モジュール化アセンブラプログラミング

藤　波　順　久[†]

プログラミング道の原点といえば、アセンブリ言語である。本稿では、すべてアセンブリ言語で書かれた中規模(ソースファイルで数万行程度)のプログラムを開発するために必要となる、モジュール化の概念を紹介する。まず、呼び出し規約を自由に決められるといった、アセンブリ言語の利点を述べる。次に、かつて流行した構造化マクロを紹介する。続いて、モジュール化を効果的に行うために、モジュールや各サブルーチンに書くべきコメントを、例を挙げながら説明する。アセンブリ言語でのテクニックもいくつか紹介する。

Modularized Assembler Programming

FUJINAMI Nobuhisa[†]

If you started to go on the tao of programming, then you know an assembly language. This article introduces the concept of modulalization, which is indispensable to develop medium-scale programs (tens of thousands of lines) written thoroughly in an assembly language. It first describes the merits of assembly languages, e.g. the freedom to decide the procedure calling conventions. Then, control structure macros, once popular, are introduced. It explains, using examples, how to write comments of the modules and the subroutines to make the concept effective. Some programming techniques in assembly languages are also introduced.

1. はじめに

プログラミング道の原点といえば、アセンブリ言語である。アセンブリ言語を使えば、その機種で用意されているすべての機能を自由に使うことができる。また、コンパイラの癖やライブラリの細かい仕様☆1を知らなくてよいため、プログラムはわかりやすい。

本発表☆2で扱うのは、すべてアセンブリ言語で書かれたプログラムである。速度が必要な部分だけアセンブリ言語で書き直すような、中途半端なものではない。後者と比べた前者の利点は、例えば次の通りである。

呼び出し規約を自由に(効率的に)決められる　これはアセンブリ言語の大きな利点である。呼び出し規約をサブルーチンごとに臨機応変に変えて、呼び出しのオーバーヘッドを可能な限り小さくできる。後者では、少なくとも入口と出口では、ABI☆3の関数呼び出し規約に従わなければならない。

極端に小さいプログラム　後者では、スタートアップルーチンや余計なライブラリがリンクされてしまうのを防ぐのは難しい。すべてアセンブリ言語で書けば、この問題は根本的に解決される。

OS非依存プログラム　一部でも高級言語で書かれたプログラムは、うっかり特有のライブラリを使ってしまうなど、特定のOSに依存してしまいがちである。すべてアセンブリ言語で書けば、OSに依存する部分が明確になり、それを避けたり分離したりすることが容易である。

最初の点について補足しておく。デバッガ、プロファイラなど、各種のツールが整った環境では、各ツールがABIに従っていることを前提としているため、そのOS上では常に従わねばならないような幻想にとらわれがちである。実際には、システムコールや共有ライブラリの呼び出しなどだけ従っていればよい。

アセンブリ言語には欠点もある。最大の問題は、プロセッサに依存することである。しかし最近では、インテルの8086の後継やその互換プロセッサ(x86プロセッサ)が使われることが多くなったので、特にパソコン☆4では問題にしなくてよいとも言える。一方、高級言語を使っていたとしても、プロセッサ(と動作環

† 株式会社ソニー・コンピュータエンタテインメント
Sony Computer Entertainment Inc.

☆1 制限事項とも言う。

☆2 本研究は、所属とは無関係に行われた。

☆3 Application Binary Interfaceの略とされるが、多くの場合、System V Application Binary Interface[6]に倣っており、C言語の処理系定義事項や、オブジェクトファイル形式なども規定する。

☆4 Macintoshも2006年からx86に移行する。

境) 非依存にするのは簡単ではない。C 言語では、ヘッダ limits.h、float.h、stdint.h で定義されているマクロを使って注意深くプログラムを書いたり、環境に特有のマクロに対し #ifdef 指令などを使って処理を分けたりする必要がある。Java のように、効率を一部犠牲にして、プロセッサ非依存の実行環境 (仮想マシン) を使う方法もある。いずれにしろ非依存にするのはそれほど簡単ではない。

