Institut fur Informatik and Praktische Mathematik

Christian-Albrechts-Universitat Kiel

Olshausenstraße 40-60

D - 2300 Kiel 1

Tel. (0431) 880 - 4461

MIMOLA REPORT

Revision 1 and

MIMOLA SOFTWARE SYSTEM USER MANUAL

CONTENTS

Page

1. INTRODUCTION

This Revision 1 replaces the original report (1). The syntax of MIMOLA has been slightly changed and extended. The hardware declaration and assignment parts have been redesigned. A MACRO feature has been added. A description of the hardware database and the control language of the MSS (MIMOLA Software System) has been added (2,3,4).

MIMOLA is a computer hardware description language (CHDL) and a programming language. It has been developed for the following applications:

a. Nonprocedural description of hardware (especially computer hardware), for declarations (e.g. appl. b.-d.), for education and documentation

b. Functional description of digital systems, **procedural, for top-down** design, education and documentation

c. Algorithmic description of digital processors for optimizing top down hardware design

d. High-level or intermediate microprogramming language for p-code generation

e. Modelling of algorithms or machine instructions on a state transition level for measurements and comparisons.

Other applications are possible. It is not the task of this REPORT to show how problems can be solved using MIMOLA. In chapter 2 some examples are explained to show the effectivity of MIMOLA. More details can be found in (5,6) or will be published.

At this point we will summarize some features of MIMOLA to give a frame for the details in the following chapters.

1. The hardware is mainly described on the <u>register transfer level.</u> Lower levels (e.g. the gate level) can be described, but the notation will not be optimal for this purpose. All modules that are of interest. in the construction of computers today and in the near future, have been included in MIMOLA as language primitives. Language extensions to new modules can be made by macro definitions or syntax extensions. Thus MIMOLA supports mainly modular and structured hardware solutions with a small number of different modules and simple interfaces, but admits also sophisticated special structures for unusual problems.

2. The <u>functional description level</u> is strictly the state transition level of a synchronous automaton. This is normally called the microprogramming level. Thus a very close connection between programs and hardware is achieved.

3. Parallelism or <u>concurrency</u> can be expressed in the range of one state transition. Besides this constraint the limits of parallel execution are given by the hardware features only. The execution of parallel or spatial sequential operations is asynchronous , as long as no states are changed. Thus concurrent statements need not be order-invariant (as e.g. in ISPS (7)).

This means that all set and store operations are executed synchronously, thus avoiding racing problems. All other operations are thought to be executed by networks with only one permanent state. Asynchronous feedbacks are prevented by the syntax of MIMOLA.

In some cases operations are too complex to be executed by a network with a reasonable amount of hardware. A compromise can be found by admitting the modules to have internal states not visible to the outside. The activity of "impure" modules has tc be controlled by additional signals. It is assumed that the control unit generates an execution sequence as in a data flow graph. Thus no synchronization problems will arise.

4. Asynchronous parallelism including more than one state transition have been excluded as a language primitive. Further investigations are necessary to find a general solution for expressions of this kind on the register transfer level.

Asynchronous parallelism on the processor level can be expressed by distinct MIMOLA programs with appropriate synchronization macros. As long as no method exists to distribute algorithms on more than one processor automatically, our design methods are sufficient to desiqn and optimize one processor at a time.

5. For the use of MIMOLA as a high level programming language a macro facility has been included. Thus expressions are allowed that cannot be directly or uniquely be built in hardware.

Standard macros like IF.. THEN ..., CASE, FOR.... CALL are parts of the syntax. The semantics is assumed to he near to the usual one. The

MIMOLA to his ideas or to a special HLL.

Additional user macros can be used to extend the language, to introduce unusual constructions (e.g. synchronization primitives). Thus experiments with different macro-replacements are possible.

6. <u>Data types</u> have been included only as far as they concern the hardware. This is due to our main applications b. and c. Words of different sizes, fields of **words and concatenations** of words or fields are the only data structures in MIMOLA that can directly be translated into hardware.

Array element references are included as a standard macro. Field declarations can be used for PASCAL RECORD-fields. Distinctions between different scalar data types can be expressed by different memory module names and different operator functions. Different memory names can also be used to express a difference between local and global variables. This represents no limitation for the hardware design space, since modules with different names can be easily merged by an edit process.

7. As CAD systems tend to become too large and inefficient, a LR(1) grammar (Appendix B) has been chosen with additional restrictions to simplify the syntax analysis.

These features could not be found in any other language. This is the only reason for the definition of a new language. It was originally written for our own research. But since our

MSS is written in PASCAL, it can be run on many installations. This may encourage other groups to use MIMOLA together with MSS.

2. APPLICATIONS

MIMOLA was designed as a tool. Therefore examples for the applications listed in Chapter 1 are given. They are partly selfexplaining. For details refer to the appropriate chapters in this report.

2.1. Nonprocedural Description of Hardware

The simple processor in fig. 2.1 can be expressed by the DECLARATION in example 2.1.1

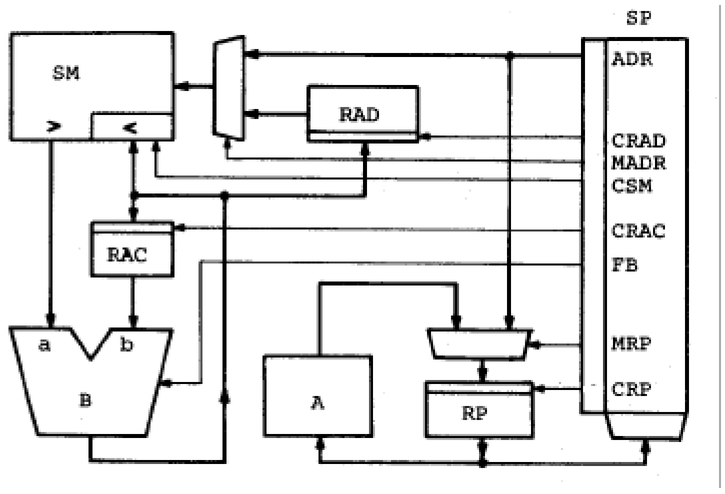


Fig. 2.1 Simple Processor

```
DECLARE
ADDMODUL
    SM<A(16384:0).BIT(15:0), SM>A
    "main memory, 16k, 16 bits, 2 ports",
    SP(1024:0).BIT(21:0) "microprogr.mem.",
    RAC.BIT(9:0) "accumulator",
    RAD.BIT(15:0) "address register",
    RAC.BIT(15:0) "accumulator",
    A(.INCR) "monadic operator",
    B(+,-,.a,.b) "adder",
    I.BIT(21)CRP.BIT(20)MRP.BIT(19:18)FB
     .BIT(17)CRAC.BIT(16)CSM.BIT(15)MADR
     .BIT(14)CRAD.BIT(13:0)ADR
    "microprogram word, field names";
ADDCONNECTION
    "declaration of data, address and
    control paths"
    B<a <- SM>A, "memory out to adder input"
    B<b <- RAC,
    B.FCT <- I.FB "microprogram field to
    adder function control",
    RAC <- B "adder out to accumulator",
    RAC.CON <- I.CRAC "accu load enable",
    RAD <- B, RAD.CON <- I.CRAD,
    SM<A <- B, SM<A.CON <- I.CSM,
    SM.ADR <- I.ADR /RAD "address field
    and address register via multiplexer
    to address port of memory",
    SM.ADR.MPX <- I.MADR "address multi-
    plexer control",
    RP <- A / I.ADR, RP.CON<- I.CRP,
    RP.MPX <- I.MPR,
    A <- RP, SP.ADR <- RP;
ENDDECLARE
```

Example 2.1.1 : Declaration of the Processor in Fig. 2.1

Besides some additional informations, the description in example 2.1.1 has no value of its own, because Fig. 2.1 gives a clearer view of the structure. In the MIMOLA design systems hardware descriptions are used as an input form for computer aided designs and transformations. The DECLARATION can be used to show that the processor in Fig. 2.1 is able to execute a given set of functions. The MSS would give an error output otherwise. It can also be used to translate algorithms to microprograms for this processor.

Another way of hardware description is of more interest: the definition of upper limits in the design space (Example 2.1.2). The meaning is: one memory SM with a maximum of 6 ports is the only memory. No more than three dyadic operators are allowed, B3 with a limited function set. All other recources are not limited. This uncomplete description is the normal way to interact with the MSS process.

```
DECLARATION

ADDMODUL
SM(65/535:Ø). WORD.MOREPORT(6),
B1,B2,B3 (+,-);
```

Example 2.1.2

2.2. Functional Description

A computer can well be characterized by a description of its machine instruction set. Only a part of the hardware is visible in this description. The hidden part is of no direct importance to the function the user of the computer sees. This "functional description" can be formalized by a CHDL. Due to our familiarity with programming languages procedural descriptiors are more natural to express sequential microprogram steps or state transitions than nonprocedural ones. The description level depends on our purpose.

```
LSTAA   SM(m/RIX -> B(+)) : = RACCA,
        RCOND.N:= RACCA.BIT(7),
        RCOND.Z:= RACCA -> A(=O),
        RCOND.V:= O;

        "RACCA   accumulator A
         RIX     indexregister
         RCOND   condition register"
```

Example 2.2.1 : Functional Description of the M68OO Instruction "STAA m,X"

Example 2.2.1 shows a high level description of a MOTOROLA M6800 microprocessor instruction. It is sufficient for an ASSEMBLER programmer to understand the function of the "store accumulator A indexed in memory, address m". In a design process this description opens the greatest design space

```
LSTAA     SM(SM(RPC -> A(.INCR))/RIX ->B(+))
                   := RACCA,
          RCOND.N := RACCA.BIT(7),
          RCOND.Z := RACCA, -> A(=0),
          RCOND.V := 0,
          RPC     := RPC/2 -> 13(+);

          "RPC    program counter"
```

Example 2.2.2 : STAA m,X

Example 2.2.2 gives more details about the instruction format and the program counter RPC behaviour.

```
LSTAA1      RIR := SM(RPC),
            RPC := RPC -> A(.INCR);
LSTAA2      RHLP:= SM(RPC),
            RPC := RPC -> A(.INCR);
LSTAA3      RHLP:= RHLP/RIX -> B(+);
LSTAA4      RCOND.N:= RACCA.BIT(7),
            RCOND.Z:= RACCA -> A(=0),
            RCOND.V:= 0;
LSTAA5      SM(RHLP):= RACCA;

            "RIR   instruction register
             RHLP  internal register"
```

Example 2.2.3 : Possible μ-instruction sequence of "STAA m.X"

In Example 2.2.3 the instruction is resolved in 5 microstatements. This might be the execution sequence of the M6800 and a description of all instructions of the M6800 in this manner would lead to a structure very close to this microprocessor.

It can be seen from these examples, how useful a functional description of a computer in MIMOLA may be in documentation and education. But every level can also be used as an input to the MSS to find different hardware structures. These will meet the requirements correctly and can be optimized with different constraints and goals.

2.3 Algorithmic Description

It has been shown in 2.2 that different description levels are possible with one language. **without passing a sharp** border we can increase the level of example 2.2.1. User problems seldom bother with details like data storage in registers. Normally transformations are applied on variables or more general data structures. The translation to register load and store operations is a necessity due to the lack of more powerful or simply suitable instructions.

If we want to design optimal structures from the users view, we must start on the users level. Problems can only be solved by computers using algorithms. Therefore a description on the "algorithmic level" is the main application of MIMOLA. Example 2.3.1 shows a short program in three different languages. The postfix-notation of MIMOLA may be unusual, but the correspondences can be found easily. The differences to example 2.2.1 are only gradual. But the point of view differs:

Example 2.2.1 describes the real function of the hardware. The only uncertainty is the probability of occurance. All functions can be listed completely.

PASCAL

```
min  := list [1]; max:= min;
for i:= 1 to (n div 2)-1     do   (* n odd *)
   begin p:= list [2*i]; q:= list [2*i+1];
   if p>q then
     begin if p > max then max:= p;
           if q < min then min:= q
     end
   else
     begin if q > max then max:= q;
           if p < min then min:= p
     end
   end;
```

FORTRAN

```
        min = list (1)
        max = min
        DO  20  i = 2,N-1,2
        p = list (i)
        q = list (i+1)
        IF (p.LE.q) GOTO 10
        IF (p.GT.max) max = p
        IF (q.LT.min) min = q
        GOTO 20
10      IF (q.GT.max) max = q
        IF (p.LT.min) min = p
20      CONTINUE
```

MIMOLA

```
L1       S(min):= S(list[1] ),
         S(max):= S(list[1] ),
         FOR i FROM 2 BY 2 TO S(n)->A(.DECR);
L2       DO(i)
            IF S(list [D(i)])= V1/
               S(list [D(i)->A(.INCR)])= V2
              ->B(>)
            THEN IF V1/S(max)->B(>)
                 THEN S(max):= V1 FI,
                 IF V2/S(min)  ->B(<)
            THEN S(min):= V2 FI
            ELSE
                 IF V2/S(max)->B(>)
                 THEN S(max):= V2 FI,
                 IF V1/S(min)->B(<)
                 THEN S(min): = V1 FI
            FI,
            OD(i);
```

Example 2.3.1

Example 2.3.1 shows a possible function. We can estimate the probability but we cannot give a limited list of all algorithms. This is an additional degree of uncertainty. It can partly be overcome by using large samples to be able to calculate "precise" mean values. Due to the lack of knowledge about user behaviour and therefore the difficulty of giving an exact task description of the design object, "precise" is very relative.

Large program samples ask for a descriptive language with high-level language features. This is an unusual demand for CHDL's but had to be met by MIMOLA.

A set of algorithmic descriptions define an automaton or hardware structure. The microinstructions of these programs cause state transitions of this automaton. Different automatons can be found by transforming these programs. On the other hand, manual changes of this automaton (by declarations, see 2.1) can cause transformations of the programs to preserve the ability of execution.

Thus the hardware can be tailored to meet constraints and a proof is given at the same time about the correct execution of the programs on this hardware. Since the programs form the task description, the correctness of the solution can be proven.

To find an optimum, the variations of the hardware are not done arbitrarily. As the design space is too large to-try all possibilities, occurence probabilities are calculated for all resources (e.g. modules, connections, instruction word fields) to guide the variations.

This is a very short description of our design method. It is implemented to a large extend in the MSS (MIMOLA Software System). The use of functional descriptions (Chapter 2.2) is a special case of the method. A better description is (6).

2.4 Microprogramming Language

A welcome byproduct is the possibility to use MIMOLA as a high-level microprogramming language. This is due to the fact that the state transition level is a basic language feature and is preserved during all transformations. For all MIMOLA programs an automaton or hardware structure exists that can execute the microstatements of the programs without further transformation. As already mentioned in 2.3 a change of the automaton causes a program transformation. A complete declaration of a computer structure (see chapter 2.1) can be seen as a change of this kind. Thus the MSS will respond with transformed programs executable on this structure. Since these programs describe state transitions, they contain the microcode in a special form. Some decoding and software tasks like storage management have to be added to change the MSS to a microprogram compiler. This application is under investigation.

3. SYNTAX

The syntax of MIMOLA is defined in two ways. The user can refer to the syntax diagrams in Appendix A. The syntax analyzer of the MSS (MIMOLA Software System) takes a production system in Backus notation, listed in Appendix B. It is an LR(1)grammar.

Additional rules are listed in Appendix C. They are used by a preprocessor in front of the syntax analyzer.

Part 1 is necessary to guarantee correct hardware functions.

Part 2 is additionally required to suppress meaningless programs.

A violation of theses rules does not necessarily lead to hardware errors.

The symbol set of MIMOLA is 96 characters of ASCII. Provisions are made to allow for 64 character sets. Throughout this report the following meta symbols are used:

::=	equivalence in Backus notation
< >	include nonterminal symbols
" "	include representatives of strings of terminal symbols
$\{\ \}_n^m$	contents may be repeated at least n times and at most m times (BNF)
*	A star * is used if the contents may be repeated indefinitely.

symb-1|symb-2 means: symb-1 <u>or</u> symb-2

For different purposes four special adapted sublanguages exist:

assignment language, declaration language, macro language, program language. The grammars of these languages differ only slightly. The differences are marked in the syntax diagrams and the list of rules and will be explained in the following chapters.

A MIMOLA string may contain a sequence of different language parts. Fig. 3.1 shows the endsymbols.

State	Control key	Endsymbol
Assignment	ASSIGN	ENDASSIGN
Declaration	ADDMODUL / ADDCONNECTION	,
Macro	**MACRO**	ENDMACRO
Program	PROGRAM	END / ENDSUB

Fig. 3.1

Different language parts may be nested either by writing a $-sign into the current source string or by using preprogrammed breakpoints (see chapter 9.1). Both conditions cause MSS to expect a (nested) new command key. Example 3.1 gives valid combinations of control commands.

```
 ADDMODUL    R1,R2 ;              "level 1: addition of modules"
 PROGRAM                          "level 1: start of program"
 begin LØ R1:= R2;
$ASSIGN a:= Ø; ENDASSIGN          "level 2: nested assignment"
 L1 S(a)= R1 ;                    "continue with level 1"
 end                              "end-of-program causes the
                                   command to be read from
                                   terminal for non-BATCH jobs"
 PRINTHARDWARE
 EXIT                             "exit from MSS"
```

Example 3.1

The set of allowed control commands depends on the context. It is not allowed to put two or more identical control commands in one nesting hierarchy (e.g. a PROGRAM part may not contain a PROGRAM command).

Assignment parts provide means to define software equivalences with no direct influence on the hardware. They are tools for program structuring and microcode generation, syntax analysis starts after "ASSIGN". A correct assignment part can be reduced to the assign-axiom, when the endsymbol "ENDASSIGN" has been found. If equivalences are not changed by other assignments, they are valid until exit from MSS.

Declaration-parts are used for unprocedural definitions or changes of hardware structures. For correct parts a syntactical reduction to the declaration-axiom is possible, when the endsymbol "; " has been found. Hardware structures are valid until exit from MSS if they are not changed by

ADDMODUL and ADDCONNECTION commands or deleted with DELMODUL and DELCONNECTION commands.

<u>Macro parts</u> allow the declaration of macro equivalences and replacements (see chapter 8). Definitions start with $MACRO and stop with ENDMACRO. Macro ranges are blockoriented in order to allow easy conversion of variable declarations of high-level programming languages. A block is opened by a $[\$]_0^1$ BLOCK command and closed by $[\$]_0$ ENDBLOCK. The first block is preopened.

The <u>program parts</u> contain the algorithmic or functional descriptions. Program parts generate hardware descriptions and statistical analyses if applied to the MSS. Program parts may be subdivided in main program parts, enclosed in BEGIN ... END, and subroutines enclosed in SUBROUTINE ... ENDSUB. The syntax analysis is separately applied if these parts are disjunctive. Nested subroutines are analysed together with the enclosing program.

4. HARDWARE DATA STRUCTURE

The MSS stores all information about the available hardware together with the statistical information in a hardware data structure (hds). The structure is similar to the network model of data bases.

Most of the entities are hierarchically ordered:

4.1. Group, Module

All modules with the same initial letter form a group.

The MSS recognizes the following groups:

A	monadic operator,
B	dyadic operator,
C	triadic operator,
D	Do - Loop - variable storage,
F	hardwired constant,
I	microinstruction,
K	stack,
N	network,
R	register,
S	storage (RAM),
V	non-stored result.

4.2 Port Input

A module has at least one port. All input and output is done

via ports. There are input ports, output ports and bidirectional

ports: Portdirections may be specified by ' < ', ' > ' or '<

a fixed number of ports (e.g. dyadic operators have three: two for input and one for output); others have a varying number of ports (e.g. random access memories). There are certain limitations to the set of allowed portnames and directions, depending on the group which the module belongs to:

A, R, V	: no	port-name (blanks),
F, I, D	: no	port-name, only output (>),
S, K	: no	restrictions,
B	. >,	<a, <b ,
C	>,	< a, <b, < c,
N	>,	<a, <b, <c, , <Z.

The control language command NOLOWERLETTER changes the above lower case letters to upper case letters.

Each port has up to four inputs or outputs: a function input, **an address input,** a control input and a data input or output. They are selected by the reserved attributes .FCT, .ADR, .CON and .DAT. The latter is assumed by default.

4.3. Field

Inputs and outputs have fields. These may be ranges of bitnumbers or attributes which stand for unassigned ranges of bits (called bitattributes). Any attribute which is not a predefined attribute, is considered to be a bitattribute. The bitrange of a bitattribute can be defined in the ADDMODUL declaration.

Input fields can be viewed as multiplexers if there is more than one connection for a field. In this case the field has an associated multiplexer address input field which the user can select with the .MPX attribute.

4.4. Connection

Connections in their most general form are concatenated subranges of data fields of outputs. Connections are the hds representation of bundles of wires connected to a destination. For example the block diagram of Fig. 4.4.1 is transformed into the nodes of Fig. 4.4.2.

