CP/M-86 のアセンブリ言語への構造化プログラミングとモジュラプログラミングの導入

内田智史 1)　槇田幸 2)　戸浪聡 3)　間野浩太郎 1)

1)青山学院大学理工学部経営工学科
2)日本コンピュータ研究所
3)野村コンピュータシステム、システム技術部

1．はじめに

アセンブリ言語には、過去十数年のソフトウェア工学の成果があまり取り入れられておらず、開発や保守の効率が一般に良くない。特に、大規模なソフトウェア開発ではこの状況は加速的に悪化する。我々の研究室では、8086を用いたパーソナルコンピュータの上で、実行効率・メモリ効率を落とさないようにエディタや言語プロセッサなどをアセンブリ言語を用いて開発してきたが、この開発では、さまざまな障害を起こしていた。たとえば、卒業していった学生の作成したプログラムの引継ぎの問題や毎年年頭に必要となる未熟練プログラマに対する教育と（特に読みやすいプログラムを作成するための）訓練には頭を悩ましていた。特に、当研究室においても作成すべきプログラムは年々巨大化し複雑になる傾向があり、引継ぎ可能なプログラムの作成と初心者教育の重要性はますます増加してきている。このことは、産業界――特に、マイコン用のソフトウェアハウス――でもほぼ同じ状況であろう。

そこで、我々はCP/M-86 のアセンブリ言語(ASM86,RASM86)に、ソフトウェア工学の研究成果（構造化プログラミング・階層的モジュラプログラミング・データ抽象化など）を導入し、アセンブリコーディングであっても、開発すべきソフトウェアが巨大化しても、開発効率・保守効率を大幅に向上させることができ、また、初心者であっても比較的短期間に読解性の高いプログラムが記述できるプログラミング環境を作成した。本報告では全体を２つに分けて、中規模程度のソフトウェアを作成するのに、アセンブリ言語に構造化プログラミング・階層的モジュラプログラミングを導入したプログラミング環境(MDL-ASM86) について述べ、引き続き次の報告で、大規模なソフトウェアを作成するのに、MDL-ASM86 にさらにデータ抽象化を導入したプログラミング環境(ADT-RASM86) について述べる。

表１　本システムとその対象プログラムサイズ

	適用プログラムのサイズ	適用する方法論	処理系	主なねらい
報告１	中規模 500 - 1000	モジュラプログラミング 構造化プログラミング	MDL-ASM86	保守性の向上・高速性 デバッグ効率の向上
報告２	大規模 1000 -	データ抽象化	ADT-RASM86	部品化による 生産性の向上

(MDL-ASM86:MoDuLar-ASM86, ADT-RASM86:AbstractDataType-RASM86)

1.1 MDL-ASM86 の特徴と応用範囲

MDL-ASM86 が、他のシステムと大きく異なる点は、アセンブリ言語の特徴を失わないで、「構造化」していることである。つまり、通常のアセンブリ言語の中に構造化プログラミングに必要とされているいくつかの論理構造文を導入している。また、通常のアセンブリ言語の文は全て記述できる。このことが、当然のことながら、高級言語処理系に比べてプログラムの実行効率・記憶効率に大きな利点を与えている。

本システムの応用範囲としては、500 - 1000ステップ程度の中規模のプログラム、特に、OSまわりのプログラムの記述・各種ユーティリティコマンド・基本ライブラリの構築などが考えられる。

2．プログラム作成の生産性・プログラム保守性の向上のための基本的分析

2.0 生産性・保守性を高めるためにアセンブリ言語に持たせる機能

「生産性・保守性の高いアセンブリ言語のプログラム」を次のように考えた。

(1) 読解性がよいこと。
(2) 実行速度が速く、記憶効率が良いこと。
(3) 信頼性が高いこと。
(4) 記述量が少ないこと。つまり表現が簡潔なこと。
(5) 大規模なプログラムの構築が容易であること。
(6) ソフトウェアの再利用ができること。
　　一度作成されたソフトウェアは、実世界で十分にテストされ検証されているのでその信頼性は非常に高い。これらのソフトウェアの再利用によって生産コストを大幅に低減させることが可能となる。
(7) これらを実現する処理系はできるだけ軽いこと。
　　現在の日本の環境では、マイコンのソフトウェアの開発は、大型コンピュータ、スーパミニコンピュータなどではなく、セルフシステムを用いて行なわれることが多い。これらが動作する環境はCPU 速度、利用可能メモリ量で制約されることが多いので、処理系自体は軽くなければならない。

