次命令アドレス指定**）

問題 1.1 アナログ情報とデジタル情報の違いを説明せよ．

問題 1.2 アナログ情報処理とデジタル情報処理の特徴を利点と欠点に分けて列挙せよ．

問題 1.3 なぜデジタル情報処理が主流になったのか理由を３項目挙げよ．

問題 1.4 現代のコンピュータ（von Neumann 型）の特徴を４項目挙げよ．

問題 1.5 コンピュータの基本的な構成要素とその接続関係をブロック図で示せ．

問題 1.6 ALU の基本機能を３項目挙げよ．

問題 1.7 シーケンサの基本機能を３項目挙げよ．

問題 1.8 次のコンピュータの構成要素の概略機能を述べよ．

(1) メインメモリ
(2) アドレスレジスタ（命令アドレスレジスタ）
(3) 命令レジスタ
(4) データレジスタ（汎用レジスタ）
(5) バス

問題 1.9 どのようなプロセッサがあって，どのような用途で使われているか調べよ．

＊前ページのコンピュータの基本構成図は，8 ビットマイクロコンピュータ（マイコン）のものを簡略化して示している．実用版のものは，後編の「第 6 章 16 ビットプロセッサ NT-Processor V1」の「6.1 プロセッサの概要」と「6.2 プロセッサのしくみ」に説明してある．これらの節を読んで，コンピュータ（プロセッサ）の構成要素と計算処理の関係を予習すること．

第２章　コンピュータとコード化

この章では，コンピュータで情報を扱うための情報のコード化について学ぶ.

2.1 コンピュータの機能

- 情報の入力（読み取り）・出力（表示）
- 情報の保存・管理
- 情報の処理
    - 数値計算
    - 情報検索
    - 情報の変換
- 情報伝達（→ネットワーク）

2.2 一般的な意味での情報の種類

- 文字情報
- 数値情報
- 音声
- 画像

これらの情報は，アナログ情報の場合とデジタル情報の場合がありうる

2.3 デジタルコンピュータ

- 現在の通常のコンピュータはデジタル情報のみを扱うことができる
- これを強調してデジタルコンピュータ（デジタル計算機）とも呼ぶ
- アナログ情報をコンピュータで扱うにはデジタル情報に変換する必要がある（画像であればデジタルカメラやイメージスキャナを使う）
- これをデジタル化（離散化）という（広い意味では A/D 変換という）
- デジタル情報をアナログ情報に変換する操作を D/A 変換という（音声出力など．専用の LSI が必要）

2.4 ビット

- ビットとは２つの状態（値）を持つ変数のようなもの．デジタル情報の基本単位.
- 通常この２つの状態を 2 値‘0’と‘1’で表現する．状態を示す‘0’または‘1’のことを**論理値**または**ビット値**ともいう.
- ビット値の‘0’または‘1’は，数値の‘0’と‘1’とを対応させる場合を除いて直接の関係はない（○と●を用いてもよいが，これらは文字であるためコンピュータでは直接扱えない）

・ コンピュータでは，すべての情報を 2 値‘0’または‘1’（‘0’／‘1’と書く）のビットに変換して格納･処理する（コンピュータ内部では, 2 値の‘0’／‘1’をスイッチの off／on, 電圧の Low／High，磁化の S／N などに対応させている）

2.5 ビット列

・ 複数のビットを並べたものをビット列という．n 個のビットがある場合は n ビットの**ビット列**という（注：ビットという言葉は，ひとつの 2 値状態を示す単位としても使われ，またビット列の桁数（ビット列が示す 2 進数の桁数）の単位としても使われる）

・ n ビットのビット列では 2^n通りの状態を表現（指定）できる．これを**状態数**が 2^n（2 の n 乗）であるという

n=1: 状態数は $2^1=2$ ：表現できる状態は ‘0’， ‘1’

n=2: 状態数は $2^2=4$ ：表現できる状態は “00”，“01”，“10”，“11”

n=3: 状態数は $2^3=8$ ：表現できる状態は “000”～“111”

n=4: 状態数は $2^4=16$：表現できる状態は “0000”～“1111”

n=8: 状態数は $2^8=256$：表現できる状態は “0000 0000”～“1111 1111”

n=10: 状態数は $2^{10}=1{,}024$: 表現できる状態は

“00 0000 0000 0000”～“11 1111 1111 1111”

n=16: 状態数は $2^{16}=65{,}536$: 表現できる状態は

“0000 0000 0000 0000”～“1111 1111 1111 1111”

注）習慣的には，1 ビットの 2 値状態には ‘ ’（シングルクォテーション），2 ビット以上のビット列には **“ ”**（ダブルクォーテション）をつけるが，省略する場合もある

・ n ビットのビット列に 2 進数の数値を対応させると，最小値は all ‘0’=$(0)_{10}$，最大値は all ‘1’ =$(2^n-1)_{10}$となる

n=1: 最小値は $(0)_2=(0)_{10}$，最大値は $(1)_2=(2^1-1)_{10}=(1)_{10}$

n=2: 最小値は $(00)_2=(0)_{10}$，最大値は $(11)_2=(2^2-1)_{10}=(3)_{10}$

n=3: 最小値は $(000)_2=(0)_{10}$，最大値は $(111)_2=(2^3-1)_{10}=(7)_{10}$

n=4: 最小値は $(0000)_2=(0)_{10}$，最大値は $(1111)_2=(2^4-1)_{10}=(15)_{10}$

n=8: 最小値は $(0000\ 0000)_2=(0)_{10}$,

最大値は $(1111\ 1111)_2=(2^8-1)_{10}=(255)_{10}$

n=10: 最小値は $(00\ 0000\ 0000\ 0000)_2=(0)_{10}$,

最大値は $(11\ 1111\ 1111\ 1111)_2=(2^{10}-1)_{10}=(1{,}023)_{10}$

n=16: 最小値は $(0000\ 0000\ 0000\ 0000)_2=(0)_{10}$,

最大値は $(1111\ 1111\ 1111\ 1111)_2=(2^{16}-1)_{10}=(65{,}535)_{10}$

2.6 コードとコード化

- 複数の事象（一般的な意味での情報の要素の集合）があり，これらを識別するために付与する符号（番号）のことを**コード**という

  例）郵便番号：住所を識別する
  電話番号：宛先ユーザを識別する
  商品コード：品物を識別する
  車のナンバー：個々の車や所有者を識別する
  （これらは 10 進数の数字によるコード）

- 情報処理のためのコード化：　コンピュータで扱う情報には 2 進数に対応したビット列のコードを付与する

  例）アルファベット大文字 A，B～Z に 5 ビットのビット列 “00000”，“00001”～“11001”のコードを付与する．
  このとき文字列“AND”のコードは ”00000 01101 00011” となる（わかりやすさのために 5 ビットずつ区切ったが，実際のコードは区切りのない 15 ビットになる）

問題2.1 上例のコード化を使用したとき

(1) 文字列 “XYZ” のコードはどうなるか.

(2) “010100100010011” はどのような文字列を表すコードか.

- 上記 (1) のように一般的な要素情報の組み合わせをコードに変換することを符号化（encode, エンコード）という
- 上記 (2) のようにコードから元の情報を取り出すことを復号化（decode, デコード）という

問題2.2 以下をコード化するのに最低何ビット必要か.

(1) 数字0 ～ 9（そのコード化例も書きなさい）

(2) アルファベット大文字・小文字と数字をあわせたもの

(3) 教育漢字 1,006文字

(4) 常用漢字 2,136文字

2.7 数のコード化（基本的な考え方）

- 離散的な有限個の数値（例：有限個の10進整数）は文字と同様にコード化できる

  例） 2 ビットを使用して 0 → “00”, 1 → “01”, 2 → “10”, 3 → “11”

  例） 3 ビットを使用して 0 → “000”, 1 → “001”, 2 → “010”, 3 → “011”,
  4 → “100”, 5 → “101”, 6 → “110”, 7 → “111”

- 上例のように数の2進法表現を利用したコード化を2進コード化という
- nビットを使用すると2^n 個の状態が存在するので$(0)_{10}$ ～ $(2^n - 1)_{10}$ の整数（2^n 個）を2進コード化できる

問題2.3　4 ビットでは 0 からいくつまでの10進整数を2進コード化できるか．またそのコードを書きなさい．

注）
- ある情報に対して，コード化は一通りに決まっているわけではない
- コード化には必ずしも連続番号が使われるとは限らない
- 目的によって同じ情報に対して異なるコード化が使用されることがある
- ビット列だけを見た場合にそれが何のコードでどのようなコード化が使用されているかを知ることはできない．それは別に記憶しておく必要がある

2.8　16 進記法

“01001010110010110011”のようなビット列はわかりにくいので

“0100 1010 1100 1011 0011”

のように4ビットごとに区切り（4の倍数個でないときは前に0を補充することもある）

“0000” → 0,　“0001” → 1,　“0010” → 2,　“0011” → 3,
“0100” → 4,　“0101” → 5,　“0110” → 6,　“0111” → 7,
“1000” → 8,　“1001” → 9,　“1010” → A,　“1011” → B,
“1100” → C,　“1101” → D,　“1110” → E,　“1111” → F

のように16進数字(0～F) を使用して

“0100 1010 1100 1011 0011” → 4ACB3

のように書くとわかりやすい．これを**16進記法**という．

- 要点：**16進数１桁は4ビット**を表している
- 16進記法で書いたビット列を“16進数”と呼ぶことがあるが，これはあくまでビット列であって“数”ではない
- ‘0’ と ‘1’ で書くのは 2進記法 と呼ばれる．ただし，これは表記法だけのことで実体はあくまでビット列である
- 3ビットずつ区切って“8進記法”を使用することもある

2.9 記法の明示

- 数字の列を示しただけでは記法がよくわからないことがある．例えば ‘1’ は2進の1と16進の 1（＝ 0001）の両方に解釈できる
- これを区別したいときには下例のように添え字（サフィックス）で記法を明示した書き方を使用する

  $(1010110101011001)_2$ ，$(4608)_{16}$ ，$(1101)_{16}$ ，$(4608)_{10}$
- 記法の英語名

  2進法：binary，16進法：hexadecimal，10進法：decimal

  プログラム内では，先頭のアルファベットb，h，dを数字列の右端に書いて記法を示す場合もある

  1010110101011001**b** ， 4608**h** ， 1101**h** ， 4608**d**

問題2.4 次の問いに答えよ．

(1) 次のビット列を16進記法で書きなさい

$(1010110101011001)_2$

(2) 次の16進記法を2進記法で書きなさい

$(4E08)_{16}$ ，　$(FFA5)_{16}$

2.10 ビット列の長さの単位

- 4ビット長のビット列を1ニブル（nibble）という（半バイトまたはハーフバイトということもある）

  16進表記の１桁は１ニブル
- 8ビット長のビット列を1バイト（byte，Bとも書く）という

  1バイトは2ニブル

2.11　10 進数値を表す 2 値コード

- 2進化10進コード（BCD: Binary Coded Decimal）

  4ビット2進数$(0000)_2$～$(1001)_2$で10進数値$(0)_{10}$～$(9)_{10}$を表す
- 3増しコード（Excess-3 Code）

