ソフトウェア第三
— ガイダンス・C 言語のプログラムから機械語まで —

機械情報工学科　水内郁夫

http://www.jsk.t.u-tokyo.ac.jp/~ikuo/lec/soft3/
(ID:soft3, pass:MechanoI)

2008 年 10 月 6 日 (月)
10 月 7 日 12:20 修正

1　ソフトウェア第三について

ソフトウェア第三のウェブページ http://www.jsk.t.u-tokyo.ac.jp/~ikuo/lec/soft3/ に，資料，サンプルプログラム，課題等を置く．アクセス制限がかかっている資料等は，ID:soft3，pass:MechanoI でダウンロードできる．

講義中に，各自がノート PC を開いて実際にサンプルをダウンロードして試したりするので，毎回ノート PC を持参のこと．gcc[1] [2]が使用できる環境がインストールされていること．できれば，SH の gcc クロスコンパイラも使用できる環境になっていることが望ましい．Windows で使用する場合は，Cygwin(http://www.cygwin.com/) が使用できるようにしておく．情報理工学系研究科で貸し出したノート PC はこれらがインストールされている．

課題を提出する場合は，次のようにすること：

- To: ikuo-soft3-08@jsk.t.u-tokyo.ac.jp
- Subject: soft3-kadai:2008mmdd:xxxxx　　(講義後に提出の課題 (原則として締め切りは次週の講義開始時))
- mm,dd,xxxxx は，月日・学生証番号を入れること．
- 本文に，学生証番号・氏名・学科を記載すること．
- 感想等もあれば適宜記載してください．

なお，今回の課題は末尾 (**8** 節) にあります．

2　コンピュータの階層構造

コンピュータは，ハードウェアのデジタル電子回路のレベルから，高水準言語 (C,C++,Java 等々) のレベルまで，各々のレベルだけに集中することができるような形で階層化されたシステムになっている (図 1) [3]．

[1] GNU Compiler Collection; http://gcc.gnu.org/
[2] GNU; http://www.gnu.org/
[3] 参考書籍：Andrew S. Tanenbaum, Structured Computer Organization, fourth edition, Prentice-Hall, Inc., 1999. (邦訳: 構造化コンピュータ構成, 第 4 版, (訳) 株式会社ロングテール, 長尾高弘, ピアソン・エデュケーション, 2000)

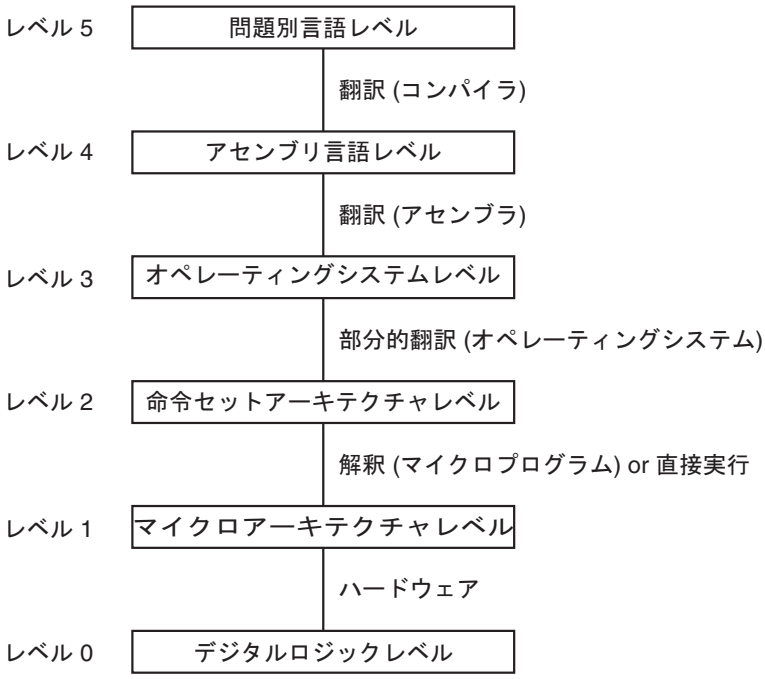


図 1: コンピュータの階層構造のうちの 6 レベル

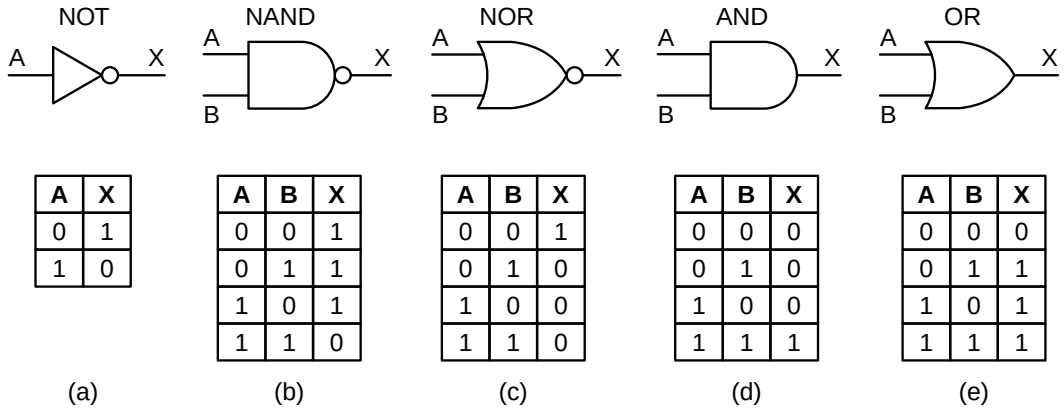
デジタルロジック (digital logic) レベルは，AND, OR, NOT 等の論理演算回路 (ゲート) をどのように組み合わせて回路を作るかというレベルである．ゲートはトランジスタ等のスイッチ素子ににより実現することができ，大規模な CPU も，プリミティブなゲート (図 2) の膨大な組み合わせで成り立っている．ゲートの入出力を全通り表にしたものを，真理値表 (truth table) と呼ぶ (図 2)．いくつかのゲートを組み合わせてメモリを作ることができ，算術演算回路も構成できる．

その上のレベルである，マイクロアーキテクチャ(micro architecture) レベルは，レジスタ (register) 等のメモリ回路，算術論理ユニット (ALU: arithmetic logic unit) をどのように用意し，どのようなデータパス (data path) を形成するかを決める．また，図 1 のレベル 2 である命令セット (instruction set) レベルで，レベル 1 でいくつかの手順を行うことを一つの命令 (instruction) で実行するような演算をマイクロプログラムと呼ばれる．

レベル 2 の命令セットレベルは，その CPU が処理可能な個々の作業を表す．処理可能な作業の種類一つ一つを命令と呼び，その一覧が命令セットである．機械語とも言われる．C 言語のプログラムをコンパイル (compile) して生成されるいわゆる実行可能バイナリファイル (binary file; 二進数のファイル) は，ある特定の CPU で実行可能な命令 (機械語) が並んでいる．命令セットの設計は，プロセッサの設計の非常に重要な部分の一つである．

オペレーティングシステム (operating system) は，プログラマの視点から見たとき，命令セットが提供する機能に様々な新機能を追加するプログラムである．追加される新機能とは，例えばメモリ管理，マルチタスク等である．これらの新機能を使うための新命令をシステムコール (system call) と呼ぶ.

