F A N L A N G U I D E

A N D R E F E R E N C E M A N U A L

FANCY Inc.
(S.G. van der Meulen
A.A. Brouwer, editors)

RUU-CS-82-3
voorjaar 1982

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

F A N L A N G U I D E

A N D R E F E R E N C E M A N U A L

FANCY Inc.
(S.G. van der Meulen
A.A. Brouwer, editors)

Technical Report RUU-CS-82-3
voorjaar 1982

Tell me, where is FANCY bred,
Or in the heart, or in the head?
How begot, how nourishēd?
(Shakespeare, Merchant of Venice)

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

F A N L A N G U I D E
A N D R E F E R E N C E M A N U A L

Utrecht, april 1982

PREFACE

FANLAN and the FANCY processor originated from an introductory course on Computer Architecture. Not having our own computer available for teaching purposes of this kind, we invented one that should also be appropriate for compiler- and operating-system courses. We named this machine "FANCY" for obvious reasons, and its assembler language "FANLAN". A preliminary report on FANLAN was issued for limited circulation in spring 1981.

The main characteristics of our (simpler) FANCY were present in the MOTOROLA-68000, which we adopted (and adapted) as a more realistic FANCY machine. From the design of a FANLAN, also suitable as a language for programming the MOTOROLA-68000, emerged the idea to aim at a 'machine-independent' assembler.

In FANLAN as it stands now, the machine-independent features (such as block-structure, conditional assembly and macro-processing) are carefully separated from the inevitable machine-bound aspects. This, together with a new approach to the use and meaning of identifiers, and the expression of the various addressing modes, makes FANLAN an assembler-language in its own rights, highly independent of the concrete underlying machine (still the MOROROLA-68000 in this publication).

This Report can be considered as a (semi-formal) defining document for the language. Anticipating the production of the here described assembler, the Report has the status of a draft report. Small changes and improvements to syntax and semantics can be expected, as also the clarification of certain novel feautures.

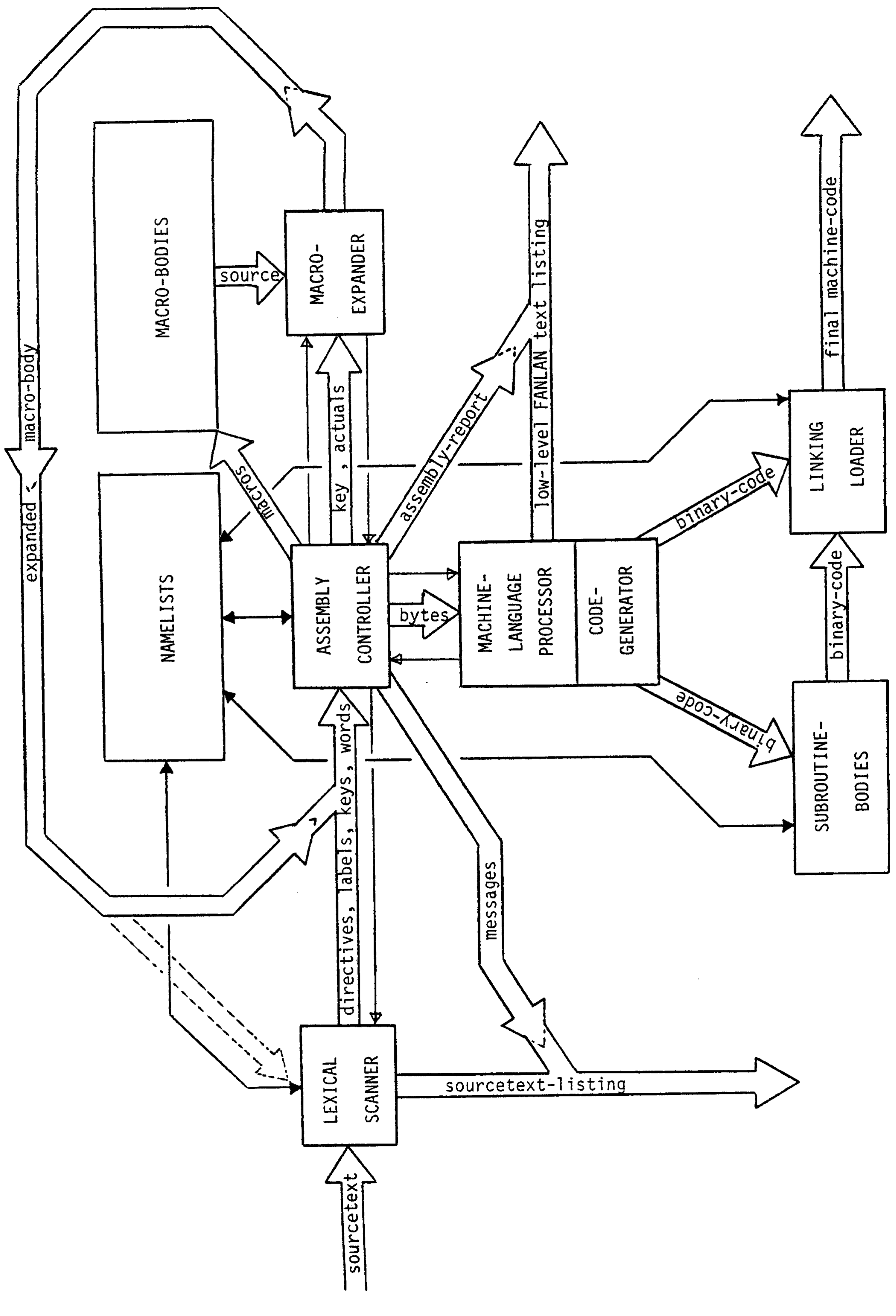
The "FANCY Inc." on the cover- and title page consists of (in order of their participation) the following persons (colleagues and students):

S.G. van der Meulen
A.P.W. Böhm
R. Gerth
J.H. Geels
H. Jensen
A.A. Brouwer

S.G. van der Meulen

G L O B A L   F A N L A N   S T R U C T U R E

1.0 THE ASSEMBLER



MACRO-BODIES
MACRO-EXPANDER
source
NAMELISTS
ASSEMBLY CONTROLLER
MACHINE-LANGUAGE PROCESSOR
CODE-GENERATOR
LINKING LOADER
SUBROUTINE-BODIES
LEXICAL SCANNER
expanded macro-body
macros
key , actuals
assembly-report
low-level FANLAN text listing
final machine-code
binary-code
binary-code
binary-code
bytes
directives, labels, keys, words
messages
sourcetext-listing
sourcetext

A FANLAN ASSEMBLER, as presupposed in this report, consists conceptually of six processing modules:

LEXICAL SCANNER ASSEMBLY CONTROLLER MACRO PROCESSOR	machine-independent	MACHINE-LANGUAGE PROCESSOR CODE-GENERATOR LINKING LOADER	machine-dependent

and three (partly permanent) information-stores:

NAMELISTS	partly machine-dependent (the machine-language keys)
MACRO-BODIES	partly machine-independent
SUBROUTINE-BODIES	machine-dependent

We give a brief functional description of these nine components which should be understood in relation to the sections on syntax and semantics in (mainly) this chapter. Unless otherwise stated, numbers refer to subsections.

LEXICAL SCANNER:

- provides for the sourcetext-listing, adding logical-line numbers and inserting messages from the ASSEMBLY CONTROLLER at the right place;
- compresses the sourcetext, eliminating superfluous blanks and all comments;
- recognizes directives;
- builds up the NAMELISTS passing unique namelist-entries to the ASSEMBLER-CONTROLLER instead of identifiers (labels, keys, names, operands etc.).

ASSEMBLY CONTROLLER:

- governs the overall assemblage process as steered by the assembler-directives and assembly-control (see 1,4,19,20);
- establishes the NAMELISTS store, following the blockstructure directives (see, 1,4);
- sends macro-body's to the MACRO-BODIES store, for later expansion;
- sends macro-insertion's to the MACRO-PROCESSOR;
- sends byte's to the MACHINE-LANGUAGE PROCESSOR;
- interweaves the assembly-report (an implementation-dependent feature) with the (possibly optional) low-level FANLAN text listing (another implementation dependent feature);
- sends message's to be weaved into the low-level FANLAN text listing.

MACRO-PROCESSOR:

- derives not-parametrized sourcetext (see 16) from the given macrobody(s) (designated by the given key) as modified by the given actuals, passing this new sourcetext back to the ASSEMBLY CONTROLLER (possibly again through the LEXICAL SCANNER) for further processing;
- can be activated in several lexical- and dynamic depths, including recursive incarnations - it is tacitly assumed that the interrelation between the ASSEMBLY CONTROLLER and the MACRO-PROCESSOR is such that all possible layers of activation will be well distinguished, including their scope-requirements.

MACHINE-LANGUAGE PROCESSOR:

- determines from the opcode/size or type/size of the given 'bytes', and the given operands and the condition-option, the binary components of machine-code for the given concrete machine;
- attaches the thus formed machine-words to the proper namelist-chain(s), in case of forward-references;
- provides (if required) for an adequate low-level FANLAN text listing;
- the CODE-GENERATOR belongs to the MACHINE-LANGUAGE PROCESSOR and supplies the prefabricated bitpatterns for the final machine-words.

LINKING LOADER:

- derives the final machine-code from the binary-code as obtained from the MACHINE-LANGUAGE PROCESSOR, by modifying (if necessary) operand-addresses for their final allocation - the precise function of the LINKING LOADER is implementation-dependent and may also heavily depend on the concrete machine addressing regime.

NAMELISTS:

- contains a table of assembler-directives:
  machine independent;
- contains a table of machine-language keys:
  machine-dependent;
- contains a table of standard subroutine-names:
  machine-independent;
- contains a table of standard macro-names:
  machine-independent;
- contains table(s) of subroutine- and macro-names which may be
  machine-dependent;
- contains stacks and chains for program-defined labels (operands), subroutine- and macro-names, and assemblytime-values:
  these stacks, chains and trees reflect the FANLAN block-structure, and all local entries disappear at their time - some entries (names of standard callable-blocks) may survive an assemblage and will then be subjoined to one of the tables above.

MACRO-BODIES:

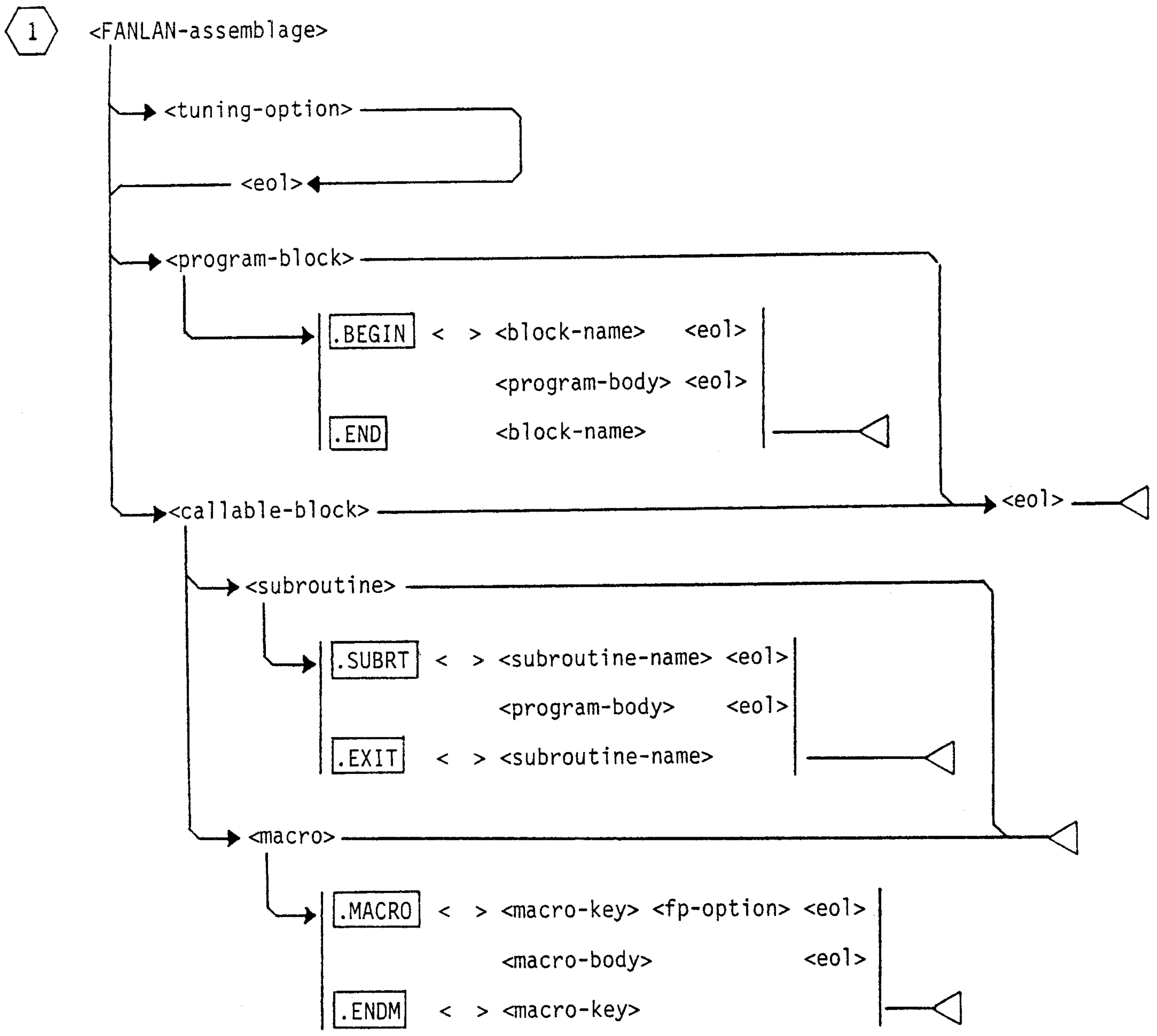
- contains standard-macrobody's, some of which may be machine-independent;
- contains program-defined macro-body's which disappear at their time - some of them, however, may survive an assemblage (if presented as a standard callable-block).

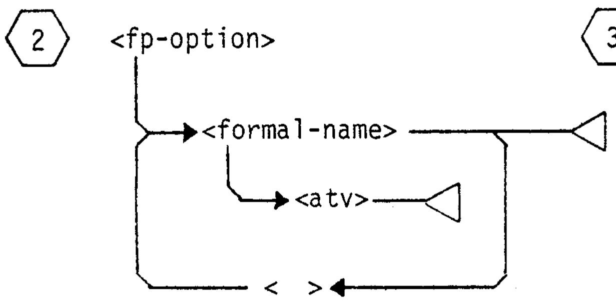
SUBROUTINES:

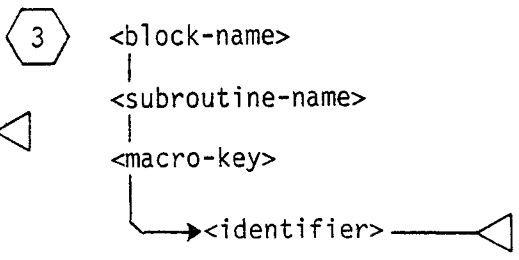
- contains the binary-code of standard-subroutines;
- contains the binary-code of program-defined subroutines which disappear at their time - some of them, however, may survive an assemblage (if presented as a standard callable-block).

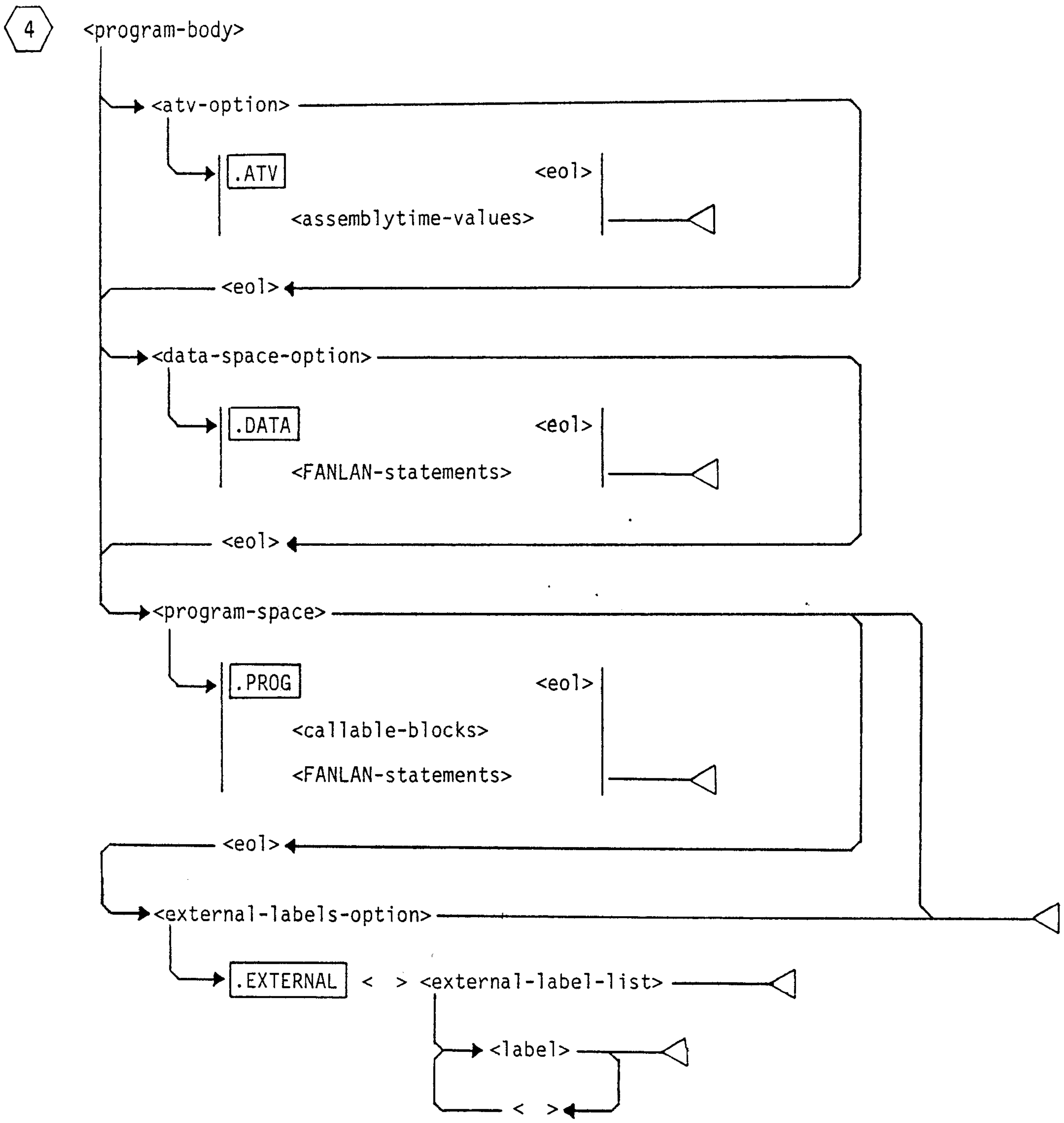
G L O B A L   F A N L A N   S T R U C T U R E