  上記のBCDに$(0011)_2$を加算した4ビット2進数$(0011)_2$～$(1100)_2$で10進数値$(0)_{10}$～$(9)_{10}$を表す．all ‘0’のコードがないようにしている．
- 2-out-of-5コード

  5ビットのうち2ビットのみが‘1’のコードを数値が小から大の順に割り付ける．連続番号を使用していない10進コードの例．1ビット誤り（1ビットの‘0’／‘1’反転）を検出できる．

10進数値を表す2値コード

10 進数値	BCD	Excess-3	2-out-of-5
0	0000	0011	00011
1	0001	0100	00101
2	0010	0101	00110
3	0011	0110	01001
4	0100	0111	01010
5	0101	1000	01100
6	0110	1001	10001
7	0111	1010	10010
8	1000	1011	10100
9	1001	1100	11000

注）BCDはコンピュータでは10進数値データを表す記法として一般的に使われている．ただし，数値データが2進であるか10進であるかはプログラムで管理する必要がある．

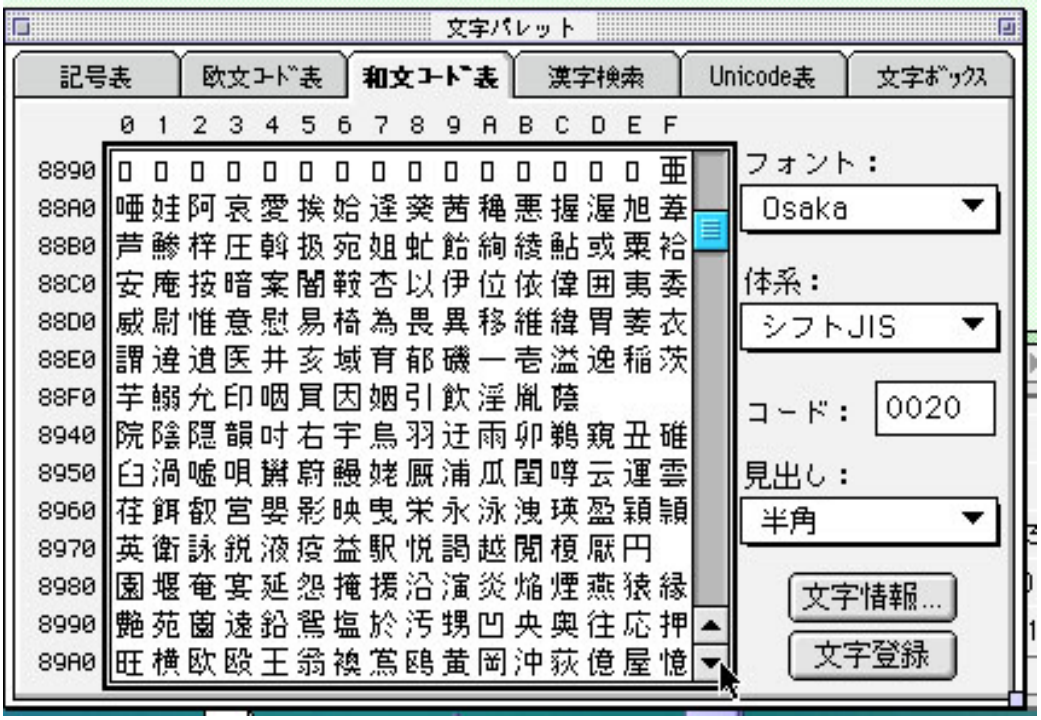
2.12 文字コード

- ASCII（American Standard Code for Information Interchange）
  コンピュータのキーボードに対応した7ビットのコードで，32種類の制御コードと96種類の文字コードを合わせた128種類が指定されている．次ページのASCIIコード表ではコードを16進表記で示している．
- 日本語文字コード：シフトJISコード
  2 バイト／文字のコード．数字，英語アルファベット，特殊記号を含む日本語文字約 8,500 種．シフト JIS コードの半角英数字と特殊記号の 1 バイト目は ASCII（7 ビット／文字）と同じ．
- その他の日本語文字コード
  JISコード：シフトJISコードの元になったコード．半角文字（1バイト）と全角文字（2バイト）を区別する記号が必要なため一般には使われていない．
  EUC（Extended UNIX Code）：Unix系OSで使用している日本語文字コード．
  Unicode：2バイトで日本語文字を含むほぼ全世界の文字を指定するコード．

ASCII コード表

制御コード		文字コード					
00	NUL	20	SPC	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	″	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	−	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	デバイス 2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	¥	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

シフトJISコード（一部）



注）

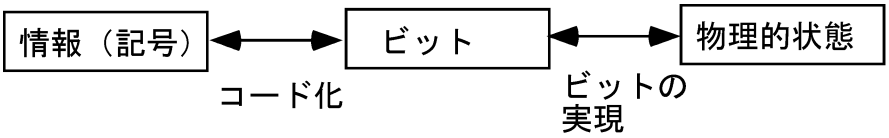
- プログラムでは，文字データは数値データと区別して扱う必要がある．日本語ワープロのローマ字入力では，キーボード入力される英語アルファベット（ASCIIコード）の列を単語辞書を用いてシフト JIS コードの日本語文字列に変換している．
- Windows OS のディスプレイ表示機能では，文字コードを文字画像データ（フォントデータ）に変換（デコード）している．

第３章　ビットの実現と論理回路

この章では，コンピュータでは計算処理がどのようにして電気的な回路で実行されるのかを学ぶ.

3.1 ビットの実現

情報はビットで表現する．この情報を記憶したり計算処理したりするためには，物理的な状態に対応させる必要がある.



非電気的な状態

ソロバン玉： 上／下，マーク： あり／なし

電気的な状態

スイッチ：on／off （閉／開），ランプ：点／滅，電圧：H(高)／L(低)，磁化：N／S

- これらの２状態の一方を ‘1’，他方を ‘0’ に割り当ててビットを実現する
- on／off，点／滅，H／L，N／S の前者を ‘1’ に後者を ‘0’ に対応させることを**正論理**，その逆を**負論理**という.
- スイッチを例にとると，正論理ではスイッチを押している状態を ‘1’，スイッチを押していない状態を ‘0’ と考える．負論理ではその逆.

3.2 論理関数と真理値表

3.2.1 ビットと論理

論理学では「a とb は等しい」などの「命題（事象）」が成立していることを「**真**（正しい）」，成立していないことを「**偽**（正しくない）」という．「真」と「偽」は互いに背反する2値状態を表している．これを「**対偶関係**」という.

コンピュータを構成しているデジタル回路（**論理回路**という）のビットにおいても正論理では「真（成立している）→ ‘1’，偽（成立していない）→ ‘0’」と対応付ける.

3.2.2 論理変数

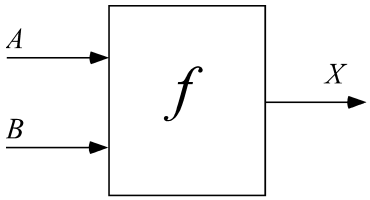
論理回路を設計するには，１つのビットの2値状態（**ビット値**ともいう）を変数で扱う．これを**論理変数**という.

論理変数を X で表すと

X ： ‘0’／‘1’ の2値状態（ビット値）の一方をとる

3.2.3 論理関数

ある論理変数のビット値が他の(複数の)論理変数のビット値によって決まる場合，ビット値どうしの関数関係を**論理関数**という．



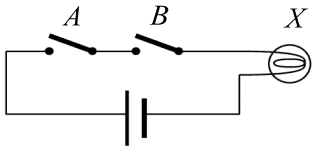
A, B ：**独立変数**（2値状態の変化をもたらす論理変数．入力変数ともいう）

X ： **従属変数**（独立変数の2値状態の組み合わせによってその2値状態が決まる論理変数．出力変数ともいう）

$X = f(A, B)$ ：論理関数（実際の論理関数の書き方は後で述べる）

例） 直列スイッチ回路

独立変数 A，B の状態（ビット値）で従属変数 X の状態（ビット値）が決まる



$A=0,\ B=0 \ \rightarrow \ X=0$

$A=0,\ B=1 \ \rightarrow \ X=0$

$A=1,\ B=0 \ \rightarrow \ X=0$

$A=1,\ B=1 \ \rightarrow \ X=1$

注）等号“=”を用いた式（**論理式**という）で論理変数のビット値を示す場合は‘0’／‘1’の ‘’ は省略する

3.2.4 真理値表

独立変数がとりうるすべてのビット値（状態）の組み合わせに対して，従属変数のビット値を表で示したものを**真理値表**という．真理値表から後で述べる論理関数が定義される．

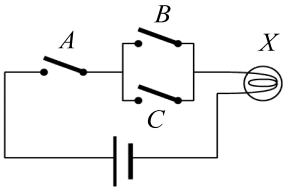
例） 直列スイッチ回路（前例）

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

◎真理値表の書き方

- 表の第１行の左側に独立変数，右側に従属変数を書く
- 表の第２行以下の左側に，独立変数がとりうるすべてのビット値の組み合わせを書く．この部分は，独立変数が2個なら4行，3個なら8行になる．通常は，2進数の順番で上からall ‘0’，最後がall ‘1’になるようにする．
- 表の第２行以下の右側の各行に，独立変数のビット値で決まる従属変数のビット値を書く．

例題3.1 次の回路（直並列スイッチ回路）の真理値表を書け



3.3 論理演算と論理回路

3.3.1 基本論理関数

基本論理回路には以下に述べる論理積(AND)，論理和(OR)，否定(NOT)がある

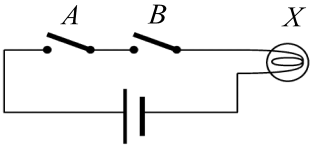
注）論理関数は等号 “=” をもつ**論理式**で表現される
論理関数の機能は**論理回路**で実現される
論理回路では，独立変数は**入力信号**，従属変数は**出力信号**に対応している
独立変数の‘0’／‘1’の値を**入力値**，従属変数の‘0’／‘1’の値を**出力値**ともいう

- **論理積（AND）**　$X = A \cdot B$

論理演算記号は “・” （論理学では $X = A \wedge B$ と書く）
事象（論理変数） A, B が共に真であれば X は真

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

例）　先の直列スイッチ回路



・ **論理和（OR）**　　$X = A + B$

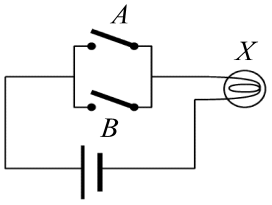
論理演算記号は“+”（論理学では $X = A \vee B$ と書く）

注）論理演算の“+”(OR)と算術加算（足し算）の“+”とを混同するな.

事象（論理変数）A または B が真のとき X は真

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

例）　並列スイッチ回路



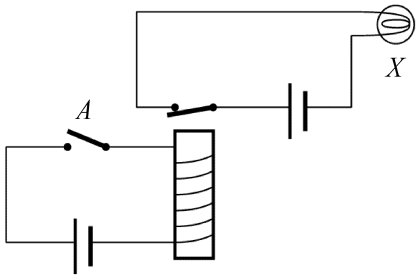
・ **否定（NOT）**　　$X = \overline{A}$

論理演算記号は“¯”（バーという）（論理学では $X = \neg A$ と書く）

事象（論理変数）A が真のとき X は偽，およびその逆（つまり‘0’／‘1’が反転する）

A	X
0	1
1	0

例）　リレー付き回路



注）

- 2値論理を扱う数学領域のことを**ブール代数**（Boolean）という
- 論理演算記号“・”，　“+”，　“¯”は**論理演算子**ともいう

3.3.2　複合的な論理関数

あらゆる論理関数は基本論理関数（論理演算子）の組み合わせで表現できる.