アセンブリ言語 (assembly language, assembler) は，1 つの文が 1 つのマシン語 (命令) に対応するような言語である．命令の一つ一つは数値であるため，マシン語のプログラムは数値の羅列であ



A	X
0	1
1	0

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

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

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

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

図 2: 基本ゲートを表す記号と真理値表

る．アセンブリ言語は，一つ一つの命令 (機械語) を人間が意味を推定しやすい文字列に変換し，メモリ番地にラベルを付ける．アセンブリ言語から機械語に変換するツールをアセンブラ (assembler) と呼ぶ．C 言語のプログラムをコンパイルすると，アセンブリ言語に変換され，さらにそれがアセンブルされ機械語のファイルが生成される．

3 アセンブリ言語

3.1 機械語

アセンブリ言語は，CPU の種類ごとに存在する．機械語すなわち命令セットは CPU によって異なるが，どの CPU でも各命令を分類すると，演算，メモリ操作，分岐の三種類の命令がある (しかない)．演算を行う命令は，加減算・算術シフト・論理シフトなどを行う．メモリ操作は，レジスタやメモリに格納されているデータのコピーを行う．分岐には無条件分岐・条件分岐がある．

3.2 オペコード・オペランド

命令は，行う操作を指定するオペコード (opcode; operation code の略)1 個と，操作の対象を指定するオペランド (operand)0 個以上からなる．例えば，A+B を計算し結果を B に格納する命令は，アセンブリ言語では，

```
add a, b
```

のように記述する．`ADD` がオペコード，`a` および `b` がオペランドである．例えば，`add` のオペコードが 0x05，レジスタ `a`，`b` がそれぞれ 0x00，0x01 ならば，上の一行のアセンブリ言語は，機械語の 0x05 0x00 0x01 に変換される．この変換を行うのがアセンブラである．アドレッシングモード (オペランドがレジスタかメモリか即値 (イミディエート; immediate; 値そのもの) か) によって，同じ `add` でも対応する機械語が異なる場合もある．

3.3 ラベル

アセンブリ言語文は，ラベル，オペコード，オペランド，コメントの 4 つの部分 (フィールド) を持つ．CPU は，プログラムカウンタ[4]が指すメモリ番地に書いてある機械語 (命令) を順に実行していく．分岐命令以外の命令の際は，プログラムカウンタは命令を読み出すたびにインクリメントされ，次の命令の読み出し点をポイントし続ける．分岐命令では，プログラムカウンタの値が上書きされ，メモリ番地順の実行に変化が生じる．アセンブリ言語のラベルは，分岐先などの場所を，アドレスでなく名前で扱うことを可能にする．仮にラベルが使えないとすると，アセンブリ言語のプログラムを編集すると，それにより分岐先のアドレスにずれが生ずるのを，手動で修正しなければならない．

```
        mov     #0x01, r2
        mov     #0x10, r3
LABEL1:
        add     r2, r2
        add     #-1, r3
        bgt     LABEL1
```

3.4 アセンブリ言語を使うシーン

アセンブリ言語によるプログラミングは難しい．デバッグも難しい．それでもアセンブリ言語を使う理由は，主に処理性能・プログラムサイズを限界まで最適化したいということや，ハードウェアデバイスに直接アクセスするプログラムを書けるという点である．組込み系のシステムや BIOS(basic input output system)，IPL(initial program loader) 等のプログラムでは，しばしばアセンブリ言語が使用されることがある．また，オペレーティングシステムの根本の部分やデバイスドライバ等でも，アセンブリ言語でのプログラミングが必要になる場面がある．

また，ソフトウェアを学ぶ人にとっては，コンピュータのハードウェアが動作するそのシステムの構造を知る上でアセンブリ言語は重要な要素の一つである．

4 コンパイラ

4.1 翻訳

ある言語で書かれたプログラムを，別の言語に変換するプログラムを，トランスレータと呼ぶ．元のプログラムの言語をソース言語 (source language)，変換後の言語をターゲット言語 (target language) と呼ぶ．アセンブラは，アセンブリ言語をソース言語，機械語をターゲット言語としたトランスレータである．

コンパイラは，高水準言語をソース言語とし，機械語やアセンブリ言語をターゲット言語としたトランスレータである．C 言語のコンパイラは実存プロセッサの命令セットに即したアセンブリ言語・機械語に翻訳するが，Java はターゲットは Java 仮想マシンの機械語である．

[4]次に実行すべきメモリ番地を保持するレジスタ

4.2 C コンパイラの仕事

C コンパイラは，C 言語で書かれたプログラムを，プリプロセッサ (pre processor)，マクロプロセッサで処理し，字句解析 (lexical analysis)[5]，構文解析 (parser) を行い，アセンブリ言語に変換する．さらにアセンブラが自動的に呼ばれ機械語のオブジェクトモジュールに変換する．さらに自動的にリンカが実行され，実行可能プログラムが生成される．

複数のオブジェクトモジュールから一つの実行可能ファイルを作ることもできる．これを分割コンパイルと言い，コンパイル時間を少なくすることができる．多くのソフトウェアは分割コンパイルの仕組みを利用している．

4.3 gcc によるコンパイルの様子を見る

`http://www.jsk.t.u-tokyo.ac.jp/~ikuo/lec/soft3/docs/20081006-sample.tar.bz2` のサンプルをダウンロードする．

```
$ gcc -E main.c | less
```

により，プリプロセッサ，マクロプロセッサの仕事を確認できる．`#define` によるマクロも展開されている．`|`は標準出力をパイプし `less` につないでいる．`less` が何なのかは，`man` を読むこと．

```
$ gcc -O -S main.c
```

により，アセンブリ言語のプログラム (`main.s`) を生成する．`-O` を付けると，最適化が行われる．最適化のレベルは，`-O3` 等として調節できる (数値が大きい方が最適化の度合いが大きい)．

C 言語のプログラムがどのように機械語に変換されるかを見るために，以下の項目はどのようにアセンブリ言語に変換されるかを調べてみよう．

- 代入文
- 四則演算
- 浮動小数点数
- if 文
- 繰り返し文 (for や while)
- サブルーチン (関数) 定義
- サブルーチン呼び出し
- 局所変数 (関数内) の取り方
- サブルーチンコールにおける実引数の渡し方
- サブルーチンの戻り値の返し方

また，最適化オプションを付けた場合と付けない場合を比較することで，最適化の効果を見ることができる．プログラムカウンタやスタックポインタの取り扱いも見てみよう．

[5] 最小単位の字句 (トークン) に分解するので tokenizer とも呼ばれる．

5 リンカ

リンカの仕事は，1 つ以上のオブジェクトモジュール及びライブラリから実行可能バイナリファイルを作成することである (図 3).

