ICOT Technical Memorandum: TM-0161

TM-0161

汎用型アセンブラの PSI への移植

立野裕和 (三菱電機)
高木茂行 (ICOT)

March, 1986

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

1．はじめに

逐次型推論マシンPSI の開発用に作成されたDEC 10 prolog で記述されている汎用アセンブラをPSI 上に移植した。移植したアセンブラはICOTのDEC 2065においてLoad factar 2.0 程度（TSS 利用で利用者が少なく計算機に負荷がほとんどない状態）でアセンブルを実施したのとほぼ同程度の[注)] 応答速度が得られた。移植に要した工数は約３人月である。

本稿の目的は、移植作業で得られた知識を紹介することである。まず、移植前後の汎用アセンブラを示す。さらに汎用アセンブラの構成について述べる。次ぎに移植の方針及び移植上の問題点と解決策について述べる。移植したアセンブラのPSI 上での実行性能を示し、最後に所感を述べる。

注）応答速度とは、アセンブラを起動して、アセンブルするマイクロプログラム名を入力した時刻から、アセンブルが終了した時刻までの経過時間。

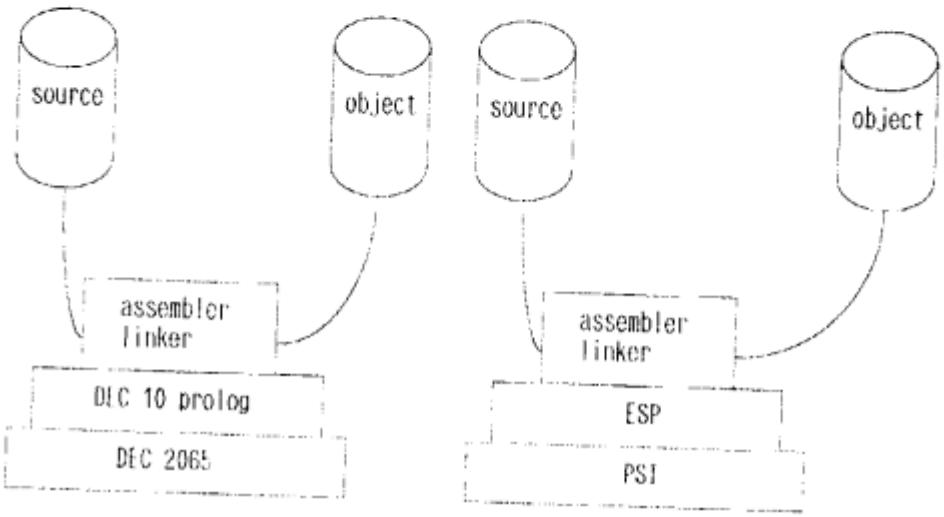
2．汎用アセンブラ概説

2.1　記述言語及びハードウェア

もとの汎用アセンブラはICOT DEC　2065上にDEC 10 prolog で記述されたものである。

それに対し、移植後の汎用アセンブラは、PSI 上で動くものである。以下にこれらの概念図を示した。

図1　移植前後の概念図



2.2　汎用アセンブラ概説

今回移植したアセンブラは従来の汎用アセンブラと同様にアセンブラ本体（核部）と機械定義部から構成される。機械定義部を各々のハードウェアに合わせて定義することにより、ハードウェアに対応した専用アセンブラとなる。図2にその概略を示す。