そこで我々は上記の(1),(3),(4) を満足するために「論理構造文」をアセンブリ言語に導入し、(2) の観点から「論理構造文」では、直接レジスタやフラグを判別できるようにした。また、全てのアセンブリ言語の記述を可能にした。また(5) を満足するために、「階層的モジュラプログラミング」のための機能を導入した。(3),(4),(5),(6) を満足するために、「データ抽象化」を導入した（データ抽象化については、次報告を見られたい）。(7) を満足するために、処理系は言語翻訳系・文書系・デバッグ系と別々に作成した。特に言語処理系は「軽さ」と「扱いやすさ」の点から、プリアセンブラプロセッサの形式を取った。また、(1) のために「日本語による標記」を前面に押し出した。

2.1 アセンブリ言語と構造化プログラミング

JMP 命令はプログラムの構造を非常に動的なものにしてしまい、読解性・保守性を大幅に低下させてしまう。そこでより抽象度の高いプログラム制御文、すなわち、IF文、DO文などの『論理構造文』を導入すれば、プログラムの記述性・読解性・保守性を高めることができる。

この論理構造文の設計に当たっては、8086の各命令を十分に反映した論理構造文の体系になっていなければならない。従って、

(1) Pascal等の高級言語の持つ論理構造文（主に、記述性・読解性向上のため）
(2) アセンブリ言語特有の命令を生かすための論理構造文（効率の良いオブジェクトコード生成のため）

が必要になる。

また、構造化することにより実行速度、記憶効率が犠牲になってはならない。そこで、いくつかの通常のアセンブリ言語で書かれたプログラムを調査したが、プログラムの上位レベルでは、多くの場合、構造化することによっても、実行速度・記憶効率が損なわれることは少ないが、プログラムの下位レベルでは、どうしても「構造化されたプログラム記述」が、それらのネックとなってしまう場合があることが分かった。我々は、この問題点をプログラムをセクションに細分することにより解決したが、この点については2.2 節で詳しく述べる。

・シンボリックデバッガの必要性

また、これらの論理構造文を使用すると、ソースコードとオブジェクトコードは1対1に対応していないので、DDT-86のようにオブジェクトプログラムを直接デバッグするデバッガではもはやデバッグすることはできない。従って、構造化アセンブリ言語のソースレベルでデバッグ可能なシンボリックデバッガが必要となる。

2.2 アセンブリ言語と階層的モジュラプログラミング

一般に、大規模なプログラムを作る際に、良く行なわれる方法に、プログラム分割がある。我々は、500 ～ 1000 ステップ程度の中規模のソフトウェアに対しては、セクションと呼ばれるプログラム単位（COBOL のセクションとほぼ同じ概念）を用い、トップダウンアプローチによるプログラミング方法論、すなわち、階層的モジュラプログラミングを採用し、それ以上の大規模なソフトウェアに関しては、データ抽象化を導入することにした。

・階層的モジュラプログラミング

プログラムの機能を同一平面上に記述するのではなくて、いくつかの副機能に階層的に細分して記述することが、大きなプログラムを作成することを可能にする。我々は、この分割する際の単位をセクションと呼ぶ。このセクションには約20行から40行程のプログラムを記述し、これに自然言語（日本語）の表題を付けることでプログラムの読解性が大きく向上する。このようにすると、プログラムの上位部分はほとんど下位セクションの呼び出しとなり、そのセクションの表題を読むことでプログラムの概略が短時間の内に分かる。さらに下位レベルにゆくにつれて、実際のプログラム記述が多くなってゆくが、このセクションの行数を前述のように20行から40行前後にしておくと、たとえその中でJMP 文を使用したとしても、セクションの管理範囲が狭いので読解性に悪影響を与えることは少ない。

2.3 処理系作成に関して

前述のように、制約条件の多いセルフシステム下で開発に用いるので、処理系はコンパクトでなければならない。そこで、我々は、既存のシステムをなるべく最大限に利用することにし、言語処理系をプリプロセッサとして実現した。また、シンボリックデバッガも、必要な変数名・アドレス対応表はASM86 の出力するシンボルファイルをそのままデバッグデータベースとして利用した。また、文書性向上のために、ソースリストを編集して出力するドキュメンテイタも作成した。

