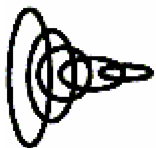
電子工学実験Ⅳ

３．ワンチップマイコン(H8)の学習

２００７年度

ARIKAWA / Haruhiko



大阪電気通信大学　工学部　第１部　電子工学科

学生番号　　　　　　　　　　　　氏名

MEMO

第３章　ワンチップマイコン(H8)の学習

０．学習の目的

・マイコン（Micro-Computer）のシステムを理解する．
・マシン語，アセンブラ言語，C 言語の相違を具体的に理解する．
・H8 用アセンブラ及び C コンパイラ、特にデバッガを使えるようになる．
・H8 が実装された評価ボードを実際に動かす．

１．マイコンの歴史と概要

１．１　世界最初のマイコン４００４の開発者

嶋　正利．1943〜．静岡生まれ．東北大学卒．

1960 年代後半の電卓業界は，LSI 化を巡って熾烈な競争を展開していた．LSI 化の先鞭をつけたのは，シャープである．４チップの MOSLSI を使って８桁電卓マイクロコンペット QT-8D を 1969 年商品化した．これは大いに注目され，電卓の LSI 化が加速した．

嶋は化学出身であるが化学会社に就職できずビジコン社に入社した．嶋の入社したビジコン社でも電卓を手がけており，シャープよりも更に高性能で安価な電卓の開発を計画していた．ビジコン社は，国内やアメリカで自分達の欲する LSI を開発してくれる企業を探したがなかなか見つからなかった．最後に訪れたインテル社でやっと話がまとまり，両社で電卓用の LSI を開発することになった．

インテル社と言えば今でこそ，世界最大の半導体メーカであるが，1968 年当時インテル社は従業員 100 人足らずのベンチャー企業であった．そこで嶋は，インテル社のテッド・ホフと電卓用 LSI の開発をすることになった．この時電卓用でなく，プログラムでいろいろな機能を実現できる４ビット CPU というアイデアが生まれ，それを実現したのが世界最初のマイクロコンピュータ 4004 である．1971 年にインテル社より 4004 マイクロプロセッサファミリが発売されている．

４ビット CPU のアイデアはホフであるが，それを実現する為の各種技術開発は嶋が行っており，実際の設計も殆ど全て嶋が行っている．その後，嶋はインテル社で発売され, ベストセラーになった名作８ビット CPU8008 を１人で設計している．更にインテル社を退社後，ザイログ社であの有名なＺ80，Ｚ8000 を設計している．当時重要なＣＰＵは，嶋がほとんど手掛けたといっても間違いない．

表１．１　インテル社マイコンの歴史（抜粋）

名称	発売年	最大速度	バス幅(BITS)	トランジスタ数
4004	1971	108KHz	4	2,300
8008	1972	108KHz	8	3,300
8086	1978	10MHz	16	29,000
i386	1985	33MHz	32	275,000
Pentium	1993	166MHz	64	3,100,000

最後の Pentium はパソコン時代となった現在最も有名です．

他に有名なマイコンには，モトローラ社の 68000 や，マイクロチップテクノロジ社の PIC シリーズがある．実験では，日立の H8 シリーズを使用する．

１．２　マイコンの基本的な内部構成例

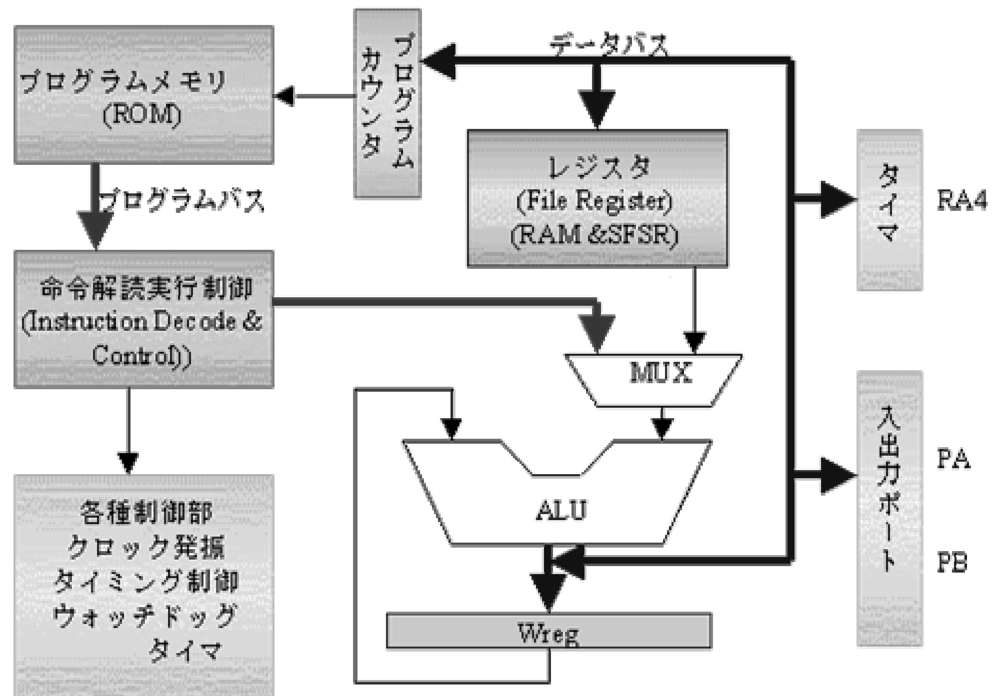


図１．１　ＰＩＣマイコンの内部構成

表１．２　ＰＩＣマイコンの内部動作

名　　称	機　能　内　容　／　動　作　内　容
プログラムメモリ　（ROM)	プログラムそのものの命令を格納する記憶場所です．　ROM は Read Only Memory の略で読み出し専用メモリという意味です．
プログラムカウンタ　(PC)	プログラムの実行を制御するカウンタで，このカウンタの内容が指しているアドレスのプログラムメモリにある命令が，次に実行される命令となる．PC: Program Counter
レジスタ　(File Register)	レジスタは２つの領域に別れる．一つは汎用のデータ格納メモリエリアでRAM(Random Access Memory)と呼ばれ，プログラム内で使う変数領域として使います．もう一つは特別なレジスタ領域で，SFR (Special Function Register) と呼ばれ，コンピュータ内の動作を決める色々な条件を設定したり，動作状態を知ることが出来ます．
命令解読，実行制御	プログラムカウンタが指すプログラムメモリ内の命令が読み出され，ここで命令種類の解読と各々に従った処理がなされます．その結果，各種の制御や，ＡＬＵ内で演算をしたりすることで命令が実行されます．
各種制御部	コンピュータ内部の直接計算とは関係の無いタイミングやウォッチドッグタイマなどの制御がなされる所です．
ＭＵＸとＡＬＵ（選択切替え部と算術演算部）	各命令の指示に従って各種レジスタや Wreg の内容との演算がなされる所で，いわゆるコンピュータの中の計算器に相当します．演算結果は再度 Wreg やレジスタに格納されたり，入出力ポートやタイマ，プログラムカウンタなどに格納されます． MUX： MultiPlexer　　ALU: Arithmetic Logic Unit

Wreg (Working register)	演算をするときの一時保管に使うレジスタで，演算の時の中心となって働きます．
内部データバス	命令実行に関係するデータを運ぶための通信路で，８ビットマイコンは８本の線で，16 ビットマイコンは 16 本の線で結ばれています．
入出力ポート	コンピュータ内部のデータを外部に出すためのポート（港）で，これが直接ＩＣのピンに接続されています．コンピュータがポートに「１」のデータを出力すると，ＩＣの対応するピンが「High(約 4.5V)」の状態になり，「０」を出力すると「Low(約 0.3V)」の状態となります．
タイマ	コンピュータ内部で特別な機能を果たす部分で，一定時間間隔を作ったり，回数をカウントしたり，というカウンターを元にした動作をします．

■命令実行時の動作概要

【例題１：加算命令】

① まず命令がプログラムメモリから取り出されます．（フェッチするという）
② 取り出された命令を解読し，加算命令と判明したら，ＡＬＵ（算術演算ユニット）に計算指示を出します．
③ ＡＬＵには，まず Wreg の内容が図の左側から入力され，右側には指定されたレジスタの値がレジスタから読み出されセットされます．（ＭＵＸの切替えも指示される）
④ 両者の加算の演算結果は，ＡＬＵの出力として現れます．この出力は，内部データバスを経由して指定されたレジスタに書き込まれます．

【例題２：分岐命令】

① まずプログラムメモリから命令を取り出します．
② 取り出した命令を解読し，分岐命令と判明したら，ＡＬＵに指示を出します．
③ 分岐先のアドレスの一部である「Address」をＡＬＵの左側に新しいアドレス下位データとしてセットします．
④ 現在のプログラムカウンタの値を読み出し，内部データバス経由ＡＬＵの右側にセットします．（ＭＵＸの切替えも命令内容で指示される）
⑤ ＡＬＵでアドレスの下位を命令で指定されたデータに置き換えをします．
⑥ 演算結果をプログラムカウンタに戻し中身を書き換えます．
⑦ 新しいプログラムカウンタの指し示す命令の実行に移ります．

【例題３：ウォッチドッグタイマ制御】

① まずプログラムメモリから命令を取り出します．
② 取り出した命令を解読し，制御命令だと判明したら，直接制御部に信号を送ります．
③ そしてウォッチドックタイマのリセットをします．

■その他の用語の解説

【　クロック　】

　クロックというのは，コンピュータの世界や，ディジタルロジックの世界では，一般に回路を動かすためのペースメーカとして使われる信号のことを言い，常に一定の周波数の信号を使います．いわば人間の心臓と同じ働きをし，全ての回路のタイミングの基となる信号です．マイコンは，外部部品で作られるクロック信号を基にして，内部クロックを作り出し，全てのタイミング制御を行っています．従って命令の実行もこのクロックが基になって実行されています．これから命令の基本サイクルが何クロックで実行されるなどという言い方がされます．

【　入出力　】

　「入出力」とは，コンピュータ自身が外部との関係を持つための出入口のことで，「入出力ポート」とか「ＩＯ（アイオー）ポート」とか呼ばれているものです．この「ＩＯ」というのは Input／Output の略です．

　一般に，1＝High(5Volt)，0＝Low(0Volt)として表現するのが普通で，この状態を「正論理」，逆の状態を「負論理」と呼んだりします．アナログ信号出力には，図のようにＤ／Ａ変換器を用います．（入力にはＡ／Ｄ変換器）

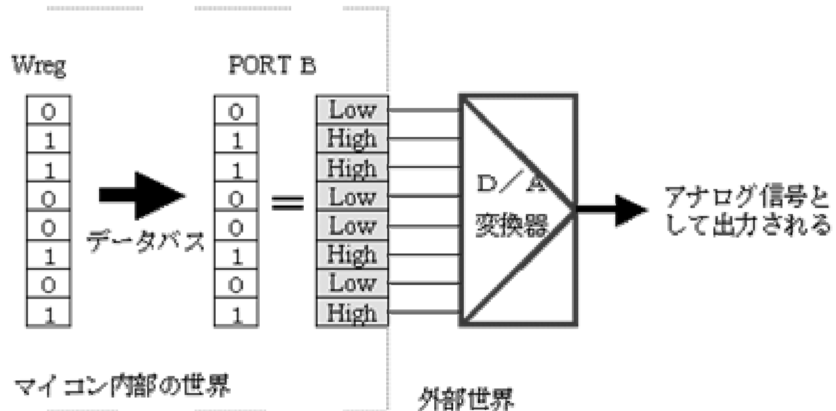


図１．２　出力ポートの一例としてのイメージ図

【　タイマー　】

　コンピュータのプログラム動作の中では，時刻，時間間隔，時間測定，などなど結構時間に関係するものが沢山あります．通常これらの「時間の素」を作るのがタイマーの役割です．多くは内部ハードウェアによってタイマーが作られていますが，外部の水晶発振子からのパルス入力も可能です．

　一般的に使われるタイマーの種類は下表の様なものです．

表１．３　マイコンで使われる主なタイマー

名　称	役　　割
リアルタイムクロック	実時間を作るタイマーでいわゆる時刻を刻む働きをします． 時刻としては年月日時分秒が一般的ですが，年はプログラムで追加する場合もあります．
インターバルタイマー	一定時間間隔を作るタイマーで，待ち時間やタイミングを調整するときに多く使われます． またＯＳ（オペレーティングシステム）の中では複数のプログラムの優先順位に従った走行制御のためにも使われています．

ウォッチドッグタイマ	プログラムの正常実行の監視用タイマーで，正常時にはタイマが一定時間間隔でクリアされ続けるようにしておき，プログラムの実行が異常になるとクリアが出来ずタイムアップして，コンピュータをリセットするようにするタイマーです.