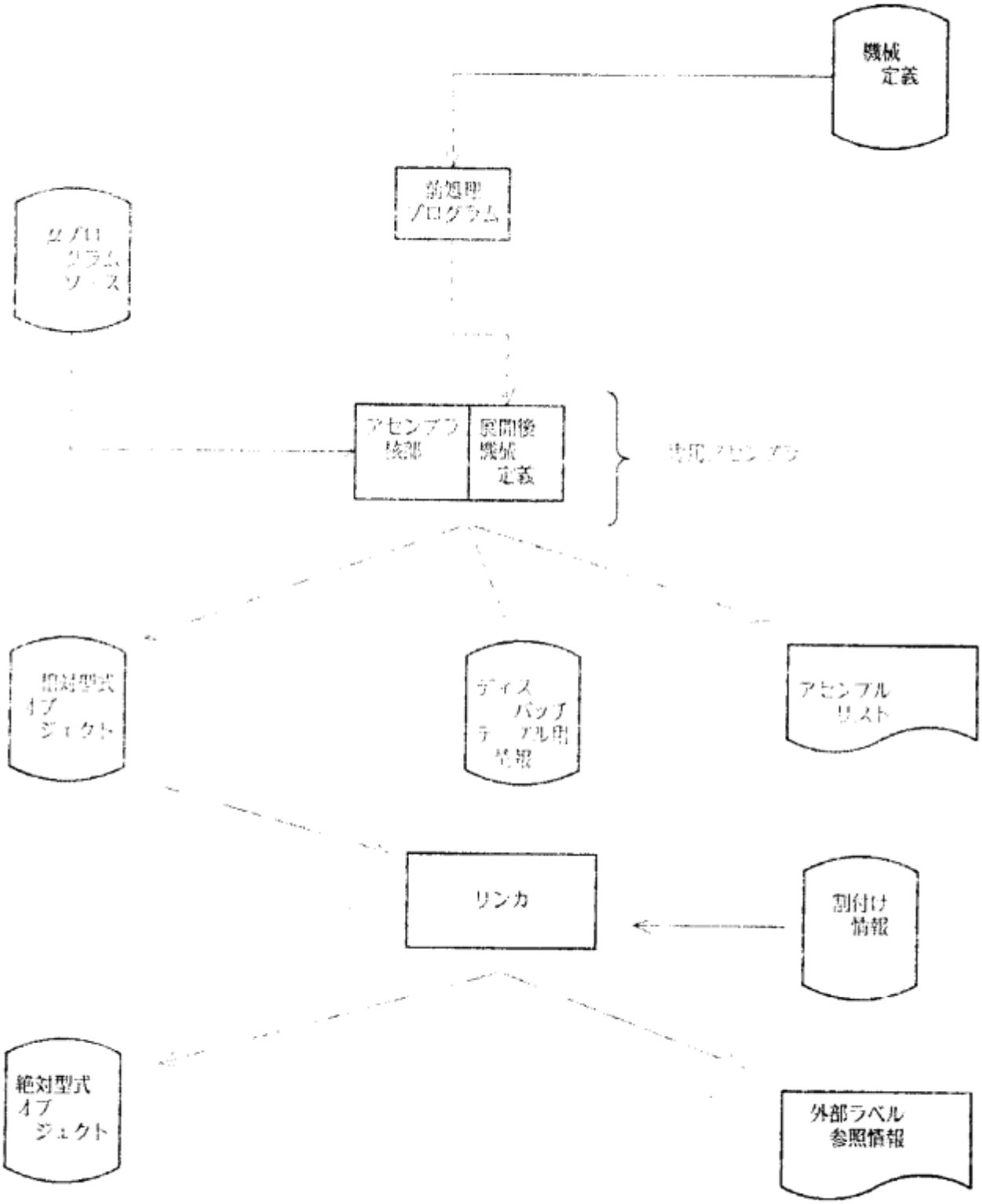


図2　汎用型アセンブラの構成

まず、機械定義（又は機械定義テーブル）を決められた形式で記述する。この機械定義テーブルは、Prolog特有のバックトラック、ユニフィケーションの機能を活用することで、従来の汎用アセンブラでは難しかったエラーの検出などを容易にもり込ませることができる。

この機械定義テーブルを前処理プログラムに入力することで機械定義部が得られる。これとアセンブラ核部を合わせて専用アセンブラが得られる。
移植の対象は、前処理プログラム、アセンブラ核部とリンカである。各プログラムは表１に示したモジュールから構成される。

表１　モジュールリスト

プログラム名	モジュール名
前処理プログラム	GEN．　SUB
アセンブラ核部	ASM．　SUB
リンカ	LK　，　SUB

３．移植

3.1　prologとESP の相違

両者の違いをコーディングレベルで比較すると、

- prologにはassert/retruct系の組込述語による述語の追加／削除機能がある。
- prologにはESP でサポートされているような強力なマクロ展開機能がない。

上記２点に集約できよう。

3.2　移植の基本方針とクラス構成

移植にあたってクラスの構成、assert/retruct系の述語の扱い等に関して次のような方針を決めた。

(1)　表１のモジュールを１個のクラスとした。各モジュール内でpublic宣言されている述語は、他モジュールから参照される。ESP においては、他クラスから参照されることになるので、public宣言されている述語は、instance method とした。public宣言されていない述語はモジュール内でしか参照されないので、ESP においても同様なlocal 述語とした。