1.1 SYNTAX

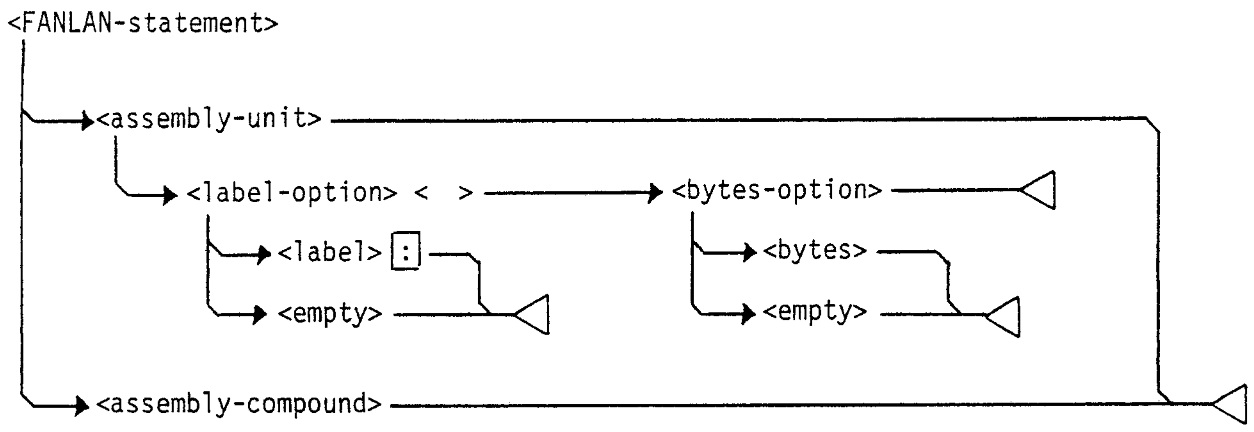

1
<FANLAN-assemblage>
<tuning-option>
<eol>
<program-block>
.BEGIN
< >
<block-name>
<eol>
<program-body>
<eol>
.END
<block-name>
<callable-block>
<eol>
<subroutine>
.SUBRT
< >
<subroutine-name>
<eol>
<program-body>
<eol>
.EXIT
< >
<subroutine-name>
<macro>
.MACRO
< >
<macro-key>
<fp-option>
<eol>
<macro-body>
<eol>
.ENDM
< >
<macro-key>


2
<fp-option>
<formal-name>
<atv>
< >

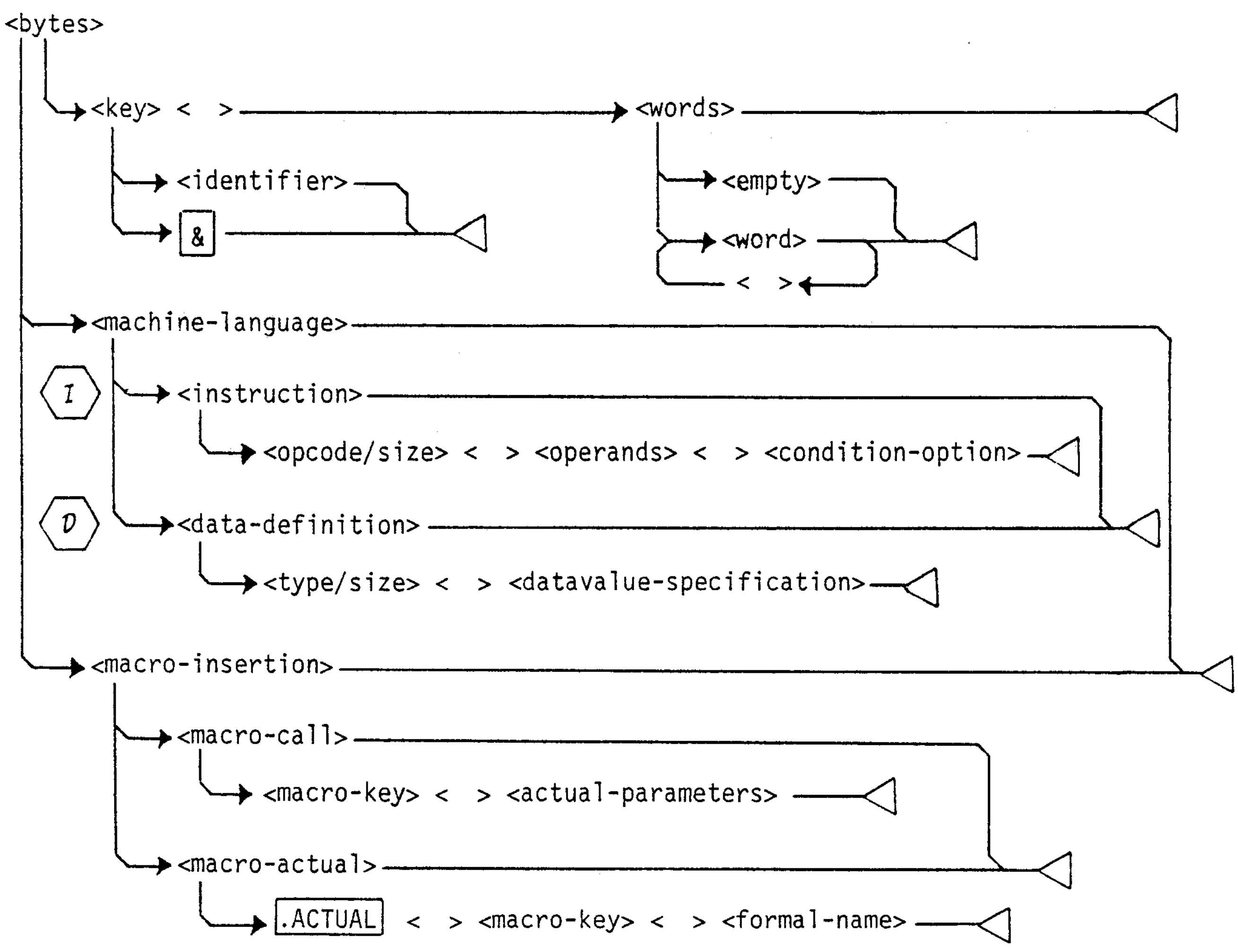

3
<block-name>
<subroutine-name>
<macro-key>
<identifier>


4
<program-body>
<atv-option>
.ATV
<eol>
<assemblytime-values>
<eol>
<data-space-option>
.DATA
<eol>
<FANLAN-statements>
<eol>
<program-space>
.PROG
<eol>
<callable-blocks>
<FANLAN-statements>
<eol>
<external-labels-option>
.EXTERNAL
< >
<external-label-list>
<label>
< >

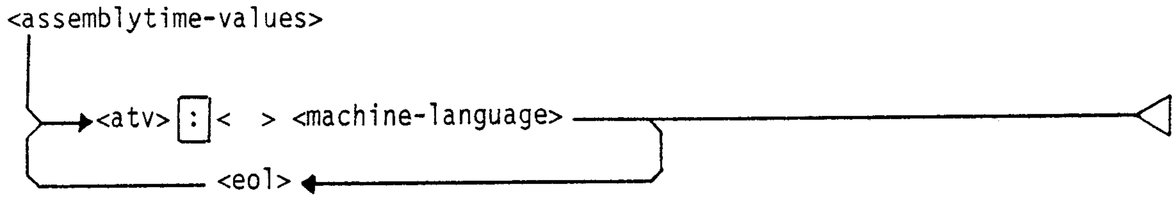
5



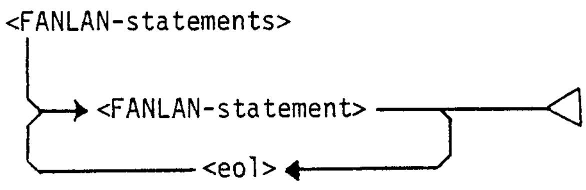
6



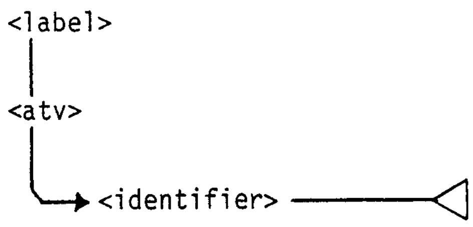
7



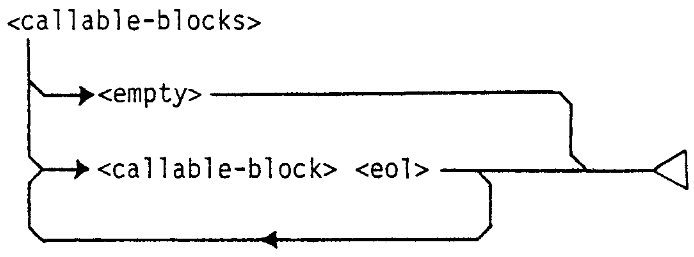
8



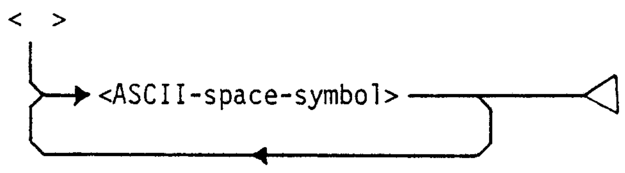
9



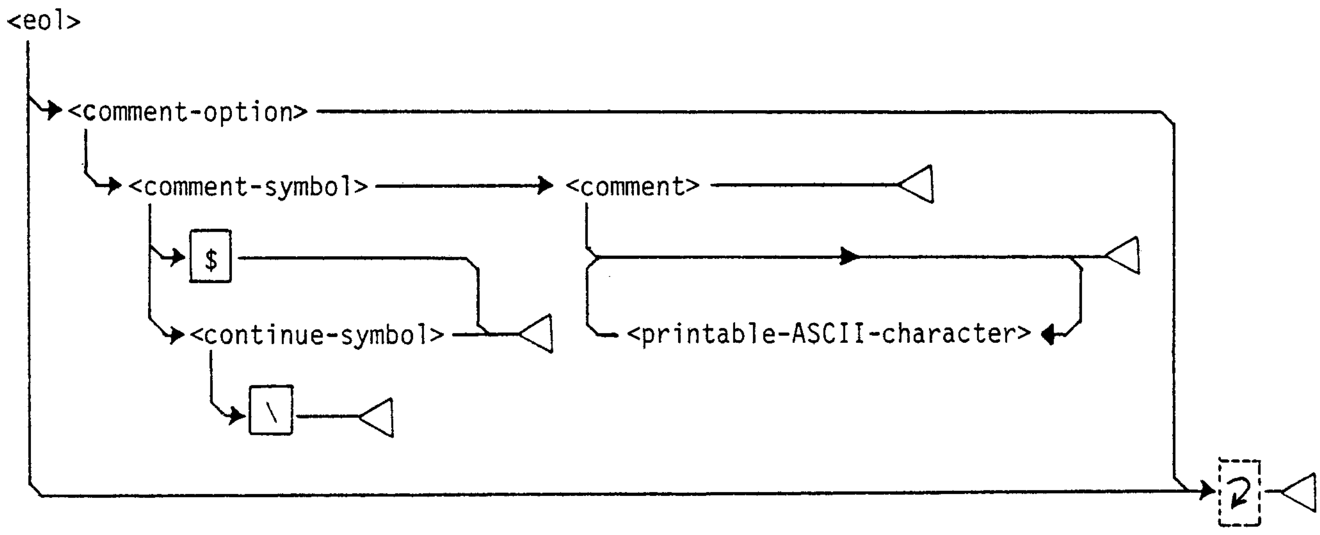
10



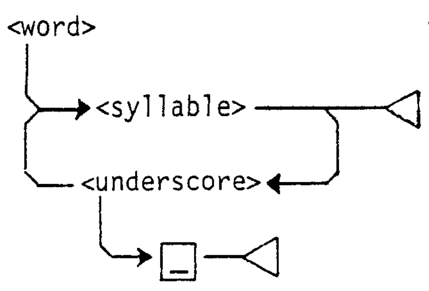
11



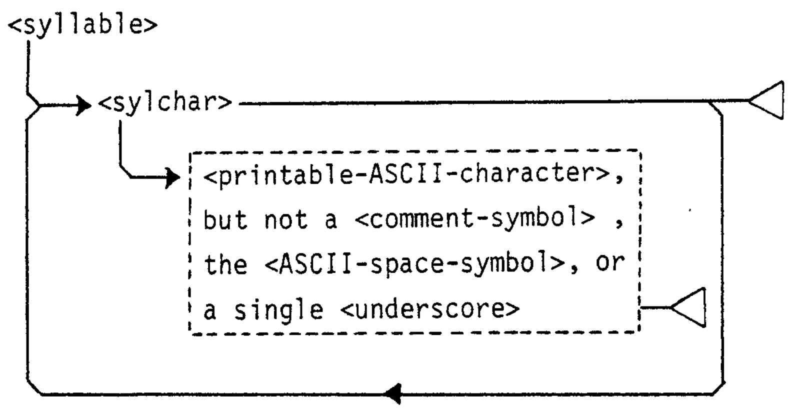
12



13

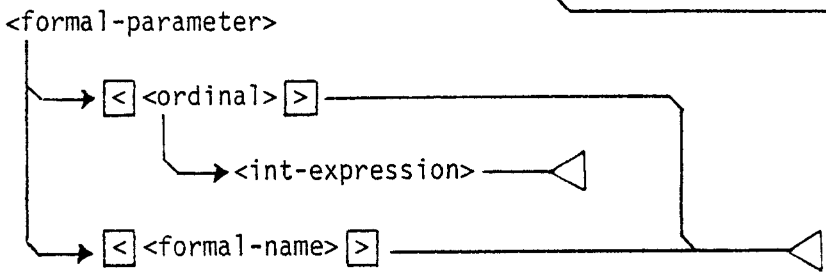


14

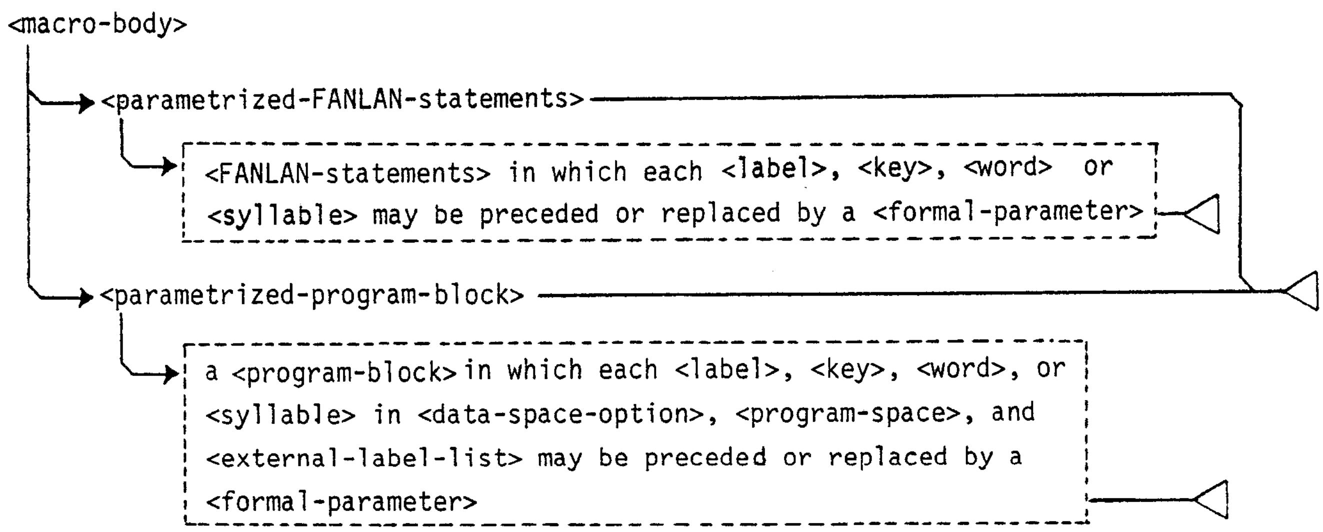


<syllable>
<sylchar>
<printable-ASCII-character>, but not a <comment-symbol> , the <ASCII-space-symbol>, or a single <underscore>

15

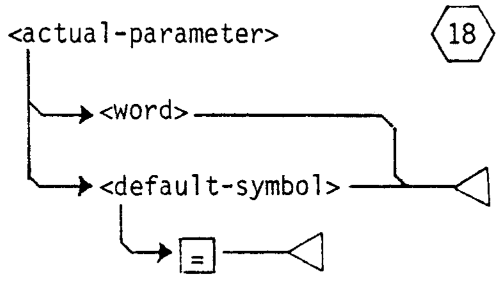


16

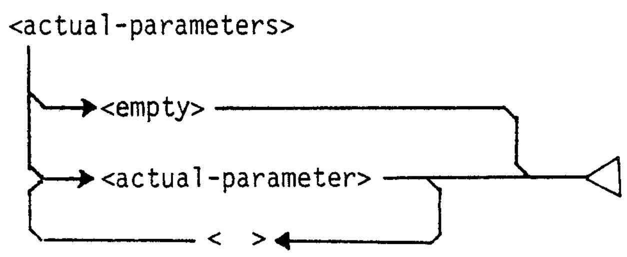


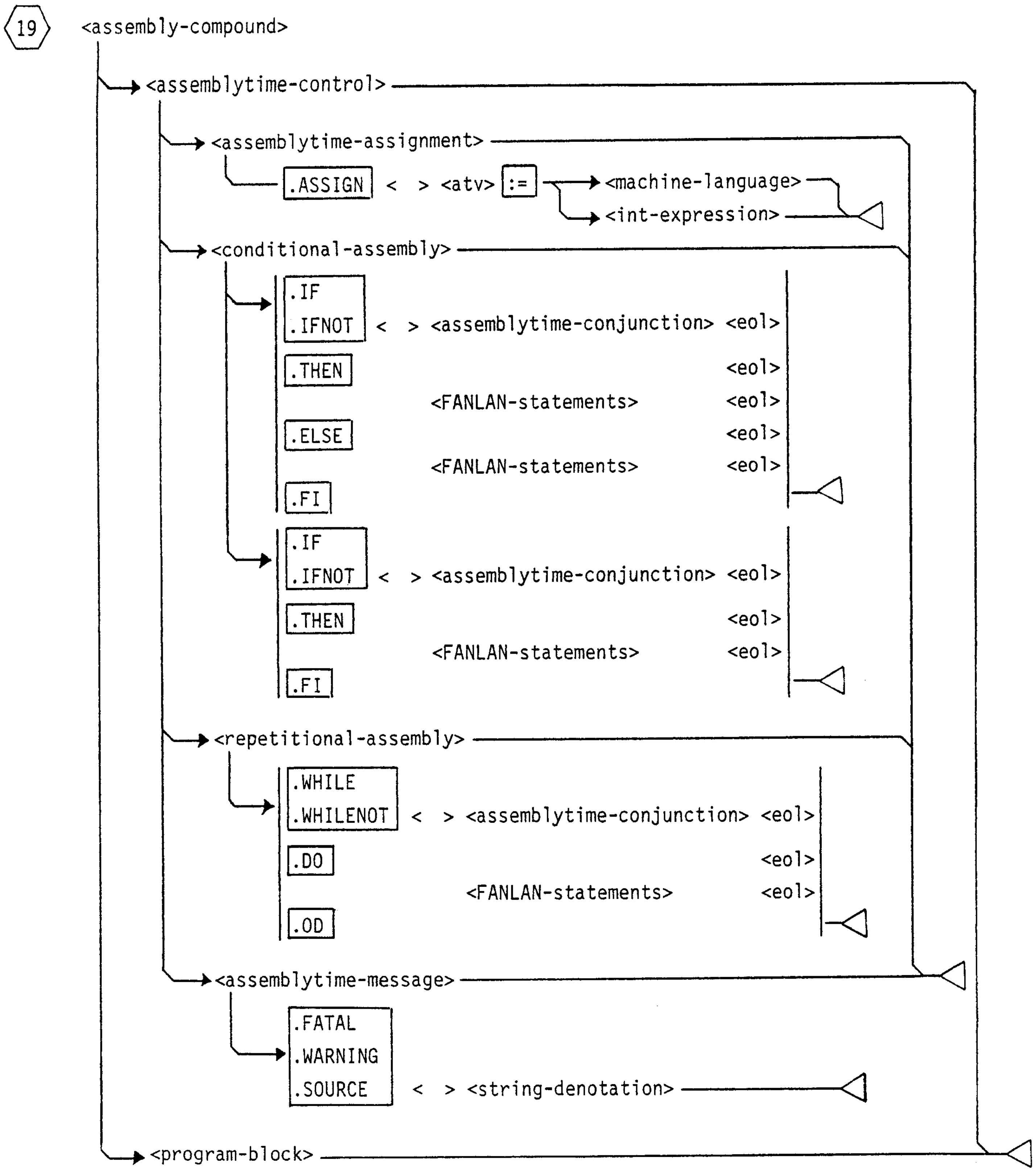
<macro-body>
<parametrized-FANLAN-statements>
<FANLAN-statements> in which each <label>, <key>, <word> or <syllable> may be preceded or replaced by a <formal-parameter>
<parametrized-program-block>
a <program-block> in which each <label>, <key>, <word>, or <syllable> in <data-space-option>, <program-space>, and <external-label-list> may be preceded or replaced by a <formal-parameter>