もう一つの問題は、プログラマ人口が少ないことである。これについては、本発表で増えることを期待する。

本発表では、すべてアセンブリ言語で書かれた中規模 (ソースファイルで数万行程度) のプログラムを開発するために必要となる、モジュール化の概念と、関連する話題を紹介する。本稿ではまず、かつて流行した技術として、小規模のプログラムの開発を効率化する、構造化アセンブラ MACRO を 2節で紹介する。しかしこれだけでは、1万行を超えるプログラムを開発し保守するには不十分である。3節では、アセンブリ言語のモジュール化の概念を説明し、それを実現するためのコメントの書き方を解説する。プログラムが大きくなってくると、途中からプログラムの構成を変更したり、さまざまな環境で同じコードを動かしたりするためのテクニックが必要になってくる。4節では、これらのいくつかを紹介する。最後に5節でまとめを行う。

なお、本発表で紹介する例は、x86 プロセッサ用の Microsoft Macro Assembler、およびその互換アセンブラ用のソースファイルの一部である。しかも、本発表用に作ったのではない、「本物」のソースファイルから取り出したものである。

2. 構造化アセンブラ MACRO

20年ほど前に、アセンブラ MACRO プログラミングの構造化アセンブラ MACRO[1)5)] が流行した。これは、if、while、switch などの制御構造を実現するもので、ループなどで必要なラベル名を考える作業からプログラマを解放し、読みやすいプログラムを書けるようにした。当時はマクロを使って構造化を実現していたが、類似のものが以後のアセンブラ (MASM6.0 など) の擬似命令として採り入れられている。

著者もこれを参考に、独自の改良を加えたマクロを作成した。**図1**は、その使用例である。$REPEAT と $UNTIL は繰り返しを表す。$UNTIL につける終了条件は、フラグを変更する命令と、条件ジャンプの条件 (条件ジャンプのニーモニックから最初の J を除いたもの) で表す。ここでは使っていないが、$REPEAT にはループ継続条件をつけられるし、ループを抜ける $EXIT な

```
PRDEC:
PSH AX,CX,DX,BX
CLR CX
$REPEAT
 CLR DX
 PSH CX
 MOV CX,10
 DIV CX
 PUL CX
 PSH DX
 INC CX
$UNTIL <TEST0 AX>,Z
$IF <SUB BL,CL>,A
 MOV AL,BH
 $REPEAT
  CALL PRCHR
 $UNTIL <DEC BL>,Z
$ENDIF
$REPEAT
 PUL AX
 ADD AL,'0'
 CALL PRCHR
$UNTIL LOOP
PUL BX,DX,CX,AX
RET
```

図1 構造化マクロの使用例 (BPP4A.ASM より)

ども使える。$UNTIL にはまた、特殊な書き方として、ジャンプ命令だけを書くことができ、無限ループなどを表すのに使う。図1の最後のループは、ジャンプ命令として LOOP 命令を使ったものである。$IF と $ENDIF は条件つき実行である。$ELSIF、$ELSE も使える。

本稿の例では、以後もこの構造化マクロを使用する。また、独自に定義したマクロとして、TEST *reg,reg* に展開される TEST0 *reg*、XOR *reg,reg* に展開される CLR *reg*、条件つきリターン RET *cond* なども使用する。

構造化マクロによって、制御構造はわかりやすくなった。次は手続き呼び出しである。しかしながら、既存の構造化 MACRO や、アセンブラ擬似命令は、高級言語の呼び出し規約をそのままサポートする方向に行ってしまい、アセンブリ言語のメリットをあまり生かしていない。例えば、アセンブラ MACRO プログラミングでは、標準入出力[2)]、文字列操作[3)]、C 言語関数呼び出し[4)] と進んだ。TASM などのアセンブラも、擬似命令で C や Pascal の手続き/関数呼び出しをサポートする。

3. モジュール化とコメント

本発表でモジュールと呼んでいるのは、一つのオブジェクトファイルにアセンブルされるソースファイルのことである。各モジュールは、PUBLIC 擬似命令などを用いて、他のモジュールに公開するサブルーチンや変数などのシンボルを定義し、また、EXTRN 擬似命令などを用いて、他のモジュールの提供するシンボルを参照する。**図2**は、シンボルを参照するモジュールが使うためのインクルードファイルの例である。

