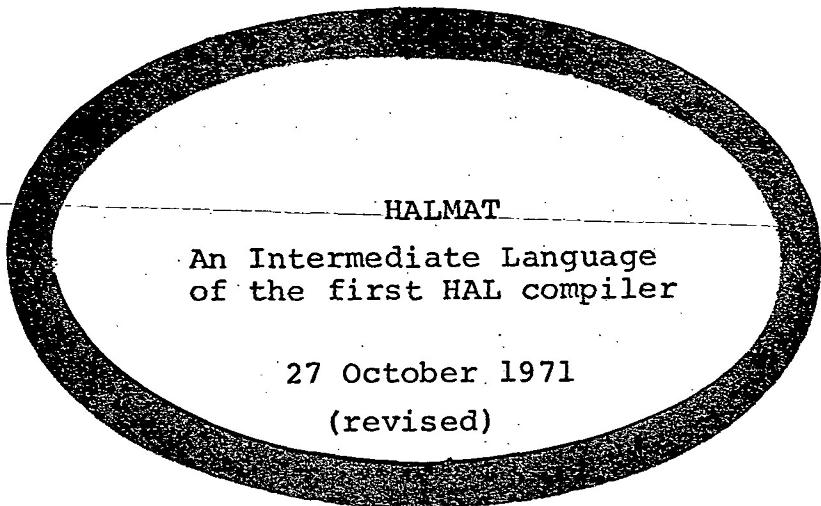
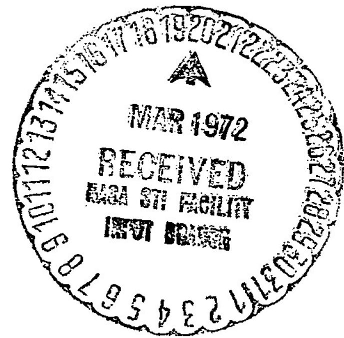


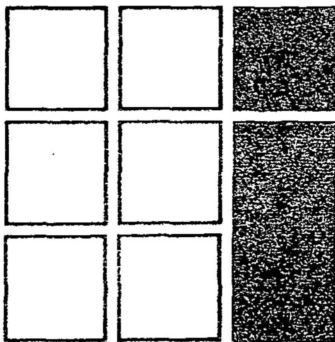
HALMAT

An Intermediate Language of the first HAL compiler

27 October 1971

(revised)





INTERMETRICS

(NASA-CR-126137) HALMAT: AN INTERMEDIATE LANGUAGE OF THE FIRST HAL COMPILER (Intermetrics, Inc.) 27 Oct. 1971 119 p
CSCL 09B

X72-10244

Unclas
B3/08 25276

CAT. 08

MSC-01847

HALMAT

An Intermediate Language
of the first HAL compiler

27 October 1971

(revised)

Submitted to:

National Aeronautics and Space Administration
Manned Spacecraft Center
Houston, Texas 77058

Submitted by:

Intermetrics, Inc.
380 Green Street
Cambridge, Mass. 02139

FOREWORD

This report was prepared by Intermetrics, Inc. under contract NAS 9-10542 from the Manned Spacecraft Center of the National Aeronautics and Space Administration. The Technical Monitor of this contract is Mr. Jack Garman/FS5.

The publication of this document does not constitute a document release by Intermetrics and information contained herein shall not be disclosed outside the Government.

PREFACE

This document is essentially a handbook of an intermediate language HALMAT produced and used during the operation of the HAL360-V1 compiler. The last phase of the compiler operation is the conversion of HALMAT text into object code. The document gives information sufficient both for the interpretation of HALMAT text, and for the writing of new code conversion phases for the HAL compiler.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	
1.1	HALMAT and the HAL Compiler	1-1
1.2	Symbolic Format of HALMAT Instructions	1-1
1.3	Symbolic Representation of OPERAND FIELD Contents	1-3
1.4	The TEXT OPTIMIZER TAG Subfield	1-6
1.5	Representation of OPERAND TYPES	1-7
1.6	Information not Supplied by an OPERAND FIELD	1-8
1.7	Physical Format of HALMAT Instructions	1-9
2.	THE HALMAT INSTRUCTION SET	
2.1	CODE MARKERS	2-1
2.2	LABELS	2-2
2.3	BRANCHES	2-2
2.4	ARRAYNESS and STRUCTURENESS SPECIFIERS	2-3
2.5	SUBSCRIPT ALLOCATORS	2-5
2.6	TERMINAL SUBSCRIPT SPECIFIERS	2-6
2.7	ARRAY SUBSCRIPT SPECIFIERS	2-7
2.8	STRUCTURE SUBSCRIPT SPECIFIERS	2-8
2.9	PRECISION CONVERSION	2-9
2.10	STATIC INITIALIZATION FLOW SPECIFIERS	2-9
2.11	ARGUMENT LIST SPECIFIERS	2-10
2.12	I/O SPECIFIERS	2-12
2.13	SUBPROGRAM SPECIFIERS	2-13
2.14	Auxiliary SHAPING FUNCTION SPECIFIERS	2-17
2.15	STRUCTURE OPERATIONS	2-18
2.16	BIT STRING OPERATIONS	2-19
2.17	CHARACTER OPERATIONS	2-21
2.18	MATRIX OPERATIONS	2-22
2.19	VECTOR OPERATIONS	2-25

2.20 SCALAR OPERATIONS 2-27
2.21 INTEGER OPERATIONS 2-30
2.22 CONDITIONAL OPERATIONS 2-32
2.23 INITIALIZATION OPERATIONS 2-38
2.24 DO FOR SPECIFIERS 2-41

3. HALMAT CONSTRUCTS
3.1 Macroscopic HALMAT Structure 3-1
3.2 Arithmetic Constructs 3-3
3.3 Flow-Control Constructs 3-8
3.4 I/O Constructs 3-15
3.5 Procedure Constructs 3-18
3.6 Normal Function Constructs 3-21
3.7 SHAPING FUNCTION Constructs 3-24
3.8 Initialization Constructs 3-32

Appendix A. HAL360-V1 Symbol and Literal Tables
Appendix B. The HALMAT Instruction Set
Appendix C. Mnemonics of HALMAT Operation Codes
Appendix D. Numerical Codes for Subfield Mnemonics
Appendix E. Shaping Function Information

Addendum

1. INTRODUCTION

1.1 HALMAT and the HAL Compiler

This document describes an intermediate language called HALMAT used in the preliminary version of a compiler written by Intermetrics for HAL language [1],[2]. This compiler, written for the IBM360, will be designated HAL360-V1.