17

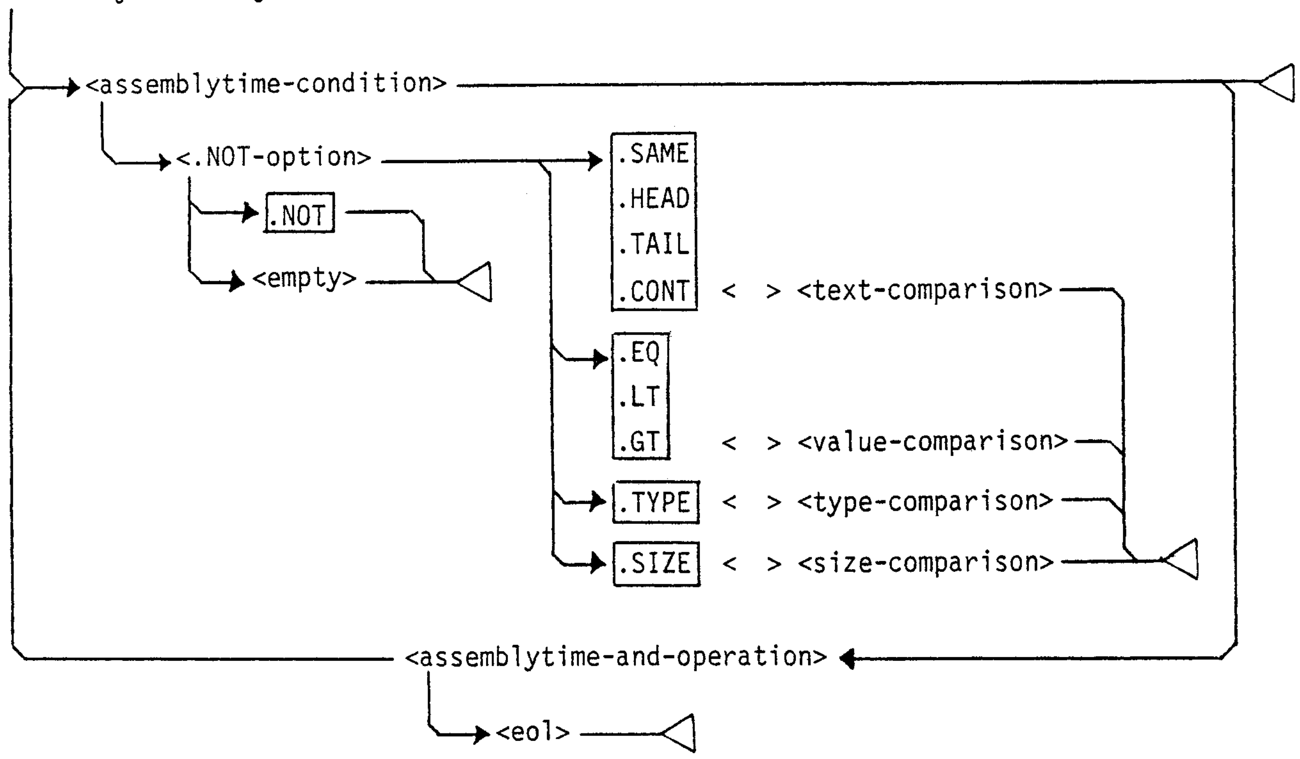


18

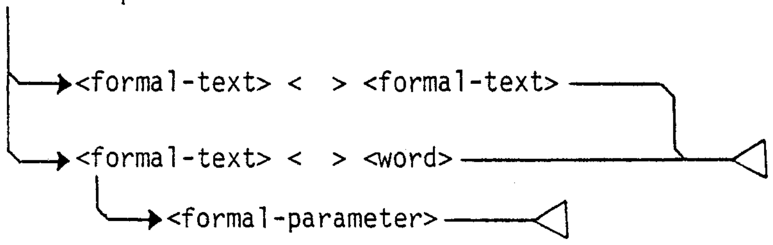



19
<assembly-compound>
<assemblytime-control>
<assemblytime-assignment>
.ASSIGN
< > <atv>
:=
<machine-language>
<int-expression>
<conditional-assembly>
.IF
.IFNOT
< > <assemblytime-conjunction> <eol>
.THEN
<eol>
<FANLAN-statements> <eol>
.ELSE
<eol>
<FANLAN-statements> <eol>
.FI
.IF
.IFNOT
< > <assemblytime-conjunction> <eol>
.THEN
<eol>
<FANLAN-statements> <eol>
.FI
<repetitional-assembly>
.WHILE
.WHILENOT
< > <assemblytime-conjunction> <eol>
.DO
<eol>
<FANLAN-statements> <eol>
.OD
<assemblytime-message>
.FATAL
.WARNING
.SOURCE
< > <string-denotation>
<program-block>

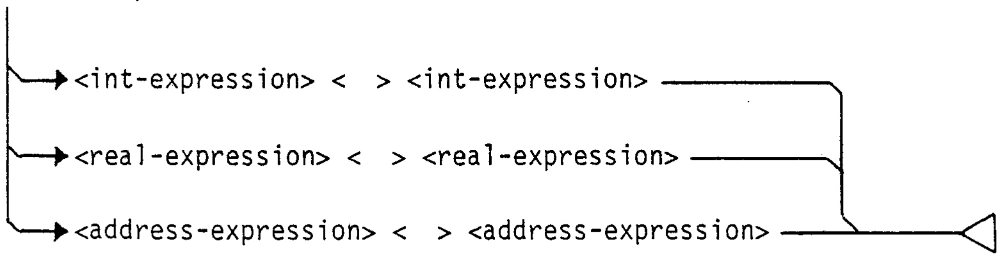
20 <assemblytime-conjunction>

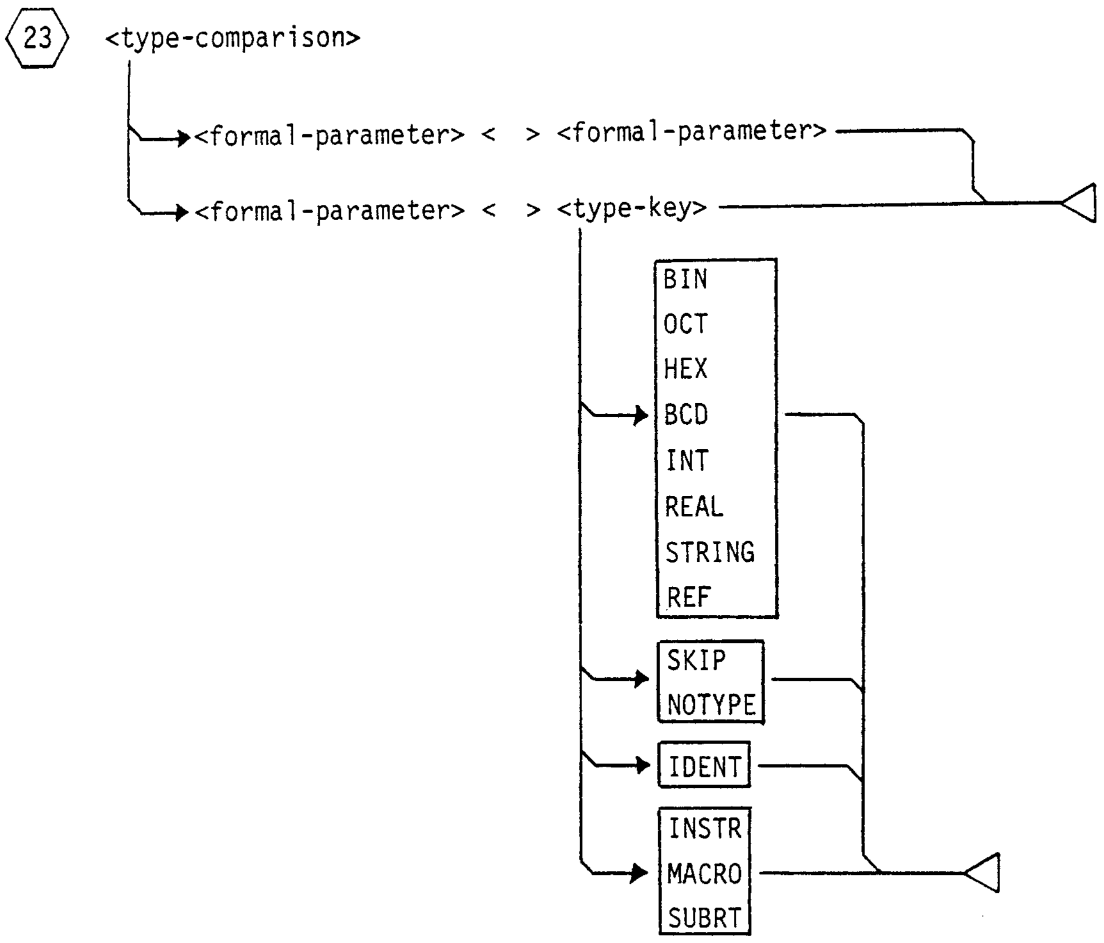


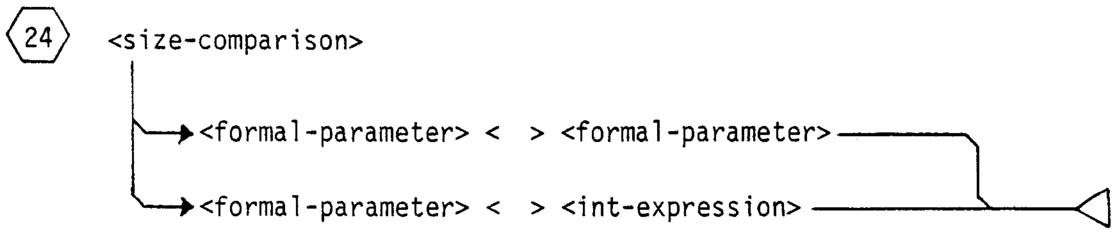
21 <text-comparison>



22 <value-comparison>




23
<type-comparison>
<formal-parameter> < > <formal-parameter>
<formal-parameter> < > <type-key>
BIN
OCT
HEX
BCD
INT
REAL
STRING
REF
SKIP
NOTYPE
IDENT
INSTR
MACRO
SUBRT


24
<size-comparison>
<formal-parameter> < > <formal-parameter>
<formal-parameter> < > <int-expression>

G L O B A L   F A N L A N   S T R U C T U R E

1.2 SEMANTICS

1

The initial contents of the namelists, together with the textual information of the predefined macros and the binary code of the preassembled subroutines, determines the *standard environment* of a FANLAN-assembly. Normally, all standard-identifiers will be in capital-letters, and all program-defined identifiers in small letters. However, it must be possible to translate a FANLAN-program from a 64-ASCII-character keyboard, and therefore all different identifiers should remain different if small letters are read as capitals, or vice versa.

A FANLAN-assemblage may begin with a tuning-option (such as temporary renaming of standard identifiers and mnemonics, particular selection from available files etc.), if desired. The precise syntax and semantics of this optional feature is implementation dependent.

All standard-identifiers, immediately preceded by a period - like ".BEGIN", ".END" etcetera -, are *assembler-directives*: they steer the ASSEMBLER CONTROLLER and should not be confused with identifiers proper such as labels, keys, subroutine-names and macro-names.

The directives ".BEGIN" - ".END", ".SUBRT" - ".EXIT" and ".MACRO" - ".ENDM" demarcate *blocks of local identification*: the *scope* of an identifier defined as a label or subroutine- or macro name within a block, is *local* to that block (see, however, 4)

A program-block is a *FANLAN-program* in the usual sense of the word: after being translated successfully, it may be loaded and executed immediately - or stored in binary code for later loading and execution.

A callable-block, after successful translation, becomes part of the standard-environment (of the assembler itself, so to say): its code or text will be stored with the assembler, and its name or key becomes a standard-identifier.

A block-name and a subroutine-name may be conceived as a label just in front of the directive ".BEGIN" or ".SUBRT" respectively. A macro-key follows the scope of a subroutine-name, both are local to the block in which they are declared. A subroutine-name and a macro-key are always distinguishable by their use, and may therefore have the same identifier (which is a useful feature for subroutine-calling macros).

⟨1 continued

The ".END" directive, closing a program-block, generates (if necessary) an unconditional "GOTO" to the (first instruction of the) textually following program-block, thus linking program-spaces (see 4) and separating them from data-spaces.

The ".EXIT" directive, closing a subroutine, generates an unconditional "RETURN" (see chapter 2, table 9) in order to prevent a not intended 'running out' a subroutine.

A macro, essentially, will be stored as (compressed) sourcetext (possibly in some intermediate code), to be expanded through a macro-insertion (see 6) in which actual-parameters can be provided to replace the corresponding formal-parameters (see 13--18).

⟨2⟩

In the fp-option the programmer may define names (identifiers) for the formal-parameters of the macro - these identifiers have the status of an 'atv' (see 4,7). The formal-parameter regime of FANLAN is such that this formal-parameter naming is optional.

⟨3⟩

See 1

⟨4⟩

The minimal constituent of a program-body is is the program-space in which the whole gamut of machine-language and macro-insertion (see 6) can be specified.

In the atv-option the programmer can specify assemblytime-values for assembly-control (see 19) and immediate-operands (see chapters 2 and 3). If formal-names are used in a macro definition (see 2: fp-option), then they are treated as (implicitly declared) atv's.

The data-space-option provides for the structuring (and naming) of (alterable) data-space, which may be allocated by the assembler in complete separation from program-space. The underlying concrete machine may have (hardware-)provisions which prevent the execution of data-space bytes or use program-space bytes as 'alterable data'. The FANLAN assembler will at least give warnings for such not-normal memory-accesses.

⟨4 continued

The name of the block to which the program-body belongs is supposed to act as a label in front of the first FANLAN-statement in program-space, following the possible (local) callable-blocks. The scope of this block-name-'label' is that of the immediately surrounding program-block.

There is an essential semantic difference between assemblytime-values in the atv-option, runtime variables (FANLAN-statements) in the data-space-option, and instructions (FANLAN-statements) in program-space:
- assemblytime-values are known and exist only at assemblytime, they have disappeared completely at runtime (though some of them may be hidden somewhere as an immediate operand in an instruction),
- runtime variables are known and exist at runtime, their value is irrelevant at assemblytime; they can be used as destination-operands (alterable data),
- instructions and data in program-space are known and exist only at runtime; they may be used as (non-alterable) operands - the instructions proper are meant to be executed.

Assemblytime-values and runtime-variables have a type and a size which may play a role at assemblytime, labels in program-space have (in principle) no type- or size-attribute.

A label occurring in the external-label-list (following the directive ".EXTERNAL") becomes thereby global in the entire FANLAN-assemblage. Its addressing mode cannot be defined by the programmer (is implementation-dependent).

If, and only if, the assemblage is a callable-block ând a subroutine, the external-label becomes also a standard-identifier (in order to provide for the possibility of different entry-points of a standard-subroutine)

⟨5⟩

A FANLAN-statement is either an assembly-unit or an assembly-compound (through which the FANLAN block-structure can be nested and assembly-control can be specified).

An assembly-unit is, normally, what is written on a line in program- or data-space, if not preceded by a directive. If the label-option is empty, we have an unlabeled bytes; an empty bytes-option provides for placing more than one label in front of the same bytes; if both constituents of an assembly-unit happen to be empty, we have an empty line. An assembly-unit may be extended over several physical lines (see 12).

⟨6⟩

All bytes have the syntactic structure of a key followed by zero, one or more words (see 13). The "&" character renews (repeats) the lastly written key, as a convenience for lazy writers.

We distinguish two kinds of bytes:
- machine-language, whereby we address the concrete machine (see chapter 2, Syntax at table-index for "*I*" and "*D*");
- macro-insertions, whereby we address the MACRO PROCESSOR.

Bytes generate (directly or indirectly) machine-code ('bytes') to be loaded in random access memory and, ultimately, to be executed as instruction-code or to be manipulated as (alterable or non-alterable) data by the instructions.

The MACHINE-LANGUAGE PROCESSOR keeps and updates a bytes-counter (PLC or Program Location Counter) which represents, at assemblytime, the PC (runtime Program Counter). Only the memory address relative to the FANLAN assemblage zero-point (the physical address of the first assembled bytes) may be of occasional importance to the FANLAN programmer: the PLC always contains this effective address of the bytes to be assembled. Conceptually, the assembled code will be loaded with these effective addresses - the physical zero-point address will be loaded in a runtime base-address register (see chapter 2.0). It is, however, immaterial whether the concrete machine has a hardware runtime base-address register or not (in which case the loading and linking process becomes a bit more complicated).

FANLAN knows two kinds of macro-insertion:
- The macro-call is the normal, convential, way of expanding a particular macro: zero, one or more actual-parameters are provided to be substituted for the corresponding formal-parameters (see also 13--18).
- The macro-actual can only originate from a macro-body: it signals that at a certain position in the actual-parameters, a macro-key was encountered, and that this (or possibly another) macro must now be expanded with the already given actuals (!) - the zero-position is given in the formal-name provided by the macro-actual.

⟨7⟩

In the definition of assemblytime-values, only machine-language (and not a macro-insertion) can specify the value.

8

See 5

9

Both labels and atv's are denoted by identifiers. Being program-defined entities, their identifier should be written in small letters. There is one exception: an external label in a subroutine as standard callable-block must be written in capital letters (see also 4).

10

The (local) callable-blocks in the beginning of program-space are optional (i.e. the program-space may begin with the FANLAN-statements proper).

11

Blank space (at least one ASCII-space-symbol) separates label-declarations, keys and words. There is no semantic difference between one or more space-symbols.

12

The eol (end-of-line) consists of an ASCII-CR (Carriage Return) immediately followed by an ASCII-LF (Line Feed) - or vice versa - (in the syntax diagram denoted by the symbol "↵") possibly preceded by a comment.

The continue-symbol "\" acts as a special comment-symbol, indicating that the logical (sourcetext-)line is to be continued on the following physical line. With the aid of the continue-symbol, macro-insertions with many actual-parameters can be given a nice (i.e. 'functional') lay-out, and also long string-denotations can be written out. Note that "↵" is not a printable ASCII-character.

Any comment will be disregarded by the assembler.

All eol's have the side-effect of a blank (see 11). A <u>not</u> to be continued eol (i.e. an eol <u>not</u> introduced by "\"), moreover, separates FANLAN-statements - the continue-symbol annihilates precisely that important function.

We thus have:
- Blanks separate words on a logical line.
- Eol's separate logical lines (and a fortiori words), unless the continue-symbol is used (in which case the word-separating function remains).

13

A logical sourcetext-line (carrying assembly-information) consists of words separated by blanks (see also 11) - label-declarations and keys are also words. Any word can be split arbitrarily into syllables, separated by underscores "_". This word-splitting makes it possible to specify a syllable (an arbitrary part of a word) as actual-parameter in a macro-insertion, an to 'formalize' it in a macro-body (see also 13--18). Some assemblers, however, may forbid the splitting of labels and keys.

In order to maintain the possibility of writing underscores in a string-denotation, we represent the character "_" by an underscore-image, consisting of two consecutive underscores "__" (see also chapter 3).

Underscores in the syntactic position of a syllable-separator disappear completely from the sourcetext (or its intermediate code) when passed to the MACHINE-LANGUAGE PROCESSOR.

14

See 13.

15

A formal-parameter, essentially, is an ordinal number designating directly the actual-parameter (by its position). As an option, formal-names may be declared instead of, or even in addition to, the (implicit) formal ordinals (see also 2). The assembler will treat formal-names as supposititious assembly-time-values - their implicit initial value is their position number in the fp-option (see 2).