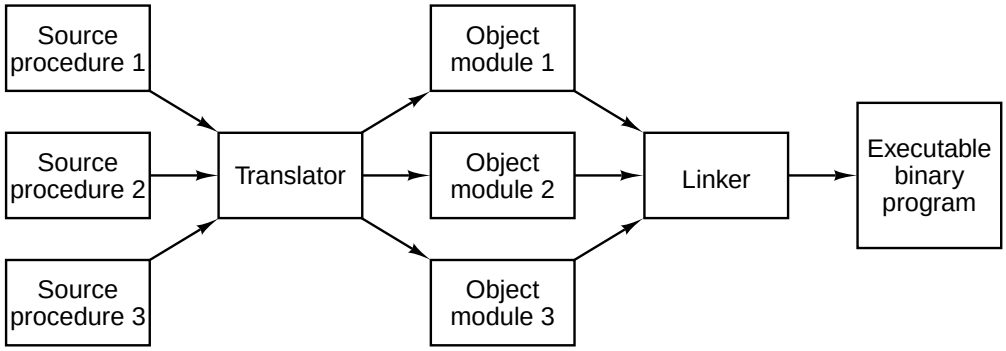


図 3: 分割コンパイルの様子

5.1 アドレス空間の再配置

各オブジェクトモジュールの中は，アドレスは 0 番地から始まっており，オブジェクトモジュール内に定義されている関数や変数のアドレスはそのオブジェクトモジュール内でしか通用しない番地になっている．これらはシンボル表という形でオブジェクトモジュール内に書き込まれており，リンカはそれを参照し，関数・変数を再配置し，呼び出し部におけるアドレスを再配置後のアドレスになるように修正する．

5.2 外部参照

オブジェクトモジュールの中には，外部の関数や変数 (external symbol) の呼び出しやアクセスが含まれる場合がある．例えば，printf は，外部関数へのアクセスである．静的リンク (static link) の場合は，リンク時に全ての外部参照が解決される (全ての外部シンボルのアドレスが決定される) が，動的リンク (dynamic link) の場合はリンク時には外部参照は解決されず実際にそのシンボルが参照される際に初めて外部参照が解決されるシステムもある．後者は，Windows では動的リンクライブラリ (dynamic link library; ファイル名は`"libxxx.dll"`の形) と呼ばれ，UNIX 系システムではシェアードオブジェクト (shared object; ファイル名は`"libxxx.so"`の形) と呼ばれる．

例えば，ライブラリ`/usr/local/lib/libfuncs.dll` に定義された関数や変数を利用するプログラムソース `src.c` から，実行可能バイナリファイル `output.exe` を作成する場合は，

```
$ gcc -c src.c -o src.o
$ gcc src.o -o output.exe -L/usr/local/lib -lfuncs
```

のようにする．1 行目は `src.c` をコンパイルして `src.o` を生成し，2 行目は`/usr/local/lib/libfuncs.dll` をリンクして実行可能ファイル `output.exe` を生成する．

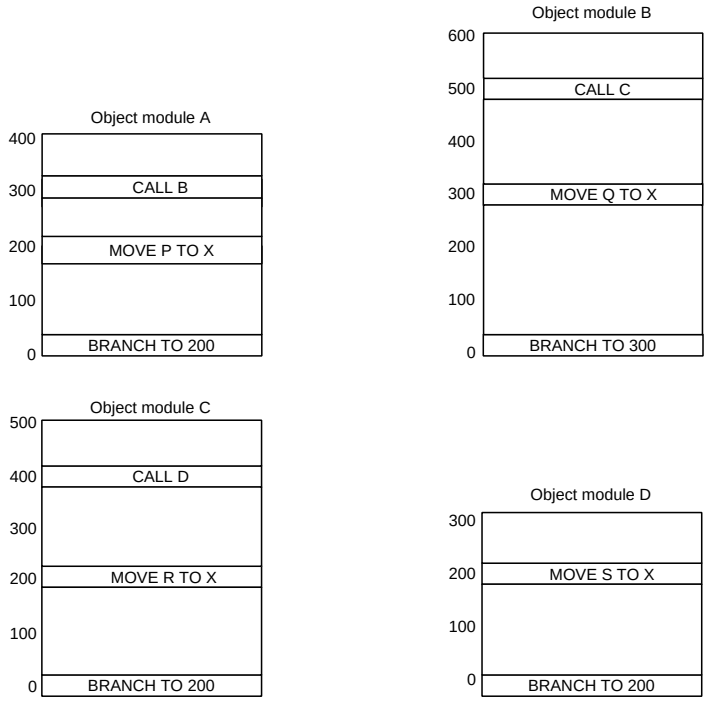


図 4: 個々のオブジェクトモジュールは固有のアドレス空間を使用している.

5.3 セクション配置

最近のある程度以上の規模のコンピュータでは，仮想メモリ・MMU などのシステムがあり，実行可能ファイルが実際に実行される時に，実メモリのどこの番地にロードされるかはわからない. それに対し，組込み系のシステムでは限られたメモリ容量の中で，プログラムが適切にメモリの利用を配分する必要がある．例えば，グローバル変数は RAM 領域の下位番地から順に，プログラム本体は ROM 領域に，スタックは RAM 領域の上位番地から降りてくるようになど，内容に応じてどのアドレス領域を使うべきかが異なってくる.

そこで，`.data`(初期値のある大域変数)，`.bss`(初期値の無い大域変数)，`.text`(read only な領域) 等のセクションを定義し，それぞれのセクションが使用するアドレス範囲を指定する．リンカスクリプトと呼ばれる設定ファイルに配置の設定を記載しリンカに渡すという方法か，リンカの実行時のコマンドラインオプションにより指定する方法がある.

5.4 バイナリファイル (オブジェクトモジュールや実行ファイル) を調査する

オブジェクトモジュールや実行ファイル等のバイナリファイルは，GNU の binutils[6]というツール群に含まれるツールを使って，逆アセンブル，シンボル一覧表示，セクション別領域サイズなどを調べることができる.

```
$ objdump -d main.o
$ size *.o *.exe
$ nm *.o
```

[6] `http://www.gnu.org/software/binutils/`

ライブラリは，複数のオブジェクトモジュールを束ねたファイルで，その中の必要な関数定義のみを参照することができるようになっている．静的リンクでは必要な関数定義部分のみを切り出してユーザプログラムとリンクする．静的ライブラリは，

```
$ ar cq libfuncs.a func1.o func2.o func3.o
```

のようにして作成し，動的ライブラリは，

```
$ gcc -shared -o libfuncs.dll func1.o func2.o func3.o
```

のようにして作成する．Windows 以外の OS では，`libfuncs.dll` は，`libfuncs.so` となる．

6 CPU の構造とアセンブリ言語

6.1 ALU，レジスタ，メモリ

CPU の最も基本的な要素は演算装置とレジスタである (図 5)．ALU とレジスタは最も高速に接続され，レジスタの値を読み出して ALU に入力し，演算結果をレジスタに書き込む．CPU の行うことの基本は，命令 (instruction) の実行である．命令には，大きく分けると演算・メモリ操作・分岐の三種類がある．演算の命令の実行が図 5 である．演算の命令一回分には，どのレジスタの値を使って (source operand)，どの演算を行い (オペコード)，結果をどのレジスタに書き込むか (destination operand) が，含まれる．

図 6 は，CPU とメモリ (主記憶) の関係の概念的な図である．CPU は，メモリから命令を読み込み，その命令を解釈し，制御回路がその命令に従い各回路を制御する[7]．命令は，一般に 1 つのオペコードと 0 以上のオペランドからなる．次の命令を読み出すべきメモリアドレスは，特殊なレジスタ (プログラムカウンタ; program counter; PC) が保持している．メモリ上に並んだプログラムを実行する際，PC が指し示すメモリ番地から命令が一つ読み込まれ，PC はその分だけ値を増やすことを繰り返す (図 7).