このように切りわけることで、prolog→ESP への移植はほとんど機械的に実施できる。

(2)　各モジュール内でのassert/retruct系述語の使われ方は、大域変数の記憶用として扱われている。ESP では、assert/retructに相当する汎用性のある述語追加機能はない。

しかし、ESP では、スロットと呼ばれる大域変数の記憶場所がありtermのよう

なスタックベクタ以外のデータはスロットに保持できる。従ってassert/retructされているﾃﾞｰﾀは、スロットに保持させるか又は、SIMPOSが提供しているプール・サブシステム（ここで述べたスロットを利用し、リストやテーブル等の登録、検索、変更などの機能を提供する）中の適当な機能を利用した。

(3) prologのプログラム内での、入出力はすべて、term単位でおこなわれている。SIMPOSにおける、term処理はトランスデューサ・サブシステムでおこなわれる。従って入出力は、同サブシステムの提供している「クラス　character_file_input_tap　character_file_output_tap」でおこなうことにした。

(4) 機械定義テーブルは、prolog版の前処理プログラムのために作成されたものをそのまま、移植した前処理プログラムで使えるように考えた。

以下にクラス構成を示す。

表2 前処理プログラム構成

表3 専用アセンブラ構成

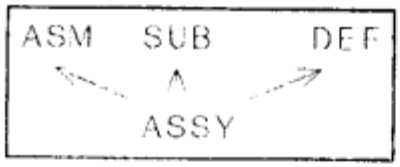


表4 改良した専用アセンブラ構成

クラス名	
GEN	表1のGENを移植したクラス。 前処理プログラムの本体。
SUB	表1のSUBを移植したクラス。 アセンブラ全体のサブルーチン群。
TOP	GENとSUBを継承しているだけのクラス
ASM	表1のASMを移植したクラス。 アセンブラプログラムの本体。
DEF	前処理プログラムに機械定義テーブルを 入力して得られるクラス。
ASSY	ASM、SUB、DEFを継承しているだけのクラス。
f_ASM	後で述べる理由で、ASMを変更したクラス。
f_SUB	後で述べる理由で、SUBを変更したクラス。
EQU_LIST	後で述べる理由で、新設されたクラス。
f_ASSY	f_ASM、f_SUB、EQU_LIST、 DEFを継承しているだけのクラス。

表5　クラス概要

3.3　オペレータ宣言

機械定義テーブル及びアセンブラ核部のコーディングのために、それぞれオペレータが宣言されている。prologでは、プログラム中に自由にオペレータを宣言しかつ宣言した時点でそのオペレータは有効となる。従って、プログラムの先頭でオペレータ宣言することでそのオペレータをプログラム中に記述することができる。又そのプログラム中でfileと入出力をする場合、プログラムの先頭で宣言されたオペレータも含めてtermの読み込み、書き出しが可能である。

SIMPOS 1.53 版では、ユーザープログラム中にユーザ定義のオペレータを含んでいるものを、ライブラリへ登録できない（故にコンパイルが不可能）（ 2.0版でサポートを予定）。

一般にユーザーが任意に定義したオペレータを含むfileをterm形式で読み込むためには、以下の手続きを必要とする。

クラスas_character _file_input _tap のオブジェクトを作成する。このオブジェクトに対して、必要なオペレータを宣言する。（ここで述べる「オブジェクト」はオブジェクト指向の「オブジェクト」でありフォートラン等の「オブジェクト」ではない。）

例

```
fileのopen  <------ :open(#binary__file,File,"Myfile");
タップの生成 <----- :create(#chracter__file__inputtap, Tap,File,512);
オペレータの宣言 <-- :add__operator(Tap,('operator'),xfx,500);
```

必要なオペレータをadd することでオペレータ宣言と同等な結果が得られる。

以上の手続によって、機械定義テーブルやマイクロプログラムが正しく読み込めるようになる。

3.4 マクロ展開

ESP とprologではマクロ展開に関するsyntaxが多少異なる。このため、prologにおける記述のいくつかはマクロ展開されてしまうものがあり、これらについては展開されないように考える必要がある。

例1.