本節では、モジュールが公開するサブルーチンに必

```
.XLIST

_BSS SEGMENT

EXTRN BIO_FLAG:BYTE

_BSS ENDS

_TEXT SEGMENT

EXTRN INIT_BIO:NEAR
EXTRN EXIT_BIO:NEAR
EXTRN TEST_CHR:NEAR
EXTRN COLOR2ATTR:NEAR
EXTRN PHY_SCROLL_UP:NEAR
EXTRN PHY_SCROLL_DOWN:NEAR
EXTRN PHY_SCROLL_LEFT:NEAR
EXTRN PHY_SCROLL_RIGHT:NEAR
EXTRN PHY_CLS:NEAR
EXTRN SAVE_TEXT:NEAR
EXTRN LOAD_TEXT:NEAR
EXTRN SCREEN_SWITCH:NEAR
EXTRN DISPLAY_CURSOR:NEAR
EXTRN ERASE_CURSOR:NEAR
EXTRN FUNCTION_KEY:NEAR
EXTRN GET_KEY:NEAR

_TEXT ENDS

.LIST
```

図2 モジュールBIO.ASMが公開するシンボルを他から参照するためのインクルードファイルBIO.EXT

ず必要になる、入出力条件のコメントの書き方を述べた後、モジュールのコメントの書き方を紹介する。

3.1 サブルーチンの入出力条件

呼び出し規約を自由に決められるというアセンブリ言語の利点を、最大限に生かすためには、サブルーチンごとに呼び出し規約を明示する必要がある。手続き型言語では、各手続きについて、機能、引数、戻り値に対するコメントを書くのが普通である。アセンブリ言語では、引数と戻り値はレジスタ名まで書く必要がある。その代わりに、戻り値は複数使用できる。さらに、値が破壊されるレジスタを書くことで、呼び出し規約についてのコメントが完成する。

図3は、サブルーチンとそれの入出力条件のコメントの例である☆。コメントは、原則として以下のように書く。

入力 レジスタ(場合によってはメモリ上の変数)名とそれに入っているべき値の説明を書く。値を使用しない(どんな値が入っていてもよい)レジスタは書かない。

出力 レジスタ(場合によってはメモリ上の変数)名とそれに返される値の説明を書く。値が不明(有用でない)の場合は「破壊」と書く。値が保存されるレジスタは書かない。

場合分け 特定の場合(操作成功など)にだけ値が設定

☆ 印刷では白黒だが、レジスタ名に色付けが可能なエディタで見るほうが見やすい。

```
;Color to Attribute
;入力：ALに色番号
;      LSB-B,R,G,reverse,blink,underline,ve
rticalline,*-MSB
;出力：AXに属性
        PUBLIC COLOR2ATTR

COLOR2ATTR:
ROR AX,1
ROR AX,1
ROR AX,1
AND AX,0E00FH
$IF <TEST AL,03H>,PO
 XOR AL,03H
$ENDIF
SHL AL,1
INC AX
OR AL,AH
CLR AH
RET
```

図3 色番号を入力とし、機種に特有の属性値を計算するサブルーチン(BIO.ASMより)

```
;Get Key
;入力：なし
;出力：NZならALに文字、Zならなし、AHは破壊
        PUBLIC GET_KEY

GET_KEY:
PSH DX
MOV DL,0FFH
F_CONOUT
$IF ,NZ,OR
INT 21H ; F_CONOUT
$C ,NZ
 $IF <TEST0 AL>,Z
  INT 21H ; F_CONOUT
  $IF ,NZ,AND
  MOV DH,AL
  INT 21H ; F_CONOUT
  $C ,NZ
   ROL AL,1
   ROL AL,1
   ROL AL,1
   ROL AL,1
   OR AL,DH
   TEST0 DL
  $ENDIF
 $ENDIF
$ENDIF
PUL DX
RET
```

図4 キーボードから文字が入力されていたら、それを返すサブルーチン(BIO.ASMより)

されるレジスタは、場合分けして説明する。後述するように、操作成功などの条件は、フラグレジスタの特定の値で示されることもある。

空の場合 入力とする値がない場合、あるいは出力となる値や破壊されるレジスタがない場合、「なし」と書く。

メモリの場合 特定のメモリ番地の値を入力または出力とする場合、レジスタ名の代わりに[BMP_WIDTH]のような角括弧で囲んだ表現を使うことがある。

図4は、入力が空で、出力で場合分けをしている例である。文字が入力されているかどうかは、ゼロフ

```
;Print Hex
;入力：DX:AXまたはAXまたはALに数値
;出力：なし

PRHEX8:
XCHG DX,AX
CALL PRHEX4
XCHG DX,AX
PRHEX4:
XCHG AH,AL
CALL PRHEX2
XCHG AH,AL
PRHEX2:
PSH AX
SHR AL,1
SHR AL,1
SHR AL,1
SHR AL,1
CALL PRHEX1
PUL AX
PRHEX1:
PSH AX
AND AL,0FH
CMP AL,0AH
SBB AL,69H
DAS
CALL PRCHR
PUL AX
RET
```

