сообщения
объединенного
института
ядерных
исследований
дубна

E11-82-82

M.Rudalics

THE MICA/1 MICROASSEMBLER

1982

1. INTRODUCTION

Microassemblers are usually implemented on conventional architectures where time and storage constraints are not stringent. MICA/1 has been designed for a working place consisting of an Intellec [INTEL] Microcomputer Development System, two floppy-disk drives, a line printer and a PROM programmer [PROM]. Therefore some care had to be put in keeping it as small as possible without great losses of efficiency.

- MICA/1 uses the "Free Memory" in the LISP sense, abandoning the concept of tables and associated lookups.

- Assembling is of the one pass/one phase behaviour. Labels are inserted as soon as their definition has been encountered.

- MICA/1 does not require different definition and program parts. Instruction definitions and usages may be sequenced dynamically, with the only requirement that each definition has to precede its first usage.

- Keywords have been avoided while attempting to maintain readability and extensibility.

- MICA/1 does not require any temporary files during assembling.

This paper should be read as follows.

- Anyone who wants to work with MICA/1 should read chapter 2, chapter 5 and the User Manual [MICA] which will contain further examples, a formal syntactic description, a detailed description of I/O, an error listing, etc.

- Anyone who wants to extend, alter or adapt MICA/1 to his personal machine should additionally consult chapters 3 and 4.

A description of microprogramming and microassemblers in general lies beyond the scope of this paper. The reader will benefit from a study of [IEEE] for a general introduction into microprogramming, [CERN] for a comparative study of various microassemblers and [M&M] for the formal description of a meta-microassembly language.

2. FUNCTIONAL DESCRIPTION

MICA/1 provides two basic operations - definitions and usages - on three data types - variables, labels and instructions. Operations may be sequenced dynamically, with the only restriction that the definition of a variable or instruction has to precede its first usage. Variables and instructions may be freely redefined, however once an identifier has been associated with a specific type this may not be altered any more.

A basic building block will be further referred to as "value-length-pair". It consists of a value which has to be provided unconditionally and optionally a length operator followed by a length definition (in bits) which defines how many bits of the value have to be taken when a usage of the variable occurs. When no length has been provided, the minimal length for representing the value without truncating significant bits will be assumed automatically. Two length operators, " #" and " @" respectively, behave in the following manner:

- Equally, when the length indicator specifies a value equal to the minimal length to represent the value: the value and its length remain unchanged.
- Equally, when the length indicator specifies a value greater than the minimal length to represent the value, in this case zero filling on the left side will occur. The length of the value will be set to the length provided by the length indicator.
- Differently, when the length indicator provides a value less than the minimal value to represent the value: with " @" truncation of high order bits takes place and the length of the value will be set to the length provided by the length indicator, while " #" will cause a reset of the value to zero and of the length to one. In the latter case an error message will be issued.

Some examples may illustrate this:

```
3        . evaluates to three, the length indication has been
         . omitted, the default length for representing 3 are
         . two bits
3 #2     . evaluates to three
3 @2     . evaluates to three
         . the last three definitions are semantically
         . equivalent
3 # 4    . evaluates to three, with two leading zeros, as
3 @4     . evaluates to three, with two leading zeros
3 # 1    . evaluates to zero (reset) and issues an error
         . message
3 @ 1    evaluates to one, the high order bit dropped. No
         message will be issued.
```

Value and length indications may be provided by constants or variables with the difference, that the value of a length has to be bound when appearing in a definition, while the value of a value may be provided later (e.g., via a label definition). In this case a length indicator has to be specified by any means:

```
11^2 @1^16  . is equivalent to the last pair above, where
            . " ^" is the base designator, which has to be
            . followed by a base between 2 and 16, when omit-
            . ted, base 10 will be assumed
A# 4        . A may be bound already or be the value of a la-
            . bel yet to be encountered
3 #B        . B has to be bound in any case, otherwise an error
            . will occur
A #B        . an error will occur when B is unbound
A           . the length of this object is inherited from the
            . length of A, an error occurs, when A is unbound
```

A program for MICA/1 is built up of statements. Each statement has to be followed by a semicolon, spaces may be freely inserted anywhere, provided they do not separate syntactical units like identifiers or constants, and are obligatory between identifiers and constants. Comments - i.e., text preceded by a period and followed by a new line - and new lines may be inserted anywhere spaces are allowed. In the following examples each line is preceded by a line numbering as it appears on a list file created by MICA/1. Left from the line numbering a MICA/1 list file displays addressed and generated code, which does not appear in our examples.