Note that the formal ordinal is defined as an int-expression and, sonsequently, can be altered at assembly time (though in many cases the int-expression will be as simple as an int-denotation).

16

In the beginning of program-space, one or more macro's may be defined. Consequently, a program-block (not yet a macro-body) may (in these inner blocks) already contain formal-parameters. A program-block as such, may be used as the body of a parameterless macro.

⟨16 continued

The difference between the two forms of macro-body (parametrized-FANLAN-statements and parametrized-program-block) is pure syntactical - the simpler form of the parametrized-FANLAN-statements may allow a simpler kind of storage of the macro-body.

FANLAN-statements or program-block may be transformed into a macro-body as described in the syntax. Note that FANLAN-statements or program-block as such may be used as the macro-body of a macro without parameters. Note also that neither an assembler-directive, nor a comment can be replaced by a formal-parameter.

A formal-parameter (easily identifiable from its syntactic structure) will be recognized as such only if it stands in front of ('against'), or in the place of, a label, key, word or syllable. Consequently, a formal-parameter always follows a blank or a (single) underscore and may be immediately followed by a label, key, word or syllable, or it may stand alone.

The label, key, word or syllable immediately following a formal-parameter (that is without a blank or a single underscore between the two), is hereby the default-value of that formal-parameter. A formal-parameter followed by a blank has an empty default-value.

A FANLAN-assembler may require a slightly more restricted syntax for a macro-body in which a formal-parameter must always have an acceptable (possibly empty) default-value. With this requirement the assembler can perform a syntactic check on the macro-body (as called without parameters) at its declaration.

Both forms of macro-body follow for their label-declarations and other names the scope-rules of a program-block (i.e. the directives ".MACRO" and ".ENDM" act as block-begin and block-end). However, if a label at its declaration is formalized and actualized at the macro-insertion, then that label has the scope of the calling environment. Labels in a parametrized program-block, occurring in an external-label-list, get at the macro-insertion the scope of the entire assemblage (see also 4).

⟨17⟩

Logically, we must distinguish three kinds of actual-parameter: the default-symbol "=", any word other than the default-symbol, and no actual-parameter in a position where one might be (i.e. incomplete actual-parameter specification).

17 continued

We call the default-symbol and the *absence* of an actual-parameter both a *default-actual*. We shall see that an overcomplete actual-parameter specification is syntactically allowed, though usually meaningless (overcomplete actual-parameters will have the same effect as a comment, but should not be abused as such).

Actual-parameters serve to specify the value of formal-parameters during the expansion of a macro as defined by the macro-insertion.

18

Note that no actual-parameters at all may be specified (the 'empty'-production), even if the macro-body has formal-parameters. In that case the actual values of all formal-parameters are their default-values (see also 16).

For overcomplete actual-parameters see 17 - it may be the consequence of certain actual-parameter specification in particular macro-bodies.

13 14 15 16 17 18

At a macro-insertion ('macro-call' or 'macro-actual', see also 5), the MACRO PROCESSOR takes over the function of the LEXICAL SCANNER: the sourcetext denoted by the macro-key is now read from the MACRO BODIES file.

The process of macro-expansion proceeds as follows:

1. Until a formal-parameter is recognized, the sourcetext (as obtained from the MACRO BODIES file) is passed on to the ASSEMBLY CONTROLLER without change.
2. If a formal-parameter is recognized, its *actual-value* is determined in the following manner:
   - the actual-parameter corresponding to the formal-parameter is designated by the current ordinal value of the formal-parameter;
   - if the actual-parameter is a default-actual (see also 17), then the actual value is the default-value of the formal-parameter (see also 16);
   - otherwise, the actual value of the formal-parameter is (the literal text of) the actual-parameter.
3. The actual-value is passed on instead of the formal-parameter, and the default-value (if any) is (further) skipped. Step 1 is taken again.

⟨13 --⟨18 continued

Observe that the MACRO PROCESSOR performs only textual substitution of actual parameters for formal-parameters. The further evaluation (if any) of words is a task of the ASSEMBLY CONTROLLER and/or the MACHINE-LANGUAGE PROCESSOR.

Note that the (additional) requirement that a macro be syntactically checkable at its declaration, is fully equivalent to the requirement that each macro expands to a syntactically correct program-block, even if the insertion without actual-parameters.

Note that actual-parameters which are not 'reached' in the expansion process, do not contribute to the final assemblage in any way. Whether an actual-parameter 'is reached' or not, may heavily depend on the evaluation of the int-expression of its formal ordinal (see also 15).

Note that a macro may 'go in recursion' ('self-insertion') either directly ('call itself'), or indirectly ('come back to itself' through other macro-insertions).

⟨19⟩

An assembly-compound is either assembly-control or a program-block - both address the ASSEMBLER CONTROLLER for either conditioned assembly of FANLAN-statements or for the establishment of nested program-blocks.

In the ASSEMBLY CONTROLLER assemblytime-values can be computed, inspected and altered, FANLAN-statements can be forwarded depending on an assemblytime-conjunction, assemblytime-messages can be specified, and the NAMELISTS can be set up in accordance to the required block-structure.

In an assemblytime-assignment a new value is assigned to an assemblytime-value. Assemblytime-values can be used (inspected) in assemblytime-conjunctions, and in the operand specification of machine language ('immediate operands'). In all these actions the ASSEMBLY CONTROLLER will perform all feasible type- and size-checks but, in the last instance, try always to follow the wishes of the programmer - an assemblytime-value, essentially, is a typeless (and, if necessary, truncated) bitpattern, just as runtime-data are.

In a conditional-assembly, the assemblytime-conjunction after the directive ".IF" or ".IFNOT" is determined (on the ground of the current assemblytime-value(s) involved). If the conjunction yields true, then the FANLAN-statements between ".THEN" and ".ELSE" (or ".FI") will be elaborated and all further sourcetext until the directive ".FI" will be skipped; otherwise, all source-

19 continued

text until the directive ".ELSE" (or ".FI" if there is no else-part) will be skipped and, if the directive ".ELSE" is present, the FANLAN-statements from this ".ELSE" until the corresponding directive ".FI" will be elaborated.

In a repetitional-assembly, the assemblytime-conjunction after the directive ".WHILE" or ".WHILENOT" is determined (on the ground of the current assembly-time-values(s) involved). If the conjunction yields true, then the FANLAN-statements between the directives ".DO" and its corresponding ".OD" will be elaborated. This process will be repeated until the conjunction becomes false.

There are three kinds of assemblytime-messages:
- a ".FATAL" message: the given string will be written in the assembly-report, and the assembler terminates the assemblage process;
- a ".WARNING" message: the given string will be written in the assembly-report, and the event is counted - after a certain number of warnings (a tunable number), the message may become 'fatal';
- a ".SOURCE" message: the given string will be written (on the spot) in the sourcetext-listing.

20

An assemblytime-conjunction is either an assemblytime-condition, or the logical conjunction of two or more assemblytime-conditions. In this particular syntactic position, the eol (unless it begins with the continue-symbol "\") acts as an assemblytime logical operator '.AND' (which does not exist as such in FANLAN). It will be clear that and how (by De Morgans law) an assemblytime logical operator '.OR' can be expressed with the aid of the directives ".IFNOT" (or ".WHILENOT"), ".NOT" and eol (acting as an '.AND').

The assemblytime-condition can use four (entirely different) kinds of predicate:
- the text-comparators ".SAME", ".HEAD", ".TAIL" and ".CONT", by which two words are compared textually;
- the value-comparators ".EQ" (=), ".LT" (<) and ".GT" (>), by which assembly-time-values are compared;
- the type-comparator ".TYPE", by which a type-check can be done;
- the size-comparator ".SIZE", by which a size-check can be done.

⟨21⟩

In a text-comparison the first of the two operands must always be a formal-text. In this syntactic position (i.e. of a 'formal-text'), a formal-parameter will always be replaced by its actual-parameter, even if the actual-parameter is "=" or empty - in other words: the default-mechanism (see also 17 and 18) does not work in this situation of text-comparison.

The predicate ".SAME" requires the first word (the actual text) to be the same as the second word.
The predicate ".HEAD" requires that the first word (the actual text) begins as given by the second word (one or more characters).
The predicate ".TAIL" requires that the first word (the actual text) ends as given by the second word (one or more characters).
The predicate ".CONT" requires that the first word (the actual text) contains the given second word as a substring.

⟨22⟩

In a value-comparison only expressions of the same type can be compared, and only if these types are 'int', 'real' or 'address'. Note that the value of an address-expression can not be compared with the value of an int-expression - the 'physical address' is an unknown entity in FANLAN (see also 6).

Predicates '≠', '<=' and '>=' can be expressed with the aid of ".NOT" (or ".IFNOT" or ".WHILENOT") and ".EQ", ".GT" and ".LT" respectively.

⟨23⟩

In a type-comparison, the type of the first word (always an actual obtained from a formal-parameter) is compared with the type given by the second word.

There are four kinds of type:
- types given by the data-keys BIN OCT HEX BCD INT REAL STRING and REF;
- the type of ".SKIP" declared data, which is essentially an 'all-type' - the type of the FANCY-data-registers DØ --- D7 is SKIP,
  and NOTYPEs: all forward-references (possibly in most assemblers all labels declared in program-space) are NOTYPEs ('NOTYPE' should be understood as 'unknown type', not to be confused with 'all-type' SKIP);
- IDENT being a (second) type of all identifiers (including keys);
- INSTR being the type of all opcode-keys and all data declared through instructions, MACRO the type of a macro-key and SUBRT the type of a subroutine-name.

⟨23 continued

A type-comparison does not compare sizes, i.e. objects of different size may be of the same type.

Types are essentially attributes of labels, expressions and denotations (that is to say that virtually all actual-parameters will have a well-defined type, the only exceptions are certain syllables which are then classified as NOTYPE).

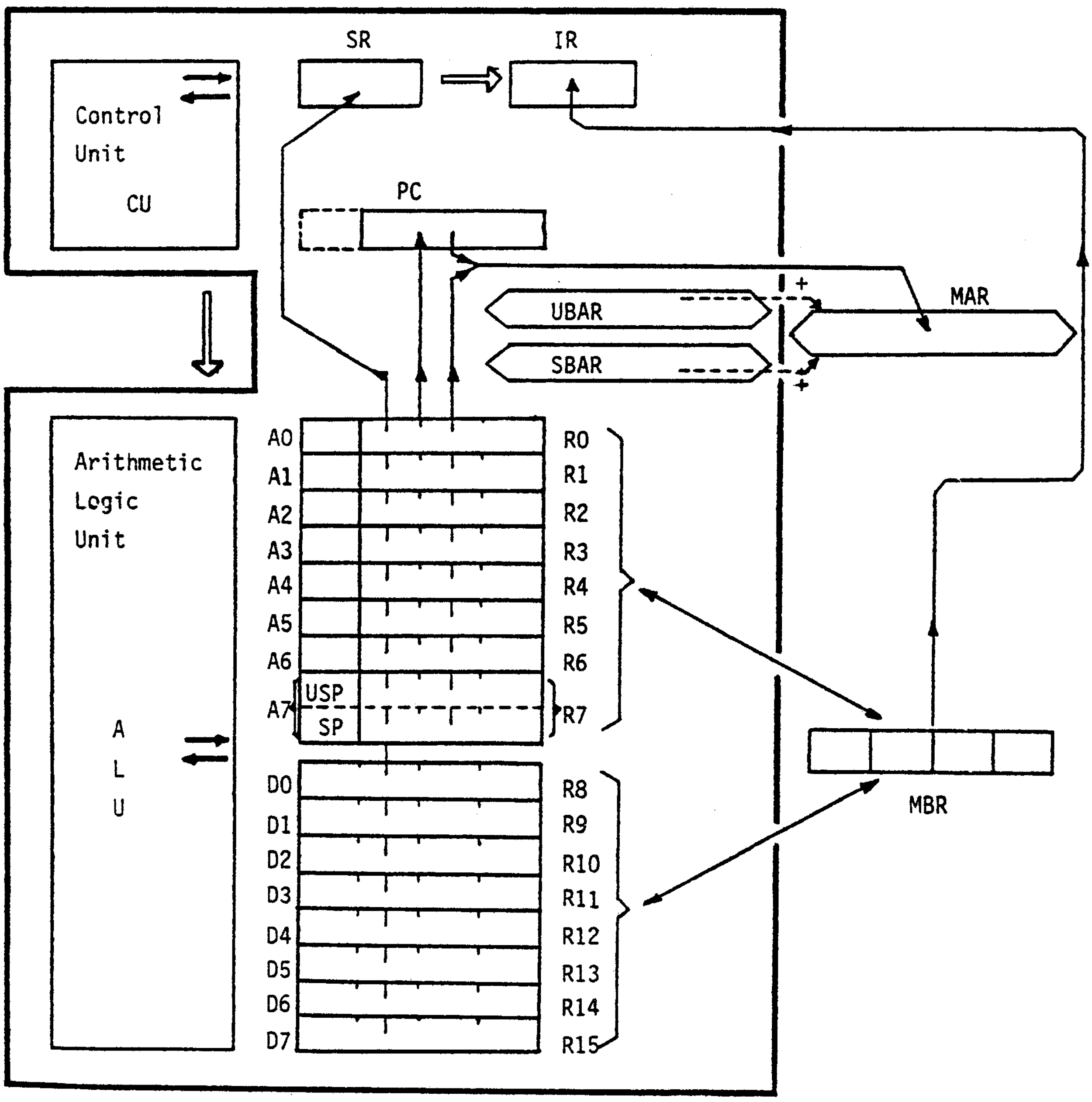
⟨24⟩

In a size-comparison the number of bytes are compared.

The sizes of instructions may be undefined (may be not defined by the opcode-key alone); the sizes of macro's and subroutines is always undefined. Undefined size results in a false yielded by every size-comparison.

C O N C R E T E   M A C H I N E   S T R U C T U R E

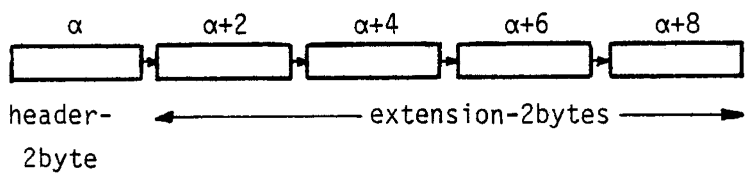
2.0 THE FANÇY PROCESSOR


SR
IR
Control Unit CU
PC
UBAR
SBAR
MAR
+
+
Arithmetic Logic Unit
A L U
A0
A1
A2
A3
A4
A5
A6
A7
USP
SP
R0
R1
R2
R3
R4
R5
R6
R7
D0
D1
D2
D3
D4
D5
D6
D7
R8
R9
R10
R11
R12
R13
R14
R15
MBR

The FANCY-processor executes an instruction in a set of small steps which can be described (in an ALGOL-like manner) as:

```
while  processor busy
   do  MAR := PC;
       PC +:= 2;
       FETCH2;
       IR := MBR2;
         if  IR requires operands
       then  FETCH operands required
         fi;
       execute IR
   od;

proc FETCH2 = void:  MBR2 := M[MAR];
    C  where MBR2 identifies the two rightmost bytes of MBR  C

proc IR requires operands = bool: <body> ;
    C  yields true if the opcode requires one or more memory-
       operands C

proc FETCH operands required = void: <body> ;
    C  fetches all operands required in (possibly) hidden registers C

proc execute IR = void: <body> ;
    C  executes the instruction in IR, as specified by the opcode  C
```

The FANCY-instructions in FANCY-RAM (Random Access Memory) consist of one or more 2bytes, arranged as follows:



In the header-2byte of the instruction (located at address α in RAM) the instruction-coding is present and - if possible - the operand-coding. In many cases, however, this operand-coding requires more information-bits which are then stored in one or more extension-2bytes.
After the FETCH of the header-2byte the programcounter (PC) points to the first extension-2byte (or the next instruction if no extra extension-information is present). For each operand-FETCH the extra extension-information is fetched first and when fetching this extension-2byte or -4byte, PC is incremented by 2 or 4 resp.

If an operand is addressed PC-relative, then PC contains at <u>execution-time</u> the address of the relevant extension-2byte or extension-4byte.
The FANLAN-assembler will take care of the correct <distance>value in the PC-relative address-coding.

<u>The FANCY-registers</u>

In the FANCY-processor we find the following registers:

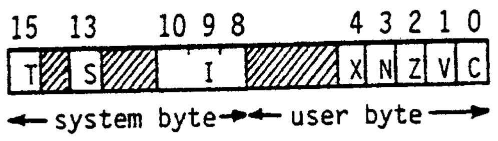
SR	= Status Register	[2 bytes]
IR	= Instruction Register	[16 bits]
PC	= Program Counter	[24 bits]
MAR	= Memory Address Register	[32 bits]
MBR	= Memory Buffer Register	[4 bytes]
UBAR	= User Base Address Register	[32 bits]
SBAR	= System Base Address Register	[32 bits]
R_i	= ALU-registers	[32 bits]

The ALU-registers can be subdivided into:

A_h	= Address register	$0 \leq h \leq 7$	$A_h \equiv R_h$
D_h	= Data register	$0 \leq h \leq 7$	$D_h \equiv R_{h+8}$

The Address Register A7 is in the FANCY always used as the Stack Pointer (SP) and the FANCY-machine does have two different stack-pointer for the User (User Stack Pointer, USP) and the operating-system (System Stack Pointer, SP).

The Status Register (SR) is subdivided as follows:



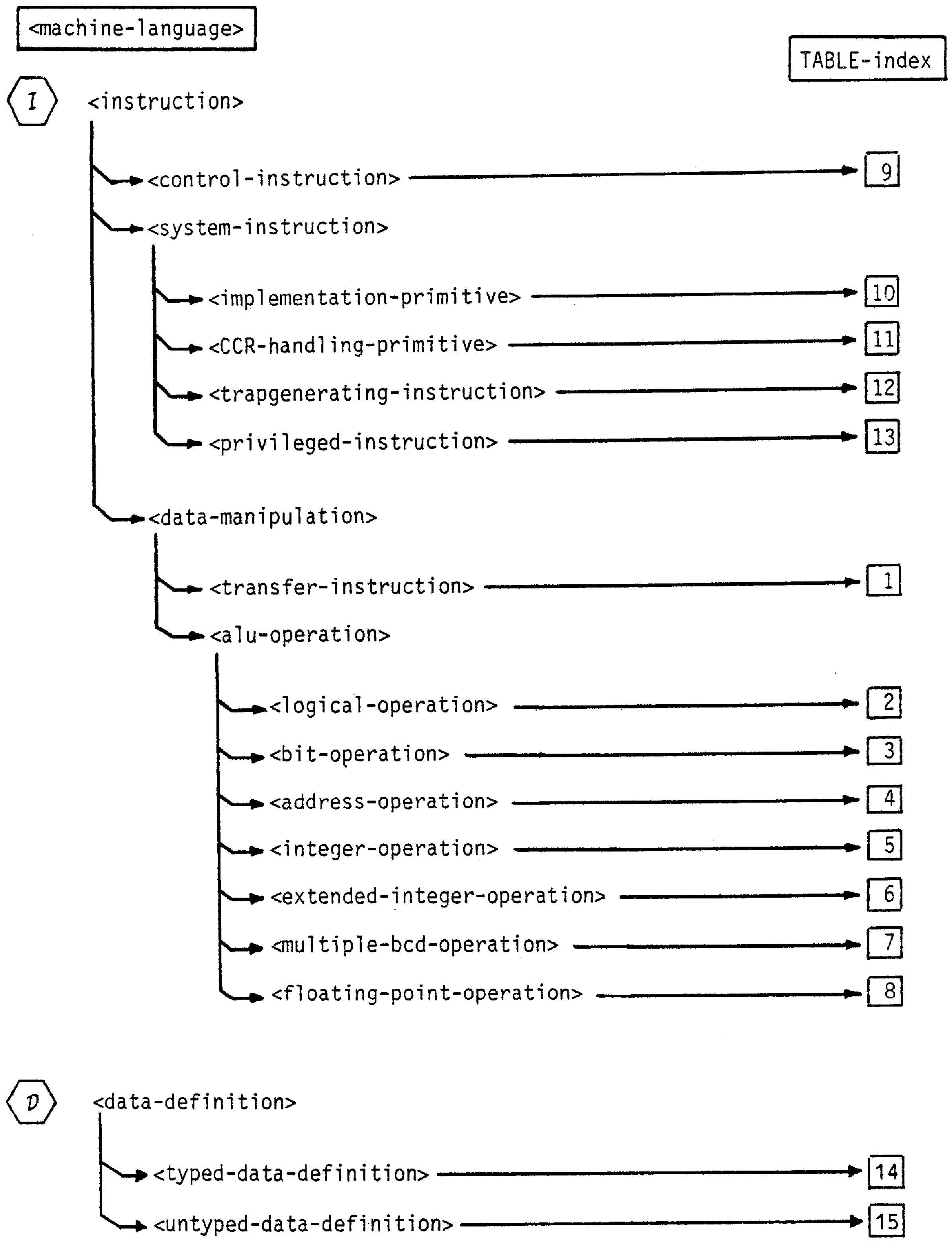
The meaning of the specified bits in SR is:

T = Trace mode	1 = on, 0 = off
S = System state	1 = on (FANCY executes in operating-system-mode),
	0 = off (user-mode)
I = Interrupt mask	$0 \leqslant I < 8$
X = Extend	
N = Negative	
Z = Zero	1 = yes (set), 0 = no (clear)
V = Overflow	
C = Carry	

The user-byte is normally specified with the name Condition Code Register (CCR).
The non-specified bits of SR are (currently) not in use by the FANCY-machine.