例）　例題3.1の直並列スイッチ回路の論理式：$X = A \cdot (B + C)$

真理値表

A	B	C	$B+C$	X
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

例題3.2　　$X = A \cdot B + \overline{C}$ の真理値表を書け

解）

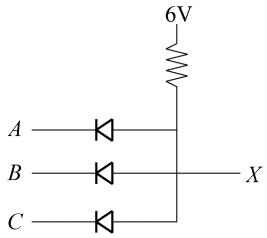
A	B	C	$A \cdot B$	$\overline{C}$	X
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

3.3.3　論理回路

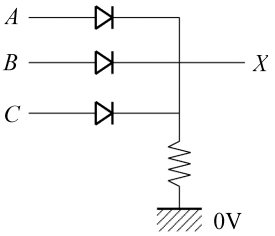
基本論理演算を行う電子回路（論理回路）は，半導体素子(ダイオードやトランジスタ)を用いて次のように構成できる．ビット値 ‘0’ を 0V，ビット値 ‘1’ を6V のように電圧に対応づける．

注）以下に示す基本論理回路はダイオード型論理回路と呼ばれる．現代のLSIではCMOS型と呼ばれる基本論理回路が使われているが詳細は省略する．

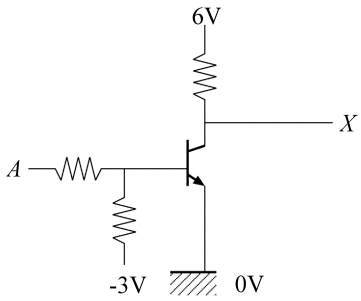
・ **論理積（AND）**



・ **論理和（OR）**



・ **否定（NOT）**



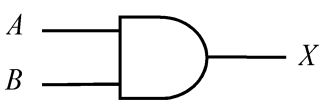
3.3.4　論理ゲートシンボル

基本論理関数の論理回路を単純な図形（記号）で表現したものを論理ゲートシンボル（または単に論理ゲート）という．AND，OR，NOTの他に，NAND（ANDの否定），NOR（ORの否定），XOR（排他的論理和），XNOR（XORの否定）がある．

注）論理ゲートシンボルおよびこれらで構成した論理回路では，左側に独立変数（入力信号），右側に従属変数（出力信号）を書く．

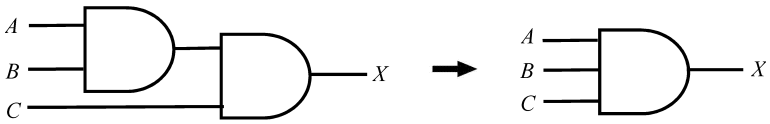
・ **論理積（AND）**

2入力ANDゲート



$$X = A \cdot B$$

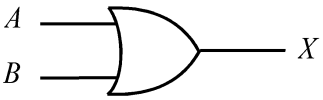
3入力ANDゲート



$$X = A \cdot B \cdot C$$

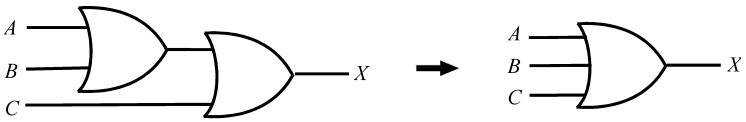
・ 論理和（OR）

2入力ORゲート



$$X = A + B$$

3入力ORゲート



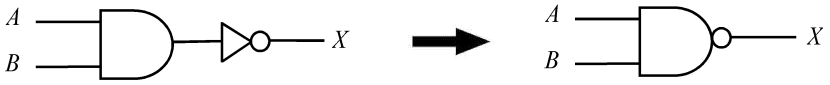
$$X = A + B + C$$

・ 否定（NOT）：NOT ゲート

$$X = \overline{A}$$

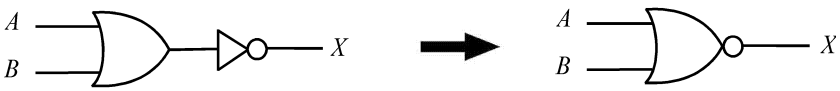
NOTゲートのことを**インバータ**（反転回路）ともいう

・ 論理積(AND)の否定：NANDゲート



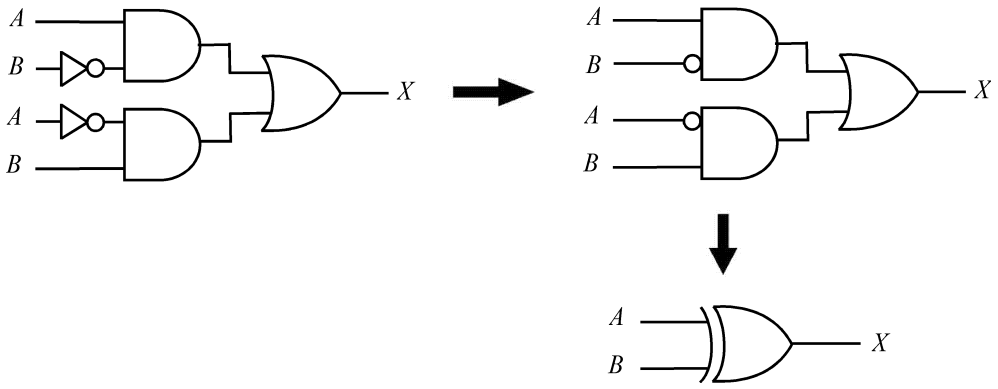
$$X = \overline{A \cdot B}$$

・ 論理和(OR)の否定：NORゲート



$$X = \overline{A + B}$$

・ **排他的論理和（XOR: exclusive OR）：XORゲート**



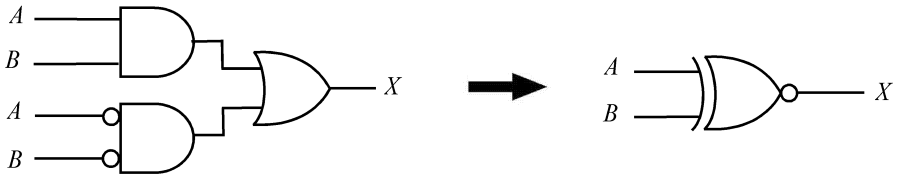
$$X = A\cdot\overline{B} + \overline{A}\cdot B = A \oplus B$$

XORゲートは A と B が‘0’と‘1’またはその逆で互いに異なるとき X が‘1’となるため**不一致回路**ともいう

注１）XORの論理演算記号“⊕”は“(+)”で代用してもよい.

注２）NOTゲートをANDゲートやORゲートに付加する場合は，否定を示す○印を使用して簡略化できる.

・ **排他的論理和の否定（XNOR: exclusive NOR）：XNORゲート**



$$X = \overline{A\cdot\overline{B} + \overline{A}\cdot B} = A\cdot B + \overline{A}\cdot\overline{B} = \overline{A \oplus B}$$

XNORゲートは A と B が‘0’と’0’または‘1’と‘1’で同じとき X が‘1’となるため**一致回路**ともいう

注） A の否定 $\overline{A}$ はスラッシュを用いて $A/$ または $/A$ と書いてもよい.

同様に $\overline{A\cdot B}$ は $(A\cdot B)/$ または $/(A\cdot B)$， $\overline{A+B}$ は $(A+B)/$ または $/(A+B)$，

A [NAND] B， A [NOR] B と書いてもよい.

例題3.3　2入力のNANDゲート，NORゲート，XOR ゲート，XNOR ゲートの真理値表をつくれ.

A	B	$\overline{A\cdot B}$	$\overline{A+B}$	$A\oplus B$	$\overline{A\oplus B}$
0	0				
0	1				
1	0				
1	1				

例題3.4 次の論理式に相当する論理回路を論理ゲートシンボルを用いて描け.

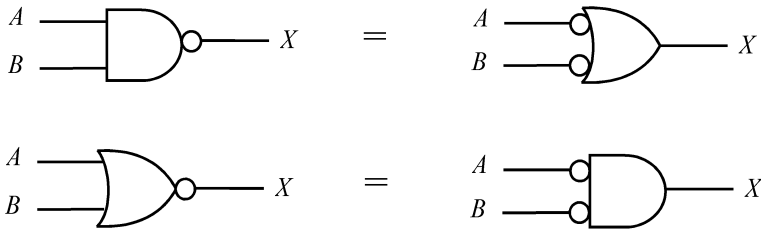
$$X = A\cdot\overline{B} + C\cdot(\overline{A+B})$$

問題3.1　次のド・モルガン則と呼ばれる次の２つの論理式が成立することを，下記の真理値表を作成して確かめよ.

$$\overline{A\cdot B} = \overline{A} + \overline{B}, \quad \overline{A+B} = \overline{A}\cdot\overline{B}$$

A	B	$A\cdot B$	$A+B$	$\overline{A}$	$\overline{B}$	$\overline{A\cdot B}$	$\overline{A}+\overline{B}$	$\overline{A+B}$	$\overline{A}\cdot\overline{B}$
0	0								
0	1								
1	0								
1	1								

注１）ド・モルガン則を論理ゲートシンボルで示すと以下となる.



ド・モルガン則は，あらゆる論理回路がORゲートとNOTゲートの組み合わせ（2入力NORゲートのみ），またはANDゲートとNOTゲートの組み合わせ（2入力NANDゲートのみ）で実現できることを意味している.

注２）ド・モルガン則から，ANDゲートが出力側にNOTゲートをもつ場合（NANDゲート）と入力側にNOTゲートをもつ場合は論理関数の意味が異なる．同様に，ORゲートが出力側にNOTゲートをもつ場合（NORゲート）と入力側にNOTゲートをもつ場合は論理関数の意味が異なる．すなわち,

$$\overline{A\cdot B} \neq \overline{A}\cdot\overline{B}, \quad \overline{A+B} \neq \overline{A}+\overline{B}$$

問題3.2　2入力変数の論理関数で名称が決まっているものは全部でいくつあるか. また, それらすべての論理式と論理ゲートシンボルを示しなさい.

注）AND 演算子“・”を次のように省略して記述する方法もあるが，NOT 演算子“¯”（否定記号：バー）が続くと見にくくなるため，ここでは省略なしとする.