【　シリアル通信　】

「シリアル通信」とは，一般にコンピュータ機器を接続する方法の一つで，送信側からは，０／１のデジタルデータを，１度に１ビットだけ送ることを順次一定時間間隔で繰り返して行き，受信側でそれを元に戻して例えば１バイトのデータ（８ビットから成る）として使うようにする方式をいいます.

これを図で表現すると下図のようになります.

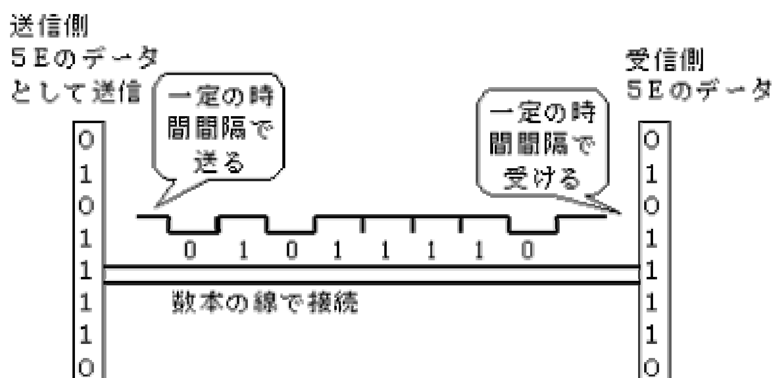


図１．３　シリアル通信のイメージ図

【 割り込みコントローラ　】

ワンチップマイコンと呼ばれるものには，周辺機器の機能も含まれています．これらの周辺機器からも割り込みが発生しますし，外部から割り込み信号を入力して，割り込み処理を取り扱うことができます．割り込み処理には，割り込みの禁止・許可，優先権の処理，ベクタの読み出しと分岐などが必要ですが，これらの処理を行う部分です.

【 ＤＭＡコントローラ　】

通常の情報の転送は，ＭＰＵ内部のレジスタを経由して行われますが，大量のデータを高速に転送したい場合には追いつかなくなりますし，プログラムが転送に明け暮れ，非常に効率が悪いことになります．レジスタを経由せずに，メモリからメモリ，あるいはメモリと周辺機器間で，直接転送する手法が DMA（Direct Memory Access）です．外部から DMA を行おうとする機器はバスの使用権を要求し，それが認知されると命令実行と次の命令実行の間に，ハードウェアの機能でデータを転送することができます．プログラムは初期の起動と，終了処理をするだけで済みますので，プログラムの負担も軽くなります.

【　リフレッシュコントローラ　】

ＤＲＡＭ（Dynamic RAM）のリフレッシュに使います．主なＲＡＭの中にはＳＲＡＭとＤＲＡＭがあります．ＤＲＡＭはアクセス時間は劣りますが，安価で長寿命のため，よく用いられています．ただし電気的なチャージ（これをリフレッシュという）を常に繰り返す必要があり，その処理を行う部分です.

２．Ｈ８マイコン

２．１　Ｈ８マイコンの概要

マイコンには，インテルの８０系，モトローラの６８系，ザイログのＺ８０が有名でしたが, それらの「良いとこ取り」プラス αをしたようなマイコンがＨ８です. 今では，少なくとも日本においては，最も使われているマイコンです.

そのＨ８にも多くの種類があり，８ビットマイコンの H8/300 や H8/300L から，データバスを 16 ビットに拡張した H8/300H や H8/500，更に高性能・低消費電力化した H8S シリーズがあります.

本講座では，その中でも最も使われている H8/300H を学習します.

また H8/300H にも種類があり，本講座では H8/3067 を選びました.

以下，日立のＨ８テクニカルブックからの抜粋です.

H8/3067

H8/3067 シリーズは，日立オリジナルアーキテクチャを採用した H8/300H CPU を核にして，システム構成に必要な周辺機能を集積したシングルチップマイクロコンピュータ (MCU) です.

H8/300H CPU は，内部 32 ビット構成で 16 ビット×16 本の汎用レジスタと高速動作を指向した簡潔で最適化された命令セットを備えており，16M バイトのリニアなアドレス空間を扱うことができます. また，H8/300CPU の命令に対しオブジェクトレベルで上位互換を保っていますので，H8/300 シリーズから容易に移行することができます.

システム構成に必要な周辺機能としては，ROM，RAM，16 ビットタイマ，8 ビットタイマ，プログラマブルタイミングパターンコントローラ (TPC)，ウォッチドッグタイマ (WDT)，シリアルコミュニケーションインタフェース (SCI)，A/D 変換器，D/A 変換器，I/O ポート，DMA コントローラ (DMAC) などを内蔵しています.

H8/3067 シリーズには，H8/3067，H8/3066，H8/3065 の 3 種類があります. H8/3067 には，128k バイト ROM と 4k バイト RAM，H8/3066 には，96k バイト ROM と 4k バイト RAM，H8/3065 には 64k バイト ROM と 2k バイト RAM がそれぞれ内蔵されています.

MCU 動作モードは，モード 1～7 (シングルチップモード 2 種類，拡張モード 5 種類) があり，データバス幅とアドレス空間を選択することができます.

H8/3067 シリーズには，マスク ROM 版のほかに，ユーザサイドで自由にプログラムの書換えができるフラッシュメモリを内蔵した F-ZTATTM*版があります. 仕様流動性の高い応用機器，さらに量産初期から本格的量産など，ユーザの状況に応じて迅速かつ柔軟な対応が可能です.

[注] * F-ZTATTM は (株) 日立製作所の商標です

（図２．１）　Ｈ８／３００Ｈ　内部ブロック図

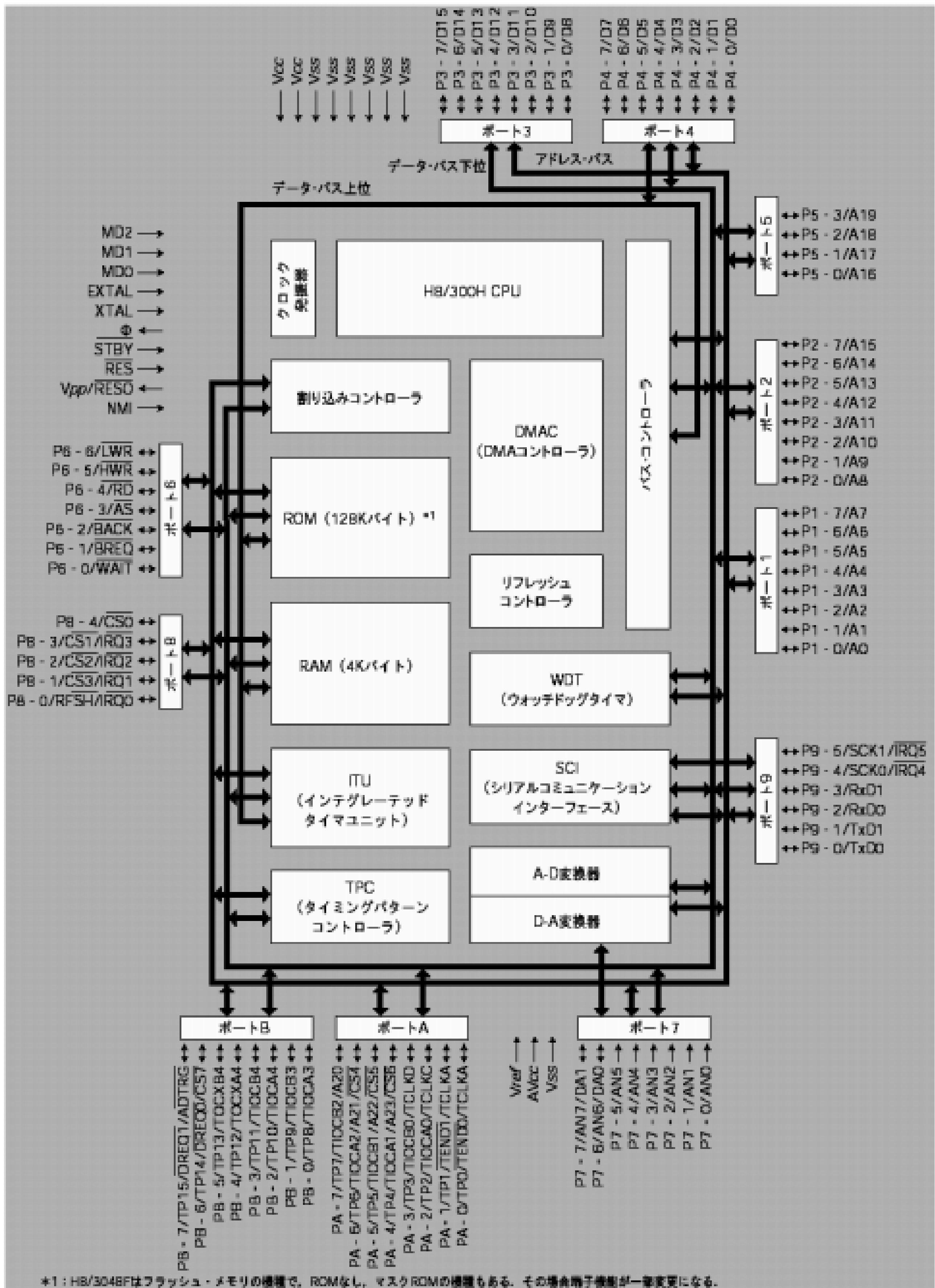


表2．1　H8／300Hマイコンの仕様

概　要

項　目	仕　様
CPU	H8/300CPU に対してオブジェクトレベルで上位互換 汎用レジスタマシン ・汎用レジスタ：16 ビット×16 本（8 ビット×16 本＋16 ビット×8 本，32 ビット×8 本としても使用可能） 高速動作 ・最大動作周波数：20MHz ・加減算：100ns ・乗除算：700ns アドレス空間 16M バイト 特長ある命令 ・8／16／32 ビット転送・演算命令 ・符号なし／符号付き乗算命令　（8 ビット×8 ビット，16 ビット×16 ビット） ・符号なし／符号付き除算命令　（16 ビット÷8 ビット，32 ビット÷16 ビット） ・ビットアキュムレータ機能 ・レジスタ間接指定によりビット番号を指定可能なビット操作命令
メモリ	H8/3067 ・ROM：128k バイト　（フラッシュメモリだから書き換えられる） ・RAM：4k バイト H8/3066 ・ROM：96k バイト ・RAM：4k バイト H8/3065 ・ROM：64k バイト ・RAM：2k バイト
割り込みコントローラ	外部割込み端子 7 本：NMI，IRQ_0～IRQ_5 内部割込み 36 要因 3 レベルの割込み優先順位が設定可能
バスコントローラ	アドレス空間を 8 エリアに分割し，エリアごとに独立してバス仕様を設定可能 エリア 0～7 に対してそれぞれチップセレクト出力可能 エリアごとに 8 ビットアクセス空間／16 ビットアクセス空間を設定可能 エリアごとに 2 ステートアクセス空間／3 ステートアクセス空間を設定可能 2 種類のウェイトモードを設定可能 エリアごとにプログラムウェイトのステート数を設定可能