C O N C R E T E   M A C H I N E   S T R U C T U R E

2.1 SYNTAX



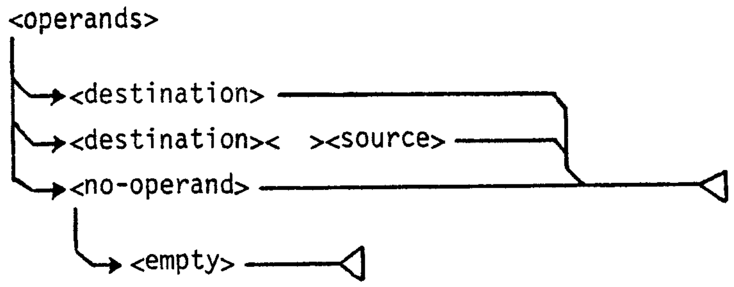
<machine-language>
TABLE-index
I
<instruction>
<control-instruction>
9
<system-instruction>
<implementation-primitive>
10
<CCR-handling-primitive>
11
<trapgenerating-instruction>
12
<privileged-instruction>
13
<data-manipulation>
<transfer-instruction>
1
<alu-operation>
<logical-operation>
2
<bit-operation>
3
<address-operation>
4
<integer-operation>
5
<extended-integer-operation>
6
<multiple-bcd-operation>
7
<floating-point-operation>
8
D
<data-definition>
<typed-data-definition>
14
<untyped-data-definition>
15

Legend to the instruction-tables.

<opcode> : FANLAN-assembler mnemonic for the operationcode.

<size> : operand-size;
when no size is specified in the FANLAN-instruction, the defaultvalue - indicated by ε - is assumed by the assembler.
1|2|4 .. 1byte, 2byte or 4byte.

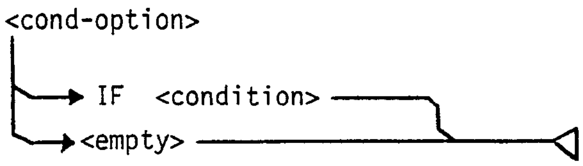
<operands> :



X N Z V C : the CCR-bits of the userbyte in the Status-Register.

- = not affected by the operation,
* = set according to the result of the operation,
0 = cleared,
1 = set,
? = undefined after the operation.

<cond-option> :



TRANSFER-INSTRUCTION

TABLE 1

<opcode>	<size>	<operands>		X N Z V C
COPY	2ε	<data-variable>	SR	- - - - -
COPY	1|2|4ε	<data-variable>	<data-source>	- * * 0 0
COPY	2|4ε	<data-variable>	$<A_k>$	- * * 0 0
COPYA	2|4ε	$<A_k>$	<operand>	- - - - -
COPYA	4ε	$<A_k>$	<address>	- - - - -
COPYA	4ε	@'SP	<address>	- - - - -
CLEAR	1|2|4ε	<data-variable>		- 0 1 0 0
SWAP	2ε	$<D_k>$		- * * 0 0
SWAP	4ε	$<R_i>$	$<R_j>$	- - - - -
SAVE	2|4ε	@'$<A_k>$ <control-memloc>	<bin-constant>	- - - - -
REST	2|4ε	@<bin-constant>	@$<A_k>$' <control-memloc>	- - - - -

LOGICAL-OPERATION

TABLE 2

<opcode>	<size>	<operands>		X N Z V C
ROL ROR	1|2|4ε	<D_k>	<D_h> <small-shift>	- * * 0 *
	2ε	<memory-variable>		
ROXL ROXR	1|2|4ε	<D_k>	<D_h> <small-shift>	* * * 0 *
	2ε	<memory-variable>		
NOT	1|2|4ε	<data-variable>		- * * 0 0
AND OR XOR	1|2|4ε	<memory-variable>	<hex-constant> <D_k>	- * * 0 0
		<D_k>	<data-source>	
LSL LSR	1|2|4ε	<D_k>	<D_h> <small-shift>	* * * 0 *
	2ε	<memory-variable>		

BIT-OPERATION

TABLE 3

<opcode>	<size>	<operands>		X N Z V C
BITTST BITCLR BITSET BITINV	1ε	<data-variable>	<D_h> <int-constant>	- - * - -
	4ε	<D_k>		

ADDRESS-OPERATION

TABLE 4

<opcode>	<size>	<operands>		X N Z V C
ADDA SUBA	2|4ε	<A_k>	<operand>	- - - - -
COMPA				- * * * *

INTEGER-OPERATION

TABLE 5

<opcode>	<size>	<operands>		X N Z V C
ADD SUB	1|2|4ε	<memory-variable>	<int-constant> <D$_k$>	* * * * *
		<D$_k$>	<data-source>	
	2|4ε		<A$_h$>	
MUL MULU	2ε	<D$_k$>	<data-source>	- * * 0 0
DIV DIVU				- * * * 0
COMP	1 2 4	<data-variable>	<int-constant>	- * * * *
		<D$_k$>	<data-source>	
	2|4ε		<A$_k$>	
	1|2|4ε	@<A$_k$>'	@<A$_h$>'	
NEG	1|2|4ε	<data-variable>		* * * * *
SIGNX	2|4ε	<D$_k$>		- * * 0 0
ASL ASR	1|2|4ε	<D$_k$>	<D$_h$> <small-shift>	* * * * *
	2ε	<memory-variable>		

EXTENDED-INTEGER-OPERATION

TABLE 6

<opcode>	<size>	<operands>		X N Z V C
ADDX SUBX	1|2|4ε	<D$_k$>	<D$_h$>	* * * * *
		@'<A$_k$>	@'<A$_h$>	
NEGX		<data-variable>		

MULTIPLE-BCD-OPERATION

TABLE 7

<opcode>	<size>	<operands>		X N Z V C
ADDBCD SUBBCD	1ε	<D$_k$>	<D$_h$>	* ? * ? *
		@'<A$_k$>	@'<A$_h$>	
NEGBCD		<data-variable>		

FLOATING-POINT-OPERATION

TABLE 8

<opcode>	<size>	<operands>		X N Z V C
ADDF SUBF	4ε	<memory-variable>	<real-constant> <D_k>	? * * * ?
		<D_k>	<data-source>	
MULF DIVF	4ε	<memory-variable>	<real-constant>	? * * * ?
		<D_k>	<data-source>	
COMPF	4ε	<memory-variable>	<real-constant>	? * * * ?
		<D_k>	<data-source>	
		∂<A_k>'	∂<A_h>'	
NEGF FLOAT	4ε	<data-variable>		? * * ? ?
FIXED	4ε	<data-variable>		? * * * ?

CONTROL-INSTRUCTION

TABLE 9

<opcode>	<operands>		<cond-option>
GOTO CALL	<memory-location>		IF <condition>
GOCNT	$<D_k>$	<named-location>	
RETURN RETRES	<no-operand>		
NOP			

SYSTEM-IMPLEMENTATION-PRIMITIVE

TABLE 10

<opcode>	<size>	<operands>		X N Z V C
LINK	4ε	$\langle A_k \rangle$	<distance>	- - - - -
UNLINK	4ε	$\langle A_k \rangle$		- - - - -
TST	1|2|4ε	<data-variable>		- * * 0 0
TAS	1ε	<data-variable>		- * * 0 0

CCR-HANDLING-PRIMITIVE

TABLE 11

<opcode>	<size>	<operands>		X N Z V C
SETBYTE	1ε	<data-variable>	<condition>	- - - - -
COPY	2ε	CCR	<data-source>	* * * * *
AND OR XOR	1ε	CCR	<hex-constant>	* * * * *

TRAPGENERATING-INSTRUCTION

TABLE 12

<opcode>	<size>	<operands>		X N Z V C
TRAP		<vector-number>		- - - - -
TRAPOF		<no-operand>		
BOUND	2ε	$<D_k>$	<data-source>	- * ? ? ?

PRIVILEGED-INSTRUCTION

TABLE 13

<opcode>	<size>	<operands>		X N Z V C
COPY	2ε	SR	<data-source>	* * * * *
AND OR XOR			<hex-constant>	
COPYA	4ε	USP	$<A_k>$	- - - - -
		$<A_k>$	USP	
RESET		<no-operand>		- - - - -
RETEXC		<no-operand>		* * * * *
HALT		<bin-constant>		* * * * *

TYPED-DATA-DEFINITION

TABLE 14

<data-key>	<data-size>	<datavalue-specification>
BIN	<rep-option> 1|2|4	<binary-sequence>
OCT	<rep-option> 1|2|4	<octal-sequence>
HEX	<rep-option> 1|2|4	<hexa-sequence>
BCD	<rep-option> 1|2|4	<decimal-sequence>
INT	<rep-option> 1|2|4	<int-expression>
REAL	<rep-option> 4ε	<real-expression>
STRING	<nat-option>	<string-denotation>
		<int-expression>
REF	<rep-option> 2ε|4	<address-expression>

UNTYPED-DATA-DEFINITION

TABLE 15

<data-key>	<data-size>	<datavalue-specification>
SKIP	<int-expression>	<empty>

C O N C R E T E   M A C H I N E   S T R U C T U R E

2.3 SEMANTICS

The FANLAN-statusvector consists of (one or more consecutive regions in) the FANCY Random Access Memory (RAM) together with the visible registers (REG) in the FANCY-processor. The RAM can be conceived as a bytes-array: M[0:upb] in which 'upb' is a machine-constant.

statusvector = RAM ∪ REG

| RAM = $\sum_{k=1}^{n}$ M[lwb_k:upb_k]

n and $lwb_1 < upb_1 < lwb_2 < \ldots . < lwb_n < upb_n$ are operating system dependent: n=1 is possible, as also $lwb_1=0$ and $upb_n=upb$.

| REG = A0 ∪ A1 ∪ A2 ∪ A3 ∪ A4 ∪ A5 ∪ A6 ∪ SP(=A7) ∪
D0 ∪ D1 ∪ D2 ∪ D3 ∪ D4 ∪ D5 ∪ D6 ∪ D7 ∪
SR ∪ PC (for further details, see 2.0)

In the context of a particular instruction, only a small part Γ of the statusvector may be involved. Γ, normally, is subdivided in two disjunct regions: Γ = Δ ∪ Σ ∪ RAM ∪ REG.

| Δ consists of those bitpattern(s) that may be changed by the instruction
= the destination.

| Σ consists of the bitpattern(s) that will not be changed, but contribute
to the operation defined by the instruction
= the source.

Δ and Σ may be the operands of the instruction, or be subdivided in operands Δ_1, Δ_2, ... and Σ_1, Σ_2, ...

In some cases an operand is to be considered explicitly as a bit array, in which cases an index or slice of that array will be specified between curly brackets '{' and '}' - an operand between curly brackets will always be interpreted as a natural (unsigned) number. Note that the numbering in a (machine-) bit-array is always from right to left, although we follow the notational tradition of writing 'lwb:upb' with lwb<upb.

The sequel contains a semantic description of the FANLAN-instructions as specified in the tables 1 to 13. The operations are specified in an ALGOL-like notation, exploring the convenience of (dyadic) assigning operators, such as '+:=', '∧:=', etc.
If necessary, the semantics will be clarified by self-explaning pictures. Special notes are to be found under REMARKS.

TRANSFER-INSTRUCTION

TABLE 1

$\boxed{\text{COPY}}$ $\boxed{\Delta := \Sigma}$

COPY data.

$\boxed{\text{COPY ..,SR}}$ $\boxed{\Delta\{15:0\} := \text{SR}}$

COPY Status Register.

$\boxed{\text{COPYA}}$ $\boxed{\Delta := \Sigma}$

COPY Address.

$\boxed{\text{CLEAR}}$ $\boxed{\Delta := 0}$

CLEAR operand.
All bits of Δ are set to zero.

$\boxed{\text{SWAP}}$ $\boxed{\begin{array}{r} D_h\{0:15\} \;\; :=: D_h\{16:31\} \\ R_i :=: R_j \end{array}}$

SWAP register halves or registers.
When one operand only is specified the two halves of the dataregister specified are interchanged.
When two registers are specified the contents of the registers specified are interchanged.

$\boxed{\begin{array}{l} \text{SAVE} \\ \text{REST} \end{array}}$ The registers specified by the binary-constant are transferred to (SAVE) or from (REST) memory, starting at the location specified. The binary-constant, always to be specified as a row of 16 bits, specifies in its most-significant 8 bits the address-registers and the least-significant 8 bits specify the dataregisters to be transferred. Thus BIN 00001011_00111000 specifies the registers A3, A1, A0, D5, D4 and D3 to be transferred.
The instructionsize specifies how much of each register is transferred. A 2byte-size specifies the loworder 2bytes of the registers.

The SAVE- and REST-instruction allow three forms of addressing:

1. @'<A_k> (predecremented)
   Only allowed for SAVE. The registers are stored starting at the specified address minus two (if size=2) or four (size=4) and down to lower addresses. The order of storing is from A7 to A0, then from D7 to D0. The decremented addressregister is updated to contain the address of the last location stored.

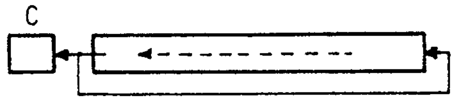
2. @<A_k>' (postincremented)
   Only allowed for REST. The values to be reloaded into the registers are located at the specified address and up through higher addresses. The order of loading is from D0 to D7, then from A0 to A7. The incremented addressregister is updated to contain the address of the last loaded location plus two (if size=2) or four (size=4).

3. <control-memloc>
   The registers are transferred starting at the address specified and up through higher addresses. The order of transfer is for the SAVE-operation as mentioned under 1. and for the REST-operation as under 2.

LOGICAL-OPERATION TABLE 2

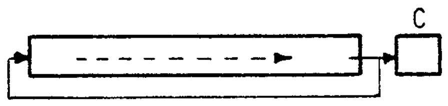
ROL Δ rol Σ

ROtate Left



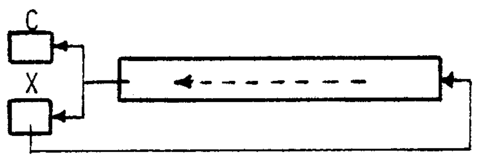
ROR Δ ror Σ

ROtate Right



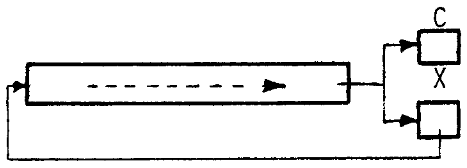
$\boxed{\text{ROXL}}$ $\boxed{\Delta\ \underline{\text{roxl}}\ \Sigma}$

ROtate Left with eXtend.



$\boxed{\text{ROXR}}$ $\boxed{\Delta\ \underline{\text{roxr}}\ \Sigma}$

ROtate Right with eXtend.



$\boxed{\text{NOT}}$ $\boxed{\Delta\ := \neg\Delta}$

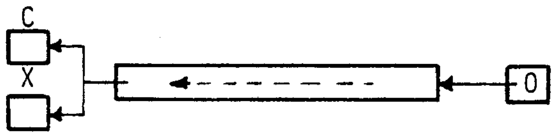
Logical NOT.
Each bit of Δ is replaced by its logical complement.

$\boxed{\begin{array}{l}\text{AND}\\ \text{OR}\\ \text{XOR}\end{array}}$ $\text{logical}\begin{cases}\text{AND}\\ \text{inclusive OR}\\ \text{eXclusive OR}\end{cases}$ $\boxed{\begin{array}{l}\Delta\ \wedge:=\ \Sigma\\ \Delta\ \vee:=\ \Sigma\\ \Delta\ |:=\ \Sigma\end{array}}$

$0\wedge0 = 0\wedge1 = 1\wedge0 = 0,\ 1\wedge1 = 1$ AND
$0\vee0 = 0,\ 0\vee1 = 1\vee0 = 1\vee1 = 1$ OR
$0|0 = 0,\ 0|1 = 1|0 = 1,\ 1|1 = 0$ XOR

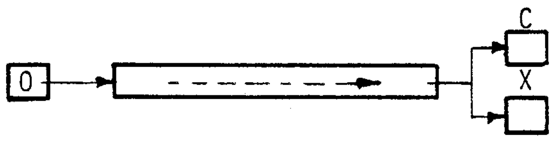
$\boxed{\text{LSL}}$ $\boxed{\Delta\ \underline{\text{lsl}}\ \Sigma}$

Logical Shift Left



$\boxed{\text{LSR}}$ $\boxed{\Delta\ \underline{\text{lsr}}\ \Sigma}$

Logical Shift Right



LOGICAL-OPERATION (cont'd) TABLE 2 (cont'd)

REMARKS:

- when shifting or rotating a memory-operand, the shiftcount is always equal to 1.
- when shifting or rotating a dataregister the shiftcount is specified by and when this is a dataregister its contents is taken modulo 64.

BIT-OPERATION

$\boxed{\text{TABLE 3}}$

$\boxed{\text{BITTST}}$ $\boxed{Z := \Sigma_1\{\Sigma_2\}}$

TeST a BIT

Bitnumber Σ_2 in Σ_1 is tested and the Z-bit in CCR is set according to the value of the tested bit.

$\boxed{\begin{array}{l}\text{BITCLR}\\\text{BITSET}\\\text{BITINV}\end{array}}$ test a BIT and $\left\{\begin{array}{l}\text{CLeaR}\\\text{SET}\\\text{INVert}\end{array}\right.$ $\boxed{\begin{array}{r}Z := \Delta\{\Sigma\};\\ \Delta\{\Sigma\} := \left\{\begin{array}{c}0\\1\\\overline{\Delta\{\Sigma\}}\end{array}\right.\end{array}}$

These instructions perform the same test and Z-bit setting as the BITTST-instruction. Afterwards the tested bit is cleared, set or inverted.

REMARK: when Δ is a dataregister the bitnumbering is modulo 32; being a memory-location then modulo 8 (due to the 1byte-operation).