3．処理システムの概要

以上の考察を基に、我々は、1982年度から上記のシステム要件を満足するシステム作成に取り掛かった。データ抽象化を用いない中規模のシステム作成に対しては、次のような処理系が用意されている。なお、これらは、CP/M-86 の ASM86の上で動く。

[1] ASM86の構造化プリアセンブラ (MDL-ASM86)
[2]ドキュメンテイタ　　(DOC:DOCumentater)
[3]シンボリックデバッガ　(IDSP:Interactive Debugging Support Program)

以下に、各処理系について説明してゆく。

3.1 ASM86 構造化プリアセンブラ　MDL-ASM86

ASM86 に「構造化プログラミング」を導入するために、プログラムの制御を明確に示すための論理構造文を導入した。表３．１(a),(b) に論理構造文の一覧を示す。これらは対応するASM86 アセンブリ言語に翻訳される。

これらの表から分かるように、論理構造文はその行の先頭に＊があることで識別される。FORTRAN のDO文と同じ意味を持つ*DO 文を用いてAXレジスタに１から10まで加算するプログラムは次のようになる。なお、MDL-ASM86 は、ASM86 と同じように英大文字と英小文字の区別はしない。

```
xor ax,ax
*do 1から10までの累計 bx = 1 up to 10
    add ax,bx
*end-do 1から10までの累計
```

プログラム3.1　*do 文の例

```
0000 XOR     AX,AX
0002 MOV     BX,0001
0005 JMP     0009
0008 INC     BX
0009 CMP     BX,000A
000C JG      0013
000E ADD     AX,BX
0010 JMP     0008
```

変換結果

表３．１(a) 高級言語と同じ論理構造文　[] は省略可能を示す。

構　　文				機　能
*if then		[*else]	*end-if	二分岐選択文
*do-case	*case	[*else-case]	*end-case	多分岐選択文
*while			*end-while	while 文
*repeat			*until	repeat文
*do			*end-do	do文
*loop			*end-loop	単純ループ文
*leave				ループ脱出文
*search	*found	[*not-found]	*end-search	表探索文

表３．１(b) アセンブリ言語特有の命令を生かすために導入された論理構造文

構　　文		機　　能
*cloop	*end-cloop	cxレジスタが０になるまでループ
*block	*end-block	短いプログラムエリアからの脱出用

また、論理構造文の*if 文などで用いられる「条件式」は、8086命令セットにあるほとんどの命令を示せるようになっている。また、単に二値比較だけでなく範囲指定、フラグの検査が出来るようになっている。しかし、条件式をAND やORなどの論理演算子で続けて記述することはできない。

条件式には、二項の比較、範囲チェック、およびフラグのチェックの３種類がある。

[1] 二項の比較

いわゆるCMP 命令を、次のような分かりやすい二項比較の形式にしたものである。

第１オペランド　比較演算子　第２オペランド

これは、『CMP　第１オペランド，第２オペランド』というアセンブラ命令に変換される。ここで、比較演算子は次の通りである。

等しい	→	=	等しくない	→	<>
より大	→	>	より小	→	<
より大か等しい	→	>=	より小か等しい	→	<=
より上	→	>>	より下	→	<<
より上か等しい	→	>>=	より下か等しい	→	<<=

比較演算子の意味

例えば、axレジスタの内容が０ならbxレジスタの内容を１つ加算するプログラムは次のように書く。

```
*if ax = 0 then
    inc bx
*end-if
```

プログラム例3.2 *if 文の例

[2] フラグのチェック
次の一般形式でフラグのチェックを行う。

第１オペランド　＝　第２オペランド

ここで第１オペランドに
%z（ゼロ・フラグ）　、　%o（オーバーフロー・フラグ）
%s（サイン・フラグ）　、　%p（パリティ・フラグ）
%c（キャリ・フラグ）

第２オペランドに

1	（フラグが立っている）	
0	（フラグが立っていない）	
same	（フラグが立っている）	文字列比較(CMPS)命令とペア ゼロ・フラグにて判断
notsame	（フラグが立っていない）	
bitoff	（フラグが立っている）	TEST命令とペア ゼロ・フラグにて判断
biton	（フラグが立っていない）	

以下にプログラム例を示す。

```
mov si,offset acc_buf
mov di,offset fcb_buf
mov cx,12
cld ! repe cmpsb
*if %z = same then
    mov ax,1
*end-if
```

プログラム例3.3 フラグのチェック（その１）

```
mov  ax,73ach
test ax,0020h
*if %z = biton then
    mov [si],ax
*end-if
```