Variable definitions appear as simple assignment statements, with an identifier on the left side and a value-length-pair on the right side of the "=" sign.

```
ØØØ1 A = 3;      . Binds A to three, length defaulted to two
ØØØ2 B = A# 6;   . binds B to A, adds two leading zeros
ØØØ3 C = B;      . binds C to B, C inherits the length six
ØØØ4             . from B
ØØØ5 A = 3 #B;   . binds A to three with one leading zero
ØØØ6             . as B will evaluate to three
```

Labels, which may head any statement, are identifiers followed by a colon. Their value is the number of the next instruction usage encountered. The length of a label is defaulted to the number of bits necessary to represent it without truncating significant bits. Labels, for obvious reasons may not be redefined:

```
ØØØ1 LAB1: A = 3 # 4;
ØØØ2 LAB2: LAB3:B = A; .LAB1, LAB2 and LAB3 will
                       .evaluate to the same value
```

Instruction definitions consist of an identifier followed by an "=" sign, followed by a value-length-pair which represents the instruction code (note that the value for the code has to evaluate immediately) and an optional parlist (whose elements will in fact be further referred to as parameters), which consists of an arbitrary number of value-length-pairs, separated by commas and enclosed within parentheses.

```
ØØØ1 INS = 3#4 (11^2, ØA^16 # 6, 2Ø @ 4);
ØØØ2 .defines an instruction with the following
ØØØ3 .default configuration:
ØØØ5 .ØØ1111ØØ1Ø1ØØ1ØØ (binary)
ØØØ6 .ØØ11 two leading zeros fill up 3 to length four
ØØØ7    .11 is 11^2 defaulted with length two
ØØØ8      .ØØ1Ø1Ø zero filled ØA^16 as length was six
ØØØ9           .Ø1ØØ high order bit of 2Ø dropped as
ØØ1Ø               .length has been indicated with 4
```

Note that this configuration will not be evaluated at define time. The code above is produced by MICA/1 when an instruction usage of the kind

```
ØØ11 INS;                   .use the default configuration
```

occurs in the text.

Generally an instruction usage is an identifier optionally followed by a parlist which consists of identifiers, constants and/or the wild sign " $ " separated by commas.

```
ØØØ1  INS ($ , 7, 24); .for the instruction defined above
ØØØ2 .will create the code:
ØØØ3 .ØØ1111ØØØ111Ø1ØØ
ØØØ4 .ØØ11  the code as above
ØØØ5     .11 as $ implies the default value
ØØØ6       .ØØØ111 is 7 right adjusted to six bits
ØØØ7             .Ø1ØØ is 24 the high order bit dropped
```

A label or variable usage is characterized by the appearance of its identifier within the parlist of an instruction usage.

```
ØØØ1 A = 7;
ØØØ2 INS = 3#4 (11^2, ØA^16 # 6, 2Ø @ 4);
ØØØ3 INS ( $ , 7, 24);        .as above
ØØØ4 LAB: INS ( $ , A, LAB); .while this instruction usage
ØØØ5  .will create the following code:
ØØØ6  .ØØ1111ØØØ111ØØØ1
ØØØ7 .ØØ11 for the instruction code
ØØØ8     .11 still the same
ØØØ9       .ØØØ111 A evaluates to seven and is adjusted
ØØ1Ø              .to length six
ØØ11            . ØØØ1 as LAB evaluates to one
ØØ12                  .adjusted to four
```

As we have seen so far, definitions are not always assignments in the usual sense of imperative programming languages. One difference is that when the value of a value-length-pair is a variable, its evaluation is postponed up to its usage (it may in fact never be evaluated at all).

