マイクロコンピュータとアセンブリ言語

1. 実験目的

本実験ではマイクロコンピュータの基礎知識と動作原理の理解と，マイクロコンピュータを制御するソフトウェアをアセンブリ言語でプログラミングする能力の取得を目的とする．

具体的には次の５点．

（１） マイクロコンピュータの基礎知識（マイコンとは，２進数）
（２） マイクロコンピュータのハードウェア構成（ＣＰＵ，メモリ，Ｉ／Ｏ）
（３） コンピュータの動作原理（フェッチ＆エグゼキューション）
（４） アセンブリ言語でのプログラミング
（５） デバッグ手法（ステップ実行，ブレークポイント，メモリ・レジスタ参照）

2. 使用テキストと実験機材

本実験で使用するテキストは次の４点．

（１） この文書
（２） 練習課題シート（1 週目用）
（３） プログラム実習課題用紙（2 週目用）
（４） プログラム実習サンプルプログラムディスク

本実験で使用する機材は次の２点．

（５） ＯＡＫＳ１６ＬＣＤボードキット
（６） ＯＡＫＳ１６リモート制御用パーソナルコンピュータ

3. 実験の進め方

本実験は原則として学籍番号順の 2 人 1 班として行う．協力して作業すること．

１週目　練習課題シートの演習・実験課題を行い解答欄・実験結果欄に記入し，その場で提出
課題は班で協力してよいが，練習課題シートは 1 人 1 人が記入して別々に提出のこと
練習課題シートを提出すればその日は終了

２週目　1 週目の課題シート採点結果を受領
プログラム課題の実験
プログラムが 1 つ完成したら担当教員に報告　→　担当教員が動作確認
全部の課題が終了したら解散．レポート提出は後日でよい．

4. マイクロコンピュータの基礎知識

4.1 ハードウェアとソフトウェア

一般のコンピュータは，大きく図 １のような構成になっている．コンピュータは複雑な機械であり様々な捉え方があるが，「**情報処理装置**」としての側面を説明すると，図 ２のように価値ある「**情報**」を生み出すための機械だと捉えることができる．

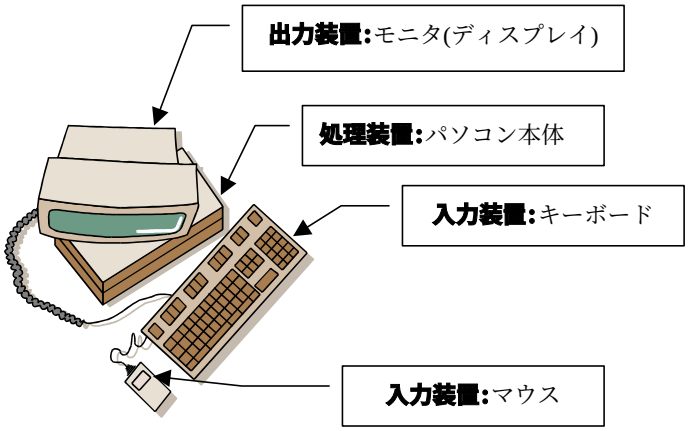


図 1　パーソナルコンピュータ

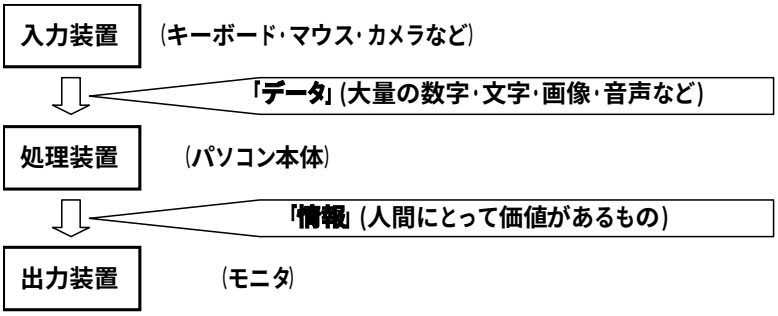


図 2　情報処理装置

どのようにデータを処理するかは，人間があらかじめ指示しておく．この処理内容を指示する記述の集まりを**「プログラム（program）」**と呼ぶ． 出力装置，処理装置，入力装置という機器類のことを**「ハードウェア（hardware）」**，プログラムのことを**「ソフトウェア（software）」**とも呼ぶ．コンピュータの最大の特徴は，プログラムを適切に用意できれば，様々なデータに対して処理を自動的に進められる点である．

4.2　マイクロコンピュータとは

「マイクロコンピュータ（Microcomputer）」とは，この処理装置の部分をマイクロサイズ（極小サイズ）で実現したもののことをいう．日本語では俗に**「マイコン」**と略されることが多い．特に，一つの IC（Integrated Circuit：集積回路）チップとして実現されたもののことを，**「ワンチップマイコン（英語では Single-Chip Microcomputer）」**と呼ぶ．

図 3に本実験で使用する OAKS16LCD ボードキットを示す．黒い小さなマイコンチップ（ルネサス製 M16C/62A）が処理装置本体にあたる．それに入力装置として８個のトグルスイッチと２個のタクトスイッチ，出力装置として８個の LED（発光ダイオード）と LCD（液晶パネル，本実験では使用しない）が付けられている．LED は「エルイーディ」と発音し小型のランプのように点灯する．トグルスイッチはレバーの上下によって二つの状態を切り替えるタイプのスイッチである．タクトスイッチは，回路基板上に直接取り付けられた小型の押しボタン式スイッチである．

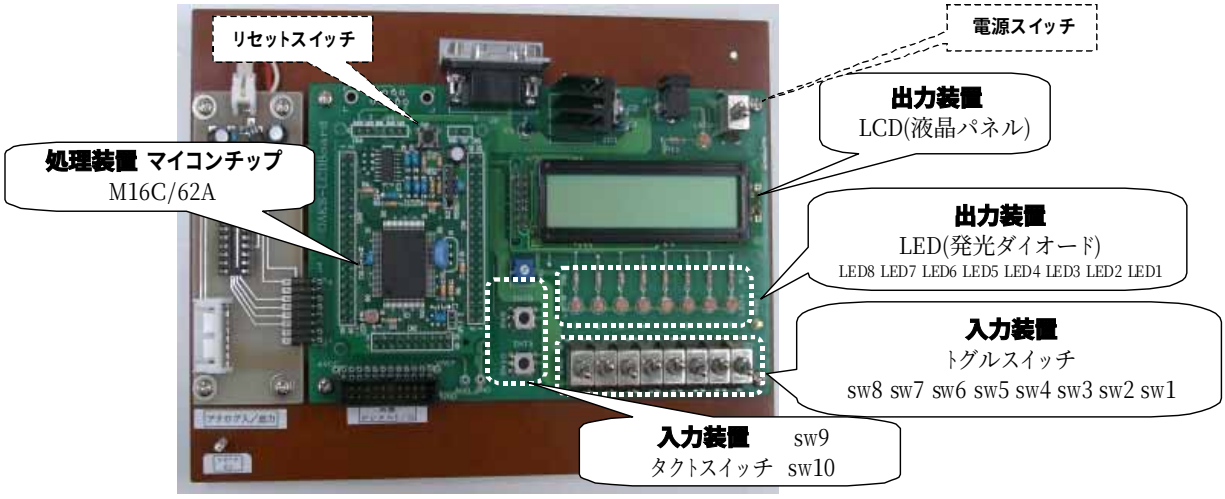


図 3　OAKS16LCD ボードキット

4.3　マイクロコンピュータ（マイクロコントローラ）の用途

マイコンチップは，そのサイズの小ささから，携帯電話，家電製品や自動車などに組み込まれて使用されることが多い．そこでのマイクロコンピュータは，情報処理装置としてだけではなく，組み込まれた先の機械（携帯電話，家電や自動車）の動作制御（コントロール）も目的としている．このため**「マイクロコントローラ（microcontroller）」**とも呼ばれる．これも略せば「マイコン」である．

このようなマイコンが組み込まれた機械（システム）のことを**「エンベッド（組み込み）システム（embedded system）」**と呼ぶ．

家電メーカや自動車メーカだけでなく広く製造業一般で，マイコンの扱いに習熟した人，例えば国家資格である情報処理技師術者試験の中で「テクニカルエンジニア試験（エンベッドシステム）」合格者などの人材を募集している．このような人材を組み込み技術者，エンベッドシステム技術者，あるいはマイコン技術者と呼ぶ．

4.4　マイクロコンピュータのプログラミング

マイクロコンピュータ（マイクロコントローラ）は，サイズが小さいだけで本質的には一般のコンピュータと何ら変わることはない．しかし，機器に組み込まれて使用されるために，様々なアプリケーションプログラムを実行する一般のコンピュータと比較して，特定の機器制御プログラム専用として動作するという特徴がある．

このため，パーソナルコンピュータ用の Windows などの多機能なオペレーティングシステムが搭載されることは少なく，例えば入力機器からデータを読み取るときでも，オペレーティングシステムが提供するライブラリなどを利用できないことが多い．マイクロコンピュータでは，入力機器などの周辺装置を直接制御するために**アセンブリ言語**でプログラムを書くことが必要になる．このため，マイクロコンピュータのプログラミングには，ハードウェアとアセンブリ言語の知識が要求される．

4.5　アセンブリ言語とは

ソフトウェアを記述するためには，例えば C 言語や Java 言語など様々なプログラミング言語が存在する．これらはコンピュータの動作をプログラムしやすいようにだけではなく，人間にとってもプ

ログラムを書きやすく，読みやすくするように工夫されたものである．

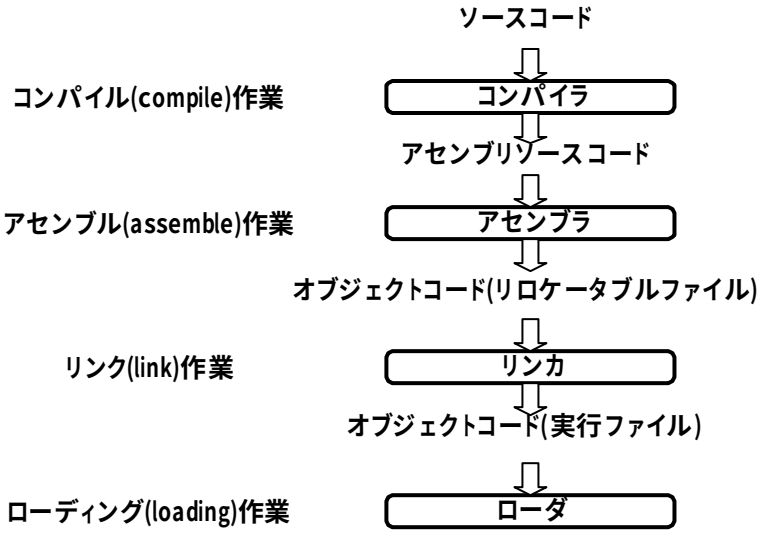
しかし，コンピュータが実際に解釈して実行できるのは「**機械語（machine language:マシン語）**」だけである．コンピュータはデジタル回路で作られているが，動作中のある瞬間ある瞬間で各部の回路を細かく制御して目的の動作を実行する．その制御を指示するものが「**機械命令（machine instruction）**」である．この機械命令はハードウェアに固有なもので，そのハードウェアが動作するのに必要な種類が存在する．機械命令のひとつひとつは特定の「**２進数**」として表現される（２進数については 5 章で説明）．目的に合わせて適切な種類の機械命令が適切な順序で並べられたものが機械語のプログラムである．これを「**バイナリコード（binary code）**」（または「**オブジェクトコード（object code）**」）と呼ぶ．C 言語などのプログラムは，一度この機械語のプログラムへ翻訳されてから実行されている．

機械命令は２進数であるため，人間が直接記述するのは不可能ではないが大変面倒である．そこで機械語で直接プログラムするために，機械命令と一対一で対応するような覚えやすい単語を用意し，それを機械語に「**アセンブル（assemble：変換）**」することが考えられた．覚えやすい単語のことを「**ニーモニック（mnemonic）**」と呼び，それを使って記述するプログラミング言語のことを「**アセンブリ言語**」と呼ぶ．機械語へのアセンブル作業を行うプログラムのことを「**アセンブラ（assembler）**」と呼ぶ．

このアセンブリ言語でのプログラミングを通じてコンピュータの動作原理を学習することができる．その目的にとって機能が単純なマイクロコンピュータは，最適な教材である．

コラム　言語処理系の基礎知識

高水準プログラミング言語 (C言語など) を機械語に翻訳して実行するためには，次のような段階がある．



リロケータブルファイルとは，メモリ上のどの位置にも格納可能なように，アドレスを決定していない状態のファイルで，複数のリロケータブルファイルをリンカがつなぎ合わせてアドレスを決定し，実行ファイルを生成する．それをローダが実際にメモリへ格納する．

見かけ上，コンパイラを呼び出してコンパイル作業を行うと，自動的に実行ファイルまで生成されるため，そこまでを含めて「コンパイル」すると称することもある．

5. 実験機材の使い方

5.1 OAKS16LCD ボードキットの準備

まず，図 ４の通りに （１）シリアル通信ケーブルでリモート制御用のパーソナルコンピュータと接続し，（２）ＡＣアダプタと接続する．

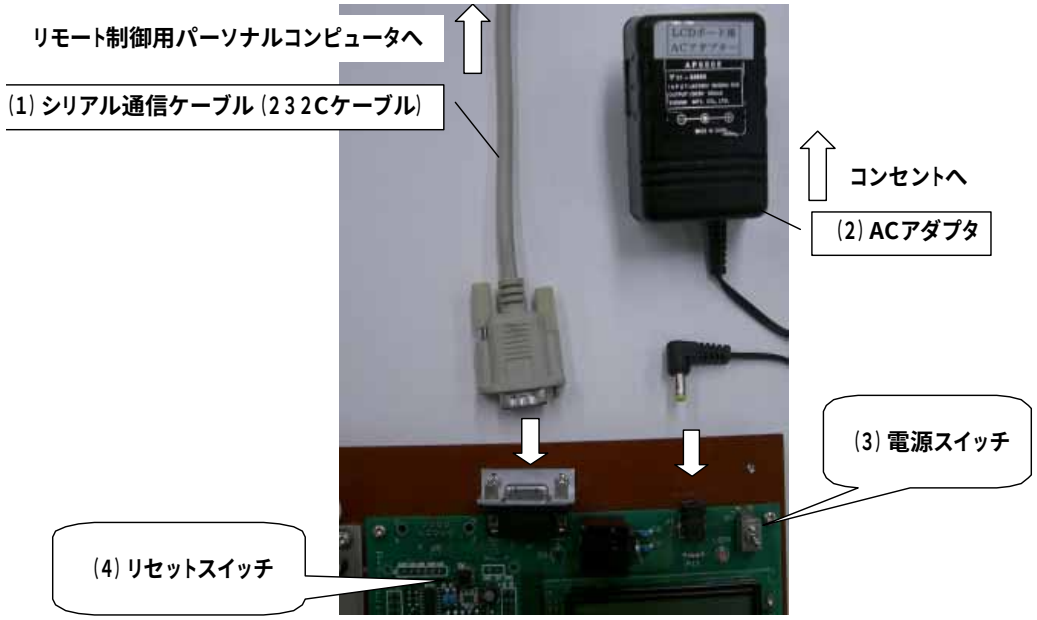


図 4　OAKS16LCD ボードキットの接続

次に（３）電源スイッチを入れ，必要に応じて（４）リセットスイッチを押す．

5.2 OAKS16LCD ボードキットのプログラム開発環境

一般のパーソナルコンピュータでは，そのコンピュータだけでプログラム開発ができる．つまりそのコンピュータの上で動作するプログラム環境として，プログラムを編集するためのエディタや，アセンブリ言語を機械語に変換するためのアセンブラ，プログラムの誤りを調べるためのデバッガなどがちゃんと揃っている．

しかし，マイクロコンピュータではそもそも高機能なオペレーティングシステムもないため，マイクロコンピュータ上で動くプログラミング環境が存在しないことが多い．本実験で使用する OAKS16 LCD ボードキットにも存在しないため，プログラミングのためには別のパーソナルコンピュータが必要となる．そのコンピュータ上で OAKS16 用のプログラムを作成してから，OAKS16LCD ボードキットへ送信し，動作を確認することになる．プログラムを開発する対象となるマイクロコンピュータ側を**「ターゲットマシン」**，プログラミング作業のためのパーソナルコンピュータのことを**「開発ホストマシン」**と呼び，このような開発のことを**「クロス開発」**と呼ぶ．

OAKS16LCD ボードのアセンブリ言語によるプログラム開発は，具体的には図 ５に従った手順で行う．開発ホストマシン上のデバッガは，ターゲットマシン上の**「モニタ」**プログラムと通信を行う．モニタプログラムはあらかじめターゲットマシンである OAKS16LCD ボードキット上に導入されているもので，開発ホストマシンのデバッガから指令を受けて，機械語プログラムを受け取って実行するためのプログラムである．また，機械語プログラム転送後にも，必要に応じてデバッガから指

令を送ることで，プログラムの実行を中断したり，途中経過を調べたり，実行を再開したりするなど，機械語プログラムの実行をリモートコントロールすることができる．これによりプログラムのデバッグ（プログラムミスの発見・修正）を行う．

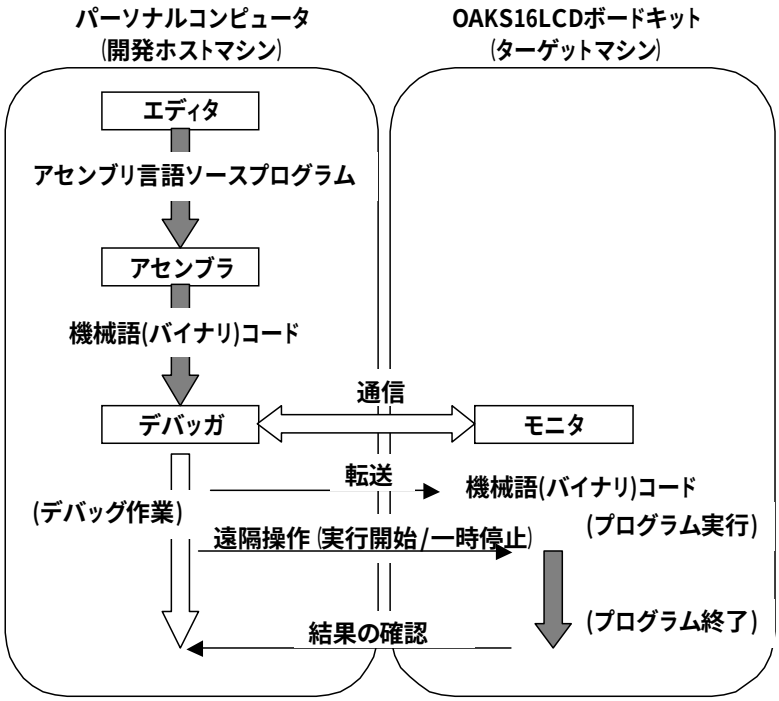


図 5　プログラム開発の流れ（クロス開発環境）

5.3　練習サンプルプログラムの起動と実行

自分でプログラミングする前に，サンプルプログラムを動作させてみよう．まず，4.1 節で説明した準備を済ませた後，OAKS16LCD ボードキットに接続されたパーソナルコンピュータを起動し，ログインする．本実験では，各自の MO ディスクや USB メモリを使用する．あらかじめ MO ディスクや USB メモリを開きその中に「microcomputer」など**アルファベット(日本語は不可)**の実験用フォルダを作成しておきくこと．

その後，スタートメニューから図 6 のように TM を起動する．

図 6　TM の起動

図 7 のように TM はアイコンバーだけのソフトで，ここで左側にある「プロジェクト新規作成」アイコンをクリックして新規プロジェクトを作成する（残りのアイコンは別途説明する）．開いた図 8 で，ターゲットチップをクリックし，プロジェクト名を半角英数文字だけを使って適当に指定する．このとき，ワーキングディレクトリとして「フォルダの参照」ボタンをクリックし，最初に作成しておいた自分の MO ディスクや USB メモリの中のフォルダを指定する．

最終的に図 8で「次へ（N）」ボタンをクリック.