ADDRESS-OPERATION

$\boxed{\text{TABLE 4}}$

$\boxed{\text{ADDA}}$	ADD Address	$\boxed{A_k \mathrel{+}:= \Sigma}$
$\boxed{\text{SUBA}}$	SUBtract Address	$\boxed{A_k \mathrel{-}:= \Sigma}$
$\boxed{\text{COMPA}}$	COMPare Address	$\boxed{CCR := A_k - \Sigma}$

REMARKS:

- when the size of an address-operation equals 2, the contents of the Δ-addressregister <u>as a whole</u> is affected: both operands are sign-extended to 32 bits before the operation is performed and this result is stored in the addressregister (if requested by the operation).
  <u>This holds for all operations having an addressregister as the destination-operand</u>!
- the arithmetic used in the address-operations is signed-integer arithmetic.

INTEGER-OPERATION

$\boxed{\text{TABLE 5}}$

$\boxed{\text{ADD}}$ $\boxed{\Delta \mathrel{+}:= \Sigma}$

ADD integer.

$\boxed{\text{SUB}}$ $\boxed{\Delta \mathrel{-}:= \Sigma}$

SUBtract integer.

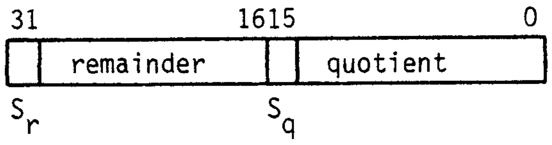
$\boxed{\text{MUL}}$ $\boxed{D_k := D_k\{0:15\}*\Sigma\{0:15\}}$

MULtiply integer.
Multiply the signed 2byte-operands yielding a signed 4byte-result.

$\boxed{\text{DIV}}$

$$\boxed{\begin{array}{l} D_k\{0:15\} := D_k \underline{\text{over}}\ \Sigma\{0:15\} \\ D_k\{16:31\} := D_k \underline{\text{mod}}\ \Sigma\{0:15\} \end{array}}$$

DIVide integer.
Divide the 4byte signed-integer Δ by the 2byte signed-integer Σ yielding a 4byte-result as specified below.



REMARKS:

- the sign of the remainder is always the same as the divident unless the remainder is equal to zero.
- division by zero causes a trap.
- if overflow is detected, this condition is flagged but the operands are <u>unaffected</u>.

$\boxed{\text{MULU}}$

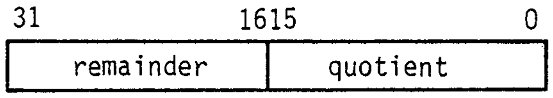
$$\boxed{D_k := D_k\{0:15\}*\Sigma\{0:15\}}$$

MULtiply Unsigned integer.
Multiply the unsigned 2byte-operands yielding an unsigned 4byte-result. The operation is performed using unsigned arithmetic.

$\boxed{\text{DIVU}}$

$$\boxed{\begin{array}{l} D_k\{0:15\} := D_k \underline{\text{over}}\ \Sigma\{0:15\} \\ D_k\{16:31\} := D_k \underline{\text{mod}}\ \Sigma\{0:15\} \end{array}}$$

DIVide Unsigned integer.
Divide the 4byte unsigned Δ by the 2byte unsigned Σ yielding a 4byte unsigned result as specified below.
The operation is performed using unsigned arithmetic.



REMARKS:

- division by zero causes a trap.
- if overflow is detected, this condition is flagged but the operands are unaffected.

$\boxed{\text{COMP}}$ $\boxed{CCR := \Sigma_1 - \Sigma_2}$

COMPare integer.

$\boxed{\text{NEG}}$ $\boxed{\Delta := 0-\Delta}$

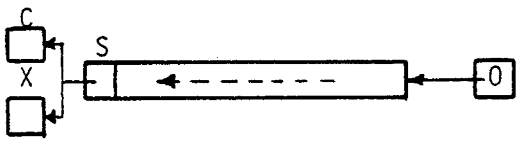
NEGate integer.

$\boxed{\text{SIGNX}}$ SIGN eXtend.

Extend the signbit of the specified dataregister from a 1byte to a 2byte or from a 2byte to a 4byte, depending on size selected. If the size equals 2, $D_k\{7\}$ is copied to $D_k\{8:15\}$; and if the size equals 4, $D_k\{15\}$ is copied to $D_k\{16:31\}$.

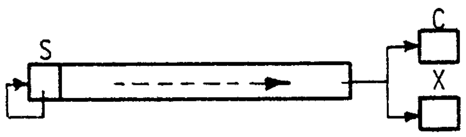
$\boxed{\text{ASL}}$ $\boxed{\Delta\ \underline{\text{asl}}\ \Sigma}$

Arithmetic Shift Left.



$\boxed{\text{ASR}}$ $\boxed{\Delta\ \underline{\text{asr}}\ \Sigma}$

Arithmetic Shift Right.



REMARKS:

- when shifting a memory-operand the shiftcount is always equal to 1.
- when shifting a dataregister the shiftcount specified by Σ is taken modulo 64 if Σ is a dataregister itself.
- the overflowbit V in CCR is set if the most significant bit of Δ is changed at any time during the shiftoperation and cleared otherwise.

EXTENDED-INTEGER-OPERATION

TABLE 6

ADDX	ADD eXtended integer.	$\Delta := \Delta+\Sigma+X$
SUBX	SUBtract eXtended integer.	$\Delta := \Delta-\Sigma-X$
NEGX	NEGate eXtended integer.	$\Delta := 0-\Delta-X$

REMARK: X is the extendbit in CCR.

MULTIPLE-BCD-OPERATION

TABLE 7

ADDBCD	ADD BCD-integer with extend.	$\Delta := \Delta+\Sigma+X$
SUBBCD	SUBtract BCD-integer with extend.	$\Delta := \Delta-\Sigma-X$
NEGBCD	NEGate BCD-integer with extend.	$\Delta := 0-\Delta-X$

REMARKS:
- these operations are performed using binary-coded-decimal arithmetic.
- the NEGBCD-instruction produces the ten's complement of Δ if the X-bit is cleared and the nine's complement of Δ if the X-bit is set.

FLOATING-POINT-OPERATION

TABLE 8

ADDF	ADD Floating-point.	$\Delta +:= \Sigma$
SUBF	SUBtract Floating-point.	$\Delta -:= \Sigma$
MULF	MULtiply Floating-point.	$\Delta *:= \Sigma$
DIVF	DIVide Floating-point.	$\Delta /:= \Sigma$

FLOATING-POINT-OPERATION (cont'd) TABLE 8 (cont'd)

$\boxed{\text{COMPF}}$ COMPare Floating-point. $\boxed{CCR := \Sigma_1 - \Sigma_2}$

$\boxed{\text{NEGF}}$ NEGate Floating-point. $\boxed{\Delta := 0.0 - \Delta}$

$\boxed{\text{FLOAT}}$ convert signed-integer to FLOATing-point. $\boxed{\Delta := \underline{\text{real}}(\Delta)}$
The 4byte signed-integer representation of Δ
is converted to a floating-point representation.
When the integervalue fits in the mantissepart of the
floating-point-representation <u>no</u> rounding takes place in
the conversion. If it doesn't fit, <u>rounding</u> off takes
place in the conversionprocess.

$\boxed{\text{FIXED}}$ convert floating-point to FIXED. $\boxed{\Delta := \underline{\text{int}}(\Delta)}$
This is the reverse operation of FLOAT.
Given the floating-point value of Δ, the integervalue
equal to or the nearest integervalue <u>below</u> the floating-
point value is computed and stored as a 4byte signed-integer
in Δ.

REMARK: all operations of table 8 are performed using floating-point
arithmetic.

<u>CONTROL-INSTRUCTION</u> $\boxed{\text{TABLE 9}}$

$\boxed{\text{GOTO}}$ $\boxed{PC := \#\Delta}$

GOTO location.
Program execution continues at the specified location.

$\boxed{\text{CALL}}$ $\boxed{\begin{array}{r} @'SP := PC \\ PC := \#\Delta \end{array}}$

CALL subroutine.
The address of the instruction immediatly
following the CALL-instruction (the returnaddress) is
pushed onto the stack and execution continues at the
specified location.

REMARK: the PC-value pushed onto the stack is stored into a 4byte memory-location.

$\boxed{\text{GOCNT}}$ GOto location if CouNTer is not exhausted.
The specified dataregister (containing the counter) is decremented by one. If the result equals -1, the counter is exhausted and execution continues with the instruction following the GOCNT. If the result is unequal to -1, execution continues at the specified location.

The GOTO-, CALL- and GOCNT-instruction may contain a condition. If the specified condition is met, the instruction is executed the way as described before; if the specified condition is not met, execution continues with the next instruction in sequence.
See next page for possible conditions and corresponding tests.

$\boxed{\text{RETURN}}$ $\boxed{\text{PC := @SP'}}$

RETURN from subroutine.
The programcounter is popped from the stack and execution continues at the reloaded location. The previous PC-value is lost.

$\boxed{\text{RETRES}}$ $\boxed{\begin{array}{l}\text{CCR := @SP'}\\ \text{PC := @SP'}\end{array}}$

RETurn from subroutine and REStore CCR.
First the conditioncodes are popped from the stack, then the programcounter is popped from the stack and execution continues at the reloaded location. The previous PC-value and the previous CCR-values are lost.
The system-byte of SR is not affected.

$\boxed{\text{NOP}}$ NO Operation.

CONTROL-INSTRUCTION (cont'd) TABLE 9 (cont'd)

condition	test performed
IF TRUE	"always"
IF FALSE	"never"
IF NOT CORZ	$\overline{C}.\overline{Z}$
IF CORZ	C+Z
IF NOT CARRY	$\overline{C}$
IF CARRY	C
IF NOT ZERO	$\overline{Z}$
IF ZERO	Z
IF NOT OF	$\overline{V}$
IF OF	V
IF NOT NEG } IF POS	$\overline{N}$
IF NEG	N
IF NOT LT } IF GE	$N.V+\overline{N}.\overline{V}$
IF LT	$N.\overline{V}+\overline{N}.V$
IF GT	$N.V.\overline{Z}+\overline{N}.\overline{V}.\overline{Z}$
IF NOT GT } IF LE	$Z+N.\overline{V}+\overline{N}.V$

SYSTEM-IMPLEMENTATION-PRIMITIVE

TABLE 10

LINK

$$\boxed{\begin{aligned} @'SP &:= A_k \\ A_k &:= SP \\ SP \mathrel{+}:= &\ distance \end{aligned}}$$

LINK and allocate.
The current content of the specified addressregister is pushed onto the stack. After the push, the addressregister is loaded with the updated stackpointer. Finally the specified distance is added to the stackpointer.

UNLINK

$$\boxed{\begin{aligned} SP &:= A_k \\ A_k &:= @SP' \end{aligned}}$$

UNLINK and de-allocate.
The stackpointer is loaded with the content of the specified addressregister. Then this address-register is loaded with the 4byte-value popped from the stack.

TST

$$\boxed{CCR := \Sigma}$$

TeST an operand.
Test the operand specified and set the N- and Z-bits in CCR accordingly.

TAS

$$\boxed{\begin{aligned} CCR &:= \Delta \\ \Delta\{7\} &:= 1 \end{aligned}}$$

Test And Set an operand.
This instruction performes the same test and N- and Z-bit setting as the TST-instruction (but only for a 1byte-operand). Afterwards the mostsignificant bit of Δ is set.
This operation is indivisible to allow synchronisation to be implemented.

CCR-HANDLING-PRIMITIVE

TABLE 11

SETBYTE

Δ := <u>if</u> condition
<u>then</u> TRUE
<u>else</u> FALSE
<u>fi</u>

SET BYTE according to condition.
The specified condition is tested;
if true then the byte specified by Δ
is set to TRUE (all ones) otherwise
this byte is set to FALSE (all zeroes).

COPY CCR ...

CCR := Σ

COPY to CCR.
The Σ is always a 2byte-operand, but only the loworder
byte is used to update the conditioncodes.

$\left.\begin{matrix}\text{AND}\\\text{OR}\\\text{XOR}\end{matrix}\right\}$ CCR ...

$\text{CCR}\left\{\begin{matrix}\wedge:=\\\vee:=\\|:=\end{matrix}\right\}\Sigma$

logically $\left\{\begin{matrix}\text{AND}\\\text{inclusive OR}\\\text{exclusive OR}\end{matrix}\right\}$ CCR with the

specified (1byte-) operand.

TRAPGENERATING-INSTRUCTION

TABLE 12

TRAP

The processor initiates exception-processing ie.
it enters the operating-system-mode and starts execution
of the traphandling-program specified by the vectornumber.

TRAPOF

TRAP on OverFlow.
If the V-bit of CCR is set, the processor enters the operating-system-mode and initiates the overflowhandling-program.
If the V-bit is cleared, no operation is performed and
execution continues with the next instruction in sequence.

$\boxed{\text{BOUND}}$ check register against BOUNDs.
The content of the loworder-2byte of the dataregister specified is compared with the (upper)bound specified by Σ. If the registervalue is less than zero or greater than the upperbound, then the processor enters the operating-system-mode and initiates the out-of-bouds-handling-program.

PRIVILEGED-INSTRUCTION

$\boxed{\text{TABLE 13}}$

The instructions mentioned in this table can only be executed when the processor is in operating-system-mode. When executed while the processor is in user-mode, they cause exceptionprocessing (on privilege-violation) by the processor.

$\boxed{\text{COPY SR ...}}$ $\boxed{\text{SR\{0:15\} := Σ}}$

COPY to SR.
All bits of the Status Register are affected.

$\boxed{\left.\begin{array}{l}\text{AND}\\\text{OR}\\\text{XOR}\end{array}\right\}\text{SR ...}}$ $\boxed{\text{SR\{0:15\} := SR\{0:15\}}\left\{\begin{array}{c}\wedge\\\vee\\|\end{array}\right\}Σ}$

$\text{logical}\left\{\begin{array}{l}\text{AND}\\\text{inclusive OR}\\\text{eXclusive OR}\end{array}\right\}\text{with SR}$

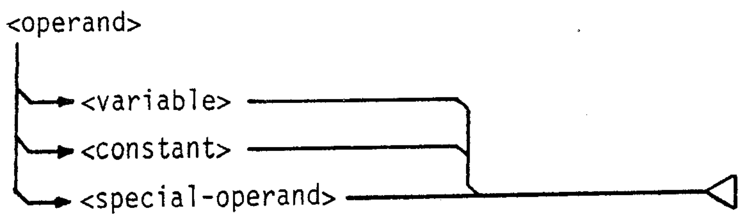
$\boxed{\begin{array}{l}\text{COPYA USP ...}\\\text{COPYA ... USP}\end{array}}$ COPY Address to/from the User Stack Pointer.

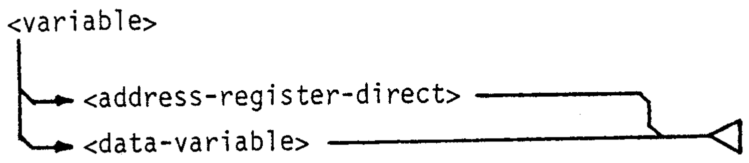
With these instructions the operating-systems programmer can load an address into/from the stackpointer of the userprogram under execution.

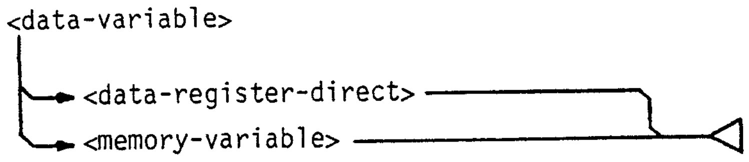
RESET	RESET external devices. The reset-line is asserted causing all external devices to be reset. Execution continues with the next instruction in sequence.
RETEXC	SR := @SP' PC := @SP' RETurn from EXCeption. The Status Register and the programcounter (in this order) are popped from the stack. The previous SR- and PC-values are lost and all bits in SR are affected.
HALT	load SR with the 16bit binary-value and HALT execution. After loading of SR with the binary-value, the programcounter is advanced to the next instruction and the processor stops fetching and executing instructions. Execution of instructions resumes when a trace-exception, an interrupt-request or a reset-exception occurs. If the bit of the binary-value cooresponding to the S-bit is cleared, execution of this instruction will cause a privilege-violation.