図5 レジスタの値を16進数で表示するサブルーチン (LHAHDR.ASMより)

ラグが降りている (NZ) か立っている (Z) かで示されている。

同じようなサブルーチンを並べて定義している場合、コメントをまとめてしまってもよい。**図5**はそのような例である。ここで、DX:AXとは、DXを上位16ビット、AXを下位16ビットとする32ビット数を表す。

3.2 特別扱いが必要なレジスタ (x86の場合)

プロセッサによっては、通常のコメントの書き方では煩雑になるような、特殊な使い方のレジスタを持っていることがある。x86の場合には、次のようなものがあり、コメントの書き方に工夫が必要になる。

フラグレジスタ 多くの演算命令で暗黙の出力レジスタとなっており、サブルーチンの出口での値は有用でない (原則に従えば「破壊」と書くべき) ことが多い。それを毎回書くのは煩わしいので、何も書かなければ破壊されるとみなす。変更されないときは「保存」と書く。フラグでエラー、ステータスなどを返すときは、それぞれ説明を書く。その際、フラグレジスタの各ビットに対して、次のような略語を使う。

C キャリー (Carry) フラグの値、またはキャリーフラグが立っている (C=1) こと

NC キャリーフラグが降りている (C=0) こと

他のフラグ (P: Parity, Z: Zero, S: Signなど) も同様である。コメントに書かれていないフラグは破壊されるとみなす。なお、ディレクション (Direction) フラグ☆だけは、モジュール全体で常に0と決めておく (1にしたらすぐに戻す) など、別の方法に従うことがある。

```
;Test Character
;入力：AXにVRAMの値
;出力：1バイト： Z,NC,NS
;       半角：    Z, C,NS
;       全角左：  NZ, C,NS
;       全角右：  NZ, C, S
        PUBLIC TEST_CHR

TEST_CHR:
$IF <TEST0 AH>,NZ
 $IF <CMP AL,09H>,AE,AND
 $C <CMP AL,0CH>,B
  CMP AL,AL
  STC
  RET
 $ENDIF
 TEST0 AL
 STC
$ENDIF
RET
```

図6 VRAMに書かれている文字の種類をフラグで返すサブルーチン (BIO.ASMより)

```
;Check for Print

CHECK:
PSH AX,DX,DI
READCHR
CALL TEST_CHR
$IF ,NZ
 $IF ,S
  CALL ERASE_LEFT
 $ELSE
  CALL ERASE_RIGHT
 $ENDIF
$ENDIF
PUL DI,DX,AX
RET
```

図7 図6のサブルーチンを使って、半角文字の表示前に余計な文字を消すサブルーチン (AIO.ASMより)

セグメントレジスタ ☆☆ モジュールで共通に決めたものは、サブルーチンの入力条件としては省略する。

スタックポインタ スタックの残り容量についての記述は、特に含めない。再帰呼び出しなどがあるときは、適当な箇所でチェックするなど、コメントとは別の方法で対処する。

部分レジスタ ☆☆☆ 一部だけに値が設定されるときは、そのレジスタ名を書く。32ビットレジスタの上位16ビットには名前がないが、EAX-AXのように書くことにする。

図6は、フラグを複数使って値を返す例である。文

☆ ストリング操作命令を番地が増加する方向に使うときに0、減少する方向なら1を設定する。

☆☆ x86のアドレスはセグメントとオフセットの組で指定するが、そのセグメントアドレスを入れるレジスタ。

☆☆☆ 例えば32ビットレジスタEAXの下位16ビットがAXで、AXは8ビットのAHとALから成る。

```
;Cons 構成
;入力：ESI に CDR セルの番地、EDI に CAR セルの
番地
;出力：EBX にセルの番地
;       C にエラー

        PUBLIC CONS
CONS:
...

;Get Array Element 配列からの要素の取り出
し
;入力：EBX に配列セルの番地、EAX に添字
;出力：EDI に要素
;       C に 0

        PUBLIC GET_ARRAY_ELEMENT
GET_ARRAY_ELEMENT:
...

;Array Size 配列の大きさ
;入力：EBX に配列セルの番地
;出力：ECX に要素数、EAX に最初の添字

        PUBLIC ARRAY_SIZE
ARRAY_SIZE:
...
```