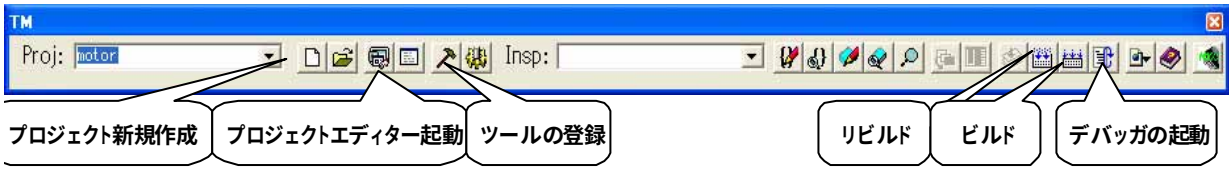


図 7　起動された TM

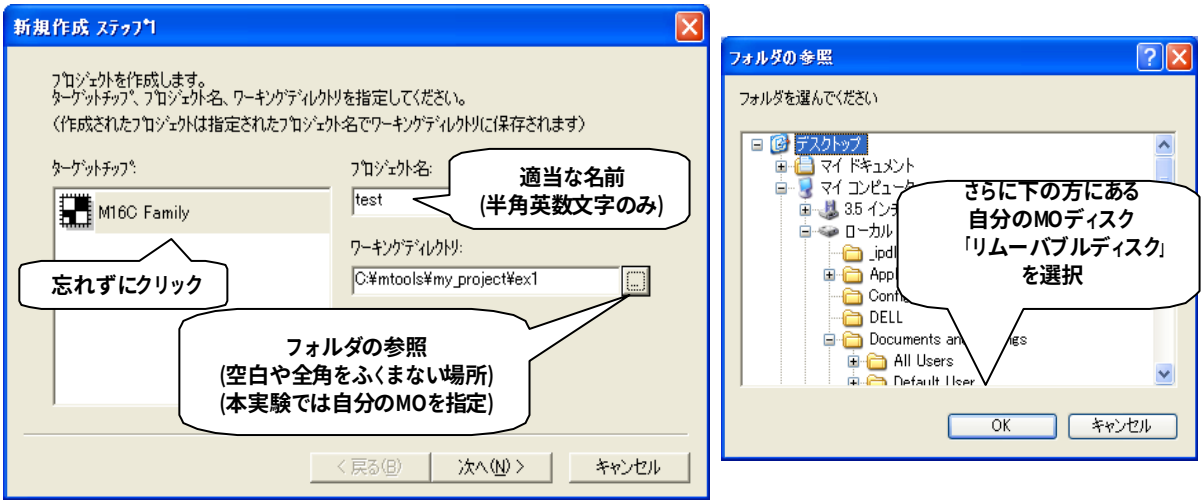


図 8　新規作成ステップ1とフォルダの参照

開いた図 9の左側の図で「アセンブラプロジェクト」を指定し「次へ（N）」．すると今度は図 9右側の図が開くのでそのまま「次へ（N）」．さらに開いた図 １０の左側の図もそのままで「完了」ボタンをクリックして作成作業を終了する.

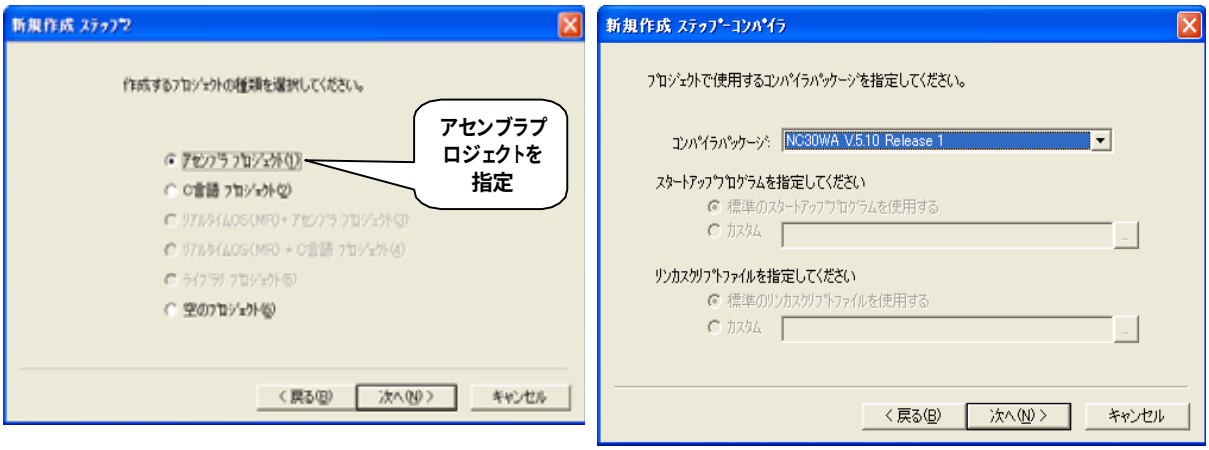


図 9　新規作成ステップ2と新規作成ステップ2－コンパイラ

ここで，作成したプロジェクトのフォルダ（図 8の段階で指定したもの）をマイコンピュータやエクスプローラで確認すると，図 １０の右側のように「（指定したプロジェクト名）.tmi」と「（指定したプロジェクト名）.tmk」という２つのファイルが生成されている.

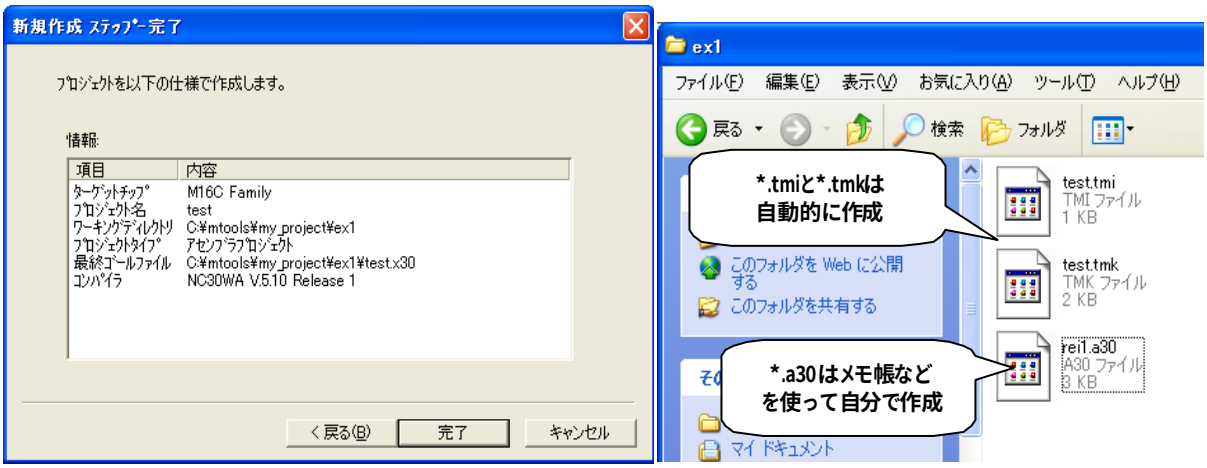


図 １０　新規作成ステップ一完了と作成したプロジェクトのフォルダ内容

このフォルダの中に，拡張子がアセンブリ言語のソースプログラムのファイルであることを表す「.a30」として「（適当な名前）.a30」というファイルを作成することになる．本実験では，フロッピィディスクに格納されているサンプルプログラム「rei1.a30」をこのフォルダにコピーすること．

図 １０が終わった段階で，プロジェクトエディター（図 １１）が開いている．もし開いていなかった場合には TM から起動し直す（図 ７）．この段階で開いているプロジェクトには，図 １０右側で作成した「.a30」アセンブリソースファイルがまだ登録されていない．

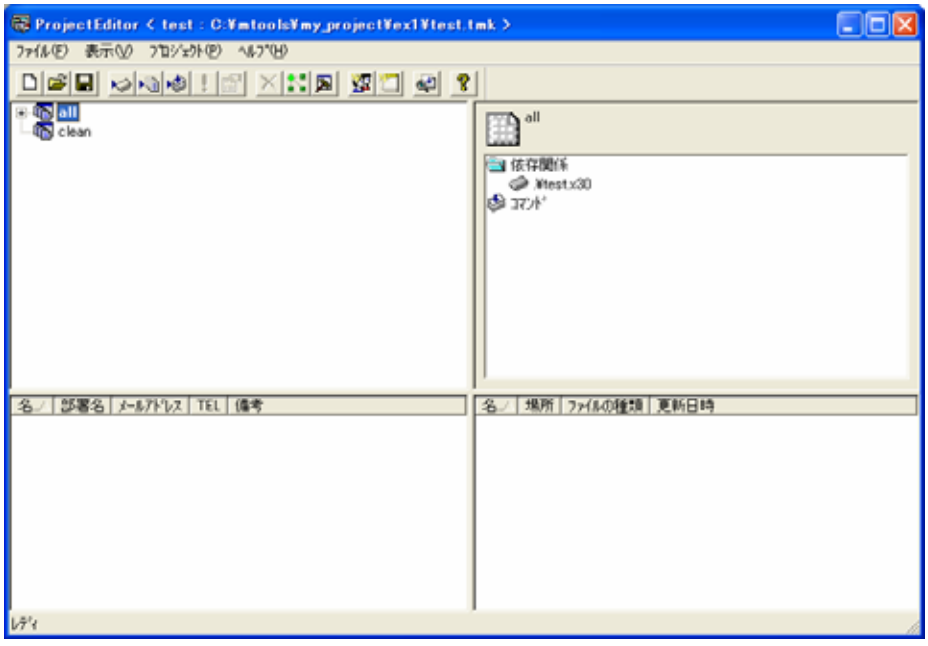


図 １１　プロジェクトエディター

図 １２のようにプロジェクタエディターに追加する．まず「all」の部分をダブルクリックして中身を開き，「（プロジェクト名）.x30」という部分で右クリックし（図 １２左），開いたショートカットメニューで「ファイルの追加（F）」をクリックする（図 １２右）．

後は追加するアセンブラソースファイルを指定する．「場所」のところが，図 ８で指定したワーキングディレクトリ（図 １０右でファイルを作成した場所）を指定することに注意．追加後，追加

した状態が図 １３右のように，**「.x30」の下の階層**に追加されていることを確認すること．例えば「all」のすぐ下の階層（.x30 と並んでいる位置）などように違う位置に追加してしまうと動作しない．

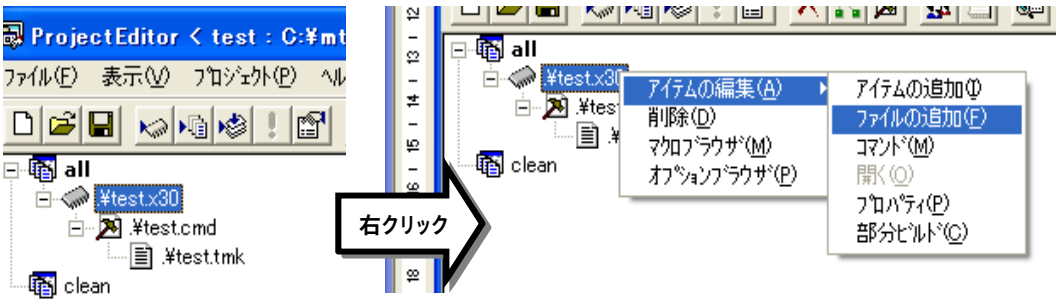


図 １２　アセンブリソースファイルの追加方法

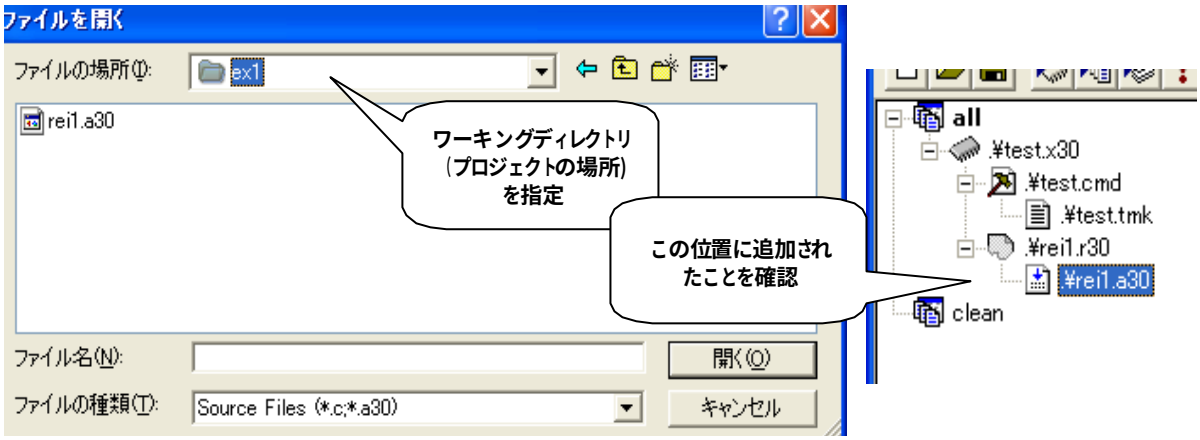


図 １３　ファイルの指定と追加されたファイルの確認

この図 １３右の「.a30」ファイルの部分をダブルクリックすると，メモ帳が開き，そのファイル（アセンブリ言語のソースプログラム）を編集することができる（図 １４）．

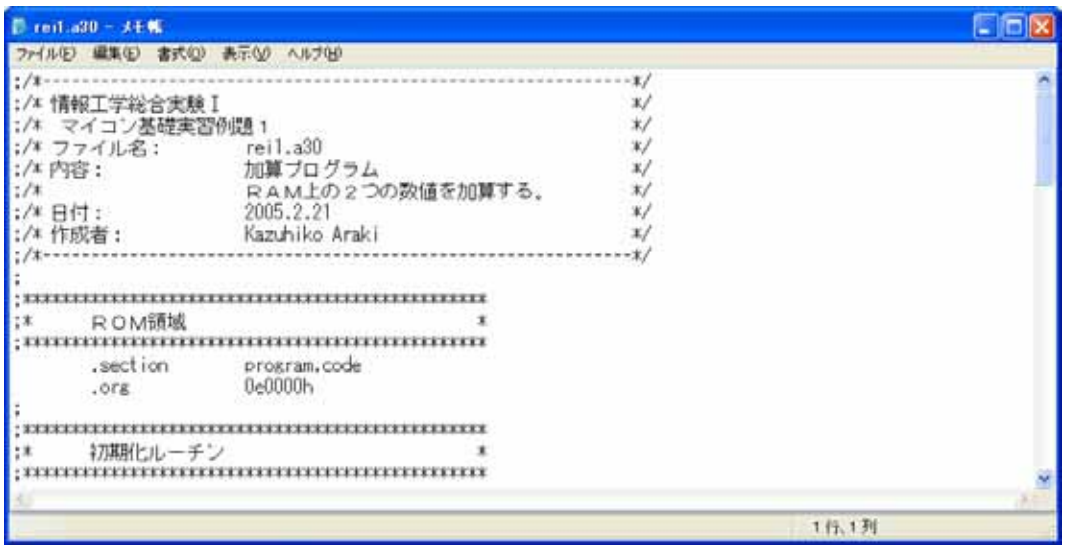


図 １４　アセンブリ言語ソースプログラムの編集

アセンブリ言語ソースプログラムをそのプログラムを実行するには，図 7の TM に戻って，「ビルド」または「リビルド」アイコンをクリックする．**「ビルド」**とは（複数の）プログラムソースファイルからアセンブル作業やコンパイル作業を行い，実行プログラムファイルを構築（ビルドアップ）することである．ビルドでは，前回ビルドした時点から変更されたファイルだけをアセンブル（コンパイル）し直して，その他は前回の結果をそのまま利用して合理的に実行ファイルを構築し直す．それに対して**「リビルド」**では，必ず全部のファイルをアセンブル（コンパイル）し直す．本実験のようにプログラムソースファイルが１つだけの場合には，両者に違いはない．