Binding is static, i.e., each value is considered to be bound in its defining context. So delivers:

```
ØØØ1  A = 3;          .definition of A
ØØØ2  INS = 3 (A);    .definition of INS, A is bound to the A
ØØØ3                  .on the preceding line
ØØØ4  A = 2;          .redefinition of A
ØØØ5  INS ( $ );      .generates 1111
ØØØ6                            .11 for the instruction code
ØØØ7                              .11 as A evaluates to three
ØØØ8 INS = 3 (A);     .redefinition of INS, A is now bound to
ØØØ9                  .the A on line four
ØØ1Ø INS ( $ );       .generates 111Ø
ØØ11                            .11 for the instruction code
ØØ12                              .1Ø as A evaluates to two
```

Evaluation is additionally determined by the associated length indicator, thus:

```
ØØØ1 INS = 3 (Ø #6);
ØØØ2 A = 3;       .A bound to 3, length defaulted to two
ØØØ3 B = A # 6;   .B when evaluated will deliver three
ØØØ4              .with four leading zeros
ØØØ5 C = B @ 4;   .C will evaluate to three, with two
ØØØ6              .leading zeros
ØØØ7 D = C @ 1;   .D will evaluate to one, the high order
ØØØ8              .bit dropped during adjustment
ØØØ9 E = D # 6;   .E will evaluate to one, with five
ØØ1Ø              .leading zeros
ØØ11 INS (E);     .here this evaluation chain will be
ØØ12              .performed actually, code generated is
ØØ13              .11ØØØØØ1
```

while:

```
ØØØ1 INS = 3 (Ø #6);
ØØØ2 A = 3;          .same as above
ØØØ3 B = A # 6;      .same as above
ØØØ4 C = B # 4;      .will produce an error message, with
ØØØ5                 .reset to zero.
```

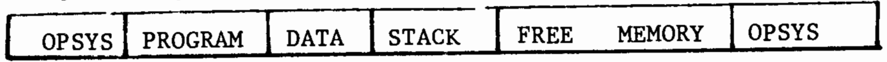
```
ØØ06 INS (C);   . will create the following code:
ØØ07            . 11ØØØØØØ
```

Note that the type of the length operator remains valid for later usages:

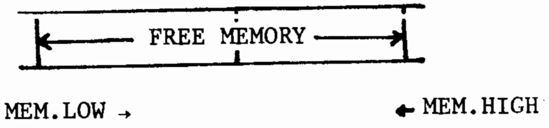
```
ØØ01 INS = 3 (3@2);
ØØ02 INS (4);   . truncates the high order bit of 4 without
ØØ03            . error message, and creates the code 11ØØ
```

3. THE MICA/1 STORAGE MANAGEMENT

Storage management is based on the usage of the free memory, similar to LISP like systems. The memory of the host computer may be divided up as follows:

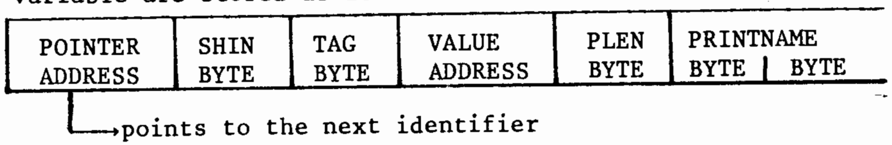


In MICA/1 occupation of the free memory is controlled by two pointers, MEM.LOW and MEM.HIGH respectively, where MEM. LOW specifies the next free location in the free memory and is advanced to the right when a new object has been created. Whenever MEM.LOM collides with MEM.HIGH, the free memory is full and assembling stops immediately (in a later version a garbage collector may part from here).