図 8 リスト処理のサブルーチン (cons セルを返す、配列から要素を取り出す、配列の大きさを返す)(CELL.ASM より)

```
XTOLIST:
PUL EBX
CALL LIST?
RET NC
...
CALL ARRAY_SIZE
ADD EAX,ECX
CLR ESI
$REPEAT <$EXIT JECXZ>
 DEC EAX
 CALL GET_ARRAY_ELEMENT
 PSH EBX
 CALL CONS
 MOV ESI,EBX
 PUL EBX
 RET C
$UNTIL LOOPD
MOV EBX,ESI
RET
```

図 9 引数をリストに変換する (ここでは引数が配列の場合以外は省略)(OBJECT.ASM より)

字の種類をフラグに反映する際、VRAM の値は 2 バイト文字なら上位 8 ビットだけでなく下位 8 ビットも 0 以外であること、全角文字の右半分ならビット 7 が立っていることを利用している。フラグを設定するための特別な命令 (STC など) はあまり使われていない。というよりはむしろ、あまり使わなくてすむように、出力のフラグの組み合わせを決めておくのが普通である。**図 7** は、フラグで返された値を使う例である。汎用レジスタで値を返した場合と異なり、すぐに条件ジャンプ命令を使うことができる。

図 8 は、フラグレジスタを含めて、複数のレジスタに値を返すサブルーチン (本体は省略) の例である。入力や出力のレジスタをうまく選んでおくことで、使う側は引数をコピーする回数を減らすことができる。**図 9** は、これらを使って配列をリストに変換するコード片である。ARRAY_SIZE に与えた配列セルの番地 EBX は、GET_ARRAY_ELEMENT にそのまま与えることができる。ARRAY_SIZE の戻り値のうち要素数 ECX は、そのままループカウンタとなる。リストは逆順 (最後の要素から最初の要素へ向かう順番) で構成するため、ARRAY_SIZE のもう一つの戻り値である最初の添字 EAX に要素数 ECX を足してから 1 ずつ減らして、GET_ARRAY_ELEMENT の添字として使っている。GET_ARRAY_ELEMENT の戻り値 EDI は、そのまま CONS の引数の一つ (CAR 要素) となる。CONS がエラーを返したら、このコード片もエラー (C) を返す。

3.3 モジュールのコメント

各モジュールには、それが提供する機能の概略と、公開するサブルーチンに共通する規約を書く。それには、共通して使うデータ構造の説明、前節で紹介したような特殊なレジスタの使い方の習慣、外部に必要とするサブルーチンや変数などが含まれる。**図 10** は、そのようなコメントの例である。マーク・スイープ方式のガベージコレクションを行うため、外部に必要とするサブルーチンの説明が複雑になっている。

ソース中のコメントで扱いきれるのは、モジュール以下の粒度までである。モジュールが複数集まって構成される部分 (著者らは例えば、カーネル部、エディタ部のように呼ぶ) の規約については、モジュール構成の説明を含めて、コメントではなく別に文書を書き起こすほうが適切と考えている。**図 11** は、別文書で書かれたカーネル部の説明の例である。この文書を読めば、プログラムが MS-DOS での拡張メモリの管理方法 4 種類すべてに対応することや、そうするためのコーディング上の注意点などがわかる。なお、ここで参照している ALLOC_SEG ルーチンは、次節で例として紹介されている。

4. 互換性テクニック

プログラムの開発中には、規約を変更したくなることがある。小規模な変更、例えば数箇所から呼ばれているサブルーチンの引数レジスタの変更程度であれば、関係箇所をすべて検索して直すこともできる。しかし、複数のモジュール全体で共通して使っている規約であれば、そうはいかない。