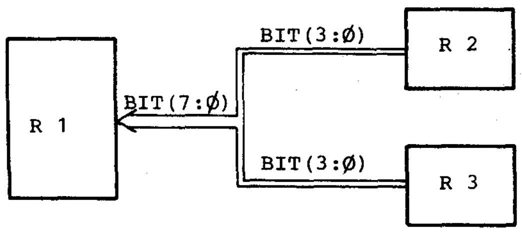


Fig. 4.4.1 Hardware Block Diagram

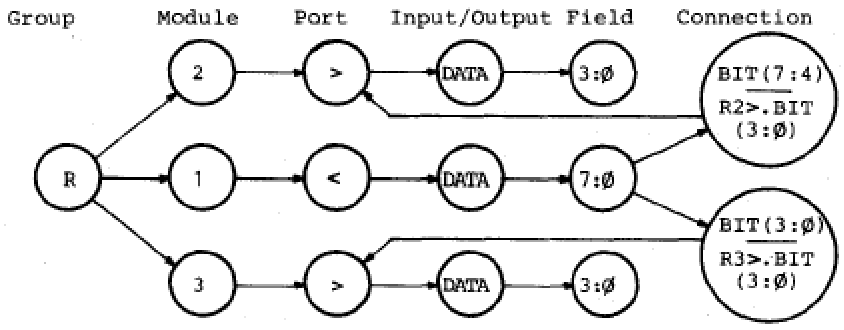


Fig. 4.4.2 Corresponding hds Structure

The hds description of connections contains independent ranges for source- and destination bits. See chapters 5.3. and 5.4. for additional information.

4.5. Data Structure Entries

The following is a list of some of the entries in the just described hierarchical part of the hds:

group :	boolean value indicates whether or not a module of this group has been declared in the declaration or not.
module :	number of possible duplicates, pointer to joint distribution table, number of additional ports allowed, frequency of use, distribution of concurrent uses of ports of this module.
port :	frequency of use, pointer to entry in joint distribution table (S and K only), list of functions, default function, boolean value indicating the right to add more functions to the list of functions.
field :	frequency of use, pointer to entry in joint distribution table (uinstruction only) symbolic value (microinstruction only)
connection:	frequency of use, multiplexer address, inhibit flag from $DELCONNECTION command.

Other tables in the MSS are:

1. Second order joint distributions for the use of modules and storage ports.
2. Second order joint distributions for the use of the microinstruction fields.

- All the above tables are.listed by $PRINTHARDWARE and $TYPEHARDWARE.

3. Frequency of function uses, listed by $PRINTFUNCTIONS and %TYPEFUNCTIONS.
4. Overlapping status for bitattributes.
5. Label table.
6. Identifier table (not yet implemented).

5. DECLARATION

5.1. Declaration of Modules, Ports and

5.1.1. Addition

Hardware modules, their ports and multiplexers can be declared with the $ ADDMODUL command. The command key must be followed by a list of hardware descriptors.
The general form of the syntax is:

$\{\$\}_0^1$ ADDMODUL <descriptor> $\{,$ <descriptor> $\}_0^*$;

<descriptor>::= <modulename> $\{$ <direction>|<direction><portname>$\}_0^1$

$\{$<functionlist>|<addressrange> $\}_0^1$ $\{$. <attribute >$\}_0^*$

<functionlist> ::= (<function> $\{,$ <function> $\}_0^*$)

<addressrange> ::= (<unsigned integer> : <unsigned integer>)

Complete syntax rules are given in Appendix B.

```
ADDMODUL
      S <> (# FFFF:Ø).BIT (31:Ø)WORD,
      "RAM of 64K cells of 32 bits which are named WORD"
      BALU (+,-,.AND,.OR,.XOR).BIT(32)CARRY.WORD,
      "arithm.-log. unit with functions and a CARRY output"
      BALU < a. WORD. INPUTMAX (2). NOMOREFIELDS;
      "the a input of BALU is 32 bits wide, has a maximum
      of two inputs to the multiplexer, no other fields
      are allowed"
```

Example 5.1.1

All ports, inputs, and outputs may be specified separately. If no direction is present, output is assumed. Default portname is 'A' for RAMs and stacks and (blanks) otherwise. .DAT is default for inputs and outputs.

Continuous bit ranges used in ADDMODUL will not be concatenated. R1.Bit (7:3,2:0) will generate two fields if used in ADDMODUL, but only one field BIT(7:∅) if it is a creating call in a PROGRAM part. A list of reserved attribute names is used for entering more information into the hardware data structure:

a) DUPLICATE (ui)   /NODUPLICATE

DUPLICATE means: this module may be duplicated ui times if necessary. If ui is omitted, a default value of 26 is assumed. Maximum value of ui is 26. NODUPLICATE is equivalent to DUPLICATE (0). If no DUPLICATE attribute is present, a value of 26 is assumed for A and B-operators declared outside of ADDMODUL and ∅ otherwise. Duplicates may not be duplicated again.

If the number of temp-registers RHLP needed for the sequentialization of a program is insufficient, the compiler creates a new one if the duplicate entry of RHLP is greater than zero. The new register will be named RHLP_xxx, where xxx is a number of three digits.

If an A or B-operator is used more than once in an esb, the compiler tries to duplicate the existing A- or B-operator. The new name will be oldname_character. 'Character' will be incremented from A to Z. The underscore character '_' will overlay the first blank character of the old name or the 7th character if 'oldname' has 7 or more characters.

Duplicates have the same functions and field. extend parameters of the output port and the same trace option as the original module. Fields are not copied.

b) MOREPORTS(ui) /NOMOREPORTS MOREPORT means: this module can have ui more ports than declared. If ui is omitted, a default value of 26 is assumed. Maximum value of ui is 26. NOMOREPORTS is equivalent to MOREPORTS (O). If no MOREPORT attribute is present, a default value of O is used for modules declared in an ADDMODUL declaration and 26 otherwise. ui is decremented for each created port. If the present number of ports of a storage S or stack K is insufficient, the compiler generates a new one if the moreport entry is greater than zero. The first character of the alphabetically last portname will be incremented by one in order to generate a new name. c) MOREFCT / NOMOREFCT

If functions for a port have been declared in ADDMODUL, the list is assumed to be complete. Addition of more functions by appearance in the program part is possible, if the user uses the MOREFCT option. If functions have not been declared, the addition of functions is allowed. If the user wants to stop the addition of functions for such ports, he may use the NOMOREFCT attribute. The list of functions influences the final statistics in two aspects: 1. The computation of microinstruction bits

$$n_F = \sum_{\text{all ports}} \lceil \text{ld(number of functions of this port)} \rceil \quad ,$$

2. Predeclared duplicates of operators may have different function sets. The allocator uses a specific 'duplicate' only if the specified function is allowed for that port; that means either the function must be present or the addi tion is allowed.

d) AUTO/NOAUTO

The automatic execution of functions may be switched on and off with AUTO and NOAUTO. .LOAD for register and storage destinations and .PUSH for stack destinations are executed by default. This option only influences counts of functions.

e) AUTOFCT (function)

This attribute defines the function which should be executed by default when that particular port is referenced. Unimplemented at this time.

f) INPUTMAX (ui)

This attribute limits the number of inputs to a multiplexer (field). Preceding attributes .DAT, .ADR, .FCT, .CON, .MPX and bitattributes may be used to select the field. If there are preceding bitnumbers/names, the limitation will apply to these bits, otherwise they will be default limits for all newly created fields. If no INPUTMAX is present, a number of 4096 will be used.

```
ADDMODUL
        B.FCT.BIT(3:0).      NOMOREFIELDS.INPUTMAX(1);
```

Example 5.1.2

g) TIME (ui)

This attribute defines maximum delay times for fields. If no field is specified, it is valid for all fields of an input or output; otherwise it is valid for one field. Default value is 1.

Field dependent run-time estimation is not yet implemented.

5.1.2 Deletion

Modules may be deleted by $\{\$\}_0^1$ DELMODUL <modulename> . Connections from ports of this module are not automatically deleted. Instead, their source modulename will print as !DELETED.

5.2 Standard Modules

5.2.1 Random Access Memory

S "name" (<operand>)

In software this means the value of a variable or the contents of a memory cell with the effective address <operand> . The word length is determined by the data type. This cell can be read as a source or altered as a destination.

In <u>hardware</u> S represents a data output or input port of a word-oriented RAM. The operand is connected to the address lines of the RAM.

Parts of memory words can be addressed by attributes:

S "name" (<operand>) <attribute>

The allocator tries to find a suitable port if "name" does not contain a portname.

5.2.2 Register

R "name"

defines a register. Every register is a module of its own with **data** input and output. In contradiction to memories S, it ι.>.**stores** only one word. The modules must be identified by names. ~i. . **Names can** be declared in the declaration part.

Registers are unnecessary from the viewpoint of algorithms. Therefore they should be avoided on level 0 of MIMOLA. Exceptions are registers with special functions, e.g. the program counter RP, I/O-registers.

Some registers can perform functions:

R "name" (<function>) The functions are executed synchronously to the <esb> clock (see chapter 7.1.3). Only the function .LOAD is a standard function, if R is a destination.

Functions can be coded by the value of an operand:

R "name" (<operand>)

The code must be declared.

Functions can be made depending on operands:

R "name" (<function> (<operand>))

By this e.g. the number of shifts can be made variable.

5.2.3 Stack

K "name"

defines a stack. An algorithm can use more than one stack. Therefore every stack must be named. This can be done as in the case of registers. The standard depth of a stack is infinite. A finite depth can be declares. The expression K "name" addresses the top of the stack while reading and the next free cell when writing. By

K "name" (<operand>) all data in the stack can be addressed.<Operand>= 0 addresses the top of the stack while reading or writing. Positive values point down the stack. Writing into the stack should be avoided. Using an <operand> , no stack function is executed.

The only standard function .PUSH is executed automatically, if K is a destination. All other functions can be declared in a declaration part or be appearance in the program, as long as no other declaration has appeared. Possible functions are:

.PUSH, .POP, .NOPUSH, .NOPOP, .CLEAR, .POINT

Only the function .POINT must be explained. Sometimes the stack

pointer must be examined to estimate the load of the stack. K "name" (.POINT) addresses the stackpointer. By this the value of the address of the top of the stack can be read or changed. The bottom of the stack has the address (b.

5.2.4 Instruction

I is the current instruction word and is an abbreviation of S (RP). There is resemblance to the "instruction register" in conventional structures, but I is no register and therefore no "instruction fetch phase" is needed. Such a phase can be programmed, if conventional cumputer structures shall be exactly simulated. By

I. <attribute>

every part of I can be addressed. These parts may be functions, addresses, constants etc. Thus the microprogram can be inserted

As long as the precise partitioning of the microprogram word is not fixed, the implementation by the instruction word can be expressed by:

I (<identifier>) and

I (<function identifier>)

5.2.5 Hardwired Constant

F "name"

is a single hardwired constant. The storage of several constants in a ROM can be expressed by

F "name" (<operand>) The operand addresses the ROM. Single constants and the contents of the ROM must be declared. Hardwired constants are one possibility to implement constants (see chapter 7.5.2).

5.2.6 Monadic Operator

A "name" (<function>)

denotes an operator with one data input and one data output. It performs a monadic operation on one operand standing left of it.

Standard operations for all applicable data types are, output type equal to input type:

- .NOT .ABS .INCR .DECR

boolean output:

.SIGN <Ø <= Ø <>Ø >= Ø >Ø

By

A "name" (<operand>)

the function code can be replaced by an operand, if this code has been defined. This is a possibility to control operations by-passing the control module.If it is used at all, great care must be applied. Attributes can be applied.

5.2.7 Dyadic Operator

B "name" (<function>)

denotes an operator with two data inputs a and b and one output. It performs dyadic operations on the two operands standing left of it. a and b are standard names and can be used to express functions. The assignment to the ports of the operator is explained in chapter 5.4.1 .

Standard operations for all applicable data types are,
output type equal to input types:

```
       +    - *      /  -a+b     .AND   .OR  .XOR   .NAND   .NOR
```

Boolean output:

```
         <_     >_<      >>_       <>
```

Other features are the same as in chapter 5.2.6.

5.2.8 Triadic Operator

C "name" (<function>)

C has 3 input ports named a, b and c. The assignment to the operands is an extrapolation from that in chapter 5.2.7.

No standard functions are declared. Other features are the same as those in chapter 5.2.6.

5.2.9 Network

N "name" (<function>)

denotes a network with any number of inputs, named a, b, ... A, B ... from the left to right. Operands are assigned in the same order.

Functions are expressions of used input names and operators

```
                 (AND )
          +      (OR )
          -      (NOT ).
```

No brackets are allowed.

It is assumed that every control unit of a computer has an instruction pointer unit (8). This unit performs the switching in the case of conditional instructions and is therefore equivalent to the operator td. The range of functions can be altered by declarations.

5.3 Declaration of Connections

5.3.1 Addition

Connections are added by the $ ADDCONNECTION command, followed by a list of connections The general form is

$\{\$\}_0^1$ ADDCONNECTION <connection>$\{$, <connnection> $\}_0^*$;

<connection>::= <destination> <- <source> $\{$/ <source> $\}_0^*$;

The meaning is: all sources are connected to the destination; the left most source gets the highest multiplexer address while the right most source gets the lowest one.
The lowest multiplexer address is zero if no sources are connected by appearance in the program. The character * may by used to define concatenated sources. The '<' character of the left-arrow must be separated from a preceding module name with atleast one blank. Otherwise it will be treated as a port direction character. V's used in ADDCONNECTION declare a name for a bundle of wires which can always be used in the PROGRAM part.

```
ADDCONNECTION
   Va <-  Ra/Rb/Stmp> B ,
   BALU < a    <-  Va,
   Vb    <-    SM > B/Ra/Rint,
   BALU < b  <-    Vb,
   COVF < a    <-   Va. BIT(31),COVF<b <- Vb.BIT(31),
   COVF < c    <-   BALU.BIT(31),COVF>.FCT <- I.FCOVF;
```

<u>Example 5.3.1</u>

5.3.2 Deletion

Deletion of connections is started by the $\{\$\}_0^1$ DELCONNECTION command. General syntax is the same as in 5.3.1. Deleted connections remain in the hardware data structure; they are marked as non-usable and are not counted as multiplexer inputs.

Example 5.3.2 : DELCONNECTION Va <- Rb ,

5.4 Data Paths Conventions

The standard declaration for data paths is: All inputs and outputs of all modules are connected via multiplexer. The number of paths can be limited by declarations (chapter 5.1.1) .

5.4.1 Statements

In the case of

<statement> _ <operand> | <source>

no data are transferred outside the source module. Generally statements specify data paths.

<destination>::= <operand>

specifies a connection between the data output of <operand> and the data input of <destination> , which must be a memory. The <operand> itself can include the connection of several

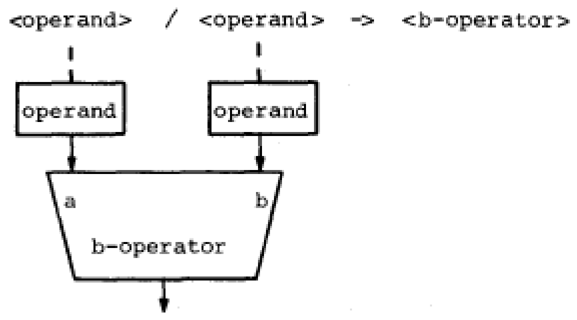


Fig. 5.1

The assignment of the input and output ports is always given by the position of the operands. The postfix-notation gives a simple unique picture of all data paths. The switches for different data paths are assumed to be multiplexers,assigned to the inputs of the modules. It is possible to declare busstructures.

5.4.2 Destinations

All destinations are assumed to be edge triggered. This means: the contents of a memory cell can be read and changed in the same <esb>

5.4.3 Bit-to-Bit Assignment

Normally, when equal data types are coupled, bits with equal bit-position are connected. The rightmost bit is always the least significant bit. Its bit-position number is zero.

If data types with unequal word length are coupled, the connection is always right justified. There is no truncation, if the word length has not yet been specified.

Free input lines are set to zero, free output lines remain open ended. Information can be lost. Correct type transformations must be made by operators.

5.4.4 Attributes

All data paths can be split up into single bits, bitgroups or bytes (8 bits) A constant selection can be made by attributes is a direct assignment. The expression may be composed of several parts, separated by commas. See also Appendix A. The meaning of these parts is:

<unsigned integer>   <unsigned integer>

is a range of bits/bytes.

<unsigned integer > is the bit/byte-position number of a single bit/byte. Allowed names are:

BIT, BYTE;, MASK; BYTEMASK+

BIT and MASK address bits.

BYTE and BYTEffASK address bytes.

BIT and BYTE effect the position of the bits/bytes. All selected bits/bytes are packed tight to the right bound of the data path.

MASK and BYTEMASK do not effect the position of the bits/bytes.

The rules of chapter 5.4.3. have to be obeyed.

Two examples show the result of attributes to bit connections:

Example 5.4.1:

```
S(a).BIT(7:5, 2)->A( - )      "Fig. 5.2"
S(a).MASK(7:5, 2)->A( - )     "Fig. 5.3"
```

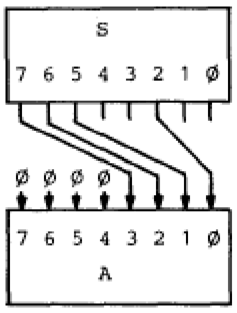


Fig. 5.2

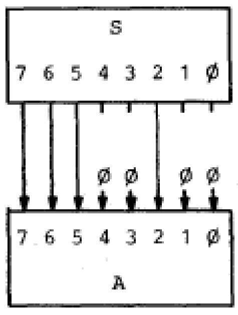


Fig. 5.3

+) not yet implemented

5.4.5 Distributors

V "name" marks a point on a data path. By

<operand> = V "name" this point is defined and can be referenced in the same <esb>. V means no storage of data!

The main application of V is the identification of intermediate results, which are used in different expressions in the same <esb> . In hardware this simply means the parallel con nection of several inputs to one output. V can be named. If V is used in the declaration part it denotes a bus-structure (see chapter 5.3.1).

5.4.6 Concatenation

<operand> * <source>

means the concatenation of all bits of the data output of <operand> with those of <source> . The result forms a new data path, whose width is that of the sum of both elements. The source forms the lower significant part of the word. Example 5.4.2 shows this relation.

Example 5.4.2:

S(a) /R1 -> B(+) * R2 ->A(-)   "Fig. 5.4"

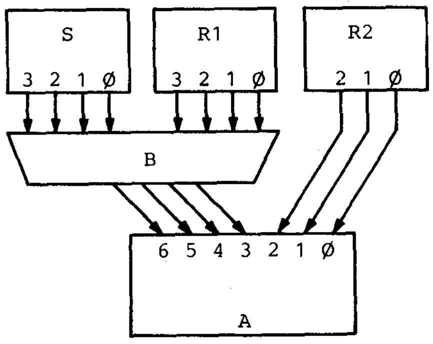


Fig. 5 .4

With V as <source> , even complicated connections can be made.

6. ASSIGNMENTS

The assignment part has little or no relevance to the design of the hardware. It is needed only when micro programs shall be generated. The assignment part is bounded by ASSIGNMENT .... ENDASSIGN The assignment statement a:= b; means: b is assigned to a. a and b may be lists. There are four kinds of assignments:

6.1. Identifier Assignment

<identifierlist> :_ <constant>

<identifierlist> is a list of scalar or **array identifiers separated** by ' , ' . All identifiers in the list get the value

<constant>
.

6.2. Storage Map Assignment

<module> ._ <identifierlist>

This is an implicit assignment of the identifiers to the next free storage cell of <module> , which is assumed to be a stack K or a storage S. <module> may contain a start address or a constant range. The right most identifier gets the lowest value.

6.3 Initial Value Assignment

<modules> :_ <constant> This assignment has influence only for simulations and will normally be regarded as a comment. It means: all or the specified part of the modules cells are assumed to hold an initial value of <constant>.

6.4 Record Assignment

<attribute> :_ <attribute list>

The attributes of the attribute list do not overlap each other and as a whole are equal to <attribute>.

6.5 Example

```
ASSIGNMENT
a,b,c,d: = Ø;  "identifier assignment"
j,k,l  : = 1;
S(1ØØ:Ø):= top, last, ar [5Ø:Ø] ;  "storage map"
"equiv. to: ar:= Ø; last:=51; top:= 52;"
R1,S(5Ø:Ø):= Ø;     "initial value"
.A:= .B.C.D   ; "record assignment"
.B:= .E.F.    ;
.INT :=.SIGN.VALUE;
ENDASSIGN
```

7. PROGRAM

7.1 Fundamental Semantics

MIMOLA is to be understood as a programming language and as a language to describe hardware or synchronous automata functions on a gate control level. Therefore we must notice the software and the hardware meaning of language details.

The fundamental nonterminal symbol of the syntax is the <elementary statement block> , short <esb> . It is composed by all those <statement> 's, which are executed in parallel. The execution of statements can depend on conditions.

The hardware meaning is: one <esb> describes completely one state transition of all synchronous automata that are large and powerful enough to accept all **<esb> 's of the programs.** To reduce the length of the programs there exist some fundamental semantics:

7.1.1 All storage cells that contribute to the state of the automaton are not changed except for those explicitly mentioned in the <esb> and except for the program counter RP.

7.1.2 Unless otherwise determined the program is assumed to be stored in a memory (see 5.2.4) with the program counter RP as a pointer. RP is assumed to be set to the label of the next <esb> in the program, when no other assignments are made.
RP is affected by: GOTO, CALL, RETURN, DO, OD.