As far as memory management is concerned we distinguish the following data types:

Short Variables (<=65535) or variables bound to another variable are stored as follows:

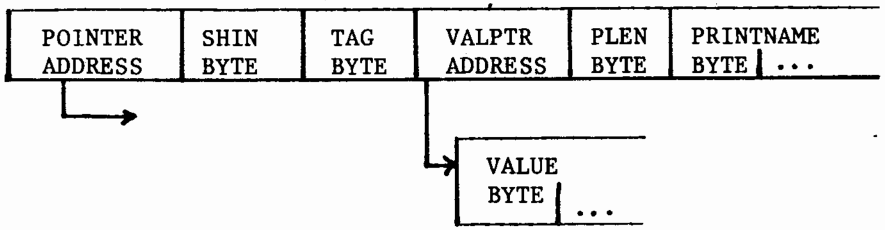


- All identifiers are linked in alphabetic order via the POINTER field. The POINTER field of the last identifier contains zero.
- SHIN and TAG bytes will be explained separately.

- The VALUE field either contains the value the variable is immediately bound to (with the low order byte in the first and the high order byte in the second byte) or contains a pointer to the variable the variable has been bound to.

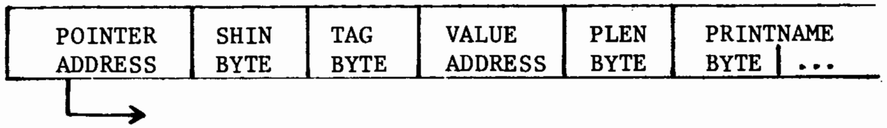
- PLEN indicates the number of bytes to store the printname.
- PRINTNAME contains the ASCII representation of the identifier´s printname.

Long Variables (> 65535) are stored as follows:



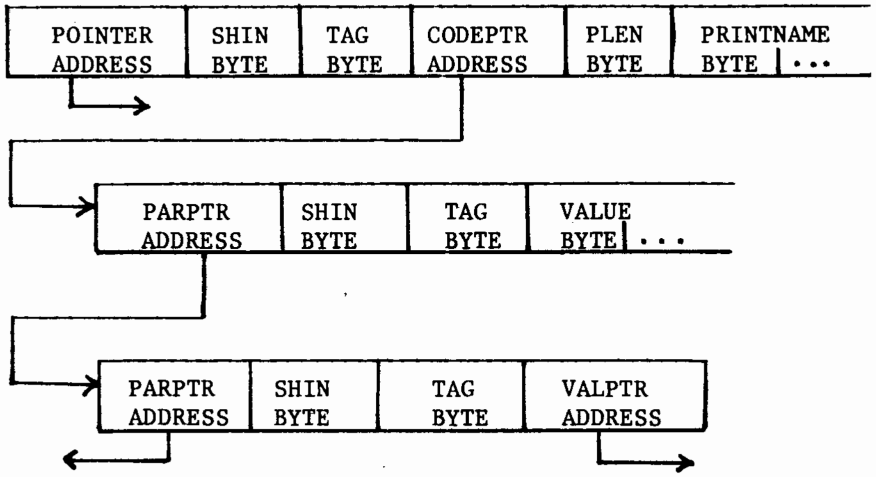
- VALPTR contains a pointer to the VALUE the variable is immediately bound to. Note that this VALUE may be stored anywhere in the free memory, and does not necessarily follow the PRINTNAME. A similar solution is planned for storing arithmetic expressions, etc.

Labels:



- Where VALUE contains the number of the next instruction usage.

Instructions:



- CODEPTR contains a pointer to the instruction code.

- The first field of the instruction code and of any following parameter contains a POINTER to the next parameter. In our example the value of the instruction code is stored immediately in its VALUE field, while the value of the first parameter has to be found on an address pointed by VALPTR.

Additionally values for (presumed) labels referenced in forward jumps - due to the one-pass behaviour - have been stored as follows:

POINTER ADDRESS	SHIN BYTE	TAG BYTE	VALPTR ADDRESS	DSHI BYTE	ERROR ADDRESS

points to the next presumed label

points to the value which has to be evaluated

...	LINE ADDRESS	PARNR ADDRESS	...