prolog

```
change__address(label+N):-~
```

body部では、label で受けたラベル名（atom）のアドレスを求めて、N で受けた値と和をとるといった処理をする。

ESP

```
change__address(Label+N):-~
```

↓ マクロ展開

```
change__address(Temp):- ~add(Label,N,Temp);
```

Label,N ともに未定義のため、“illegal input ” という実行時のエラーが発生してしまう。そこでマクロ展開を抑止するため、クォートする。

```
change_address( '( Label,N)):- ~
```

例２

prolog

```
conv_to_int(#H,I):- ~
conv_to_int(N#S,I):-~
```

body部では、H を16進数として10進変換し I へ値を返す。又はS をN 進数として10進変換し I へ値を返す。

ESP

```
conv_to_int( '( #H),I):-~
conv_to_int( ' '( N#S),I).- ~
```

この例では、 '(N#S)としてもライブラリへ登録時にエラーとなる。
バッククォートを２個付けるとうまくいく。

```
'( …)
' '( …)
```

マクロ展開の抑制され方が異なる。

例３

読み込んだtermをマクロ展開するのに以下のような例があった。

```
:read_term(!N,Term).        Termに#(4)のtermが読み込めた。

sub   max(Term,I).
                                呼ばれる側
                          sub  _max(I0,I1):-unbound(I0);
                          sub   max( '( X)), '( #(X))):

:write_term(out,out(I));
```

ファイルにはout(#4);と出力される。この述語を含むファイルをライブラリに登録すると、

```
UNKNOWN Class `4`      とエラーメッセージが出力される。
```

つまり(#4)は「クラス4」とマクロ展開される。

これを防ぐには、出力した結果に、バッククォートが付いていなくてはならない。

```
sub __max( `( #(X)), `( #(X)));
                ↓
sub __max( `( #(X)), ` `(  `(#(X))));
```

と変更した。

前項で示した:read _term(IN,Term) で、Termに、[X | Y]のかたちのtermが読み込まれた場合、

```
[X | Y]  →     `( #(4))   がｕｎｉｆｙする。!
```

このことに対応するため、sub __max の述語に新しく選択子を追加する必要があった。

```
sub __max(T,T1):-unbound(T),!;
sub __max( `( #(X)), ` `(  `(#(X))));
                ↓
sub __max(T,T1):-unbound(T),!;
sub __max([X | Y],[X  | Y]) :-unbound (X),!;
sub __max( `( #(X)), ` `(  `(#(X))));
```

3.5 assel/retruct 系述語

assert/retructは人域情報の記憶場所として活用されている。利用方法は大きくわけて以下の３種である。

type1.　アトミックデータで常に値が１種だけ保持されている。（アセンブル中のアドレスなど）

type2.　アトミックデータで次々にassertされているもの。（μプログラム中に現れるラベル名など）

type3.　タームを次々にassertしているもの

例１．(type1.)　アドレスのカウントアップの述語

prolog

```
asm _cf_up:- address(Adr),abolish(address,1),
             New is Adr + 1,assert(address(New)).
```

ESP

```
asm _cf_up(Obj):- New is Obj!adr + 1,   ; スロットから値をとり出し
                                        ;   １を加える。
                  Obj!adr:= New.

                                        ; スロットに値を代入する。
```

スロットアクセスで書き換えられる。

例２　(type2.)　ラベル名とラベルの存在するアドレスの登録。

prolog

```
asm _lab _proc(L):-address(A),assera(labs(L,A)).
```

ラベル名L とアドレスA をassertしている。

ESP

```
asm _lab _proc(L,Obj):-set_data(L,Obj!adr,D),
                       :add_first(Obj!labs,D);
```

set _dataの述語は２要素のheap_vector Dを作り、第１、第２引数をheap_vectorの要素へ代入する。add _first/2 はSIMPOSの提供するプール・サブシステムのmethodでlistの先頭に第２引数を追加する機能を持つ。

これによりラベル名とそのアドレスはlist中のheap_vectorの要素として記憶される。

例３　(type3)

termをassertしているのは一例だけである。どのように書替えたかを示す前に、

これに関するアセンブラの機能について述べる。

アセンブラは、それ自身が規定したニーモニックだけではなく、規定されたニーモニックを他の文字列へ変更することを許している。たとえば、ハードウェア資源名として、「r1」のレジスタがあるとする。このレジスタをμプログラム中で記述する場合に、「r1」と記述する代りに「work_reg 」と書くことも出来る。この機能を実現するために「r1」と「work_reg 」が等価であることとを宣言しておく必要がある。又アセンブラはこれらの宣言を記憶しておく必要がある。

```
::  work_reg        equ        r1    .

    work_reg := 0   .
```