7.1.3 All statements of one <esb> are executed within one clock period. Reading of, switching of and operations on data are assumed to be network functions needing no clock. The clock changes the information of the destinations,executes the functions of registers and stacks and terminates the action of the statements by setting RP. The clock is one edge of a clock puls and is synchronous for all statements of one <esb> The clock is not periodic. Its interval depends on the slowest statement in every <esb>

7.1.4 All resources and data paths that occur in one <esb> must be available in parallel. The syntax makes no limitations as to the number of resources or paths. The number can be limited by declarations as a part of the definition of a special automaton. The set of possible resources is defined by the standard declarations in chapter 5.2.

7.1.5 Unless otherwise determined it is assumed that there exists a control module. It must generate the clock, decode the current program word to control the resources and data paths and it must take into account the conditions. There exist no sequential steps within any <esb> controlled by hard-wired or firmware microprograms. The level of MIMOLA is therefore the microprogramming level.

7.2 Labels

L "name" $\{$. <unsigned integer> $\}_{0}^{*}$

Every <esb> begins with a label. It can be used as a line number, as an <esb> address or as a program counter value. The direct assignment between labels and memory addresses of microinstructions is made by the MSS on the lowest level of MIMOLA or by the program loader.

Seven characters are significant for the label name.

During the design process the MIMOLA programs are often compiled from one level n to the next lower level n+1. Every compilation divides several < esb>'s into smaller ones. The label number space is extended by joining unsigned integers at every change of the level. Thus the level can be evaluated from the label structure.

Example 7.2.1 :

Level: Ø 1 2

$$\begin{tikzcd}
La \arrow[r] \arrow[dr] \arrow[ddr] & La.1 \arrow[r] \arrow[dr] & La.1.1 \\
Lb \arrow[ddr] \arrow[dddr] & La.2 \arrow[dr] & La.1.2 \\
Lc & La.3 \arrow[dr] \arrow[ddr] & La.2.1 \\
\dots & Lb.1 & La.3.1 \\
 & Lb.2 & La.3.2
\end{tikzcd}$$

7.3 Functions

The standard functions have been described in chapter 5.2. Special functions have to be declared. It is assumed that the compiler knows the code to control the modules.

Normally the function code is part of the instruction word I. This can be expressed by

    <function>   = I (<function identifier >   )

In hardware it is possible to make a function depend on conditions. This can be expressed by

IF <operand> THEN <function> ELSE <function> *FI*

Multiple applications of the same function to registers and stacks can be expressed by

```
        <function>     _ <function>      ( <operand> )
```

It has to be regarded that the multiple function must be executed synchronously during one clock period. Chapter 7.1.3 has to be obeyed. The result of <operand> must be of type < unsigned integer >

Example 7.3.1:

```
       S(a)= V -> A(IF V -> A(<O) THEN - ELSE +
       FI),
       K1(.POP(3)),
       R1(.SHIFTL(.S(a)))
```

7.4 Identifiers

Identifiers are representatives of constants. After compiling there should exist a list of the values of all identifiers. There are two kinds of assignments:

7.4.1 Identifiers can be assigned a value in the assignment part. These identifiers can be operands. Their values are known
to the programmer.

7.4.2 Identifiers can be assigned a value by their appearance in a program. Identifiers forming the addresses of memory cells are typical examples. In this case the value of the identifier is the address, not the contents of the memory cell.

The only standard data structure is the array. Elements of arrays can be addressed by

```
     <array identifier>      _  <identifier>[ <operand>
                                          [,<operand>)0*]
```

There exists no fixed scheme for the evaluation of effective addresses of array elements. If array indices shall be compiled, array dimensions must be fixed in the assignment part and an evaluation algorithm must be declared in the macro definition part.

7.5 Operands

7.5.1 General Features

Every point on a data path having unique sources can be an operand.

A switching between operands depending on a condition can be expressed by

```
( IF  <operand>  THEN  <operand>  ELSE  <operand>  FI )
```

This construct is equivalent to a multiplexer in hardware.

The root of an operand is the source. Most of the possible sources have been treated in chapters 5.2 and 7.4 .

7.5.2 Constants

Standard constants are all sorts of decimal, hexadecimal and binary numbers. Their use in a program implies no hardware solution for their real source:

hardwired: F	chapter 5.2
part of instruction word I	chapter 5.2
stored in memory S	chapter 5.2

Part of the instruction word is default, another choice must be declared by macros. If this choice is clear from the beginning, no constant should be used.

7.5.3 FOR Loop Control Variable

Independent of a hardware or software solution of the FOR loop, the control variable value can be accessed by D .
In the case of nested loops, the control variable of outer loops is addressed by

```
D ( <unsigned integer> )
```

The number controls the distance to the outer loop. The identity D(Ø) = D specifies the begin of counting.

Another possibility to access the control variable is

D (<identifier>)

The identifier must be the control variable of the current or of an outer loop.

7.5.4 Dummy Source

X is a dummy source. It is needed, if an operand has no influence but must be present to fit the syntax.

Example 7.5.1

```
S(a) / X -> B(-a)   "dyadic operator executing a monadic
                     operation"
IF a THEN X ELSE ... FI "dummy statement"
a / b -> BV(X).EQUAL "operator without function input"
```

7.6 Allocator Conventions

7.6.1 Default Replacement of Identifiers, Functions and Constants

Any constant, identifier or function, which is not removed by a macro, is substituted by a reference to a field of the microinstruction I. The symbolic name of this field depends on the destination. The first letter reflects the type of the input: A for address inputs, C for control inputs, D for data inputs and F for on the left. If the field addressed by this string is already set to another symbolic value, a '@' is added on the right, and the following character will be incremented from 'A' to 'z' until a free field has been found. The resulting string is truncated to an implementation-dependent number of significant characters (e.g. 8,12 or 16). Therefore long names should be avoided, especially for modules which may be duplicated.

For example, the operand SM > B (Ø) will create a connection

SM>B. ADR <- I.ASMB and assign Ø to ASMB.

If the NOMOREFIELDS attribute has been written for I and the resulting fieldname is not already a fieldname of I, the allocator will inhibit the new field if the MSS runs in the batch mode. If the MSS runs in terminal mode, it will ask the operator if he allows the new field. The user may allow the field by typing 'Y', inhibit the field by typing 'N', or he may rename it by typing a name not starting with 'N' or 'Y'. Asking will stop for a particular destination after the user typed 'N' for that field. At least one field for a destination is always allowed.

The symbolic values of the fields created in this way may be put out in each <esb> if the $\{\$\}_0^1$ CODE command is used. In this case the COD-file contains a symbolic microprogram.

7.6.2 Duplicates of Operators

Refer to 5.1.1 a) and 5.1.1 c)

7.6.3 Port Allocation

The omission of port names for stacks and storages indicates that the allocator shall select suitable ports. The allocator then will first try to find a port where the address connection already exists and is usable. If no such port exists, the user may select between taking the first free port by giving the $FIRSTPORT command and optimizing data connections (default). In the second case the allocator tries to find a port where the required data connection already exists and can be used. If there is more than one port with the required data connection, the one with the most frequently used connection is taken. In case of equal frequencies, the alphabetically first port is taken. This optimization is essential for the design process because it tries to create few frequently used connections instead of scattering uses over the set of all possible connections. The designer's part of the optimization process is to delete some unfrequently used connections (whose usefulness could not be foreseen by scanning only once over the entire program) with the $DELCONNECTION command. A second pass over the program, started by $ RESET, will avoid these connections, when this feature has been implemented.

The allocator will create new ports if the current number is not sufficient and if creation is allowed. Creation is allowed either if the MOREPORT entry is greater than zero or if no stack or storage has been declared. If no new port is allowed and the current number is not sufficient, the compiler will split the <esb> .

7.6.4 Default Bitnumbers

If no bitnumbers/names are present in a description of a connection, the allocator inserts default ones according to the following rules:

1. If I is the source, chapter 7.6.1 applied.
2. If there is no field at the source or destination, the attribute .WORD is used.
3. If there is a field present, the field's bitnumbers will be converted into attributes.
4. If there are several fields present, bitnumbers/names, <u>which are not overlapping</u> (see chapter 6.4) are concatenated in the following sequence: .WORD fixea bitnumbers .INT .REAL .BOOL user-defined bitnames in the order of their appearance in the program <u>Example 7.6.1</u>

```
Assume:
 S<A.DAT       contains no field,
 S<A.ADR       contains fields      BYTE () , WORD and USER-ATRB,
 BALD > .DAT contains the non-overlapping fields OVFL and WORD.
```

Then

```
        S<A (... ->BALU (x)):= Ø
will expand as
        S<A.DAT.WORD <- I.DSA
        S<A.ADR.WORD <- BALU> . OVFL * BALU> .WORD
```

There is no truncation for bitnames, only for bitnumbers!

7.7 High Level Language Elements

Many algorithms use repetitions, conditional branches, condition dependent execution of simple or compound actions, functions and subroutines (procedures) to get economic and structured programs. Therefore special language elements have been incorporated in the HLL's and for the same reason in MIMOLA. The MIMOLA elements are: GOTO, FOR FROM BY TO WHILE DO OD, WHILE DO OD, IF THEN ELSE FI, CASE OF THEN FI ELSE ESAC, CALL, SUB, RETURN.

These elements differ in many senses from the elements in the previous chapters. The semantics are a little bit different in different **languages** and should not be fixed in MIMOLA to avoid a restriction of the design space. There exists no unique hardware equivalent of these elements. Thus a direct entry into the hardware data structure is impossible. The designer has to define algorithms that may replace the HLL-elements. These algorithms may use special hardware structures, designed to execute the actions of these elements, or may use the same hardware as the rest of the program. In both cases the HLL-elements can be interpreted as macros. Therefore we will treat these elements in chapter 8.5 together with suggested macro replacements.

8. MACROS

8.1 Use of Macros

8.1.1 Standard Macros

High-level language elements, e.g. FOR FROM ..., do not generate a unique hardware replacement. These "standard macros" have to be replaced by other language elements. This can be done by the macro facility of MSS. The declaration of these macros has to be done by the programmer or the designer. The designer can experiment with different declarations and gains a great design space. Chapter 8.5 explains the standard macros in more details.

8.1.2 User-Defined Macros

The letter "M" is reserved for user-defined macros.

$$\text{<operand> ::= M "name"} \left\{ \text{(<operand>} \;\middle|\; \text{<destination>} \;\middle|\; \text{<function>} \left\{ \text{, <operand>} \;\middle|\; \text{<destination>} \;\middle|\; \text{<function>} \right\}_{\emptyset}^{*} \text{)} \right\}_{\emptyset}^{1}$$

The macros are identified by their "name" and may have parameters of different types. Since this macro is reduced to <operand> and <operand> is used in many syntax rules, user-defined macros are very flexible.

User-defined macros can be used as abbreviations for frequent program constructions. Thus the program text can be kept short and clear. The expansion will be defined prior to the application of the macro. ("Application" in this context means the act of replacing a fitting program string by the declared substitute).

Another use is the introduction of new HLL constructs which are not standard. The programmer may have a certain idea of the semantics, but wants to postpone the decision on, the best implementation. Examples are synchronization primitives. Semaphore operations can be introduced by

My (semaphore), Mp (semaphore),

monitor calls by

Monitor (entry, parameters) .

8.1.3 Software and **Hardware Replacements**

During the design process situations may occur where it is desirable or necessary to replace certain program constructs. It may be the replacement of a complex operator module, e.g. a floating point adder, by a sequentialalgorithm using only simple operators. Another example is the introduction of a special hardware module instead of a software structure that is used frequently in order to increase the execution speed.

To meet these requirements, very flexible macro facilities have been implemented in MSS. Independent of standard or user defined macros nearly all possible program structures can be replaced by others.

This powerful tool must be handled very carefully. The

8.2 Macro Declarations

8.2.1 Syntax

A macro declaration has the normal form

```
 $ MACRO
           expression-1  && expression-2   &&{expression-3 }^1_0
   ENDMACRO
```

The control key MACRO may be replaced by RECMACRO (see chapter 8.3).

The result of this declaration is: expression-1 will be replaced by expression-2 as often as it is found in the original program. Expression-3 will be inserted ahead of the elementary statement block, where a replacement takes place.

expression-1 must be reducable to a nonterminal symbol except 100,101,102,105,106 (see Appendix F) by a syntax rule of the program part (see Appendix B).

expression-2 must be syntactically correct together with its context in its new place.

expression-3 must be one or more esb's. The labels of these esb's must not have more than two characters (excl."L").

expression-2 and expression-3 may use parameters that have to be introduced in expression-1. The names of these parameters are valid only inside the defining macro.

8.2.2 Parameters

The range of application of a macro can be increased by parameters. In MIMOLA three different parameter types can be used in macros: macroparameters, the "?" and special labels.

8.2.2.1 Macroparameters

&"name"' is a free parameter which can be used in a macro declaration. During the syntax check of the macro declaration,

<operand> , <storage> , <register> , or <stack> are tested instead of the parameter, because &"name" is no syntactical symbol. The correspondence to one of the four nonterminals is not remembered. Thus a parameter fits all terminals and nonterminal& during the application of the macro.

If none of the four possibilities can be used to meet the syntax in the declaration, a macro parameter can be bound to a definite terminal or nonterminal, represented by its number in in Appendix F, by

&"name". <unsigned integer> This assignment will also be used during the application of a macro to a program string.

If a parameter is used several times in expression-1, the assignments must be equal. In expression-2 and expression-3, different assignments can be used. During the application the assignment.corresponding to the place of the parameter in the declaration.is valid.

```
MACRO
&arrayname.129[&index]
      &&        &arrayname/&index ->B(+)
ENDMACRO
PROGRAM
.....
   S(aa [ 3 ] ):= S(bb [ S(b)  /R1 ->B(+) ] ),
......
```

After the macro replacement process this will be:

```
......
   S(aa/3->B(+)):= S(bb/S(b)/R1->B(+)),
......
```

Example 8.1

    &."name"

is a special parameter for functions.

A further degree of freedom can be added to parameters by attaching

_"name" .

"name" will be attached without blanks to the string that replaces the parameter. Examples 8.4 and 8.5 in chapter 8.5.2 show an application.

8.2.2.2 Numbering of Names

"names" of elements, used in expression-2 or expression-3 of a macro declaration can be automatically varied to make a distinction between consecutive applications of the element. A "?" following a "name" (no portname!) will be replaced by a number (5 digits), built from a number (3 digits) representing the macro and a number (2 digits) that is a count of the uses of this macro. The numbers are concatenated and reversed. E.g. in the 23. application of macro no. 14,

Vn? will expand to Vn3241Ø .

Thus the digits changing most often are placed directly behind the "name". This increases the significance of the generated names in the case of a truncation (after 8 characters at the moment) of "name"s on the right side.

8.2.2.3 Special Labels

Labels as parameters cannot always be handled with macro parameters. Therefore some necessary special constructions have been included:

L&Ø is the label of the current esb ,

L&DO is the label of the esb that contains the corresponding DO of the current nesting level,

L&OD1 is the label of the esb following the esb that contains the corresponding OD of the current nesting level.

L&DO and L&OD1 can also be used in program parts and will be replaced by the fitting labels. Examples 8.5 and 8.6 show typical applications.

8.3 Controlled Application of Macros

8.3.1 RECMAC

Macros are applied during the syntax analysis process. This process sequentially scans the program string. If a macro becomes applicable to an already scanned string by a macro replacement, this cannot be detected in the same pass. By the key "RECMAC" in the declaration those macros can be marked, which shall be used in another pass. The key "ONLYRMAC" will inhibit all macros declared with "MACRO". See e.g. example 8.8 .

8.3.2 Blocks

The range of macros can be controlled by nested blocking. Block delimiters are

ʘ BLOCK and ʘ BLOCKEND .

One global block is default without delimiters.

8.3.3 Order of Macros

The application of macros is influenced by the order of their declaration.

1. During the declaration of a macro Mi all prior declared matching macros, not containing esb's, are inserted into Mi.

2. The macros are tested in the reverse order of their declaration. This is important if two different macros fit in the same esb before the application of the first fitting macro but not after it.

In both cases the block ranges are observed.

8.4 Application Rules

8.4.1 The Test

A macro fits a string in a program, if the syntax tree of expression-1 of the macro declaration is equivalent to the syntax tree of the string. This equivalence is tested every time a rule is applied by the syntax analyzer. The equivalence of macro parameters has been explained in must be identical. Different attributes for equal fields are not equivalent. If a port is declared in a macro, only this port fits.

Local distributors V "name" (see chapter 5.4.5) will be replaced by their definition prior to the test.

8.4.2 The Application

In the case of a positive test, the program string is replaced by expression-2. The new esb's in expression-3 are inserted on top of the currently analyzed esb. No further test is made in the expanded parts in this pass.

Labels of inserted esb's are automatically numbered as in the case of a "?" (see chapter 8.2.2..2) to avoid double defined label errors. The labels of the current and the first inserted esb are exchanged. Thus the correct pointers to labels are reconstructed.

8.5 Standard Macros .

The HLL elements of MIMOLA can be interpreted as macros.. The replacement is not unique and depends on the exact semantics of the element that is used in an algorithm and on the existing or proposed hardware. The given example declarations are only possible solutions. They have been included to describe the HLL elements, to give ideas for macro declarations and to serve as examples for correct macros.

8.5.1 GOTO

This unconditional jump can be replaced by

```
$ MACRO
  GOTO &lb.115    && RP:= &lb.115
  ENDMACRO
```

Example 8.3

8.5.2 FOR Loop

A FOR loop consists of three parts:
setup, condition test and control variable updating and return.

8.5.2.1 Setup

A complete setup statement is

$$\text{FOR a } \{\text{FROM b}\}_0^1 \ \{\text{BY c}\}_0^1 \ \{\text{TO d}\}_0^1$$

a is the control variable. It can be referenced by D(a), not by itself, to show its storage requirements.
Missing b or c are assumed to be 1.
Missing d implies the ommission of the upper/lower limit test.

A macro declaration using cells in an addressable memory Sm is presented in Example 8.4. Many other possibilities exist e.g. using stacks to store the parameters. For incomplete setups, slightly varied macros must be written.

```
$ MACRO
  FOR &id.129    FROM &opf  BY &opb  TO &opt
 &&   Sm (&id.129_Ø):= &opf,
      Sm (&id.129_1):= &opb,
      Sm (&id.129_2):= &opt
ENDMACRO
```

Example 8.4

8.5.2.2 Condition Test

The FOR loop condition test must follow the setup in another statement. The syntax is

<label> $\left\{\text{WHILE}\quad e\right\}_{0}^{1}$ DO(a)

A macro declaration fitting to example 8.4 is example 8.5.

```
$ MACRO
   &lb.115  WHILE & opw DO (&id.129) &statementblock.1Ø7;
&&    &lb.115 IF&opw /
                    Sm (&id.129_Ø)/
                    Sm (&id.129_1)/
                    Sm (&id.129_2)->C(.LOOP)->B(.AND)
                    THEN & statementblock.107
                    ELSE GOTO L&OD1 FI;
ENDMACRO
```

Example 8.5

If different forms of setup statements exist in a program, it may be necessary to. distinguish between different forms of condition tests. In this case user defined macros for DO and OD must be used.