...	LSTBLCK ADDRESS	LSTBYTE ADDRESS	OBJBLCK ADDRESS	OBJBYTE ADDRESS

- DSHI indicates the location within the generated code, which has to be filled after the label has been resolved.

- ERROR points to the last error encountered so far.

- LINE contains the number of the input line where the label usage occured and is needed by the error handling routines.

- PARNR is the number of the parameter where the label appears, and is needed for error handling too.

- LSTBLCK, LSTBYTE, OBJBLCK and OBJBYTE refer to the position of the involved instruction on the list and object files respectively, and will be further explained in chapter four.

Similar solutions have been followed for storing errors:

POINTER ADDRESS	CHIFFRE BYTE	BYTE	BYTE	BYTE	IDPTR ADDRESS	..

to the next error

to the identifier involved in the error

..	LINE ADDRESS	PARNR ADDRESS

and Cross-References:

POINTER ADDRESS	LINE ADDRESS

points to the next reference

When a Cross-Reference Listing has been demanded, each data (instructions, variables and labels) is preceded by the following structure:

FIRST ADDRESS	LAST ADDRESS	LINE ADDRESS	.. (this is the POINTER field)

points to the first reference

points to the last reference

TAG-BYTES are interpreted as follows:

		Ø/1	Ø/1	ØØ // 11	ØØ //

The garbage collector bit (presently unused)
The dynamic evaluation bit (presently unused)
The " @" operator bit, when set truncation may occur in evaluation
The redefine bit, used in searching
The BIND bits
The TYPE bits

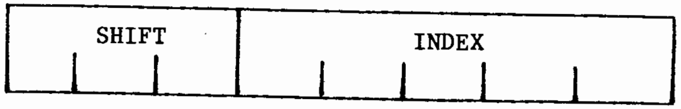
The TYPE bits are defined as follows:

ØØ	For a yet unresolved type
Ø1	For instructions
1Ø	For variables
11	For labels

The BIND bits are defined as follows:

ØØ For unbound
Ø1 For short variables
1Ø For long variables
11 For bound

The SHIN BYTE has to be interpreted as follows:

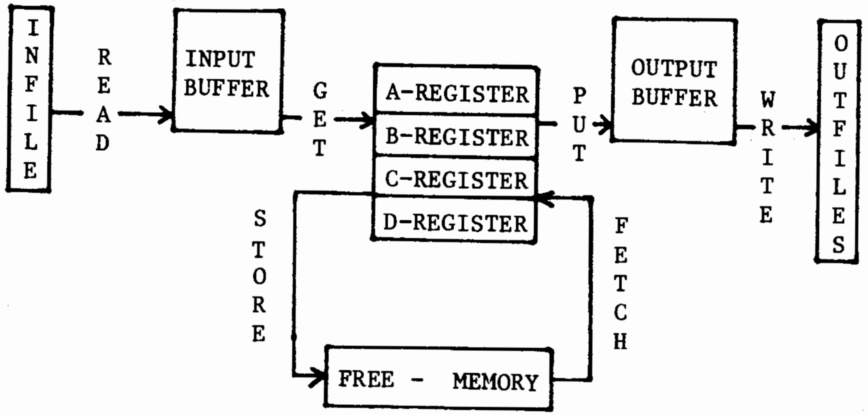


INDEX indicates the index of the first (highest order) byte occupied by the value, whereas SHIFT indicates the position of the first (highest order) bit in this byte.

As for the PLEN byte the maximal length for values and printnames is thus fixed with 256 ASCII-characters and bits respectively. Note that for all lengths stored in the SHIN/PLEN fields, due to the PL/M characteristic of starting array boundaries at zero, not the actual lengths but the index of the last element are stored. Thus these values may be used in DO-Loops operating on the associated structures.

4. IMPLEMENTATION DETAILS

MICA/1 operates on the following data structures (note that registers are not the hardware registers of the host computer, but software registers which however operate in a similar way).