	バーストROMを直接接続可能 最大8MバイトのDRAMを直接接続可能（またはインターバルタイマとして使用可能） バス権調停機能
DMAコントローラ（DMAC）	ショートアドレスモード ・最大4チャネルを使用可能 ・I/Oモード／アイドルモード／リピートモードの選択可能 ・起動要因：16ビットタイマチャネル0～2のコンペアマッチ／インプットキャプチャA割込み，A/D変換器の変換終了割込み，SCIの送信データエンプティ／受信データフル割込み，外部リクエスト フルアドレスモード ・最大2チャネルを使用可能 ・ノーマルモード／ブロック転送モードの選択可能 ・起動要因：16ビットタイマチャネル0～2のコンペアマッチ／インプットキャプチャA割込み，A/D変換器の変換終了割込み，外部リクエスト，オートリクエスト
16ビットタイマ×3チャネル	16ビットタイマ3チャネルを内蔵．最大6端子のパルス出力，または最大6種類のパルスの入力処理が可能 16ビットタイマカウンタ×1（チャネル0～2） アウトプットコンペア出力／インプットキャプチャ入力（兼用端子）×2（チャネル0～2） 同期動作可能（チャネル0～2） PWMモード設定可能（チャネル0～2） 位相計数モード設定可能（チャネル2） コンペアマッチ／インプットキャプチャAの割込みによりDMAC起動可能（チャネル0～2）
8ビットタイマ×4チャネル	8ビットアップカウンタ（外部イベントカウント可能） タイムコンスタントレジスタ×2 2チャネルの接続が可能能
プログラマブルタイミングパターンコントローラ（TPC）	16ビットタイマをタイムベースとした最大16ビットのパルス出力が可能能 最大4ビット×4系統のパルス出力が可能（16ビット×1系統，8ビット×2系統などの設定も可能） ノンオーバラップモード設定可能 DMACによる出力データの転送可能
ウォッチドッグタイマ（WDT）×1チャネル	オーバフローによりリセット信号を発生可能 リセット信号の外部出力可能（ただしF-ZTAT版は不可） インターバルタイマとして使用可能

<table>
<tr><td>シリアルコミュニケーションインタフェース（SCI）×3 チャネル</td><td>調歩同期／クロック同期式モードの選択可能
送受信同時動作（全二重動作）可能
専用のボーレートジェネレータ内蔵
スマートカードインタフェース拡張機能内蔵</td></tr>
<tr><td>A/D 変換器</td><td>分解能：10 ビット
8 チャネル：単一モード／スキャンモード選択可能
アナログ変換電圧範囲の設定が可能
サンプル&ホールド機能付
外部トリガまたは 8 ビットタイマのコンペアマッチによる A/D 変換開始可能
A/D 変換終了割込みによる DMAC 起動可能</td></tr>
<tr><td>D/A 変換器</td><td>分解能：8 ビット
2 チャネル
ソフトウェアスタンバイモード時 D/A 出力保持可能</td></tr>
<tr><td>I/O ポート</td><td>入出力端子 70 本
入力端子 9 本</td></tr>
<tr><td>動作モード</td><td>7 種類の MCU 動作モード
<table>
<tr><th>モード</th><th>アドレス空間</th><th>アドレス端子</th><th>バス幅初期値</th><th>バス幅最大値</th></tr>
<tr><td>モード 1</td><td>1M バイト</td><td>A_{19}〜A_0</td><td>8 ビット</td><td>16 ビット</td></tr>
<tr><td>モード 2</td><td>1M バイト</td><td>A_{19}〜A_0</td><td>16 ビット</td><td>16 ビット</td></tr>
<tr><td>モード 3</td><td>16M バイト</td><td>A_{23}〜A_0</td><td>8 ビット</td><td>16 ビット</td></tr>
<tr><td>モード 4</td><td>16M バイト</td><td>A_{23}〜A_0</td><td>16 ビット</td><td>16 ビット</td></tr>
<tr><td>モード 5</td><td>16M バイト</td><td>A_{23}〜A_0</td><td>8 ビット</td><td>16 ビット</td></tr>
<tr><td>モード 6</td><td>64K バイト</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>モード 7</td><td>1M バイト</td><td>-</td><td>-</td><td>-</td></tr>
</table>
・モード 1〜4 では内蔵 ROM は無効となります．</td></tr>
<tr><td>低消費電力状態</td><td>スリープモード
ソフトウェアスタンバイモード
ハードウェアスタンバイモード
モジュール別スタンバイ機能あり
システムクロック分周比可変</td></tr>
<tr><td>その他</td><td>クロック発振器内蔵</td></tr>
</table>

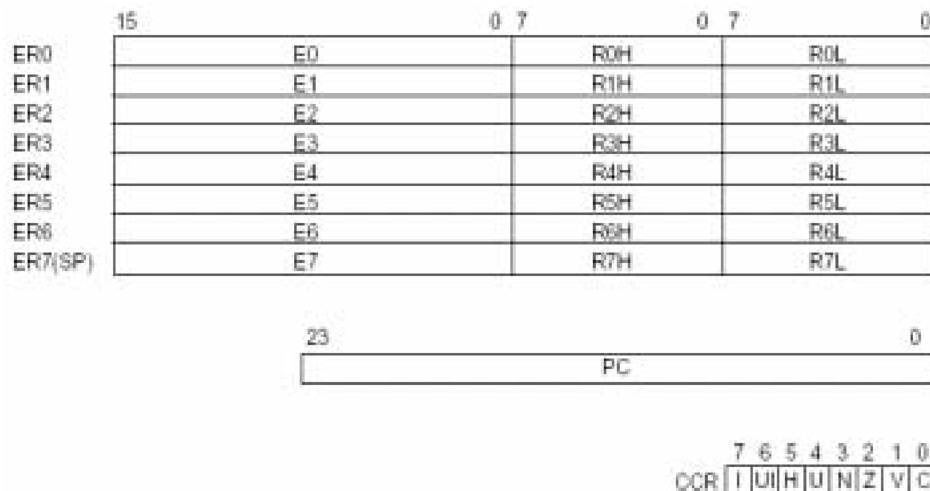
２．２　H8/300H マイコンのプログラミング

詳細は別途配布（貸与）の「H8/300H シリーズ プログラミングマニュアル」を参照して下さい．また下記のＵＲＬからもダウンロードできます：

http://www.hitachisemiconductor.com/sic/resource/japan/jpn/pdf/mpumcu/j602071_h8300hpm.pdf

２．２．１　H8/300H レジスタ構成

汎用レジスタ（Rn）と拡張レジスタ（En）



コントロールレジスタ（CR）

《記号説明》

SP　：スタックポインタ
PC　：プログラムカウンタ
CCR ：コンディションコードレジスタ（フラグレジスタともいう）
I　：割り込みマスクビット
UI　：ユーザビット／割り込みマスクビット
H　：ハーフキャリフラグ
U　：ユーザビット
N　：ネガティブフラグ ●重要
Z　：ゼロフラグ　●最重要
V　：オーバフローフラグ
C　：キャリフラグ

図２．２　Ｈ８／３００Ｈのレジスタ

■汎用レジスタ

汎用レジスタは，**ER0～ER6** の 32 ビット×７本です．

これらは上下 16 ビットづつに分割して，E0～E6，R0～R6 の 14 本の独立した 16 ビットレジスタとして使うこともできます．

更に R0～R6 の７本の 16 ビットレジスタは，上位バイトと下位バイトに分けて R0H～R6H, R0L～R6L という 14 本の独立した８ビットレジスタとして使うこともできます．

その辺の関係を ER0 を例にとって表にすると，以下のようになります．

表２．２　Ｈ８／３００Ｈ汎用レジスタの概念

32ビットレジスタ (L)	ER0		
16ビットレジスタ (W)	E0 (上位)	R0 (下位)	
8ビットレジスタ (B)		R0H	R0L

基本的にすべての汎用レジスタは対等です．

使用できるレジスタが固定されているのは EEPMOV 命令(LDIR 相当)だけです．

汎用レジスタ **ER7** には，汎用レジスタの機能に加えて**スタックポインタ（SP）**としての機能が割り当てられており，例外処理やサブルーチンコールなどで暗黙的に使用されます．

■プログラムカウンタ（PC）

PC は 24 ビットのカウンタで, CPU が次に実行する命令のアドレスを指します．CPU の命令はすべて偶数番地から始まる 2 バイト（ワード）を単位としているため，PC の最下位ビットは命令コードを読み出す時は 0 とみなされます．PC はリセット例外処理の過程で生成されるベクタアドレスによってスタートアドレスをロードすることにより初期化されます．

■コンディションコードレジスタ（CCR）

ビット 7：（I）割り込みマスクビット

このビットが 1 にセットされると, 割り込み要求がマスクされます. ただし，NMI は I ビットに関係なく受け付けられます．I ビットは例外処理の実行が開始されたときに 1 にセットされます.

ビット 6：（UＩ）ユーザビット

ソフトウェア（LDC，STC，ANDC，ORC，XORC 命令）でリード／ライトできます．割り込みマスクビットとしても使用可能.

ビット 5：（H）ハーフキャリフラグ

ADD.B，ADDX.B，SUB.B，SUBX.B，CMP.B，NEG.B 命令の実行により，ビット 3 にキャリまたはボローが生じたとき 1 にセットされ，生じなかったとき 0 にクリアされます．ADD.W，SUB.W，CMP.W，NEG.W 命令の実行によりビット 11 にキャリまたはボローが生じたとき，または ADD.L，SUB.L，CMP.L，NEG.L 命令の実行によりビット 27 にキャリまたはボローが生じたとき 1 にセットされ，生じなかったとき 0 にクリアされます.