HAL360-V1 has two phases of execution. PHASE I carries out all the syntactic and virtually all the semantic analysis of the source text in one pass. At the same time symbol and literal tables are generated, together with text in the intermediate language HALMAT to be described. PHASE II takes this intermediate text, and in two passes carries out the remainder of the semantic analysis, performs a small amount of code optimization, and generates executable object text in Fortran IV.

The compiler has been structured this way to make its modification to produce object text in different languages relatively simple. It is specifically the aim of this document to impart sufficient knowledge of HALMAT to the reader to enable him to write a new PHASE II for this purpose.

The remainder of this chapter is devoted to preliminary definitions and concepts. Chapter 2 is devoted to a description of the HALMAT Instruction Set. In Chapter 3 various constructs which may appear in the HALMAT text emanating from PHASE I of HAL360-V1 are studied. Appendix A gives some brief information regarding the symbol and literal tables.

1.2 Symbolic Format of HALMAT Instructions

In this section the symbolic, as opposed to physical, format of HALMAT instructions is presented. Chapters 2 and 3 are couched in terms of this symbolic format.

All HALMAT instructions consist of two OPERAND FIELDS a and b, an OPERATOR FIELD r, and an EXTENSION FIELD e as shown in Figure 1.2.1. Depending on the instruction, any of the fields may be empty except for the OPERATOR FIELD. If any field, or part of a field is empty it is shown shaded.

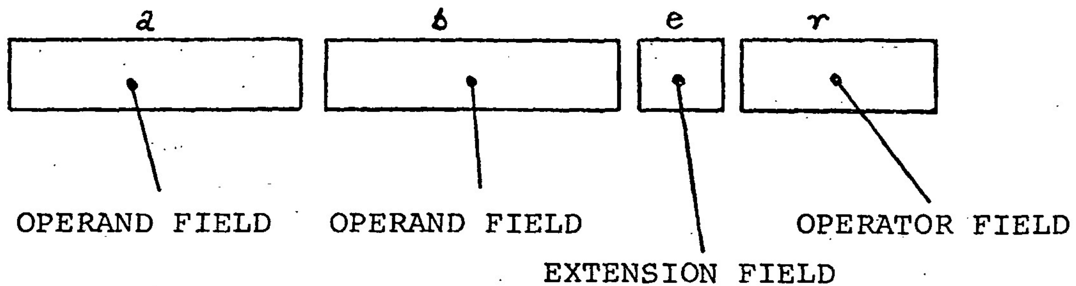


Figure 1.2.1 HALMAT instruction fields

The OPERATOR FIELD is divided into two subfields r_{op} and r_{co}, containing respectively the OPERATION CODE of the HALMAT instruction and a TEXT OPTIMIZER TAG, as shown in Figure 1.2.2.

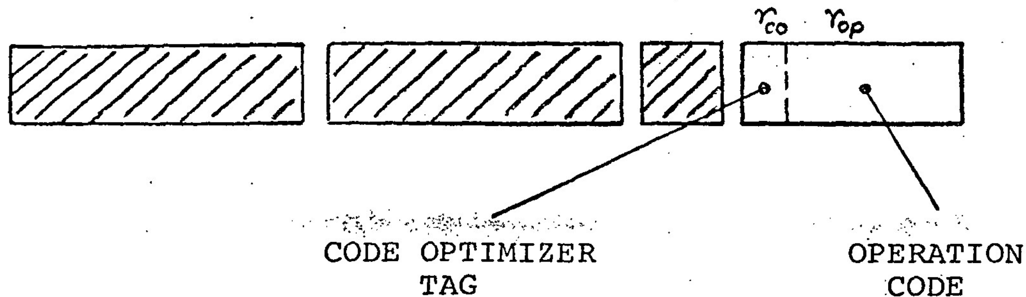


Figure 1.2.2 Contents of the OPERATOR FIELD of a HALMAT instruction

Both the OPERAND FIELDS are subdivided identically: for convenience consider the OPERAND FIELD a, shown in Figure 1.2.3. It has three subfields a_{op}, a_t, and a_q. Generally speaking the subfield a_{op} contains either an operand of the HALMAT instruction, or a pointer to some table where the operand may be found. The subfields a_t and a_q are used to qualify the contents of a_{op}.

Note that if the EXTENSION FIELD e is not empty, then both a_t and b_t are empty; and if either a_t or b_t is not empty, then e is empty. The reason for this will be seen when the physical format of HALMAT instructions is discussed in Section 1.7.

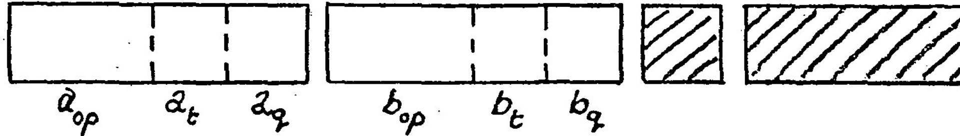


Figure 1.2.3 Subdivision of the OPERAND FIELDS of a HALMAT instruction

NOTATION: A special notation for the contents of a field or subfield is introduced at this point. Suppose that x is a HALMAT instruction field or subfield. Then C{x} denotes the contents of x. If some item of information is spread out over two fields x_1 and x_2 of a HALMAT instruction as if the two fields were contiguous, then the item is denoted by $C\{x_1||x_2\}$, where $x_1||x_2$ is conceptually a field formed by "catenating" x_1 and x_2.

1.3 Symbolic Representation of OPERAND FIELD Contents

A general OPERAND FIELD x will now be defined: let x be either of the OPERAND FIELDS a or b of a HALMAT instruction.