The INPUT BUFFER is parsed in one pass, no character is read twice. Identifiers and constants pass through the A- or B-REGISTER before either being stored in the memory or passed through the D-REGISTER to the OUTPUT BUFFER. Values stored in the memory pass via the D-REGISTER, which serves actually for the code construction, to the OUTPUT BUFFER. The C-REGISTER serves for building the values of lengths according to which a value in the A- or B-REGISTER will be adjusted (remember that the value of a length has to be adjusted too). The A/B/C/D-REGISTERS are 256 bytes long and are headed by the following information:

BOUND BYTE	CHECK BYTE	DYNEV BYTE	INDEX BYTE	SHIFT BYTE	REGISTER BYTES..

True when the associated value is bound
True when truncation has to be checked (" # ")
Will be used for dynamic evaluation
The INDEX from SHIN
The SHIFT from SHIN

Note that values in the A/B/C-REGISTERS are stored according to general rules bytewise in ascending order, with the lowest order byte in register position Ø, while the D-REGISTER stores values in descending order with the highest order byte in position Ø. Zero filling occurs right hand side in all registers, within a byte however on the left in the A/B/C-REGISTERS and on the right in the D-REGISTER (INDEX and SHIFT have a different meaning for the D-REGISTER: INDEX denotes the presently filled byte and SHIFT the next bit to fill within this byte).

What follows is a description of some of the modules MICA/1 has been built of:

EXLOOP: Provides syntax check mainly, its length (about 2K-bytes of hex-code) is mainly due to the fact, that keywords have been avioded and occasionally a quite long string has to be parsed before its syntax has been recognized.

Error recovery is performed when either a") " or ";" are encountered, thus usually more than one statement should not get lost during syntactic analysis.

General Routines:

STORE-/FETCH-: Transfer values (and lengths) from the A- or B-REGISTER to memory and backwards, according to the basic design of the memory management system.

GET-(IS-)/PUT-: Perform I/O between A/B/C/D-REGISTERS and INPUT-/OUTPUT BUFFERS. Note that these modules make extensive use of ISIS-II [ISIS] System Routines like READ, WRITE,SEARCH, etc.

PREPARECODE: Is the code constructing routine. The code is established in the D-REGISTER.

SAVE-/RESOLVELABEL: Perform the saving and resolving of labels in forward jumps. In the present ("batch") version RESOLVELABEL is called once at the end of processing. The reason for this is mainly to save code as RESOLVELABEL is in the last overlay of MICA/1, and without a garbage collector immediate resolving (i.e., when a label is defined) would not be efficient (however the data/code tradeoff shall be considered in the near future). At the moment the following two strategies have been choosen to remedy crowding of the free memory with too much unresolved labels.

(1) Whenever a label definition is encountered after its identifier has already appeared in an instruction definition, its corresponding TAG is immediately set to bound. This provides that only labels in forward jumps are saved for later resolution, on the other hand a program may contain in its definition part a lot of unresolved labels without giving any warning.

(2) When more than one unresolved label occurs in one instruction usage, the addresses of the code on the object and list files respectively (OBJBLCK, OBJBYTE,..) are remembered only once. This however operates correctly only in the present environment, where labels are resolved in the final step.

Assistant Routines:

ALLOCATE: Performs memory allocation by advancing MEM.LOW and Comparing it with MEM.HIGH. The message "MEMORY FULL-ASSEMBLING STOPPED" is issued from here.

ADJUST: Adjusts a value in the A- or B-REGISTER, according to a length in the B- or C-REGISTER. When the length indicated is greater than the minimal length to store the value, zero filling on the left occurs. Otherwise either an error message with reset to zero (when " # " has been used) or cutoff (with the " @ " operator) takes place.

COPYSHIFTINDEX: Takes a value stored in a register and stores it into the INDEX/SHIFT field of this register. Note that a length zero may not be represented.

LOOKUP: Returns (if successfull) a pointer to the searched identifier.

INSERT: Inserts an identifier in the id-list, no insertion is performed (and an error message is printed) when the identifier is already in use for a different type.