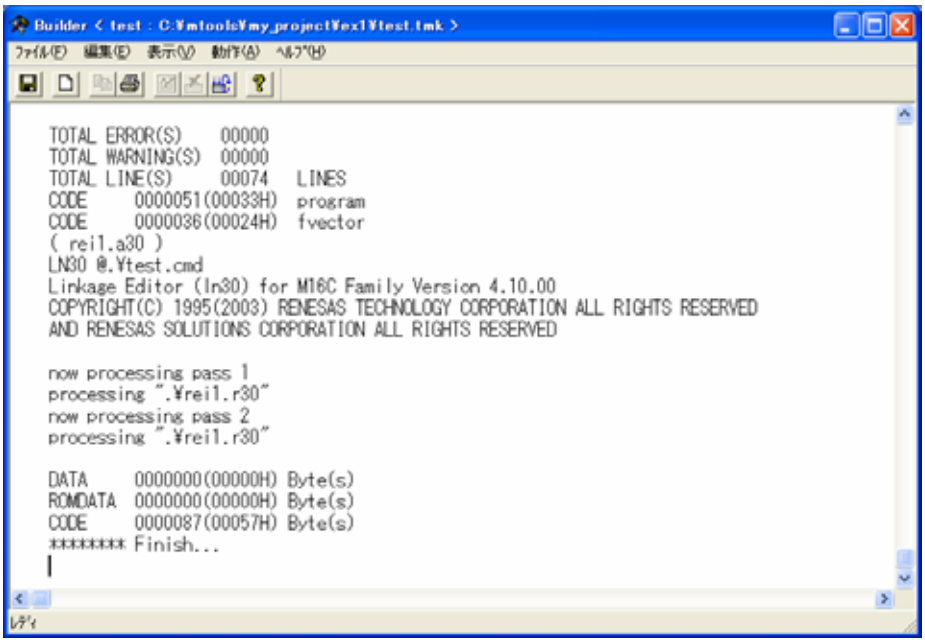


図 １５　ビルド中の表示画面

「ビルド」（「リビルド」）すると図 １５のようなビルド作業の結果が表示される．ここで注意する点は，画面をスクロールして全体を見渡し，途中に図 １６のようなエラー表示が出ていないことを確認することである．この段階で表示されるエラーは，プログラムに文法上の間違いがあることを示している．ちなみにこのエラー表示の行をダブルクリックすると，メモ帳などのエディタが自動的に開き，プログラムのエラーがある（と思われる）行にカーソルを移動してくれるので，その場で修正作業ができる．

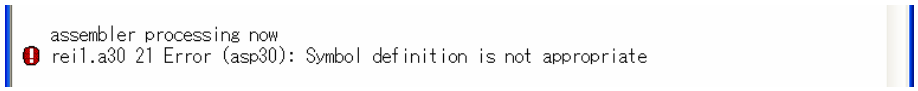


図 １６　ビルド中のエラー表示

１つでも文法エラーがある場合，実行ファイルを作成できないためプログラムをそもそも実行することができない．すべての文法エラーを修正し終えて，はじめて実行ファイルが生成される．わざわざ確認する必要はないが，プロジェクトのフォルダを再確認してみると図 １７のように拡張子「.x30」の実行ファイルが生成されている様子が見える．最初に存在した３つのファイル（図 １０右）以外にその他の中間ファイルなどが４つほど生成されている．

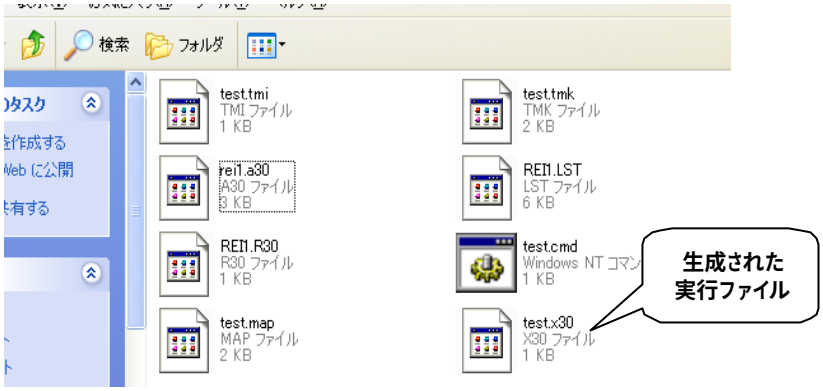


図 １７　生成された実行ファイル

プログラムを実行するには，やはり図 ７の TM 上で，「デバッガの起動」アイコンをクリックする．そのプロジェクトにおいて最初の起動時には，図 １８のようにデバッガを指定する画面が開く．

図 １８　デバッガの指定

KD30 というツールが，本実験で用いるデバッガの名前である．ここでは KD30 にチェックをいれて OK をクリックすればよい．２回目以降はこの画面は開かない．

すると図 １９左が開くので，普段はそのまま「OK」ボタンをクリックする．もしもここで図 １９右側の通信エラー画面が開いた場合には，再度 4.1 節の図 ４の接続を確認し，電源スイッチを入れなおし，確実にリセットボタンを押し直してから再度デバッガを起動する．

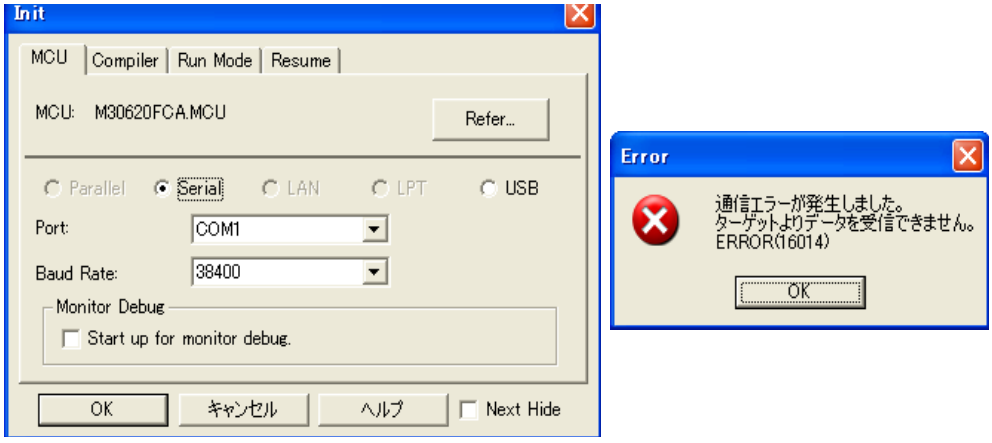


図 １９　デバッガ通信画面と通信エラー画面

無事にデバッガが起動すれば，図 ２０のような画面が表示される．これで左にある「Go」ボタンをクリックすれば，プログラムを実行することができる．

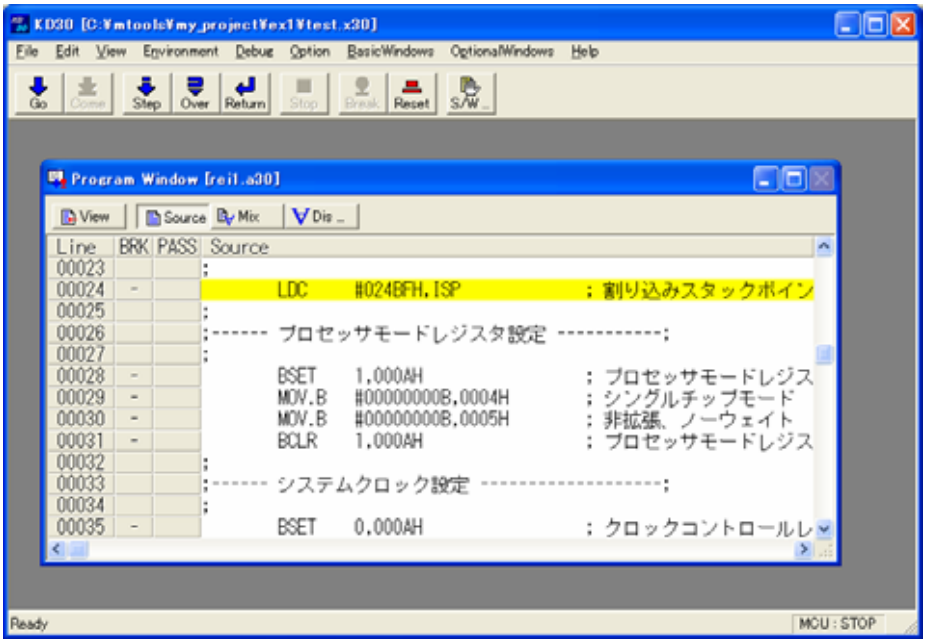


図 ２０　KD30 起動画面

6.　２進数の算術演算と論理演算

6.1　符号の表現

マイクロコンピュータだけではなく，現在すべてのコンピュータは電子回路によって作成されている．電子回路の中でも，電圧が「高い」か「低い」（電流を「流す」か「流さない」）かという電気的な２状態だけを利用して動作させるものを**「デジタル回路」**あるいは**「論理回路」**と呼ぶ．

デジタル回路は，複雑な内部状態を有する．すなわち，デジタル回路のある部分の電圧が「高い」か「低い」か，別の部分の電圧が「高い」か「低い」かのような，回路各部での電圧の組み合わせによって様々な種類の内部状態を持つ．それぞれの内部状態に人間が適当な意味を持たせることで，多数の「情報」を表現させることができる．

表 １　符号の表現

1	0
電圧が高い	電圧が低い
5V	0V
高	低
High	Low
H	L
ON	OFF
True	False
T	F
真	偽
はい	いいえ
Yes	No
Y	N

言い換えれば，デジタル回路ではすべての情報をこの**２つの電圧の組み合わせだけで表現**する．この「電圧」を記号に置き換えたものを**「符号」**と呼ぶ．デジタル回路（コンピュータ）では使える符号が２種類だけとなる．この２種類の符号は，時と場合により表 １のようにいろいろな言葉として表現される．呼び方は特に重要ではなく，「２種類のうちのどちら側か」が重要である．どの表現を耳にしても混乱しないようにして欲しい．本書では以後「1」と「0」という表現を用いる．

6.2 ２進数と情報量の単位

我々人間が普段利用している数字は，「0」から「9」までの 10 個の符号を組み合わせて数値データを表現している．このため「0」から順に数え上げると「9」の次には（使える符号を全部使い切ってしまったので）「10」のように２桁で表現する．これを「桁上がり」と呼び，桁上がりを最初に起こす「10」という数字により，この表現を**「10 進数」**と呼ぶ．これに対して符号が「0」と「1」しかない場合には，「0」「1」と数えると次はもう桁上がりがおこるため**「２進数」**と呼ぶ．コンピュータは内部で扱える符号が２種類しかないため，必然的に数値を２進数で扱わなければならない．

10 進数では２桁目が「10」「20」，３桁目が「100」「200」などを表すように，２進数では２桁目が「2」，３桁目が「4」，４桁目が「8」，n桁目が「2^n」を表す（図 ２１）．これを考えれば２進数と 10 進数の間に対応がとれ，相互に変換することができる．

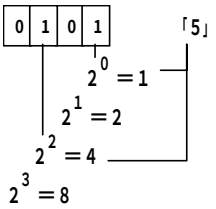


図 ２１　２進数と桁の重み

桁数に制限さえなければ，２進数でも 10 進数でも表現可能な数に差はない．しかし現実のコンピュータでは扱える桁数には制限がある．例えば，（ありえないほど少ないが）２桁だけだった場合，符号の組み合わせ方は「00」「01」「10」「11」の 4 通り（通常の 10 進数の「0」から「3」まで）しか表現できない．桁数によって２進数が表現できる整数を表 ２としてまとめる．

表 ２　２進数の桁数と表現可能な数値の数

桁数	組み合わせの数		
1	2^1	2	2
2	2^2	2×2	4
3	2^3	2×2×2	8
4	2^4	2×2×2×2	16
5	2^5	2×2×2×2×2	32
6	2^6	2×2×2×2×2×2	64
7	2^7	2×2×2×2×2×2×2	128
8	2^8	2×2×2×2×2×2×2×2	256
9	2^9	2×2×2×2×2×2×2×2×2	512
10	2^{10}	2×2×2×2×2×2×2×2×2×2	1024
16	2^{16}	省略	65536
24	2^{24}	省略	16777216
32	2^{32}	省略	4294967296
48	2^{48}	省略	281474976710656
64	2^{64}	省略	18446744073709551616

この組み合わせの数は，桁１つあたり２種類の符号の組み合わせ数として，2^2(=2×2)=4 として求めることができる．４桁の場合には，2^4(=2×2×2×2)=16 通りとなる．桁数によって表現可能な数が決まることからも分かるように，２進数の桁数は**「情報量」**をあらわす重要な量である．そこで桁１つ（「0」か「1」か）のことを**「ビット（bit）」**と呼んで情報量を表現するための単位とする．例えば 16 桁の２進数の持つ情報量を「16 ビット」と表現する．また現代のコンピュータでは慣例的に 8 ビットのことを**「1 バイト（byte）」**と表現する．例えば 16 ビットのことを２バイトと呼ぶ.

6.3　16 進数

ある程度大きい数を２進数で表現しようとするとビット数（桁数）がたくさん必要となる．例えば 8 ビットだけではわずか 0 から 255 までの数しか表現できない．16 ビットや 32 ビットで表現すると表記が大変になるため，2 進数の代わりに**「16 進数」**がよく使われる．

16 進数は，10 進数と同様の「0」から「9」までの符号のほかに「A」から「F」までの６個の符号を追加して表現される数である．10 進数と２進数と 16 進数の対応を表 ３に示す．このように 16 進数では同じ数を表現する２進数はもちろん，10 進数よりも桁数が少なくてすむ.

表 ３　10 進数と２進数と 16 進数の対応

10 進数	２進数	16 進数
00	0000	0
01	0001	1
02	0010	2
03	0011	3
04	0100	4
05	0101	5
06	0110	6
07	0111	7
08	1000	8
09	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

16 進数が２進数の代わりに使われるのは，４ビット分の２進数と１桁の 16 進数がちょうど一対一に対応するため簡単に相互に変換できるためである．例えば図 ２２のように，２進数を 4 ビット別にそれぞれを表 ３に従って置き換え，つなぎ直せば 16 進数に変換できる．逆に 16 進数を２進数に変換することも同様に 16 進数１桁を４ビットの２進数に置き換えるだけである.

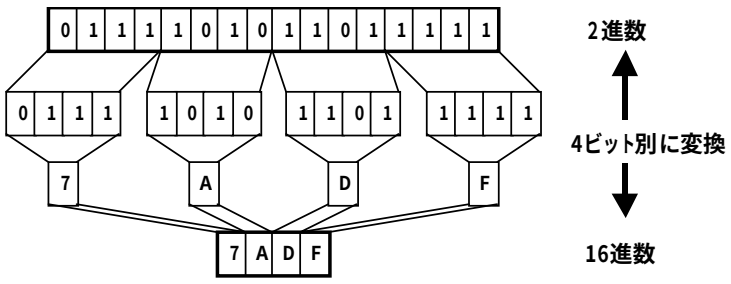


図 ２２　２進数と１６進数の変換

プログラミング言語によっては 10 進数だけでなく２進数や 16 進数で数を表記できるものがあるが，その場合 10 進数と区別するために特別な書き方をする必要がある．本実験で用いる M16C/62A 用の

アセンブリ言語では，16 進数表記する時には「**0**ABCD**h**」のように先頭に「0（ゼロ）」，最後に 16 進数（Hexadecimal Numeral）をあらわす「h」を付記する．２進数の時には「01010101**b**」のように２進数（Binary Numeral）を示す「b」を付記する．これらは大文字・小文字どちらでもよい．先頭にゼロを付けるのは，先頭がアルファベット文字だと英単語と区別できなくなるためであり，数値であることを明確にする意味がある．

6.4 論理演算

２進数の計算では，「AND」「OR」「NOT」「XOR」などの論理演算子を用いた**「論理演算」**がある（表 ４）．論理演算とは，「1」と「0」に対して演算を行い，その結果もやはり「1」と「0」のいずれかを得る演算である．

表 ４　基本論理演算

入力 A	入力 B	論理積 AND A・B	論理和 OR A+B	排他的論理和 XOR (EXOR,EOR) A⊕B	否定 NOT $\bar{A}$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

複数の「1」と「0」，例えば８ビットの２進数に対しては８個のビットの集まり（**ビットパターン**）と見て，それぞれのビットごとに論理演算すればよい(図 ２３).