8.5.2.3 Updating and Return

At the end of a FOR loop the control variable must be updated and a jump to the head of the loop occurs. The syntax of

this statement is

OD(a)

In correspondence to examples 8.4 and 8.5, a macro declaration is:

```
$ MACRO
   OD(&id.129)
&& Sm(&id.129_Ø):= Sm (&id.129_Ø)
         /Sm(&id.129_1) ->B(+),
   GOTO  L&DO
ENDMACRO
```

Example 8.6

8.5.3 WHILE loop

The syntax of the WHILE loop head and tail is:

```
   <label>  WHILE    <operand>   DO

    OD
```

A macro declaration is given in example 8.7.

```
Ø MACRO
   &lb.115 WHILE &opa DO &statementblock.1Ø7;
&& &lb.115 IF &opa  THEN &statementblock.1Ø7
                       ELSE  GOTO L&OD1
           FI;
ENDMACRO
Ø MACRO
   OD
&& GOTO L&DO
ENDMACRO
```

<u>Example 8.7</u>

8.5.4 Conditions

8.5.4.1 Conditional Statement Block

The execution of statement blocks can be condition dependend. The syntax is:

```
IF  <operand>   THEN  <statementblock>
                                        1
               {ELSE  <statementblock> }
                                        Ø
FI
```

The semantics is the same as in other HLL's. The hardware realization of conditional executions is normally a part of the control module of a computer. Therefore no generally applicable macro can be given. In our example 8.8 an IF THEN ELSE construction is split into two esb's, one executing the THEN part, the other the ELSE part. The second macro seems to be senseless, but if we observe chapter 8.3.3 we can see that in this way the application of the first macro to already split IF THEN ELSE constructions is prevented.

```
"THE FIRST MACRO WILL NOT BE INSERTED INTO THE SECOND,BECAUSE IT WILL
 INSERT A <ESB>.
 THE SECOND MACRO WILL BE CHECKED  AND REPLACED FIRST.
 THEREFORE THE FIRST MACRO CANNOT BE INSERTED SENSELESS."
$RECMACRO  IF &opb.113  THEN   &sbthen.107   ELSE  &sbelse.107   FI
 &&     IF &opb.113  THEN   &sbthen.107       FI
 &&    Lii     IF &opb.113 THEN  X  ELSE  &sbelse.107   FI ;
ENDMACRO
$RECMACRO  IF &opb.113  THEN   X    ELSE  &sbelse.107   FI
 &&        IF &opb.113 THEN  X  ELSE  &sbelse.107 FI
ENDMACRO
```

Example 8.8

8.5.4.2 Conditional Operands

If the THEN and ELSE; parts of a conditional statement block differ only in one of the operands, this can be directly shown by applying the condition to the operand itself. The syntax is

```
operand  ::= (IF  <operand>  THEN  <operand>
                             ELSE  <operand>  FI)
```

Example 8.9 shows an application. The semantics and the hardware are the same for both statements.

```
IF S(cond) THEN R1:= S(a)/R2->B(+)
           ELSE R1:= S(b)/R2->B(+)  FI
is equivalent to
R1:= S((IF S(cond) THEN a ELSE b FI ))/R2->B(+)
```

Example 8.9 Conditional Operand

8.5.4.3 Conditional Functions

The same reason as in chapter 8.5.4.2 holds for conditional functions. The syntax is

```
   <function> ::= IF  <operand>  THEN  <function>
                                  ELSE  <function>  FI
```

```
     S(a) ->A(IF S(a) ->A( >= Ø) THEN + ELSE - FI)
```

Example 8.10 Absolute Value of a

8.5.5 CASE Statements

The CASE statement is a convenient tool to express multiple choices. The syntax is

$$
\begin{array}{l}
\text{CASE <operand> OF} \\
\left\{ \text{<identifier> <signed integer>} \right. \\
\quad \left\{ \text{, <identifier> <signed integer>} \right\}^{*}_{\emptyset} \\
\qquad \left. \text{THEN <statement block> FI} \right\}^{*}_{1} \\
\qquad \left\{ \text{ELSE <statement block>} \right\}^{1}_{\emptyset} \\
\text{ESAC}
\end{array}
$$

The macro of example 8.11 is a simple solution of the complex CASE statement. The generated esb's with only one case can be further expanded by another macro.

```
ØRECMACRO    CASE     &op     OF      &cl.17Ø
 &&      CASE     &op     OF
 &&    Lca     CASE    &op     OF     &cl.17Ø    ESAC ;
ENDMACRO
"THIS MACRO REDUCES CASE STATEMENTS TO CASE STATEMENTS WITH
A SINGLE CASE LINE AND IS NOT SUITABLE FOR CASES INCLUDING
ELSE."
```

<u>Example 8.11</u>

8.5.6 Subroutines

The semantics of subroutines and procedures vary strongly between different languages. Therefore only very general language elements have been incorporated in MIMOLA.

Three general actions can be distinguished: the call, the parameter handling and the return to the calling program.

CALL <identifier>

CALL <identifier>(<operand> $,\{$<operand>$\}_{\emptyset}^{*}$)

are statements for subroutine or procedure calls without and with parameters. Possible actions are e.g. save program status, jump to subroutine head. The identifier points to the subroutine code. <operand> is an actual parameter.

SUB

SUB(<operand>$\{$, <operand>$\}_{\emptyset}^{*}$)

are statements that indicate necessary action at the start of the subroutine, e.g. copy the actual parameters. <operand> is a formal parameter.

RETURN

is the usual statement to restore the old program status and jump back to the calling program. If values must be returned, user macros should be defined.

Subroutines are separated from other program parts by the header

SUBROUTINE <identifier>

and the tail

ENDSUB

Example 8.12 is a simple subroutine mechanism. The return address is pushed onto stack Kcall. The parameter values are pushed onto stack Kop in the calling sequence and stored in memory locations by SUB. The parameter lists must be of equal length. The example mainly demonstrates a possibility to handle calls with different numbers of parameters.

```
"WITHOUT PARAMETERS"
$MACRO   CALL &id.129
  &&       RP:= &id.129,
           Kcall(.PUSH):= RP -> A(.INCR)
 ENDMACRO
"WITH PARAMETERS"
"SEQUENCE OF MACROS IS ESSENTIAL, BECAUSE THE LAST MACRO
 CAN BE INSERTED INTO THE FIRST TWO MACROS"
$MACRO   CALL &id.129 ( X )
  &&       RP:= &id.129,
           Kcall(.PUSH):= RP -> A (.INCR)
 ENDMACRO
$MACRO   CALL &id.129 ( X , &op. 113
  &&       CALL &id.129 ( X
  &&    LCA        Kop(.PUSH):= &op;
 ENDMACRO
$MACRO   CALL &id.129 (      &op.113
  &&       CALL &id.129 ( X
  &&    LCA        Kop(.PUSH):= &op ;
 ENDMACRO

"SEQUENCE OF DECLARATION IS ESSENTIAL!"
$MACRO    SUB   ( X )
  &&      X
 ENDMACRO
$MACRO    SUB   ( X , &op.113
  &&      SUB   ( X
  &&      LSU     &op := Kop (.POP) ;
 ENDMACRO
$MACRO    SUB   ( &op.113
  &&      SUB   ( X
  &&      LSU   ( &op:= Kop(.POP) ;
ENDMACRO

$MACRO      RETURN
 &&     RP:=  Kcall (.POP)    ENDMACRO
```

Example 8.12

9. CONTROL LANGUAGE

9.1 General

The control language is used to start different language parts, to cause typeouts and to set options. Commands may be input from any of the three input channels: the INP(ut)-file, the LIB(rary)-file and the terminal. The first command is read from the INP-file. Changing the input stream permanently is done with the INSERT command. A breakpoint facility is supplied for temporary command input from the terminal. Nonrecursive nesting of control commands is possible because a new control command is expected if there is a $-sign in the parameter string of the last command.

Breakpoints allow interaction by the terminal operator. A breakpoint is inserted by the user with a $BREAK command or by the MSS under the following conditions:

1. a declaration part has been completely analysed,
2. the DISPLAY option is true and an <esb> has been analysed,
3, the SELF option is false and the hardware is not sufficient to execute the current <esb> ,
4. a DEBUG test is true,
5. a WAIT condition is acknowledged with a $-sign,

6. an internal error has been found. Breakpoints (except case 6.) are ignored if the BATCH option is true. At breakpoints the operator may inform himself of most of the MSS data structures, including structures which

are useful during the program analysis (e.g. the analyser stack, the current connection etc.).

When the MSS expects a control command, all special characters preceding a command (except - and @) are ignored. Therefore a $-character may always precede control commands for clarity. Prompt characters, sent to the terminal whenever the computer expects an input, may be sent back.

Strings enclosed in " ...." are treated as comments.

Because multiple successive commands may be input at breakpoints, a special command is required which returns control to the original language part. This is done by the RESUME command. In order to reduce necessary writing, any special character except (blank), - and @ preceding (end-of-line) is treated as a RESUME command. On some installations the prompt character sent to the terminal automatically may be used as the RESUME command.

Nesting of control commands may either result from a construction or a breakpoint. Normally, nesting does not go beyond level 3:

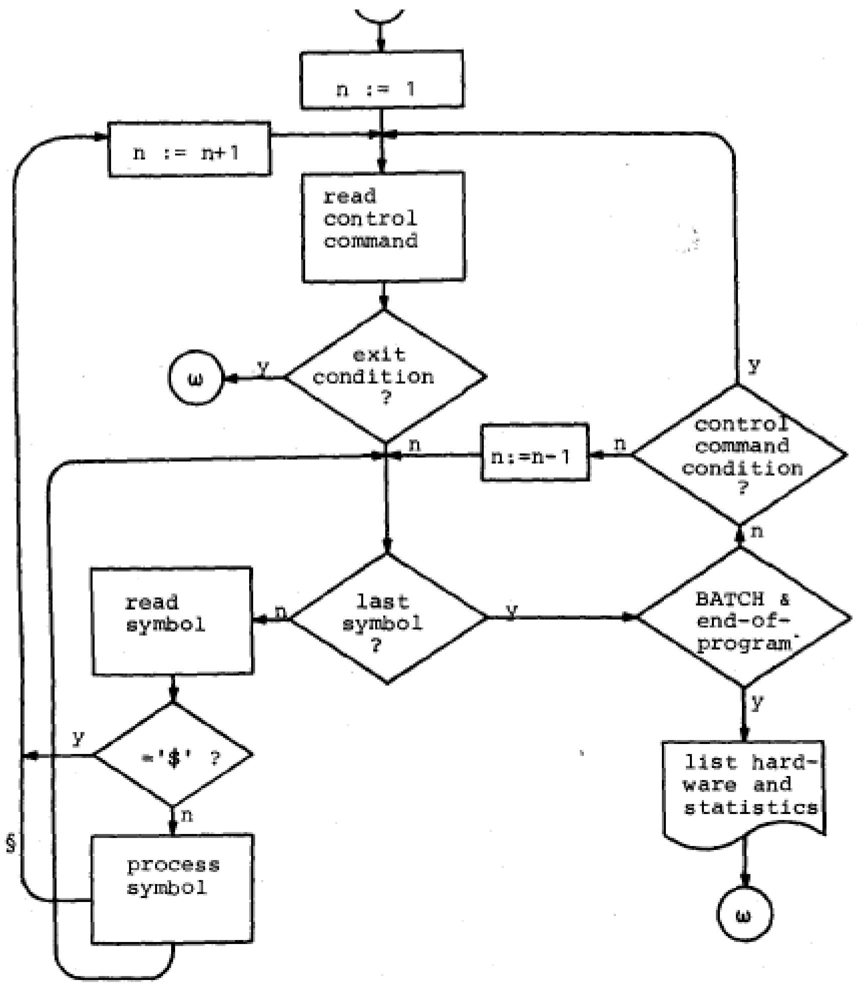
level 1:	normal control flow
level 2:	a) $ -constructions within MIMOLA language parts
	b) breakpoints
level 3:	breakpoint in level 2 case a).

If after a long MSS run an important breakpoint has been reached, the user may increment the current level by typing BREAK if he wants to be safe against a single, erroneously typed RESUME.

Decrementing the nesting level will cause the continuation of analysis at the next lower level. For breakpoints this will also reset the input stream to the old input channel, except if an INSERT command has been written. If the current level is maintained after the execution of a command, a new control command is expected.

RESUME, CORRECT and GO will decrement the current nesting level. Nesting levels >1 are also decremented if the input stream is not the terminal or after an INSERT command. Therefore no level 2 command is expected after e.g. a $ FACTOR command in the INP-file. The same command from the terminal requires a RESUME in order to leave level 2 because of the multiple successive command feature for the terminal. No control command is expected after a level 2 INSERT. A level 1 insert, however, may not be decremented because this would cause an EXIT to the operating system. Fig. 9.1 shows a flow diagram for control commands.

In order to exclude nestings which may cause MSS errors, the set of allowable control commands depends on the context. The user can get a list of allowed commands with the KEY command.



Abbreviations:

§ = breakpoint in symbol processing routine (cf. DEBUG-cmd.)

exit condition = EXIT-cmd. or (RESUME-cmd. and (n=1))

n = nesting level of interpreter

```
control command condition = not (
     RESUME-cmd. or GO-cmd. or CORRECT-cmd. or (
     (n > 1) and (not terminal input or INSERT-cmd. )))
```

```
last symbol = no symbol expected after command (e.g. BREAK)
              or last symbol for this command (e.g. END for
              a PROGRAM command)
```

Fig. 9.1 Control language interpreter flow chart

9.2 Command Description

9.2.1 General Commands

PROGRAM	Start of a program part. Ends with END or ENDSUB.
ASSIGN	Start of an assignment part. Ends with ENDASSIGN.
LIST	Copy source input to OUT-file. Normally only in the case of errors the last two source lines are copied to the .OUT-file If the user wants to have a better overview of the locations of errors or if he wants to document the complete source together with the hds listing, he should use this command.
DISPLAY (default) / NODISPLAY	If the DISPLAY option is on, the MSS shows each completed <esb> at the terminal and ,inserts a breakpoint. The commands can be used to switch the option on and off.
WAIT / NOWAIT (default)	In the wait mode the MSS asks the terminal operator to type in an acknowledgment character after the writing of error messages to the terminal. The MSS expects a control command if a $ -sign is input.
BATCH	The BATCH command includes the NODISPLAY, NOWAIT and SELF commands and additionally will cause breakpoints to be ignored. No messages will be written to the OUTPUT-file (the terminal) if the MSS runs in the BATCH mode. The allocator will not ask for new microprogram fields if the number of fields is restricted and not sufficient. See chapter on allocator conventions.

	At the end of the first PROGRAM part, the MSS inserts a $PRINTHARDWARE command and then returns to the operating system. Although intended for batch tasks, this option may be set for other tasks if no terminal interaction is desired. The INPUT- and OUTPUT-files are still required. The INPUT-file may be empty.
ONLY / WITH (default)	The command ONLY allows a fast syntax check. All semantic actions are disabled (including the additional rules of Appendix C).
BREAK	BREAK sets a breakpoint for terminal interaction. A control key is expected from the terminal (cf. BATCH). After the breakpoint has been left with RESUME, control is returned to the original input stream.

RESUME = (string of special characters except (blank), _ and @ before (end-of-line)):

	leave the current control command interpreter nesting level.
INSERT "name"	If "name" starts with L, I or T the input stream is switched to the library, the INP-file or the terminal, respectively. Otherwise the library is reset to its beginning and scanned for the string !"name". When this string has been found, the input stream is set to the library. The !-construction allows the user to build a library of commonly used macros. Insertion stops with the next INSERT command.
$K\{EY\}_0^1$	Type the list of currently possible commands.
EXIT	Return to operating system.
EX	Fast return to operating system (no run-time statistics etc.)

RESET	Sets all hds frequencies to zero and starts reading from the beginning of the INP-file again. This command is used to obtain a second pass over the entire program after e.g. certain connections have been marked with DELCONNECTION (see below). This allows a better optimization by the allocator.
PRINTSTACK / TYPESTACK	The analyser stack is dumped into the OUT-file or to the terminal. PRINTSTACK is executed automatically when syntax errors are encountered. The dump contains the internal numbers of the base symbols and names, numbers and some special characters. If the basic symbol is a reserved word, this is also written. A right arrow points to the symbol the analyser looks at. The user has to use the symbol table in Appendix F for recoding the basic symbols. All non-terminal symbols have codes >= 100. All terminals, which consist of only one character, use their ASCII number as code.
DEBUG <character>	This command is used for debugging MSS.<character> is added to a test set. The MSS contains tests which will produce a certain action if a certain character is included in the test set. Initially the test set is empty. Some of the actions are:

	character action 1 list intermediate connection descriptors 4 set breakpoints within allocator execution 7 include connections to CONTROL inputs set breakpoints within compiler execution
DEBUGOFF <character>	<character> is subtracted from the test set.
DECLARE, ENDDECLARE	Ignored, provided for compatibility with older MSS versions.

9.2.2 Macro Commands

MACRO	Start of macro definition. Ends with ENDMACRO.
RECMACRO	Same as **MACRO except only RECMAC** macros are inserted if the command
ONLYRMAC	has been given.
BLOCK	Defines the beginning of a range of macro definitions. The following macros are valid only until a corresponding
BLOCKEND	has been found. These commands are intended for the easy substitution of variable declarations of high-level block-oriented languages.

9.2.3 Compiler Commands

NOGENERATE	This option disables the internal storage of a syntax tree and disables all actions which rely on this tree. With the NOGENERATE option no macro can be inserted, <esb>' s cannot be splitted and no GEN-file can be

generated. The allocator and
statisti-
cal analyser, however, are
unaffected.
This option saves core and time.
ESB If a connection error (insufficient
hardware) occured in an <esb> ,
the compiler will insert a break-
point at the end of the esb if
the SELF option (see below) is
false.
The user then may use ESB to get a
listing of the current esb at his
terminal. He may use
CON to show the connection errors.
CORRECT will start the splitting of <esb>'s
containing connection errors,

GO	will cause the <esb> to be ignored.
USING $\{S \mid R\}_1^1$	The compiler needs temporary cells in which it stores intermediate operands. In the default case, the compiler uses registers RHLP for this purpose. After a USING S command the compiler uses cells of a random access memory named SHLP. The source program may already contain SHLP references. Often a mapping of SHLP to commonly used register-files is possible. The compiler indicates an error if the number of ports of SHLP is restricted in a way that splitting is impossible.
MINHLP	If MINHLP is true, temporary cells replace only the reference which caused a connection error. In the default case the compiler replaces all references to an operand by a

	temporary cell if one reference causes a connection error. MINHLP reduces the number of temporary cells in some cases.
NFULBUF	With this option the compiler buffers operands after CASE and IF only if it is necessary. In the default case these operands may be-buffered in order to free hardware for the following statements.
9.2.4 Commands	Referencing the Data Base
ADDMODUL	This command declares modules, ports, fields and functions. See chapter 5.1.1 for full explanation.
DELMODUL	**Deletes** modules. cf. chapter 5.1.2 .
ADDCONNECTION	Create connections . cf. chapter 5.3.1 .
DELCONNECTION	Mark connection as not usable. cf. chapter 5.3.2 .
PRINTHARDWARE,	PH Dump the hds into the OUT-file. Appendix E shows the format of this listing. It contains the following information: 1. frequency of module uses This information is required in order to select all the modules which shall be deleted in the next design step. 2. frequency distribution of the number of ports required in <esb>'s It is counted, how many times parallel execution of an <esb> requires n (O<-n<_9) ports of a storage module. The number of concurrent uses of different ports of the same module determines the number of independent ports in the final design. 3. frequency of use of a parti-cular port This information determines the uti-lization factor of particular ports.

4. frequency of connection usage
Connections are basic cost factors and the frequency of their use is needed in an economic design.

5. joint frequency distribution of concurrent module and port uses
Often a hardware unit may be substituted by another hardware unit, e.g. an incrementer by an arithmetic function box (horizontal migration). The joint distribution indicates whether the substitution of two or more units by a single unit would significantly increase the number of <esb>'s. The same is true of ports of storages and stacks. For example, two ports with different direction, which are rarely used concurrently, may be substituted by a single bidirectional port.

6. Estimation of run-time
It is assumed that addressing of storage ports, propagation through operators, microinstruction reading and the write cycle of storage ports each require one unit of time. Overlapping is considered correctly.

Manual setting of weighting factors for the <esb> is important for this computation. cf. chapter 9.2.6 .

7. joint frequency distribution of concurrent µinstruction field uses (cf. chapter 9.2.6 : JOINT command)
This distribution is required if one wants to reduce µinstruction word length by replacing certain fields by references to other fields.

8. average number m and standard deviation s of used µinstruction fields and bits

$$m = \sum_{\substack{\text{all} \\ \mu\text{instructions}}} \frac{\text{used } \mu\text{instruction bits} * \text{weighting factor}}{\text{weighted number of } \mu \text{ instructions}}$$

$$s^2 = \sum_{\substack{\text{all} \\ \mu\text{instructions}}} \frac{(\text{used } \mu\text{instruction bits})^2 * \text{weighting factor}}{\text{weighted number of } \mu\text{instructions}} - m^2$$

The µinstruction is badly used if m is much less than the wordlength or if s/m > 0.5.

9. the number of enable bits
This number is incremented by one for every port having .LOAD as a function.

10. the number of function select bits

$$fs = \sum_{\substack{\text{all}\\\text{ports}}} \lceil ld^{*}(\text{number of functions in the function list}) \rceil$$

$\lceil n \rceil$ means the smallest integer greater or equal to n.

$$ld^{*}(n) = \begin{cases} ld(n) & \text{for } n>\emptyset \\ \emptyset & \text{for } n=\emptyset \end{cases}$$

11. the number of multiplexer address bits

$$ma = \sum_{\substack{\text{all}\\\text{multi-}\\\text{plexers}}} \lceil ld^{*}(\text{number of multiplexer inputs}) \rceil$$

12. the sum of multiplexer inputs

$$mi = \sum_{\substack{\text{all}\\\text{multi-}\\\text{plexers}}} \text{number of multiplexer inputs}$$

13. the number of connections

$$co = \sum_{\substack{\text{all}\\\text{connections}}} 1$$

Items 9 to 13. are essential for computing the cost of a design.

TYPEHARDWARE, TH	same as PRINTHARDWARE, except output goes to the terminal.
TYPEMODUL, TH modulename	The hds entries for one module, including port, input/output, field and connection entries are output to the terminal.

TRACE <modulename>$\{$, <modulename> $\}_{0}^{*}$;

The allocator will print sources for the specified module ports each <esb> Only one source is printed for port.

If function, address and data input are used, the data input has priority over the other inputs.

NOTRACE <modulename> {, <modulename> }$^{*}_{O}$; (default)

Contrarv of TRACE.

PRINTFUNCTIONS, PF TYPEFUNCTIONS, TF	List used and predeclared functions together with the frequency of their use and the names of ports, which can execute them. Listing goes to the OUT-file for PRINT-FUNCTIONS and PF,and to the terminal for TYPEFUNCTIONS and TF.