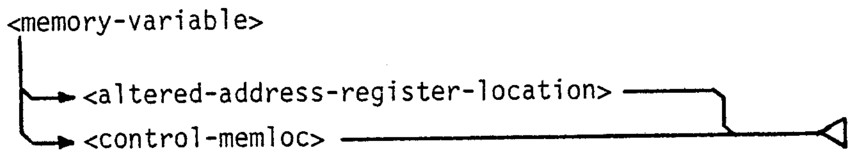
O P E R A N D   S T R U C T U R E

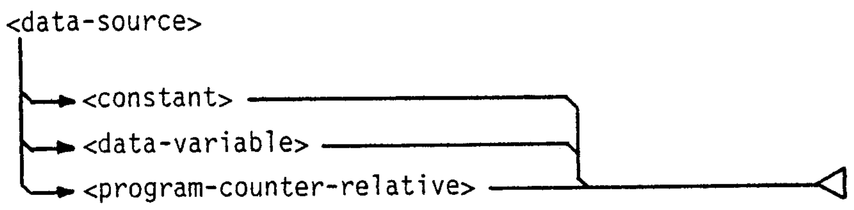
3.1 SYNTAX

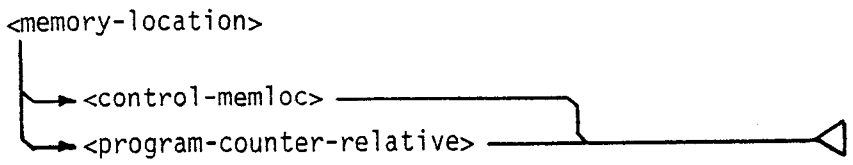
1


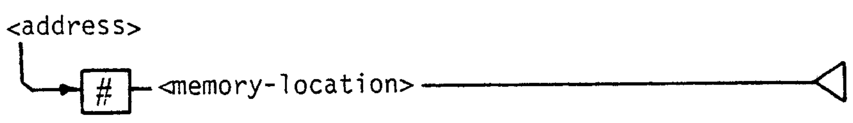
2


3


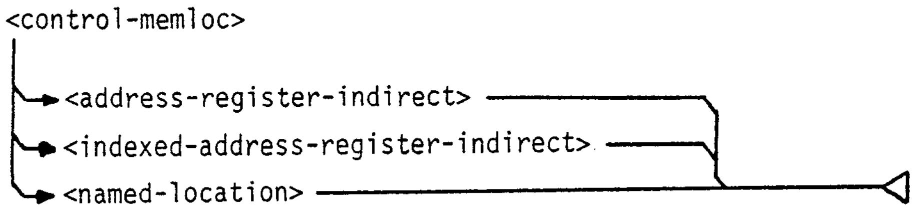
4


5


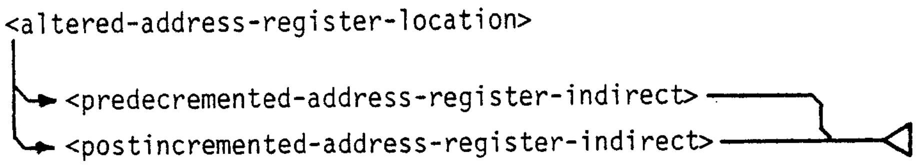
6


7


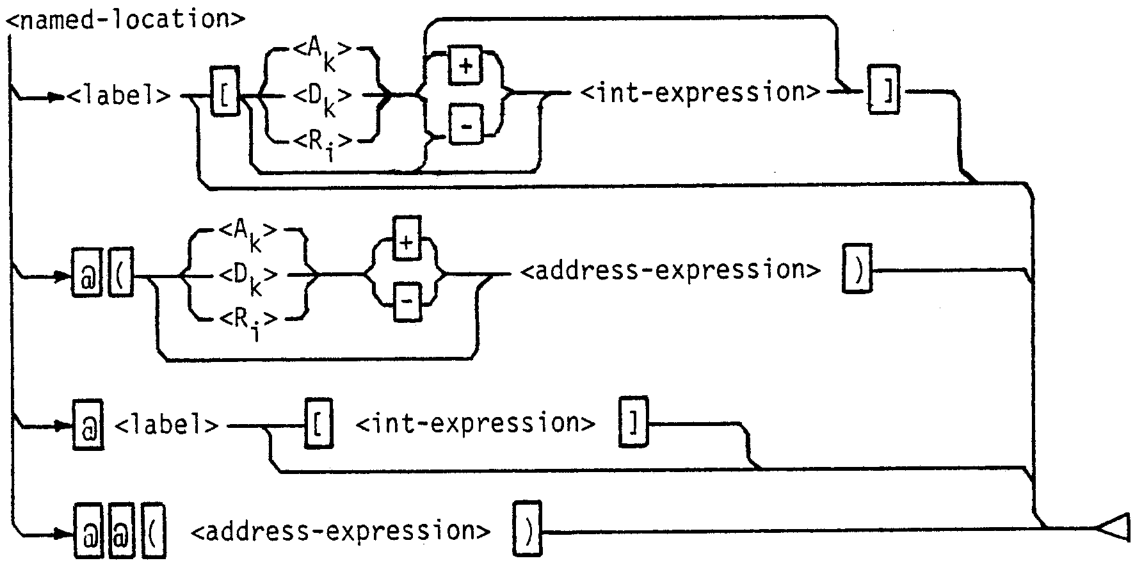
8



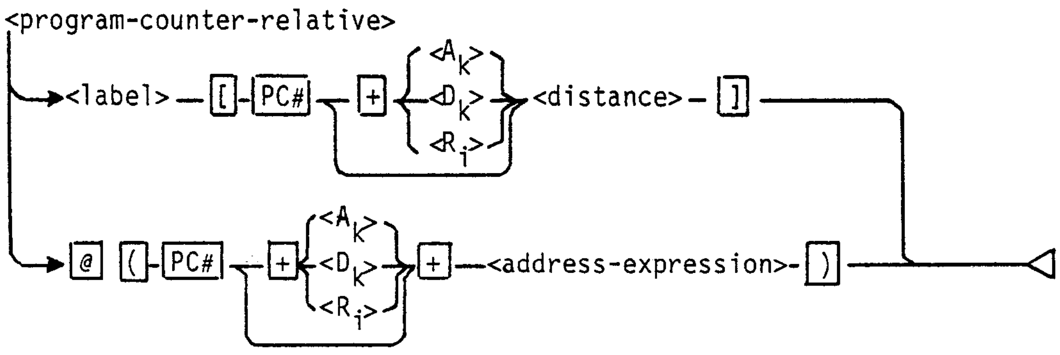
9



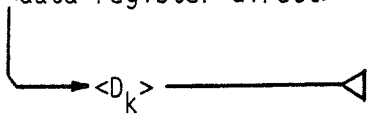
10



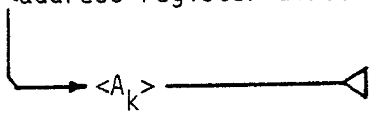
11



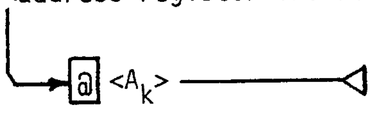
(12) <data-register-direct>



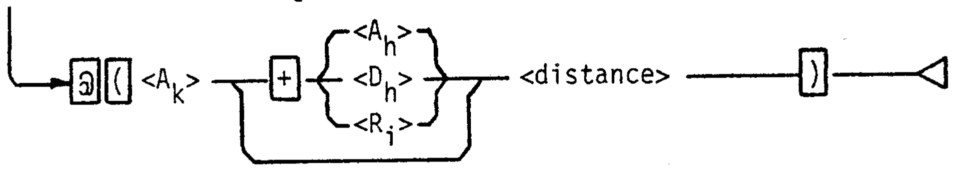
(13) <address-register-direct>



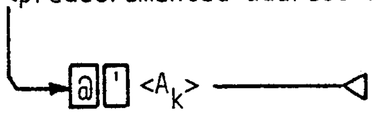
(14) <address-register-indirect>



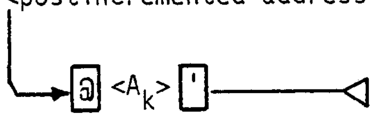
(15) <indexed-address-register-indirect>



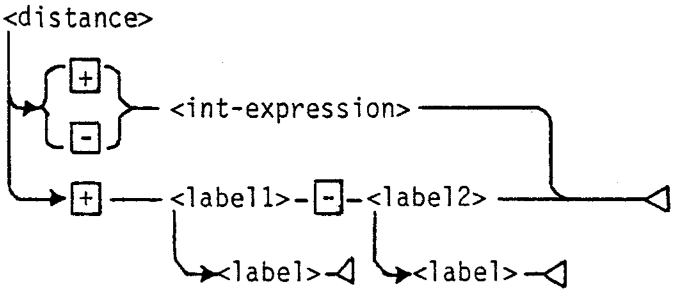
(16) <predecremented-address-register-indirect>



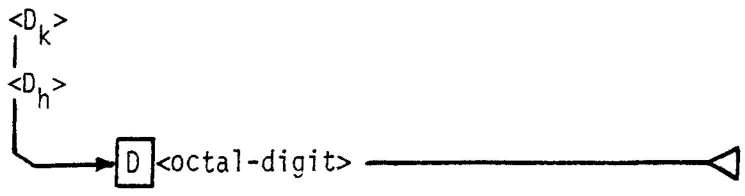
(17) <postincremented-address-register-indirect>



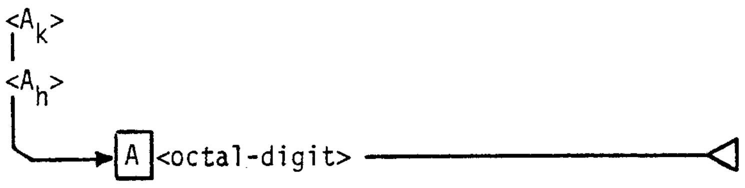
(18) <distance>



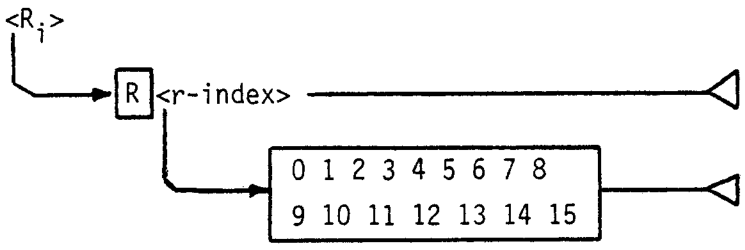
19



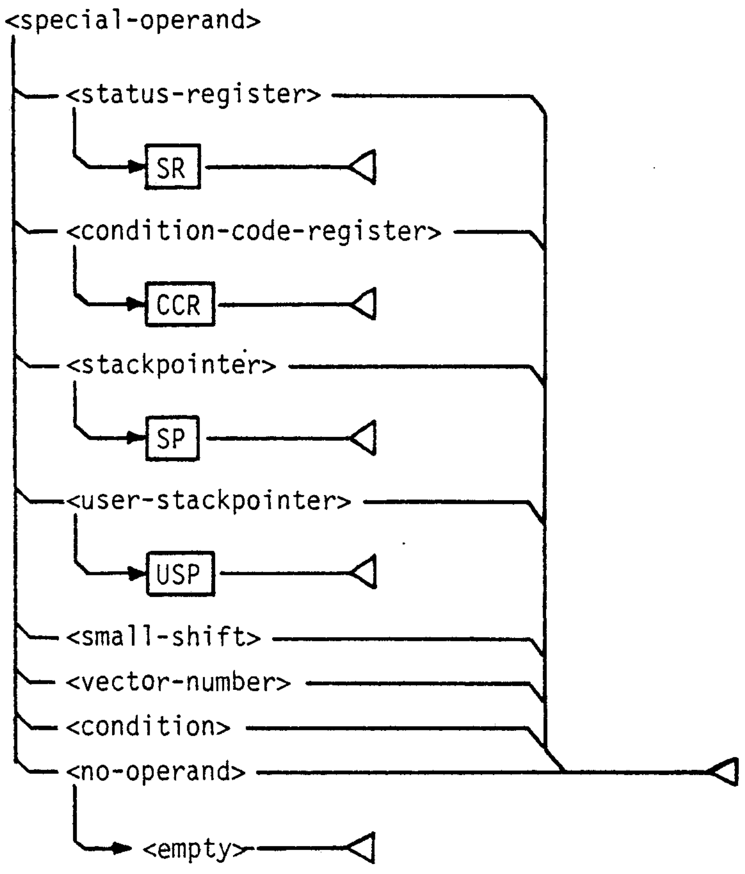
20

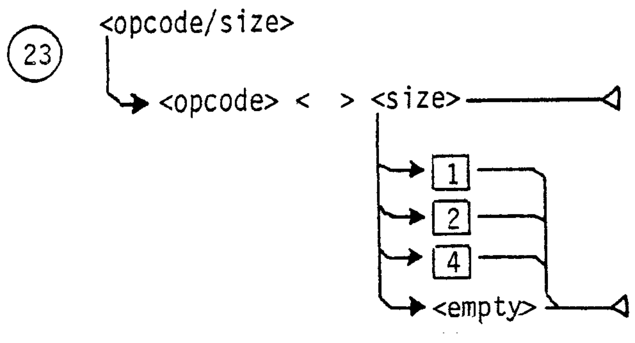


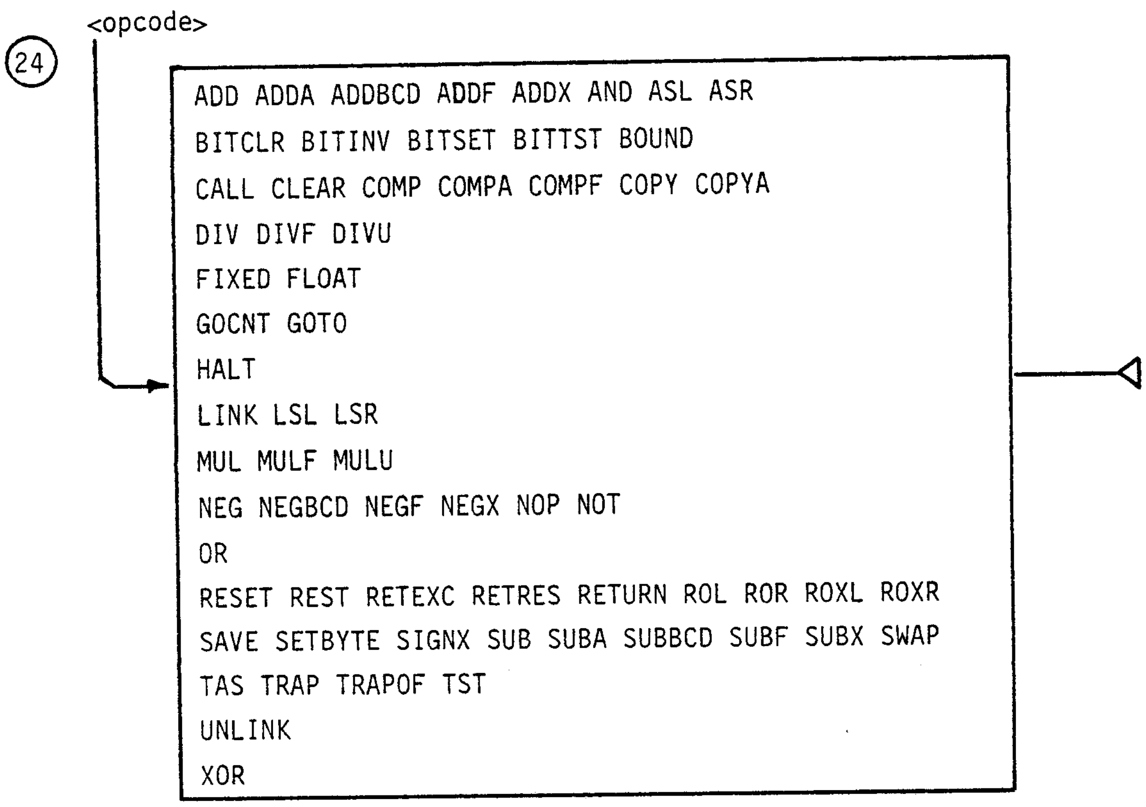
21



22



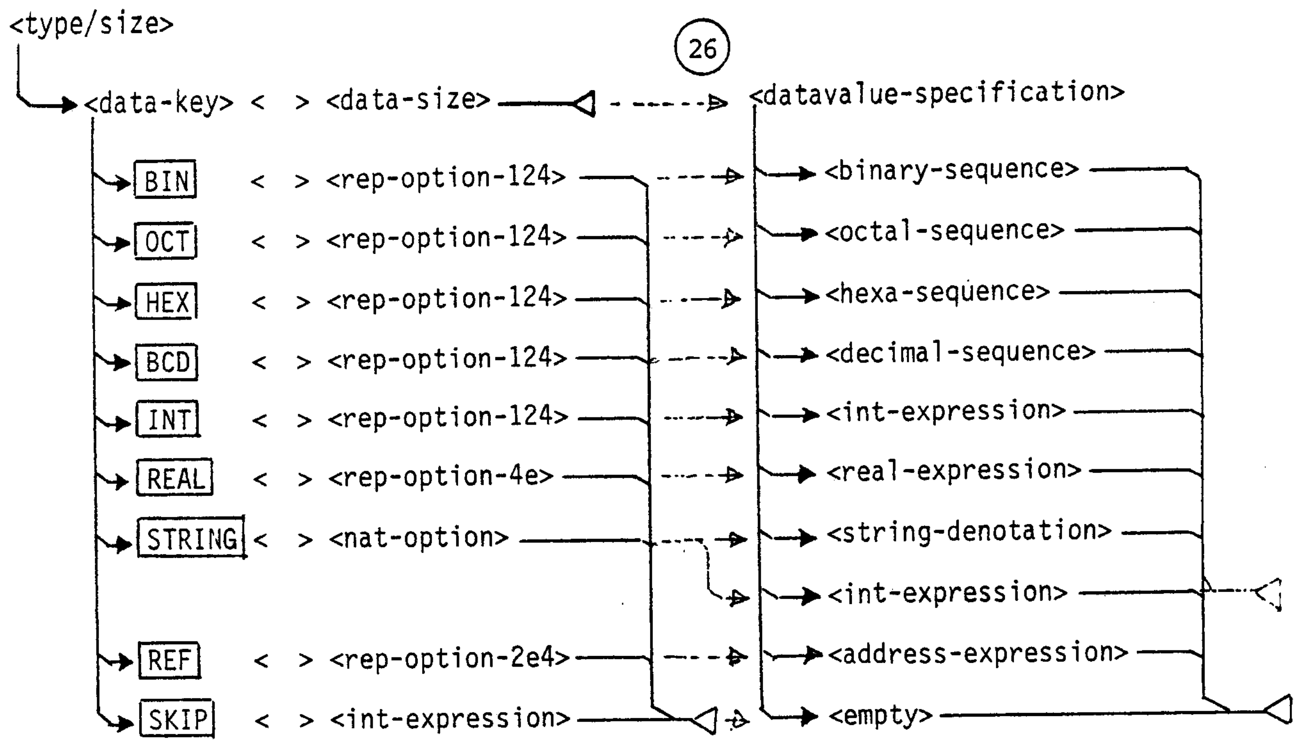




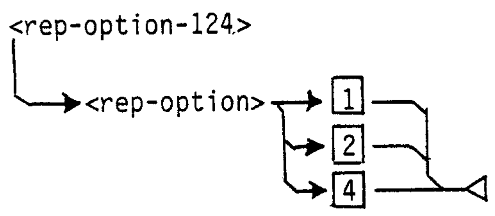
ADD ADDA ADDBCD ADDF ADDX AND ASL ASR
BITCLR BITINV BITSET BITTST BOUND
CALL CLEAR COMP COMPA COMPF COPY COPYA
DIV DIVF DIVU
FIXED FLOAT
GOCNT GOTO
HALT
LINK LSL LSR
MUL MULF MULU
NEG NEGBCD NEGF NEGX NOP NOT
OR
RESET REST RETEXC RETRES RETURN ROL ROR ROXL ROXR
SAVE SETBYTE SIGNX SUB SUBA SUBBCD SUBF SUBX SWAP
TAS TRAP TRAPOF TST
UNLINK
XOR

25

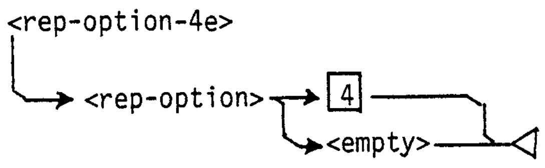
26



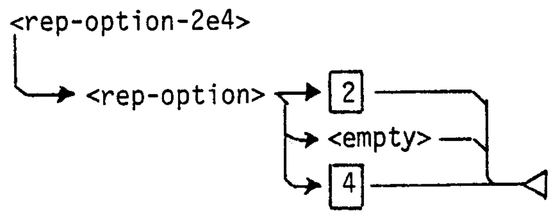
27



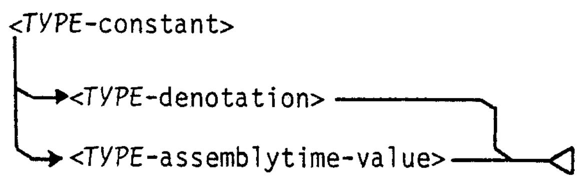
28



29

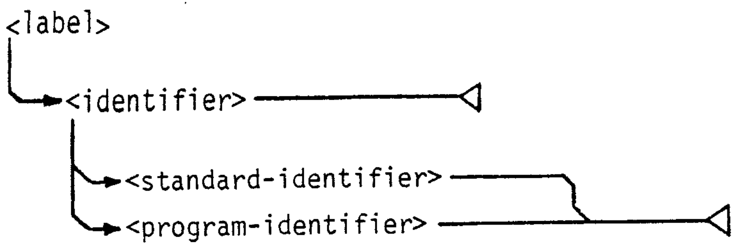


30

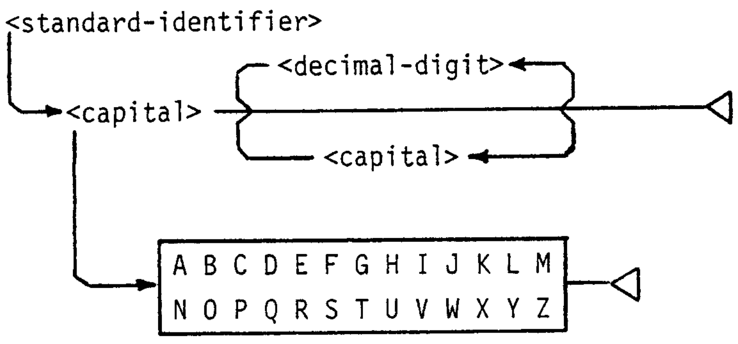


TYPE :: bin | oct | hex | bcd | int | real | string | address

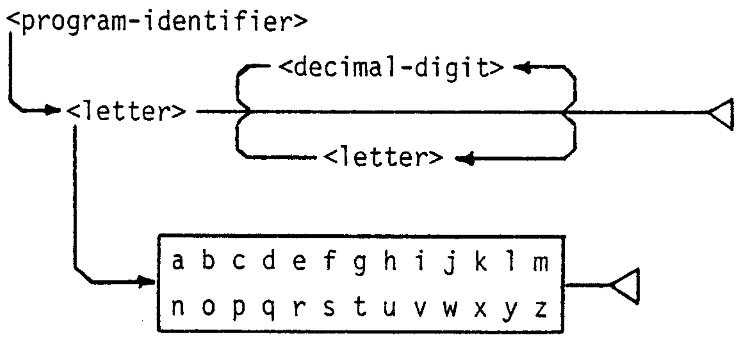
31


<label>
<identifier>
<standard-identifier>
<program-identifier>

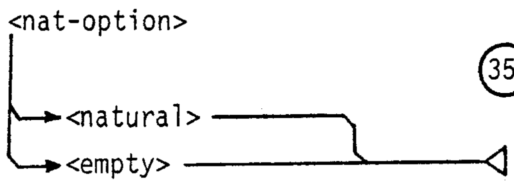
32


<standard-identifier>
<capital>
<decimal-digit>
<capital>
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