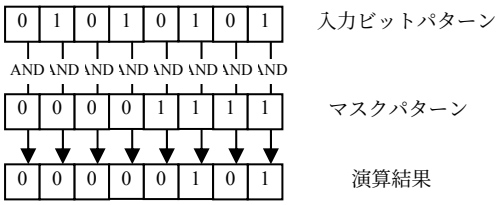


図 ２３　ビットパターンの演算

あるビットパターンを目的に応じて修正したいとき，その目的に応じて別のビットパターンを用意して論理演算することを**「ビットマスク演算（操作）」**と呼ぶ．目的に応じた別のビットパターンのことを**「マスクパターン」**あるいは単に**「マスク」**と呼ぶ．顔につけるマスクが顔の一部を覆い隠すように，必要のない部分を隠してしまうような操作を表している．

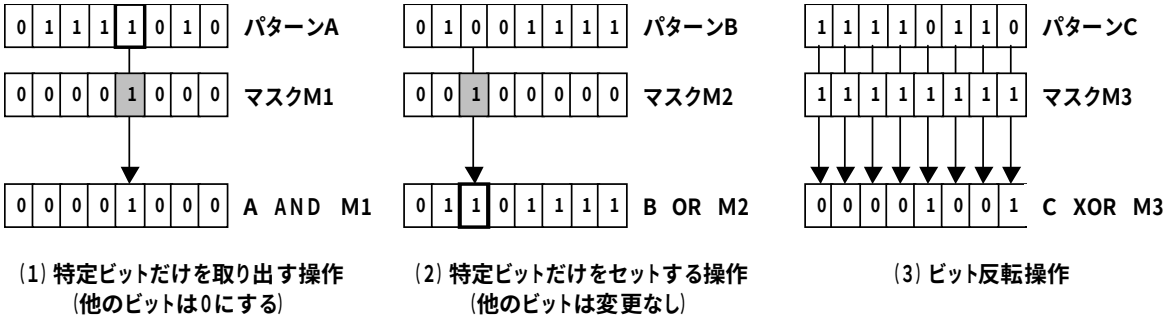


図 ２４　ビットマスク演算（ビットマスク操作）

代表的なビットマスク演算を図 ２４に示す．（１）あるビットパターンから特定ビットだけ取り出し，残りのビットを全部 0 にしてしまうためには，取り出したいビットだけを「1」にしたマスクと AND を取ればよい．（２）特定のビットだけを「1」にしたい（それ以外はそのままにしておきたい）場合には，セットしたいビットだけを 1 にしたマスクと OR を取ればよい．（３）全部のビットを反転したい（ 各ビットを NOT 演算する）場合、全ビットが 1 のマスクと XOR を取ればよい．

6.5 算術演算

加算・減算・乗算・除算など通常の算術演算では，論理演算の時のようにビットパターンではなく，２進数が扱われる．２進数（16 進数）でも通常の 10 進数と同じように行うことができる．例えば 10 進数で「5+4」を計算しても，２進数で「0101b+0100b」を計算しても結果は「9」と「1001b」となり，どちらでも同じ値になる．

ここまでは正の整数の場合であったが，負の数を表現するために２進数では**「２の補数」**を用いる．「２の補数」とは，先頭に正の数なら「0」負の数なら「1」とする**「符号ビット」**を付け足して，２進数の各桁をビット反転し，最後に 1 を加えたものである．図 ２５に例を示す．

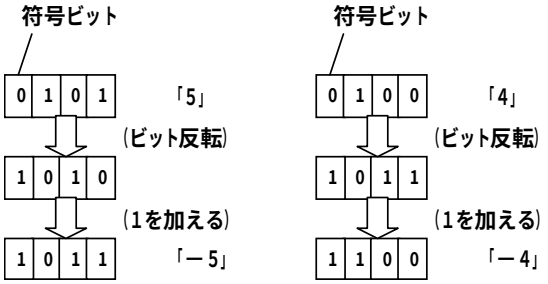


図 ２５ ２の補数の求め方

6.6 シフト演算

論理演算・算術演算の他に，一般には「シフト演算」がある．シフト演算には具体的に，「論理シフト演算」と「算術シフト演算」と「ローテート（回転）演算」が存在する．これらはさらに「右シフト（ローテート）」と「左シフト（ローテート）」操作から成り立っている．

「論理シフト」とは，ビットパターンを右・左の桁にずらす演算のことである．例えば１ビットだけ左右に論理シフトさせた動作を図 ２６に示す． 左シフトの場合には，一番左のビットが追い出されて消滅し，一番右のビットには新たに「0」を追加される．右シフトの場合も左右が逆なだけで同様な動作である．

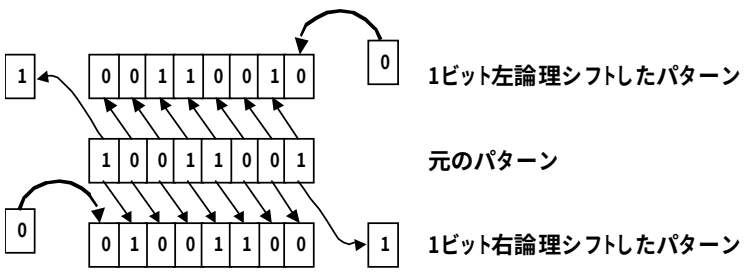


図 ２６ 論理シフト操作

「ｎビット右論理シフト」のように複数のビットで表現される場合には，１ビットのシフト動作をｎ回繰り返すことを意味する．図のように８ビットのパターンの場合には，左右に関わらずどちらでも８ビット論理シフトすれば全ビットが０になってしまうため，それ以上のシフトは無意味である．

「算術シフト」とは，左シフトの場合には論理シフトと同様であるものの，右シフト動作の際には符号ビットによって挙動が変化する．右シフト時に追加されるものが元のパターンで一番左のビット（符号ビット）と同じものとされる．つまり図 ２７のように「１」なら「１」が追加される．

論理シフトの場合には，対象とするものがあくまでビットパターンである．一方の算術シフトの場合には，符号ビットを含む２進数という数値データが対象である．左方向へのシフトは問題なくても，右シフトの場合には，２進数をそのまま論理シフトしてしまうと，「プラス」か「マイナス」かという符号を変化させてしまうことがある．算術シフトを使えば符号は変化しない．

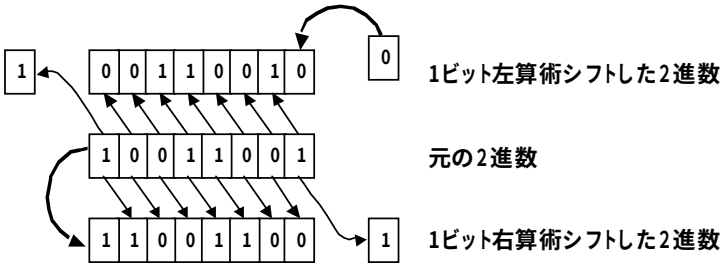


図 ２７　算術シフト操作

２進数は先の 5.2 節の図 ２１に示したように，ｎ桁目が2^nの重みを持っている．このため，１ビット左算術シフトをすることはそれぞれのビットのｎを１つ分ずらすことになり，結果的に元の数のちょうど２倍の数にすることになる．逆に右算術シフトすれば 1/2 にすることと等しい．

最後の**「ローテート」**も左右両方存在する．これは「シフト」では追い出されたビットが消滅してしまったが，それを反対側のビットとして追加する操作である（図 ２８）．

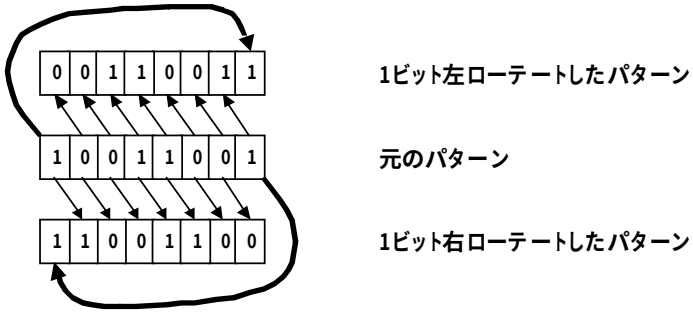


図 ２８　ローテート操作

6.7　BCD 演算

２進数の演算の中で，特殊なものが**「BCD（Binary Coded Decimal）」**を使った演算である．BCD は，２進数を４ビットごとに区切り（16 進数と同様），各４ビットで 10 進数の１桁分を表現する表記方法である．16 進数では「A」から「F」までの符号に対応する２進数「1010」から「1111」までを使用せず，「9」を示す「1001」からいきなり「0001 0000」のように 4 ビット分の繰り上がりを行う．人間が 10 進数を筆算するのと同じように，本来２進数しか扱えないコンピュータに（擬似的

に）10 進数のルールに従うように演算させることができる．

例えば 8 ビットの 2 進数から 3 桁の BCD コードへ変換するには，2 進数をまず 3 桁の 10 進数である 100 で割り，その答えを BCD の 3 桁目とし，余りを今度は 10 で割り，その答えを 2 桁目，最後の余りが 1 桁目として扱えばよい．

このような演算方法は，電卓のように人間同様の計算をすべき機器に組み込まれるマイクロコンピュータには必須の演算である．マイクロコンピュータの中には，このような演算方法に対応したハードウェアを持つものも存在するが，そうでない場合には BCD 表現に従うような計算プログラムを人間が用意してやる必要がある．

7. マイクロコンピュータのハードウェア構成

7.1 基本構成

アセンブリ言語でプログラミングを行うためには，対象となるマイクロコンピュータのハードウェアに関する知識が必要である．ここでハードウェアの基礎知識について説明する．まず，マイクロコンピュータに限らず，一般のコンピュータでもその基本構成は図 ２９ように模式化できる．

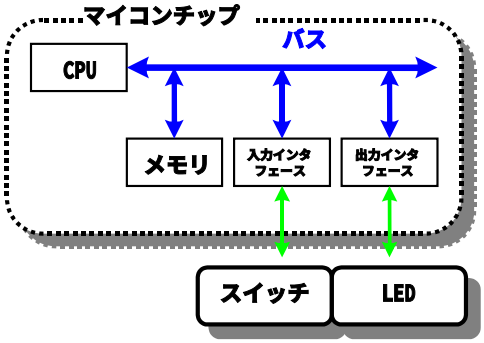


図 ２９　マイクロコンピュータの基本構成

「CPU（Central Processing Unit：中央処理装置）」とは，実際の計算処理の行う演算回路などを含むコンピュータの頭脳部分であり，また，メモリや入出力インタフェースすべてを制御するための制御回路などを含むコンピュータの中枢部分でもある．**「メモリ（Memory）」**とは記憶装置のことで，機械語プログラムやデータなどが格納されている．入力インタフェースと出力インタフェースはまとめて**「入出力インタフェース（Input/Output Interface）」**と呼ばれる．略して**「I/O」**（アイ・オー）と呼ばれる．

これらの各要素を用いて，コンピュータはプログラムされた通りの処理を実行する．コンピュータは CPU がメモリへ要求を出して，次に実行すべき機械語のプログラムを呼び出して実行を始める．計算すべき最初のデータは入力インタフェースを通じて入力機器から受け取ったり，最初からメモリに書かれていたりする．そうしたデータを使って計算を始め，途中の計算の中間結果などをその都度メモリに書き込んで記憶しておき，元のデータやそうした中間結果の値を組み合わせながら演算をプログラムどおりに進める．最終的に出力インタフェースを通じて結果を出力する．

説明の都合上，メモリ，入出力インタフェース，CPU の順で詳細を説明し，その後でコンピュータの動作の詳細を説明する．

7.2　メモリ

7.2.1　メモリの基本構造

メモリの基本構造は極めて単純で，図 ３０のように８ビット分ずつデータを格納するための場所が一列に並び，それぞれを区別するために順番に**「番地（アドレス）」**と呼ぶ番号が割り当てられている．メモリは 0 番地から始まり，そのハードウェアの CPU で扱える最大番地まで続く．最大桁数が多くなるため番地は 16 進数で表記されることが多い．

ただし，実際に最大量までメモリが搭載されることは，コストの面からほとんどないため，そのハードウェアに実際に搭載されたメモリ量を仕様書などで確認する必要がある．また，ハードウェアによっては番地の途中が抜けていることもあるので注意が必要である．本実験で使用する OAKS16LCD ボードについては次で説明する．

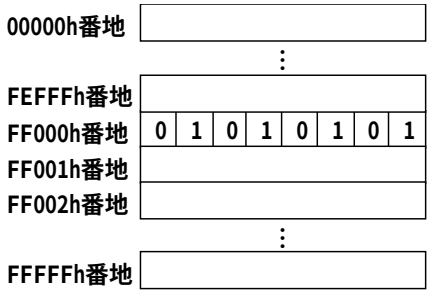


図 ３０　メモリの基本構造

メモリに格納されるデータはすべて２進数のデータである．ただし，3.4 で述べたように機械命令は２進数で表現されており，**機械語のプログラムはそのままメモリに格納できる**．

7.2.2　メモリの種類とメモリマップ

メモリの種類には大きく，**「RAM（Random Access Memory）」**と**「ROM（Read Only Memory）」**がある．RAM は読み書き自由であるものの，電源を切ってしまうと中身のデータは消えてしまう．それに対して ROM は読み出し専用メモリで中身のデータを原則として書き換えることができない．しかし，電源を切っても中身のデータを保持している．そこで組み込み機器の場合には，完成した制御用の機械語のプログラムを ROM に書き込んで（「ROM に焼いて」と表現する）出荷される．そのプログラム実行中に必要な作業領域として RAM を利用する．

ROM にもいくつか種類があり，通常は書き込むことができないものの，電圧を上げたり，特定のスイッチを入れたりすることで電気的に書き込むことができるものもある．OAKS16 では，デバッガなどを通じてユーザのプログラムを ROM に転送して書き込むことができるようになっている．

図 ３１に OAKS16 のデバッグ中のメモリマップを示す．これはメモリの ROM・RAM の区別などを示したものである．例えば，ユーザが自分のプログラムで自由に読み書きに使える RAM の領域は 400h 番地から 24BFh 番地までの「ユーザ RAM」領域である．ユーザのプログラム自体は，E0000h 番地から FBDFFh 番地までの「ユーザ ROM」へ転送されて書き込まれることになる．

先頭の SFR については次節の I/O インタフェースの所で説明する．また図中に「モニタ」とあるのは，4.2 節で説明したように開発ホストマシンと通信を行ってユーザのプログラムが転送されてきた時にそれを受け取ったり，デバッグ作業で開発ホストマシンからリモートコントロールをされるときの指示を受け取ったりするための部分である．このモニタは OAKS16LCD ボードでは最初から RO

M に書き込まれた状態となっている（はずである）．モニタ自体の説明はこれ以上詳しく行わないが，**ユーザがモニタのための領域を使用することはできない**点には注意しておくこと．最後のベクタテーブルは，本実験では説明しないが割り込み処理のための部分であり，一部分をモニタも使用している．実験ではここを書き換えないこと．

ちなみに，ユーザプログラムが完成した後は，モニタプログラムを ROM から削除する．

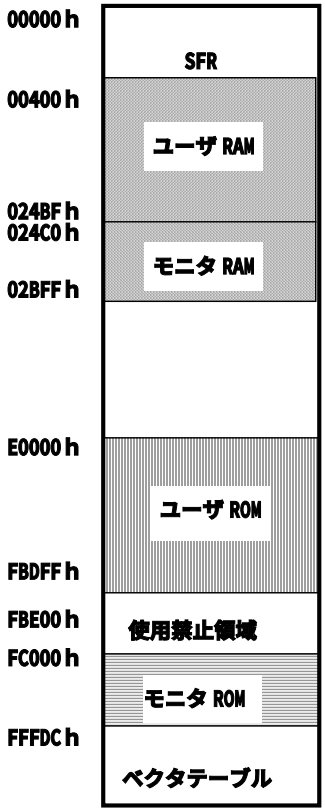


図 ３１　OAKS16 のデバッグ中のメモリマップ

7.3　I/O インタフェース

7.3.1　I/O ポートとは

「I/O」とは「Input/Output」すなわち「入出力」のことであり，I/O インタフェースのことを日本語では「入出力インタフェース」とも呼ぶ．コンピュータでは，入出力装置はすべて I/O インタフェース回路を通して CPU と接続される．この回路は，入力機器からの信号を CPU が読み込める形に変換したり，その逆に出力機器に合わせた出力をしたり，複数接続された機器の中から適切なものを選択したりするためのものである．

このインタフェースを媒介にして，コンピュータに接続されている入出力機器の「接続口」のことを**「I/O ポート」**と呼ぶ．入出力機器から見た場合には，コンセントや端子などの物理的な（外見上の）接続口のことを指す．逆に CPU 側から見た場合には，CPU から見た電子回路レベルでの接続口を意味する．これ以降アセンブリ言語など CPU 内部の話に限定するので，「I/O ポート」と表記した場合 CPU から見た I/O ポートのことを指すものとする．

I/O ポートには，原則として入出力装置 1 個に対して 1 個のポートを必要とするため複数の I/O ポートが用意される．それぞれを区別するため，I/O ポートにはメモリと同様にアドレス（番号）が

振られており，例えば１番にたまたま割り当てられたポートのことを，「１番ポート」「ポート１番」「ポート１」などと呼ぶ．この番号を**「ポートアドレス（ポート番号）」**という．

どの番号のポートにどの入出力装置を割り当てるかは，ハードウェア設計者が適切に（無駄な回路が生じないようになど）決めることで，何か規則があるわけではない．そのため，ある特定のコンピュータシステムで I/O ポートを直接アクセスして入出力装置を利用する場合には，そのシステムの仕様書から I/O ポートの割り当てを調べておく必要がある．

OAKS16LCD ボードでは表 ５のように割り当てられている．スイッチや LED の様に ON か OFF かの 1 ビット分しか状態のない単純な入出力装置の場合には，わざわざ 1 個に対して 8 ビット分のデータを扱えるポートを１つ用意する必要もないため，複数のスイッチをまとめて１つのポートに割り当てている．スイッチ 1 つに 1 ビット分で，合わせて 8 個のスイッチに対応するポート（8 ビット）を用意してある．例えば LED5 は，7 番ポートのビット 4 に対応させてある（表 ５）．

表 5　OAKS16LCD ボードの入出力ポート

ポート番号	ビット番号	入出力	装置
0	0	出力	CPU ボード上の LED
3	0	入力	トグルスイッチ（sw1）
3	1	入力	トグルスイッチ（sw2）
3	2	入力	トグルスイッチ（sw3）
3	3	入力	トグルスイッチ（sw4）
3	4	入力	トグルスイッチ（sw5）
3	5	入力	トグルスイッチ（sw6）
3	6	入力	トグルスイッチ（sw7）
3	7	入力	トグルスイッチ（sw8）
7	0	出力	LDE1
7	1	出力	LED2
7	2	出力	LED3
7	3	出力	LED4
7	4	出力	LED5
7	5	出力	LED6
7	6	出力	LED7
7	7	出力	LED8
8	2	入力	タクトスイッチ（sw9）
8	3	入力	タクトスイッチ（sw10）