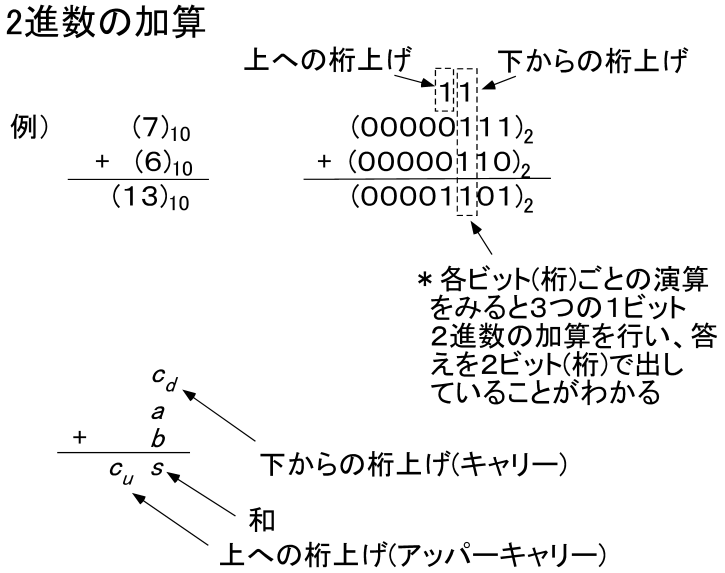
$$A \cdot B = AB, \quad \overline{A \cdot B} = \overline{AB}, \quad \overline{A} \cdot \overline{B} = \overline{A}\ \overline{B}$$

3.4 算術演算回路

この節では，2 進数加算（足し算）や減算（2 の補数加算）などの算術演算回路が 2 値論理の論理関数，すなわち論理ゲートシンボルを組み合わせた論理回路で実現されることを学ぶ.

3.4.1　2 進数加算の分析

8 ビット（1 バイト）2 進数データワードの算術加算（足し算）を考える.



ここで,

a ：今考えているビット（桁）の被加数の独立変数

b ：そのビットの加数の独立変数

c_d ：そのビットの下位ビットからの桁上がり（carry from downward bit）の独立変数

s ：そのビットの和（sum）の従属変数

c_u ：そのビットから上位ビットへの桁上がり（carry to upper bit）の従属変数

である.

最下位ビット（LSB: least significant bit）どうしの加算ではc_dは‘0’とし，最上位ビット（MSB: most significant bit）どうしの加算ではc_uは切り捨てる（無視する）．

3.4.2　1ビット分の加算回路（全加算器：Full Adder）の作成

上記の３つの独立変数に含まれる‘1’を２進数加算（‘1’の個数を計数）し，結果を２つの従属変数（2 ビット）で表現する回路を**全加算器（Full Adder）**という．

加算処理におけるすべてのビット値の組み合わせを数式で示すと以下となる．

$c_d + a + b = c_u s$

0 + 0 + 0 = 00

0 + 0 + 1 = 01

0 + 1 + 0 = 01

0 + 1 + 1 = 10

1 + 0 + 0 = 01

1 + 0 + 1 = 10

1 + 1 + 0 = 10

1 + 1 + 1 = 11

注）上記は数式であるため“+”記号は算術加算で，論理和（OR）演算ではない

真理値表の作成：　これらの数式から真理値表を作成する．（上記の数式における数値の並びが真理値表のビット値にそのまま対応する）．

c_d	a	b	c_u	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

論理式の導出：　真理値表の従属変数sとc_uについて，それぞれ独立に正論理論の論理式を導出する．

- 従属変数のビット値が‘1’の行（それぞれ 4 行ある）に着目する
- その行の 3 つの独立変数c_d，a，bについてビット値が‘0’の独立変数には否定記号“¯”（バー）をつけて論理積(AND)の項（積項という）を作成する

・ 各行について作成した積項を論理和(OR)で結合する

このようにして作成した論理式を**積項の和形式**または**積和標準形**という.

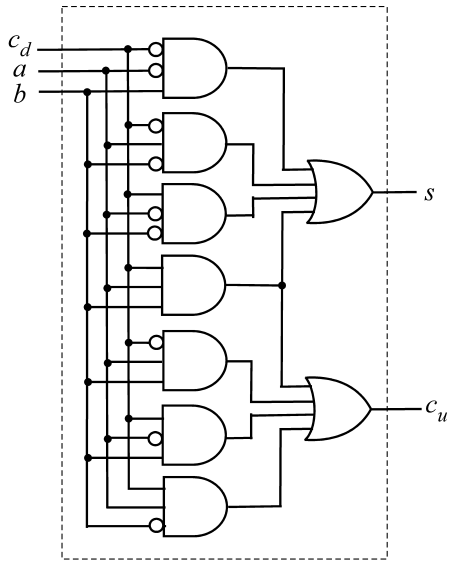
$$s = \overline{c_d} \cdot \overline{a} \cdot b + \overline{c_d} \cdot a \cdot \overline{b} + c_d \cdot \overline{a} \cdot \overline{b} + c_d \cdot a \cdot b$$
$$c_u = \overline{c_d} \cdot a \cdot b + c_d \cdot \overline{a} \cdot b + c_d \cdot a \cdot \overline{b} + c_d \cdot a \cdot b$$

注)従属変数のビット値が'1'の行にのみ着目するのは,正論理では対偶関係の真(成立する）と偽（成立しない）のうち真の場合にのみ着目すれば十分であるからである.

論理回路の作成

論理式の否定記号を NOT ゲート（○印），積項を AND ゲート，論理和を OR ゲートで表現して論理回路を得る.

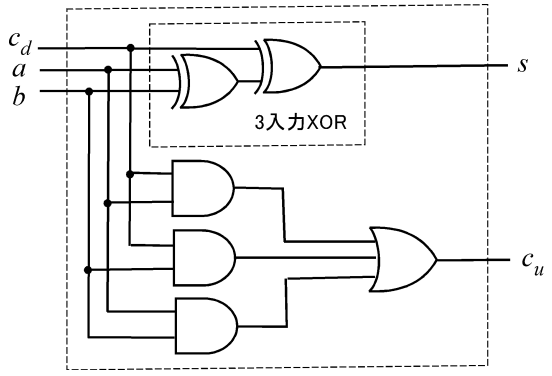
全加算器（Full Adder）：積項の和形式



注）積項 $c_d \cdot a \cdot b$ の AND ゲートの出力は共通使用している

参考：全加算器は次のように簡略化できる.

簡略化した全加算器（Full Adder）



例題 3.5 なぜこのような簡略化が可能なのか真理値表を見て考えてみよ．

ヒント：3 入力 XOR ゲートは入力の 3 つの独立変数 c_d，a，b のうち奇数個（1 個または 3 個）のビット値が‘1’のとき出力の従属変数 s のビット値が‘1’となる．従属変数 c_u はどのようなとき‘1’になるのか？

注）簡略化した回路は 1 種類とは限らない．

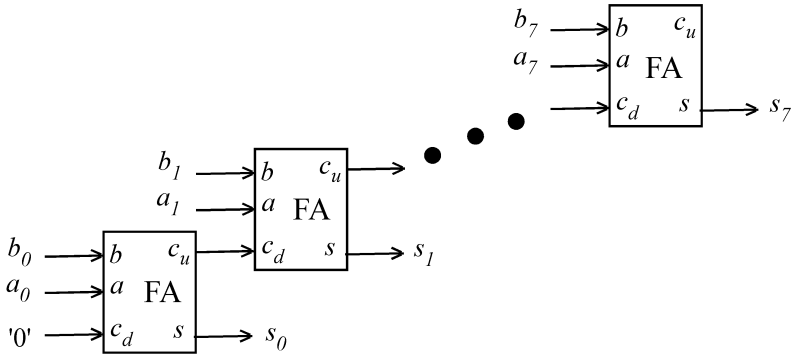
3.4.3 8 ビット加算回路

8 ビット 2 進数加算におけるデータワードの独立変数と従属変数を次のように定義する．

$$\begin{array}{rl} & (a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0)_2 \\ + & (b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)_2 \\ \hline & (s_7\ s_6\ s_5\ s_4\ s_3\ s_2\ s_1\ s_0)_2 \end{array}$$

8 ビット加算回路（Byte Adder）は全加算器 8 個を次のように接続して構成される．

8 ビット加算回路（Byte Adder）

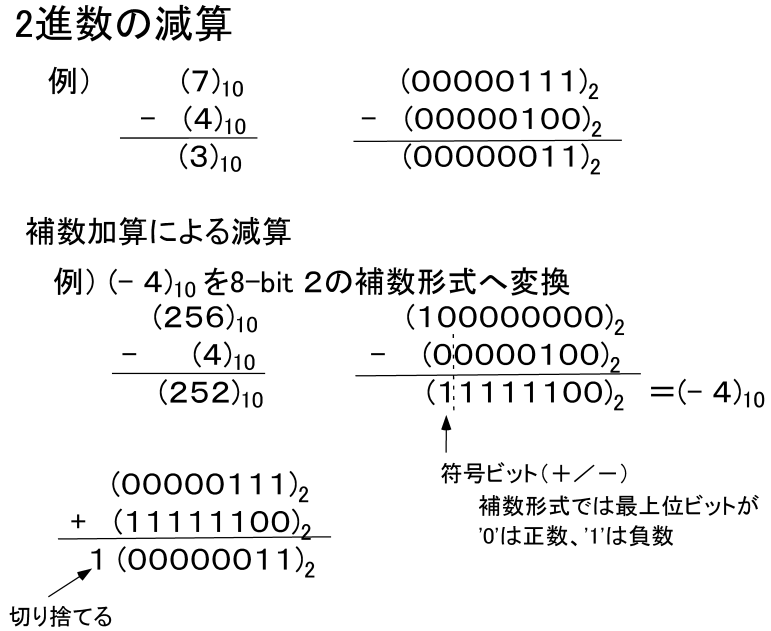


注）このような構成で任意のビット幅の加算回路を構成できる．ただし，この構成は桁上げ伝播型（ripple carry）と呼ばれ，8 ビット以上では遅延が大きくなるため，桁上げ先見型（carry look ahead）という構成が用いられる（詳細は省略）．

3.4.4　減算のための論理回路：2 の補数回路

コンピュータでは，正（ゼロおよび＋）と負（－）の場合がある数値データは「2 の補数形式」で表現する．減算は，減ずる数値（引く数値）が正の場合は負に，負の場合は正に **2 の補数変換**して加算により実行している．

（2 の補数形式の詳細は補足編の 0.7 節を参照せよ）



2 の補数形式では，最上位ビット（MSB）は正／負を表す符号ビットで，‘0’ならゼロまたは正，‘1’なら負である．8 ビット 2 の補数形式では，MSB を除いた 7 ビットがデータビットで，8 ビット全体で$(-127)_{10}$～$(127)_{10}$を表記できる．

この節では，加算回路と同様な方法で 8 ビットの 2 の補数変換回路を構成する．

8 ビット分の独立変数と従属変数の定義

上図に沿って 8 ビット分の独立変数と従属変数を定義する．

	元の数（独立変数）		変換結果（従属変数）
	$(b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)_2$	⇒	$(h_7\ h_6\ h_5\ h_4\ h_3\ h_2\ h_1\ h_0)_2$
例：	$(0000\ 0100)_2$		$(1111\ 1100)_2$