アセンブラは、work_reg をr1と読みなおして処理を実行する。

故ち

```
::  a       equ      b       .
```

の宣言から、アセンブラはa をkey として検索すればb が得られるようなテーブルを作成する必要がある。

prolog版では、上のような宣言を読み込むとそのままそれをassertしている。

```
(a1 equ b1).
(a2 equ b2).
    :
```

左のようなテーブルが作成できる。

ここで述べたような機能は、SIMPOSのプール・サブシステム中の[注)]「クラス hash_index 」が提供している。しかし、a , b ともに登録時、検索時にアトミックではなくスタックベクタであることが多い。

注)　「クラス　hash_index 」に登録できるのはスタックベクタ以外のデータである。

```
::  r=0      equ      Z  32

if(r=0)   goto Label.
```

（フラグ z_32の結果で分岐）

上の例では、(r=0) をkey として、Z _32が検索できるように登録するが(r=0) はスタックベクタであり、データタイプを変換する必要がある。このデー

タタイプの変換には、「クラス　string_IO_buffer」を利用した。このクラスはtermをstringに変換したり、又逆にstringをtermに変換する機能を提供している。

登録

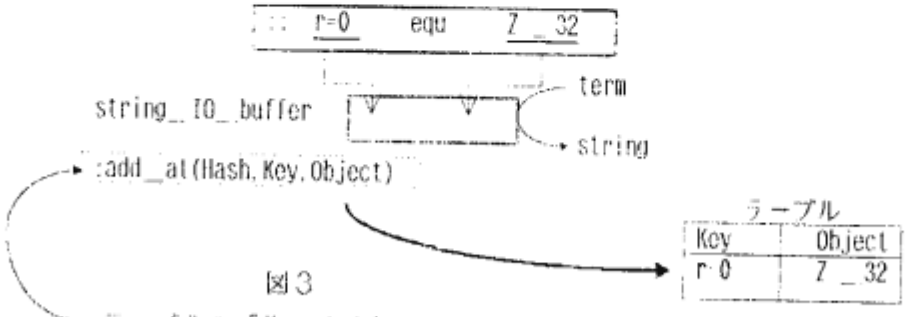
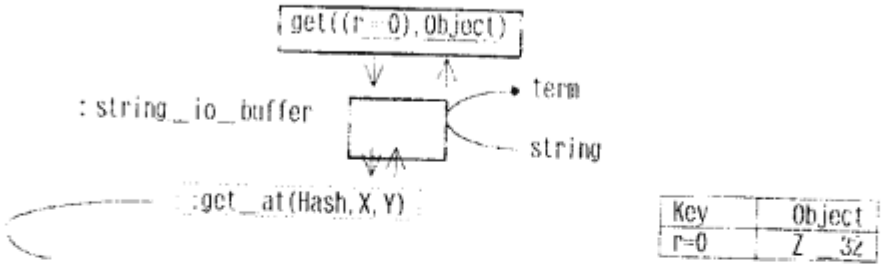


図3

テーブルへ「Key」をkey として「Object」を登録する述語

図3にテーブルへ登録する場合のフローを示した。以下に検索手順を示す。get なる述語は第1引数をKey としてテーブルを検索し第2引数にObjectを返す。
（検索に失敗するとfailする）



テーブルへX をkey として検索し見つかったものをY へ返す。
（検索に失敗するとfailする）

図4

アセンブラは、アセンブルするマイクロプログラム一行を読み込むと、そのたびに数回～数10回さきに述べたテーブルを検索する。このとき、検索するたびにterm→Stringの変換が必要であり、かつこの検索は、ほとんどの場合failする。

アセンブラの処理上このテーブルの作成は、マイクロプログラム本来の命令をアセンブルする前に完了させることができる。そこで、テーブルをプール・サブシステムの機能で実現するのではなく、1個のクラスとしてアセンブル時に動的にライブラリへ登録するように考えた。（このようにして作成した専用アセンブラがクラスの構成の章で述べた「改良された専用アセンブラ」である。）