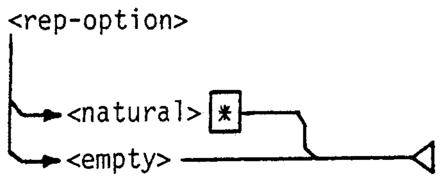
33


<program-identifier>
<letter>
<decimal-digit>
<letter>
a b c d e f g h i j k l m
n o p q r s t u v w x y z

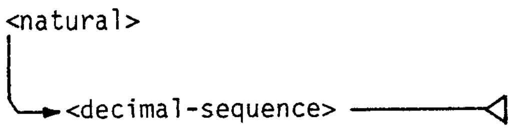
34



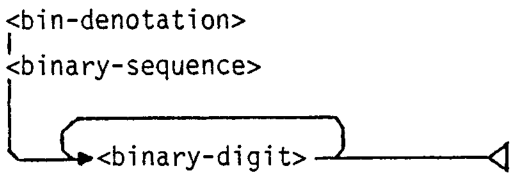
35



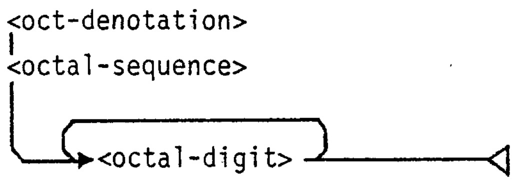
36



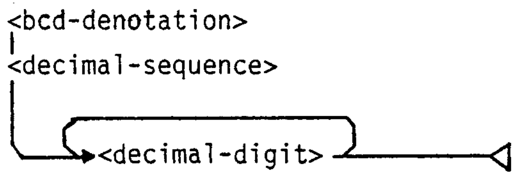
37



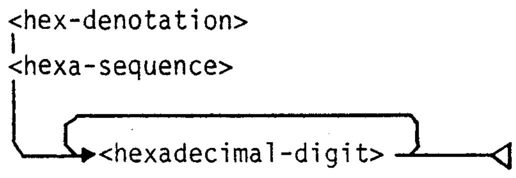
38



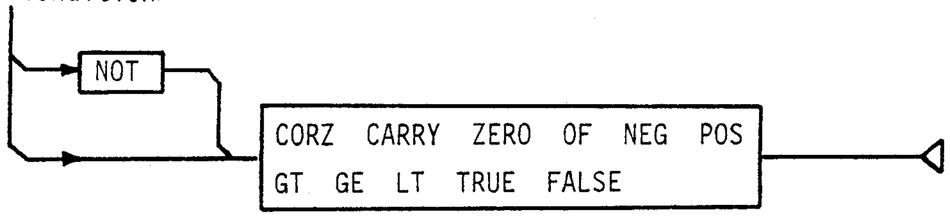
39



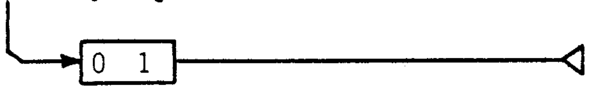
40



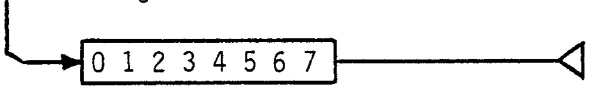
41 <condition>



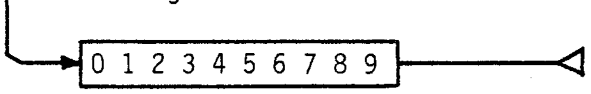
42 <binary-digit>



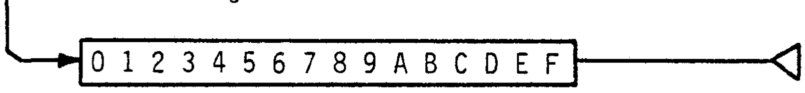
43 <octal-digit>



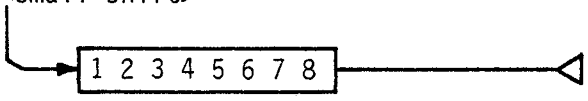
44 <decimal-digit>



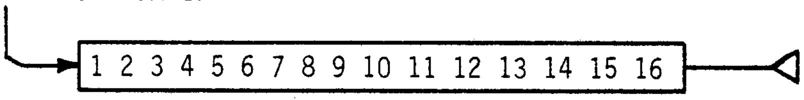
45 <hexadecimal-digit>



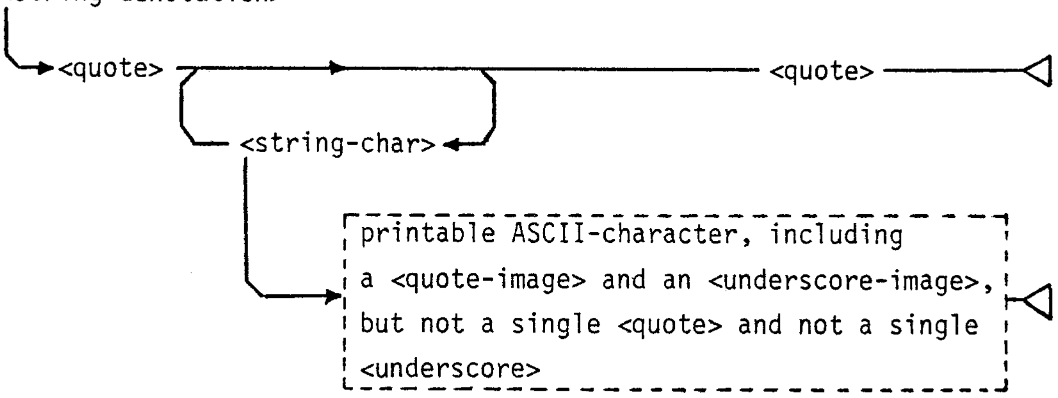
46 <small-shift>

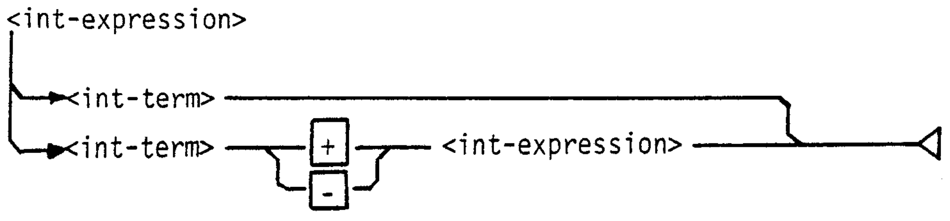


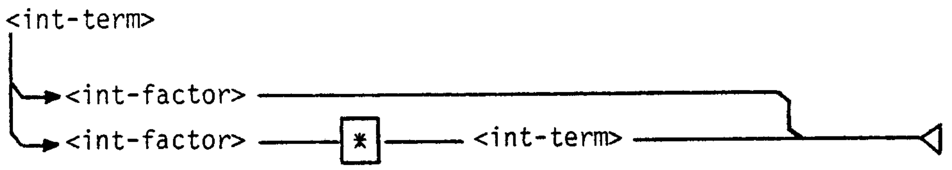
47 <vector-number>

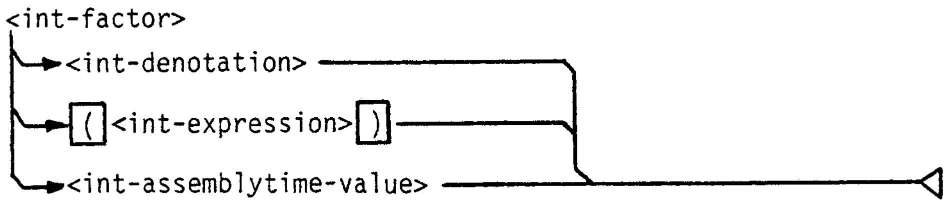


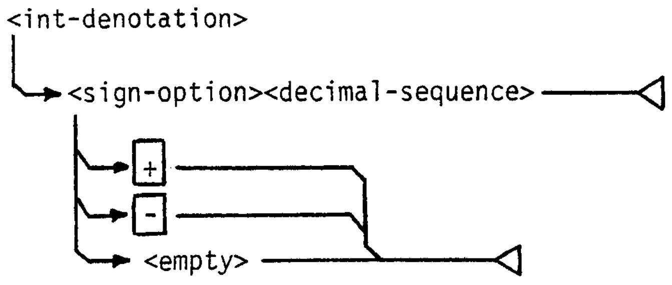
48 <string-denotation>

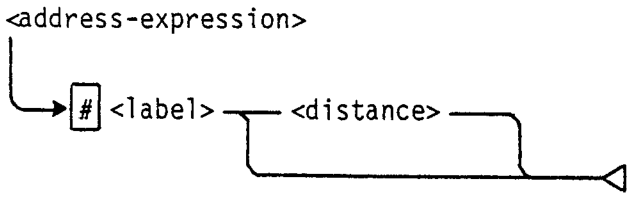

<quote>
<quote>
<string-char>
printable ASCII-character, including
a <quote-image> and an <underscore-image>,
but not a single <quote> and not a single
<underscore>

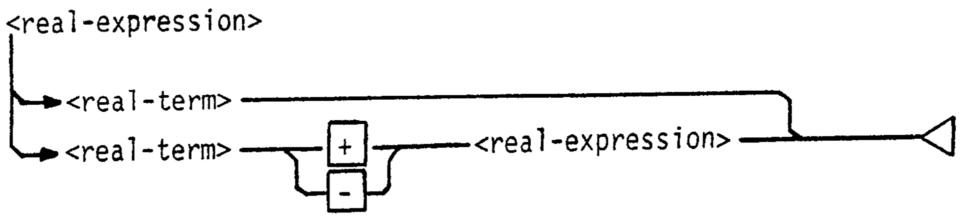
49


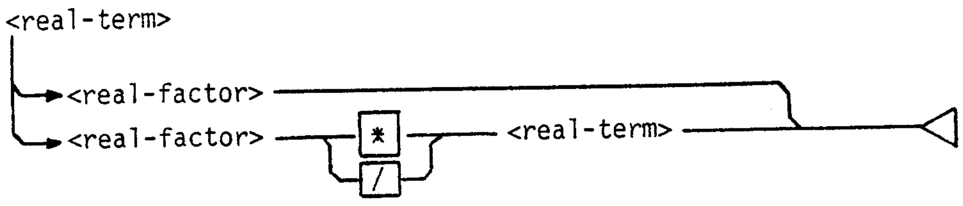
50


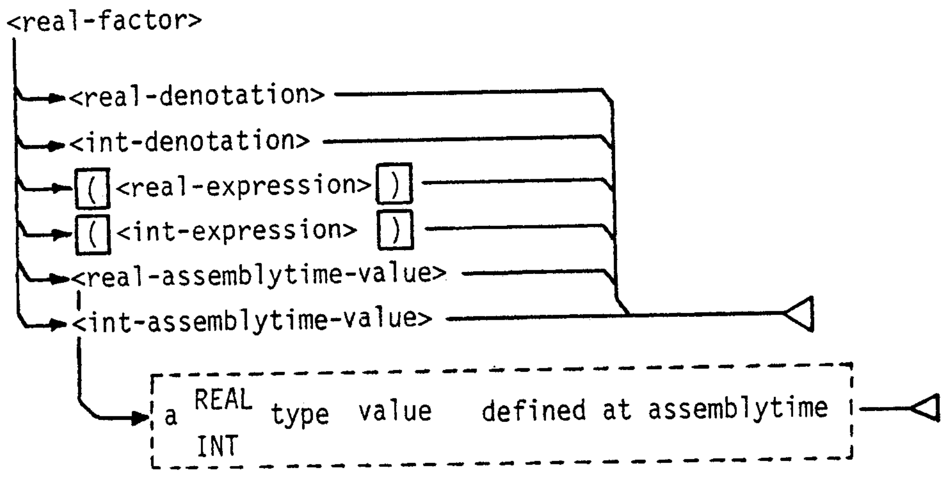
51


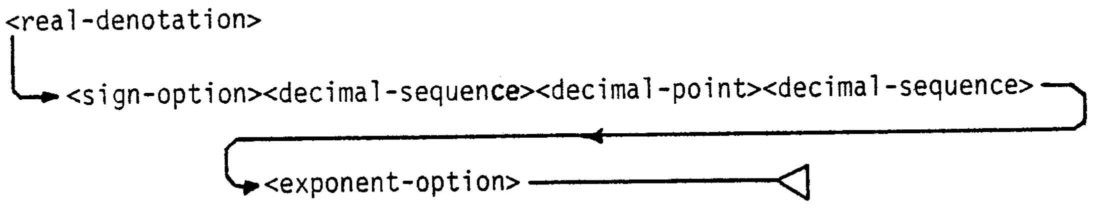
52


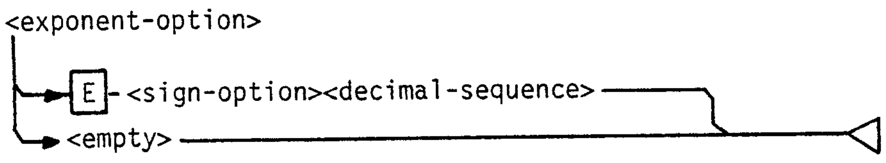
53


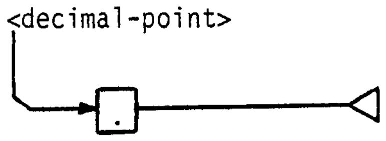
54


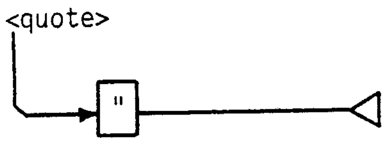
55


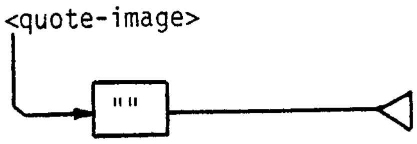
56


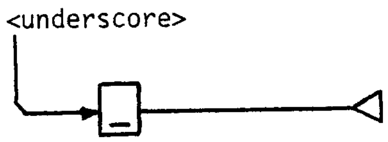
57


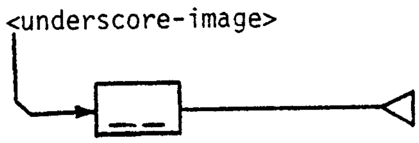
58


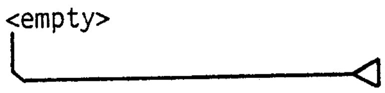
59


60


61


62


63


64


O P E R A N D   S T R U C T U R E

3.2 SEMANTICS

10
11
15

The indexregister used in the address-calculations can be specified in 2 ways:

1. $<A_k>$ or $<D_k>$

   This means the 4byte-content of this register is the indexvalue.

2. $<R_i>$

   This means the loworder-2byte content of this register is the indexvalue and this value is signextended to 4bytes before being used in the address-calculation.

See 2.0 "the FANCY processor" for the correspondence of $<R_i>$ with $<A_k>$ and $<D_k>$.

---

10 For the semantics of the address-notators, see INTRODUCTION 4.

---

11 The notation 'PC#' enforces the program-counter-relative addressing mode. The precise meaning of 'PC#' is: PC-#*here* , where *here* = the (possibly hidden) label just in front of the bytes in which 'PC#' is used.

16
17

Predecremented: $\boxed{A_k \text{ -:= size; operandvalue = } @A_k}$

Postincremented. $\boxed{\text{operandvalue = } @A_k\text{; } A_k \text{ +:= size}}$

The de(in)crement of the addressregister is always with the size of the operand. Is the addressregister the stackpointer (A7) and the operandsize equals 1 (1byte), then the stackpointer is de(in)cremented by 2 iso. 1 to keep the stackpointer alined on a 2byte-address. The value loaded/stored from/into memory is - of coarse - a 1byte-value.

---

(18) The resulting <distance>-value is restricted to the context in which the <distance> is used.

operand-addressing with indexregister	<distance>-value restriction
no	2byte
yes	1byte

(31) (32) (33) The FANLAN-assembler accepts both uppercase (<capital>) and lowercase (<letter>) lettersymbols.
The <standard-identifier>s all have a predefined meaning, see INTRODUCTION, while the <program-identifier>s are defined in the userprogram and have a userdefined meaning.

In case a user only has the uppercase lettersymbols to his position, the assembler accepts the <capital>s as <letter>s in a <program-identifier> and then a (re-)definition of a <standard-identifier> by the userprogram results in overriding the standardmeaning of this identifier (in the block in which the redefinition takes place).

A L P H A B E T I C   I N D E X

SYNTAX-DEFINITIONS

FANLAN-MNEMONICS

Syntax-definitions

definition-item	def-tree
address	7
address-expression	53
address-register-direct	13
address-register-indirect	14
A_h A_k	20
altered-address-register-location	9
bcd-denotation	39
binary-digit	42
binary-sequence	37
bin-denotation	37
capital	32
condition	41
condition-code-register	22
constant	30
control-memloc	8
data-key	25
data-register-direct	12
data-size	25
data-source	5
datavalue-specification	26
data-variable	3
decimal-digit	44
decimal-point	59
decimal-sequence	39
D_h D_k	19
distance	18
empty	64
exponent-option	58
hexadecimal-digit	45
hexa-sequence	40
hex-denotation	40

definition-item	def-tree
identifier	31
indexed-address-register-indirect	15
int-denotation	52
int-expression	49
int-factor	51
int-term	50
label	31
letter	33
memory-location	6
memory-variable	4
named-location	10
nat-option	34
natural	36
no-operand	22
octal-digit	43
octal-sequence	38
oct-denotation	38
opcode	24
opcode-size	23
operand	1
postincremented-address-register-indirect	17
predecremented-address-register-indirect	16
program-counter-relative	11
program-identifier	33
quote	60
quote-image	61
real-denotation	57
real-expression	54
real-factor	56
real-term	55
rep-option	35, 27,28,29
R_i	21
r-index	21

definition-item	def-tree
sign-option	52
size	23
small-shift	46
special-operand	22
stackpointer	22
standard-identifier	32
status-register	22
string-char	48
string-denotation	48
type-size	25
underscore	62
underscore-image	63
user-stackpointer	22
value-specification	26
variable	2
vector-number	47

FANLAN-mnemonics

opcode-mnemon.	TABLE	opcode-mnemon.	TABLE	opcode-mnemon.	TABLE
ADD	5	DIVU	5	RETRES	9
ADDA	4	FIXED	8	RETURN	9
ADDBCD	7	FLOAT	8	ROL	2
ADDF	8	GOCNT	9	ROR	2
ADDX	6	GOTO	9	ROXL	2
AND	2,11,13	HALT	13	ROXR	2
ASL	5	LINK	10	SAVE	1
ASR	5	LSL	2	SETBYTE	11
BITCLR	3	LSR	2	SIGNX	5
BITINV	3	MUL	5	SUB	5
BITSET	3	MULF	8	SUBA	4
BITTST	3	MULU	5	SUBBCD	7
BOUND	12	NEG	5	SUBF	8
CALL	9	NEGBCD	7	SUBX	6
CLEAR	1	NEGF	8	SWAP	1
COMP	5	NEGX	6	TAS	10
COMPA	4	NOP	9	TRAP	12
COMPF	8	NOT	2	TRAPOF	12
COPY	1,11,13	OR	2,11,13	TST	10
COPYA	1,13	RESET	13	UNLINK	10
DIV	5	REST	1	XOR	2,11,13
DIVF	8	RETEXC	13		

type-mnemon.	TABLE	DIAGR	type-mnemon.	TABLE	DIAGR
BCD	14	23	NOTYPE	--	23
BIN	14	23	OCT	14	23
HEX	14	23	REAL	14	23
IDENT	--	23	REF	14	23
INSTR	--	23	SKIP	15	23
INT	14	23	STRING	14	23
MACRO	--	23	SUBRT	--	23

these numbers
refer to chapter 1