1 ビット分の 2 の補数変換回路（Bit CMPL: bit complimentary circuit）の作成

2 の補数変換の変換規則**「最下位ビット（LSB）から最初の‘1’まではそのまま，それ以上は‘0’／‘1’反転」**を 1 ビット単位で実行する論理回路（Bit CMPL）を実現するにあたり，次の独立変数，従属変数を定義する．

b ：今考えているビット（桁）の元の数の独立変数

c_d ：そのビットの下位に‘1’があるかどうかを示す独立変数

h ：そのビットの変換結果の従属変数

c_u ：そのビットおよび下位ビットに‘1’があったかどうかを示す従属変数

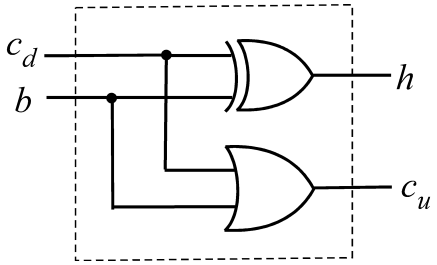
Bit CMPL の真理値表は以下となる．

c_d	b	c_u	h
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

上記の真理値表から論理式は以下となる．

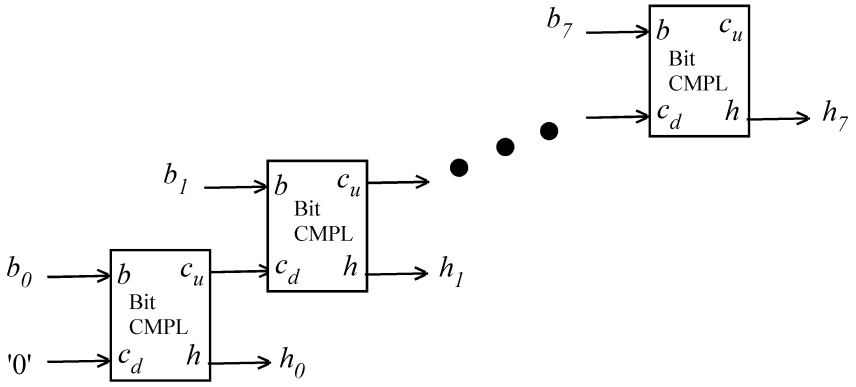
$$h = \overline{c_d} \cdot b + c_d \cdot \overline{b} = c_d \oplus b$$
$$c_u = c_d + b$$

上記の論理式から Bit CMPL の論理回路は以下となる．



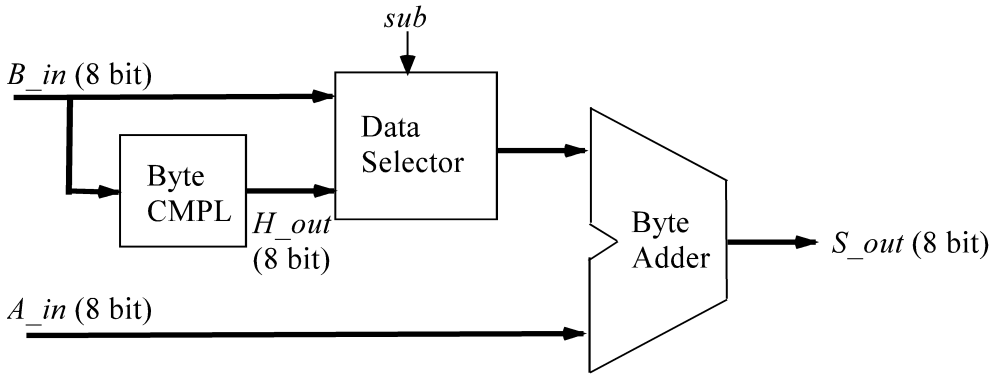
8 ビット 2 の補数変換回路（Byte CMPL）の作成

Byte CMPL は Bit CMPL8 個を次のように接続して構成される．



注）このような構成方法で任意のビット幅の 2 の補数変換回路を実現できる．

参考：8 ビットプロセッサの ALU には次のような構成の 8 ビット加減算回路がある．（16 ビットプロセッサ，32 ビットプロセッサもビット幅が異なるだけで構成は同じ．）



注：Data Selectorは*sub*信号が'0'のときは*B_in*
'1'のときは*H_out*を選択するスイッチ回路

問題3.3　全加算器（Full Adder）の真理値表，論理式，論理回路（積項の和形式）を示せ．また，真理値表の従属変数の論理値を何故そのように設定するのかを説明せよ．

問題 3.4　8 ビット加算回路（Byte Adder）の構成方法を図示せよ．

問題3.5　全加算器（Full Adder）から独立変数 c_d を除去した論理回路を半加算器（Half Adder）という．この回路は8ビット加算回路（Byte Adder）の最下位ビットのFull Adderの代わりに使用できる．半加算器（Half Adder）の真理値表，論理式，論理回路を示せ．

問題3.6　1ビット分の2の補数変換回路（Bit CMPL）の真理値表，論理式，論理回路を示せ．また，真理値表の従属変数の論理値を何故そのように設定するのかを説明せよ．

問題3.7　8ビット2の補数変換回路（Byte CMPL）の構成方法を図示せよ．

応用問題3.1　8ビットの2の補数は元の数値の全ビットを‘0’／‘1’反転し（これを1の補数という），$(0000\ 0001)_2$を加算して作ることもできる．3.4.3の8ビット加算回路（Byte Adder）に論理ゲートを適宜追加し，変数名を変更するなどして，この方法による8ビット2の補数変換回路のブロック図を作成せよ．

（注：この方法による2の補数変換回路は，回路規模が大きい加算回路が必要な

ため，一般には使用しない.）

応用問題3.2　4ビット乗算回路のブロック図を示せ．但し，4ビット加算回路は１ブロックとして書く．すべて正数として処理する．

ヒント：積は8ビットで求まる．2進乗算の筆算（補足編0.4節参照）と同様の方式で行える．

3.5 まとめ

・数値はコード化によりビットで表現される
・ビットは電気的状態で実現される
・電気的回路では論理演算を実現できる
・数値演算は論理演算の組み合わせで表現できる

第４章　コンピュータにおける計算処理のしくみ

コンピュータの基本構成については 1.6 で述べた．この章では，命令による具体的な計算処理のしくみについて学ぶ．（コンピュータの構成は 1.6 の図を参照）

4.1 メインメモリの構成

コンピュータのメインメモリ（主記憶装置）には，機械命令プログラムと処理対象のデータが格納されている．

- 機械命令プログラムは，複数の命令ワード（命令語ともいう）を実行する順番に並べたものである．命令ワードのサイズのことを命令ワード長といい，通常のコンピュータでは 1 バイトから 6 バイト程度まで複数種類の長さのものを併用している．それぞれの命令のワード長は命令コードから判別できるようになっている．
- 処理対象のデータは，データワードを単位に区切られて並べられている．データワードのサイズのことをデータワード長といい，プロセッサ（CPU）の ALU で実行する加算などの演算のビット幅に対応している．プロセッサ（LSI）は演算のビット幅によって次のように分類されている．
  - 8 ビットプロセッサ（データワード長 1 バイト）: マイコン
  - 16 ビットプロセッサ（データワード長 2 バイト）: 組込みプロセッサなど
  - 32 ビットプロセッサ（データワード長 4 バイト）: 一般的なパソコン
  - 64 ビットプロセッサ（データワード長 8 バイト）: 高性能パソコン／ワークステーション，サーバ，ゲーム機など
- 通常のコンピュータのメインメモリは，**記憶領域が 1 バイト単位に仕切られた構造**になっていて，この仕切りごとに**番地（アドレス）**が付けられている．これを**バイトマシン**という（**注：この目的は様々な長さの命令ワードに対応するため**）．メインメモリからのデータワードの読み出しは，先頭番地を指定するとデータワード長分のバイト数を並列に読み出すようになっている．データワードの書き込みも同様である. 命令ワードについては, 最大ワード長分のバイト数を 1 回で読み出すもの，命令ワード長がデータワード長を超える場合は 2 回に分けて読み出すものなど様々である．
- 一方，メインメモリの**記憶領域が複数バイト単位に仕切られていて**，この仕切りごとに番地（アドレス）が付けられているものもある．このタイプは**ワードマシン**と呼ばれている．（後述する NT-Processor V1 は 2 バイトワードマシンで，データワードも命令ワードも 2 バイト長である）
- なお計算処理においては，規模が大きいデータに対してはプログラム処理で複数のデータワードを組みにして扱えるようになっている．例：2 ワード演算（倍精度演算ともいう），同様に多倍長演算，配列を用いた可変長ワードなど．

4.2 メインメモリの容量と番地

- メインメモリの記憶領域の大きさ（容量）は記憶できるワード数（語数）で表現する．バイトマシンであれば **64MB, 256MB** などのように表現する（「M」はメガ，「B」はバイトの略号）．
- 番地は 0 番地から付けられている．バイトマシンの場合，メモリ容量が 100B であれば，0 番地から 99 番地までを付与する．

  注）
  - 「K(キロ)」は2^{10} = 1,024 （約10^3，約千）
  - 「M(メガ)」は$(2^{10})^2$ = 1,048,576 （約10^6，約百万）
  - 「G(ギガ)」は$(2^{10})^3$ = 1,073,741,824 （約10^9，約10億）
  - 「T(テラ)」は$(2^{10})^4$ = 1, 099,511, 627,776 （約10^{12}，約1兆）

問題 4.1 以下の問に答えよ．

(1) 1バイトで何種類の記号がコード化できるか．

(2) 1バイトデータは16進数何桁で表現できるか．

(3) 16進数が16桁ある．これは何バイト分にあたるか．

4.3 番地指定（アドレス指定）

- メインメモリからデータを取り出したり格納したりするためには，どの番地を使用するかを指定する必要がある．これを「番地指定」といい，メインメモリを操作する命令では，どこの番地を対象にしているかという情報が含まれる．
- 番地指定ではメインメモリの容量だけある番地の中から１つを指定するのでコード化と同じビット数が必要になる．

  例）バイトマシンで，メインメモリ容量が64MB であれば64M = 2^{26} なので番地も2^{26}個ある．そのうちの１つのバイトを指定するには26ビット必要．

問題 4.2 以下の問に答えよ．

(1) バイトマシンで，メインメモリ容量が256MB であれば１つのバイトを指定（番地指定）するのに何ビット必要か．

(2) 256MBの最後のバイトの番地は10進数でいくつか（$2^?$ を使用してよい）．

(3) 256MBのメインメモリの番地を16 進数で表現すると何桁になるか．

(4) 16ビットでは何バイトのメモリの番地指定ができるか．

4.4 メモリ内容の表示方法

メインメモリの大量の記憶内容をまとまった形で表示するには下例のような記憶ダンプ表という形式が用いられる．左側のADDR （アドレス）の下の16進数は，その右の STORAGE CONTENTS の１行のバイト列の左端の先頭番地を示す（値はすべて16進数）.

記憶ダンプ表（バイトマシンの例）

ADDR	STORAGE CONTENTS
001FD0	00123400 01234000 01256400 01278400
001FE0	00100000 01245000 0125C400 0127C400
001FF0	00100000 01246000 01260400 01270400
002000	00B02000 00B01FE2 00B01FC1 00B01F83

例題 4.1 上の記憶ダンプ表について答えよ.

(1) １行に何バイトの記憶が表示されているか.

(2) 001FDA 番地のバイトの内容は何か.

(3) 内容が12 (16進) のバイトの番地は何か.

問題 4.3 上の記憶ダンプ表について答えよ.

(1) 全部で何バイトの記憶が表示されているか.

(2) 001FE6 番地のバイトの内容は何か.

(3) 内容が23 (16進) のバイトの番地は何か.

4.5 命令と実行

4.5.1 命令とは

・プロセッサ（CPU）で実行する処理の最小単位の内容を決めている情報を**機械命令**という.