As stated in Section 1.2, the subfield x_{op} contains the primary operand identification, while subfields x_t and x_q contain qualifying information. Ignoring special cases, which are dealt with as they arise in Chapter 2, there are two major classes of operands, VARIABLE OPERANDS, and LABEL OPERANDS. These two classes will be dealt with separately in turn.

VARIABLE OPERANDS

$C\{x_{op}\}$ is either the numerical value of the operand, or more frequently a pointer to an entry in the symbol or literal table. Alternatively $C\{x_{op}\}$ may be the specification of some variable internally generated in PHASE I of the compiler. The subfield x_q is used to distinguish between these different forms of $C\{x_{op}\}$. Table 1.3.1 is a table of mnemonics for the possible $C\{x_q\}$ with explanations of their meanings, and the corresponding symbolic designations for x_{op}. The numerical codes for the mnemonics are given in Appendix D.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

mnemonic	name	$C\{x_{op}\}$	explanation
Ø	-		x_{op} is either empty or is reserved for a special purpose
SYT	symbol-table variable	$\vec{syt}$	$C\{x_{op}\}$ is a pointer into the symbol table
VAC	virtual accumulator	$\vec{line}$	$C\{x_{op}\}$ indexes some previous instruction in the HALMAT text the result of whose execution is used as the operand
LIT	literal	$\vec{lit}$	$C\{x_{op}\}$ is a pointer into the literal table
IMD	immediate	i	$C\{x_{op}\}$ is the actual numerical value of the operand
GLI	global internal	n	$C\{x_{op}\}$ is either zero or an internal flow number, identifying a unique internally generated integer variable
EXV	external variable	exv	generic mnemonic for any one of: SYT, LIT, VAC
EEV	extended external	eev	generic mnemonic for any one of: SYT, LIT, VAC, IMD
VAR	variable	var	generic mnemonic for any one of: SYT, LIT, VAC, IMD, GLI

Table 1.3.1 Mnemonics qualifying the different forms of VARIABLE OPERAND

LABEL OPERANDS

$C\{x_{op}\}$ is either a pointer to the symbol table or to another HALMAT instruction, or an INTERNAL FLOW NUMBER. INTERNAL FLOW NUMBERS are positive integers generated in PHASE I of the compiler primarily to provide unique symbolic addresses for the destinations of branches of execution and other flow-directing constructs of HALMAT. A full explanation of their role is given in Chapter 3. Table 1.3.2 shows mnemonics for the possible $C\{x_q\}$, with explanations of their meanings, and the corresponding symbolic designations for x_{op}. Appendix D gives the numerical codes for these mnemonics.

mnemonic	name	$C\{x_{op}\}$	explanation
INL	internal label	n_f	$C\{x_{op}\}$ is an internal flow number
SYL	symbol-table label	$\vec{syt}$	$C\{x_{op}\}$ is a pointer into the symbol-table
ABL	absolute label	$\vec{line}$	$C\{x_{op}\}$ indexes some subsequent instruction in the HALMAT text
EXL	explicit label	exl	generic mnemonic for either of: INL, SYL
LAB	label	lab	generic mnemonic for any one of: INL, SYL, ABL

Table 1.3.2 Mnemonics qualifying the different forms of LABEL OPERAND

NOTATION: If an asterisk appears in a subfield instead of a mnemonic, then this implies that the subfield is used for a special purpose to be explained.

The contents of the qualifer subfield x_t have no standard form. Each case will be dealt with as it arises in Chapter 2.

1.4 The TEXT OPTIMIZER TAG Subfield

The TEXT OPTIMIZER TAG is used to distinguish the presence or absence of HALMAT constructs which might be transformed or simplified by the optimization pass in PHASE II of the compiler. Table 1.4.1 gives the mnemonics for the different tags in use and explains their significance.

mnemonic	significance of $C\{r_{co}\}$
Ø	empty subfield
E	end-of-expression marker
IW	automatic initialization instruction
Il	static initialization instruction
II	instruction relocation marker
F	function specification instruction
FE	F and E combined
A	subscript allocation instruction
AE	A and E combined
D	arrayness or structureness specification instruction
M	instruction relocation marker
S	subscript specification instruction
SE	S and E combined
N	instruction relocation markers
L	instruction relocation markers

Table 1.4.1 Significance of TEXT OPTIMIZATION TAG Mnemonics

The significance of the mnemonics will be appreciated more fully after Chapters 2 and 3 have been studied.

NOTATION: In Chapter 2 a TEXT OPTIMIZATION TAG mnemonic K denotes that $C\{r_{CO}\}$ of the HALMAT instruction is dependent on the construct of which the instruction is an element.

1.5 Representation of OPERAND TYPES

VARIABLE OPERANDS have an associated OPERAND TYPE, which might for example be integer or matrix or any other of the types of variable to be found in the HAL language. The OPERAND TYPES associated with VARIABLE OPERANDS do not however exhaust all possibilities. There are in addition OPERAND TYPES associated with operands falling outside the two major classes described in Section 1.3. For example the OPERAND FIELD may specify a HAL structure.

In Chapter 2 the OPERAND TYPES are shown symbolically by a field t to the right of the HALMAT instruction fields, as shown in Figure 1.5.1. There is no physical counterpart to this field. Some but not all HALMAT instructions actually contain specifications of the OPERAND TYPES, either explicitly in subfields a_t and b_t or implied by the instruction itself.

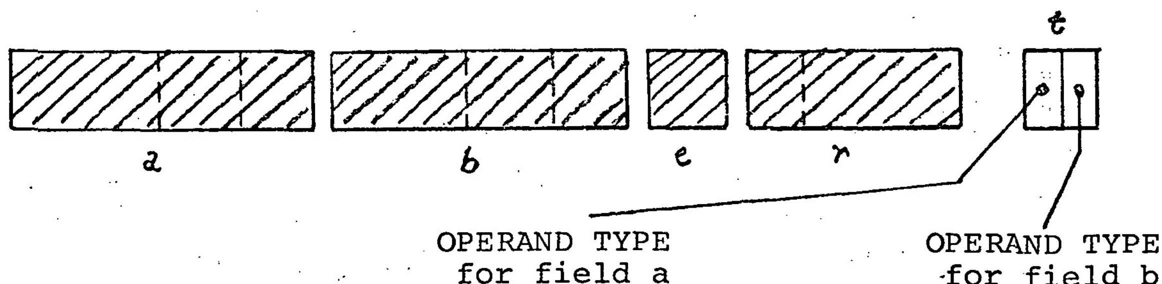


Figure 1.5.1 Specification of OPERAND TYPES

Table 1.5.1 below gives a list of the mnemonics used in the representation of C{t}.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

mnemonic	name	explanation
B	bit	HAL variable types
C	character	
M	matrix	
V	vector	
S	scalar	
I	integer	
L	logical	result-type of a compare instruction
T	structure	HAL structure
Z	any	any of the types B, C, M, V, S, I, T

Table 1.5.1 Mnemonic OPERAND TYPES specifications

The numerical codes for the OPERAND TYPE mnemonics are given in Appendix D.