ビット 4：（U）ユーザビット

ソフトウェア（LDC，STC，ANDC，ORC，XORC 命令）でリード／ライトできます.

ビット 3：（N）ネガティブフラグ

データの最上位ビットを符号ビットとみなし，最上位ビットの値を格納します.

ビット 2：（Z）ゼロフラグ

データがゼロのとき 1 にセットされ，ゼロ以外のとき 0 にクリアされます.

ビット 1：（V）オーバフローフラグ

算術演算命令の実行によりオーバフローが生じたとき 1 にセットされます．それ以外のとき 0 にクリアされます.

ビット 0：（C）キャリフラグ

演算の実行により，キャリが生じたとき 1 にセットされ，生じなかったとき 0 にクリアされます．キャリには次の種類があります.

・加算結果のキャリ
・減算結果のボロー
・シフト／ローテートのキャリ

また，キャリフラグにはビットアキュムレータ機能があり，ビット操作命令で使用されます.

２．２．２　H8/300H データ構成

H8/300H CPU は，1 ビット，4 ビット BCD，8 ビット（バイト），16 ビット（ワード），および 32 ビット（ロングワード）のデータを扱うことができます．

1 ビットデータはビット操作命令で扱われ，オペランドデータ（バイト）の第 n ビット（n＝0，1，2，……，7）という形式でアクセスされます．

10 進補正命令 DAA，DAS ではバイトデータは 2 桁の 4 ビット BCD データとなります．

H8/300H CPU は，メモリ上のワードデータ／ロングワードデータをアクセスすることができます．これらは，偶数番地から始まるデータに限定されます．奇数番地から始まるワードデータ／ロングワードデータをアクセスした場合，アドレスの最下位ビットは 0 とみなされ，1 番地前から始まるデータをアクセスします．この場合，アドレスエラーは発生しません．命令コードについても同様です．

ER7（SP）をアドレスレジスタとしてスタック領域をアクセスするときは，必ずワードサイズまたはロングワードサイズでアクセスしてください．

メモリ上のデータ形式を次の図に示します．

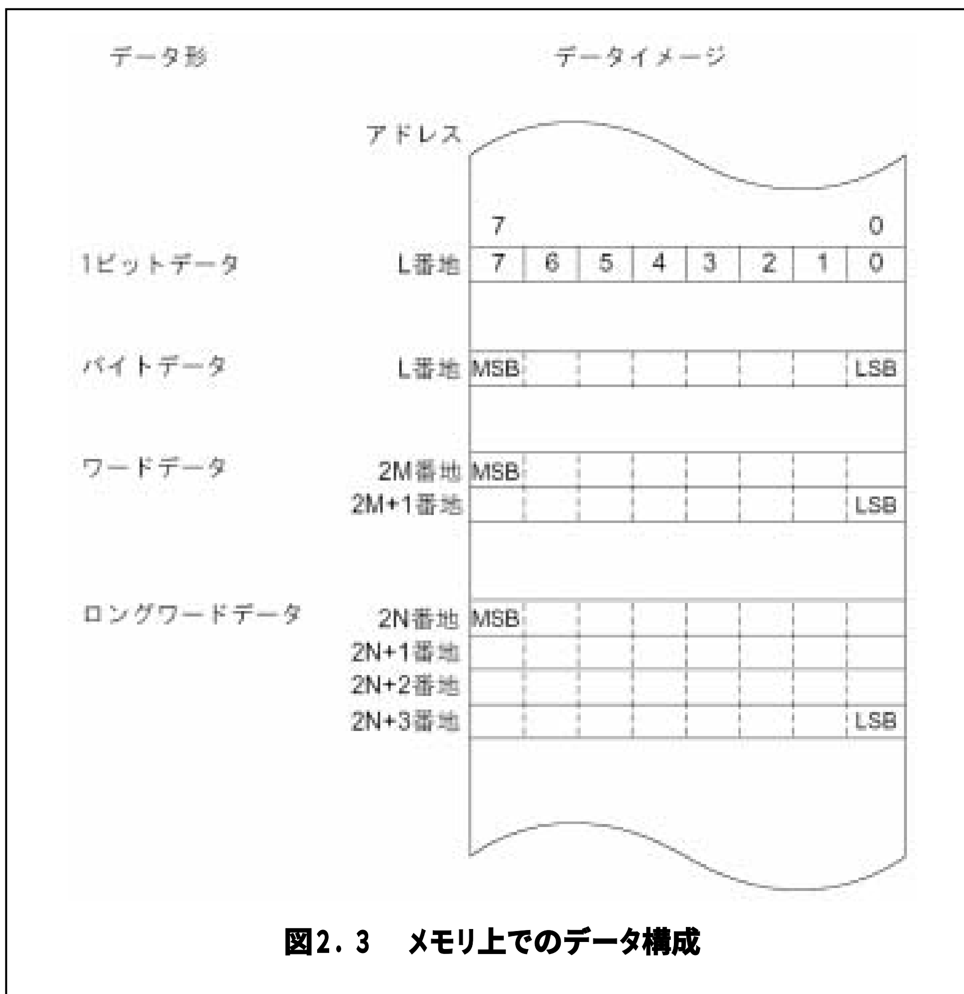


図2．3　メモリ上でのデータ構成

２．２．３　H8/300H 命令構成

機　能	命　令	種　類
データ転送命令	MOV、PUSH*1、POP*1、MOVTPE、MOVFPE	3
算術演算命令	ADD、SUB、ADDX、SUBX、INC、DEC、ADDS、SUBS、DAA、DAS、MULXU、MULXS、DIVXU、DIVXS、CMP、NEG、EXTS、EXTU	18
論理演算命令	AND、OR、XOR、NOT	4
シフト命令	SHAL、SHAR、SHLL、SHLR、ROTL、ROTR、ROTXL、ROTXR	8
ビット操作命令	BSET、BCLR、BNOT、BTST、BAND、BIAND、BOR、BIOR、BXOR、BIXOR、BLD、BILD、BST、BIST	14
分岐命令	Bcc*2、JMP、BSR、JSR、RTS	5
システム制御命令	TRAPA、RTE、SLEEP、LDC、STC、ANDC、ORC、XORC、NOP	9
ブロック転送命令	EEPMOV	1

合計 62 種類

【注】 ▒ ：H8/300H CPU で追加された命令

*1　POP.W Rn、PUSH.W Rn は、それぞれ MOV.W @SP+,Rn、 MOV.W Rn,@-SP と同一です。
また、POP.L ERn、 PUSH.L ERn は、それぞれ MOV.L @SP+,Rn、 MOV.L Rn,@-SP と同一です。

*2　Bcc は条件分岐命令の総称です。

表２．３　Ｈ８／３００Ｈの命令の分類

■ 命令の基本フォーマット

H8/300H CPU の命令は 2 バイト（ワード）を単位としています．各命令はオペレーションフィールド（OP），レジスタフィールド（r），EA 拡張部（EA）およびコンディションフィールド（cc）から構成されています．

（ 1） オペレーションフィールド

命令の機能を表し，アドレッシングモードの指定，オペランドの処理内容を指定します．命令の先頭 4 ビットを必ず含みます．2 つのオペレーションフィールドを持つ場合もあります．

（ 2） レジスタフィールド

汎用レジスタを指定します．アドレスレジスタのとき 3 ビット，データレジスタのとき 3 ビットまたは 4 ビットです．2 つのレジスタフィールドを持つ場合やレジスタフィールドを持たない場合もあります．

（ 3） EA 拡張部

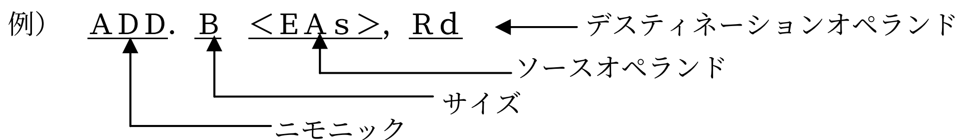
イミディエイトデータ，絶対アドレスまたはディスプレースメントを指定します．8 ビット，16 ビット，32 ビットです．24 ビットアドレスおよびディスプレースメントは上位 8 ビットをすべて 0（H'00）とした 32 ビットデータとして扱われます．

（ 4） コンディションフィールド

条件分岐命令の分岐条件を指定します．

表２．４　命令のフォーマット別分類

オペレーションフィールドのみ	NOP，RTS など
オペレーションフィールドとレジスタフィールド	ADD.B　Rn，Rm など
オペレーションフィールド，レジスタフィールドおよび EA 拡張部	MOV.B　@(d:16, Rn)，Rm
オペレーションフィールド，EA 拡張部およびコンディションフィールド	BRA d:8



■アドレッシングモード

H8/300H CPU は、以下(1)～(8)の 8 種類のアドレッシングモードをサポートしています。命令ごとに使用できるアドレッシングモードが異なります。

演算命令ではレジスタ直接、およびイミディエイトを使用できます。

転送命令ではプログラムカウンタ相対とメモリ間接を除くすべてのアドレッシングモードを使用できます。

ビット操作命令では、オペランドの指定にレジスタ直接、レジスタ間接、および絶対アドレス（@aa:8）を使用できます。さらに、オペランド中のビット番号の指定にレジスタ直接（BSET、BCLR、BNOT、BTST の各命令）、およびイミディエイト（3 ビット）を使用できます。

（ 1） レジスタ直接　Rn

命令コードのレジスタフィールドで指定されるレジスタ（8 ビット，16 ビットまたは 32 ビット）がオペランドとなります.

8 ビットレジスタとしては R0H～R7H，R0L～R7L を指定可能です.

16 ビットレジスタとしては R0～R7，E0～E7 を指定可能です.

32 ビットレジスタとしては ER0～ER7 を指定可能です.

（ 2） レジスタ間接　@ERn

命令コードのレジスタフィールドで指定されるアドレスレジスタ（ERn）の下位 24 ビットをアドレスとしてメモリ上のオペランドを指定します.

（ 3） ディスプレースメント付きレジスタ間接　@（d:16, ERn）／@（d:24, ERn）

命令コードのレジスタフィールドで指定されるアドレスレジスタ（ERn）の内容に，命令コード中に含まれる 16 ビットディスプレースメントまたは 24 ビットディスプレースメントを加算した内容の下位 24 ビットをアドレスとしてメモリ上のオペランドを指定します. 加算に際して，16 ビットディスプレースメントは符号拡張されます.

（ 4） ポストインクリメントレジスタ間接　@ERn＋
／ プリデクリメントレジスタ間接　@－ERn

・ポストインクリメントレジスタ間接　@ERn＋

命令コードのレジスタフィールドで指定されるアドレスレジスタ（ERn）の内容の下位 24 ビットをアドレスとしてメモリ上のオペランドを指定します.
その後，アドレスレジスタの内容（32 ビット）に 1，2 または 4 が加算され，加算結果がアドレスレジスタに格納されます. バイトサイズでは 1，ワードサイズでは 2，ロングワードサイズでは 4 がそれぞれ加算されます. ワードサイズ，ロングワードサイズのときはレジスタの内容が偶数となるようにしてください.