このような場合、高級言語でよく行うのと同様に、両方の規約が共存できる環境にまず移行し、次に新しい規約だけの環境に移行する方法が良く用いられる。規約を共存させるのは、きちんと説明すると意外とややこしくなりがちだが、それほど難しいことではない。そのようなテクニックは、複数の環境で動くプログラ

```
; 外部に必要とする変数

;STACK_BTM: スタックポインタの初期値

; 外部に必要とするルーチン

;CHECK_STACK: スタックの残りが十分か調べる
;入力：なし
;出力：Cにエラー

;CELL_ERROR: セル領域不足のエラーメッセージを表示する
;入力：なし
;出力：Cに1

;MARK_OTHER: 変数など以外のデータをマークする
;入力：なし
;出力：EAX,ECX,EBX,ESI,EDIは破壊
; ルーチンでは、EBXにマークするセルの番地を入れてMARK_CELLを呼ぶことを、必要な
;だけ繰り返す。
; 内部でセルをアクセスするときには、GCM（CAR）またはGCX（ADR）が立っていること
;があるのでマスクする必要がある。
; ルーチン内では、ヒープの大きさを縮小するためにRESIZE_HEAPを呼んでもよい。
;GET_ARRAY_ELEMENTとARRAY_SIZEで配列を参照してもよい。
; MARK_OTHERが呼ばれるときは、ESはDSと等しくなっている。MARK_CELLを呼ぶときも、
;リターンするときも、これを変えてはならない。DGROUPが必要ならFSを使う。

;<<< Cell Format >>>

; ################################# #################################
;                                 ||  |                               |
;                                 ||  GC eXtended Mark               |
;                                 |GC Mark                           |
;                                 0:Heap                             0:Heap
;                                 1:Cons                             1:Cons
; ------------Hash/Tag------------ *-------------Address------------
; --------------CAR-------------*** ---------------CDR-------------***
```

図10　リスト処理モジュールのコメント(一部)(CELL.ASMより)

```
;Allocate Segment
;入力：AXにセグメントアドレス
;出力：AXにコードセグメントセレクタ、BXにデータセグメントセレクタ
;       Cにエラー

        PUBLIC ALLOC_SEG
ALLOC_SEG:
$IF <CMP MMTYPE,4>,E
 ...
 RET
$ENDIF
$IF <CMP MMTYPE,3>,E
 ...
 RET
$ENDIF
MOV BX,AX
CLC
RET
```

図12　現在のモードに適したセグメントセレクタを返すサブルーチン(CONFIG.ASMより)

ムを作ったり、ツールの想定とは異なる環境で動作するプログラムを作ったりするのにも役立つ。

複数の環境の例として、リアルモード☆1とプロテクトモード☆2の両方で動くプログラムを書くためのサブルーチンを**図12**に紹介する。メモリ管理の種類を表す変数MMTYPEの値は、図11の4つの方式に対応している。3または4なら、VCPIまたはDPMIであり、プログラムはプロテクトモードで実行されている。サブルーチンは、メモリ管理に応じた方法で、必要ならセグメントを割り当て、セグメントセレクタを取得して返す。このサブルーチンがあれば、リアルモード専用だったプログラムを、プロテクトモードでも動くように書き換えることができる。

次に紹介するのは、16ビットコードで64KBの壁があった頃に有効だったテクニックである。プログラムが大きくなって、一つのセグメントに入りきらなくなると、複数のセグメントに分けなければならない。そのとき、すべてのサブルーチンをnear☆3からfar☆4に変えるのはたいへんなので、多くの場合にnearですむようにうまく分ける。それでも残る、異なるセグメントからの呼び出しには、nearサブルーチンに皮をかぶせてfarサブルーチンに見せかけたり、呼び出し元のセグメントのnearサブルーチンに見せかけたりする。

図13は後者の例である。マクロ定義で作った見

☆1 x86で、初代の8086と互換性のあるモードである。セグメントレジスタの値が16倍されて物理アドレスの計算に使われる。

☆2 セグメントの保護が使えるモードで、セグメントレジスタの値(セグメントセレクタ)は、セグメントの番号と特権レベルを表す。

☆3 同じセグメントから呼ばれるサブルーチンの属性で、nearリターン命令で終わる。

☆4 異なるセグメントから呼ばれるサブルーチンの属性で、farリターン命令で終わる。nearコール/リターンより時間がかかる。

```
FAR_TEXT SEGMENT DWORD PUBLIC 'CODE' USE16

ASSUME CS:FAR_TEXT,FS:DGROUP

TO_NEAR MACRO ADDR
        PUBLIC FAR_&ADDR
 FAR_&ADDR:
 MOV TO_NEAR_OFFSET,OFFSET ADDR
 JMP TO_NEAR_CALLER
ENDM

TO_NEAR CONS
TO_NEAR EQUAL
TO_NEAR LPUT
TO_NEAR APPEND
TO_NEAR CELL
TO_NEAR MKATOM_OPEN
TO_NEAR MKATOM_WRITE1
TO_NEAR MKATOM_CLOSE
TO_NEAR SEARCH_ATOM

TO_NEAR_CALLER:
CALL TO_NEAR_ADDR
RET

TO_NEAR MKARRAY
TO_NEAR GET_ARRAY_ELEMENT
TO_NEAR SET_ARRAY_ELEMENT
TO_NEAR ARRAY_SIZE
TO_NEAR MKHEAP
TO_NEAR RESIZE_HEAP
TO_NEAR CHECK_HEAP
TO_NEAR GCOLL
TO_NEAR SCAN_CELL

FAR_TEXT ENDS
```