注: sw や LED が 1 番から始まるのに対して，ポート番号やビット番号は 0 番から始まる点に注意すること

7.3.2　I/O ポートへのアクセス方法

ソフトウェアから出力装置へ何かデータを出力する場合（逆に入力装置からのデータを入力する場合も），I/O ポートアドレスへアクセスすることで行う．ここでいうアクセスとは，目的の I/O ポートアドレスからデータを読み出したり（その I/O ポートに接続されている入力装置からデータをもらうこと，「入力」），書き込んだり（その I/O ポートに接続されている出力装置にデータを渡すこと，「出力」）することを意味する．多くの CPU では，任意の I/O ポートからデータを読み込む機械命令，逆にデータを書き込む機械命令が用意されている．

同じようにアドレス（番地）で管理されてはいるが，I/O ポートアドレスとメモリアドレスは独立した性質のものである．つまり，I/O ポートの FFh 番とメモリの FFh 番地は，それぞれ別個の存在として扱われる．そこで，両者を明確に区別するために，I/O ポートの 1 番から始まる一連のアドレスのことを**「I/O 空間」**，メモリの 1 番地から始まる一連のアドレスのことを**「メモリ空間」**と呼ぶ．I/O 空間へアクセスする（任意の I/O ポートを読み書きする）ための機械命令と，メモリ空間へアクセスする（任意のメモリ番地を読み書きする）ための機械命令は別に用意されることが原則である．

ここまでは，一般論としての話である．本実験で用いる M16C/62A では I/O 空間へアクセスするた

めの機械命令が，実は用意されていない．M16C/62A ではメモリ空間の一部に I/O 空間を埋め込んでしまい，メモリと区別なく I/O へアクセスできるような仕組みになっている．これを**「メモリマップド I/O（memory-mapped I/O）」**と呼ぶ．（ちなみに，接続しなければならない入出力機器の数はそれほど多くないため，メモリ空間の方が I/O 空間よりも一般的に大きくなる．そのため，I/O 空間の方へメモリ空間を埋め込むことは通常ありえない．）

例えば，ポート 0 番をメモリの 03E0h 番地に割り付けて（「マップ」して）おく．すると，普通のメモリの読み書きと同様に 03E0h 番地へアクセスすると，ポート 0 番に接続されている入出力機器へのアクセスと見なされることになる．

M16C/62A では，メモリ空間の先頭の方に各 I/O ポートを埋め込んである．これが，6.2.2 節の図３１の中で**「SFR（Special Function Register：特殊機能レジスタ）」**として示されている部分である．**「レジスタ」**とは一時的な記憶領域を指す言葉で，この場合には I/O ポートと直結する記憶領域としての意味を持つ．SFR 領域は，もう一度 図 ３１を見れば分かる通り，0000h 番地から 03FF 番地が予約されている．

具体的な SFR 領域の内部は表 ６のようになっている．表 ６の左列に示した各アドレスへ通常のメモリに対してと同様にデータを読み書きするだけで，対応する表 ５の I/O ポートを通じて入出力装置からデータを受け取ったり，出力したりすることができる．ポートひとつあたり 8 ビット単位のデータを扱うことも，メモリ同様である．

表 ６　SFR の一覧表（一部）

メモリ空間上のアドレス	機能(対応する I/O ポート)	略称	接続装置
0004h	プロセッサモードレジスタ 0	PM0	
0005h	プロセッサモードレジスタ 1	PM1	
0006h	システムクロック制御レジスタ 0	CM0	
0007h	システムクロック制御レジスタ 1	CM1	
000Ah	プロテクトレジスタ	PRCR	
03E0h	ポート 0	P0	CPU ボード上の LED
03E1h	ポート 1	P1	
03E2h	ポート 0 方向レジスタ	PD0	
03E3h	ポート 1 方向レジスタ	PD1	
03E4h	ポート 2	P2	
03E5h	ポート 3	P3	トグルスイッチ
03E6h	ポート 2 方向レジスタ	PD2	
03E7h	ポート 3 方向レジスタ	PD3	
03E8h	ポート 4	P4	
03E9h	ポート 5	P5	
03EAh	ポート 4 方向レジスタ	PD4	
03EBh	ポート 5 方向レジスタ	PD5	
03ECh	ポート 6	P6	
03EDh	ポート 7	P7	LED
03EEh	ポート 6 方向レジスタ	PD6	
03EFh	ポート 7 方向レジスタ	PD7	
03F0h	ポート 8	P8	タクトスイッチ
03F1h	ポート 9	P9	
03F2h	ポート 8 方向レジスタ	PD8	
03F3h	ポート 9 方向レジスタ	PD9	
03F4h	ポート 10	P10	
03F5h	-	-	
03F6h	ポート 10 方向レジスタ	PD10	

ただし，表 ６の先頭の方にあるプロセッサモードレジスタや，システムクロック制御レジスタなどは，CPU の動作状態を制御するためのレジスタで，入出力機器とは接続されていない．このレジスタに特定の値を書き込むことで，例えば CPU の動作速度などを指定することができる．M16C/62A

では，プログラムの最初でこれらを初期化する必要がある．具体的な設定例は 8.1 節で説明する．

また，各ポートに対応して**「方向レジスタ」**が存在する．一般の CPU では出力専用のポート，入力専用のポートがそれぞれ別々に用意されることも多いが，M16C/62A では各 I/O ポートが必要に応じて出力と入力を切り替えられるようになっており，それを切り替えるのが方向レジスタである．

少々複雑な話となったので，具体的な例を図 ３２として説明しておく．実際の OAKS16LCD ボードを使ったプログラムで，例えば右から 3 番目のトグルスイッチ状態を確認するためには，まず表５を参照する．すると「sw3」が「ポート 3 番」の「ビット 2 番」に割り当てられていることが分かる（表 ５の上から 4 行目の項目）．そこで，ポート 3 番がメモリ空間上ではどのアドレスにマップされているかを表 ６を参照する．するとそのアドレスは「03E5h 番地」であることが分かる（表 ６の上から 11 行目）．これにより，「03E5h 番地」の「ビット 2」を読み込めば，スイッチ 3 の状態を確認できることが分かる．ただし，ポート 3 番の方向が，ポート 3 番用の方向レジスタで，正しく「入力」に設定されていることが条件である．表 ６ではポート 3 番用の方向レジスタも確認でき，「03E7h 番地」である（上から 13 行目）．「03E7h 番地」の「ビット 2」が「0」ならば「入力」を表し，スイッチからの情報が入力されていることを示す．

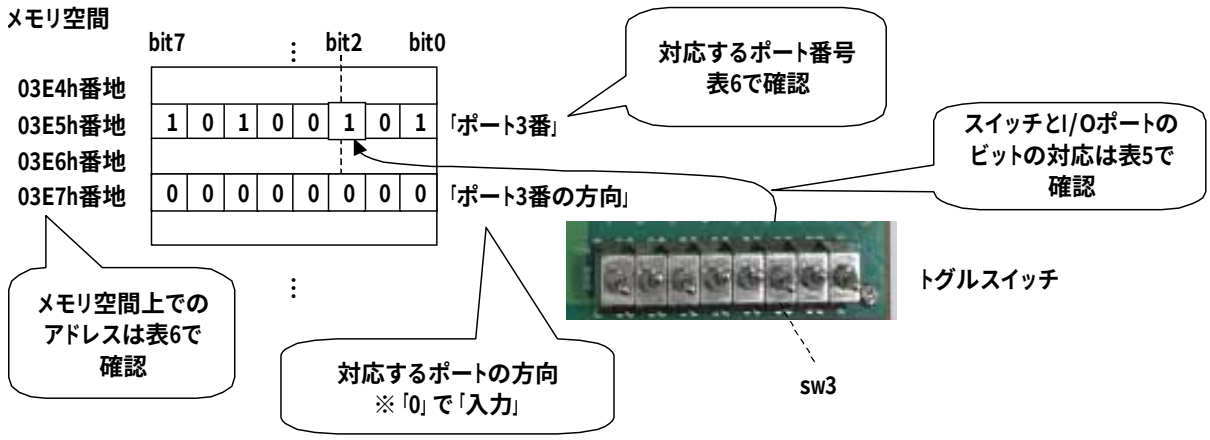


図 ３２ メモリマップド I/O の実際

パソコン用のプログラムの場合，キーボードなどの入出力装置へアクセスする時は Windows など OS が提供するライブラリを利用する．実はこうした OS が提供するライブラリの中で I/O ポートへのアクセスが行われており，一般のプログラマはあまり I/O ポートを意識することがない．

7.4 CPU

7.4.1 CPU の基本ハードウェア構造

図 ３３に CPU が演算を行うための回路の概念図を示す．この図ではメモリへアクセスする部分が大きく省略されているが，基本演算の行う部分の説明になっている．

CPU はメモリ中に格納されている機械命令をひとつずつ読み出して，いったん命令レジスタ（IR：Instruction Register）に格納する．その命令のビットパターン（ビットの 0 と 1 の並び方のパターン）を命令デコーダが解釈し，命令内容に即した演算回路を使用する．例えば，M16C/62A では加算命令は先頭が「1010000」のパターンになっており，このパターンが IR に読み込まれたときには加算回路を使用することになる．

機械命令には演算対象を指示するビットパターンがそれに続く．例えば先ほどの加算命令では先のパターンに続いて「何を何に足すのか」を指示するためのビットパターンが続き，演算対象となるデータを格納したメモリのアドレスが示されていたりする．

演算の途中には CPU 内部の一時記憶領域である**「データレジスタ」**を利用する．例えば「(a+b) – (c+d)」を計算するときには，括弧の中の計算を先に行っておいてその中間結果をどこかに覚えておく必要がある．このような中間結果をその都度メモリに書き戻していては計算速度が遅くなってしまう．なぜなら一般に CPU の速度に対してメモリの速度は非常に遅いため，メモリを読み書きしている最中には CPU は何もできずに単なる待ち時間になってしまうためである．そこで高速に読み書きできる作業領域として CPU 内部にデータレジスタが用意されている．M16C/62A ではデータレジスタは４個あり，その時使用されていないものならばどれを用いてもよい．データレジスタへ演算の途中結果を入れたり，集計結果を入れたりして，最終的な計算結果をメモリへ書き戻す．

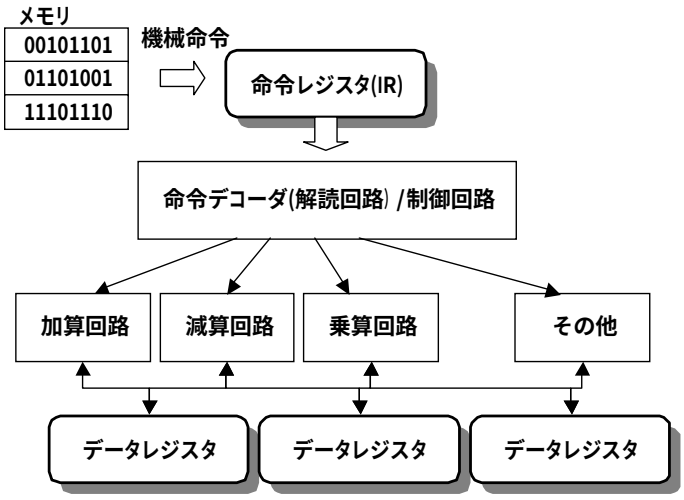


図 ３３　CPU の基本的な演算実行回路の概念図

7.4.2　ソフトウェアから見た CPU の構造

アセンブリ言語でプログラミングする場合，データレジスタなどを都合に合わせてうまく使い分けてプログラムを記述する．その他にプログラムから利用できる一般的な CPU 上の資源には，次のようなものがある．

「アドレスレジスタ」は，図 ３３では省略されていたメモリへアクセスする部分に用いるもので，機械命令の演算対象を指示する番地を格納するために用いる．

「プログラムカウンタ」は，プログラムの実行中に次の機械命令が格納されている番地を記憶するものである．これは「カウンタ」と呼ばれるように，自動的にカウントアップされ，次々と機械命令を実行していくことができる．このプログラムカウンタの内容を直接書き換えることで，任意の場所のプログラムを実行することができる．

「フラグレジスタ」は，演算回路で演算した結果生じた様々な状況を報告するためのものである．「フラグ」とは「旗」のことであり，ある条件が成立したときに「旗を立てて合図する」ように，ある特定の場所のビットに「1」をセットして合図を報告する．これを「フラグを立てる」と表現する．それ以外の時には「0」をセットして「フラグを下ろして」おく．例えば加算回路でオーバーフローが発生してしまったときなどに，その印として「オーバーフローフラグ」を自動的に CPU がセット

する．他には減算回路で結果がマイナスになってしまった時には「キャリーフラグ」などがあり，これら何種類ものフラグを寄せ集めたものがフラグレジスタである．

実際の M16C/62A では図 ３４のような各種レジスタが存在し，プログラムで使用することができる．データレジスタとして 16 ビットのものが４個（R0，R1，R2，R3），アドレスレジスタとして２個（A0，A1），20 ビットのプログラムカウンタ（PC），フラグレジスタ（FLG）として 16 ビットのものが用意されている．ただし，データレジスタ R0 と R1 に限って上位（High）と下位（Low）を分離して R0H と R0L という 8 ビットのレジスタとして扱うこともできる．

また**「割込みスタックポインタ」**（ISP）があるが，これは**「割込み処理」**をするときに使用するレジスタである．割込み処理とは，通常のプログラム実行の途中に何か緊急な別プログラムを間に「割り込む」ように実行するようなことをいう．例えばロボットを制御する場合を考える．障害物をセンサーで感知した時には緊急に回避行動を取る必要がある．そのため回避行動プログラムを，そのときロボットがどんなプログラムを処理している最中であっても，最優先に割り込んで処理させる必要がある．割込み処理は本実験では使用しないが，メモリのところで説明した「モニタ」プログラムが使用するため，アセンブリ言語でプログラムする際には適切な値に初期化しておく必要がある．

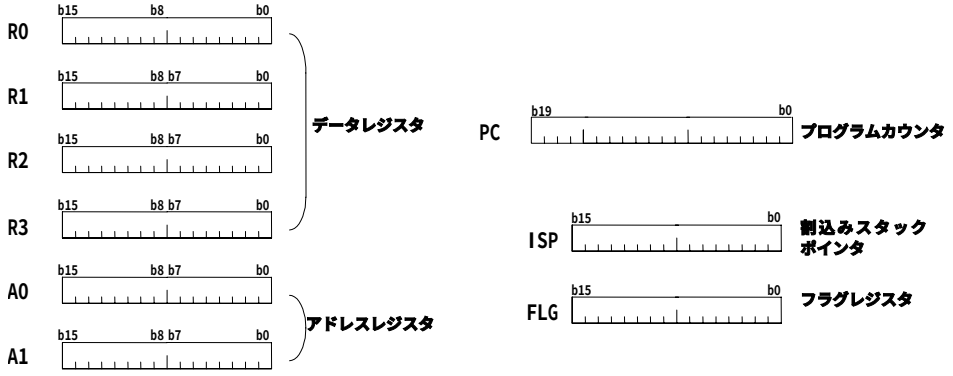


図 ３４ M16C/62A のレジスタ構成(一部)

8. マイクロコンピュータの基本動作

8.1 フェッチ＆エクゼキューション（命令の取得と実行）サイクル

コンピュータの構成要素が，どのように連携してプログラムを実行していくのかを説明する．例として，E0000h 番地から３命令の簡単なプログラムを実行する例を，図 ３５以降の図に示す．

図 ３５に示したように，CPU は，（１）プログラムカウンタの値に示されているアドレス使ってメモリを参照し，（２）そこに格納されている機械命令を命令レジスタに取得する．この動作を**「フェッチ（fetch：取得）」**と呼ぶ．この図ではアセンブリ言語のニーモニックとして「mov.b 400H,R0L」と書かれているが，実際にはそれに対応する２進数のコード「01110010111100000000010000000000」がメモリには格納されている．機械命令は 1 バイトとは限らず，この場合には 4 バイトでひとつの機械命令をあらわしている．

実は，メモリに格納された２進数のデータには，それが数値データなのか機械命令なのかを区別するための仕組みは存在しない．ただ，プログラムカウンタが指している場所には機械命令があるものだと CPU が勝手に思い込んでいるだけである．プログラムミスによってプログラムカウンタが機械

命令でないアドレスを指すことがあると，ただの数値データを無理やり命令として解釈し実行しようとするため予測不能な動作になり，大抵の場合にはコンピュータが**「暴走」**し，とんでもないアドレスでプログラムが停止することになる．

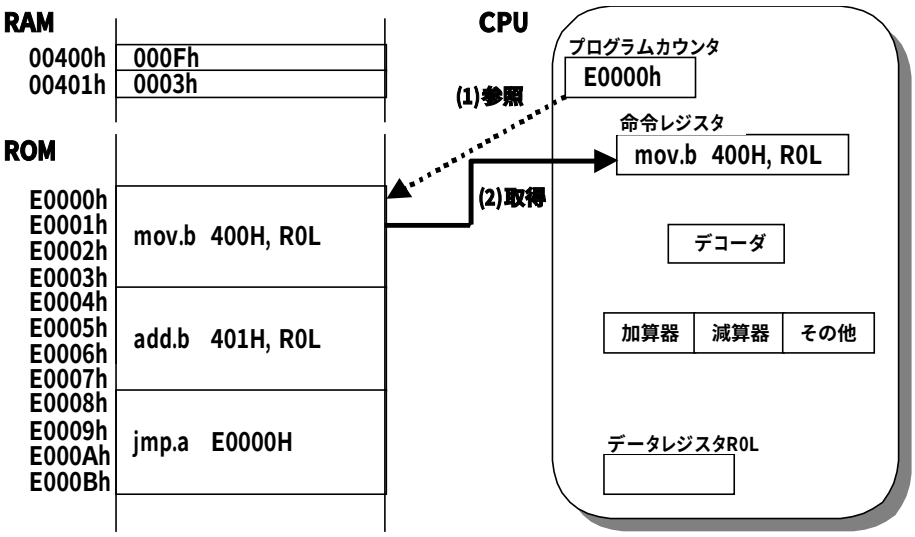


図 ３５　命令のフェッチ動作

次に図 ３６に移り，（３）取得した命令をデコーダで解釈し，（４）適切な演算回路を使って命令内容を実行する．この例では「mov.b 400H,R0L」という 400h 番地のメモリ内容を，R0L レジスタに転送を行う命令である．この結果，（５）R0L レジスタに「000Fh」という値が読み込まれる．そして読み込んだ命令が４バイトの命令だったので次の命令を実行する準備として，（６）プログラムカウンタの値を「＋４」カウントアップしておく．