9.2.5 Allocator Commands

NOLOWERLETTERS	This option causes the allocator to use upper case letters instead of lower case letters as portnames of B, C and N modules.
FIRSTPORT	This option causes the allocator to use the alphabetically first free port of an S or R module while looking for a suitable port. In the default case, the allocator tries to find a port where the required connection already exists.
CURHSTAC	The allocator lists the currently valid intermediate connection describing structure at the terminal.
CODE	Write a symbolic μinstruction code into the COD-file. The form of the output is: RP= <program counter value> <label name> TIME=<run-time> {<symbolic μinstruction field name> = <symbolic value> , }$^{*}_{O}$

9.2.6 Statistical Analyser Commands
The analyser counts all uses of
hardware
with a factor f = fac * fact * fac3
.
Default value of all factors is
one. The
following commands set the
factors to
other values:
FACTOR <n> Set fac to n.
FACTOR2 <n> Set fact to n.
FACTOR3 <n> Set fac3 to n.
n must be decimal. Changing of
the
factors takes place at the
beginning
of the <esb> following the
control
command.
NOFACTOR Disable all following FACTOR, FACTOR2
and FACTOR3 commands. This
command is
useful if unweighted
informations are
desired.
JOINT Compute relative joint distributions
of module and storage port uses
and
of microinstruction field uses.
Computation starts with the first
completed <esb> after this
command.
This option requires about 12.8
Kbytes
on the heap. Listing is
requested
with the PRINTHARDWARE
command.
9.3 Example
The following example shows how a small program is
analysed and how the LIB-file may be used to contain
common module declarations, assignments and macro definitions.
It is important to see how the input stream is switched
between the various input channels. Note that breakpoints
for terminal interaction are automatically inserted after
the module declaration (because BATCH is false), and after
<esb>'s are analysed (because DISPLAY is true by default).

Example 9.3.1

INP-file	LIB-file	terminal	
		-input	-output
INSERT LIB			
	$ADDMODUL		
			MONITOR LEVEL 1 ADDMODUL
	$MAIN.MOREPORT(10);		
			MONITOR LEVEL 2 EXPECTS KEY!
		TM SMAIN	
			MONITOR LEVEL 2 TM (hds listing of SMAIN) MONITOR LEVEL 2 EXPECTS KEY!
		/	
			MONITOR LEVEL 2 RESUME
	$ASSIGN a,b,c:=Ø;		
			MONITOR LEVEL 1 ASSIGN
	ENDASSIGN $ INSERT INP		
			MONITOR LEVEL 1 INSERT
PROGRAM			
			MONITOR LEVEL 1 PROGRAM

begin
LØ SMAIN(a):=b;			

			LØ.1 SMAIN<A(a):=b; MONITOR LEVEL 2 EXPECTS KEY!
		/SELF	
			MONITOR LEVEL 2 SELF MONITOR LEVEL 2 EXPECTS KEY!
		+	
			MONITOR LEVEL 2 RESUME
$ INSERT GOTO			
			MONITOR LEVEL 2 INSERT
	$ ADDMODUL ... ! CALL ... ! GOTO	searched for ! GOTO from beginning of file	
	$ MACRO		
			MONITOR LEVEL 2 MACRO
	GOTO &lb.115 && RP:= &lb ENDMACRO $ INSERT INP		
			MONITOR LEVEL 2 INSERT
L1 IF SMAIN(a). BIT(1) THEN GOTO LØ FI;			
			L1.1 IF SMAIN>B(a).BIT(1) THEN RP:= LØ FI; MONITOR LEVEL 2 EXPECTS KEY!
		/	
			MONITOR LEVEL 2 RESUME
END			
			<CHARACTER STRING ACCEPTED> MONITOR LEVEL 1 EXPECTS KEY!
		PH	
			MONITOR LEVEL 1 PH MONITOR LEVEL 1 EXPECTS KEY!
		/	
			MONITOR LEVEL 1 RESUME
	(exit to operating system)		

10. EXAMPLES

10.1 Output Listing Examples

The following short program is used to show the format of the output files

```
ADDMODULE
 SA>A,SA<B;
CODE JOINT
PROGRAM
BEGIN
LO SA(1):=SA(0)/SA(2)->B1(+);
L2 IF SA(R1)->A1(.INCR).BIT(3)
    THEN R1:=SA(1), SA(1):=R1, GOTO LO
    ELSE R1:=SA(3), R2:=SA(4) FI,   R3:=F1;
END
```

Fig. 10.1.1

The GEN-file demonstrates how the original esb's have been splitted in order to be executable on the hardware :

```
"MIMOLA VERSION 2.2 OF MAR 29, 1979 RULES 84 & 116 "
"EXECUTION DATE :  05/16/79  16:33:55"
INSERT LIBRARY
PROGRAM
        BEGIN
L0.1
        RHLP_101:=SA>A(0);
L0.2
        SA<B(1):=RHLP_101/SA>A(2)->B1(+);
L2.1
        R3:=F1,
        RHLP_101:=SA>A(R1)->A1(.INCR);
L2.2
        IF RHLP_101.BIT(3)
          THEN
            R1:=SA>A(1),
            SA<B(1):=R1,
            GOTO L0.1
          ELSE
            R1:=SA>A(3) FI
        ;
L2.3
        R2:=SA>A(4);

        END
```

Fig. 10.1.2

One memory port was missing in the original esb's. Register RHLP-101 is used as a substitute. The dump routine appends the names of the used ports to the module names.

The COD-file contains symbolic

```
-------- PC=   1 L0        TIME=    2
ASAA     =0      ,
-------- PC=   2 L0        TIME=    3
ASAA     =2      ,FB1      =+       ,ASAB    =1       ,
-------- PC=   3 L2        TIME=    3
FA1      =.INCR  ,
-------- PC=   4 L2        TIME=    2
ASAA     =1      ,ASAA@A   =3       ,ASAB    =1       ,
-------- PC=   5 L2        TIME=    2
ASAA     =4      ,
```

Fig. 10.1.3

Enable fields are not considered at present. Therefore the first esb only needs an address field for storage port SA>A. This field is named ASAA by default and is set to zero. Estimated run time is two units : one unit until data from SA is valid and one is required as data set up and hold time for RHLP and for the reading of the next microinstruction. The second esb requires three time units because the data from SA>A(2) have to propagate through B1. The write cycle for SA<B is assumed to be one time unit because a constant address is used. The second esb uses three microinstruction fields, two for addresses and one for the function code of B1.

The OUT-file contains the hds listing

```
MIMOLA VERSION 2.2 OF MAR 29, 1979 RULES 84 & 116
EXECUTION DATE :  05/16/79  16:33:55
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
  1 HELP-STORAGES USED AND 1 ESBS CREATED
  1 HELP-STORAGES USED AND 2 ESBS CREATED
***     2 ESBS READ    (        2 WEIGHTED)
***     3 ESBS CREATET (        3 WEIGHTED)
***     5 ESBS TOTAL   (        5 WEIGHTED)
  1 HELP-STORAGECELLS (OR REGISTERS) USED IN PROGRAM
ESTIMATED RUN TIME:           12
*** GROUP : A
A1            (     0:     0) MOREPORT:-2 DUPLICATIBLE :26    FREQ :        1 =    20.00%
PORTUSINGS , OUTPUT : 0 :     4X  1 :     1X
             INPUT  : 0 :     4X  1 :     1X
    <         (     0:     0).AUTO(NONE    ).STATUS(#08000000) FREQ:        1 =    20.00%
       DATA      WORD      ADR  D-BITS    MODULE     PORT    S-BITS  FREQ
                             0 WORD      SA       A        WORD          1 = 20.0%
     >        (     0:     0).AUTO(NONE    ).STATUS(#44020000) FREQ:        1 =    20.00%
       DATA      WORD               FREQ:     1
       FUNCTION WORD       ADR  D-BITS    MODULE     PORT    S-BITS  FREQ
                             0 WORD      I                 FA1           1 = 20.0%
```

```
*** GROUP : B
B1              (     0:     0) MOREPORT:-3 DUPLICATIBLE :26     FREQ :         1 =   20.00$
PORTUSINGS , OUTPUT : 0 :      4X  1 :     1X
             INPUT  : 0 :      4X  2 :     1X
     >          (  -  0:     0).AUTO(NONE     ).STATUS(#44000100) FREQ:         1 =   20.00$
       DATA      WORD                FREQ:     1
       FUNCTION WORD        ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        I                  FB1          1 = 20.0$
    < a         (     0:     0).AUTO(NONE     ).STATUS(#08000000) FREQ:         1 =   20.00$
       DATA      WORD       ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        RHLP_101           WORD         1 = 20.0$
    < b         (     0:     0).AUTO(NONE     ).STATUS(#08000000) FREQ:         1 =   20.00$
       DATA      WORD       ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        SA        A        WORD         1 = 20.0$
----------------------------------------------------------------------
*** GROUP : F
F1              (     0:     0) MOREPORT:-1 DUPLICATIBLE : 0     FREQ :         1 =   20.00$
PORTUSINGS , OUTPUT : 0 :      4X  1 :     1X
             INPUT  : 0 :      5X
     >          (     0:     0).AUTO(NONE     ).STATUS(#44000000) FREQ:         1 =   20.00$
       DATA      WORD                FREQ:     1
----------------------------------------------------------------------
*** GROUP : I
I               (     0:     0) MOREPORT:-1 DUPLICATIBLE : 0     FREQ :         5 =  100.00$
PORTUSINGS , OUTPUT : 1 :      5X
             INPUT  : 0 :      5X
     >          (     0:     0).AUTO(NONE     ).STATUS(#44000000) FREQ:         5 =  100.00$
       DATA      ASAA     4          FREQ:     4
       DATA      ASAA@A              FREQ:     1
       DATA      FB1                 FREQ:     1
       DATA      ASAB                FREQ:     2
       DATA      FA1                 FREQ:     1




       DATA      ASAA@B              FREQ:     0
----------------------------------------------------------------------
*** GROUP : R
RHLP_101        (     0:     0) MOREPORT:-2 DUPLICATIBLE : 0     FREQ :         3 =   60.00$
PORTUSINGS , OUTPUT : 0 :      4X  1 :     1X
             INPUT  : 0 :      3X  1 :     2X
    <           (     0:     0).AUTO(.LOAD    ).STATUS(#88200000) FREQ:         2 =   40.00$
       DATA      WORD       ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        A1                 WORD         1 = 20.0$
                              1 WORD        SA        A        WORD         1 = 20.0$
     >          (     0:     0).AUTO(NONE     ).STATUS(#44000000) FREQ:         1 =   20.00$
       DATA      WORD                FREQ:     1
----------------------------------------------------------------------
R1              (     0:     0) MOREPORT:-2 DUPLICATIBLE : 0     FREQ :         2 =   40.00$
PORTUSINGS , OUTPUT : 0 :      3X  1 :     2X
             INPUT  : 0 :      4X  1 :     1X
     >          (     0:     0).AUTO(NONE     ).STATUS(#44000000) FREQ:         2 =   40.00$
       DATA      WORD                FREQ:     2
    <           (     0:     0).AUTO(.LOAD    ).STATUS(#88200000) FREQ:         1 =   20.00$
       DATA      WORD       ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        SA        A        WORD         1 = 20.0$
----------------------------------------------------------------------
R2              (     0:     0) MOREPORT:-1 DUPLICATIBLE : 0     FREQ :         1 =   20.00$
PORTUSINGS , OUTPUT : 0 :      5X
             INPUT  : 0 :      4X  1 :     1X
    <           (     0:     0).AUTO(.LOAD    ).STATUS(#A8200000) FREQ:         1 =   20.00$
       DATA      WORD       ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        SA        A        WORD         1 = 20.0$
----------------------------------------------------------------------
R3              (     0:     0) MOREPORT:-1 DUPLICATIBLE : 0     FREQ :         1 =   20.00$
PORTUSINGS , OUTPUT : 0 :      5X
             INPUT  : 0 :      4X  1 :     1X
    <           (     0:     0).AUTO(.LOAD    ).STATUS(#88200000) FREQ:         1 =   20.00$
       DATA      WORD       ADR  D-BITS     MODULE      PORT     S-BITS  FREQ
                              0 WORD        F1                 WORD         1 = 20.0$
```

```
*** GROUP : S
SA              (    4:    0) MOREPORT:-2 DUPLICATIBLE : 0    FREQ :         5 =  100.00%
PORTUSINGS , OUTPUT : 1 :     5X
             INPUT  : 0 :     3X  1 :    2X
     >A         (    4:    0).AUTO(NONE    ).STATUS(#54000000) FREQ:         5 =  100.00%
       DATA     WORD                FREQ:    5
       ADDRESS  WORD       ADR  D-BITS    MODULE       PORT    S-BITS  FREQ
                             0 WORD       I                   ASAA@A        1 = 20.0%
                             1 WORD       R1                  WORD          1 = 20.0%
                             2 WORD       I                   ASAA          4 = 80.0%
    < B         (    1:    0).AUTO(.LOAD   ).STATUS(#88200000) FREQ:         2 =   40.00%
       DATA     WORD       ADR  D-BITS    MODULE       PORT    S-BITS  FREQ
                             0 WORD       R1                  WORD          1 = 20.0%
                               WORD       B1                  WORD          1 = 20.0%
       ADDRESS  WORD       ADR  D-BITS    MODULE       PORT    S-BITS  FREQ
                             0 WORD       I                   ASAB          2 = 40.0%
-------------------------------------------------------------------------------------
J O I N T MODULE A P P E A R A N C E S
======================================
             I         FO        SA        SA>A      SA<B      B1        R1        A1        R2        R3
-----------------------------------------------------------------------------------------------------------
I           100.00     0.00    100.00    100.00     40.00     20.00     40.00     20.00     20.00     20.00
FO            0.00     0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
SA          100.00     0.00    100.00    100.00     40.00     20.00     40.00     20.00     20.00     20.00
SA>A        100.00     0.00    100.00    100.00     40.00     20.00     40.00     20.00     20.00     20.00
SA<B         40.00     0.00     40.00     40.00     40.00     20.00     20.00      0.00      0.00      0.00
B1           20.00     0.00     20.00     20.00     20.00     20.00      0.00      0.00      0.00      0.00
R1           40.00     0.00     40.00     40.00     20.00      0.00     40.00     20.00      0.00     20.00
A1           20.00     0.00     20.00     20.00      0.00      0.00     20.00     20.00      0.00     20.00
R2           20.00     0.00     20.00     20.00      0.00      0.00      0.00      0.00     20.00      0.00
R3           20.00     0.00     20.00     20.00      0.00      0.00     20.00     20.00      0.00     20.00
F1           20.00     0.00     20.00     20.00      0.00      0.00     20.00     20.00      0.00     20.00
JOINT APPEARANCE OF MICROINSTR.FIELDS
             ASAA      FB1       ASAB      FA1       ASAA@A
-----------------------------------------------------------------------------------------------------------
ASAA         80.00    20.00     40.00      0.00     20.00
FB1          20.00    20.00     20.00      0.00      0.00
ASAB         40.00    20.00     40.00      0.00     20.00
FA1           0.00     0.00      0.00     20.00      0.00
ASAA@A       20.00     0.00     20.00      0.00     20.00
AVERAGE NUMBER OF FIXED BITS : +0.000E+00,   STD. DEVIATION : +0.000E+00
AVERAGE NUMBER OF VAR FIELDS : +1.799E+00,   STD. DEVIATION : +9.797E-01
# OF ENABLE BITS :     5
# OF FCT-SEL BITS:     0
# OF MPX-ADR BITS:     4
# OF MPXER INPUTS:     7
# OF CONNECTIONS :    16
```

Fig. 10.1.4

Compare this listing with the general form in Appendix E and with the description of %PRINTHARDWARE in chapter 9.2.4.

10 2 Computation of a Bessel Function

The following FORTRAN subroutine from the IBM scientific subroutine package computes the J-Bessel function

```
      SUBROUTINE BESJ(X,N,BJ,D,IER)
      BJ=0.0
      IF (N) 10,20,20
10    IER=1
      RETURN
20    IF (X) 30,30,31
30    IER=2
      RETURN
31    IF (X-15.) 32,32,34
32    NTEST=20.+10.*X-X*X /3
      GOTO 36
34    NTEST=90.+X/2
36    IF (N-NTEST) 40,38,38
38    IER=4
      RETURN
40    IER=0
      N1=N+1
      BPREV=.0
C     COMPUTE STARTING VALUE OF M
      IF (X-5.) 50,60,60
50    MA=X+6.
      GOTO 70
60    MA=1.4*X+60./X
70    MB=N+IFIX(X)/4+2
      MZERO=MA
      IF (MA-MB) 80,90,90
80    MZERO=MB
C     SET UPPER LIMIT OF M
90    MMAX=NTEST
100   DO 190 M=MZERO,MMAX,3
C     SET F(M),F(M-1)
      FM1=1.0E-28
      FM=.0
      ALPHA=.0
      IF (M-(M/2)*2) 120,110,120
110   JT=-1
      GOTO 130
120   JT=1
130   M2=M-2
      DO 160 K=1,M2
      MK=M-K
      BMK=2.*FLOAT(MK)*FM1/X-FM
      FM=FM1
      FM1=BMK
      IF (MK-N-1) 150,140,150
140   BJ=BMK
150   JT=-JT
      S=1+JT
160   ALPHA=ALPHA+BMK*S
      BMK=2.*FM1/X-FM
      IF (N) 180,170,180
170   BJ=BMK
180   ALPHA=ALPHA+BMK
      BJ=BJ/ALPHA
      IF (ABS(BJ-BPREV)-ABS(D*BJ))200,200,190
190   BPREV=BJ
      IER=3
200   RETURN
      END
```

distributors can be inserted.

Fig. 10.2.3 shows module declarations which introduce a limit to the allowable hardware. Macros are declared in order to use a compare unit with simultaneously accessible comparison results (e.g. SN 7485) instead of a compare unit with a function input.

```
ADDMODULE
 Bc.BIT(0)LT.BIT(1)LE.BIT(2)EQ.BIT(3)GE.BIT(4)GT,
 Ba,   Bd,Bm.DUPLICATE(2),Bs.DUPLICATE(1),
 Si<A,Si>B,Si<>C, Sr<A,Sr>B,Sr<>C, SHLP<A, SHLP>B,SHLP<>C;
MACRO Bc(< ) && Bc(X).LT  ENDMACRO
MACRO Bc(<=) && Bc(X).LE  ENDMACRO
MACRO Bc(= ) && Bc(X).EQ  ENDMACRO
MACRO Bc(>=) && Bc(X).GE  ENDMACRO
MACRO Bc(> ) && Bc(X).GT  ENDMACRO
USING S
INSERT INP
```

<u>Fig 10.2.3</u>

This declaration causes the MSS to generate the following program :

```
"MIMOLA VERSION 2.2 OF MAR 29, 1979 RULES 84 & 116 "
"EXECUTION DATE :  05/16/79  16:12:05"
INSERT LIBRARY
PROGRAM
        SUBROUTINE   besj
Lsub.1
        sub (x,n,bj,d,ier);
Lz1.1
        Sr<A(bj):=0,
        if Si>B(n)/0->Bc(X).LT
          then
            Si<A(ier):=1,
            return fi
        ;
L20.1
        if Sr>B(x)/0->Bc(X).LE
          then
            Si<A(ier):=2,
            return fi
        ;
L31.1
        if Sr>B(x)/15->Bc(X).LE
          then
            Si<A(ntest):=20./10./Sr>B(x)->Bm(*r)->Ba(+r)/Sr>B(x)/Sr>B(x)->Bm_A
            (*r)/3.->Bd(/r)->Bs(-r)
          else
            Si<A(ntest):=90./Sr>B(x)/Sr>C(two)->Bd(/r)->Ba(+r) fi
        ;
```

```
L36.1
        if Si>B(n)/Si>C(ntest)->Bc(X).GE
          then
            Si<A(ier):=4,
            return fi
        ;
L40.1
        Si<A(ier):=0,
        Si<C(n1):=Si>B(n)->A(.INCR),
        Sr<A(bprev):=0.,
        SHLP<A(iden101):=Sr>C(n)/Sr>B(x)->A_A(.IFIX)/4->Bd(/)->Ba(+);
L40.2
        Si<A(mb):=SHLP>B(iden101)/2->Ba(+);
L40.3
        if Sr>B(x)/5.->Bc(X).LT
          then
            Si<A(ma):=Sr>B(x)/6.->Ba(+r)
          else
            Si<A(ma):=60./Sr>B(x)->Bd(/r)/1.4/Sr>B(x)->Bm(*r)->Ba(+r) fi
        ;
Lz6.1
        if Si>B(ma)/Sr>B(mb)->Bc(X).LT
          then
            Si<A(mzero):=Si>C(mb)
          else
            Si<A(mzero):=Si>C(mb) fi
        ;
L90.1
        Si<A(mmax):=Si>B(ntest)=V,
        for m from Si>C(mzero) by 3 to V;
$FACTOR = 4
Lz81.1
        do (m) Sr<A(fm1):=1.E-28,
        Sr<C(fm):=0,
        for k to D(m)/Sr>B(two)->Bs(-),
        if D(m)/V/2->Bd(/)/V1->Bm(*)->Bc(X).EQ

          then
            Si<A(jt):=-1
          else
            Si<A(jt):=1 fi
        ;
Lz81.2
        Sr<A(alfa):=0;
$FACTOR =16
L121.1
        do (k) Si<A(mk):=D(m)/D(k)->Bs(-)=V,
        SHLP<A(iden101):=D(m)/D(k)->Bs(-),
        Sr<A(bmk):=2/V->A(.FLOAT)->Bm(*r)/Sr>B(fm1)->Bm_A(*r)/Sr>C(x)->Bd(/r)/
        Si>B(fm)->Bs_A(-r)=V1,
        SHLP<C(iden102):=2/V->A(.FLOAT)->Bm(*r)/Sr>B(fm1)->Bm_A(*r)/Sr>C(x)->Bd
        (/r)/Si>B(fm)->Bs_A(-r),
        Si>C(jt)->A_A(.COMPL);
L121.2
        Sr<A(fm):=Sr>B(fm1),
        Sr<C(fm1):=SHLP>B(iden102);
L121.3
        Sr<A(s):=Sr>B(s)->A(.INCR)=V2,
        SHLP<A(iden103):=Sr>B(s)->A(.INCR);
L121.4
        Sr<A(alfa):=Sr>B(alfa)/SHLP>B(iden102)/SHLP>C(iden103)->Bm(*r)->Ba(+r)
        ;
L121.5
        if SHLP>B(iden101)/Si>B(n)->A(.DECR)->Bc(X).EQ
          then
            Sr<A(bj):=SHLP>C(iden102) fi

        ,
        od (k);
        $FACTOR = 4
L181.1
        SHLP<A(iden101):=2./Sr>B(fm1)->Bm(*r)/Sr>C(x)->Bd(/r);
L181.2
        Sr<A(bmk):=SHLP>B(iden101)/Sr>B(fm)->Bs(-r)=V,
        SHLP<A(iden101):=SHLP>B(iden101)/Sr>b(fm)->Bs(-r);
L181.3
        Sr<A(alfa):=Sr>B(alfa)/SHLP>B(iden101)->Ba(+r),
        if Si>B(n)/0->Bc(X).EQ
          then
            Sr<C(bj):=SHLP>B(iden101) fi
```

```
L21.1
         Sr>B(bj)/Sr>C(alpha)->Bd(/r);
L21.2
         Sr>B(bj)/Sr>C(bprev)->Bm(*r)->A(.ABS)=V1,
         SHLP<A(iden101):=Sr>B(bj)/Sr>C(bprev)->Bm(*r)->A(.ABS);
L21.3
         Sr>B(d)/Sr>C(bj)->Bm(*r)->A(.ABS)=V2,
         if SHLP>B(iden101)/V2->Bo(X).LE
           then
             return fi
         ;
 L190.1
         Sr<A(bprev):=Sr>B(bj),
         od (m);
         $FACTOR = 1
 L22.1
         Si<A(ier):=3,
         return;
         ENDSUB
```