・プリデクリメントレジスタ間接　@－ERn

命令コードのレジスタフィールドで指定されるアドレスレジスタ（ERn）の内容から 1，2 または 4 を減算した内容の下位 24 ビットをアドレスとして，メモリ上のオペランドを指定します. その後，減算結果がアドレスレジスタに格納されます. バイトサイズでは 1，ワードサイズでは 2，ロングワードサイズでは 4 がそれぞれ減算されます. ワードサイズ，ロングワードサイズのときはアドレスレジスタの内容が偶数となるようにしてください.

（ 5） 絶対アドレス　@aa:8／@aa:16／@aa:24

命令コード中に含まれる絶対アドレスでメモリ上のオペランドを指定します．絶対アドレスは 8 ビット（@aa:8），16 ビット（@aa:16），または 24 ビット（@aa:24）です．　8 ビット絶対アドレスの場合，上位 16 ビットはすべて 1（H'FFFF）となります．

16 ビット絶対アドレスの場合，上位 8 ビットは符号拡張されます．

24 ビット絶対アドレスの場合，全アドレス空間をアクセスできます．

（ 6） イミディエイト　#xx:8／#xx:16／#xx:32

命令コードの中に含まれる 8 ビット（#xx:8），16 ビット（#xx:16），または 32 ビット（#xx:32）のデータを直接オペランドとして使用します．なお，ADDS, SUBS, INC，DEC 命令ではイミディエイトデータが命令コード中に暗黙的に含まれます．ビット操作命令では，ビット番号を指定するための 3 ビットのイミディエイトデータが命令コード中に含まれる場合があります．また，TRAPA 命令ではベクタアドレスを指定するための 2 ビットのイミディエイトデータが命令コード中に含まれます．

（ 7） プログラムカウンタ相対　@（d:8, PC）／@（d:16, PC）

条件分岐命令，BSR 命令で使用されます．PC の内容で指定される 24 ビットのアドレスに命令コード中に含まれる 8 ビット，または 16 ビットディスプレースメントを加算して，24 ビットの分岐アドレスを生成します．加算に際して，ディスプレースメントは 24 ビットに符号拡張されます．また加算される PC の内容は次の命令の先頭アドレスとなっていますので，分岐可能範囲は分岐命令に対して－126～＋128 バイト（－63～＋64 ワード）または－32766～＋32768 バイト（－16383～＋16384 ワード）です．このとき，加算結果が偶数となるようにしてください．

（ 8） メモリ間接　@@aa:8

JMP，JSR 命令で使用されます．命令コードの中に含まれる 8 ビット絶対アドレスでメモリ上のオペランドを指定し，この内容を分岐アドレスとして分岐します．メモリ上のオペランドはロングワードサイズで指定します．このうち先頭 1 バイトは無視され，24 ビット長の分岐アドレスを生成します．図 2.8 にメモリ間接による分岐アドレスの指定方法を示します．

絶対アドレスの上位ビットはすべて 0 となります．このため分岐アドレスを格納できるのは 0～255（H'0000～H'00FF）番地です．ただし，このうちの先頭領域は例外処理ベクタ領域と共通になっているので注意してください．

■命令とアドレッシングモードの組み合わせ

「H8/300H シリーズ　プログラミングマニュアル」P12 を参照

■命令の機能別一覧

「H8/300H シリーズ　プログラミングマニュアル」pp.13-17 を参照.

■実行アドレスの計算方法

「H8/300H シリーズ　プログラミングマニュアル」pp.21-24 を参照

■各命令の説明

「H8/300H シリーズ　プログラミングマニュアル」pp.25-163 を参照

■命令セット一覧

「H8/300H シリーズ　プログラミングマニュアル」pp.165-176 を参照

■参考　このページでは、H8 と Z80 の両アセンブラを比較してみます.

アドレッシングモード	H8	Z80
イミディエイト	MOV.B　#H'12,R0L	LD　A,012H
レジスタ直接	MOV.B　R1H,R0L	LD　A,B
絶対アドレス	MOV.B　@H'001234,R0L	LD　A,(01234H)
レジスタ間接	MOV.B　@ER3,R0L	LD　A,(HL)
ディスプレースメント(オフセット)付きレジスタ間接	MOV.B　@(10,ER3),R0L	LD　A,(IX+10)
プログラムカウンタ相対	BRA　label	JR　label

H8 特有のアドレッシングモードは以下の通り.

アドレッシングモード	H8	Z80
ポストインクリメントレジスタ間接 （転送方向「メモリ→レジスタ」のみ）	MOV.B　@ER3+,R0L	LD　A,(HL) INC　HL
プリデクリメントレジスタ間接 （転送方向「レジスタ→メモリ」のみ）	MOV.B　R0L,@-ER3	DEC　HL LD　(HL),A

「メモリ間接」は，ジャンプやコール（JSR）のみで使います.

その他の注意すべき相違点：

・**メモリーレジスタ間の演算命令はない**

Z80 と違い ADD　A，(HL) のように出来ないので，以下のようにする：

```
MOV.B　@ER3，R0H
ADD.B　R0H，R0L
```

・**転送しただけでゼロフラグ，符号フラグが変わる**

例えば　MOV.B　@H'123456, R0L を実行すると 123456 番地の内容が R0L に転送されるが，同時にその値によってゼロフラグ及び符号フラグが変化します.

・**交換命令がない**

H8 には Z80 の EX 命令のような交換命令がない．例えば EX　DE，HL なんてのは，ＤＥがはじめからＨＬと同じ機能を持っていれば必要ない.

・**条件コール，条件リターンがない**

H8 では条件分岐でコール（ＪＳＲ，ＢＳＲ）やリターン（ＲＴＳ）を用いる.

・**EEPMOV について**

H8 のこの命令の実行中は割り込みを受け付けない.（EEPMOV.W の場合は NMI のみ受け付ける）

・**ポインタのインクリメント・デクリメントにはＡＤＤＳ／ＳＵＢＳを使う**

ポインタを４バイト進めるのに,

```
INC.L　#2,ER3　　　; ER3 に 2 を足す
INC.L　#2,ER3　　　; ER3 に 2 を足す
```

という方法もあるが,

```
ADDS　#4,ER3
```

とやればシンプルだ.

但しＩＮＣ／ＤＥＣはフラグに影響を与えるが，ＡＤＤＳ／ＳＵＢＳはフラグに全く影響を与えない．したがってループのカウントなどにはＳＵＢＳではなくＤＥＣを使わないと，無限ループになってしまうので気を付けよう.

３．プログラミング実習

３．１　Yellow ソフト

YellowIDE（C コンパイラとアセンブラ）と **YellowScope**（デバッガ）のセットアップを行う．

■エクスプローラ等により次の作業を必ず行う．
作業フォルダ Ｚ￥Ｈ８ を作成する．
作業フォルダに次の３つのファイルを配布します．
cc0d.asm（元々、C:¥Program Files ¥YH8¥LIB¥SRC¥ にある）
LH8S.LIB と LH8E1.LIB（元々、C:¥Program Files¥YH8¥LIB¥ にある）

■ソフトを立ち上げる　［プログラム］ → ［YellowIDE5］ → ［YIDE］

■デバッガとの統合環境を作る（最初に一度だけ行う）
（［参照］ボタンがある時は必ずそれを利用すること．以下同様）
［ツール］ → ［登録］ の 【ツール１】タブ において，
【有効】にチェック［レ］する
アプリケーション名：YellowScope
プログラムファイル：C:¥Program Files¥YellowScope¥BIN¥YScope.exe
コマンドライン：　$3.YEX（デバッグ用ファイル）
【スピードボタンを有効】にチェック［レ］して
次のファイルを選択：C:¥Program Files ¥YellowIDE¥YSCOPE.BMP

■コンパイラ／アセンブラの所在を設定する（最初に一度だけ行う）
［設定］ → ［メイク環境］
【Ｈ８】タブで　YCH8／YAH8 インストールディレクトリ
［参照］ボタンにより　C:¥Program Files¥YH8

■プロジェクトの作成
［ファイル］ → ［プロジェクトの新規作成］ → 作業フォルダ内にプロジェクト名を入力する → プロジェクトウインドウが開く
（既に作成した次回からは，　［ファイル］ → ［プロジェクトを開く］ ）

■作成したプロジェクトに各種設定を行う
［プロジェクト］ → ［設定］
（かつて作成した他のプロジェクトから引用・参照したい場合は：
［他のプロジェクトからコピー］ボタンを押して，
ダイアログウインドウからファイル名を選ぶ）
・【ターゲット】タブ
ＣＰＵの種類：　Ｈ８／３００Ｈ
オブジェクトの形式：　イエロースコープでシミュレーションデバッグ
（実機の場合は：　イエロースコープでリモートデバッグ）
開始アドレス：　２０００００Ｈ（実習ボードに合わせてある）
その他：　［レ］ スタートアップルーチンはデバッグしない
［レ］ Ｃソースもアセンブラレベルでデバッグする

・【スタートアップ】タブ

シミュレーションデバッグする場合（参照を押して指定）

スタートアップルーチン：　作業フォルダ内の　cc0d.asm

標準ライブラリ：　作業フォルダ内の　LH8S.LIB

実機ボードで、リモートデバッグする場合（参照を押して指定）

スタートアップルーチン：　作業フォルダ内の　cc0d.asm

標準ライブラリ：　作業フォルダ内の　LH8E1.LIB

・【セグメント定義】タブ

ＲＡＭの先頭番地：200000　を入力し → 自動生成 で自動生成され,

```
;segment        start        end        rom
main            $0                      ,$0
TEXT
DATA_CONST
DATA            $200000
HEAP
(  STACK        $FFFB20
```
←←この行は、今回は不要）

（注意：　最終的にはマップリストを見て確認すること）

■プログラムエディタを開く　ファイル → 新規作成 →（エディタが開く）

■Ｃプログラムを書く

ここでは例として次のように入力する：

```
#include <stdio.h>  /* スペルを間違わないよう注意深く入力せよ */
main()
{
        printf( “Hello World !” );
}
```

ファイル → 名前を付けて保存　例：　**Hello.c**　（**.c** まで入力する）

■プロジェクトに追加する

プロジェクト → ファイルの追加 → ソース **Hello.c** を指定

■コンパイル＆リンクする

プロジェクト → メイク（▶ ボタン、またはその左隣のボタンも可）

警告０　エラー０　を確認して OK ボタンを押す.

警告・エラーが出た場合は，下段に英語でエラーの種類などが表示され,

そこをダブルクリックすれば，ソースのエラー箇所が反転表示される.

■デバッガを立ち上げる（ YS ボタンも可）

ツール → YellowScope　（YS ボタンも可）→ デバッグ用ウインドウが開く

最初は「セグメントが設定されていない」等の警告がでるので、次を設定する.