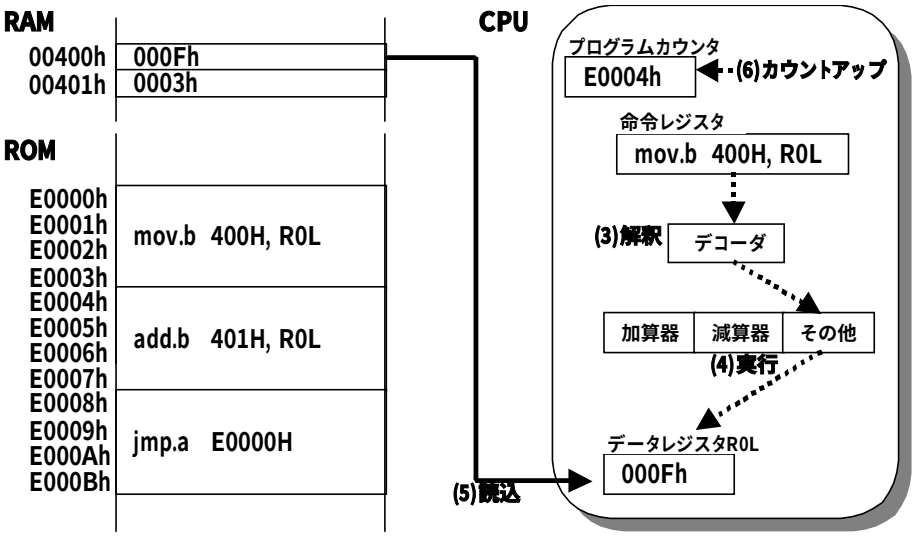


図 ３６　mov.b 命令の解釈実行動作

図 ３６の（４）演算回路の動作実行，（５）メモリの読み込みと，（６）カウントアップという動作は，表現の都合上分けて書かれているが，全体としてひとつの機械命令「mov.b」の実行動作として続けざまに行われるものである．

次に図 ３７に移り，そして，先の命令と同様に（7）次の命令を参照して（8）フェッチする．

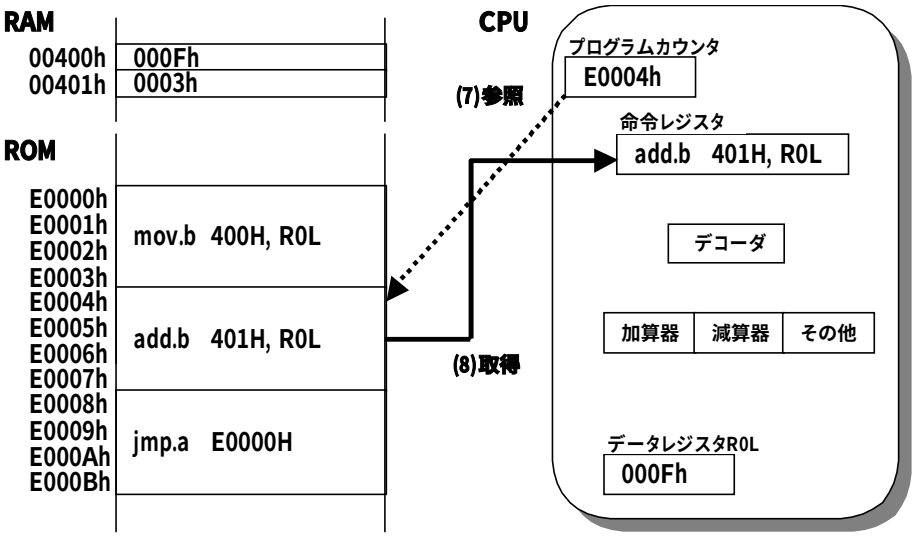


図 ３７ 次の命令のフェッチ動作

次に図 ３８に移り，（9）解釈し，（１０）実行する．今度の命令は「add.b 401H,R0L」である．これは加算命令なので加算器が使用される．その命令内容は 401h 番地のデータとデータレジスタ R0L との内容を足し算し，その結果を R0L に書き戻すというものである．そして次の命令のために（１２）プログラムカウンタをカウントアップする．

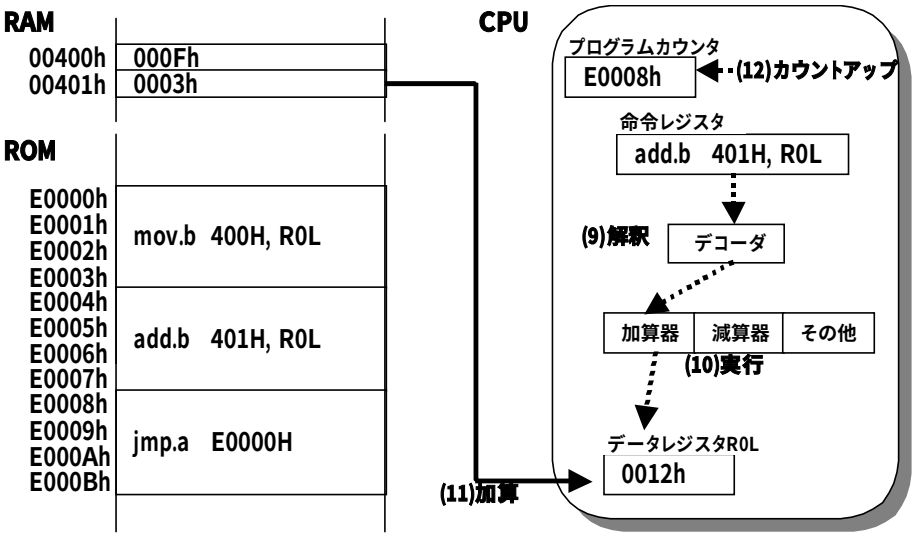


図 ３８ add.b 命令の解釈実行動作

次に図 ３９に移り，同様に（１３）次の命令を参照して（１４）フェッチして，次の命令「jmp.a E0000H」を読み出す．

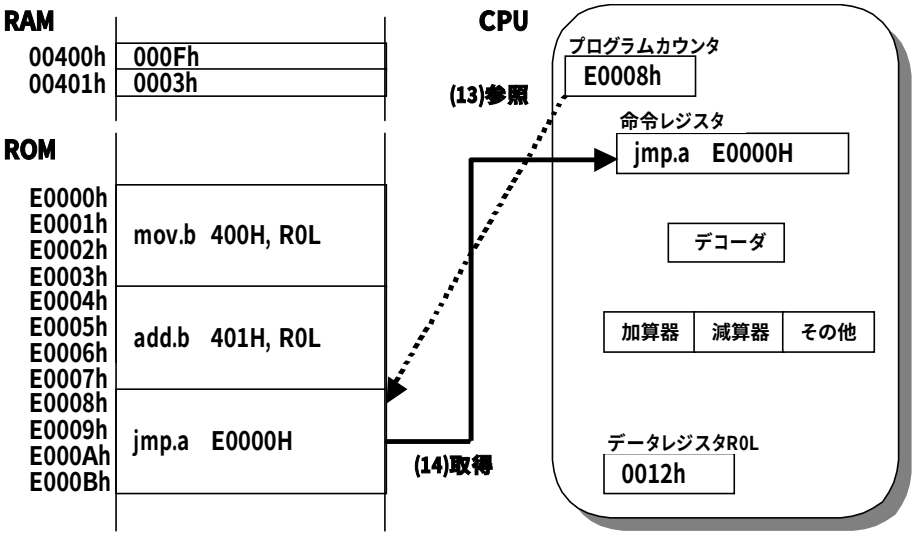


図 ３９　３番目の命令のフェッチ動作

図 ４０に移って命令を（１５）解釈し，（１６）実行する．jmp.a 命令は，ジャンプ命令と呼ばれ，プログラムカウンタの値を任意に設定しなおすものである．プログラムカウンタの値を書き換える，すなわち，プログラムの実行先をジャンプさせるためのものである．その実行結果として（１２）プログラムカウンタに指定された番地 E0000h がセットされる．

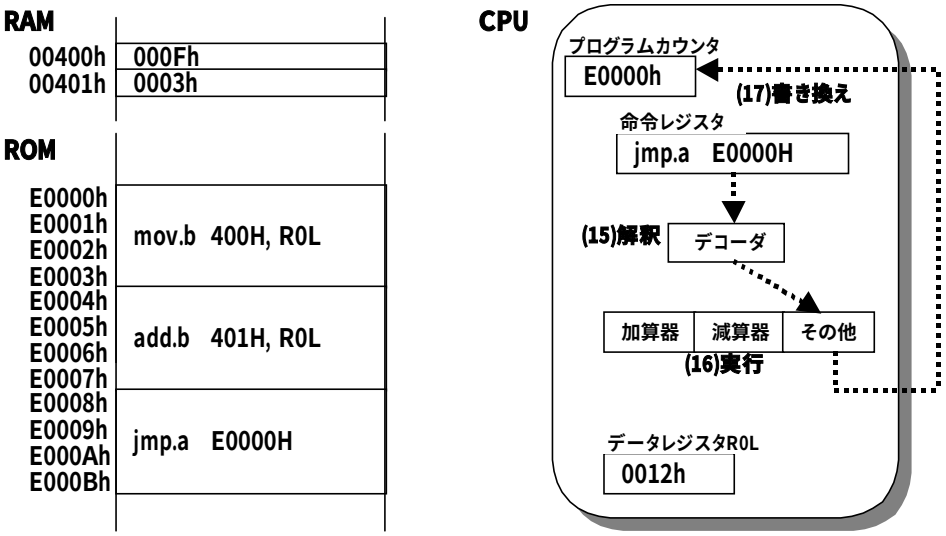


図 ４０　jmp.a 命令の解釈実行動作

このようなコンピュータの基本動作を**「フェッチ&エクゼキューション（fetch & execution：命令の取得と実行）サイクル」**と呼ぶ．つまり，命令をメモリから取り出してそれを実行するという単純動作を，ただひたすらに繰り返すのである．

これはイギリスのケンブリッジ大学で世界最初のプログラム内蔵（stored program）方式のコンピュータ「EDSAC」が 1949 年に作成されて以来，驚くべきことに**半世紀以上に渡って不変のコンピュータの基本動作**である．確かにレジスタの数や機械命令の種類，高速化のための改良がなされてきたものの，現代のコンピュータでも本質的な部分はまったく変更されていない．現代のパーソナ

ルコンピュータがどれほど派手なグラフィクスを提供しようが，スーパーコンピュータがどれほど高速で大容量であろうが，恐れる理由はいささかも存在しない．所詮すべてのコンピュータは命令を取り出して実行する以外に能のない単純な機械でしかないのだ．

9. アセンブリ言語

9.1 サンプルプログラムの解説

実際にアセンブリ言語でプログラムを作成する．ここで説明するアセンブリ言語は M16C/62A 用のものであり，その他の CPU（例えば Intel の Pentium4 など）用のものとは異なる．ただし，基本的な枠組みや考え方などは基本動作原理が同じである以上，どの CPU でも同じようなものである．M16C/62A 用のアセンブリ言語を使えれば，その他の CPU 用のものも簡単に使えるようになる．

一番単純なサンプル例を図 ４１に示す．左端の行番号は，説明の都合上ここで書き加えたもので，**実際には行番号は不要**である． 単純なものでも 35 行もあるように見えるが，そのうち大部分は，表記上の約束や，マイコンの初期化のためどんなプログラムにも付け加えなければならないような決まりきったルーチンである．肝心な部分は 14 行目から 19 行目の **5 行だけ**である．

```
01: .section        program,code
02: .org    0e0000h
03:
04:         LDC     #024BFH,ISP             ; 割り込みスタックポインタセット
05:         BSET    1,000AH                 ; プロセッサモードレジスタ書き込み許可
06:         MOV.B   #00000000B,0004H        ; シングルチップモード
07:         MOV.B   #00000000B,0005H        ; 非拡張、ノーウェイト
08:         BCLR    1,000AH                 ; プロセッサモードレジスタ書き込み拒否
09:         BSET    0,000AH                 ; クロックコントロールレジスタ書き込み許可
10:         MOV.B   #00001000B,0006H        ; 発信
11:         MOV.B   #00100000B,0007H        ; 分周なし
12:         BCLR    0,000AH                 ; レジスタ書き込み拒否
13:
14:         mov.b   #45H, 400H              ; 400H番地 (RAM) に被加数を書き込む
15:         mov.b   #67H, 401H              ; 401H番地 (RAM) に加数を書き込む
16: main:
17:         mov.b   400H, R0L               ; 400H番地 (RAM) のデータをレジスタR0Lに読み込み
18:         add.b   401H, R0L               ; 401H番地のデータとレジスタR0Lのデータを加算
19:         jmp     main
20:dummy_int:
21:         reit
22:
23: section         fvector
24:.org     0fffdch
25:
26:         .LWORD  dummy_int       ; FFFDC～F Undefined instruction
27:         .LWORD  dummy_int       ; FFFE0～3 Overflow
28:         .LWORD  dummy_int       ; FFFE4～7 BRK instruction
29:         .LWORD  dummy_int       ; FFFE8～B Address match
30:         .LWORD  dummy_int       ; FFFEC～F Single step
31:         .LWORD  dummy_int       ; FFFF0～3 Watchdog timer
32:         .LWORD  dummy_int       ; FFFF4～7 DBC
33:         .LWORD  dummy_int       ; FFFF8～B NMI
34:         .LWORD  start           ; FFFFC～F RESET
35: .end
```

マイコンの初期化

プログラム本体 (課題ではこの部分だけを書き換える)

割込み処理の初期化

図 ４１　簡単なプログラム例

まず，1 行目（と 23 行目）にある「.section」（先頭のドットに注意）は，**「指示命令」（擬似命令）**と呼ばれ，どのように機械語へのアセンブルを行うかを指示するものである．これはアセンブ

ラ（アセンブル作業を行うプログラム）への指示を行うための記述であり，機械命令を示すニーモニックではないため，実際にプログラムコードとしてマイコンのメモリに格納されることはない．２行目の「.org」も同様，先頭にドットのある命令は指示命令として扱われる．

「.section」命令は，プログラムファイルの中身をいくつかの**「セクション(部分)」**に分割するための命令である．プログラムファイルと呼ばれてはいるが，その中身にはプログラムコードだけでなく，データそのものの値や，データを格納しておく場所の指示も必要である．アセンブリ言語で「セクション」とは，例えば「プログラムセクション」「データセクション」などのように，種類別にまとめられてメモリ上に割り付けるための，１つの単位として扱われる．

まず１行目で，このセクションの名前が「program」と定義され，その属性は「code」すなわち機械語の命令であることを示している．「program」の部分は任意な名前に変更可能である．それから 2 行目によって，この program セクションを「0e0000h」番地を先頭にしたメモリ上の領域に格納することを意味する．program セクションは，３行目から次の.section 命令（23 行目）までの部分，すなわち４行目から 22 行目までの部分となる．

program セクションの中身としてプログラムコードが書かれているが，4 行目から 12 行目まではマイクロコンピュータの初期化部分である．本実験で使用する M16C/62A では，4 行目の割込みスタックポインタの初期化，５～８行目のプロセッサモードレジスタの初期化，９～１２行目のクロックコントロールレジスタの初期化が必要である．これら初期値などに関しては CPU などハードウェアに依存する話であり，本実験での学んで欲しい本質的な部分からは少々はずれるため，ここでの解説は省略する．本実験で作成するプログラムには，**毎回この 12 行分を書いておく（サンプルからコピーしてペースト）だけでよい．**

3 行目から 19 行目までのプログラムコードの部分には，行の途中の「;（セミコロン）」の後に**「コメント（注釈，プログラムの解説）」**が書かれている．アセンブラは，セミコロン記号の後は人間が読むためのコメントだと解釈する．コメントは機械語には一切影響を与えない．

肝心のプログラムの本体部分が 14 行目から 19 行目で，図 ３５から図 ４０まで説明した 3 行のプログラムに，400 番地と 401 番地にあらかじめ値を入れておく２行を追加しただけのものである．

20 行目以降は「割込み処理」のための記述である．割込み処理については本実験では使用しないが，6.4.2 節のところで少し解説した．20 行目 21 行目にあるのは，割込み処理のための**「ダミールーチン」**である．「ダミー」とは身代わりのことで，本実験では割込み処理を使用しないため，ここでは何もしないルーチンを用意している．続く 23 行目からは割込み処理のためのルーチンを登録しておく「割込みベクタ」を設定するセクションである．先ほどのダミーを登録してある．本実験では，20 行目以降最後までの記述も，プログラムファイルの**最後に毎回書いておく必要がある．**

自分でプログラムを書くためには，結局サンプルプログラムの 14 行目から 19 行目までにあたる部分を目的に合わせて書き換えればよい． 5 行に限定されず，必要なら何行でもこの部分に追加してよい．本実験ではその作業に集中すればよい．

9.2 アセンブリ言語の構文（ニーモニックの使い方）

M16C/62A には多数の機械命令が用意されているが，主要なものだけでも本実験の課題のプログラムを作成することができる．なお，ニーモニックを機械語に翻訳する M16C/62A のアセンブラ AS30 では，アルファベットの**小文字・大文字は区別しない**ため，どちらでニーモニックを表記しても構わない．