動的にライブラリへ登録するクラスのイメージを図5に示した。

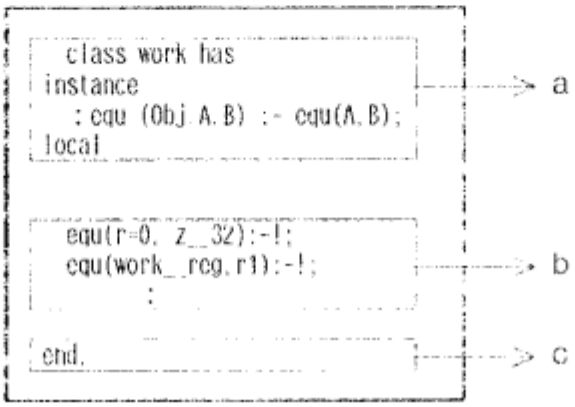


図5

図5に示したクラスを「string__IO__buffer」へ書き出し、ニーモニックの定義部が終了した時点で「strint__IO__buffer」からbuffer内のデータを読み出してライブラリへ登録する。

参考のために図6にこれらの手順を示した。

図4　aの部分をbufferに出力する。

図4　bの部分をbufferへ必要なだけ出力する。

図4　cの部分をbufferへ出力する。

bufferから読みだす。

ライブラリーへの登録。

インスタンスの生成

```
:create(#string __io__buffer,Buffer);

:put_ string(Buffer,"class work has...");

:write__term(Buffer,
              equ( ' '(  '(Term)),
                   (  ' '(  '(Key) )),
:out_ string(Buffer,";-!;");
           :
           :

:put__string(Buffer,"end.");

:read __term  (Buffer,Term);
           :

:make __class __template__from__ source(#library,
      term,Class__name);
:make __class __object(#library,Class __name);
:compile(#library,Class   name);
:get__class __object(#library,Class __name,
       Class __object);

:new(Class__object,Instance __object);
```

図6

4．評価

アセンブラの実行速度を測定した。測定には、96ステップのマイクロプログラム（組込述語add のソース）及び1263ステップ（ガーベジコレクション用のソースgc_mark）のマイクロプログラムを使用した。測定時間は、初めに述べた応答速度（アセンブラを起動後、アセンブルするfile名を入力した時刻からアセンブルが終了した時刻までの時間）である。

表6　応答速度

	PSI 上に移植したアセンブラ		DEC2065 上の
	専用アセンブラ	改良した専用アセンブラ	prolog版アセンブラ
add	１２２秒	７０秒	６６秒
gc_mark	１５９７秒	８７０秒	―――

DEC2065 はload factor 2.04

はじめにも述べたように、移植したアセンブラはDEC 2065上のProlog版とほぼ同等の性能が得られた。

次にアセンブラの各処理に分けて実行時間の若干の解析を行った。

「専用アセンブラ（以下assyと略す）」と「改良されたアセンブラ（以下 f_assyと略す）」の述語呼出回数を測定した。測定結果を表 7,8に示した。

assy,f_assyともに呼出頻度の高い述語は、term処理に関するものとassert/retructの代わりに利用したプール・サブシステムに属するものである。

assyと f_assyで、最も出現頻度の高い述語は55% 程度呼出回数が減少している。これは、すでに述べたテーブルの持ちかたを変更したことによる。さらに f_assyでは、assy以上にterm処理に関する述語の出現頻度が高いので、アセンブル処理時間に占める時間比を測定した。

アセンブラの処理は２パスから成り、マイクロプログラムを１行ずつ読み込んでその１行に対してオブジェクト生成をする。

図７にアセンブラの処理フローを示したが、term処理に要す時間を測定するため、図７中３，６の処理を除き、かつ７のリストとオブジェクトの出力のかわりに読み込んだtermのみを出力するような簡単なプログラムを作成した。