プログラム例3.4 フラグのチェック（その２）

[3] 範囲のチェック
次のような範囲指定演算ができる。
一般形式　　between 第１オペランド　範囲指定
これは例示した方が分かりやすいだろう。

```
between ax   0:10      →      0 ≦ ax ≦  10
between bx  <0:10>     →      0 < bx <  10
between cx <<0:10>>    →      0 << cx << 10
between dx  <0:10      →      0 < dx ≦  10
```

例：blレジスタの内容をアスキ文字列としてみたときに、英大文字なら１、英小文字なら２、特殊文字なら３、制御文字なら４、カタカタなら５をaxに返すプログラムは、

```
*do-case
*exit-to-case 英大文字 ( between bl 'A':'Z' )
*exit-to-case 英小文字 ( between bl 'a':'z' )
*exit-to-case 特殊文字 ( between bl ' ':'/' )
*exit-to-case 特殊文字 ( between bl ':':'@' )
*exit-to-case 特殊文字 ( between bl '[':'_' )
*exit-to-case 特殊文字 ( bl = 0a0h          ) ; カタカナ空白
*exit-to-case 制御文字 ( between bl 00h:10h )
*exit-to-case カタカナ ( between bl '｡':'ﾟ' )
*case 英大文字
      mov ax,1
*case 英小文字
      mov ax,2
*case 特殊文字
      mov ax,3
*case 制御文字
      mov ax,4
*case カタカナ
      mov ax,5
*end-case
```

プログラム例3.5　　between 使用例

また、階層的モジュラプログラミングを実現するために、セクションという概念を導入した。これは、COBOL のセクションや PL/I の内部サブルーチンとほぼ同じもので、*exec 文によって実行される。

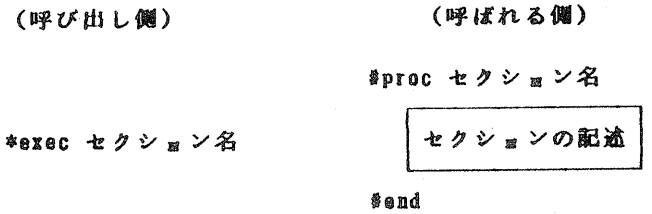

図3.2　セクションの構造

ここで、＃で始まる文はセクション制御文と呼ばれるもので、セクションの範囲を示すためのものである。表３．２にセクション制御文の一覧を示す。

表3.2 セクションの構造

$proc main	... $end main	主セクションを記述する(retf 文生成)
$proc	... $end	サブセクションを記述する（ret 文生成）
$aproc	... $aend	ラベルを用いるサブセクションの記述
$lproc	... $lend	大きいセクションの記述　*1)
$data	... $end data	データセグメントの記述
$extra	... $end extra	エクストラセグメントの記述
$stack	... $end stack	スタックセグメントの記述
$code	... $end code	コードセグメントの記述　*2)

*1) *if 文などは、それに対応する条件付きJUMP命令に展開されるが、この命令の飛び先は256 バイトまでなので、*if ブロックの内容が大きいとその範囲を超えて LABEL OUT OF RANGE のエラーを起こしてしまう。そのような場合には、$lprocを指定する。

*2) コードセグメントにデータやプログラムを記述する場合

最後にCP/M-86 のtypeコマンドをMDL-ASM86 で記述した例を示す。

```
#proc main
        *exec ファイルのオープン
        *if al <> ErrorCode THEN
          *exec ファイルをコンソールへ出力
        *else
          *exec エラーメッセージ出力
        *end-if
#end main
#proc ファイルのオープン
        mov cl,BdosOpen
        mov dx,FCB
        int BDOS
#end
#proc ファイルをコンソールへ出力
        *loop バッファごとの出力
           mov cl,BdosRead
           mov dx,FCB
           int BDOS
           *do 一文字ごとの出力 bx = 0 up to BffuerSize
              mov dl,DMA[bx]
              *leave バッファごとの出力 ( dl = EOF )
              mov cl,BdosConsoleOut
              push bx
              int BDOS
              pop bx
           *end-do 一文字ごとの出力
        *end-loop バッファごとの出力
#end
#proc エラーメッセージ出力
        mov cl,BdosPrintString
        mov dx,offset ErrorMessage
        int BDOS
#end
#data
      ;アスキーコード ------------------------------
        Bell              equ 07h    ; ベル コード
        EOF               equ 1ah
      ;使用変数 ------------------------------------
        BffuerSize        equ 127
        DMA               equ 0080h
        FCB               equ 005ch
        ErrorMessage      db  Bell,'*** ファイルが見付かりません  ***','$'
      ;BDOS コード ---------------------------------
        BDOS              equ 224
        BdosconsoleOut    equ 02h
        BdosOpen          equ 0fh
        BdosPrintString   equ 09h
        Bdosread          equ 14h
        ErrorCode         equ 0ffh
#end data
```