M16C/62A のアセンブリ言語の構文は，原則として**「オペレーション．サイズ　ソース，デスティネーション」**の形式である．**「オペレーション」**とは機械命令の種類を示す．サイズはその命令の対象の大きさが 1 バイトなら「.B」，2 バイトなら「.W」（Word:ワードの略）が存在する．また後述する JMP 命令ではそれに加えて 20 ビットのアドレスを示す「.A」も使える．**「ソース（src: source）」**とは命令対象のデータが存在する場所のことを示す用語である．**「デスティネーション（dest: destination）」**とは，命令を実行した結果の格納先を示す用語である． ソースとデスティネーションは，命令の演算対象を示す言葉であり，**「オペランド」**と呼ばれる．

例えば「mov.b #45H, 400H」のようになる． これは「mov」命令という転送命令を示す．ソースとなるデータは「45h」という２進数そのもの，デスティネーションとして 400H 番地が指定されたものである．ソースとして指定できるのは主として表 ７の４つである．

表 ７　主だったソースの指定方法

データの位置	記述方法	記述例
（1）即値データ	#数値	#45H
（2）レジスタ上のデータ	レジスタ名	R0L
（3）メモリ上のデータ	アドレス	400H
（4）アドレスレジスタで指定されたメモリ上のデータ	[アドレスレジスタ]	[A0]

（1）**「即値（イミディエイト）データ」**とは，データのありかを指示するのではなく，直接に演算対象のデータそのものを数値として命令中に記述することを意味している．他の指定方式と区別するために，数値の先頭に「＃」をつけておく．数値表現は 2 進数でも 16 進数でも 10 進数でもよい．２進数の場合には「#01010101B」のように最後に「B」を，16 進数の場合には「H」を付記する．なにもなければ 10 進数だと解釈される．

（2）レジスタ上のデータを指定するためには，6.4.2 節の図 ３４にあるレジスタ名を指定する．ただし，R0 と R1 に関しては 8 ビットずつ R0L と R1H，あるいは R1L と R1H としても指定できる．

（3）メモリ中のデータを指定するには，アドレスをそのまま指定するだけでよい．ただし，2 バイトのデータ（ワード）の場合，図 ４２のように格納される．

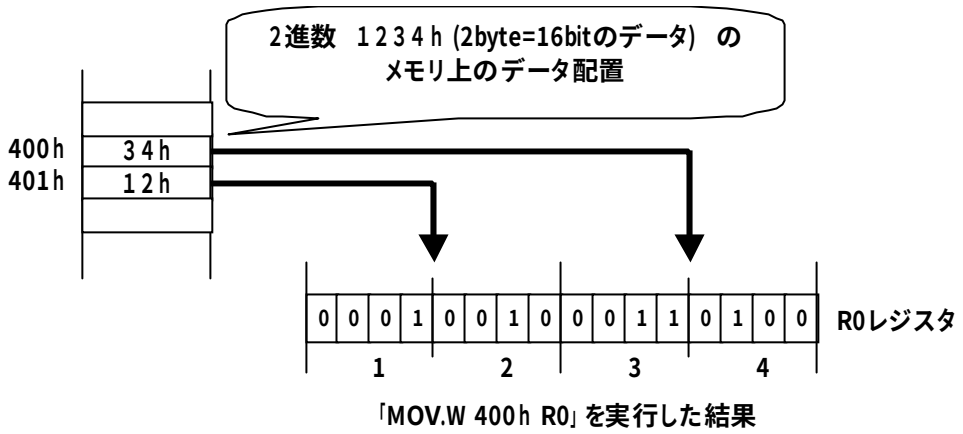


図 ４２　メモリ上のデータ配置（エンディアン）

M16C/62A では最下位バイト（数が小さい方の桁）から順にメモリに格納されるが，これとは逆順に格納されるような CPU も存在する．この方式の違いを**「エンディアン」**と呼ぶ．M16C/62A のような方式を**「リトルエンディアン」**，逆を**「ビッグエンディアン」**と呼ぶ．ちなみに，パソコンで

よく使われている Intel 社の CPU は，このリトルエンディアン方式である．

最後は，アドレスレジスタの内容によって示されているメモリアドレスにデータが可能されていることを示す．他の（１）から（３）のようにデータ位置を直接指定せずに，一度アドレスレジスタを介するため，これを**「間接アドレッシング方式」「間接アドレス指定方式」**などと呼ぶ（図 ４３）．

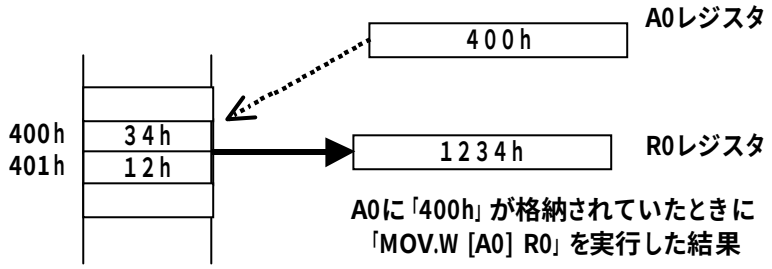


図 ４３　間接アドレッシング方式

この方式にはデータレジスタは使用できず，アドレスレジスタ A0 と A1 だけが使用できる．この方式ならデータの場所が固定されないため，何か計算によってアドレスレジスタの値を書き換えるようなプログラムを書けば，様々なデータを取り替えて使用できる．

デスティネーションとして指定できるものも表 ７に準じる．ただし（１）即値だけは指定できない（考えてみればあたりまえ）．

9.3　ニーモニック一覧

実験で使用するニーモニックは表 ８の通りである．項目の最後にある「フラグ変化」については次の 8.4 節で説明する．また，最後の「JMP」「J 条件」命令については続く 8.5 節で説明する．

9.4　フラグレジスタ

6.4.2 節で説明したように，M16C/62A にはフラグレジスタ（FLG）が存在する．各命令の演算結果に応じて，特定の条件が成立したときに「旗を立てて合図する」ように，フラグレジスタの特定ビットに「1」がセットされる．

わざわざこのようなフラグを用意してあるのは，8.5 節で説明する分岐命令が使用するためである．例えば，ある計算での減算の結果がマイナスになったかどうかで，実行する処理を切り替えるような時にこのフラグが意味を持つ．

M16C/62A のフラグレジスタの中で，本実験で使用する各機械命令によって変化するフラグは図４４の４つである．各フラグは表 ９の条件の時にセットされる．

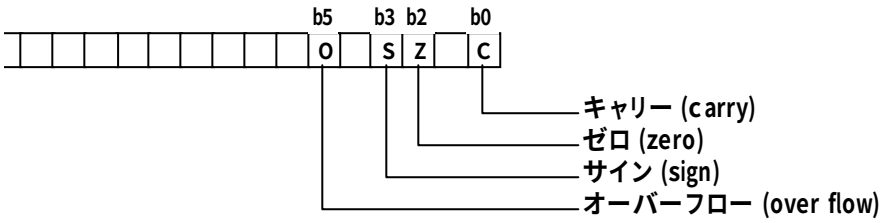


図 ４４　フラグレジスタ（FLG）内のフラグビット（一部）

表 8　ニーモニック一覧（一部）

命令	サイズ	オペランド	内容	記述例	フラグ変化
MOV	B か W	src,dest	src を dest にコピー	「mov.b R0,R1」 「mov.w #1234h, 400h」	S,Z
LDC	なし	src,dest	src を dest に，ただし dest は ISP や FLG など専用レジスタのみ	「ldc #024BFH,ISP」	dest が FLG なら全変化
XCHG	B か W	src,dest	src と dest を交換，ただし src はデータレジスタのみ	「xchg.w R0 A0」	なし
BCLR		N,dest	dest の N ビット目を 0 にクリア	「bclr 0,R0」 「bclr 4,400H」	なし
BSET		N,dest	dest の N ビット目を 1 にセット	「bset 4,R1」 「bset 7,[A0]」	なし
AND	B か W	src,dest	src と dest の論理積をとり dest に	「and.w R0 R1」	S,Z
OR	B か W	src,dest	src と dest の論理和をとり dest に	「or.w R0 R1」	S,Z
XOR	B か W	src,dest	src と dest の排他的論理和をとり dest に	「xor.w R0 R1」	S,Z
NOT	B か W	dest	dest をビット反転	「not.w R0」	S,Z
ADD	B か W	src,dest	src と dest に加算，結果を dest に	「add.w R0,R1」 「add.b R0L,400H」	O,S,Z,C
SUB	B か W	src,dest	dest から src を減算，結果を dest に	「sub.w R0,R1」 「sub.b R0L,400H」	O,S,Z,C
INC	B か W	dest	dest に 1 加えて結果を dest に	「inc.w R0」	S,Z
DEC	B か W	dest	dest から 1 引いて結果を dest に	「dec.w R0」	S,Z
SHL	B,W,L	src,dest	src に示された数だけ dest を論理シフト，src>0 のとき左シフト，src<0 のとき右シフト，src には R1H か即値のみ	「shl.w R1H,a0」 「sh.w #-8,R1」	S,Z,C
SHA	B,W,L	src,dest	src に示された数だけ dest を算術シフト，src>0 のとき左シフト，src<0 のとき右シフト，src には R1H か即値のみ	「sha.w R1H,a0」 「sha.w #-8,R1」	O,S,Z,C
ROT	B か W	src,dest	src に示された数だけ dest をローテート，src>0 のとき左回転，src<0 のとき右回転，src には R1H か即値のみ	「rot.w R1H,a0」 「rot.w #-8,R1」	S,Z,C
CMP	B か W	src,dest	dest から src を減算した結果でフラグレジスタを変化（dest は何も変化しない）	「cmp.w R0,R1」	O,S,Z,C
FCLR	なし	dest	dest として，「O」「S」「Z」「C」などフラグを指定．フラグレジスタの指定されたフラグビットをゼロクリア	「fclr Z」	指定フラグ
JMP	S,B,W,A	Label	指定された label を PC にセット（無条件ジャンプ），詳しくは 8.5 節で説明	「jmp main」 「jmp.a #e0000h」	なし
J 条件	なし	Label	条件によって指定された label を PC にセット(条件ジャンプ，分岐命令)．詳しくは 8.5 節で説明	「jeq main」 「jnc main」	なし

表 9 フラグ変化の条件

O	SHA	演算した結果最上位ビットの値が変化したときに 1 にセット．それ以外のときは 0.
	その他	符号付き演算だとした場合，「.W」のときには 15 ビット，「.B」のときには 7 ビットで表現可能な数の範囲を超えたときに 1 がセット．それ以外は 0.
S	全命令	演算の結果最上位ビット(MSB)が 1 なら 1 にセット．0 なら 0 にセット.
Z	全命令	演算の結果が 0 なら 1 にセット．それ以外なら 0.
C	ADD	符号なし演算の結果，「.W」のときには 16 ビット，「.B」のときには 8 ビットで表現可能な範囲を超えたときに 1 がセット．それ以外は 0.
	SUB，CMP	符号なし演算の結果が 0 以上のとき 1 にセット．0 未満のときは 0.
	SHA，SHL，ROT	最後に追い出されたビットが 1 なら 1 にセット．それ以外のとき 0.

9.5　ジャンプ命令とアドレスラベル

機械語プログラムは，7.1 節で説明したようにメモリに格納された順番に 1 命令ずつ実行される．ただし，「jmp.a #e0000h」のようなジャンプ命令によって，次に実行する命令の番地を変更することができた.

しかし，命令がどのようにメモリに格納されるのかいちいち考えながらでは，プログラムを書くことが大変な作業となる．そこで表 8に示したように jmp 命令ではオペランドに**「ラベル」**を指定することが一般的である．ラベルとはメモリ上のある番地につける名前のことで，図 ４１で示したプ

ログラムでも使われている．このプログラムの 14 行目から 19 行目をもう一度図 ４５として示す．

```
14:         mov.b    #45H, 400H          ; 400H番地 (RAM) に被加数を書き込む
15:         mov.b    #67H, 401H          ; 401H番地 (RAM) に加数を書き込む
16: main:
17:         mov.b    400H, R0L           ; 400H番地 (RAM) のデータをレジスタR0Lに読み込み
18:         add.b    401H, R0L           ; 401H番地のデータとレジスタR0Lのデータを加算
19:         jmp      main
```

図 ４５　サンプルプログラムの一部（再掲）

16 行目にある「main:」がラベルである．アルファベットでつけられた名前の後にコロン(:)を付けることでラベルの定義として解釈される．実際の番地とは異なるが，14 行目の命令から E0000H 番地に格納された場合のイメージを図 ４６に示す．１命令が 4 バイトのためこのように順に格納される．16 行目に書かれたラベルはメモリには格納されず，次の 17 行目に書かれた命令の場所を「main」と名付けることを意味する．そして 19 行目の「jmp main」と書かれた命令では，実際にはこの「E0008h」が指定されたと見なされる．

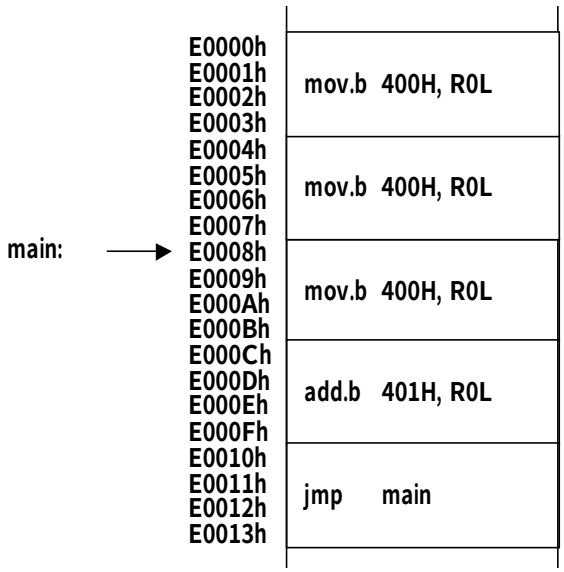


図 ４６　命令のメモリ格納イメージ

このような「main」というラベルを「E0008h」に置き換える作業は，アセンブラなどが自動的に行い，アセンブリ言語のプログラムが機械語にアセンブルされた段階で終了する．実際にメモリに格納され実行される機械語の動作そのものには一切関係しない．ただし，コンピュータの動作にとっては意味がなくとも，このプログラムをデバッグする人間にとっては重要な意味を持つため，ラベルに関する情報も，参考として実行プログラムに付け加えられており，デバッガで利用される．

ジャンプ命令ではサイズのところに「.S」「.B」「.W」「.A」が指定できることになっているが，ジャンプする先が現在地からどの程度離れているか指定するもので，通常これは省略してかまわない．

このジャンプ命令は，無条件にジャンプ先を指定するため特に**「無条件ジャンプ命令」**とも呼ばれる．これに対し，ジャンプするかどうか条件がつくものを**「条件ジャンプ命令」**と呼ぶ．つまり，ある条件が成り立っている時だけジャンプし，それ以外のときはそのまま次の命令を実行する命令のことである．プログラムの流れがその命令によって分岐することから**「分岐命令」**とも呼ばれる．

表 １０　条件ジャンプ命令の条件一覧

条件名　別名	フラグ条件	その意味
GEU　C	C が 1 (C=1)	以上(<=)だった
LTU　NC	C が 0 (C=0)	未満(<)だった
EQ　Z	Z が 1 (Z=1)	等しかった(=)
NE　NZ	Z が 0 (Z=0)	等しくなかった
GTU	C が 1 しかも Z が 0 ($C \bullet \overline{Z} = 1$)	大きい(<)
LEU	C が 0 または Z が 1 ($C \bullet \overline{Z} = 0$)	以下(>=)だった
PZ	S が 0 (S=0)	0 以上の正の数だった
N	S が 1 (S=1)	負数だった
GE	S と O が一致 ($S \oplus O = 0$)	符号つき演算だとして，以上(<=)だった
LE	S と O が不一致，または Z が 1 ($(S \oplus O) \bullet Z = 1$)	符号つき演算だとして，以下(>=)だった
GT	S と O が一致，しかも Z が 0 ($(S \oplus O) \bullet Z = 0$)	符号つき演算だとして，大きかった(<)
LT	S と O が不一致 ($S \oplus O = 1$)	符号つき演算だとして，小さかった(>)
O	O が 1 (O=1)	オーバーフローした
NO	O が 0 (O=0)	オーバーフローしてない

条件ジャンプ命令で使われる条件は，表 １０の 14 種類である．すなわち条件ジャンプ命令は 14 種類が存在する．それぞれの条件ジャンプ命令は，条件名を「J」に続けて表記したもので，例えば「GEU」条件を使う命令は「JGEU」（または「JC」）命令である．「JGEU main」のようにラベルを指定して利用する．それぞれ表に示したフラグの条件が成り立ったときに指定されたラベルへのジャンプが行われ，それ以外の時には何もしない（次の命令が実行される）．

9.6　I/O ポートを使ったプログラム例

実際にスイッチからデータを読み，LED を点灯させるためには，6.3 節で説明したように I/O ポートへアクセスすればよい．具体的には表 ５で各装置のポート番号を確認し，そのポートへアクセスするための SFR アドレスを表 ６で確認する．その SFR アドレスへ通常のメモリと同様にアクセスするだけで，各装置をコントロールすることができる．ただし，M16C/62A の I/O ポートは入出力兼用なので，あらかじめ入力ポートか出力ポートかを，方向レジスタ（これも表 ６で確認）に設定しておく．入力ポートの場合，方向レジスタの各ビットを「0」，出力ポートの場合には「1」を指定すること．表 ５と表 ６の関係を，もう一度図 ３２で確認しておくこと．

例えば，　トグルスイッチから入力したい場合には，

```
mov.b #00000000b, (トグルスイッチに対応したポート用の方向レジスタの SFR アドレス)
mov.b (トグルスイッチに対応したポートの SFR アドレス), R0L
```

とするだけで，トグルスイッチの状態をデータレジスタ R0L へ読み込むことができる．

LED に出力したい場合には，

```
mov.b #11111111b, (LED に対応したポート用の方向レジスタの SFR アドレス)
mov.b R0L, (LED に対応したポートの SFR アドレス), R0L
```