設定 → 環境設定 → セグメントタブ（RAM／ROM／入出力を選び追加）

ROM：　開始アドレス 00000000　～　終了アドレス 0001FFFF

RAM：　開始アドレス 000FF000　～　終了アドレス 000FFFFF

RAM：　開始アドレス 00200000　～　終了アドレス 0021FFFF

I/O：　開始アドレス 00F00000　～　終了アドレス 00FFFFFF

■デバッグする

YScope ウィンドウの ｿｰｽﾌｧｲﾙ にあるデバッグしたいファイルをＷクリック
→ 対象のファイルが開く

■ログウインドウを表示させる

[表示] → [ログ]

■プログラムを実行させる

[デバッグ] → [実行] → ログに ”Hello World !” と実行結果が表示される．

■各種の実行方法（本実験では主に[Ｆ11]と[Ｆ５]を使用する）

[Ｆ５]：　一気にすべてを実行
[Ｆ９]：　ＢＰ（ブレイクポイント）の作成／削除
[Ｆ10]：　ステップ実行（サブルーチンには飛ばず 1 行づつ実行）
[Ｆ11]：　トレース実行（サブルーチンまで飛んで 1 行づつ実行）

■その他[表示]メニューから開く主なウインドウ

[レジスター]：　ステップ実行などをしながらレジスタの変化を見る．
[ウォッチー]：　指定した変数の変化などを見る．
[メモリー]：　16M バイトのメモリー空間を見る．

■**YellowSoft** のホームページ（詳しいマニュアル等をダウンロード可）

http://www.yellowsoft.com/index.htm

３．２　シミュレーション実習

重要！
C ソースは：
name.c
アセンブラは：
name.asm

■新たにファイルを作って，今度はアセンブラで書いてみよう．

プロジェクトウインドウ内の Hello.c をクッリクして、
[プロジェクト] → [ファイルの削除]
[ファイル] → [新規作成] →（エディタが開く）

［練習１］（単純なレジスタへの即値設定とレジスタ間データ転送）

```
segment TEXT  ATR_CODE
public  _main
_main:
        MOV.W   #30,R1   ; R1 に 30（十進）を代入
        MOV.W   R1,R2    ; R2 に値をコピー
        RTS
end
```

（segment / public / _main: ：今後テキストでは表示しないが、常に必要）
（RTS / end：今後表示しないが必要）

; 最後に必ず改行を入れないとエラーとなる

[ファイル] → [名前を付けて保存]　例：　movreg.asm（.asm まで入力する）
[プロジェクト] → [ファイルの追加] → ソース movreg.asm を指定
[プロジェクト] → [メイク]
[ツール] → [YellowScope] →（デバッグ用ウインドウが開く）

表示 → レジスター（ここが先程と違うので注意!!）
F11： トレース実行でレジスタの変化を見よ.

Yellow Scope 表示の中から必要なウインドウ（今回はレジスタ）を開いた状態で，F11 を用いて１行ずつトレース実行して，変化を確認せよ．（以下同様）

［練習２］（レジスタの値をメモリーの特定番地に転送）
R0L に H’41 を代入した後，0x210000 番地に転送せよ.
（segment… や …end などは省略したので［練習 1］を参考にせよ．以下同様）

```
MOV.B   #H’41, R0L        ; 0x41 は“A”の ASCII コード
MOV.B   R0L,@(H'210000)   ; デバッガの表示→メモリーで確認せよ
```

［練習 3］（メモリーの内容を他の番地に転送）
練習２の続きで、0x210000 番地のメモリの内容を，0x210010 番地に転送せよ.

```
MOV.B   @(H'210000), R0L        ; @( )はメモリの内容を指す
MOV.B   R0L,@(H'210010)         ; R0L の L に注目せよ
```

（デバッガ表示→メモリーで予めデータを入力して実行せよ．以下同様）

［練習４］（各種アドレッシングモード：ＡＭ）
Ｐ１５～Ｐ１６の内容を具体的にシミュレーションします.

```
_main:  ; 関数名（具体的なアドレスを指している）
        MOV.W   #30, R1
        MOV.W   R1, R2
                ; 以上は練習１と同じ．以下はＰ１７参照
label:  ; ラベル名（具体的なアドレスを指している）
        MOV.B   #H'12, R0L      ;イミディエイト（即値）ＡＭ
        MOV.B   #H'34, R1H
        MOV.B   R1H, R0L        ;レジスタ直接ＡＭ
        MOV.B   @H'200000, R0L  ;絶対アドレスＡＭ
        MOV.L   #H'200008, ER3
        MOV.B   @ER3, R0L       ;レジスタ間接ＡＭ
        MOV.B   @(3,ER3),R0L    ;オフセット付レジスタ間接ＡＭ
        BRA     label           ;プログラムカウンタ相対ＡＭ
```

■■［問題１］（レジスタ間の値の交換）
R0L に H’AA，R0H に H’BB を代入した後，両レジスタ の値を入れ換えよ.

［練習５］（ブロックデータ転送）
デバッガにより 0x210000～0x21000F 番地にデータを直接書き込んだ後、
0x210000 番地から 16 バイトを，0x210010 番地以降に転送せよ.

```
MOV.B   #16, R4L          ; 転送データ数のセット
MOV.L   #H'210000, ER5    ; 転送元の番地をセット
MOV.L   #H'210010, ER6    ; 転送先の番地をセット
EEPMOV                    ; ブロック転送命令
```

（EEPMOV 命令を，配布する PDF ファイルのマニュアルで確認せよ）

［練習６］（スタックポインタ SP＝ER7 と，PUSH／POP 命令の働き）

0x210010→ER7，0x1234→R1，0xABCD→R2 を設定し，PUSH／POP 命令によって R1 と R2 の値を交換するプログラムを作れ．

また全体の動きを レジスタ と メモリー の各ウインドウから説明せよ．

```
MOV.L   #H'210010, ER7   ; メモリーでは 0x210000 から表示せよ
MOV.W   #H'1234, R0      ; 1234 は見易い値というだけの意味
MOV.W   #H'ABCD, R1      ;         同　上
PUSH    R0       ; SP の示す番地にデータ退避後、SP 減算に注目
PUSH    R1       ;       同　上
POP     R0       ; 通常は POP の順序が逆だが，SP/PUSH/POP 動作
POP     R1       ; を理解してもらう便宜上，値交換役に利用した
```

［練習７］（Ｃ言語における変数のアセンブル結果）

下記Ｃプログラムを，アセンブラレベルで詳細に対比せよ．

```
int  global;      // グローバル変数の宣言
main()
{
        int  local;       // ローカル変数の宣言
        global= 0xAAAA;   // グローバル変数に値を代入
        local= 0xBBBB;    // ローカル変数に値を代入
}
```

余裕のある人は
static int slocal;
もやってみよう

上記Ｃプログラムのアセンブル結果：

```
                segment  TEXT ATR_CODE
                public   _main
_main:
; ローカル変数 local の所在地 ＝ -2(er6)
        PUSH.L  ER6              ; ベースポインタを退避（Ｃ特有）
        MOV.L   ER7,ER6          ; 新たなベースポインタ
        SUBS    #2,ER7           ; 変数 global のワードサイズ２を減算
; グローバル変数 global = 0xAAAA を代入
        MOV.W   #H'AAAA,R0       ; int は PC ではワードサイズ
        MOV.W   R0,@_global      ; 代入前の“初期値”に注目せよ
; ローカル変数 local = 0xBBBB を代入
        MOV.W   #H'BBBB,R1       ; ローカル変数の扱いと“初期値”＆
        MOV.W   R1,@(-2,ER6)     ; ER6 とのオフセット表現に注目せよ
_main_end:               ; 以下，Ｃコンパイラ特有の約束事
        MOV.L   ER6,ER7  ; ベースポインタをスタックポインタに復帰
        POP.L   ER6      ; 退避したベースポインタを復帰
        RTS              ; ReTurn Subroutine
                public _global
                segment  DATA ATR_DATA
_global:                 ; グローバル変数 global の所在地
        DS.B    2        ; DS = Define Storage, 2 = Word Size
end
```

［練習８］（ループ処理）

0x210000 番地以降に自分の名前をローマ字で表示するプログラムを作れ．

```
                segment  TEXT  ATR_CODE
        public  _main
        _main:
                MOV.L   #_string, ER0   ; 転送元セット
                MOV.L   #H'210000, ER1  ; 転送先セット
        _loop1:
                MOV.B   @ER0, R2L       ; 読み込み
                MOV.B   R2L, @ER1       ; 書き込み
                BEQ     _loop2  ; Branch if EQual (flag Z=1)
                INC.L   #1, ER0 ; 次の番地に増やす
                INC.L   #1, ER1 ;     同上
                BRA     _loop1  ; BRanch Always 無条件ジャンプ
        _loop2:
                RTS
                segment DATA  ATR_DATA
        _string: DC.B   "ARIKAWA"  ; ヌル文字(0)が最後に付くことに注目
        end             ; ↑ 自分の名前を使うこと
```

■■［問題２］（ループ処理）

次のＣプログラムをコンパイルして，アセンブラレベルで解釈して，それをコメント形式でアセンブラソースに書き込みなさい．（例えば練習７のように書く）

```
        static char *myname = {"ARIKAWA"}; /* 各自の名前にせよ*/
        char  *copyto, *copyfrom;
        void main( void )
        {
                copyfrom = myname;
                copyto = (char *)0x00210000;
                while( *copyfrom )
                {
                        *copyto = *copyfrom;
                        copyto++;
                        copyfrom++;
                }
        }
```

［練習９］（算術演算の加算・減算）

レジスタ R1 に「－300」，レジスタ R2 に「500」を入れ，

和(R1+R2)を 0x210000 番地に，差(R1－R2)を 0x210010 番地に格納せよ．

また，得られた答を「電卓」で計算して確認せよ．

```
        MOV.W   #－300, R1
        MOV.W   # 500, R2
        MOV.W   R1, R0
        ADD.W   R2, R0   ; R0 に加算結果
        MOV.W   R0, @(H'210000)
```

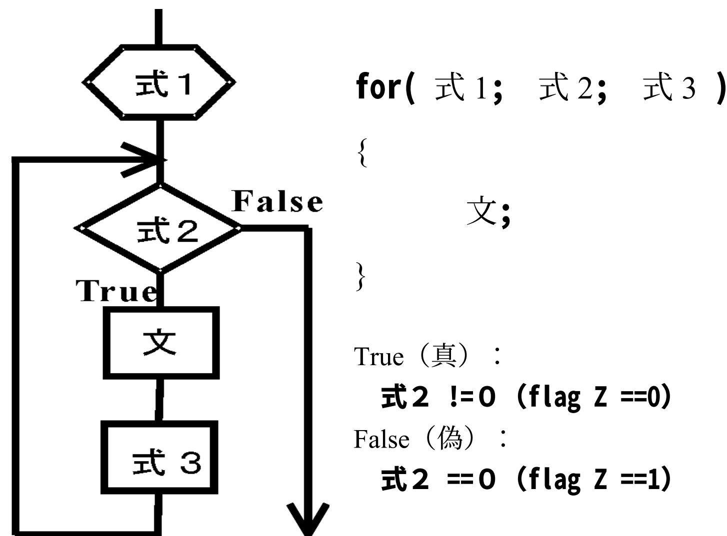
```
MOV.W    R1, R0
SUB.W    R2, R0   ; R0 に減算結果
MOV.W    R0, @(H'210010)
```