プログラム例3.6 typeコマンド

3.2 ドキュメンテイタ　DOC

これは、MDL-ASM86 のソースプログラムを編集してプログラムの理解を高め、デバッグの助けとして出力するものである。

上の例のプログラムをdoc にかけると次ページのようになる。

```
CP/M-86  日本語標記浄書用段付けリスタ   Ver. 2.10  02/14/85
  パス１の終了
    ＴＹＰＥＣＭＤ．ＵＣＨ      ８５年０５月１６日  １０時１４分３２秒

                                ( by  日本語浄書用段付けリスタ  Ver.  2.10 )


section <<  0>>   #proc main

            1 *   :  *exec<  1>ファイルのオープン
            2 *   :  *if al <> ErrorCode THEN
            3 *   :  :  *exec<  2>ファイルをコンソールへ出力
            4 *   :  *else
            5 *   :  :  *exec<  3>エラーメッセージ出力
            6 *   :  *end-if

                  #end main


section <<  1>>   #proc ファイルのオープン

            7 *   :  mov cl,BdosOpen
            8 *   :  mov dx,FCB
            9 *   :  int BDOS

                  #end


section <<  2>>   #proc ファイルをコンソールへ出力

           10 *   :  *loop バッファごとの出力
           11 *   :  :  mov cl,BdosRead
           12 *   :  :  mov dx,FCB
           13 *   :  :  int BDOS
           14 *   :  :  *do 一文字ごとの出力 bx = 0 up to BffuerSize
           15 *   :  :  :  mov dl,DMA[bx]
           16 *   :  <------*leave バッファごとの出力 ( dl = EOF )
           17 *   :  :  :  mov cl,BdosConsoleOut
           18 *   :  :  :  push bx
           19 *   :  :  :  int BDOS
           20 *   :  :  :  pop bx
           21 *   :  :  *end-do 一文字ごとの出力
           22 *   :  *end-loop バッファごとの出力

                  #end


section <<  3>>   #proc エラーメッセージ出力

           23 *   :  mov cl,BdosPrintString
           24 *   :  mov dx,offset ErrorMessage
           25 *   :  int BDOS

                  #end


  data   section  #data

                      ;アスキーコード ------------------------------
                        Bell              equ 07h    ; ベル コード
                        EOF               equ 1ah
                      ;使用変数 ------------------------------------
                        BffuerSize        equ 127
                        DMA               equ 0080h
                        FCB               equ 005ch
                        ErrorMessage      db  Bell,'*** ファイルが見付かりません  ***','$'
                      ;BDOS コード ---------------------------------
                        BDOS              equ 224
                        BdosconsoleOut    equ 02h
                        BdosOpen          equ 0fh
                        BdosPrintString   equ 09h
                        Bdosread          equ 14h
                        ErrorCode         equ 0ffh

                  #end data


      プログラムのエラー総数     0

  パス２の終了
浄書用段付けリスタの終了
```

プログラムリスト例

3.3 シンボリックデバッガ　IDSP

図3.1 に示すように、ソースプログラムとオブジェクトプログラムの間には 1 対多対応の関係があるので直接DDT86 でデバッグすることは困難である。

```
 5 *    :  *do-case
 6 *    : <---*exit-to-case ひらがな ( between ax 'あ':'ん' )
 7 *    : <---*exit-to-case カタカナ ( between ax 'ア':'ン' )
 8 *    :  *case ひらがな
 9 *    :  :  mov bx,1
10 *    :  *case カタカナ
11 *    :  :  mov bx,2
12 *    :  *else-case
13 *    :  :  mov bx,3
14 *    :  *end-case
```

```
0013 CMP    AX,82A0
0016 JL     0020
0018 CMP    AX,82F1
001B JG     0020
001D JMP    0030
0020 CMP    AX,8341
0023 JL     002D
0025 CMP    AX,8393
0028 JG     002D
002A JMP    0036
002D JMP    003C
0030 MOV    BX,0001
0033 JMP    003F
0036 MOV    BX,0002
0039 JMP    003F
003C MOV    BX,0003
```