EVAL: Is an inherently (non tail-) recursive procedure, operating on the A- and B-REGISTER. In the calling part it follows the backward chain of definitions and in the tail part it calls COPYSHIFTINDEX to establish the length in the C-REGISTER and ADJUST to adjust the value in the A- or B-REGISTER according to this length. It returns false when the deepest value has been encountered unbound. This will cause the saving of a label during PREPARECODE and an error message when encountered in RESOLVELABEL.

5. ADDITIONAL REMARKS

MICA/1 programs require no special headers or EOF marks. A missing ";" after the last statement however will not appear in the error listing but cause a message on the console: "PREMATURE EOF ENCOUNTERED" which may be considered insignificant. Note that the semicolon to conclude statements is basically redundant, and serves only for recovery purposes in the syntax check.

The "ORG" (ORIGINATE) statement has been realized as a constant followed by a colon, and may head any statement (even be intermingled with labels). When its value is greater than the present code line, all lines up to the line indicated by the ORG statement are filled with zeros, otherwise an error message will be returned.

MICA/1 has been written in PL/M-80 (PL/M-8Ø) on an INTELLEC MDS. PL/M-8Ø has been the only high-level language available providing based variables (PASCAL existing only in the interpretative version). We hope to obtain a PASCAL compiler in the near future and to rewrite the whole system in PASCAL, thus being able to run it on other architectures. Note that MICA/1´S only requirements are direct addressable bytes and the possibility to use up the free memory. MICA/1 has run through its development the following design steps:

(1) The GET-(IS-) and PUT-routines have been designed, using the ISIS-II interfaces READ, WRITE and SEEK mainly.

(2) The external loop, thus being able to work with "real" files already.

(3) The FETCH- and STORE-routines.

(4) The actual code building routines.

(5) Saving and resolution of labels.

(6) Elimination of parameters, this caused a saving of some 4K of memory with some loss of program readability, however interfaces in PL/M-8Ø had never been very clean at all.

The present version of MICA/1 uses about 2ØK bytes of main storage. A "natural" overlay (dividing MICA/1 in an input, mai and output step) would reduce this to some 15K and allow MICA/1 to run on 32K byte. A second overlay structure with however an already drastic increase of load operations should run on 12K with some 2K of working storage.

The forthcoming MICA/2 will provide two additional binding mechanisms, dynamic- and let-binding respectively. To maintai compatibility with future versions you should refrain from using the following features provided by MICA/1:

- Factored variable and instruction definitions.
- Omitting commas between parameters in instruction usages.
- Symbolic names like: "DO", "END", "IF", "CALL", "RETURN" and others known from high-level languages.

ACKNOWLEDGEMENTS

We thank Professor N.N.Govorun for his support of our work Angelika and Holger Leich have been of substantial help durin the implementation of MICA/1.

REFERENCES

(CERN) Halatsis C. Software Tools for Microprocessor Based Systems. Proceedings of the 1980 CERN School of Computing. Geneva. 1980. pp.241-281 (1981).

(IEEE) Rauscher T.G., Adams P.M. Microprogramming: a Tutorial and Survey of Recent Developments. IEEE Transactions on Computers. Jan., 1980, vol.C-29, No.1, pp.2-20.

(INTEL) INTELLEC MDS OPERATOR'S MANUAL. INTEL CORP., DOC. NO.98-132, 1976.

(ISIS) ISIS-II USER'S GUIDE. INTEL CORP., DOC.NO.98-3068, 197

(MICA) The MICA User's Manual (in course of preparation).

(M&M) E.Skordalakis. Towards a More Flexible Micro-Language for Bit-Sliced Microcomputers. Microprocessing and Microprogramming. Jan., 1981, vol.7, No.1, pp.46-57.

(PL/M-80) PL/M-80 Programming Manual. INTEL CORP., DOC.NO. 98-268, 1976.

(PROM) UNIVERSAL PROM PROGRAMMER REFERENCE MANUAL, INTEL CORP., DOC.NO. 98-133, 1976.

Received by Publishing Department
on February 2 1982.

WILL YOU FILL BLANK SPACES IN YOUR LIBRARY?