6.2 命令

命令は次の三種類に分類できる．

演算命令 加減 (乗除)，シフトなどの演算を行う．演算器への入出力はオペランドで指示する．

データコピー命令 レジスタ・メモリ (主記憶) のデータのコピーを行う．

分岐命令 分岐命令は，PC の値を変更する．これによりプログラムの実行番地がジャンプする．条件分岐と無条件分岐

[7] メモリから命令を読み込み，その命令に従いメモリの内容を更新してゆくような構成のコンピュータを，ノイマン型アーキテクチャ(von Neumann architecture) と呼ぶ．これに対し，命令とデータを別の装置・経路で取り扱うコンピュータを，ハーバードアーキテクチャ(Harvard architecture) と呼ぶ．現在の CPU のほとんどはノイマン型アーキテクチャだが，命令キャッシュとデータキャッシュを別にするような，ハーバードアーキテクチャの特徴を活かすような CPU も増えている．

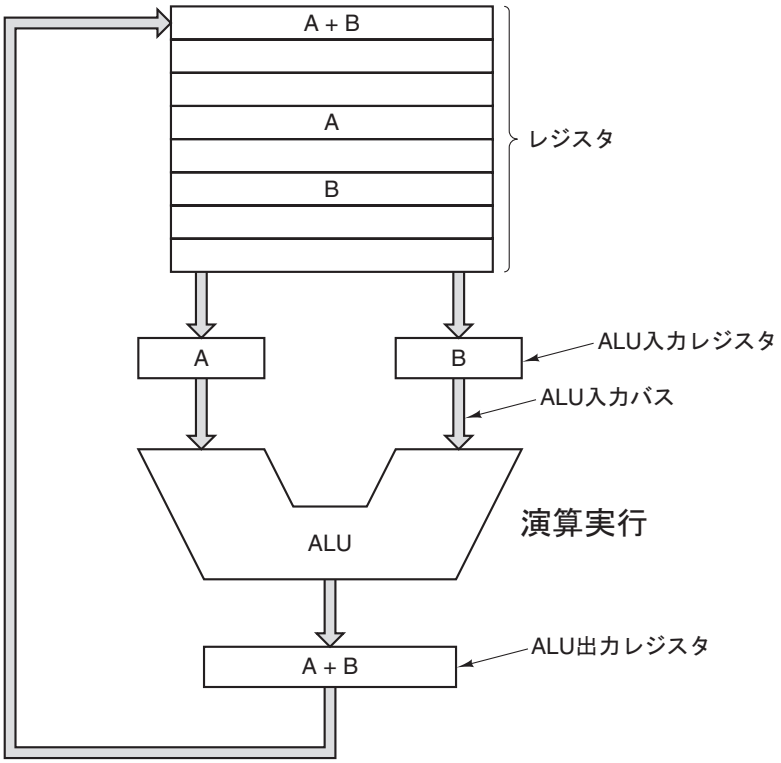


図 5: ALU とレジスタの接続の概念図: ALU とレジスタは CPU 内で最も基本的な要素である．レジスタは，演算を行う回路 (ALU) に最も近い最も高速な記憶装置である．A，B，A+B は，レジスタに記憶される値.

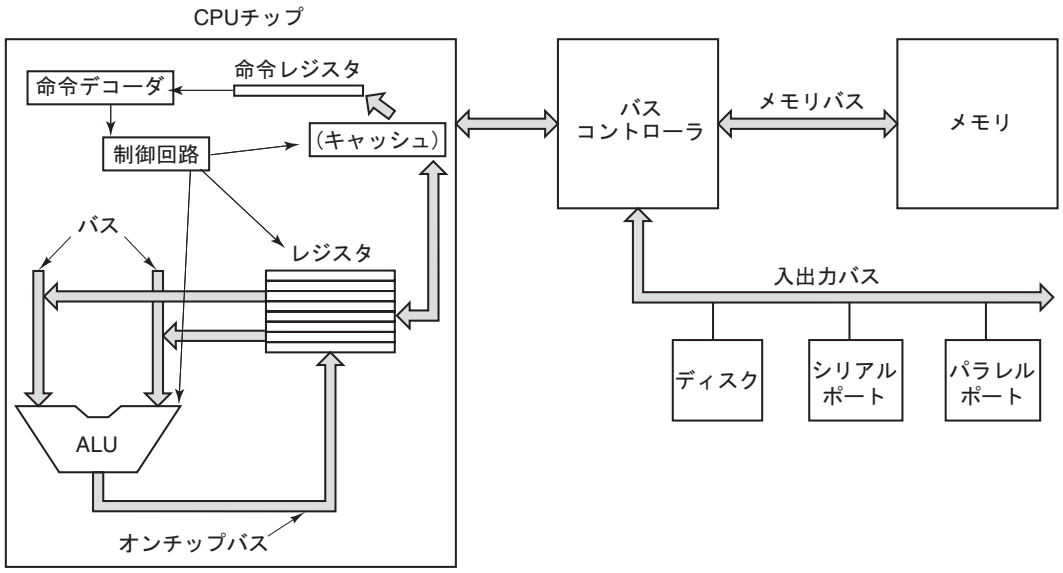


図 6: CPU チップとメモリ・入出力装置の接続例: CPU とメモリ・入出力装置は，それぞれ別のバス (メモリバス・入出力バス) を介してデータのやり取りをする．CPU はメモリから命令を読み込み (fetch)，解釈し (decode)，実行する.

何をオペランドとするか (アドレッシング) は，CPU の設計時に決められる．アドレッシングには，即値アドレッシング，レジスタ直接アドレッシング，レジスタ間接アドレッシング，メモリ直接アドレッシング，メモリ間接アドレッシング，インデックスアドレッシング，スタックアドレッシング等がある．即値 (immediate) とは，オペランドがデータの値やメモリ番地であるような場合を言う．その CPU が実行できる命令とアドレッシングを命令セット (instruction set) と呼ぶ．

6.3 ニーモニック，アセンブリ言語

各オペコードには対応する数値が割り当てられている．メモリ上のプログラムは,「オペコードに対応する数値とオペランドに対応する数値の組み合わせ」が羅列されている．これを機械語 (マシン語) プログラムと呼ぶ．また，各オペコードには対応する文字列が割り当てられており，オペランドの記述法と合わせてニーモニック (mnemonic) と言う．アセンブリ言語は，ニーモニックやラベルを使ってプログラムが書かれる．機械語に変換される行以外に，擬似命令 (pseudoinstruction, アセンブラディレクティブ (assembler directive), 単にディレクティブとも言う) と呼ばれる，アセンブラへの指示を行うコマンドも，アセンブリ言語プログラムには含まれる．擬似命令には，他のファイルを読み込む `include`，条件アセンブリを行う `if`, `else`, `endif` などがある．一連のアセンブリ言語プログラムをまとめて複数個所で使うことができる，マクロも，たいていのアセンブラでは扱える．各命令に対応するニーモニックは，プロセッサによって異なるが，一般的なニーモニックの例を次に挙げておく．講義のページからリンクの張られている `http://www.jsk.t.u-tokyo.ac.jp/~ikuo/lec/soft3/instruction-sets.html` にも典型的なニーモニックを挙げてあるので参照すると良い．

加減乗除 `ADD`，`SUB`，`MUL`，`DIV`．オペランドのバイト数によって `B`(1byte)，`W`(2byte)，`L`(4byte) 等の修飾が付くこともある (`ADD.B`，`ADDB` 等)．

論理演算 `AND`，`OR`，`XOR`，`NOT` 等．

シフト演算 算術シフト (`SAR`:shift arithmetic right，`SAL` 等)，論理シフト (`SLR`，`SLL` 等) がある．

データのコピー `MOV`，`MOV.B`，`MOV.L`，`MOVB` 等．プロセッサによっては `LD/ST` となっている場合もあるが，これらは，load/store を意味し，メモリからレジスタへ/レジスタからメモリへのデータのコピーを意味する．

無条件分岐 `JMP`，`JA`:jump address，`BRA`:branch 等．サブルーチン呼び出し (`JSR`:jump subroutine，`CALL` 等) や，サブルーチンからの戻り (`RTS`:return subroutine，`RET` 等) もある．更に，割り込みルーチンからの戻り (`RTE`:return exception) もある．