f_assyでアセンブルに70秒を要したadd の場合、上に述べたプログラムの実行に45秒必要であった。（ 測定はアセンブラの場合と同様な応答時間である。）

PREDICATE CALL COUNTER OUTPUT LIST　　表7

ORDER	COUNT	RATE	(ACCUM)	CLASS / NAME / TYPE / ARITY
1:	72203	3.99%	(3.99%)	character / is_character / cmp / 2
1:	72203	3.99%	(7.99%)	character / is_character / local / 1
3:	64056	3.54%	(11.5%)	with_link / get_next / imp / 2
4:	52980	2.93%	(14.4%)	index_entry / get_key / imp / 2
5:	52896	2.92%	(17.3%)	list_index / equal_key / local / 2
6:	48930	2.70%	(20.1%)	esp_character / get_type / local / 3
6:	48930	2.70%	(22.8%)	esp_character / ct / local / 2
8:	43914	2.43%	(25.2%)	as_character_file_input_tap / at_terminal / local / 3
9:	43868	2.42%	(27.6%)	as_character_file_input_tap / eof_code / imp / 2
10:	40779	2.25%	(29.9%)	list_entry / get_object / imp / 2

f_ASSY で add をアセンブル

PREDICATE CALL COUNTER OUTPUT LIST　　表8

ORDER	COUNT	RATE	(ACCUM)	CLASS / NAME / TYPE / ARITY
1:	160689	5.28%	(5.28%)	character / is_character / cmp / 2
1:	160689	5.28%	(10.5%)	character / is_character / local / 1
3:	137309	4.51%	(15.0%)	esp_character / get_type / local / 3
3:	137309	4.51%	(19.6%)	esp_character / ct / local / 2
5:	104929	3.45%	(23.0%)	as_esp_unparser / reconc / local / 3
6:	98456	3.23%	(26.2%)	with_link / get_next / imp / 2
7:	88223	2.90%	(29.1%)	index_entry / get_key / imp / 2
8:	88179	2.89%	(32.0%)	list_index / equal_key / local / 2
9:	62296	2.04%	(34.1%)	list_entry / get_object / imp / 2
10:	60532	1.99%	(36.1%)	list_index / get_at / local / 3

ASSY で add をアセンブル

$$\frac{\text{アセンブラ入出力処理時間}}{\text{アセンブラ全処理時間}} \times 100 \fallingdotseq 64[\%]$$

図7

ここで述べるfileとの入出力処理は、ハードウェア的ないわゆる入出力のみでなく、読み込んだものをtermにまとめる処理を含んでいる。

これらのfile入出力はSIMPOSの「character _file input/output tap」で実現しておりこれらの高速化がPSI 上のアセンブラ高速化に大きく寄与するであろう。

5．所感

5.1 移植に関して

「SP のマクロ展開機能及びオペレータ宣言との関連で機械定義テーブルを一部修正する必要があった。

移植を機械的に実施するために、クラス分けはすでに述べたような方式をとった。このため、クラスが大きくなりデバッグの効率が思わしくなかった。

評価の項で述べたように、termをassert/retructしているようなものを移植する場合、それなりの工夫が必要である。本稿で初めに示した例では特に工夫がなされておらず実行性能は思わしくなかった。

5.2 実機上の開発環境

本稿で述べた移植作業はすべて実機上でおこなった。作業中に感じた点をいくつか述べる。

- ファイルマニピュレータ；操作がほとんどマウスのみでできる点は便利。しかし、複数の同一名称のfile extentionを持つものを削除するとか複数のfileをfloppyへコピーするとかの場合、file名等にワイルドカードで扱えたり又は、マウスで複数のfileを選択して〈do it 〉といったイメージの操作方法があるとさらに扱いやすい。
- デバッガ；デバッガは欲をだすと限りないが、「デバッグのLogging」、「spy point のセット及びtrace point の変更、これらdebug 環境のsave」といった機能があると便利である。
- 全般；いわゆるマウス&マルチウィンドウで作業能率は充分良いと思われるが、各PS単位でマウス等の操作方法が違いがあり、とまどうことが多かった。今後操作方法の統一が望まれる。

6． まとめ

移植した汎用アセンブラは、冒頭にも述べたように、「DEC 2065」のprolog版と同等の実行性能を示し一応の目標を達成した。

本稿が今後同様の作業をする方の参考になれば幸甚である。

参考資料

1．汎用型マイクロプログラム・アセンブラ　　高木茂行　　8-AUG-83 ICOT TR-021

2．ＳＩＭＰＯＳ使用説明書　　MP3002-1　ICOT(1985)

3．KL0 組込述語使用説明書　　MP7002-1　ICOT(1985)