［練習 10］（算術演算の乗算・除算）

レジスタ R1 に「－300」，レジスタ R2 に「500」を入れ，
積(R1＊R2)を 0x210000 番地に，商(－R2／R1)を 0x210010 番地に格納せよ．
また，得られた答を「電卓」で計算して確認せよ．

```
MOV.W    #－300, R1
MOV.W    # 500, R2
MOV.W    R1, R0
MULXS.W  R2, ER0  ; ER0 に乗算結果
MOV.L    ER0,@(H'210000)
MOV.W    R2,R0
NEG.W    R0       ; 2 の補数をとる(符号が変わる)
EXTS.L   ER0      ; ロングワードサイズに符号拡張
DIVXS.W  R1,ER0   ; ER0 に除算結果
EXTS.L   ER0      ; ロングワードサイズに符号拡張
MOV.L    ER0,@(H'210010)
```

［練習 11］（for ループ処理と配列構造）



次のＣプログラムを，アセンブラレベルで解釈せよ．

```
unsigned char msk8[8];
main()
{
        int i;
        msk8[0] = 1;
        for( i=1; i<8; i++ )
                msk8[i] =  msk8[i-1]*2;
}
```

以下，コンパイル後のアセンブラソースを示す：

```
        segment  TEXT ATR_CODE
public   _main
_main:
; -2(er6)が int i の所在
        PUSH.L   ER6               ; BP ベースポインタ退避
        MOV.L    ER7,ER6           ; SP→BP
        SUBS     #2,ER7            ; int i のサイズ分 SP を減算
; msk8[0] = 1;
        MOV.B    #H'01,R0L         ; 初期値１を R0L に入れ
        MOV.L    #_msk8,ER1        ; msk8[]の先頭番地を ER1 にして
        MOV.B    R0L,@ER1          ; その値をその番地に書き込む
; for( i=1; i<8; i++ )
                 ; 【式１】 i=1 初期化
        MOV.W    #H'0001,R2        ; i=1 の初期値を R2 にセットして
        MOV.W    R2,@(-2,ER6)      ; i に代入する
        BRA      ?00001:8          ; ループ継続判定にジャンプ
   ?00000:       ; 【式３】 i++ 処理
        MOV.W    @(-2,ER6),R0      ; i の値を R0 に取ってきて
        INC.W    #H'0001,R0        ; i++ インクリメント
        MOV.W    R0,@(-2,ER6)      ; i に代入する
   ?00001:       ; 【式２】 ループを継続するか i<8 判定
        MOV.W    @(-2,ER6),R0      ; i の値を R0 に取ってきて
        CMP.W    #H'0008,R0        ; i<8 のチェック
        BGE      ?00002:8          ; i>=8 ならループ終了
; msk8[i] =  msk8[i-1]*2;  【文】
        MOV.W    @(-2,ER6),R0      ; i を R0 に転送して
        SUBS.L   #H'00000001,ER0   ; ER0 = i-1 を実行
        EXTS.L   ER0
        MOV.L    #_msk8,ER1        ; mak8[]の先頭番地を ER1 に
        ADD.L    ER0,ER1           ; 先頭番地に i-1 の分を加算
        MOV.B    @ER1,R0L          ; その番地の内容を R0L に
        EXTU.W   R0
        SHLL.W   R0                ; 左シフトして２倍にする
        MOV.W    @(-2,ER6),R1      ; i を R1 に転送して
        EXTS.L   ER1               ; ER1 として使えるようにする
        MOV.L    #_msk8,ER2        ; mak8[]の先頭番地を ER2 にして
        ADD.L    ER1,ER2           ; i を加算にて目的の番地を得る
        MOV.B    R0L,@ER2          ; その番地に計算結果を転送する
        BRA      ?00000:8          ; ループ継続で無条件ジャンプ
   ?00002:       ; 【ループ終了】
   _main_end:
        MOV.L    ER6,ER7           ; SP を元の値に戻す
        POP.L    ER6               ; BP を元の値に戻す
        RTS
public _msk8
```

```
segment  DATA ATR_DATA
    _msk8:
         DS.B     8                 ; msk8[8]の所在地
```

［練習 12］（サブルーチン）

次のＣプログラムを，アセンブラレベルで解釈せよ．

```
int max2( int, int );
int a, b, c;
void main( void )
{
        a = 0xAAA;
        b = 0xBBB;
        c = max2( a, b );
}
int max2( int x, int y )
{
        if( x > y )
                return x;
        else
                return y;
}
```

重要ポイント：
スタック領域を
デバッガで観察せよ!!
PC、SPにも注目!!

- **引数の渡し方**
- **サブ関数への飛び方**
- **戻値の返し方**
- **元の関数への帰り方**

以下，コンパイル後のアセンブラソースを示す：

```
        segment  TEXT ATR_CODE
public  _main
_main:
; a = 5;
        MOV.W    #H'0AAA,R0
        MOV.W    R0,@_a
; b = 6;
        MOV.W    #H'0BBB,R1
        MOV.W    R1,@_b
; ここから特に PC，SP，BP，スタック及びデータ領域を全て注意深く観察せよ．
; c = max2( a, b );  引数の後ろから渡される
        PUSH.W   R1                ; 引数 b をスタックに
        PUSH.W   R0                ; 引数 a をスタックに
        JSR      @_max2            ; max2()関数にジャンプ
        ADDS.L   #H'00000004,ER7   ; SP を引数のサイズ分だけ調節
        MOV.W    R0,@_c            ; 戻り値 R0 を変数 c に代入
_main_end:
        RTS
segment  TEXT ATR_CODE
public  _max2
_max2:
;       x        8(er6)
;       y        10(er6)
```

```
        PUSH.L  ER6             ; BPを退避
        MOV.L   ER7,ER6         ; SP → BP
; if( x > y )
        MOV.W   @(+8,ER6),R0    ; x → R0
        MOV.W   @(+10,ER6),R1   ; y → R1
        CMP.W   R1,R0           ; 大小比較
        BLE     ?00000:8        ; xが小さいとジャンプ
; return x;
        MOV.W   @(+8,ER6),R0    ; xが大きいのでR0に
        BRA     _max2_end:8
        BRA     ?00001:8
  ?00000:
; else  return y;
        MOV.W   @(+10,ER6),R0   ; yが大きいのでR0に（戻値）
  ?00001:
_max2_end:
        POP.L   ER6     ; BPを元の値に戻す
        RTS             ; PC プログラムカウンタを観察せよ
        public _a
        public _b
        public _c
                segment DATA ATR_DATA
        _a:     DS.B    2
        _b:     DS.B    2
        _c:     DS.B    2
        end
```

■■［問題３］（ループ処理とサブルーチン）

１）上図のように，仮想テキスト画面（**0x210000 番地**～）に，「*****」**(0x2A)**で作った三角形を表示するＣプログラムを作れ．**for** ループを使用せよ．

２）作成した三角形（頂点：**0x210000 番地**）が外接する矩形領域を，

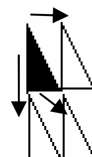
すぐ右隣り（頂点：**0x210006 番地**）

すぐ真下に（頂点：**0x210060 番地**）

ｘｘすぐ右斜下（頂点：**0x210066 番地**）

にコピーするＣプログラムを作れ．

コピーには，サブルーチンを用いよ．



＜ヒント＞（例）

```
void CopyBox( char * );
char *p;
main()
{
        int i, j;
        for( i=1; i<=6; i++ )
        {
                for( ？？？？ )
                ？？？？
        }
        CopyBox( (char *)0x210006 );
        CopyBox( (char *)0x210060 );
        CopyBox( (char *)0x210066 );
}
void CopyBox( char *top )
{
        ？？？？？？
}
```

１）の部分（int i, j; から for ループの } まで）

２）の部分（CopyBox の呼び出しから関数 CopyBox の終わりまで）

■■［問題４］（ループ処理とサブルーチン）その２

デバッガの多くの機能を活用して，次の問題をレポート提出：

１）次のＣプログラムを，Ｃ言語レベルで解釈せよ．

２）さらにコンパイルして，アセンブラレベルで解釈せよ．

```
static char *myname = {"ARIKAWA"}; //自分の名前にする
char  *copyfrom, *copyto=(char *)0x00210000;
int ret = -1;
int display( char letter )
{
        *copyto++ = letter;
        if( letter ) return -1;
        else return 0;
}
void main( void )
{
        for( copyfrom=myname; ret; copyfrom++ )
                ret = display( *copyfrom );
}
```

【注意】文字フォントはＭＳゴシックにすること

MOV.L　ER7,ER6（ＭＳゴシック）

MOV.L　ER7,ER6（ＭＳ明朝）

MOV.L　ER7,ER6(Century)

MOV.L　ER7,ER6(Times New Roman)

３．３　実験ボードによる実習

YellowIDE を立ち上げ，ファイル→プロジェクトの新規作成をする．
プロジェクト → 設定**の**【ターゲット】タブで，他のプロジェクトからコピーボタンを押して，前節まで使用していた「シミュレーションデバッグ用のプロジェクトをロードして，下記の３個所を変更する．

オブジェクトの形式：　イエロースコープでリモートデバッグ
開始アドレス：　０００FF１７０　　（内蔵RAMを利用する）
レCソースもアセンブラレベルでデバッグのチェックを外す

実機用のプログラムをプロジェクトに追加して，メイクして，YellowSoft デバッガを立ち上げ，設定→環境設定で下記の設定をする．

【通信ポート】　　通信ポート＝COM１
ﾎﾟｰﾄ設定＝ﾋﾞｯﾄ/秒 9600，ﾃﾞｰﾀﾋﾞｯﾄ 8，ﾊﾟﾘﾃｨなし，ｽﾄｯﾌﾟﾋﾞｯﾄ 1，ﾌﾛｰ制御なし
【システム】　　デバッグモード＝リモート
【セグメント】　Ｐ２２下を参照.