・機械命令は特定のビット列のコードで表現されている．このコード（**機械命令ワード**または**機械語**という）を特定の目的のためにメインメモリに並べたものが**機械命令プログラム**である.

・CPUではメインメモリ（主記憶）から機械命令を１つずつ読み出して実行する.

・機械命令プログラムを作成する際に, 機械命令のコードのままでは記述が困難な場合, 人間が理解しやすい別のコードに対応づけることがある．これを**アセンブラ表記**という．ただし，プロセッサ（CPU）で実行する際はすべて機械命令に変換する.

4.5.2 命令処理のサイクル

プロセッサ（CPU）では１個の機械命令ワードごとに次の６つの処理を順に実行する．この一連の処理を「**CPUの命令サイクル**」という．以下に，加算命令など一般的な演算命令での処理を示す．

(1) **命令読み出し（命令フェッチ）**：命令アドレスレジスタ（プログラムカウンタともいう）に保持しているメインメモリの番地（アドレス）から機械命令ワードを読み出して命令レジスタに格納（一時記憶）する．**（注：フェッチとは取り出す意味）**

(2) **命令デコード（解読）**：命令レジスタの機械命令ワードのコードをシーケンサで解読し，CPU内部の制御信号やデータレジスタのワード番号，データワードを格納しているメインメモリの番地（アドレス）を生成する．

(3) **データ読み出し（オペランドフェッチ）**：データレジスタ（汎用レジスタ）またはメインメモリから処理対象のデータワードをバスへ読み出す．

(4) **演算実行**：データワードをALUに入力し，デコードした機械命令で指定している演算を実行する．

(5) **結果の格納**：ALUから出力される演算結果のデータワードをバスを経由してデータレジスタ（汎用レジスタ）に格納する．

(6) **次の命令番地（アドレス）の決定**：現在実行している機械命令ワードの次に実行する機械命令ワードが格納されているメインメモリの番地（アドレス）を決定し，命令アドレスレジスタに格納する

（注：ただし (6) は，通常の命令では (4) の開始時に実行し，(4)と(5)の期間に次の命令の(1)を平行に実行する．）

以上の処理を繰り返すことを「**逐次制御方式**」といい，現在のデジタル計算機の標準的な処理方式となっている．

現在の汎用PCやサーバのプロセッサは1GHz程度（G: giga，10億(≒2^{30})）のクロック（基本制御信号）で動作しているが，１つの命令サイクルは10クロック程度で実行される．このときプロセッサの性能は100MIPS（mega instruction per second）となる．

一方，書き換え可能論理LSI（FPGA: field-programmable gate array）を用いた組込みプロセッサではメモリ回路で論理回路を実現しているため20MHz程度のクロックで動作しているが，１つの命令サイクルは4クロック程度で実行されている（使用している命令の種類が少なく命令デコードを1クロックで実行できるため）．このときプロセッサの性能は5MIPSとなる．

問題 4.4　CPUの命令サイクルで行う６つの処理を実行可能な順で挙げよ．

4.5.3 命令の種類

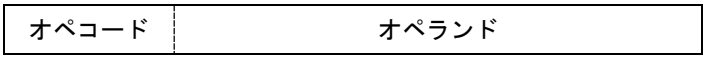
機械命令の種類はプロセッサごとに決まっている．これを命令セットという．ただし，どのようなプロセッサでも概ね次のような命令を備えている．

- **算術演算命令**：加算命令，減算命令，比較命令など
- **論理演算命令**：AND命令，OR命令，XOR命令，NOT（反転：インバート）命令など
- **ビット列操作命令**：上位／下位シフト命令，上位／下位回転シフト命令など
- **データ移動命令**：データレジスタ（汎用レジスタ）のワード間移動命令（注：データレジスタは通常複数ワード分の記憶容量をもつ），メインメモリからのデータ読み出し／書き込み命令など
- **分岐命令**：命令アドレスレジスタ（プログラムカウンタ）に保持している次に読み出す命令のメインメモリの番地を離れた（分岐先）番地に強制的に書き換える強制分岐命令と，算術演算結果（正，負，ゼロなど）によってこれらの番地を切り替える条件分岐命令がある．
- **特殊命令**：空命令（何もしない命令），一時停止命令，終了停止命令など
- **その他**：特定の制御用レジスタに対してデータ書き込み／読み出しを行う命令，カウント機能をもつ命令など

4.5.4 命令の構造

１つの機械命令のビット列は次の２つの部分からなっている．

- **オペコード**（オペレーションコードの略，OPコードまたは命令コードとも書く）：加算やデータ移動などの操作（演算）の種類を指定する．また次のオペランドの形式およびそれが指すデータの形式も指定している．
- **オペランド**（操作（演算）されるものという意味）：操作（演算）の対象となるデータの所在，結果を格納する場所などを指定する．



オペランド部に記述する内容には，

1) メインメモリ（主記憶）の番地
2) データレジスタ（汎用レジスタ）のワード番号
3) 即値（処理対象データそのもの）

などがある．

ほとんどのプロセッサは複数のオペランドを持つ機械命令を使用している．このとき，オペランドの個数，各オペランドの意味と形式はすべてオペコードによって決まっている．

オペコード	第１オペランド	第２オペランド	・・・

なお，現在のプロセッサでは２つのオペランドをもつ命令形式が主流になっている．例えば算術加算命令では，被加数と加数のデータワードをそれぞれ指定する必要があるが（**二項演算**という），第１オペランドで被加数，第２オペランドで加数，同じく第１オペランドで加算結果の格納先を指定している（演算後に第１オペランドのデータは書き換わる）．入力データのオペランドを**ソースオペランド**，出力（演算結果）を書き込むオペランドを**デスティネーション（宛先）オペランド**という．NOT命令やデータ移動命令ではソースオペランドは１つであるが（**単項演算**という），第２オペランドでソースを指定し，第１オペランドでデスティネーションを指定している．

このような機械命令の構造は，プロセッサの構造（**プロセッサアーキテクチャ**という），特にバスの本数に依存している．

4.5.5 オペランドの指定方式

機械命令でオペランドを指定する方式として次の２つがある．

- **オペランド明示方式**：オペランドをすべて命令ワード内に記述する方式．命令ワードは長くなるが命令デコードが簡単で高速化できる．ワード数が多いレジスタ群をデータレジスタとしてもつ方式（**汎用レジスタ方式**という）のプロセッサや，単純な機能の命令を少数もつ方式のプロセッサ（RISC: reduced instruction　set computerという）ではこの方式を用いている．
- **オペランド暗黙指定方式**：オペコードによって使用するデータレジスタの一部または全部を指定してこれらのオペランドを命令ワード内に記述しない方式．Intel社の8ビットマイコン8085では，算術演算命令や論理演算命令での第１ソースオペランドとデスティネーションオペランドを特定のデータレジスタになるようにして第２ソースオペランドのみを指定するようになっている（これを**アキュムレータ方式**という．バスは１本）．また，データレジスタ間のデータ移動命令については，ソースオペランドとデスティネーションオペランドのすべての組み合わせごとに互いに異なるオペコードのみの命令を用意してある．この方式では，ワード長が小さい機械命令を組み合わせてプログラムを記述できるためプログラムが使用するメインメモリの記憶領域を小さくできるが，命令デコードが複雑になるため命令サイクルのクロック数が多くなる．複雑な機能の命令を多数もつ方式のプロセッサ（**CISC: complex instruction set computer**という）の多くはこの方式の命令とオペランド明示方式の命令を併用している．

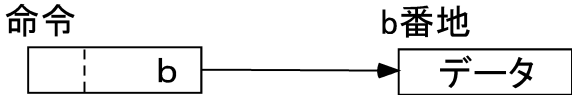
4.5.6 番地指定方式

オペランドでメインメモリ（主記憶）の番地（アドレス）を与えて操作（演算）に用

いるデータワードを指定することを**番地指定（アドレス指定）**という．このときオペランドに書く値と番地との対応についていくつかの方式があり，これを**番地指定方式（アドレッシングモード）**という．例えば，汎用プロセッサでは，同じ加算処理を行う場合でもオペコードが異なる複数の加算命令があり，それぞれ異なる番地指定方式を用いている．通常は，第１オペランドはデータレジスタ（汎用レジスタ）の番号，第２オペランドでメインメモリアクセス（読み書き）のための番地指定を行う．

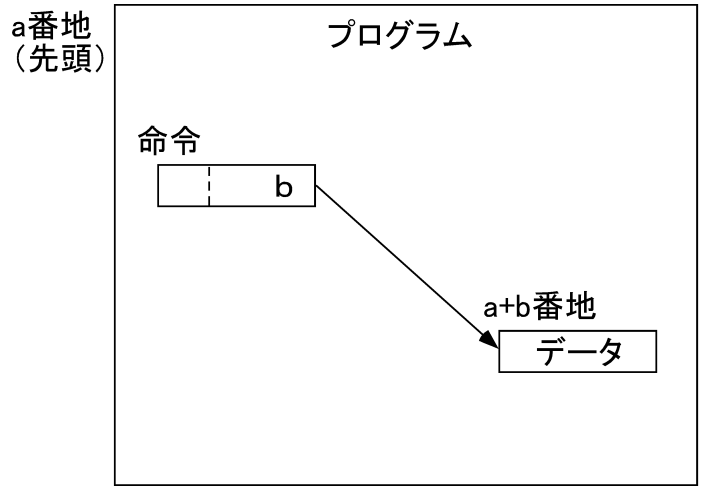
以下の 1）～ 6）の図の命令では，オペコードと番地指定を行う１つのオペランド（通常は第２オペランド）のみからなる構造を示す．

1） **直接番地方式**： 番地をそのままオペランドにする．オペランドのコードにはメインメモリの容量に応じたビット数が必要となる．

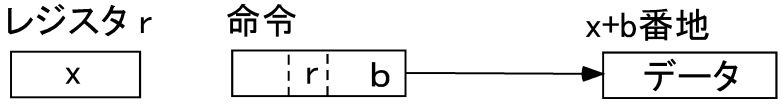


2） **命令相対番地方式**：この方式の命令が格納されている番地とその命令で用いるデータワードが格納されている番地の差分（変位という）すなわち相対位置をオペランドで示す方式．

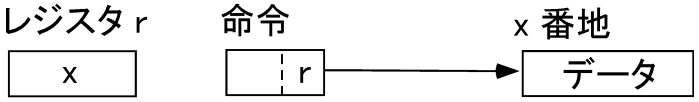
3） **プログラム相対番地方式**： プログラムに応じてその先頭番地が書かれたレジスタがあり，そこからの変位をオペランドで示す方式．



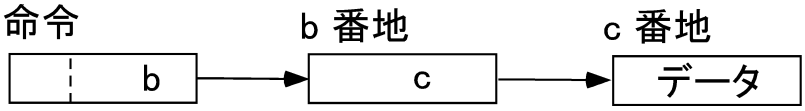
4） **レジスタ相対番地方式**： 基準となる番地が書かれたレジスタ（インデックスレジスタという）があり，そのレジスタの番号とそこからの変位をオペランドで指定する方式．



5) レジスタ間接番地方式： 番地をレジスタに入れてオペランドではレジスタの番号だけを指定する．レジスタにデータレジスタ（汎用レジスタ）を用いるものが多い．これにより，レジスタ内の番地に一定値を加算することで連続番地の複数のデータワードへのアクセス（読み書き）が簡単に行える．



6) メモリ間接番地方式： オペランドが指定する番地の内容をデータではなくデータワードの番地とみなしてその番地からデータを取る方式．１命令サイクルで，命令読み出し（命令フェッチ），データワードの番地の読み出し，データワードの読み書きの３回のメインメモリへのアクセスが必要なため特殊なプロセッサでのみ使われている．



注１）オペランドで番地指定の代わりに即値を指定する命令もある．

注２）直接番地方式とメモリ間接番地方式以外ではオペランドのビット数を少なくして命令長を短縮する（プログラムサイズの縮小）効果がある．

注３）後で演習に使用するNT-ProcessorV1プロセッサでは，メインメモリ容量が256ワード（512バイト）と小さいため，8ビット（1バイト）で番地指定を行う直接番地方式のみを使用している．その拡張版のNT-ProcessorV1plusプロセッサではメインメモリ容量を最大256×256ワード（64kワード）まで拡張できる．256ワードを１ページとして，256ページの１つを指定する8ビットの上位アドレスを指定してインデックスレジスタに書き込む命令とアドレス部が8ビットの直接番地方式の命令をペアにして用いる方法を採用している（レジスタ相対番地方式の一種）．また，16ビットの汎用レジスタに16ビットのアドレスを書き込んでメインメモリをアクセスするレジスタ間接番地方式の命令も備えている．

問題 4.5　命令のオペコードで指定するものをすべて挙げよ.