ソースプログラム　　DDT86 による逆アセンブルリスト

そこで、ソースプログラムをそのままの形でデバッグできるシンボリックデバッガを開発した。そのコマンドの一覧を次に示す。

表3.3　シンボリックデバッガコマンド一覧表

	コマンド名	意　　味
ブレイクポイント	at ap off	ブレイクポイント設定 ブレイクポイント表示 ブレイクポイント解除
実行	go t u	実行開始 トレース アントレース
表示	{b,w,d}p a{b,w,d}p rd x	変数表示（b:バイト、w:ワード、d:ダブル） 配列表示 レジスタ表示 フラグとレジスタ表示
トレース	t{b,w,d} ta{b,w,d} tp	変数トレース設定 配列トレース設定 変数トレースポイント表示
変更	c{b,w,d} x{r,f}	変数変更 レジスタ(r),フラグ(f) 変更
その他	i m s q	ユーザfcb の作成 メモリ表示 メモリ変更 終了

＊注意　{b,w,d} や{r,f} は、この中から 1 つ選択して記述することを示す。

4．評価

MDL-ASM86 は、1982年秋に最初の版が完成し、その後、いくつかの変遷をたどりながら現在の日本語対応版になった。最初の版から約２年半の使用経験があるが、その経験とアセンブリ言語(ASM86) との比較評価実験によると次のことが言える。

(1) アセンブリ言語に慣れたプログラマであれば、生産性はアセンブリ言語を直接用いた場合に比べ、ほとんど同じかほんの少しだけ向上する。

理由：これは、適当な大きさにプログラムをモジュール化してプログラムを作成すれば、アセンブリ言語でも十分にプログラムを速く作成できるからである。

(2) プログラマの疲労度は「構造化」した方がはるかに少ない。

理由：これは、アセンブリ言語ではプログラムを作成するときにちょっとしたことでも考えて記述しなければならないからであろう。たとえば、MDL-ASM86 では、文字が英小文字かどうか判断するプログラムは、*case 文や betweenを用いて簡潔に記述できるのに対し、アセンブリ言語では、次のように書かなければならない。このことは慣れてしまえば、記述時間はそれほど違わないが、脳を使う度合いがかなり違う。

```
cmp al,'a'
jb      not_alpa
cmp al,'z'
ja      not_alpa
```

```
*exit-to-case 英小文字 ( between al 'a':'z' )
```

アセンブリ言語(ASM86) による記述　　MDL-ASM86 による記述

プログラム例4.1 英小文字判断

(3) 他人のプログラムの読み易さ及び理解に要する時間はMDL-ASM86 の方がはるかにASM86 を凌ぐ。これは、初心者の書いたプログラムについても同様のことが言える。

理由：まず、日本語を各セクションの表題に使用していることと、各セクションの内容が、論理構造文によって理解しやすくなっているからであろう。

また、２年半の使用経験により使い勝手と処理系の信頼性は大幅に向上したが、いくつかのベテランプログラマのコーディング（特にCP/M-86 の BDOS や CCP）と比較して、それと同等のオブジェクトコードを生成するには以下の機能が必要なことが分かった。

(1) 条件をAND や OR で組み合わせて記述できるようにすること
　理由：不要な*else ブロックの削除のため。

(2) セクションのビルトイン展開が可能になること
　理由： call 文の数を減らして実行速度を速めるため

5．参考文献

[内田85]　内田智史・間野浩太郎
CP/M-86 アセンブリ言語へのデータ抽象化機能の導入
情報処理学会　マイクロコンピュータ研究会資料36-2,pp.11-20 (1985.6)

[槇田83a] 槇田幸・内田智史・佐藤術・間野浩太郎
マイコンOSで動くマイクロコンピュータソフトウェア開発のための
高級アセンブリ言語とデバッキング支援ツール
情報処理学会第26回全国大会　3N-1,(1983.3)

[槇田83b] 槇田幸
構造化プログラミングの思想を取り入れたマイクロコンピュータの
開発ツールの作成
1982年度、青山学院大学理工学部経営工学科卒業論文 (1983.3)

[戸浪85]　戸浪聡
CP/M-86 上の構造化プログラミング用アセンブリ言語プロセッサの
日本語標記化と改良
1984年度、青山学院大学理工学部経営工学科卒業論文 (1985.3)