実験ボードの電源を入れ，パソコンと RS232C で接続して下さい.
デバッグ→実行で実行されます。（デバッグ→終了で終了されます）
必要に応じて，ファイル→再ロードや，設定→通信ポート初期化を使用する.

```
//---------------------------------------------------------------------
//  練習問題  その１      jikken1.c    LED の点灯がシフトする
//---------------------------------------------------------------------
#define P4DDR (*(volatile unsigned char *)0xFEE003) /* P4DDR  Address*/
#define P4DR  (*(volatile unsigned char *)0xFFFFD3) /* P4DR   Address*/
void Wait(void);
unsigned char msk8[8];

void main(void)
{
        int i;

        #asm ANDC.B      #B'01111111,CCR  ;       // interrupt enable
        msk8[0] = 1;
        for( i=1; i<8; i++ )
                msk8[i] =  msk8[i-1]*2;
        P4DDR = 0xff;  //Port4 を全て出力モードにする。
        for(;;){
                for(i=0; i< 8; i++){
                        P4DR = ~msk8[i];  //出力は負論理
                        Wait();
                }
        }
}

void Wait(void)
{
```

```
        volatile unsigned int n;
        for(n=0xffff; n; n--){
                #asm nop
        }
}

//------------------------------------------------------------------
// 練習問題　その２　jikken2.c　LED 点灯が S1 で右に，S2 で左にシフト
//------------------------------------------------------------------
struct BIT{      /* ビットフィールドはＬＳＢから割り当てられる */
       unsigned char B0:1;       /*    Bit 0     */
       unsigned char B1:1;       /*    Bit 1     */
       unsigned char B2:1;       /*    Bit 2     */
       unsigned char B3:1;       /*    Bit 3     */
       unsigned char B4:1;       /*    Bit 4     */
       unsigned char B5:1;       /*    Bit 5     */
       unsigned char B6:1;       /*    Bit 6     */
       unsigned char B7:1;       /*    Bit 7     */
};
#define P4DDR (*(volatile unsigned char *)0xFEE003) /* P4DDR Address*/
#define P4DR  (*(volatile unsigned char *)0xFFFFD3) /* P4DR  Address*/
#define P8DR  (*(volatile struct BIT    *)0xFFFFD7) /* P8DR  Address*/
void Wait(void);         // 前問から参照コピーせよ
const unsigned char msk8[8]
        ={1<<0, 1<<1, 1<<2, 1<<3, 1<<4, 1<<5, 1<<6, 1<<7};
int Direction = 0;

void main(void)
{
        unsigned int i;

        #asm ANDC.B      #B'01111111,CCR  ;       // interrupt enable
        P4DDR = 0xff;   //Port4 を全て出力モードにする。
        for( i=3; ;i+=Direction ){
                if( ! P8DR.B0 ) Direction = 1;    // increment
                if( ! P8DR.B1 ) Direction = -1;   // decrement
                i &= 0x07;               // keep i 0～7
                P4DR = ~msk8[i];         // output to LED
                Wait();
        }
}

//------------------------------------------------------------------
//  練習問題　その３ jikken3.c  DIP スィッチの入力を LED に反映させる
//------------------------------------------------------------------
#define P4DDR (*(volatile unsigned char *)0xFEE003) /* P4DDR  Address*/
```

```
#define P4DR  (*(volatile unsigned char *)0xFFFFD3) /* P4DR   Address*/
#define P7DR  (*(volatile unsigned char *)0xFFFFD6) /* P7DR   Address*/
void main(void)
{
        unsigned char LED;
        #asm ANDC.B      #B'01111111,CCR  ;        // interrupt enable
        P4DDR = 0xff;  //Port4を全て出力モードにする。
        do{
                LED = 0;         // reset
                if ((~P7DR) & (1<<2)) LED |= (1+2);       // set
                if ((~P7DR) & (1<<3)) LED |= (4+8);
                if ((~P7DR) & (1<<4)) LED |= (16+32);
                if ((~P7DR) & (1<<5)) LED |= (64+128);
                P4DR = ~LED;     // LEDに負論理で出力する。
        }while(1);
}

//---------------------------------------------------------------------
// 練習問題　その４　jikken4.c　ヴォリューム(AD)でLED点灯幅を増減する
//---------------------------------------------------------------------
struct st_ad_csr {       /*  Bit  Access */
        unsigned char CH  :3;            /*    CH        */
        unsigned char CKS :1;            /*    CKS       */
        unsigned char SCAN:1;            /*    SCAN      */
        unsigned char ADST:1;            /*    ADST      */
        unsigned char ADIE:1;            /*    ADIE      */
        unsigned char ADF :1;            /*    ADF       */
};
#define P4DDR (*(volatile unsigned char *)0xFEE003) /* P4DDR  Address*/
#define P4DR  (*(volatile unsigned char *)0xFFFFD3) /* P4DR   Address*/
#define AD_DRA (*(volatile unsigned int *)0xFFFFE0) /* DRA    Address*/
#define AD_CSR (*(volatile struct st_ad_csr *)0xFFFFE8) /* AD CSR Address*/
unsigned int AD_Get(void);
void LedDsip(unsigned int Len);

void main(void)
{
        #asm ANDC.B      #B'01111111,CCR  ;        // interrupt enable
        P4DDR = 0xff;    // Port4を全て出力モードにする。
        do{
                LedDsip( AD_Get() >> 13 );  // 0～7の値に変換
        }while(1);
}

unsigned int AD_Get(void)// ADから値を得る
{
```

```
        AD_CSR.ADST = 1;          // AD StarT
        for( ;!AD_CSR.ADF; ) ;    // AD Flag 新規データ待ち
        AD_CSR.ADF  = 0;          // reset flag
        return AD_DRA;            // 新規データを戻り値に
}

void LedDsip(unsigned int Len)    // LED を点灯
{
        switch (Len){
        case 0:  P4DR = ~0;               break;
        case 1:  P4DR = ~1;               break;
        case 2:  P4DR = ~(1+2);           break;
        case 3:  P4DR = ~(1+2+4);         break;
        case 4:  P4DR = ~(1+2+4+8);       break;
        case 5:  P4DR = ~(1+2+4+8+16);    break;
        case 6:  P4DR = ~(1+2+4+8+16+32);         break;
        case 7:  P4DR = ~(1+2+4+8+16+32+64);      break;
        case 8:  P4DR = ~(1+2+4+8+16+32+64+128);  break;
        default:
                P4DR = ~(1+4+16+64);      break;
        }
}

//-----------------------------------------------------------------
//  練習問題 その５    jikken5.c       スピーカより音が出ます。
//-----------------------------------------------------------------
union un_da_cr{   /* DACR */
    unsigned char BYTE;                   /*  Byte Access */
    struct {                              /*  Bit  Access */
          unsigned char wk   :5;                  /*              */
          unsigned char DAE  :1;                  /*    DAE       */
          unsigned char DAOE0:1;          /*    DAOE0     */
          unsigned char DAOE1:1;          /*    DAOE1     */
          }     BIT;
};
#define DA_CR (*(volatile union un_da_cr *)0xFFFF9E) /* CR  Address*/
#define DA_DR0 (*(volatile unsigned char *)0xFFFF9C) /* DR0 Address*/
void Wait(void);
const unsigned char msk[]={ 128,191,238,255,238,192,128,65,18,0,18,64 };
unsigned char *pMsk;

void main(void)
{
        int i;

        pMsk=msk;
```

```
        DA_CR.BIT.DAOE0=1;
        DA_DR0=*pMsk;
        #asm ANDC.B     #B'01111111,CCR  ;      // interrupt enable

        for(;;){
                DA_DR0=*pMsk;
                pMsk++;
                if(pMsk >= (msk+sizeof(msk))) pMsk=msk;
                Wait();
        }
}

void Wait(void)
{
        volatile unsigned int n;
        for(n=0xff; n; n--)
        {
                #asm nop
        }
}
```

◆ ◆ 演習問題１
ボリュームによって、ＬＥＤの点灯シフト速度が変わるようにしなさい。
（ヒント：「練習問題その４」と「その１」から取捨選択する）

◆ ◆ 演習問題２
「練習問題　その４」のプログラムでは、ボリュームを最大に廻しても最後の１個のＬＥＤが点灯しない。点灯できるようにプログラムを改良せよ。ただし「リニア」に動作すること。つまり回転角度１／８で１個、……５／８で５個……、８／８のフルスケールで８個が点灯。

◆ ◆ 演習問題３
ボリュームによって、ブザーの音が高低するように、「練習問題その５」buzz.c を改良しなさい。

```
//-------------------------------------------------------------------
//   発展問題 （割込みで LED の点灯がシフト。Timer0 の値を自動的に再設定）
//-------------------------------------------------------------------
```

■ 割り込みプログラムを実行するための設定：
１）cc0d.asm に INTVECT.DEF をインクルードする.

```
        segment  main           ; この直後
        INCLUDE(INTVECT.DEF)    ; ここです
```

２）プロジェクトの設定の割り込みタブにて，次のように設定する．

ベクタ番号	関数名
24	InterruptIMIA0

1
関数名

割込関数名をコピーしてきて貼り付け、ベクター番号＝24で 登録をクリック

★注意）
割込関数を使わないときは、設定を元に戻さないと誤動作します.

```
#include  "IV3067f.h"    //別途配布します
#include<stdio.h>
void Timer0_init(unsigned int n);
unsigned int Time_N, TimeCnt;

void main(void)
{
        int i;
        P4DDR = 0xff;                       //Port4 を全て出力モードにする
        P4DR.BYTE = ~0x01;                  //Port4 の初期値を 0xfe にする
        Timer0_init(50);                    //10msS X 50 = 500mS
        #asm ANDC.B  #B'01111111,CCR;       //intrupt enable

        for(;;){
                #asm nop
        }
}

void Timer0_init(unsigned int n)
{
        ITU0_TCNT=0;                // 周期カウントの為０からスタート
        ITU0_GRA=10000;             // 10mS(8MHz/8 /10000)
        ITU0_TCR.BIT.TPSC=3;        // internal clk 1/8
        ITU0_TCR.BIT.CCLR=1;        // GRA によりクリア
        ITU_TISRA.BIT.IMIEA0=1;     // GRA comper machi intrrupt enable */
        ITU_TSTR.BIT.STR0=1;        // count start */
        TimeCnt=Time_N=n;           // 10mS X Time_N
}

interrupt void InterruptIMIA0(void)
{
        ITU_TISRA.BIT.IMFA0=0;      // GRA comper machi intrrupt flag clear
        if (TimeCnt){
                TimeCnt--;
        }else{
                TimeCnt=Time_N;                     // 10mS X Time_N
                P4DR.BYTE       <<=     1;
                P4DR.BYTE       |=      0x01;
                if (P4DR.BYTE==0xff)    P4DR.BYTE = ~0x01;
        }
}
```

【参考】　音を出すときの参考にしてください：

n	音名	平均律	純正律	百分比
0	A (ラ)	440　Hz	440　Hz	100
2	B (シ)	493.88　Hz	495　Hz	113
4	C (ド)	554.37　Hz	550　Hz	125
5	D (レ)	587.33　Hz	586.67　Hz	133
7	E (ミ)	659.25　Hz	660　Hz	150
9	F (ファ)	739.99　Hz	733.33　Hz	167
11	G (ソ)	830.61　Hz	825　Hz	188
12	A (ラ)	880　Hz	880　Hz	200

「純正律」とは，自然倍音を用いて作った音階です．周波数の関係が整数倍だと非常によくハモれて，ドミソ、ファラド、ソシレは，すべて周波数比で４：５：６の関係です．

「平均律」は、オクターブ（周波数が２倍）を、数学的に１２等分して半音を決めるもので、すべての半音・全音が同じ周波数比となります。つまり、

基準音の周波数をＦ、

オクターブを１２等分する半音をｎ（＝１～１２）、

それに相当する音の周波数をｆｎとすると、

$$fn = F \times 2^{\left(\frac{n}{12}\right)}$$

となります．蛇足ですが，基本音Ａ（ラ）の 440Hz は，それは音速約 320m／ｓを，宇宙の基本波長 72.3ｃｍで割ったものです．

★レポートの提出について

・中間レポートは電子提出で提出する．（やり方は授業で説明します）

・最終レポートは電子提出と、印刷したものを次の実験の先生に提出する．

・採点は、最終レポートも、電子提出に対し行うので、必ず電子提出する．

★レポートの採点基準について

・プラグラムやコメントがちゃんと理解して書かれているか？

・マイコンの知識と、その動きがちゃんと理解できているか？

・デバッガがちゃんと使えているか？（文章だけでは表現できないので、操作画面を貼り付ける．ただし必要部分のみカットすること．やり方は授業で説明します.）