条件分岐 `BEQ`:branch equal，`BGT`:branch greater than，`BLE`:branch less than or equal 等々．

即値 `#0x1234`，`$1234` のような形．PIC のリテラルも即値．

レジスタ `R1`，`%EAX`，のような形．

レジスタ参照 レジスタの値のメモリ番地の内容を指す場合，`@R1`，`(%EAX)`，のような形．

3.3 節に載っていた下の例は，実行するとレジスタ r2 が 2^{0x10} になる．

```
        mov     #0x01, r2     ; レジスタ r2 の値を 0x01 にする.
        mov     #0x10, r3     ; レジスタ r3 の値を 0x10 にする.
LABEL1:                       ; ラベル
        add     r2, r2        ; r2 の値と r2 の値を加算．結果を r2 に.
        add     #-1, r3       ; -1 とレジスタ r3 の値を加算し，結果を r3 に.
                              ; (r3=r3-1)
        bgt     LABEL1        ; 最後の演算結果が 0 より大きければ LABEL1 に分岐.
                              ; (bgt は"branch if greater"の意味)
```

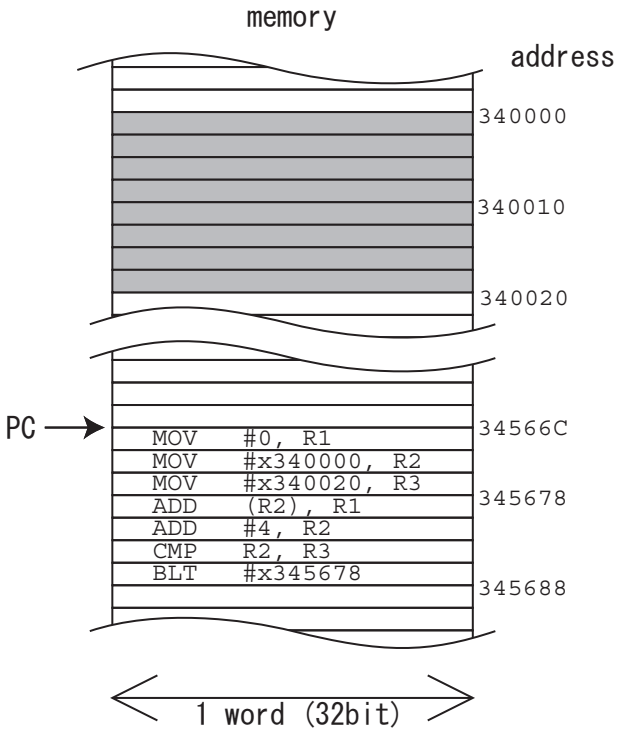


図 7: メモリ上にプログラムとデータを配置: プログラムカウンタ (PC) が指し示すアドレスの命令を読み込み実行する．PC の値は命令を読み込むたびに増加する．この例では薄く色の付いた領域はデータが格納されている．

◇ **課題** 1 ◇

1. 図 7 は，メモリ番地 0x34566C から順にプログラムを実行しようとしている．ここから 0x345688 までのプログラムは，何を行うプログラムかを説明せよ．ただし，ADD は二つのオペランドの指す値を加算し，結果を第二オペランドに書き込むものとする．また，(レジスタ) の形式はレジスタ間接アドレッシング[a]で，レジスタの値を番地とするメモリ内容を意味する．

2. 図 7 のメモリ番地 0x34566C〜0x345688 のプログラムを，C 言語で表現せよ．ただし，配列 `long val[8];` はメモリ番地 0x340000 からの 8 ワードに確保されている (つまり (val==0x340000) である) としてよい．

3. 2. で作成した C 言語プログラムを，適当な名前の関数として xxx.c として保存し，
   `gcc -S -O xxx.c -o xxx-native.s`,
   としてできたファイルを見てみよう．また，-O をはずしたり-O1, -O2, -O3 でどのように変わるかを見てみると良い．

1. の説明と 2. の C 言語表現をメールで提出すること．

```
To: ikuo-soft3-08@jsk.t.u-tokyo.ac.jp
Subject: soft3-kadai:20081006:xxxxx
```

[a] わからない単語は調べてみよう．

6.4 C コンパイル，リンク

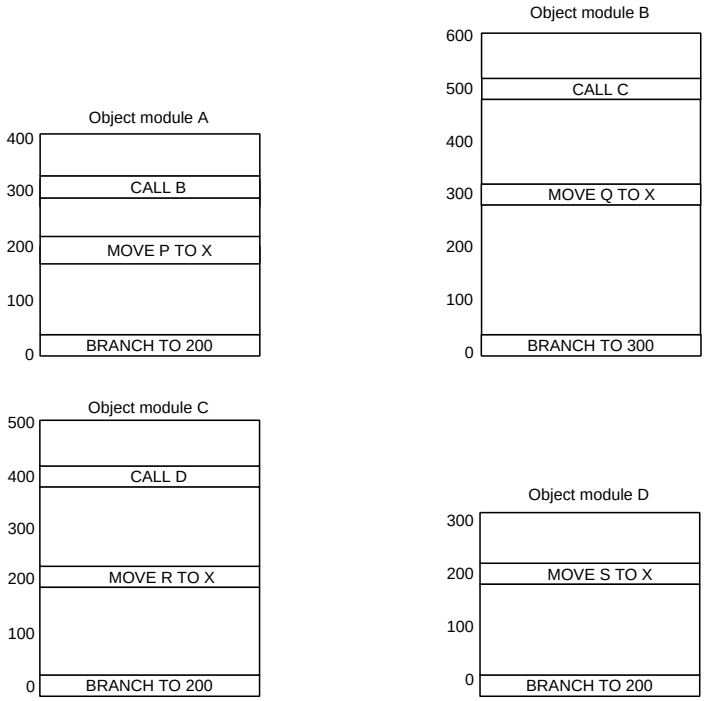


図 8: 個々のオブジェクトモジュールは固有のアドレス空間を使用している．

C 言語のプログラムは，プリプロセッサ，コンパイラ (アセンブラ)，リンカによる処理を経て，実行可能な機械語に変換される．

コンパイル (アセンブル) されたバイナリファイル (機械語のファイル) を，オブジェクトモジュール (オブジェクトファイル) といい，1 つ以上のオブジェクトモジュールから実行可能バイナリファイルを作成する．個々のオブジェクトモジュールは，コンパイル (アセンブル) された結果である機械語の他に，他のモジュールから参照できるシンボル[8]の一覧である**エントリポイント表**，そのモジュールが使用する外部シンボルの一覧である**外部参照表**，再配置が必要なアドレスのリストである**再配置辞書**等の情報が含まれる (図 10)．

リンカの 2 工程 (2 パス) のうち 1 工程目は，全てのオブジェクトモジュールから，全てのエントリポイントと外部参照から構成される，大域シンボル表を作成する．2 工程目では，1 工程目に作成した大域シンボル表に基づいて，必要なアドレスの再配置を行う．