REFERENCES

(1) G. Zimmermann, Report on the Computer Architecture Design Language MIMOLA, Bericht Nr. 4/77 des Instituts fur Informatik and Praktische Mathematik, Kiel, 1977

(2) P. Marwedel, The MIMOLA Design System: Detailed Description of the Software System, 16th Design Automation Conference Proceedings, 1979

(3) U. Zimmermann, Ein Compiler zur Sequentialisierung von MIMOLA-Programmen, Diploma-Thesis, Kiel, 1979

(4) R. Hollenbach, Ein Macro Prozessor fur das MIMOLA Software System (preliminary title), Diploma-Thesis (in preparation), **Kiel, 1979**

(5) G. Zimmermann, Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA, Informatik Fachberichte 5, 465-478, Berlin, 1976

(6).G. Zimmermann, The MIMOLA Design System: A Computer Aided Digital Processor Design Method, 16th Design Automation Conference Proceedings, 1979

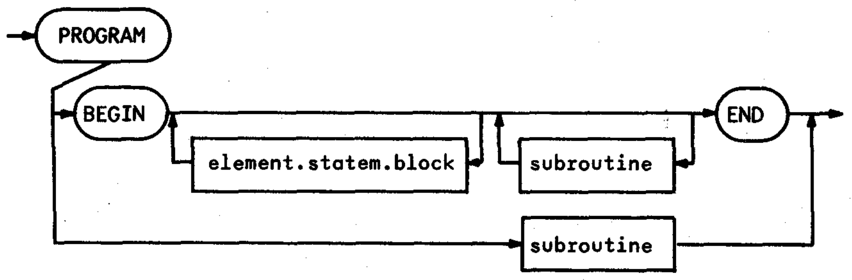
(7) M.R. Barbacci, G.E. Barnes, R.G. Lattell, D.P. Siewiorek, The ISPS Computer Description Language, technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, P.A.,1977

(8) S. Wendt, Models and Structures for Microprogramming, EUROMICRO Symp. Microprocessing and Microprogramming, Venice, 1976

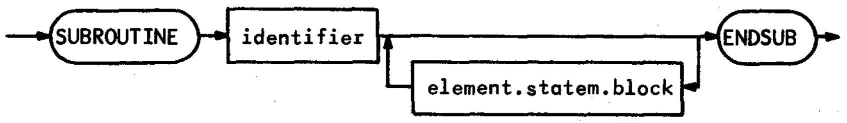
APPENDIX A

Program Part Syntax Diagrams

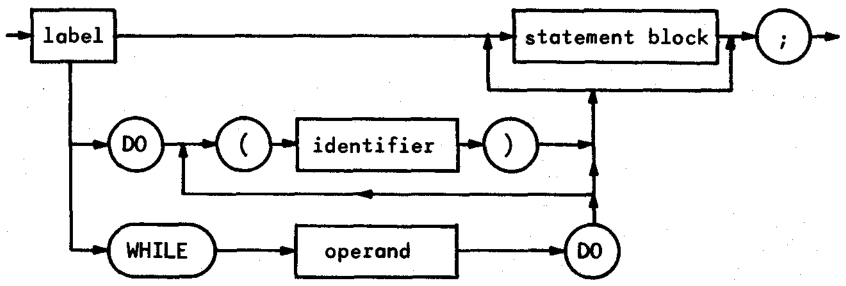
program axiom



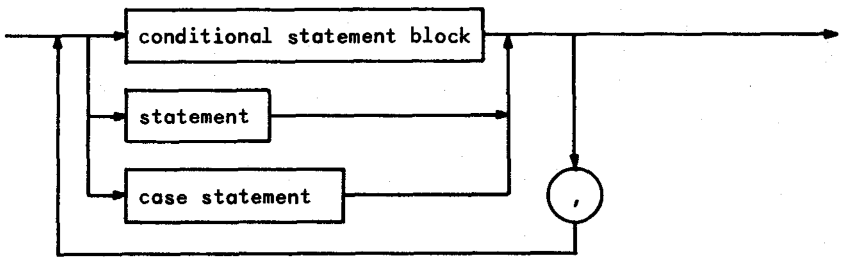
subroutine



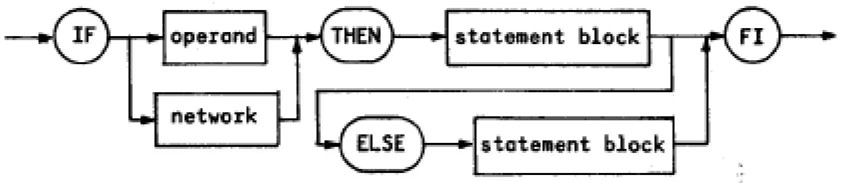
elementary statement block



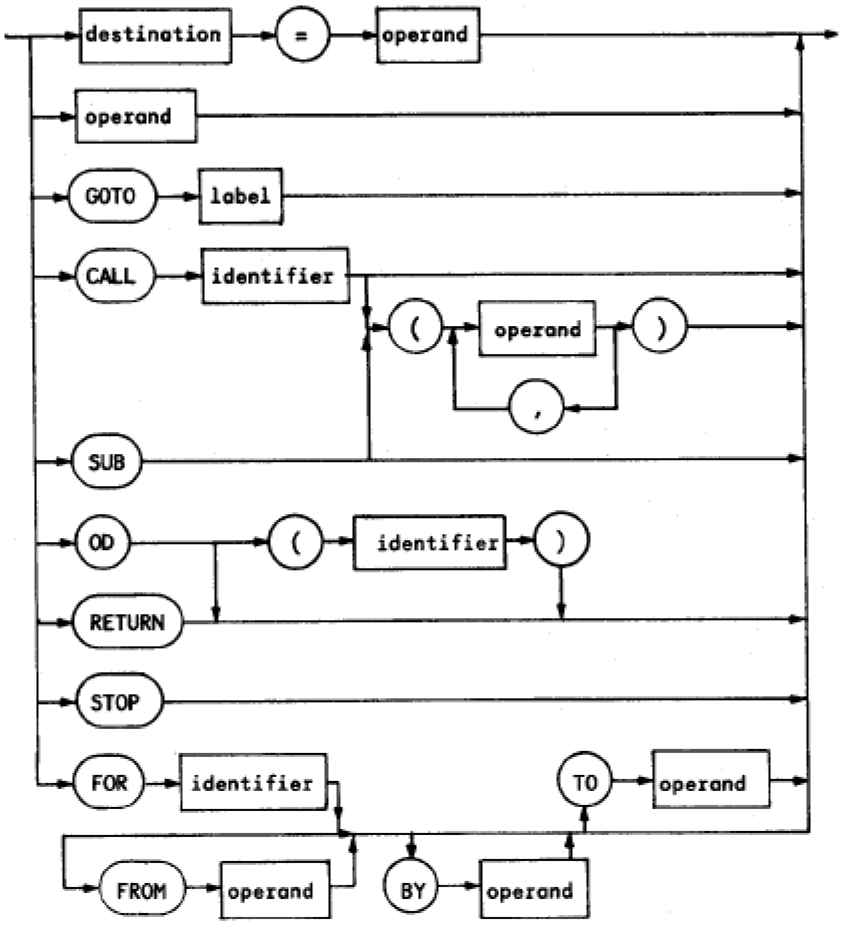
statement block



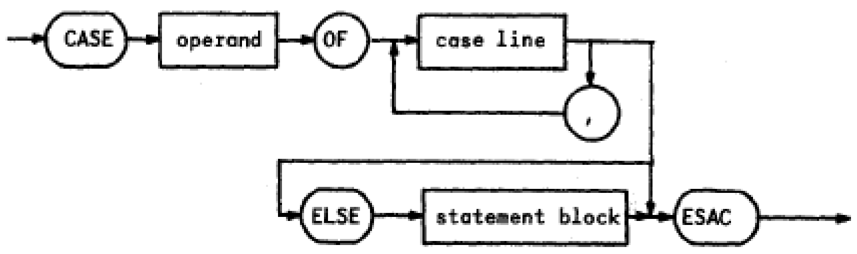
conditional statement block



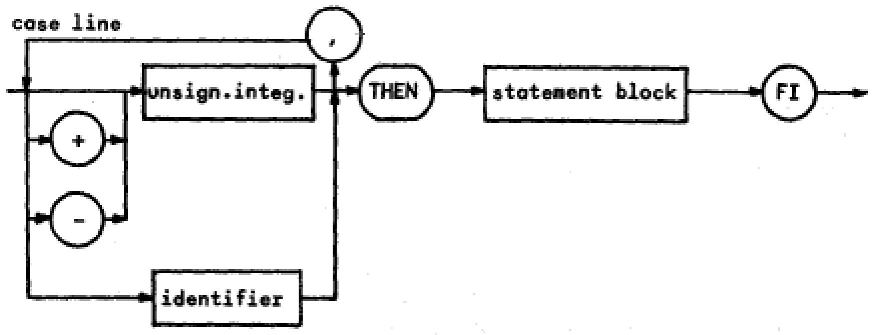
statement



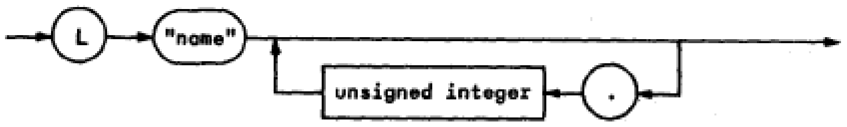
case statement



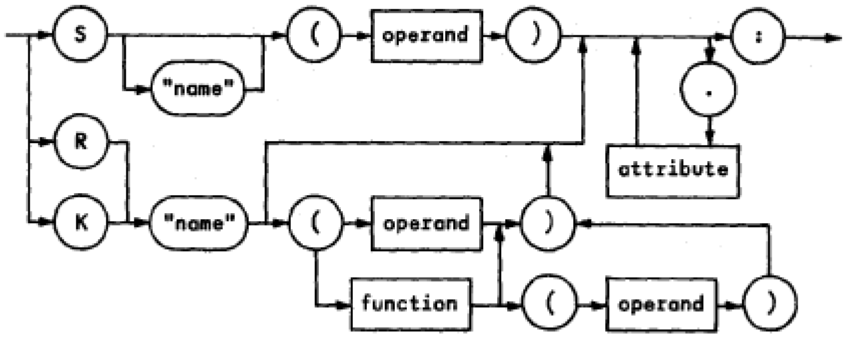
case line



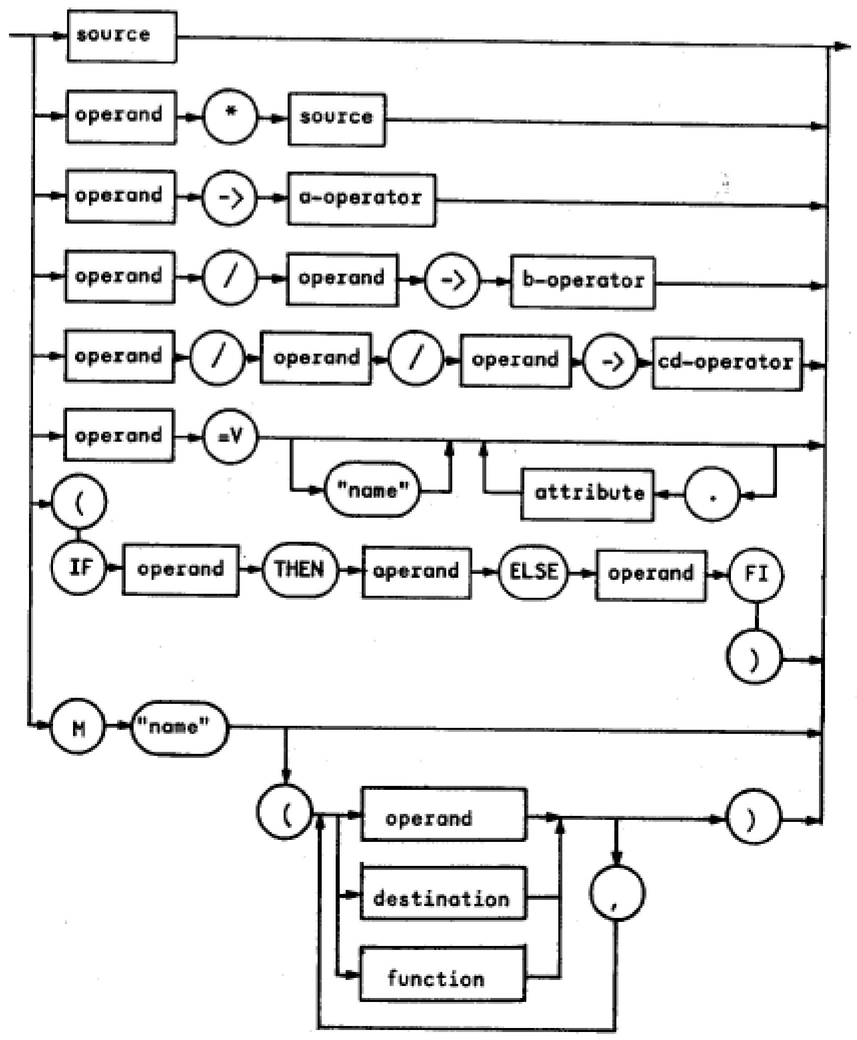
label



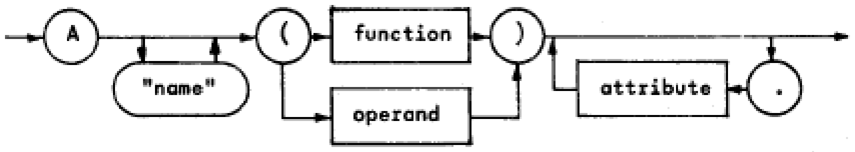
destination



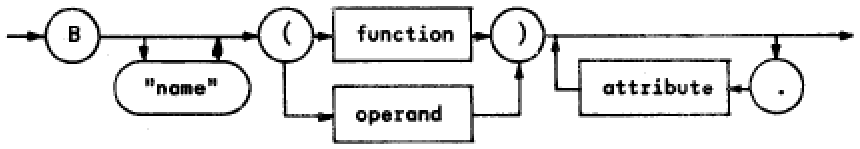
operand


source
operand
*
source
operand
->
a-operator
operand
/
operand
->
b-operator
operand
/
operand
/
operand
->
cd-operator
operand
=V
"name"
attribute
.
(
IF
operand
THEN
operand
ELSE
operand
FI
)
M
"name"
(
operand
destination
function
,
)

