A Language for System Programs and Its Implementation Using a Macro Processor

Shuichiro YANO*, Jun OKUI** and Nobuki TOKURA***

1. Introduction:

A new programming language ML-11 has been designed and implemented with the following features:

(1) system implementation language

(2) structured programming oriented language

(3) implemented by using macro assembly language MACRO-11 for the PDP-11/20.

The space and time efficiency of the program is an important factor in the system programming. The structured programming is useful for productivity including the documentation, the debugging and the maintenance of the programs. But these requirements are often conflicting. To resolve this conflict, ML-11 is implemented as a compound language of a low level assembly language and high level varietal macros.

A set of new macros listed in Table-1 are defined. They are classified into four groups.

(1) control macros (2) subroutine macros (3) data macros (4) field handling macros.

This language is implemented by adding the macro definitions to the system library of the macro assembler. This approach is much easier than preparing the complier. Furthermore, the modification and the extension of this language are very easy.

2. An outline of ML-11:

An ML-11 source program is a sequence of source lines, where each line contains a single assembly language statement or a ML-11 macro call statement.
Several macros make up an ML-11 statement. Any ML-11 statement may be nested within another statement. Indentation may be used to emphasize the control structure as shown in Fig.1. The ML-11 macros are underlined in Fig.1.

(1) *Control macros*: Several control macros oriented to structured programming are available. In Table-1, cond. is a conditional expression. Possible conditional

This paper first appeared in Japanese in Joho－Shori (Journal of the Information Processing Society of Japan), Vol. 16, No. 9 (1975), pp. 760～766.

* Fujitsu, Ltd.
** Department of Information Engineering, Nagoya Institute of Technology
*** Faculty of Engineering Science, Osaka University

expressions are summarized in Table-2. In the table, operand A and B can be a register, a variable or a field.

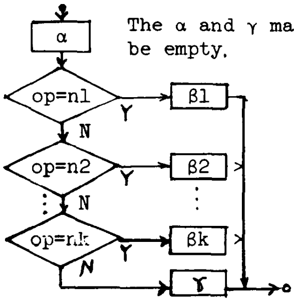
(2) *Data macros*:The field is a consecutive bit string in a word or a register (Fig.2).

```
SUBROUTINE MAX, <RØ,R1>, <RØ,R1>
  IF RØ  GE R1          ;RØ>R1
     RETURN <RØ>        ;YES
  ELSE
     RETURN <R1>        ;NO
  IFEND
SUBEND
```

Fig. 1 An example of ML-11 program

Tabel-1 ML-11 macros

Control macros

IF* cond. ... [ELSE] ... IFEND	IF statement
WHILE* cond. ... WHEND	WHILE statement
UNTIL* cond. ... UNTEND	DO UNTIL statement
SELECT# op [α] CASE n1 β1 CASE n2 β2 : CASE nk βk [OTHER γ] SELEND	 The α and γ may be empty.
SWITCH# op NODE φ βφ NODE 1 β1 : NODE k βk SWEND	SELECT# op CASE φ βφ CASE 1 β1 : CASE k βk SELEND
DOINC op, ℓ, m[, n][, UNSIGN] α DOEND	Iterate α increasing op index by n from ℓ to m. If n is not specified, then n=1. If UNSIGN is specified, then ℓ, m and n are treated as unsigned integers.
DODEC op, ℓ, m[, n][, UNSIGN] α DOEND	Iterate α decreasing op index by n from ℓ to m.
EXIT n	Jump to the statement next to the END macro(IFEND, WHEND, UNTEND, DOEND, SELEND, SWEND) of the n-th surrounding scope.

Subroutine macros

SUBROU[TINE] name[,formal parameters][,registers and variables to be saved] ... {RETURN rp} ... SUBEND	Subroutine definition. The rp is a unique formal return parameter.
Call name[,formal parameters][,return parameter][,registers and variables to be saved]	The subroutines are called by value.

Data macros

DCLB name {$ name|$$ name<i,j>|FILLER} DCLEND	Declaration of a data block.
DCLF name<i,j>	Declaration of a field
MAP name, formal parameters ... {RETURN rp} ... MAPEND	Definition of address mapping

Field handling macros

MOVE A,B	Extended MOV instruction which moves the field A to the field B. Some other extended instructions to specify the field in the operands are available.

notes: [ ] denotes that it is optional. {b|c|...} denotes that there may be an arbitrary number (≥1) of b, c or the others.] denotes the scope of the execution. * is space, B or C and # is space or B. If *(#) is space, word data or field will be tested. If *(#) is B, byte data will be tested. If * is C, the processor status will be tested.

The field specifier <i,j> denotes the field from the i-th bit to the (i+j-1)-th bit, where $0 \le i \le 15$, $1 \le i+j \le 16$. Each field can be refered to directly by the field specifier or by the field name defined by DCLF macro as follows:

word name† <i,j>

word name <field name>

Table-2. Conditional expressions

A	EQ	B	A=B	
A	NE	B	A≠B	
A	GT	B	A>B	(signed)
A	GE	B	A≥B	(signed)
A	LT	B	A<B	(signed)
A	LE	B	A≤B	(signed)
A	HI	B	A>B	(unsigned)
A	HIS	B	A≥B	(unsigned)
A	LO	B	A<B	(unsigned)
A	LOS	B	A≤B	(unsigned)
A	PL	#ϕ	A≥0	(signed)
A	MI	#ϕ	A<0	(signed)

EQ	Z=1†
NE	Z=ϕ
MI	N=1
PL	N=ϕ
CS	C=1
CC	C=ϕ
VS	V=1
VC	V=ϕ

† Z, N, C and V are Zero, Negative, Carry or oVerflow bits in Processor Statue Word, respectively.