[8] シンボルとは，具体的には，関数名，変数名 (固定アドレスのもの) である．

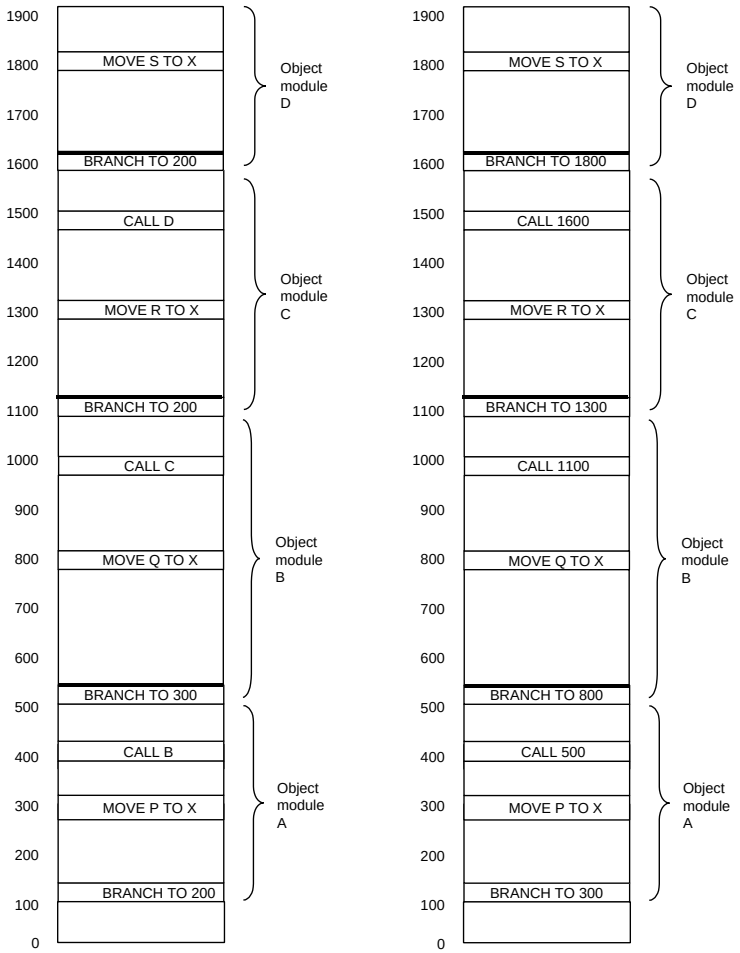


図 9: (a) 図 8 のオブジェクトモジュールを並べたが，まだ再配置・リンクを行っていない状態．(b) リンクと再配置を行った後の状態 (実行可能).

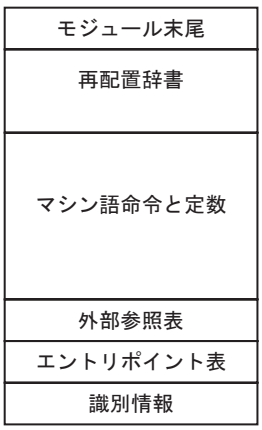


図 10: オブジェクトモジュールの内部構造

6.5 関数呼び出し

関数呼び出し (サブルーチンコール) は，専用の命令 (`jsr`(jump subroutine) や `CALL` 等) が用意されているが，これらの命令はサブルーチンに飛ぶ前に実行していた次のアドレスを記憶してからジャンプするということを行う．しかし，関数の引数の受け渡しや，戻り値 (返り値) の受け渡しは，これらの命令は感知しない．引数や戻り値の受け渡し方法は**呼出規約**と呼ばれ，いくつかの方法がある．

スタックを介した受け渡し cdecl: インテル x86[9]は，レジスタが少ないのでスタックを介した引数の受け渡しである cdecl(C 規約) や stdcall(Windows デファクト標準規約) が使われることが多い．戻り値は，EAX(x86 系の汎用レジスタの一つ) を介して呼出元に戻される．サンプルからアセンブリ言語のプログラム `sample1-cygwin.s` を生成して見てみると良い．ちなみに，`sample1-sh.s` は，SH-Linux 用にコンパイルした場合のアセンブリ言語のプログラムである (後日クロス開発を扱う際に詳しく説明予定).

レジスタを介した受け渡し fastcall: 汎用レジスタのいくつかを，引数受け渡し用として決めて，呼び出し側と呼び出され側で同じレジスタを引数と解釈する．引数が多くレジスタの数が不足する場合は，一部の引数はスタックを介して受け渡す.

一つのプログラムでは，同一の呼出規約でないといけない．どの呼出規約を使うかは，CPU の特徴 (レジスタが多いか少ないか等) やコンパイラの種類などによって異なる (コンパイラは複数の呼出規約に対応しているものもある)．cdecl, stdcall, fastcall 以外にどのような呼出規約があるかを調べてみると良い.

引数の受け渡しに使ったスタックの破棄を，呼出側で行うか，呼び出され側で行うか等も，異なる場合がある.

6.6 実際のアセンブリ処理

アセンブリ処理は，ニーモニックを機械的に機械語に置き換えてゆくという作業だけでは少し足りない．分岐先のラベルをアドレスに直す作業は，分岐先のアドレスがわかっていなければならないが，ソースファイルの先頭から順に読んでいった時に，分岐命令の時点ではそのラベルの箇所をまだ読んでいない可能性があるからである．この問題を前方参照問題という.

前方参照問題を解決する方法は 2 通りある．第 1 の方法は，ソースプログラムを 2 回読むというもので，ソースを読み込むことをパス (pass) と言い，アセンブラは一般に 2 パスが必要とされる．1 パス目では，1 行ずつマシン語に翻訳すると同時に記号表:symbol table を組み立てる．2 パス目でプログラムコード毎にアドレスをつけてゆく．分岐先のアドレスも 1 パス目で作成した表を参照して付けてゆく．前方参照問題を解決する第 2 の方法も，2 パスが必要であるが，1 パス目で中間的な形式として保存しておき，2 パス目では参照アドレスの解決が必要な箇所だけを読み込み解決する.

以下に，単純なアセンブラの 1 パス目と 2 パス目の概要を示す.

```
1   public static void pass_one( ) {
2     // この手続きは，単純なアセンブラの 1 パス目の概要を示す.
3     boolean more_input = true;    // 1 パス目を終了させるフラグ
4     String line, symbol, literal, opcode; // 命令のフィールド
```