1.6 Information Not Supplied by an OPERAND FIELD

The OPERAND FIELD does not give all the information necessary to specify a VARIABLE OPERAND. It has already been stated in Section 1.5 that the TYPE of the VARIABLE is only sometimes given. Other information is never supplied.

No precision specification is supplied.

Arrayed variables are not distinguished from unarrayed variables.

No feature distinguishes a variable which is a structure terminal from any other variable.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

All such information required during the object text generation pass of PHASE II of the compiler must therefore originate from the symbol table.

1.7 Physical Format of HALMAT Instructions

The relation between the physical and symbolic formats of HALMAT instructions is given in Figure 1.7.1.

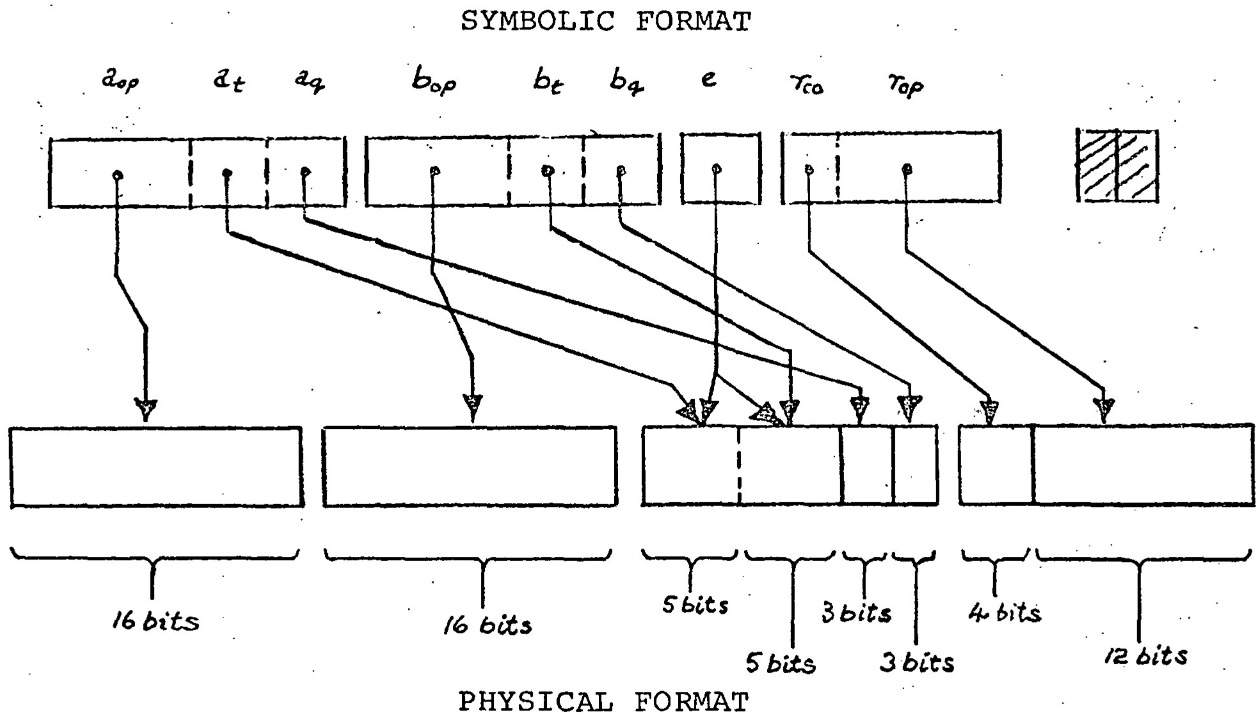


Figure 1.7.1 Physical and Symbolic formats of HALMAT instructions

It is clear that each HALMAT instruction occupies four IBM360 halfwords. It is now clear also why field e cannot be used at the same time as subfields a_t and b_t: they are in fact physically the same subfields.

Between PHASE I and PHASE II of the compiler, HALMAT text is stored in PAGES, one to each block of a disk file[1]. One PAGE of HALMAT text is shown in Figure 1.7.2.

---

[1] Both PHASE I and PHASE II of the HAL360-V1 compiler use a disk file of block size 7200 bytes.