図13 CELL.ASMのサブルーチンに皮をかぶせる(FARCELL.ASMより)

```
;Macro Definition

SET_BASE_ADDR MACRO
 LOCAL RIP_BASE
 DB 48H,8DH,2DH ; LEA RBP,[RIP+0]
 DD 0
RIP_BASE:
 REX_W <SUB RBP,OFFSET RIP_BASE>
ENDM
```

図14 RBPに値を設定するマクロ定義(MONITOR.ASMより)

せかけのnearサブルーチンは、メモリに呼びたいサブルーチンのオフセットアドレスを書き込んで、TO_NEAR_CALLERにジャンプする。するとそこの命令CALL TO_NEAR_ADDR(1)が、呼び出し先のセグメントにあるfarサブルーチンTO_NEAR_CALL(2)を呼び、それがさらに、目的のnearサブルーチンを、先ほどメモリに書いたアドレスを使って呼ぶ。なお、(1)と(2)で名前が異なっているのは、(1)は実はメモリ上の変数で、呼び出し先のセグメントセレクタを図12のALLOC_SEGルーチンで計算して、(2)のオフセットとともに、あらかじめ(1)に書いておくためである。

最後に、AMD64アーキテクチャ☆のプログラムを、x86にしか対応していないアセンブラで書くためのテクニックを紹介する。AMD64では、アドレスは64ビットであるが、64ビットのアドレスを直接指定してメモリを読み書きできる命令は限られている。x86で32ビットアドレスを直接指定する命令はたいてい、AMD64ではRIP(プログラムカウンタ)相対と解釈されてしまう。そこで、RBPに適当な値を入れておき、レジスタ間接でメモリにアクセスすることにする。RBPに入れるべき適当な値とは、アセンブラが思っているアドレスと実際のアドレスの差である。**図14**のマクロを使えば、プログラムを好きな番地にロードしても、正しく設定される。**図15**では、このマクロを使ってスタックポインタを初期化したり、TSSに値を設定したりしている。

☆ x86をほぼそのまま64ビットに拡張したもので、64ビットレジスタはRAXなどの名前になる。IntelのEM64Tも、これとほぼ互換である。

```
;RBPには常に、セグメントベースアドレスを入
れておき、オフセットアドレスをリニア
;アドレスに変換するために使う

        PUBLIC START
START:
CLD
SET_BASE_ADDR
;スタック
REX_W <LEA RSP,STACK0_BOTTOM[RBP]>
;TSS
REX_W <MOV QWORD_ PTR TSS[04H][RBP],RSP>
REX_W <MOV QWORD_ PTR TSS[0CH][RBP],RSP>
REX_W <MOV QWORD_ PTR TSS[14H][RBP],RSP>
MOV WORD PTR TSS[66H][RBP],68H
```

図15 64ビットのスタックポインタとTSSの初期化(MONITOR.ASMより)

5. ま と め

アセンブリ言語でプログラムを書くのは楽しい。高級言語のプログラマには隠されている、すべての情報を見ることができるし、効率的なプログラムを書く努力は、できあがるプログラムコードに直接反映される。

アセンブリ言語はまた、意外と長寿命である。高級言語の変遷と比べて、プロセッサファミリの命令セットの変化がそれほど多いわけではないし、もちろん互換性(それもバイナリレベルの互換性)は普通保証される。30年後にx86とC言語のどちらが生き残っているかは容易に断言できない。

そしてもちろん、アセンブリ言語を使えば、まだ高級言語の処理系が用意されていないプロセッサや、高級言語からは利用の難しい拡張機能など、最先端の技術をすぐにでも利用することができる。

皆さん、アセンブリ言語を使いましょう。

質疑応答(敬称略)