[9]8086 から代々受け継がれているアーキテクチャ，80286, 80386, 486, Pentium,

```
5     int location_counter, length, value, type; // その他の変数
6     final int END_STATEMENT = -2; // 入力の末尾を知らせる
7
8     location_counter = 0;          // まず，0 の位置の命令をアセンブルする
9     initialize_tables( );          // 初期化処理全般
10
11    while (more_input) {           // more_input は END によって false になる
12      line = read_next_line( );    // 入力から 1 行を取得
13      length = 0;                  // 命令内のバイト数
14      type = 0;                    // 命令のタイプ (形式)
15
16      if (line_is_not_comment(line)) {
17        symbol = check_for_symbol(line); // 行にラベルは付けられているか？
18        if (symbol != null)        // ラベルがあるなら，記号と値を記憶する
19          enter_new_symbol(symbol, location_counter);
20        literal = check_for_literal(line); // 行にリテラルは含まれているか？
21        if (literal != null)       // リテラルがあるなら，記号と値を記憶する
22          enter_new_literal(literal);
23
24        // ここでオペコードタイプを判定する．-1 は無効オペコードを意味する
25        opcode = extract_opcode(line); // オペコードニーモニックを探す
26        type = search_opcode_table(opcode); // 例えば OP REG1,REG2 のような形を探す
27        if (type < 0)              // オペコードでなければ擬似命令か
28          type = search_pseudo_table(opcode);
29        switch(type) {             // この命令の長さを判定する
30        case 1: length = get_length_of_type1(line); break;
31        case 2: length = get_length_of_type2(line); break;
32        // その他の case 文
33        }
34      }
35
36      write_temp_file(type, opcode, length, line); // 2 パス目に役に立つ情報
37      location_counter = location_counter + length; // loc_ctr の更新
38      if (type == END_STATEMENT) { // 入力がなくなったか
39        more_input = false;        // 無いなら，
40        rewind_temp_for_pass_two( ); // 一時ファイルのファイルポインタ移動
41        sort_literal_table( );     // リテラル表をソート
42        remove_redundant_literals( ); // リテラル表から重複の除去などの管理処理を行う
43      }
44    }
45  }
```

```
1   public static void pass_two( ) {
2     // この手続きは，単純なアセンブラの 2 パス目の概要を示す．
3     boolean more_input = true;     // 2 パス目を終了させるフラグ
4     String line, opcode;           // 命令のフィールド
5     int location_counter, length, type; // その他の命令
6     final int END_STATEMENT = -2; // 入力の末尾を知らせる
7     final int MAX_CODE = 16;       // 命令あたりのコードのバイト数の上限
8     byte code[ ] = new byte[MAX_CODE]; // 1 つの命令に対して生成されたコードを格納する
9
10    location_counter = 0;          // まず，0 の位置の命令をアセンブルする
11
12    while (more_input) {           // more_input は END によって false になる
13      type = read_type( );         // 次の行のタイプフィールドを取得
14      opcode = read_opcode( );     // 次の行のオペコードフィールドを取得
15      length = read_length( );     // 次の行の長さフィールドを取得
16      line = read_line( );         // 入力から実際の行を取得
17
18      if (type != 0) {             // タイプ 0 はコメント行
19        switch(type) {             // 出力コードを生成
20        case 1: eval_type1(opcode, length, line, code); break;
21        case 2: eval_type2(opcode, length, line, code); break;
22        // その他の case 文
23        }
24      }
25      write_output(code);          // バイナリコードを出力
26      write_listing(code, line);   // リストを 1 行出力
27      location_counter = location_counter + length; // loc_ctr を更新
28      if (type == END_STATEMENT) { // 入力が無くなったか?
29        more_input = false;        // 無いなら，管理処理を行い，
```

```
30          finish_up( );              // 終了
31        }
32      }
33    }
```

7 PIC

デバイス	プログラムフラッシュ	データメモリ	データEEPROM
PIC16F873	4K	192 バイト	128 バイト
PIC16F876	8K	368 バイト	256 バイト

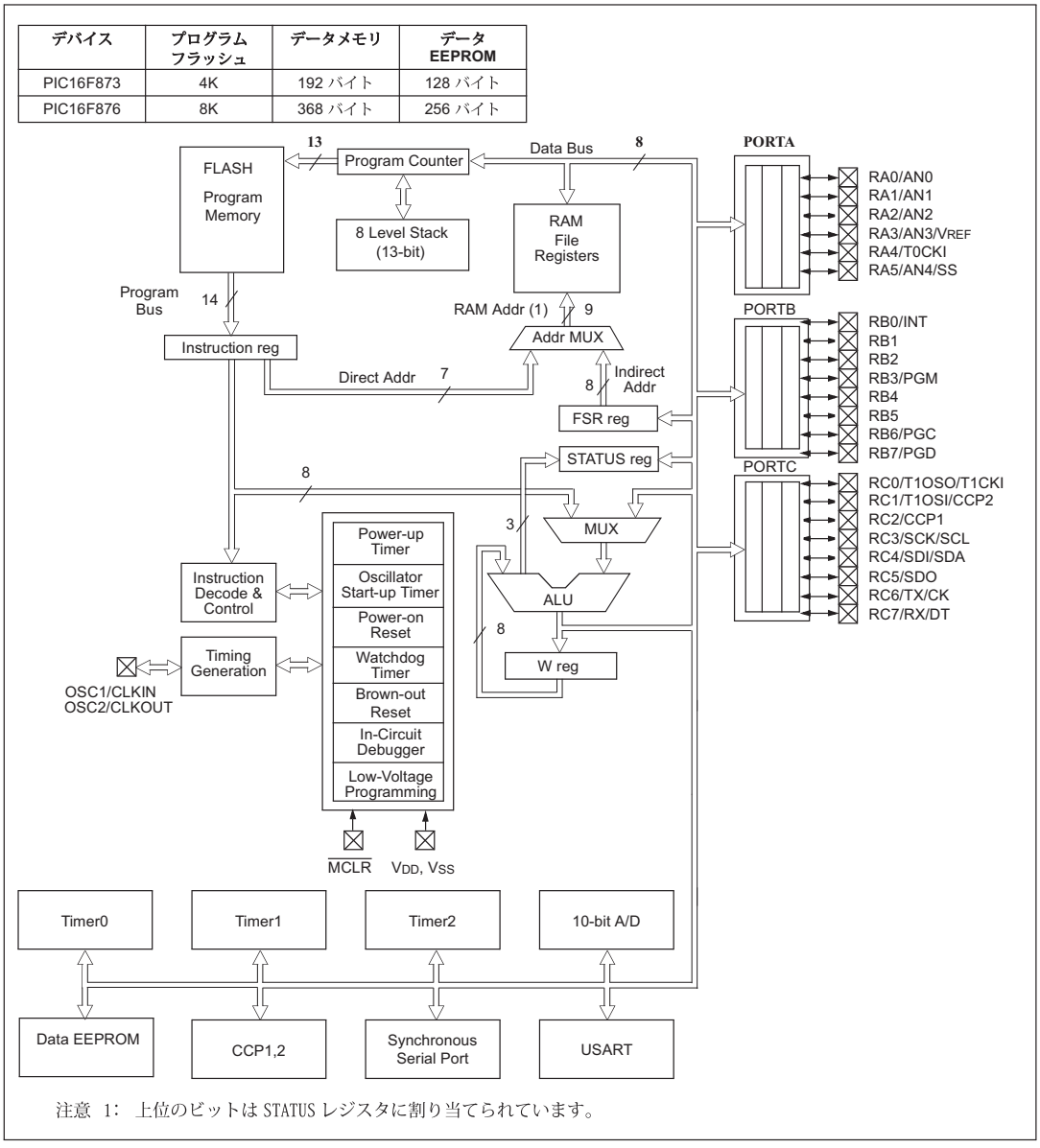


図 11: PIC16F876 のブロック図

図 7 のように，プログラム (0x1234566C〜) と，データ (0x12340000〜) が同じメモリ上にあるような場合を，**ノイマン型アーキテクチャ**と言う.