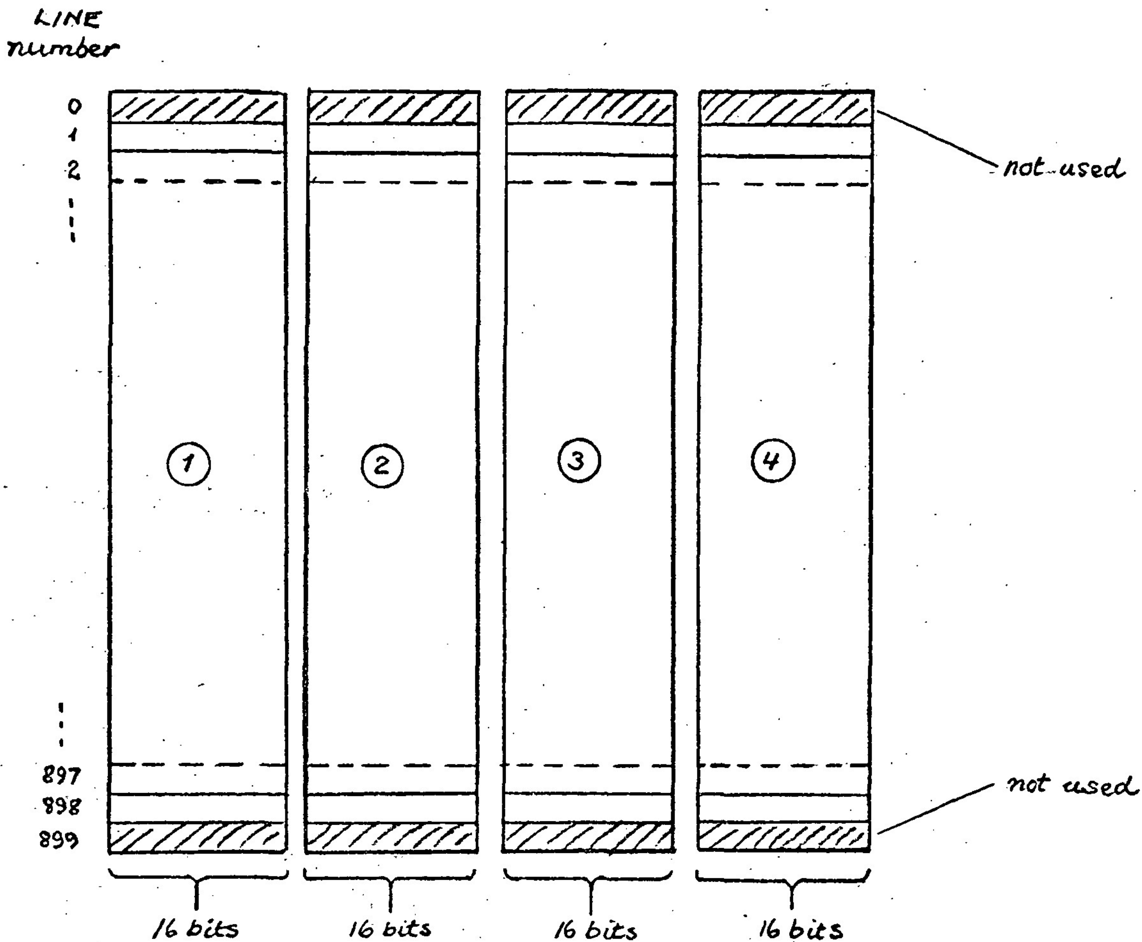


Figure 1.7.2 Format of a PAGE of HALMAT text

Each PAGE of HALMAT text can hold a maximum of 898 LINES of text, one HALMAT instruction per LINE. There are usually a number of blank LINES between the last HALMAT instruction and the end of the PAGE for reasons which will become apparent in Chapter 3. Each of the four columns, LINES 0 thru 899 inclusive of each PAGE, are stored <u>sequentially</u> on the disk file.

1.8 Summary

In this Chapter the concepts and conventions regarding the symbolic representation of HALMAT instructions have been introduced. While significant information has been omitted for the purposes of simplification, the detailed descriptions of individual HALMAT instructions to be presented in the next Chapter will now be intelligible.

2. THE HALMAT INSTRUCTION SET

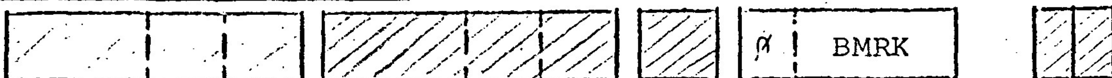
In this Chapter each instruction of the HALMAT Instruction Set is described in turn, and a brief explanation of its function given. HALMAT instructions can be grouped roughly according to their function: the Chapter is organized according to this grouping.

The HALMAT instructions are identified by means of their mnemonics and not by their operation codes. This is because the operation codes are defined arbitrarily in PHASE I of the HAL360-V1 compiler so as to be convenient for use in PHASE II. In any subsequent implementation of HAL where a new PHASE II is used to generate object text, it would be entirely possible, and may in fact be desirable, to redefine these operation codes. The operation codes for the current implementation are given in Appendices B and C.

2.1 CODE MARKERS

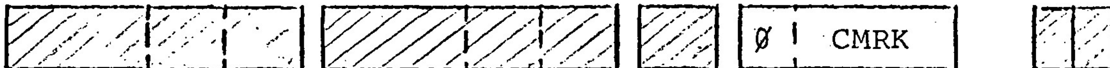
CODE MARKER instructions indicate boundaries between sections of HALMAT text.

BMRK: end of block marker



BMRK marks the end of a page of HALMAT text.

CMRK: end of code marker



CMRK marks the end of the HALMAT text representing an entire HAL program.

SMRK: statement marker

SMRK marks the start of a paragraph of HALMAT text representing one HAL statement. The a_{op} and b_{op} fields are treated as

catenated. $C\{a_{op}||b_{op}\}$ is a HAL statement number. cc is the highest error condition code obtained during the production of the subsequent HALMAT text representing the statement.

2.2 LABELS

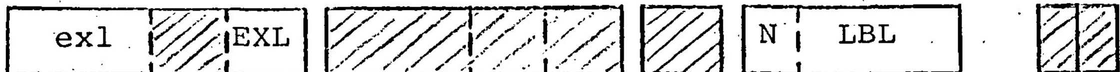
LABELS indicate the destinations of branches in execution, either internally generated, or explicitly translated from the source text.

CLBL: computed branch label

CLBL indicates the destination of a computed branch specified by the instruction CBRA. $C\{a_{op}\}$ is either a symbol-table pointer or an internal flow number.

Each appearance of the instruction CLBL causes the internal flow number $C\{b_{op}\}+1$ to be reserved. n_c denotes that the CLBL is the n_c^{th} alternate destination of the originating CBRA unstruction. n_c lies within the range $0 \leq n_c < 1024$.

LBL: label



LBL indicates the destination of a branch specified by one of the instructions BRA, FBRA or BBRA. $C\{a_{op}\}$ is either a symbol-table pointer or an internal flow number.

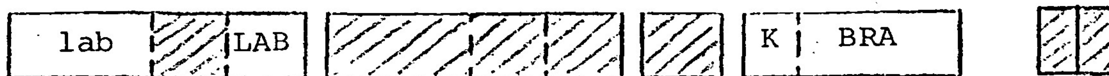
2.3 BRANCHES

BRANCH instructions specify conditional or unconditional branches in the flow of execution of HALMAT text.