You can receive by post the books listed below. Prices - in US $, including the packing and registered postage

D9-10500	Proceedings of the Second Symposium on Collective Methods of Acceleration. Dubna, 1976.	11.00
D2-10533	Proceedings of the X International School on High Energy Physics for Young Scientists. Baku, 1976.	11.00
D13-11182	Proceedings of the IX International Symposium on Nuclear Electronics. Varna, 1977.	10.00
D17-11490	Proceedings of the International Symposium on Selected Problems of Statistical Mechanics. Dubna, 1977.	18.00
D6-11574	Proceedings of the XV Symposium on Nuclear Spectroscopy and Nuclear Theory. Dubna, 1978.	4.70
D3-11787	Proceedings of the III International School on Neutron Physics. Alushta, 1978.	12.00
D13-11807	Proceedings of the III International Meeting on Proportional and Drift Chambers. Dubna, 1978.	14.00
	Proceedings of the VI All-Union Conference on Charged Particle Accelerators. Dubna, 1978. 2 volumes.	25.00
D1,2-12450	Proceedings of the XII International School on High Energy Physics for Young Scientists. Bulgaria, Primorsko, 1978.	18.00
D-12965	The Proceedings of the International School on the Problems of Charged Particle Accelerators for Young Scientists. Minsk, 1979.	8.00
D11-80-13	The Proceedings of the International Conference on Systems and Techniques of Analytical Computing and Their Applications in Theoretical Physics. Dubna, 1979.	8.00
D4-80-271	The Proceedings of the International Symposium on Few Particle Problems in Nuclear Physics. Dubna, 1979.	8.50
D4-80-385	The Proceedings of the International School on Nuclear Structure. Alushta, 1980.	10.00
	Proceedings of the VII All-Union Conference on Charged Particle Accelerators. Dubna, 1980. 2 volumes.	25.00
D4-80-572	N.N.Kolesnikov et al. "The Energies and Half-Lives for the α - and β-Decays of Transfermium Elements"	10.00
D2-81-543	Proceedings of the VI International Conference on the Problems of Quantum Field Theory. Alushta, 1981	9.50
D10,11-81-622	Proceedings of the International Meeting on Problems of Mathematical Simulation in Nuclear Physics Researches. Dubna, 1980	9.00

Orders for the above-mentioned books can be sent at the address:
Publishing Department, JINR
Head Post Office, P.O.Box 79 101000 Moscow, USSR

SUBJECT CATEGORIES
OF THE JINR PUBLICATIONS

Index	Subject
1.	High energy experimental physics
2.	High energy theoretical physics
3.	Low energy experimental physics
4.	Low energy theoretical physics
5.	Mathematics
6.	Nuclear spectroscopy and radiochemistry
7.	Heavy ion physics
8.	Cryogenics
9.	Accelerators
10.	Automatization of data processing
11.	Computing mathematics and technique
12.	Chemistry
13.	Experimental techniques and methods
14.	Solid state physics. Liquids
15.	Experimental physics of nuclear reactions at low energies
16.	Health physics. Shieldings
17.	Theory of condenced matter
18.	Applied researches
19.	Biophysics

Рудалич М. E11-82-82
MICA/1-микроассемблер

Предлагается универсальный язык для описания в мнемонической форме алгоритмов работы устройств с микропрограммным управлением, а также рассматривается программа преобразования элементов языка в микрокод /микроассемблер MICA/1/.

Работа выполнена в Лаборатории вычислительной техники и автоматизации ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна 1982

Rudalics M. E11-82-82
The MICA/1 Microassembler

MICA/1 is a Universal (Meta-) Microassembler implemented in PL/M-80 on an Intellec Development System. The following description of MICA/1 contains a general part, explaining the definition and usage of microinstructions and special parts on memory management and implementation. MICA/1 has been developed at the Laboratory of Computing Techniques and Automation of the Joint Institute for Nuclear Research in Dubna.

The investigation has been performed at the Laboratory of Computing Techniques and Automation, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna 1982