これに対し，MICROCHIP 社の CPU である PIC シリーズは，プログラム用のメモリとデータ用のメモリが完全に分離した，**ハーバードアーキテクチャ**である (図 11)．プログラム用のメモリは 8bit の倍数の幅に限らず，命令数やオペランドの長さ等から 14bit 等のワードサイズになっている．データ用のメモリはレジスタファイル等と呼ばれ，全てのデータメモリはレジスタであると考えてよい．ただし (ゆえに)，記憶容量は限られておりメモリの増設のようなことも難しい．

PIC はメモリが全てレジスタと考えてよいので，桁あふれ等のフラグを保持する STATUS レジスタや，メモリ領域を切り替える「Page」を管理する PCLATH レジスタ等の，特殊機能のレジスタも，それに割り当てられたアドレスを指定することでアクセスする．

その他，チップに内蔵されている周辺機能 (シリアル通信や IO ポートなど) を操作するためのレジスタも，その他の汎用レジスタ (メモリとして使う) と同じアドレス空間に並んでいる (図 12)．

レジスタファイル (メモリ空間) の中のレジスタのうち，上に述べたような STSTUS レジスタや PCLATH レジスタ，各種機能の設定レジスタや IO ポート用のレジスタ等は，SFR(Special Function Register) と呼ばれている．図 12 で「汎用レジスタ」となっている部分は，ユーザ (プログラマ) がメモリとして使用できるメモリ範囲で，それ以外のアドレスは SFR である．

PIC の記憶領域は，基本的に，プログラムメモリとレジスタファイル (SFR を含む) だが，スタックメモリとプログラムカウンタは別に用意されている．また，ALU のそばには，一回分の計算結果を保持することができる W レジスタ (W reg; Working register) がある．PIC の多くの命令は，結果の保存先として，レジスタファイルの中の特定のアドレスか，W レジスタかどちらかを指定する．

- PIC のアーキテクチャの説明:`http://www.microchip.co.jp/seminar/200203/architecture_x14.pdf`
- PIC の命令セット `http://www.microchip.co.jp/seminar/200203/meireiset_x14.pdf`
- PIC の日本語ドキュメント `http://www.microchip.co.jp/document.htm`
- MICROCHIP 社のホームページ `http://www.microchip.com/`

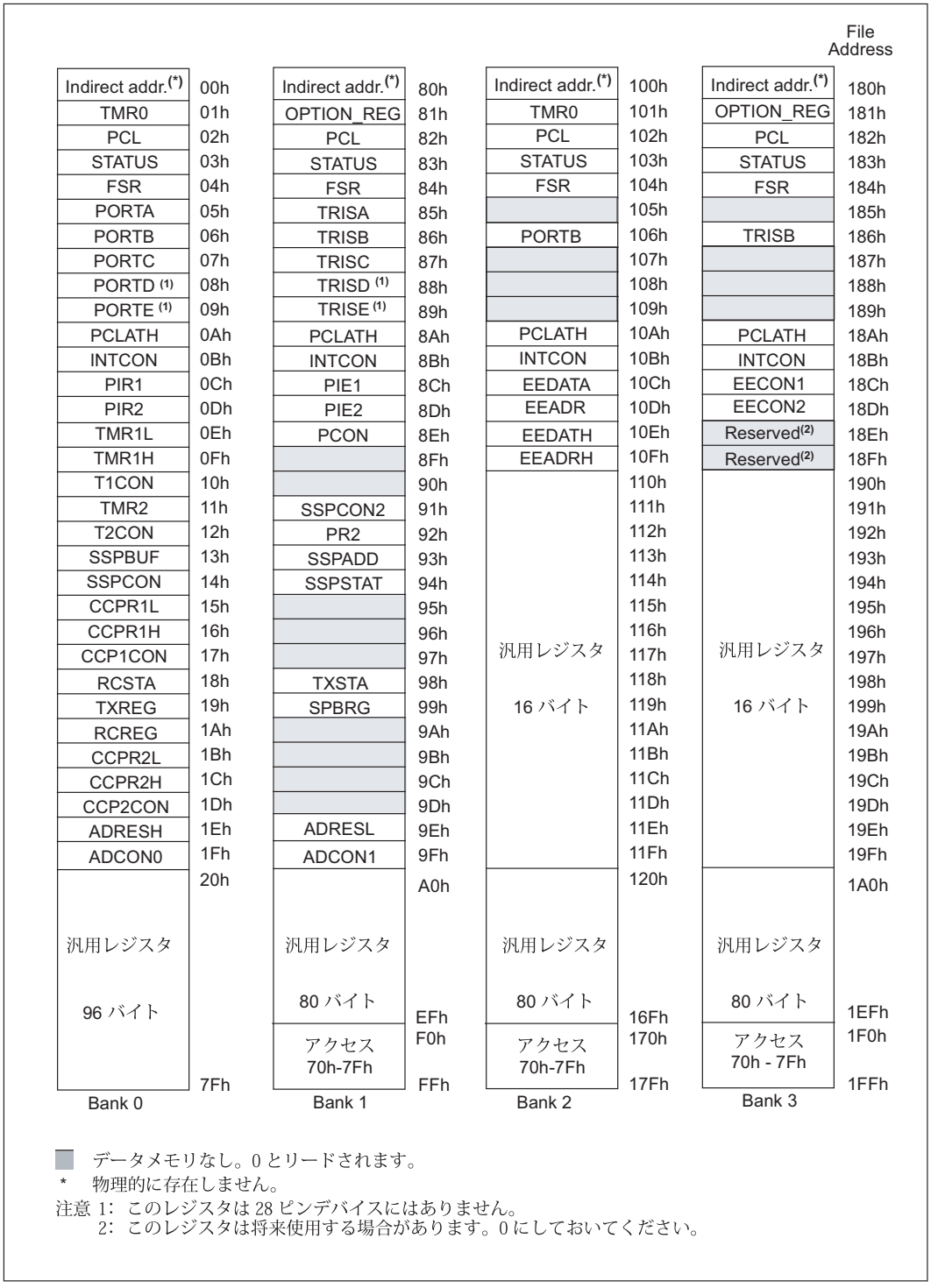


図 12: PIC16F876 のレジスタファイル

8 課題

課題は，ikuo-soft3-08@jsk.t.u-tokyo.ac.jp 宛てに提出する．メールのサブジェクトは，「`soft3-kadai:20081006:xxxxx`」(xxxxx は学生証番号) とする．締切は次回の講義開始時 (10 月 20 日 (月)AM10:15) までとします．その他，質問・要望・意見などがあれば，遠慮なく ikuo@jsk.t.u-tokyo.ac.jp にメールをください．

8.1 課題 1

6.3 節にある課題 1 を実施せよ．

8.2 課題 2

次に挙げる単語で，わからない，あるいは理解が足りないと感じる単語を列挙せよ．さらに，これまでの講義で出てきた単語や関連しそうな単語で，わからない・理解不足と感じる単語も，列挙せよ．

キーワード一覧：
算術シフト，論理シフト，即値アドレッシング，レジスタ直接アドレッシング，レジスタ間接アドレッシング，メモリ直接アドレッシング，メモリ間接アドレッシング，インデックスアドレッシング，スタックアドレッシング，MMU

8.3 課題 3

次に挙げる単語で，わからない，あるいは理解が足りないと感じる単語を列挙せよ．さらに，今日の講義で出てきた単語や関連しそうな単語で，わからない・理解不足と感じる単語も列挙せよ．

さらに，挙げた単語について各自で調査を行い，簡単に説明せよ．調べても良くわからなかった単語は，そのように記載すること．

キーワード一覧：
アセンブリ言語，オペコード，オペランド，ニーモニック，ラベル，機械語，擬似命令，マクロ，引数，ディレクティブ，アセンブラ，コンパイラ，リンカ，サブルーチン，プログラムカウンタ，スタックポインタ，スタックフレーム，プリプロセッサ，マクロプロセッサ，パーザ，字句解析，構文解析，
割り込み，割り込みハンドラ，割り込みベクタ，リターンエクセプション，ステータスレジスタ，インラインアセンブラ，最適化，コマンドライン引数，環境変数，a.out，オブジェクトモジュール，実行可能ファイル，分割コンパイル，セグメント，シンボル表，ロード，再配置，バインド，ダイナミックリンク/スタティックリンク，dll(so)，
"gcc -E"，"gcc -S"，"size"，"nm"，"objdump -d"，"ar"，"gcc -shared"，セクション，.data，.bss，.text