BBRA: bit-string branch on false

BBRA specifies a conditional branch to a LBL instruction carrying the same $C\{a_{op}\}$, to be executed if the low-order bit of the bit-string operand specified by $C\{b\}$ is not set.

BRA: unconditional branch

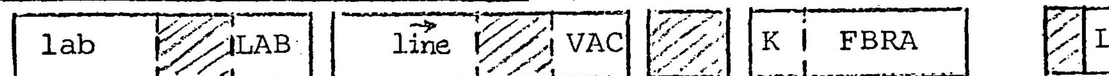


BRA specifies an unconditional branch to either a LBL instruction carrying the same C{a_{op}}, or if C{a_q} ≡ ABL, to the HALMAT instruction indexed by C{a_{op}}. This latter usage only occurs in the HALMAT 'do for' construct.

CBRA: computed branch

CBRA specifies a computed branch to one of a set of CLBL instructions which precede it in the HALMAT text. C{a_{op}} is identical to the C{a_{op}} of each of the CLBL instructions in the set. These CLBL instructions are termed the alternate destinations of the CBRA instruction. n_c denotes that there are n_c alternate destinations, where $1 \leq n_c \leq 1024$. The OPERAND FIELD b specifies the internal integer variable whose value determines which alternate destination is reached at execution time.

FBRA: false branch on condition

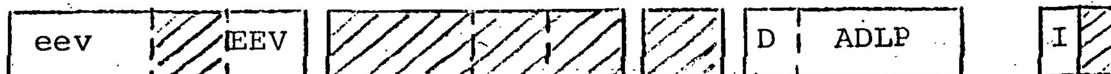


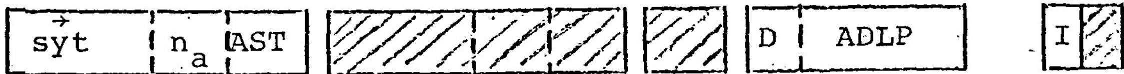
FBRA specifies a conditional branch to either a LBL instruction carrying the same C{a_{op}}, or if C{a_q} ≡ ABL, to the HALMAT instruction indexed by C{a_{op}}. The branch is executed if the value of the logical operand specified by C{b} is FALSE.

2.4 ARRAYNESS and STRUCTURENESS SPECIFIERS

A HAL statement may specify arithmetic or conditional operations to be carried out on arrayed variables. Such a statement is said to have FREE ARRAYNESS. Similarly, a HAL statement may specify arithmetic or conditional operations to be carried out simultaneously on multiple copies of structure terminal variables. Such a statement is said to have FREE STRUCTURENESS. FREE ARRAYNESS and FREE STRUCTURENESS may coexist in one statement.

ADLP: arrayness specifier

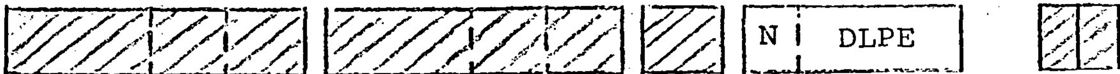




ADLP specifies one dimension of the possibly multidimensional FREE ARRAYNESS. There are two versions of the ADLP instruction. In the first version C{a} specifies a variable whose value is the arrayness in question. In the second version the value of the arrayness is that of the n_a^{th} array dimension of the symbol-table variable referenced by C{a_{op}}. C{a_q} ≡ AST distinguishes this special use of the field a.

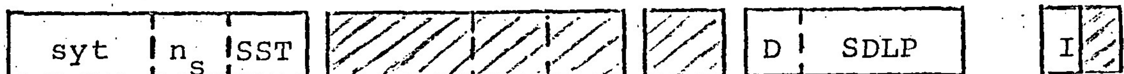
There is a further complication in the first version of the ADLP instruction. If C{a_q} ≡ VAC and C{a_{op}} indexes an ATSB instruction then the operand specified is the result of subtracting the operand specified by C{b} of the ATSB instruction from the operand specified by C{a} of that instruction. On the other hand if C{a_q} ≡ VAC and C{a_{op}} indexes an AASB instruction then the operand specified is the operand specified by C{a} of the AASB instruction.

DLPE: end of array and structureness specification



DLPE is used as a terminator immediately following a sequence of ADLP and SDLP instructions.

SDLP: structureness specifier



SDLP specifies one dimension of the possibly multidimensional FREE STRUCTURENESS. These are two versions of the SDLP instruction. In the first version C{a} specifies a variable whose value is the structureness in question. In the second version the value of the structureness is the number of copies of the n_s^{th} multiple-copy minor structure starting from the structure terminal variable referenced by C{a_{op}} and working backwards towards Level 1 of the structure. C{a_q} = SST distinguishes this special use of the field a.

There is a further complication in the first version of the SDLP instruction. If $C\{a_q\} \equiv$ VAC and $C\{a_{op}\}$ indexes a STSB instruction, then the operand specified is the result of subtracting the operand specified by C{b} of the STSB instruction from the operand specified by C{a} of that instruction. On the other hand if $C\{a_q\} \equiv$ VAC and $C\{a_{op}\}$ of the SDLP instruction indexes a SASB instruction, then the operand specified is the operand specified by C{a} of the SASB instruction.

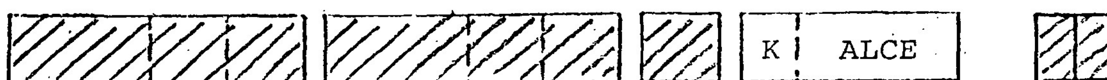
2.5 SUBSCRIPT ALLOCATORS

A SUBSCRIPT ALLOCATOR instruction identifies a variable to which subscripts are to be attached. The subscripts themselves are identified by SUBSCRIPT SPECIFIER instructions which immediately follow the ALLOCATOR instruction.