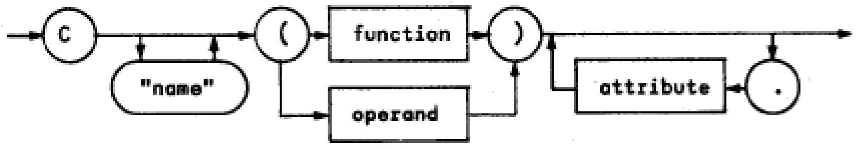
a-operator



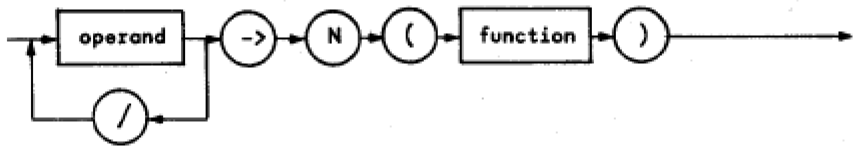
b-operator



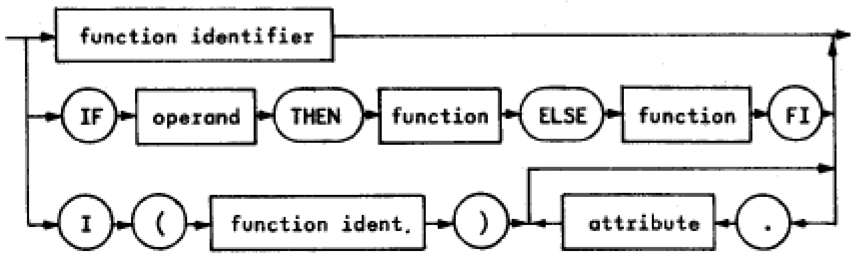
c-operator



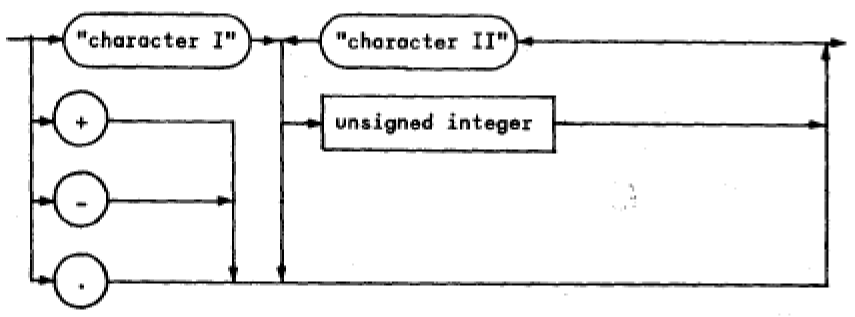
network



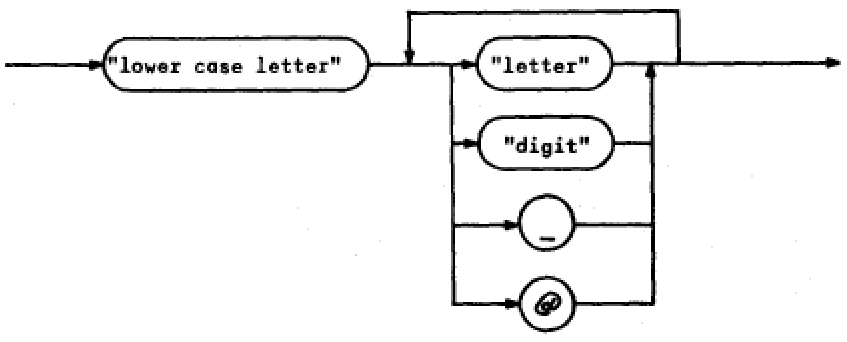
function



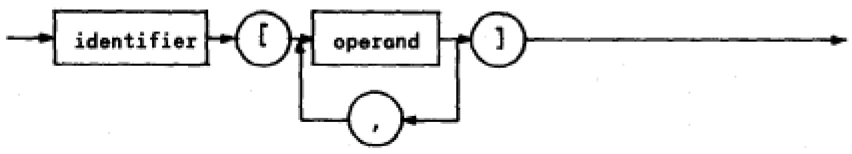
function identifier



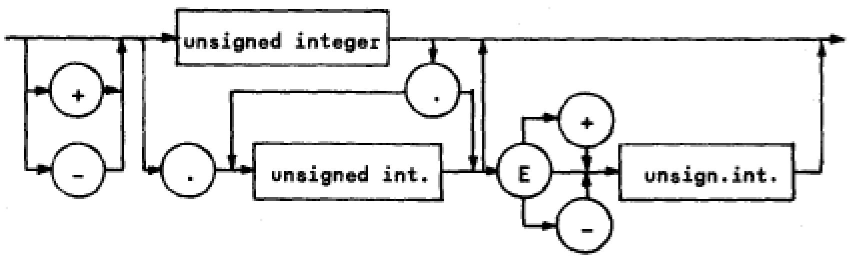
identifier



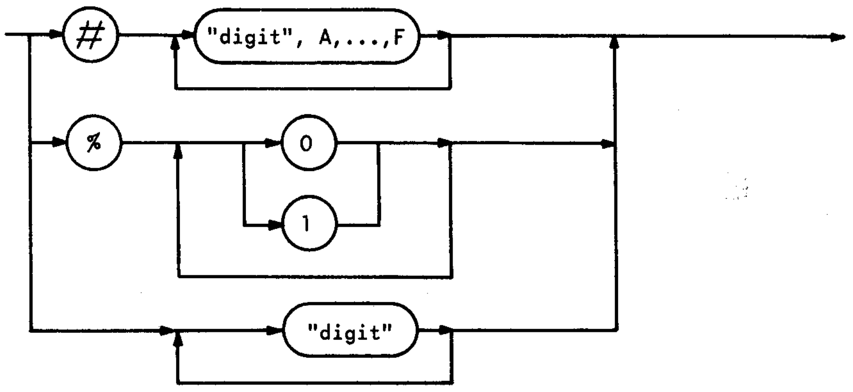
array identifier



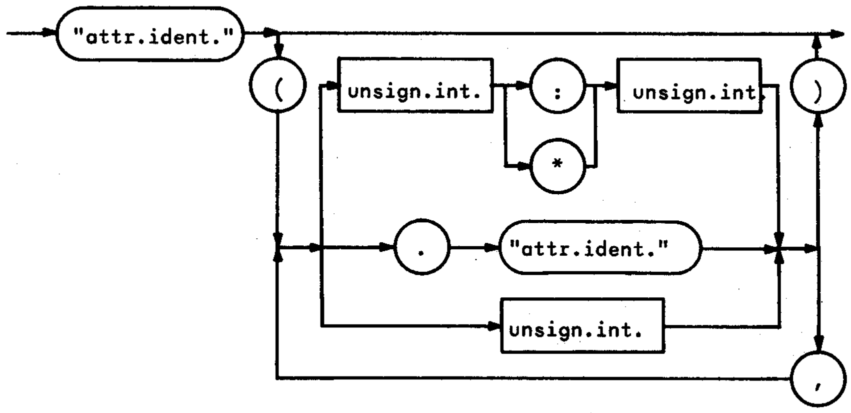
constant



unsigned integer

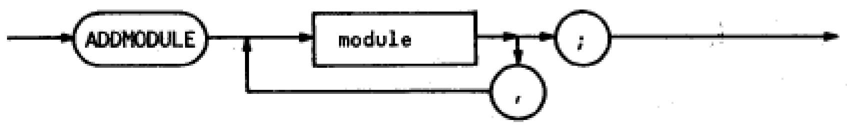


attribute

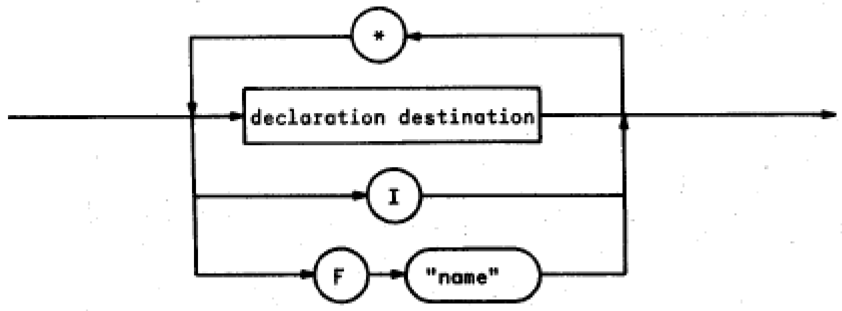


Declaration Part Syntax Diagrams

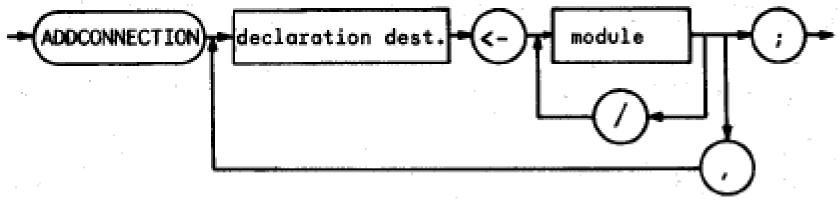
addmodule axiom



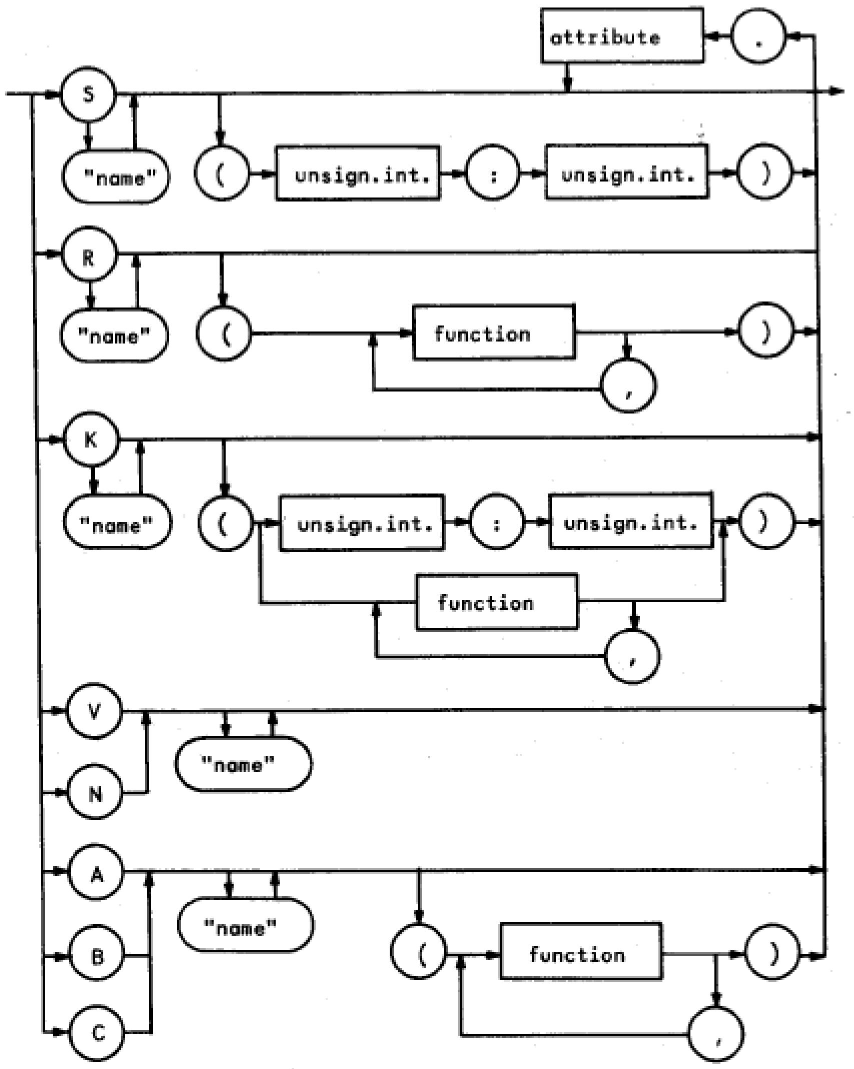
module



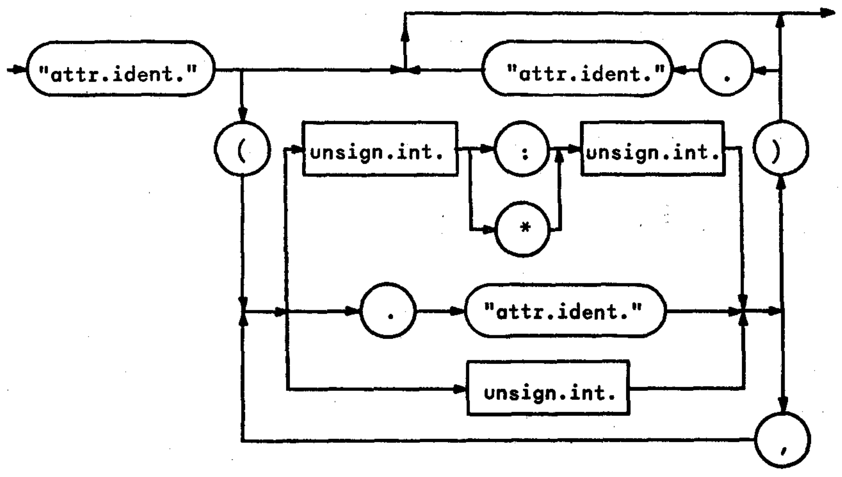
addconnection axiom



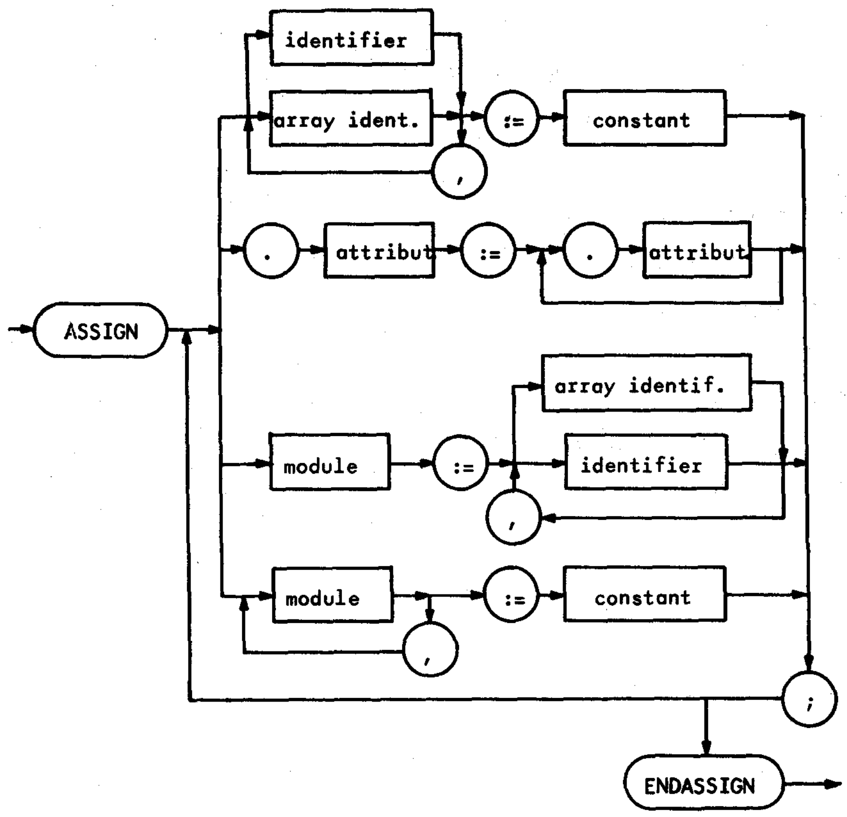
declaration destination



attribute



Assignment Part Syntax



Pretranslator Syntax

"string" ::= "letter" $\{$ "letter" | "digit" $\}_{0}^{*}$

"digit" ::= $\{$ Ø | .. | 9 $\}_{0}^{1}$

"letter" ::= $\{$ "upper case letter" | "lower case letter" $\}_{1}^{1}$

"upper case letter" ::= $\{$ A | .. | Z $\}^{1}$

"lower case letter" ::= $\{$ a | .. | z | _ | @ $\}^{1}$

"name" ::= $\{$ "string" $\}_{0}^{1}$ $\{$ < | > | <> | <"string" | >"string" | <>"string" $\}_{0}^{1}$

"character I" ::= $\{$ * | / | = | : | ^ | < | > $\}^{1}$

"character II" ::= $\{$ "letter" | "character I" | + | - | . $\}_{1}^{1}$

Reserved words are:

IF, FI, DO, OD, BY, TO, FOR, END, SUB, THEN, ELSE,
FROM, GOTO, CALL, STOP, BEGIN, WHILE, RETURN, ENDSUB,
ENDASSIGN, OF, CASE, ESAC, ENDMACRO, SUBROUTINE

They may be written either with upper or lower case characters.
"attr.ident" is a string after '.' or ') ' which is neither a reserved word nor a part of a function identifier.
"Identifier" is a string which is neither a reserved word nor an "attr.ident" and which does not start with
A, B, C, D, E, F, I, K, L, M, N, R, S, T, V, X.
(Identifiers starting with an upper case letter are not recommended because the above list may expand in future.)