問題 4.6　以下の問に答えよ.

(1) 命令のオペコードのビット数が8であれば何種類の命令を定義できるか.

(2) 命令のオペランドで8種類のレジスタのうち１つを指定するためには何ビット必要か.

(3) バイトマシンで，プログラム相対番地指定の命令のオペランドが12ビットであれば，何バイトのプログラム域が指定できるか.

(4) 800番地にある命令で1000番地のデータを操作したいとき, この命令の番地指定方式が次のようである場合オペランドで指定すべき値を書け．ただし，この命令の属するプログラムは600番地から入っているとし，数値は10進とする.

1) 直接番地

2) 命令相対番地

3) プログラム相対番地

(5) 主記憶が以下のような内容であったとする.

番地：200 400 600 700 800 900

内容：400 500 300 900 700 800

このとき，700番地にある加算命令のオペランドが200であるとし，この命令の番地指定方式が次のようであるとき，実際に加算される値を求めよ．なお，プログラムは600番地から始まるとする．また，命令でレジスタが指定される時は，そのレジスタには500が格納されているとする．数値はすべて10進とする

1) 直接番地

2) 命令相対番地

3) プログラム相対番地

4) レジスタ相対番地

5) メモリ間接番地

6) 即値

4.6 まとめ

- CPUによってどのような命令があるか（命令セット）が決まっている.
- １つの命令は決まった形式（コード化，ビット数）のデータしか処理しない.
- １つの命令は決まった番地指定方式によりオペランドでデータを指定する.

第 5 章　ALU で行う演算の 16 進表記

この章では，第 6 章で扱う16ビットプロセッサNT-ProcessorV1を対象としたプログラミング演習の準備として，ALU（算術論理演算ユニット）で行う演算の16進表記を学ぶ．このプロセッサのデータワード長は16ビット（2バイト）で，16進表記では4桁（4ニブル）で表現される．

5.1 算術演算

5.1.1 加算

2進表記で2の補数形式（16進表記では16の補数形式）での加算．第 1 オペランドが被加数，第 2 オペランドが加数となる．演算結果は第 1 オペランドに上書きされる．2進表記で数値のMSB（最上位ビット）が‘0’のときはゼロまたは正数，‘1’のときは負数．2の補数形式では，正数の場合でもデータビットは15ビットである点に注意．**16進表記**では**MSN（最上位ニブル）が0～7のときはゼロまたは正数，8～Fのときは負数**．なお，2の補数形式（16の補数形式）では符号ビットも計算対象になる点に注意．加算結果の正側の最大値は$(7FFF)_{16}$（$=(32,767)_{10}$），負側の最大値は$(8001)_{16}$（$=(-32,767)_{10}$）．以下の例では2進表記も示すが，16進表記のみで筆算できるようになることを目指す．

・正数どうしの加算例

$(0001\ 0010\ 0011\ 0100)_2$	$(1234)_{16}$	←第 1 オペランド
$+\ (0101\ 1010\ 1101\ 1111)_2$	$+\ (5ADF)_{16}$	←第 2 オペランド
$(0110\ 1101\ 0001\ 0011)_2$	$(6D13)_{16}$	

16進表記では，各桁（ニブル）の加算は10進数に変換して計算する．各桁の加算結果が $(15)_{10}$（$=(F)_{16}$）を超える（＞）と上位桁へ$(1)_{10}$（$=(1)_{16}$）桁上がりし，その桁の和は$(16)_{10}$を引いた値となる．（各桁の加算結果の最大値は下位からの桁上がりがある場合で$(31)_{10}$，この場合，上位桁へ$(1)_{16}$）桁上がりし，その桁の和は$(F)_{16}$となる）正数どうしの加算では結果のMSNが$\geqq(8)_{16}$になると負数となり，オーバーフロー（計算不能）となる．オーバーフローとは，2進表記でデータビットから符号ビットへの桁上がりによって符号を壊す（値を反転させてしまう）ことを指す．

・負数どうしの加算例

$(1110\ 0010\ 0011\ 0100)_2$	$(E234)_{16}$
$+\ (1100\ 1010\ 1101\ 1111)_2$	$+\ (CADF)_{16}$
$1\ (1010\ 1101\ 0001\ 0011)_2$	$1\ (AD13)_{16}$
↑切捨て	↑切捨て

正数どうしの場合と同様に計算する．ただし16進表記で5桁目への桁上がりは切り捨てる．切り捨て後の加算結果のMSNが$\leqq(7)_{16}$となると正数となり，オーバーフロー

となる．正確には加算結果が$(8000)_{16}$の場合もオーバーフローとする．

・正数と負数の加算例（切捨てありの例）

$(1110\ 0010\ 0011\ 0100)_2$	$(E234)_{16}$
$+\ (0111\ 1010\ 1101\ 1111)_2$	$+\ (7ADF)_{16}$
$1\ (0101\ 1101\ 0001\ 0011)_2$	$1\ (5D13)_{16}$
↑切捨て	↑切捨て

正数どうしの場合と同様に計算する．16進表記で5桁目への桁上がりがある場合とない場合があり，ある場合は切り捨てる．加算結果が正数になる場合と負数になる場合がある．オーバーフローは発生しない．

5.1.2 減算

ALUでは，減算は第２オペランドの減数（引く数）を2の補数（16の補数）に変換して加算により減算を実行するが，ここでは，はじめに通常の筆算と同じ方法での16進減算を行う方法を学ぶ．次いで16進表記による2の補数変換（16の補数変換）を学ぶ．

・正数どうしの減算で被減数＞減数の例

$(0011\ 1010\ 0101\ 0111)_2$	$(3A57)_{16}$ ←第１オペランド
$-(0001\ 1011\ 1101\ 1111)_2$	$-(1BDF)_{16}$ ←第２オペランド
$(0001\ 1110\ 0111\ 1000)_2$	$(1E78)_{16}$

16進表記では，各桁（ニブル）の減算は10進数に変換して計算するとよい．各桁の減算で被減数＜減数の場合は上位桁（ニブル）からの借り（borrow），すなわち上位桁では$-(1)_{16}$，その桁では$(16)_{10}$が発生する．

・正数どうしの減算で被減数＜減数の例

↓仮定する借り	↓仮定する借り
$1\ (0001\ 1011\ 1101\ 1111)_2$	$1\ (1BDF)_{16}$
$-(0011\ 1010\ 0101\ 0111)_2$	$-(3A57)_{16}$
$(1110\ 0001\ 1000\ 1000)_2$	$(E188)_{16}$

2進表記では17ビット目に借り$(1)_2$，16進表記では5桁目に借り$(1)_{16}$があることを仮定して，前記の「正数どうしの減算で被減数＞減数の例」と同様に計算を行う．

・負数から負数の減算・負数から正数の減算・正数から負数の減算

- **負数から負数の減算**：「正数どうしの減算で被減数＞減数の例」または「正数どうしの減算で被減数＜減数の例」と同様に行う．オーバーフローは発生しない．
- **負数から正数の減算**：「正数どうしの減算で被減数＞減数の例」と同様に行う．結果が正数の場合はオーバーフローである．

- **正数から負数の減算**：「正数どうしの減算で被減数＜減数の例」と同様に行う．結果が負数の場合はオーバーフローである．

・2の補数変換（16の補数変換）

$(0001\ 1010\ 0110\ 0000)_2$ $(1A60)_{16}$

↓ ↓

$(1110\ 0101\ 1010\ 0000)_2$ $(E5A0)_{16}$

2進表記における2の補数変換は**「LSBから見て最初の'1'まではそのまま，それ以上は'0'／'1'反転」**で行える．16進表記における16の補数変換は**「LSN（最下位ニブル）から見て最初の 0 以外の桁（ニブル）までの 0 はそのまま，最初の 0 以外の桁は$(16)_{10}$から引き，それ以上の桁は$(15)_{10}$から引く」**で行える．

これにより正数は絶対値が同じ負数に，負数は絶対値が同じ正数に変換される．補数変換を再度行うと元の数値に戻る．

$(1110\ 0101\ 1010\ 0000)_2$ $(E5A0)_{16}$

↓ ↓

$(0001\ 1010\ 0110\ 0000)_2$ $(1A60)_{16}$

2の補数変換・16の補数変換は次の減算を行うことと等価である．

$1\ (0000\ 0000\ 0000\ 0000)_2$ $1\ (0000)_{16}$

$-(0001\ 1010\ 0110\ 0000)_2$ $-(1A60)_{16}$

$(1110\ 0101\ 1010\ 0000)_2$ $(E5A0)_{16}$

注）NT-Processor V1 では，ALUで加算または減算を行うと演算結果に応じて，2ビットのコンディションコードレジスタCC（フラグレジスタともいう）に次の値が設定される．CCの内容は条件分岐命令で使われる．

演算結果が正数：$CC = (10)_2 = (2)_{10}$

演算結果がゼロ：$CC = (00)_2 = (0)_{10}$

演算結果が負数：$CC = (01)_2 = (1)_{10}$

演算結果がオーバーフロー：$CC = (11)_2 = (3)_{10}$

5.1.3 比較

第１オペランドと第２オペランドの数値の**符号を含めた絶対値**の大小関係を判定し，コンディションコードレジスタCC（フラグレジスタともいう）に次の値を設定する．CCの内容は条件分岐命令で使われる．比較する数値は変更されない．