ALC: allocate subscript header

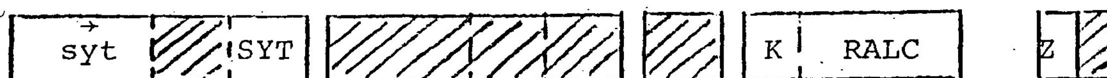
ALC specifies the symbol-table variable to which the subscript identified by the HALMAT text following the instruction are to be attached. It is used for all subscripted variables in the HAL source text except those which appear on the left-hand side of assignment statements, or in assign parameter lists.

ALCE: allocate subscript end



ALCE terminates the list of SUBSCRIPT SPECIFIER instructions which immediately follows a RALC or an ALC instruction.

RALC: receiver allocate subscript specifier



RALC, like the ALC instruction, specifies the symbol-table variable to which the subscripts identified by the HALMAT text following the instruction are to be attached. However, RALC is only used for subscripted variables in the HAL source text appearing on the left-hand side of assignment statements (RECEIVERS),

or in assign parameter lists.

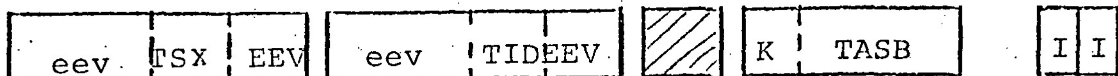
2.6 TERMINAL SUBSCRIPT SPECIFIERS

The following instructions identify the terminal subscripts of a matrix, vector, character or bit-string symbol-table variable. Special uses are made of the subfields a_t and b_t of these instructions.

Vectors, characters and bit-strings only have one TERMINAL DIMENSION. For these types $C\{b_t\} = 1$, except for bit-strings, where $C\{b_t\} = \emptyset$. Matrices have two TERMINAL DIMENSIONS, the first being the ROW DIMENSION and the second being the COLUMN DIMENSION: $C\{b_t\} = 2$ is used to denote a ROW subscript, and $C\{b_t\} = 1$ a COLUMN subscript. If for some TERMINAL SUBSCRIPT SPECIFIER $C\{b_t\} = n_t$ then n_t is called the $n_t{}^{th}$ TERMINAL SUBSCRIPT. The related TERMINAL DIMENSION would be called the $n_t{}^{th}$ TERMINAL DIMENSION. This use of b_t is denoted by the mnemonic $C\{b_t\} \equiv$ TID.

Use of the subfield a_t is made in the special case $C\{a_q\} \equiv$ IMD and $C\{a_{op}\} = \emptyset$, or when some other associated instruction makes such use of its own subfields a_q and a_{op}. If in this case $C\{a_t\} = n_t$ then $C\{a\}$ specifies as operand the $n_t{}^{th}$ TERMINAL DIMENSION of the variable specified in the proceding RALC or ALC instruction. This use of a_t is denoted by the mnemonic $C\{a_t\}$ TSX. Apart from this case $C\{a\}$ is as normally specified for a VARIABLE OPERAND.

TASB: terminal at- subscript specifier



TASB identifies a terminal at-subscript according to the conventions described. If the at-subscript is denoted symbolically by 'α AT β', then $C\{a\}$ specifies the integer operand β, and $C\{b_{op}\}$, $C\{b_q\}$ the integer operand α.

TIDX: terminal index specifier

TIDX identifies a terminal subscript according to the conventions described. The operand specified by $C\{a\}$ is always of integer type.

TTSB: terminal to-subscript specifier

TTSB identifies a terminal to-subscript according to the conventions described. If the to-subscript is denoted symbolically by 'α TO β', then C{a} specifies the integer operand α, and $C\{b_{op}\}$, $C\{b_q\}$ the integer operand β.

2.7 ARRAY SUBSCRIPT SPECIFIERS

The following instructions identify the array subscripts of an array variable of any type. Special uses are made of the subfields a_t and b_t of these instructions.

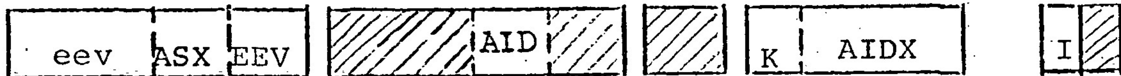
The ARRAY DIMENSIONS of an array variable may be indexed from one, right to left as they appear in the array declaration in the HAL source text. $C\{b_t\}$ is used to specify the ARRAY DIMENSION to which the ARRAY SUBSCRIPT corresponds: if $C\{b_t\} = n_a$, then the ARRAY SUBSCRIPT corresponds to the $n_a{}^{th}$ ARRAY DIMENSION of the variable. This use of b_t is denoted by the mnemonic $C\{b_t\} \equiv$ AID.

Use of the subfield a_t is made in the special case $C\{a_q\} \equiv$ IMD and $C\{a_{op}\} = \emptyset$, or when some other instruction makes such use of its own subfields a_q and a_{op}. If in this case $C\{a_t\} = n_a$ then C{a} specifies as operand the $n_a{}^{th}$ ARRAY DIMENSION of the array variable specified in the preceding RALC or ALC instruction. This use of a_t is denoted by the mnemonic $C\{a_t\} \equiv$ ASX. Apart from this case C{a} is as normally specified for a VARIABLE OPERAND.

AASB: array at-subscript specifier

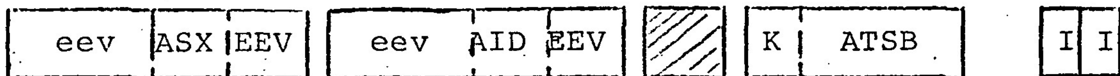
AASB identifies an array at-subscript according to the conventions described. If the at-subscript is denoted symbolically by 'α AT β', then C{a} specifies the integer operand β, and $C\{b_{op}\}$, $C\{b_q\}$ the integer operand α.

AIDX: array index specifier



AIDX identifies an array subscript according to the conventions described. The operand specified by C{a} is always of integer type.

ATSB: array to-subscript specifier



ATSB identifies an array to-subscript according to the conventions described. If the to-subscript is denoted symbolically by 'α TO β', then C{a} specifies the integer operand α, and C{b_{op}}, C{b_q} the integer operand β.

2.8 STRUCTURE SUBSCRIPT SPECIFIERS

The following instructions identify the structure-copy subscripts of structure terminal variables. Special uses are made of the subfields a_t and b_t of these instructions.