Appendix B : Syntax Rules in BNF-Notation

```
Column 1      : p indicates a rule applicatable to the program part
Column 2      : m   -  "    -       "      -    "    -  macro definition part
Column 3      : h   -  "    -       "      -    "    -  hardware declaration
Column 4      : a   -  "    -       "      -    "       assignment part
Column 6-8    : Internal rule's number
Column 10-23  : Left side of production
Column 29-85  : Right sides of productions

p        1 <PROGRAM>      ::=    <PGM HEAD>    END
p        2 <PROGRAM>      ::=    <PGM BODY>    END
p        3 <PROGRAM>      ::=    <SUBROUTINE>
p        4 <PGM HEAD>     ::=    BEGIN
p        5 <PGM HEAD>     ::=    <PGM HEAD>    <ESB>
p        6 <PGM BODY>     ::=    <PGM HEAD>    <SUBROUTINE>
p        7 <PGM BODY>     ::=    <PGM BODY>    <SUBROUTINE>
pm       8 <ESB>          ::=    <LABEL>    <STMT BLOCK> ;
pm       9 <ESB>          ::=    <WHILE DO>    ;
pm      10 <ESB>          ::=    <WHILE DO>    <STMT BLOCK> ;
pm      11 <WHILE DO>     ::=    <LABEL>    DO   (   <IDENTIFIER>   )
pm      12 <WHILE DO>     ::=    <LABEL>    WHILE    <OPERAND>    DO
pm     221 <WHILE DO>     ::=    <LABEL>  WHILE  <OPERAND> DO   ( <IDENTIFIER> )
pm      13 <SUBROUTINE>   ::=    <SBR HEAD>    ENDSUB
pm      14 <SBR HEAD>     ::=    SUBROUTINE    <IDENTIFIER>
pm      15 <SBR HEAD>     ::=    <SBR HEAD>    <ESB>
pm      16 <STMT BLOCK>   ::=    <STATEMENT>
pm      17 <STMT BLOCK>   ::=    <COND STMT>
pm      18 <STMT BLOCK>   ::=    <STMT BLOCK> ,    <STATEMENT>
pm      19 <STMT BLOCK>   ::=    <STMT BLOCK> ,    <COND STMT>
pm      20 <STATEMENT>    ::=    <DESTINATIO> =    <OPERAND>
pm      22 <STATEMENT>    ::=    <OPERAND>
pm      23 <STATEMENT>    ::=    GOTO    <LABEL>
pm      24 <STATEMENT>    ::=    CALL    <IDENTIFIER>
pm      25 <STATEMENT>    ::=    <CALL STMT>   )
pm      26 <STATEMENT>    ::=    SUB
pm      27 <STATEMENT>    ::=    <SBR ENTER>   )
pm      28 <STATEMENT>    ::=    OD
pm     222 <STATEMENT>    ::=    OD    (   <IDENTIFIER>   )
pm      29 <STATEMENT>    ::=    RETURN
pm      30 <STATEMENT>    ::=    STOP
pm      31 <STATEMENT>    ::=    FOR    <IDENTIFIER>
pm      32 <STATEMENT>    ::=    FOR    <IDENTIFIER> FROM    <OPERAND>
pm      33 <STATEMENT>    ::=    FOR <IDENTIFIER> FROM <OPERAND> BY <OPERAND>
pm      34 <STATEMENT>    ::=    FOR<IDENTIFIER>FROM<OPERAND>BY<OPERAND>TO<OPERAND>
pm      35 <STATEMENT>    ::=    FOR    <IDENTIFIER> FROM    <OPERAND>    TO    <OPERAND>
pm      36 <STATEMENT>    ::=    FOR    <IDENTIFIER> BY    <OPERAND>
pm      37 <STATEMENT>    ::=    FOR    <IDENTIFIER> BY <OPERAND> TO <OPERAND>
pm      38 <STATEMENT>    ::=    FOR    <IDENTIFIER> TO    <OPERAND>
pm      39 <CALL STMT>    ::=    CALL    <IDENTIFIER> (    <OPERAND>
pm      40 <CALL STMT>    ::=    <CALL STMT>    ,    <OPERAND>
pm      41 <SBR ENTER>    ::=    SUB    (    <OPERAND>
pm      42 <SBR ENTER>    ::=    <SBR ENTER>    ,    <OPERAND>
pm      43 <COND STMT>    ::=    IF    <OPERAND>    THEN    <STMT BLOCK> FI
pm      44 <COND STMT>    ::=    IF <OPERAND> THEN <STMT BLOCK> ELSE <STMT BLOCK> FI
pm      45 <COND STMT>    ::=    IF    <NETWORK>    THEN    <STMT BLOCK> FI
pm      46 <COND STMT>    ::=    IF <NETWORK> THEN <STMT BLOCK> ELSE <STMT BLOCK> FI
pm      47 <DESTINATIO>   ::=    <STORAGE>    :
pm      48 <DESTINATIO>   ::=    <REGISTER>    :
pm      49 <DESTINATIO>   ::=    <STACK>    :
pm a    50 <OPERAND>      ::=    <SOURCE>
pm      51 <OPERAND>      ::=    <OPERAND>    *    <SOURCE>
pm      52 <OPERAND>      ::=    <OPERAND>    -    >    <A-OPERATOR>
pm      53 <OPERAND>      ::=    <OPERAND>    /    <OPERAND>    -    >    <B-OPERATOR>
pm      54 <OPERAND>      ::=   <OPERAND>/<OPERAND>/ <OPERAND> - > <C-OPERATOR>
pm      55 <OPERAND>      ::=   <OPERAND>   =   <DISTRIBUTOR
pm      56 <CD.OP HEAD>   ::=   (   IF   <OPERAND>   THEN   <OPERAND>
pm     132 <OPERAND>      ::=   <CD.OP HEAD> ELSE   <OPERAND>   FI   )
pm     184 <OPERAND>      ::=   M   "NAME"
pm     185 <OPERAND>      ::=   <MACALL HEA> )
pm a    57 <SOURCE>       ::=   <CONSTANT>
pm a    58 <SOURCE>       ::=   <IDENTIFIER>
pm a    59 <SOURCE>       ::=   <ARRAY IDEN>
pm     206 <SOURCE>       ::=   <LABEL>
pm      60 <SOURCE>       ::=   X
pm      61 <SOURCE>       ::=   <STORAGE>
pm      62 <SOURCE>       ::=   <REGISTER>
pm      63 <SOURCE>       ::=   <STACK>
pm      64 <SOURCE>       ::=   <INSTRUCTIO>
pm      65 <SOURCE>       ::=   <DISTRIBUTOR
pm      66 <SOURCE>       ::=   <FIX CONST>
pm      67 <SOURCE>       ::=   <CONTR VAR>
pm      68 <LABEL>        ::=   L   "NAME"
pm      69 <LABEL>        ::=   <LABEL>   .   "U.INTEGER"
pm a    70 <STORAGE>      ::=   S   (   <OPERAND>   )
pm a    71 <STORAGE>      ::=   S   "NAME"   (   <OPERAND>   )
pmh     72 <STORAGE>      ::=   <STORAGE>   .   <ATTRIBUTE>
pmha    73 <REGISTER>     ::=   R   "NAME"
pm      74 <REGISTER>     ::=   R   "NAME"   (   <OPERAND>   )
pmh     75 <REGISTER>     ::=   R   "NAME"   (   <FUNCTION>   )
pmh     76 <REGISTER>     ::=   R   "NAME"   (   <FUNCTION> (  <OPERAND>  ) )
pmh     77 <REGISTER>     ::=   <REGISTER>   .   <ATTRIBUTE>
pmh     78 <STACK>        ::=   K   "NAME"
pm a    79 <STACK>        ::=   K   "NAME"   (   <OPERAND>   )
pmh     80 <STACK>        ::=   K   "NAME"   (   <FUNCTION>   )
pm      81 <STACK>        ::=   K   "NAME"   (   <FUNCTION> (  <OPERAND>  ) )
pmh     82 <STACK>        ::=   <STACK>   .   <ATTRIBUTE>
pmh     83 <INSTRUCTIO>   ::=   I
pm      84 <INSTRUCTIO>   ::=   I   (   <IDENTIFIER> )
pmh     85 <INSTRUCTIO>   ::=   <INSTRUCTIO> .   <ATTRIBUTE>
pmh     86 <DISTRIBUTOR   ::=   V
pmh     87 <DISTRIBUTOR   ::=   V   "NAME"
pmh     88 <DISTRIBUTOR   ::=   <DISTRIBUTOR .   <ATTRIBUTE>
pmh     89 <FIX CONST>    ::=   F   "NAME"
pm      90 <FIX CONST>    ::=   F   "NAME"   (   <OPERAND>   )
pmh     91 <FIX CONST>    ::=   <FIX CONST>  .   <ATTRIBUTE>
pm      92 <CONTR VAR>    ::=   D
pm      93 <CONTR VAR>    ::=   D   (   "U.INTEGER"  )
pm      94 <CONTR VAR>    ::=   D   (   <IDENTIFIER> )
pm      95 <CONTR VAR>    ::=   <CONTR VAR>  .   <ATTRIBUTE>
pmh     96 <A-OPERATOR>   ::=   A   (   <FUNCTION>   )
pmh     97 <A-OPERATOR>   ::=   A   "NAME"   (   <FUNCTION>   )
pm      98 <A-OPERATOR>   ::=   A   (   <OPERAND>   )
pm      99 <A-OPERATOR>   ::=   A   "NAME"   (   <OPERAND>   )
pmh    100 <A-OPERATOR>   ::=   <A-OPERATOR> .   <ATTRIBUTE>
pmh    101 <B-OPERATOR>   ::=   B   (   <FUNCTION>   )
pmh    102 <B-OPERATOR>   ::=   B   "NAME"   (   <FUNCTION>   )
pm     103 <B-OPERATOR>   ::=   B   (   <OPERAND>   )
pm     104 <B-OPERATOR>   ::=   B   "NAME"   (   <OPERAND>   )
pmh    105 <B-OPERATOR>   ::=   <B-OPERATOR> .   <ATTRIBUTE>
pmh    106 <C-OPERATOR>   ::=   C   (   <FUNCTION>   )
pmh    107 <C-OPERATOR>   ::=   C   "NAME"   (   <FUNCTION>   )
pm     108 <C-OPERATOR>   ::=   C   (   <OPERAND>   )
pm     109 <C-OPERATOR>   ::=   C   "NAME"   (   <OPERAND>   )
pmh    110 <C-OPERATOR>   ::=   <C-OPERATOR> .   <ATTRIBUTE>
pm     111 <NETWORK>      ::=   <OPERAND>   -   >   N   (   <FUNCTION>   )
pm     112 <NETWORK>      ::=   <OPERAND>   /   <NETWORK>
pmh    113 <FUNCTION>     ::=   "FUNCT-ID"
pm     114 <FUNCTION>     ::=   IF <OPERAND> THEN <FUNCTION> ELSE <FUNCTION> FI
pm     115 <FUNCTION>     ::=   <FCT INST P>
pm     116 <FCT INST P>   ::=   I   (   "FUNCT-ID"   )
pm     117 <FCT INST P>   ::=   <FCT INST P> .   <ATTRIBUTE>
pm a   118 <IDENTIFIER>   ::=   "STRING EXP"
pm a   119 <IDENTIFIER>   ::=   "LOWLETTSTR"
pm a   120 <ARRAY IDEN>   ::=   <AR ID HEAD> ]
pm a   121 <AR ID HEAD>   ::=   <IDENTIFIER> [   <OPERAND>
pm a   122 <AR ID HEAD>   ::=   <AR ID HEAD> ,   <OPERAND>
pmh    123 <ATTRIBUTE>    ::=   "ATTRB-ID"
pmh    124 <ATTRIBUTE>    ::=   <ATTR HEAD>  )
pmh    125 <ATTR HEAD>    ::=   "ATTRB-ID"   (   "U.INTEGER"
pmh    217 <ATTR HEAD>    ::=   "ATTRB-ID"   (   .   "ATTRB-ID"
pmh    126 <ATTR HEAD>    ::=   "ATTRB-ID"   (   "U.INTEGER"  :   "U.INTEGER"
pmh    179 <ATTR HEAD>    ::=   "ATTRB-ID"   (   "U.INTEGER"  *   "U.INTEGER"
pmh    127 <ATTR HEAD>    ::=   <ATTR HEAD>  ,   "U.INTEGER"
pmh    218 <ATTR HEAD>    ::=   <ATTR HEAD>  ,   .   "ATTRB-ID"
pmh    128 <ATTR HEAD>    ::=   <ATTR HEAD>  ,   "U.INTEGER"  :   "U.INTEGER"
pmh    180 <ATTR HEAD>    ::=   <ATTR HEAD>  ,   "U.INTEGER"  *   "U.INTEGER"
pm a   129 <CONSTANT>     ::=   <SIGNED INT>
pm a   130 <CONSTANT>     ::=   <FIX POI CO>
pm a   131 <CONSTANT>     ::=   <FIX POI CO> E   <SIGNED INT>
pm a   133 <FIX POI CO>   ::=   <SIGNED INT> .
pm a   134 <FIX POI CO>   ::=   <SIGNED INT> .   "U.INTEGER"
pm a   135 <FIX POI CO>   ::=   .   "U.INTEGER"
pm a   136 <FIX POI CO>   ::=   <SIGN>   .   "U.INTEGER"
pm a   137 <SIGNED INT>   ::=   "U.INTEGER"
pm a   138 <SIGNED INT>   ::=   <SIGN>   "U.INTEGER"
pm a   139 <SIGN>         ::=   +
pm a   140 <SIGN>         ::=   -
pm     186 <MACALL HEA>   ::=   M   "NAME"   (   <OPERAND>
pm     187 <MACALL HEA>   ::=   M   "NAME"   (   <DESTINATIO>
pm     188 <MACALL HEA>   ::=   M   "NAME"   (   <FUNCTION>
pm     189 <MACALL HEA>   ::=   <MACALL HEA> ,   <OPERAND>
pm     190 <MACALL HEA>   ::=   <MACALL HEA> ,   <DESTINATIO>
pm     191 <MACALL HEA>   ::=   <MACALL HEA> ,   <FUNCTION>
 m     192 <PROGRAM>      ::=   <PGM HEAD>   ENDMACRO
pm     194 <STMT BLOCK>   ::=   <CASE STM>
pm     195 <STMT BLOCK>   ::=   <STMT BLOCK> ,   <CASE STM>
pm     196 <CASE STM>     ::=   <CASE HEAD>  <CASE TAIL>
pm     197 <CASE HEAD>    ::=   CASE   <OPERAND>   OF
pm     198 <CASE HEAD>    ::=   <CASE HEAD>  <CASE LINE>
pm     199 <CASE TAIL>    ::=   <CASE LINE>  ESAC
pm     200 <CASE TAIL>    ::=   <CASE LINE>  ELSE   <STMT BLOCK> ESAC
pm     201 <CASE LINE>    ::=   <CASE LIST>  THEN   <STMT BLOCK> FI
pm     202 <CASE LIST>    ::=   <SIGNED INT>
pm     203 <CASE LIST>    ::=   <IDENTIFIER>
pm     204 <CASE LIST>    ::=   <CASE LIST>  ,   <SIGNED INT>
pm     205 <CASE LIST>    ::=   <CASE LIST>  ,   <IDENTIFIER>
  h    141 <FUNCTION>     ::=   <FUNCTION>   ,   <FUNCTION>
  ha   142 <STORAGE>      ::=   S   (   "U.INTEGER"   :   "U.INTEGER"   )
  ha   143 <STORAGE>      ::=   S   "NAME"   (   "U.INTEGER"   :   "U.INTEGER"   )
  ha   144 <STACK>        ::=   K   "NAME"   (   "U.INTEGER"   :   "U.INTEGER"   )
  h    145 <ATTRIBUTE>    ::=   <ATTR HEAD>  )   "ATTRB-ID"
  h    147 <ADDMOD AX>    ::=   <ADDMOD BOD> ;
  h    148 <ADDMOD BOD>   ::=   <MODULE>
  h    149 <ADDMOD BOD>   ::=   <ADDMOD BOD> ,   <MODULE>
  h    150 <ADDCONN AX>   ::=   <CONN BODY>  ;
  h    151 <CONN BODY>    ::=   <CONNECTION>
  h    152 <CONN BODY>    ::=   <CONN BODY>  ,   <CONNECTION>
  h    153 <CONNECTION>   ::=   <DCL DEST>   <   -   <MODULE>
  h    154 <CONNECTION>   ::=   <CONNECTION> /   <MODULE>
  ha   155 <MODULE>       ::=   <DCL DEST>
  h    156 <MODULE>       ::=   <INSTRUCTIO>
  h    157 <MODULE>       ::=   <FIX CONST>
  h    223 <MODULE>       ::=   <MODULE>  *   <MODULE>
  ha   158 <DCL DEST>     ::=   <STORAGE>
  ha   159 <DCL DEST>     ::=   <REGISTER>
  ha   160 <DCL DEST>     ::=   <STACK>
  h    161 <DCL DEST>     ::=   <DISTRIBUTOR
```

```
h  162 <DCL DEST>     ::=   <A-OPERATOR>
h  163 <DCL DEST>     ::=   <B-OPERATOR>
h  164 <DCL DEST>     ::=   <C-OPERATOR>
h  219 <DCL DEST>     ::=   N
h  220 <DCL DEST>     ::=   N   "NAME"
h  165 <A-OPERATOR>   ::=   A
h  166 <A-OPERATOR>   ::=   A   "NAME"
h  167 <B-OPERATOR>   ::=   B
h  168 <B-OPERATOR>   ::=   B   "NAME"
h  169 <C-OPERATOR>   ::=   C
h  181 <C-OPERATOR>   ::=   C   "NAME"
h  182 <STORAGE>      ::=   S
h  183 <STORAGE>      ::=   S   "NAME"
 a 170 <ASSIGN AX>    ::=   <ASSIGN LIS> ENDASSIGN
 a 171 <ASSIGN LIS>   ::=   <ASSIGN ELM> ;
 a 172 <ASSIGN LIS>   ::=   <ASSIGN LIS> <ASSIGN ELM> ;
 a 173 <ASSIGN ELM>   ::=   <REC ASSIGN>
 a 174 <ASSIGN ELM>   ::=   <IDF ASSIGN>
 a 175 <ASSIGN ELM>   ::=   <INI ASSIGN>
 a 176 <ASSIGN ELM>   ::=   <MAP ASSIGN>
 a 177 <REC ASSIGN>   ::=   .   "ATTRB-ID"  :  =  .   "ATTRB-ID"
 a 178 <REC ASSIGN>   ::=   <REC ASSIGN> .   "ATTRB-ID"
 a 207 <IDF ASSIGN>   ::=   <IDF LIST>  :=   <CONSTANT>
 a 208 <IDF LIST>     ::=   <IDENTIFIER>
 a 209 <IDF LIST>     ::=   <ARRAY IDEN>
 a 210 <IDF LIST>     ::=   <IDF LIST>  ,  <IDENTIFIER>
 a 211 <IDF LIST>     ::=   <IDF LIST>  , <ARRAY IDEN>
 a 212 <MAP ASSIGN>   ::=   <MODULE>    :=   <IDF LIST>
 a 213 <INI ASSIGN>   ::=   <MODULES>   :=   <CONSTANT>
 a 214 <MODULES>      ::=   <MODULE>
 a 215 <MODULES>      ::=   <MODULES>   ,    <MODULE>

   Note : Rules > 205 not yet implemented (5/4/79)
```

APPENDIX C

Additional Rules Part 1

These rules are necessary for a correct hardware operation.

1.1 In one <esb> the contents of **a storage** cell may change by one assignment (as a destination) or by one function only. A function parallel to an assignment is allowed, if it doe: not alter the contents of the cell.

1.2 GOTO, CALL, RETURN, DO, OD alter the contents of the program counter RP. Rule 1.1 must be obeyed.

1.3 In conditional statements rule 1.1 must be obeyed in all statements, which depend on nondisjunctive conditions.

1.4 Distributors must be defined before they are used.

Additional Rules Part 2

These rules are necessary for meaningful programs.

2.1 No arithmetic type transformations or checks are made in arithmetic expressions.

2.2 The operands (conditions) after IF, WHILE must be of the type boolean (1 bit).

2.3 Loops must be complete. Minimal loops must contain:

FOR DO OD or

WHILE DO OD

2.4 The identifier in D (<identifier>) must be the control variable of the current or of one outer FOR - loop.

2.5 The FOR - loop, which is referenced by D (<unsigned integer >), must exist.

2.6 The statements SUB and RETURN may be used in subroutines only..

2.7 The number of <operand> 's (parameters) in corresponding CALL and SUB statements must be equal,

2.8 " " include comments.

APPENDIX D

Control Language Commands

ADDCONNECTION
ADDMODUL
ASSIGN

BATCH
BLOCK
BLOCKEND
BREAK

CODE
CON
CORRECT
CURHSTAC

DEBUG
DEBUGOFF
DECLARE
DELCONNECTION
DELMODUL
DISPLAY

ENDDECLARE
ESB
EX
EXIT

FACTOR
FACTOR2
FACTOR3
FIRSTPORT

GO

INSERT

JOINT

K$\left\{\text{EY}\right\}\begin{matrix}1\\0\end{matrix}$

LIST

MACRO
MINHLP

NFULBUF
NODISPLAY
NOFACTOR
NOLOWERLETTERS
NOTRACE
NOWAIT

ONLY
ONLYRMAC

PF
PH
PROGRAM
PRINTFUNCTIONS
PRINTHARDWARE
PRINTSTACK

RECMACRO
RESET
RESUME

TF
TH
TM
TRACE
TYPEFUNCTIONS
TYPEHARDWARE
TYPEMODUL
TYPESTACK

USING

WAIT
WITH

(special character
before end-of-line)

APPENDIX E

```
***GROUP : <group name>
 <module name> (<address range>)MOREPORT :<moreport counter> DUPLICATABLE :<duplic.counter> FREQ:<absol.frequency>=<rel.freq.>%
 PORTUSINGS , OUTPUT : (<n>:<weighted number of esb's using <n> output ports of this module (0≤n≤9)>}_0^9
 PORTUSINGS , INPUT  : (<n>:<weighted number of esb's using <n> input ports of this module (0≤n≤9)>}_0^9
  <direction><port name>(<address range>).AUTO(<automatic function>).STATUS(<coded stat. & fcts >)FREQ:<abs.freq.>=<rel.freq.>%
   <input name><field name> ADR        D-BITS       MODULE           PORT        S-BITS      FREQ  {<output field frequency>}_0^1
                      <mpx address> <dest.bits>  <source module> <source port> <source bits> <abs.freq.> = <rel.freq.>%
                      <mpx address> <dest.bits>  <source module> <source port> <source bits> <abs.freq.> = <rel.freq.>%
                            .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
              <field name> ADR        D-BITS       MODULE           PORT        S-BITS      FREQ
                      <mpx address> <dest.bits>  <source module> <source port> <source bits> <abs.freq.> = <rel.freq.>%
                            .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
   <input name><field name> ADR        D-BITS       MODULE           PORT        S-BITS      FREQ
                      <mpx address>  .    .    .    .    .    .
  <direction><port name>(<address range>)  .    .    .    .    .
                            .    .    .    .    .    .    .    .
 <module name> (<address range>) .    .    .    .    .    .    .
                                 .    .    .    .    .    .    .
***GROUP : <group name>
   .    .    .    .    .    .    .    .    .    .    .    .    .
<listing of joint distribution of module and storage port uses>           | only if JOINT-option
<listing of joint distribution of microinstruction field uses>            | is true
<average and standard deviation of the number of used microinstruction fields>
<number of enable bits> <number of function select bits> <number of multiplexer select bits>
<number of multiplexer inputs> <number of connections>
```

APPENDIX F

Symbol Table

1	IF	2	FI	3	DO
4	OD	5	BY	6	TO
7	OF	8	FOR	9	END
10	SUB	11	THEN	12	ELSE
13	FROM	14	GOTO	15	CALL
16	STOP	17	CASE	18	ESAC
19	BEGIN	20	WHILE	21	RETURN
22	ENDSUB	23	ENDMACRO	24	ENDASSIGN
25	SUBROUTINE	26		27	"unsigned integer"
28	"attribute identifier"	29	"string expression"	30	"left"
31	"right"	32	"function identifier"	33	"lower letter string"
34	"name"	35		36	
37	"macro delimiter"	38		39	"macroparameter"
40	(	41	)	42	*
43	+	44	,	45	-
46	.	47	/	48	
58	:	59	;	60	<
61	=	62	>	63	
64		65	A	66	B
67	C	68	D	69	E
70	F	71	G	72	H
73	I	74	J	75	K
76	L	77	M	78	N
79	O	80	P	81	Q
82	R	83	S	84	T
85	U	86	V	87	W
88	X	89	Y	90	Z
91	[	92	\	93	]
94	^	95	_		
100	<program>	101	<program head>	102	<program body>
103	<elementary stmt.block>	104	<while do statement>	105	<subroutine>
106	<subroutine head>	107	<statement block>	108	<statement>
109	<call statement>	110	<subroutine enter>	111	<conditional stmt.block>
112	<destination>	113	<operand>	114	<source>
115	<label>	116	<storage>	117	<register>
118	<stack>	119	<instruction>	120	<distributor>

121 <fixed constant>	122 <control variable>	123 <a-operator>
124 <b-operator>	125 <c-operator>	126 <network>
127 <function>	128 <function instruc.part>	129 <identifier>
130 <array identifier>	131 <array identifier head>	132 <attribute>
133 <attribute head>	134 <constant>	135 <fixed point constant>
136 <signed integer>	137 <sign>	138 <addmodule axiom>
139 <addconnection axiom>	140 <connection body>	141 <connection>
142 <declaration destinat.>	143 <addmodule body>	144 <module>
145 <condit.operand head>	146	147
148	149	150
158 <macro call head>	159 <macro declarat.axiom>	160 <assignment axiom>
161 <assignment list>	162 <assignment element>	163 <record assignment>
164 <identifier assignm.>	165 <map assignment>	166 <initial value assignm.>
167 <case statement>	168 <case head>	169 <case tail>
170 <case line>	171 <case list>	

APPENDIX G

Error Messages

MSS number	MSS error text	MSS parameter output	remarks
a) Warnings			
Ø	unknown connection deleted	destination module	
1	double assignment to attribute		ignore after $ RESET commands!
2	elsepart unimplemented		change cond.oper.to cond.statement!
b) Errors			
1	rulenumber too large	rulenumber of RULES-file	incorrect file RULES ?
2	SORTMAX too small	SORTMAX+1,..+2,..	increment SORTMAX in MSS source!
3	syntaxrule too long	number of right symbols	>8 right sides in RULES-file
4	no matching rule of syntax	stack dump	this esb is ignored
5	near heap overflow	procedure name,heap addr	program is too complicated
6	EOF found, syntax error	Ø if INP-, 1 if LIB-file	
7	unexpected connection error	esb's subnumber	internal error
8	no such !"name" in library	"name"	
9	conflict at destination	module, number of use	error in algorithm
12	only one OD permitted		conflict at program counter
13	too many DO's		
14	more OD's than DO's		
15	less OD's than DO's		
16	less DO's than FOR's		
17	RETURN not in subroutine		possibly previous syntax error
18	SUB not in subroutine		" - " - " - "
19	>1 networks in esb		error in old MSS versions
20	V defined too often		definition does not depend on condit.
21	V not yet defined		definition of V must precede it's use
22	>10 V's not implemented		change VMAX in MSS source
23	multiple definition of label	label	

MSS number	MSS error text	MSS parameter output	remarks
24	no such connection found	source module	error from $ DELCONNECTION
27	can't find/create function	function identifier	addition to fct. list inhibited
29	warning: character ignored	character and it's code	
30	unexpected exceeding of inputmax	destination module	will disappear in future versions
31	invalid monitor keyword	keyword	stops MSS if read from INP-file
32	exceeding stackmax		expression too complicated ?
33	multiplexer is full	destination module	
34	no portnames or directions		(for a label)
35	MPX attribute not implemented		
36	new attribute not accepted		increm. MGAMAX in MSS source
37	inverted bits not implemented		LSB = Ø !
38	number(s) not accepted		(for this attribute)
39	number(s) expected		" - " - "
40	can't find or create field		.MOREFIELD limitation
41	can't find or create module	module	module of this group is declared!
42	MOREPORT<=Ø, no new port	next portname, old use	will stimulate compiler
43	too many error statements		only in older MSS versions
44	illegal module/port/direction		
45	declaration attrib. not in dcl	attribute-identifier	
46	too many connections/esb	number of connections	increm. CONNMAX in MSS source
47	portdirection is incorrect		
49	exceeding CLMAX		too many case lines
50	destination already used	dest.module, old use #	will stimulate compiler
51	error in compiler	procedure name	internal error
52	not a function of port	function identifier	
54	undefined label:	label	

MSS number	MSS error text	MSS parameter output	remark
55	unknown module:	module	
57	exceeding ESBMAX		too many generated esb's
58	non-reparable connection err.		compiler cannot repair esb
59	impossible USING (ignored)		
60	exceeding HLPMAX		
61	can't find or create port	portname	name not allowed
62	mixing with undefined attrib.		(or illegal bit sequence)
63	connect ignored, old error	source module	caused by previous error
67	uncorrected connection error		GENERATE-option is needed
68	macro param.: defin.op.,use ds		operand param. used as destination
70	left side = macro parameter		
71	different explication of param		
72	no definition of macro param.		
73	&name not in macro declaration		
77	a new attribute is expected		