＞：$CC = (10)_2 = (2)_{10}$

＝：$CC = (00)_2 = (0)_{10}$

＜：$CC = (01)_2 = (1)_{10}$

例１：正数どうしの比較

第１オペランド　$(3A57)_{16}$

第２オペランド　$(1BDF)_{16}$

＞：$CC = (10)_2 = (2)_{10}$

例２：正数と負数の比較

第１オペランド（正数）　$(3A57)_{16}$

第２オペランド（負数）　$(8BDF)_{16}$

＜：$CC = (01)_2 = (1)_{10}$

符号ビットを含めた絶対値比較であるため正数が小と判定される．

正数を大と判定するにはそれぞれの数値に$(8000)_{16}$を加算して比較する．

第１オペランド（正数）　$(BA57)_{16}$

第２オペランド（負数）　$(0BDF)_{16}$

＞：$CC = (10)_2 = (2)_{10}$

元の数値に戻すには再度$(8000)_{16}$を加算する．

第１オペランド（正数）　$(3A57)_{16}$

第２オペランド（負数）　$(8BDF)_{16}$

5.2 論理演算

２つの16ビット(2バイト)データワード間でビットごとに論理積(AND)，論理和(OR)，または排他的論理和（XOR）をとる演算（これらは二項演算）．演算結果は第１オペランドに上書きされる．単項演算である論理否定（NOT (Invert)）では，第２オペランドのデータワードに対して演算が行われ，演算結果は第１オペランドに格納される．論理演算では数値データは符号なし形式として扱われる．**手計算で論理演算を行うには，単純な場合を除いて2進表記で行う**．

5.2.1 論理積(AND)

・演算例

$(0001\ 0010\ 0011\ 0100)_2$	$(1234)_{16}$	←第１オペランド
AND $(0101\ 1010\ 1101\ 1111)_2$	AND $(5ADF)_{16}$	←第２オペランド
$(0001\ 0010\ 0001\ 0100)_2$	$(1214)_{16}$	

2進表記で第１オペランドのビットと第２オペランドの対応するビットがともに‘1’の場合にのみ演算結果のビットが‘1’になる．

・AND演算によるビット列マスク処理

2進表記で特定のビット列の値を‘0’にする操作を**マスク処理**という．次の例では第２オペランドに上位バイトがall ‘0’，下位バイトがall ‘1’のマスク値を設定してAND演算を行うことにより，第１オペランドの上位バイトをall ‘0’にしている

$(0001\ 0010\ 0011\ 0100)_2$　　$(1234)_{16}$　←第１オペランド

AND $(0000\ 0000\ \mathbf{1111\ 1111})_2$　　AND $(00\mathbf{FF})_{16}$　←第２オペランド

$(0000\ 0000\ \mathbf{0011\ 0100})_2$　　$(00\mathbf{34})_{16}$

5.2.2 論理和（OR）

・演算例

$(0\mathbf{0}01\ 00\mathbf{1}0\ 00\mathbf{11}\ 0\mathbf{1}00)_2$　　$(1234)_{16}$　←第１オペランド

OR　$(\mathbf{0}101\ \mathbf{1}010\ \mathbf{0}100\ 000\mathbf{1})_2$　　OR　$(5A41)_{16}$　←第２オペランド

$(\mathbf{0}101\ \mathbf{1}010\ 0\mathbf{111}\ 010\mathbf{1})_2$　　$(5A75)_{16}$

2進表記で第１オペランドのビットと第２オペランドの対応するビットの少なくとも一方が‘1’の場合，演算結果のビットが‘1’になる．

（注：OR演算記号は“+”であるが算術加算と間違えるため，ここでは使っていない）

・OR演算によるビット列合成処理

２つのデータワードを部分的に組み合わせる操作を**ビット列合成処理**という．次の例では第１オペランドの上位バイトをマスク処理でall ‘0’にしたデータワードと，第２オペランドの下位バイトをマスク処理でall ‘0’ にしたデータワードとのOR演算により，下位バイトと上位バイトを組み合わせたデータワードを生成している．

$(0000\ 0000\ \mathbf{0011\ 0100})_2$　　$(00\mathbf{34})_{16}$　←第１オペランド

OR　$(\mathbf{0101\ 1010}\ 0000\ 0000)_2$　　OR　$(\mathbf{5A}00)_{16}$　←第２オペランド

$(\mathbf{0101\ 1010\ 0011\ 0100})_2$　　$(\mathbf{5A34})_{16}$

5.2.3 排他的論理和（XOR）

・演算例

$(0\mathbf{0}01\ 00\mathbf{1}0\ 00\mathbf{11}\ 0\mathbf{1}00)_2$　　$(1234)_{16}$　←第１オペランド

XOR　$(\mathbf{0}101\ \mathbf{1}010\ \mathbf{0}100\ 000\mathbf{1})_2$　　XOR　$(5A41)_{16}$　←第２オペランド

$(0\mathbf{1}00\ \mathbf{1}000\ 0\mathbf{111}\ 010\mathbf{1})_2$　　$(4875)_{16}$

2進表記で第１オペランドのビットと第２オペランドの対応するビットが互いに‘1’と‘0’または‘0’と‘1’と異なるとき演算結果のビットが‘1’になる．

XOR演算では，2進表記でall ‘0’のデータワードとXORすると元の値が維持され，all ‘1’のデータワードとXORすると次のNOT演算と同様に各ビットの‘0’／‘1’が反転する．

$(1234)_{16}$　　$(1234)_{16}$

XOR　$(\mathbf{0000})_{16}$　　XOR　$(\mathbf{FFFF})_{16}$

$(1234)_{16}$　　$(EDCB)_{16}$

5.2.4 論理否定（NOT (Invert)）

・演算例

NOT $(0001\ 0010\ 0011\ 0100)_2$　　NOT $(1234)_{16}$　←第２オペランド

$(1110\ 1101\ 1100\ 1011)_2$　　$(EDCB)_{16}$　←第１オペランド

2進表記で各ビットの‘0’／‘1’が反転する．

NOT演算は次に示す$(FFFF)_{16}$からの減算（2進表記ではall ‘1’からの減算）と等価．

$$\begin{array}{rl} & (FFFF)_{16} \\ - & (1234)_{16} \\ \hline & (EDCB)_{16} \end{array}$$

数値データをNOT演算で‘0’／‘1’反転したものを**1の補数（16進表記では15の補数）**という．**2の補数（16進表記での16の補数）は1の補数に$(0001)_{16}$を加算しても得られる**．

5.3 ビット列操作

NT-Processor V1では次の4種類の1ビットシフト（桁ずらし）操作が可能．**手計算でビット列操作を行うには2進表記で行う．**第２オペランドのデータワードに対してシフト操作が行われ，結果は第１オペランドに格納される．

5.3.1　１ビット上位シフト

シフトするデータワードを2進表記の変数で示す．

$(\boldsymbol{b_{15}}\, b_{14}・・・b_1\, b_0)_2$　←第２オペランド

↓

$(b_{14}・・・・b_1\, b_0\ \ \mathbf{0})_2$　←第１オペランド

CC=$(0\ \boldsymbol{b_{15}})_2$

ビット列が1ビット上位シフト（左シフト）し，LSB（最下位ビット）に‘0’が入る．桁あふれした元のMSB（最上位ビット）はコンディションコードレジスタCCの下位ビットに入る．CCの上位ビットは‘0’になる．

16進表記でMSN（最上位ニブル）が$(3)_{16}$以下の正の数値データに対して1ビット上位シフトを行うと値が２倍になる．すなわち$(2)_{16}$の乗算処理と等価．（注：MSNが$(7)_{16}$の場合は2倍するとオーバーフローになる）

$(0001\ 0010\ 1010\ 0100)_2$　　$(12A4)_{16}$

↓　　↓

$(0010\ 0101\ 0100\ 1000)_2$　　$(2548)_{16}$

CC = $(00\ )_2 = (0)_{10}$

5.3.2　1ビット下位シフト

$(b_{15}\, b_{14}$・・・$b_1\, \boldsymbol{b_0})_2$　　←第２オペランド

↓

$(\boldsymbol{0}\, b_{15}\, b_{14}$・・・$b_1)_2$　　←第１オペランド

CC=$(0\ \boldsymbol{b_0})_2$

ビット列が1ビット下位シフト（右シフト）し，MSBに‘0’が入る．桁落ち（アンダーフロー）した元のLSBはコンディションコードレジスタCCの下位ビットに入る．CCの上位ビットは‘0’になる

正の数値データに対して1ビット下位シフトを行うと，値が1/2になる．すなわち$(2)_{16}$による除算処理と等価．（以下ではCCは省略）

$(0010\ 0101\ 0100\ 1000)_2$　　$(2548)_{16}$

↓　　↓

$(0001\ 0010\ 1010\ 0100)_2$　　$(12A4)_{16}$

CC = $(00\ )_2 = (0)_{10}$

注）数値データに対して符号ビットを保存したシフトを算術シフト，すべてのビットを一様にシフトすることを論理シフトというが，上記の1ビット上位／下位シフトは論理シフトである．CC設定は条件分岐命令と組み合わせて，1ビット上位シフトでは正数／負数の判定処理，1ビット下位シフトでは奇数／偶数の判定処理に応用できる．

5.3.3　1ビット上位回転シフト

$(\boldsymbol{b_{15}}\, b_{14}$・・・$b_1\, b_0)_2$　　←第２オペランド

↓

$(b_{14}$・・・・$b_0\, \boldsymbol{b_{15}})_2$　　←第１オペランド

ビット列が1ビット上位シフト（左シフト）し，桁あふれした元のMSBがシフト後のLSBに入る．CC設定はない．

この操作は次のような数値データ操作に応用される．

$(1000\ 0000\ 0000\ 0000)_2$　　$(8000)_{16}$

↓　　↓

$(0000\ 0000\ 0000\ 0001)_2$　　$(0001)_{16}$

5.3.4　1ビット下位回転シフト

$(b_{15}\, b_{14}$・・・$b_1\, \boldsymbol{b_0})_2$　　←第２オペランド

↓

$(\boldsymbol{b_0}\, b_{15}\, b_{14}$・・・$b_1)_2$　　←第１オペランド

ビット列が1ビット下位シフト（右シフト）し，桁落ちした元のLSBがシフト後のMSB

に入る．CC設定はない．

この操作は1ビット上位回転シフトの逆の数値データ操作に応用される．

問題 5.1　次の演算を手計算で行い結果を16進表記で示せ．

(1) 加算：$(1B3D)_{16} + (2CEA)_{16}$

(2) 加算：$(F5D2)_{16} + (EF1C)_{16}$

(3) 減算：$(671A)_{16} - (3B4F)_{16}$

(4) 減算：$(15C4)_{16} - (5EE7)_{16}$

(5) 16の補数変換：$(13EF)_{16}$

(6) 16の補数変換：$(8BF1)_{16}$

(7) AND演算：$(1B3D)_{16}$ [AND] $(2CEA)_{16}$

(8) OR演算：$(671A)_{16}$ [OR] $(3B4F)_{16}$

(9) XOR演算：$(671A)_{16}$ [XOR] $(3B4F)_{16}$

(10) NOT演算：[NOT] $(5EE7)_{16}$

(11) 1ビット上位シフト（CC設定は省略）：$(EF1C)_{16}$

(12) 1ビット下位シフト（CC設定は省略）：$(1B3D)_{16}$

(13) 1ビット上位回転シフト：$(8BF1)_{16}$

(14) 1ビット下位回転シフト：$(13EF)_{16}$

以下，第６章・第７章は後編に記載．

第１章著者：　津田 伸生 教授

第２～４章著者：　津田 伸生 教授

宮田 英男 名誉教授

鷹合 大輔 准教授

第５章著者：　津田 伸生 教授