A *block* consists of n consecutive words. For the convenience of block reference, a block template definition facility is introduced. The template of a block is declared by using DCLB ... DCLEND macros (Fig.2). The logical name of each word of the block is given by using "$" macro. To denote the word of the block which has no logical name, FILLE-R macro is used. The logical name of the field in a block is given by "$$" macro. These declarations give the relative position from the block address and do not assign any area of the main storage. The format of the reference to the block and the field in the block is;

```
DCLB  BKNAME
   $ WORD1
      $$ FIELD1<3,6>
   FILLER
   $ WORD2
   FILLER
      $$ FIELD2<Ø,8>
      $$ FIELD3<15,1>
DCLEND


DCLF  SIGN <15,1>
DCLF LOW <Ø,8>
```

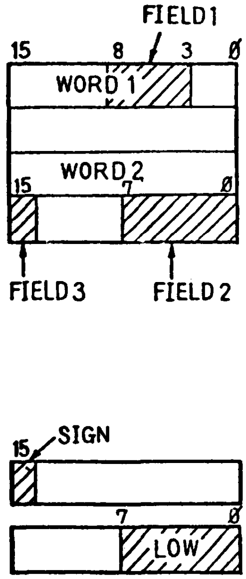


Fig.2 Declarations of block and field

block name <block address, word name or a field in the block>

† The word name is a character string which represents a word in the assembly language.

The block name defines the data structure of the block, and each field of the block is identified uniquely on the main storage by the block address.

The procedure which computes a unique address of the main storage with some actual parameters for data access (for example, hashing function) can be defined by the *address mapping* mechanism. Fig.3 shows an example of the address mapping of an array. The reference format of the address mapping is;

(1) mapping name <<actual parameters>>

(2) mapping name <<actual parameters>, field name>

(3) mapping name <<actual parameters>, i, j>

The field of the word whose address is computed by address mapping can be gained by using (2) or (3) formats. Fig.4 shows an example of the reference to the data structures. All the reference mechanism so far (field, block and address mapping) can appear as an operand in the following way:

(1) the operands in the conditional expressions

(2) the operands of extended instructions

(3) the parameters of subroutines.

```
MAP  ARRAY  <RØ,R1> ;  SIZE = ARRAY  SIZE
   WHILE  RØ  GT  #1;  RØ > 1 ?
      ADD  #SIZE , R1 ;  R1 ← SIZE + R1
      DEC  RØ  ;  RØ ← RØ - 1
   WHEND
   ADD  #START - 1 , R1 ; START = START  ADDRESS  OF  THIS  ARRAY
   RETURN  <R1>          ; R1 ← START + SIZE   (RØ-1) + R1 - 1
MAPEND
```

Fig.3 An example of address mapping definition

3. Macro expansion:

```
(a) IF  RØ  NE  <BKNAME<#1ØØØ,FIELD1>>
(b) MOVE  <BKNAME <RØ,WORD2>>,<R1 <Ø,1Ø>>
(c) AND  #PATTERN , <ARRAY <<R2,#1>, LOW>>
```

Fig.4 Some references to data structures

Fig.5 shows an example of the macro expansion of "if" statement. In the expanded text, .1, .2 and .3 are newly generated labels. To generate these labels, a label generating counter $n macro and a stack are used. When $n macro is called, it assigns the new value to "n" of the label .n and push down its value to the stack. In Fig.5, "n" is 1 for the label .n of the first IF macro expansion, and "n" is 2 for the label .n of the second IF macro expansion. The stack contains 2 and 1 in this order. The IFEND macro corresponding to the second IF macro pops up the stack using .2 as its label. The ELSE macro pops the stack, gets new

```
IF  A  EQ  B                 CMP  A , B ; compare
                             BNE  .1     ; jump to ELSE if A ≠ B
   IF  C  GT  E                 CMP  C , D
                                BLE  .2 ; skip [α] if C ≤ D
     : [ α ]                       : [ α ]
   IFEND                     .2  :
     : [ β ]                       : [ β ]
ELSE                         BR  .3     ; skip else clause
                         .1  :
   : [ γ ]                         : [ γ ]
IFEND                    .3  :
```

(a) before expansion　　　(b) after expansion

Fig.5 An example of macro expansion

label .3 by calling $n and generates

```
    BR  .3
.1:
```

The last IFEND macro pops up the stack using .3 as its label. In this way, $n macro and the stack are used to transfer information between macros.

To enhance the efficiency of the object code, ML-11 includes several facilities to control object codes. For example, user can specify which instructions should be used between Branch instructions and Jump instructions. ML-11 is implemented in about 60 man-days which include the period of the design, the coding and the debugging.

References:

1) D.E. Knuth: Structured Programming with Go To Statements, Stanford CS Report, 74-416 (1974).

2) E.C. Haines: A Structured Assembly Language, SIGPLAN Notices, Vol.8, No.1, pp.15-20 (1973).

3) N. Wirth: PL 360, A Programming Language for the 360 Computers, J. Association for Computing Machinery, Vol.15, No.1, pp.37-74 (1968).

4) BATCH-11/DOS-11 Assembler (MACRO-11), DEC (1973).