とするだけで，R0L の値を LED へ出力することができる．

10. デバッグ方法

10.1　プログラムの動作確認とデバッグ作業

自分で作成したプログラムをアセンブルして実行するには，4.3 節で説明したサンプルプログラム

の実行方法のところを，自作のプログラムに読み替えて行えばよい．

このプログラムが正しく動いているかどうかの確認が必要である．プログラムミスのことを俗に**「バグ（虫）」**と呼び，これを探して修正する作業のことを**「デバッグ作業」**と呼ぶ．デバッグ作業を行うためのプログラミングツールとして**「デバッガ」**が存在する．M16C/62A の開発環境では，デバッガとして**「KD30」**が用意されている．

デバッグ作業として一般的な方法は，プログラムを１命令ずつ実行し，その都度書き込まれたメモリの値やレジスタの値を確認し，プログラムがその段階まで正しく動いているか（**プログラマの企図している通りに動いているか**）どうかを確認する．この作業をプログラム動作の**「トレース（追跡）」**と呼ぶ．

このデバッグ方法は，7.1 節でプログラムのフェッチ＆エクゼキューションサイクルを説明した通りに，コンピュータの動作を確かめながら進む作業である．コンピュータの動作原理が半世紀以上に渡って変更されていないことと同様，プログラムのトレースによるデバッグ方法もコンピュータの歴史と共にずっと行われ続けてきた．逆にいえば，好むと好まざるとに関係なく，すべてのプログラマはこの手法によるデバッグ方法を習得せざるを得ない．C 言語や Java 言語のプログラムの場合でも，機械命令ではなくプログラムの一行一行を追跡し，本質的には同じ作業が必要とされる．

KD30 にはプログラムのトレースによるデバッグを行うための機能が付与されている．4.2 節で説明したように，OAKS16LCD ボードキットの開発環境ではボードキットに接続されたパーソナルコンピュータ側からリモートコントロールする形でデバッグ作業を行う．

10.2 ステップ実行

機械命令を１つだけ実行することを「１ステップ」，2 命令分実行することを「２ステップ」と呼ぶ．そこから，１命令ずつ区切って段階的に実行することを**「ステップ実行」**と呼ぶ．M16C/62A 用のデバッガ KD30 では，M16C/62A をリモートコントロールしてステップ実行させるための機能がある．

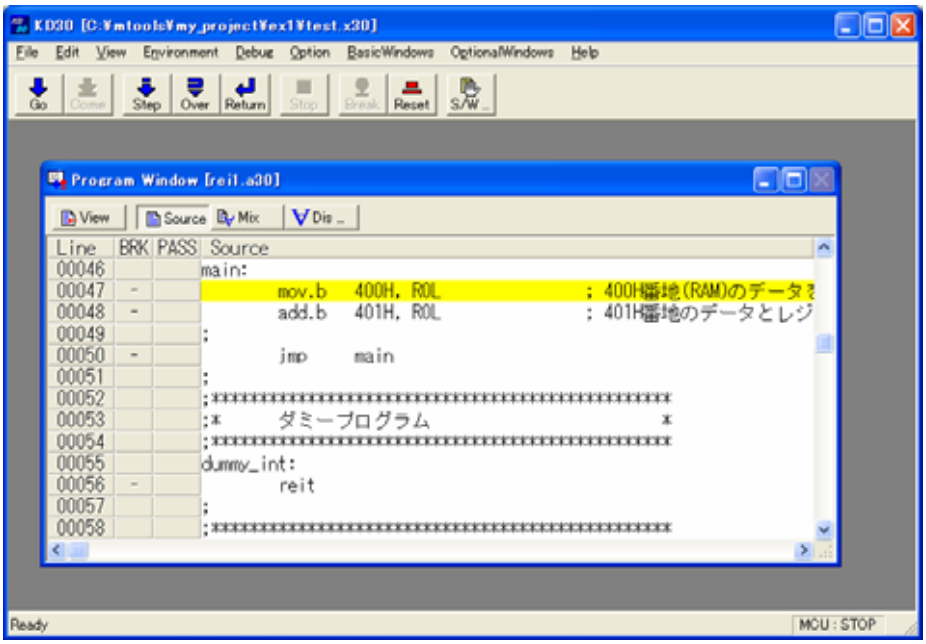


図 ４７　ステップ実行例

実際にステップ実行する操作は簡単で，対象とするプログラムを 4.3 節のように KD30 でロードし，その後「Go」ボタンではなくて**「Step」アイコン**をクリックするだけで，プログラムの先頭から 1 ステップだけ対象のプログラムを実行する．図 ４７に示したように，黄色いラインで示されているのが現在実行している機械命令であり，ボタンをクリックするたびに１つ進む．これを使って，例えば条件ジャンプが，プログラマが企図したように正しく行われているか確認したりする．

10.3 メモリやレジスタの確認

プログラムをステップ実行している際に，その瞬間正しくプログラマが意図した通りに計算を行っているかどうか，レジスタやメモリ内容を確認する．確認の仕方は，図 ４８のようにメニューの中から「BasicWindows」を選択し，その中から「Register Window」と「Memory Windows」をそれぞれクリックすればよい．

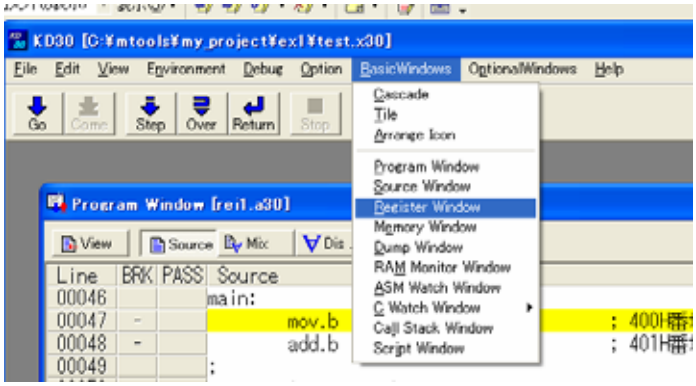


図 ４８　レジスタとメモリの内容を表示するウィンドウの表示指定

この作業によって，図 ４９の左に示したレジスタの内容を表示するレジスタウィンドウが表示される．PC レジスタや R0 レジスタなど各レジスタに格納されている値を 16 進数で示している．一番下に表示されている部分がフラグレジスタで，例えばキャリーフラグが「C」として表示されている．フラグレジスタの各記号は，8.4 節図 ４４のものと同じ意味を示す．

デバッグ時にはこの値を暫定的にその場で書き換えたいことがある．その場合には各レジスタの部分をダブルクリックして書き換えることができる．例えば R0 レジスタをダブルクリックすると，図 ４９の右に示したような画面が開く．

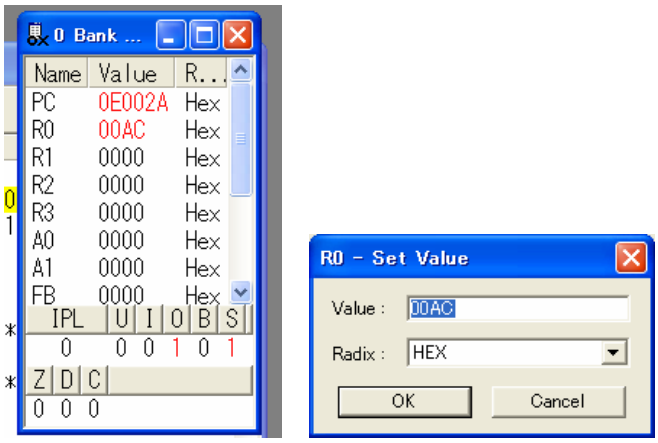


図 ４９　レジスタウィンドウと値の書き換えボックス

メモリの内容を表示するメモリウィンドウを図 ５０に示す．図は各アドレスのメモリ内容を 1 バイトずつ 16 進数で一列に表示したものである．このウィンドウ上部にあるアイコンをクリックすることで，10 進数として表示したり，機械語として表示したり，別の見方をできるようにしてある．7.1 節で触れたように，メモリ内容がデータであるか機械語であるかの区別は，メモリ自体ではできないため，それはデバッグする人間が判断することになる．

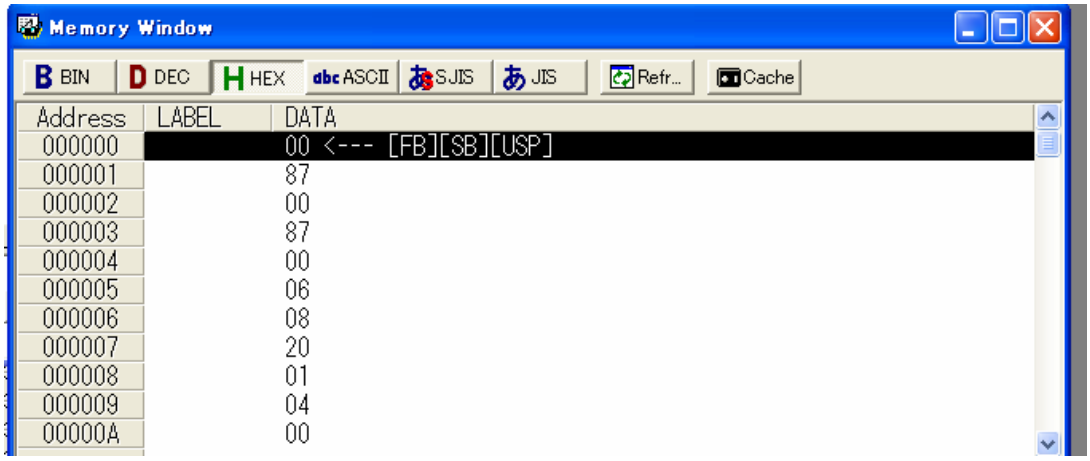


図 ５０　メモリウィンドウ

入力ポートに対応した番地をメモリウィンドウで表示した場合，表示内容が実際の値とずれることがある．入力ポートから読み込む値はハードウェア側の状態に応じてリアルタイムに変化するものの，デバッガ上でプログラムの実行を一時中断しているときには画面表示が切り替わらないためである．右上にある 「Refresh」ボタンを押すと，メモリウィンドウの表示が更新され，入力ポートの値やメモリ内容を最新のものに切り替える．

また，メモリウィンドウの表示は図 ５０のようにバイト単位の表示だけでなく，2 バイトのワード単位などでも表示することができる．表示単位の切り替えは，図 ５１のように「Option」→「View」→「Data Length」で選択することができる．Byte が 1 バイト単位，Word が 2 バイト単位，Lword が 4 バイト単位を示す．複数バイトを表示する場合，図 ４２で説明したエンディアンをよく考えること．

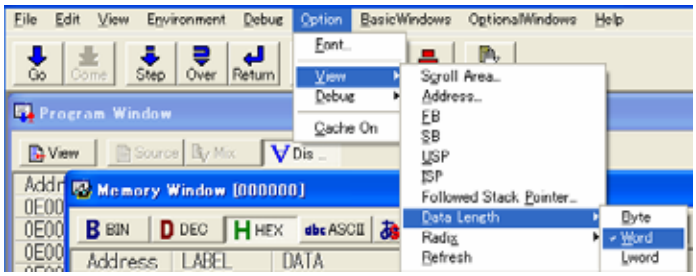


図 ５１　表示単位の切り替え

メモリウィンドウと同様にメモリ内容を表示するものの，一列ではなく，大量に表示するための画面が図 ５２のダンプウィンドウである．メモリの内容を片端から 16 進データとして全部吐き出す作業のことを**「メモリダンプ」**と呼ぶ．画面左の Address 行を見るとわかるように 16 進数で「10h」ごとになっている．その間のデータが横に一列に 16 個並べられた表示である．

メモリウィンドウとダンプウィンドウは，都合に合わせて見やすい方を選択すればよい．レジスタと同様，メモリ中のデータもデバッグ中に都合に合わせて書き換えたい場合がある．そのときには，両方のウィンドウとも書き換えたいデータ部分をダブルクリックすればよい．図 ５３のようなウィンドウが開き，値を直接書き換えることができる．

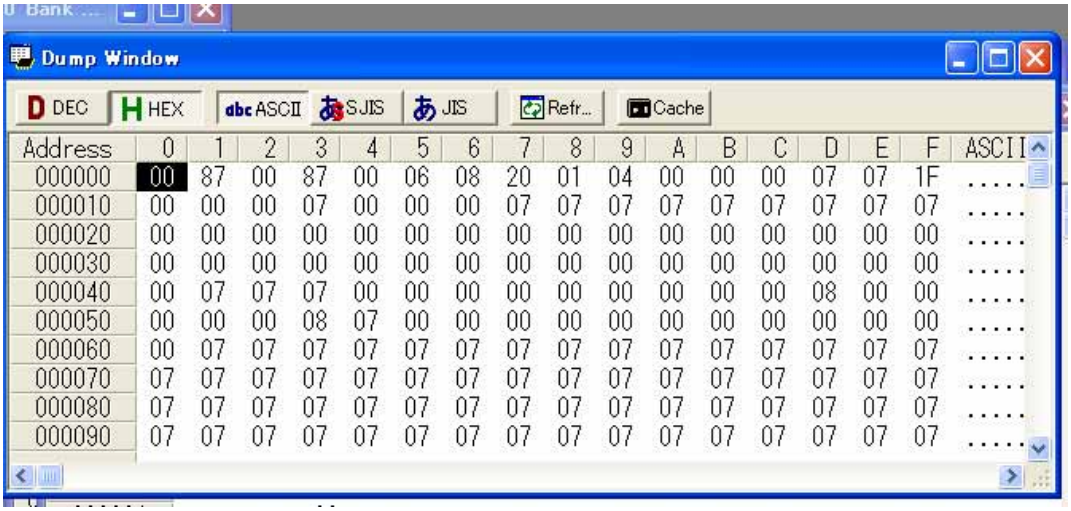


図 ５２　ダンプウィンドウ

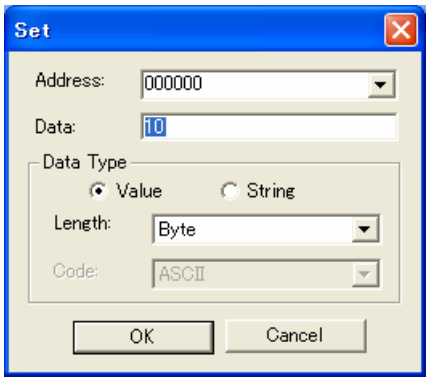


図 ５３　メモリ書き換えウィンドウ

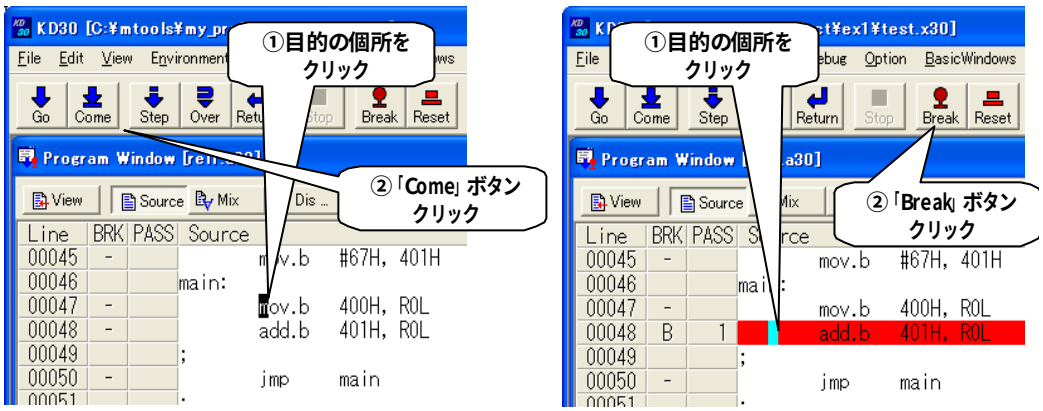
10.4 ブレークポイントの利用

例えば，大きなプログラムをデバッグしていたとして，その最後の部分をデバッグしたいとしよう．そのプログラムの先頭からステップ実行していては，肝心な部分に到達するまでに Step アイコンをクリックしなければならない回数を考えただけで大変になる．

そこで，今デバッグしたい部分まで，プログラムの実行を一気に進め，肝心な部分だけステップ実行することを考える．そのための機能が**「ブレークポイント」**である．これはプログラムの実行を中断したい個所（ポイント）を意味し，あらかじめデバッグしたい部分の先頭に指定しておくことで，そこまでは実行を進めてしまうことができるようになる．

ブレークポイントのうち，設定が簡単なものを**「簡易ブレークポイント」**と呼ぶ．これは図 ５４左に示したように，目的のプログラム個所をクリックしてカーソルを移動し，その後「Come」ボタンをクリックすると，その行まで実行が進ませることができる．無論，その行を実行することなくプ

ログラムが終了した場合には，その位置でとまることはない．これは，いわば一回限りのブレークポイントとして利用される．



(a) 簡易ブレークポイント機能　　(b) ブレークポイントの設定

図 ５４　ブレークポイント機能の利用

それに対して，図 ５４右側の方は，永続的なブレークポイントとして設定する方法を示している．簡易ブレークポイントと同様に目的の行にカーソルを移し，「Break」ボタンをクリックすることで設定できる．プログラム表示の左欄に「BRK」と書かれた欄があるが，ブレークポイントが設定されるとこの欄に「B」と表示され，画面上でその行が赤くなる．複数のブレークポイントも設定できるが，設定できる数は 4 個所までである．すでにブレークポイントが設定されている個所で，もう一度同じ操作をするとブレークポイントを解除できる．

ブレークポイントを設定したら，後は「Go」ボタンをクリックすると実行が開始され，実行がどれかブレークポイントに遭遇したところで中断する．中断した個所からステップ実行したり，あるいは再度「Go」ボタンで実行を再開したりすることができる．

最低限の紹介ではあるが，これら機能を使えばデバッグ作業を進めることができる．肝心な点は，自分で書いたプログラムがどのように実行されるのか頭でイメージする能力である．その自分が正しいと思っている動作イメージと，実際のステップ実行された動作が一致しないところに，プログラムミスが存在するわけである．自分で正しい動作イメージを思い浮かべられないとすればデバッグできるはずがない．