浜田 昔アセンブリ言語を使ったときは、開発効率が悪いと思ったが、この発表では、アセンブリ言語の見方がずいぶん違う。

２　カーネル
２．１　CONFIG.ASM
CONFIG.ASMでは、CONFIG.3Dの処理のほか、CPUのチェック、メモリの取得、浮動小数点演算環境の設定などを行う。
２．１．１　メモリ管理
メモリ管理はBIOSワークエリア（ドライバなし）、XMS、VCPI、DPMIの４つの方式に対応しており、後のものから優先して使用する。CPUが仮想8086モードのときには、VCPIとDPMIのみが使用できる。BIOS、XMSでは、4GBまでアクセス可能に設定したリアルモードで実行される。VCPI、DPMIではプロテクトモード（16ビットセグメント）で実行される。
4GBまでアクセス可能に設定したリアルモードとは、セグメントリミットに4GBを設定したままプロテクトモードからリアルモードに戻ったときに起きる特殊な状態である（仮想8086モードではこの状態にできない）。実際には、セグメントリミットを越えてメモリアクセスしたときに起きる割り込み0Dh（擬似保護例外）のハンドラでこの設定を行っているため、レジュームなどでセグメントリミットが元に戻ってしまっても動作を続行できる。
(中略)
２．２　コーディング上の注意点
ここで、カーネル部のコーディングで注意するべき点を述べる。メモリ管理や浮動小数点演算がどの方式であっても動くようにするために、注意、制限があったり、CONFIG.ASMのルーチンや変数を使う必要があったりする。
２．２．１　セグメント
セグメントセレクタは、直接計算してはいけない。DGROUPならWORK_SEGを、_TEXTならCSを、_STACKならSSを、32ビットアドレス空間ならFLAT_SEGを用いる。これ以外のセグメントが必要なら、ALLOC_SEGルーチンを呼ぶ。

図11　カーネル部の説明(一部)

著者　ありがとうございます。

湯浦(日立製作所)(座長)　書く前に規約を考えているところが楽しいのではないか。

著者　書きながら呼び出し規約を改良していくところが楽しい。

近山(東大)　リンカが好きなことをさせてくれない。外部参照アドレスを2bitシフトした値を使いたいことがあって困った。

著者　普通は実行時にシフトして使うが、どうしてもいやなら自己書き換えという方法もある。

多田(電通大)　人々はなぜアセンブラを使わないのかと思った。

著者　私の布教が足りなかった。

和田(IIJ)　Knuthは実行時間を正確に、$O(n)$の係数まで見積もるために、アセンブラ(MIX)を使う。Knuthも布教をがんばって欲しい。組み込みは?

清木(任天堂)　最近はC言語。

和田　堕落した。

紀(KLS研究所)　IBMの370にも構造化マクロがあり、抽象度を上げていくと、370の命令を書かなくてよくなった。そこで他の機種に持って行こうとしたが、アセンブラに依存していた。アセンブラも書いてはどうか。

著者　アセンブラも作っている。

和田　森口先生のマシン独立アセンブラSIPはどこでも動く。しかし能率が悪いので機種別になった。

参 考 文 献

1) 編集部, 西村卓也. アセンブラMACROプログラミング(1) 構造化アセンブラMACRO (Control Logic Structure Macro)をつくる, 月刊アスキー, Vol.10, #7, pp. 165–170, 1986年7月号.
2) 編集部, 西村卓也. アセンブラMACROプログラミング(2) 標準入出力MACRO (Standard Input/Output Macros)をつくる, 月刊アスキー, Vol.10, #8, pp. 213-218, 1986年8月号.
3) 編集部, 西村卓也. アセンブラMACROプログラミング(3) データ形式変換と文字列操作, 月刊アスキー, Vol.10, #9, pp. 197–198, 1986年9月号.
4) 西村卓也, 寺沢任弘. アセンブラMACROプログラミング 第4回 C言語インターフェイスMACROS, 月刊アスキー, Vol.10, #10, pp. 189–192, 1986年10月号.
5) 矢部和博. アセンブラMACROプログラミング第5回 構造化アセンブラMACROによるRAMディスクドライバ, 月刊アスキー, Vol.10, #12, pp. 209–213, 1986年月12号.
6) *SYSTEM V APPLICATION BINARY INTERFACE Edition 4.1*, The Santa Cruz Operation, Inc. and AT&T, 1997. (http://refspecs.freestandards.org/)