The STRUCTURE-COPY DIMENSIONS of a structure terminal variable may be indexed from one backwards up the structure tree to the major structure name. Each minor structure, and the major structure, are included in the indexing only if they have multiple copies. C{b_t} is used to specify the STRUCTURE-COPY DIMENSION to which the STRUCTURE SUBSCRIPT corresponds: if C{b_t} = n_s, then the STRUCTURE SUBSCRIPT corresponds to the n_sth STRUCTURE-COPY DIMENSION of the variable. This use of b_t is denoted by the mnemonic C{b_t} ≡ SID.

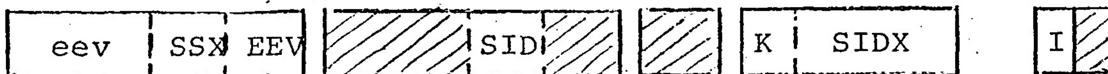
Use of the subfield a_t is made in the special case C{a_q} ≡ IMD and C{a_{op}} = Ø or when some other associated instructions makes such use of its own subfields a_q and a_{op}. If in this case C{a_t} = n_s, then C{a} specifies as operand the n_sth STRUCTURE-COPY DIMENSION of the structure terminal variable specified in the preceding RALC or ALC instruction. This use of a_t is denoted by the mnemonic C{a_t} ≡ SSX. Apart from this case C{a} is as normally specified for a VARIABLE OPERAND.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

SASB: structure at-subscript specifier

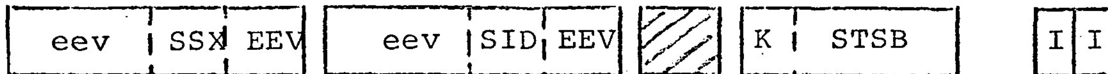
SASB identifies a structure at-subscript according to the conventions described. If the at-subscript is denoted sumbolically by 'α AT β', then C{a} specifies the integer operand β, and C{b_{op}},C{b_q} the integer operand α.

SIDX: structure index specifier



SIDX identifies a structure subscript according to the conventions described. The operand specified by C{a} is always of integer type.

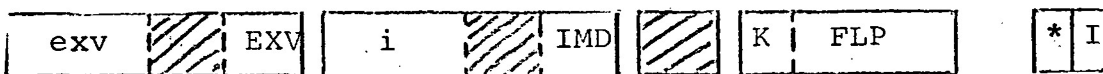
STSB: structure to-subscript specifier



STSB identifies a structure to-subscript according to the conventions described. If the to-subscript is denoted symbolically by 'α TO β', then C{a} specifies the integer operand α, and C{b_{op}}, C{b_q} the integer operand β.

2.9 PRECISION CONVERSION

FLP: precision converter



FLP specifies the conversion of the matrix, vector or scalar operand specified by C{a} to the precision (number of decimal digits) specified by the integer specified by C{b}.

2.10 STATIC INITIALIZATION FLOW SPECIFIERS

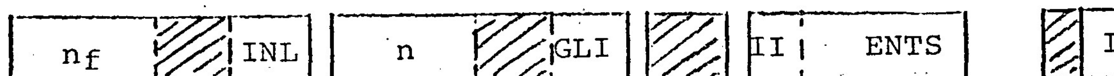
The following instructions are used to control the execution of initialization instructions for variables with the STATIC

attribute.

ENDS: end of static bypass blocks

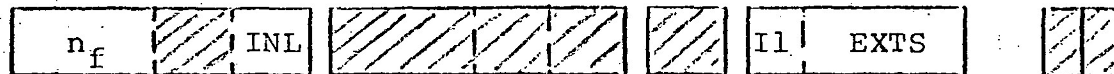
ENDS indicates the end of a sequence of HALMAT constructs for specifying the initialization of STATIC variables. The integer operand specified by C{b} is supplied for use as a flag to prevent reinitialization of the STATIC variables subsequent to the first initialization.

ENTS: enter static bypass block



ENTS prefaces the HALMAT construct for specifying the initialization of a STATIC variable. In effect it is like a conditional branch to an EXTS instruction carrying the same C{a}, dependent on a flag which is the integer operand specified by C{b}.

EXTS: exit static bypass block

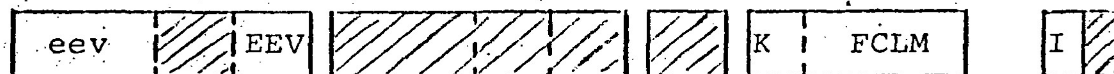


EXTS ends the HALMAT construct for specifying the initialization of a STATIC variable. It is in effect the destination of a preceding ENTS instruction with the same C{a}.

2.11 ARGUMENT LIST SPECIFIERS

The following instructions are used in the specification of argument lists in function and procedure invocations and definitions, and in I/O statements. Where an instruction is only found in a particular variety of argument list, a remark to that effect is given in its description.

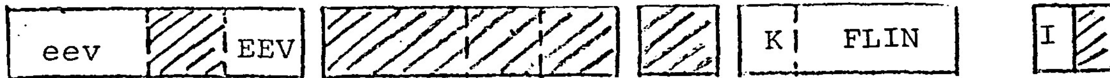
FCLM: column I/O control specifier



INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

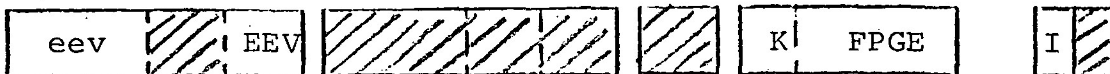
FCLM specifies a move of the read- or write-mechanism to column k, where k is the value of the integer operand specified by C{a}. (I/O argument lists only)

FLIN: line I/O control specifier



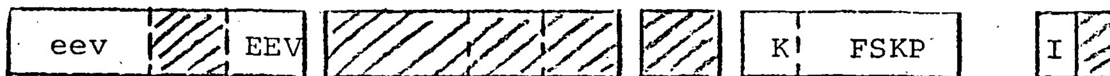
FLIN specifies a move of the read- or write-mechanism to line k, where k is the value of the integer operand specified by C{a}. (I/O argument lists only).

FPGE: page I/O control specifier



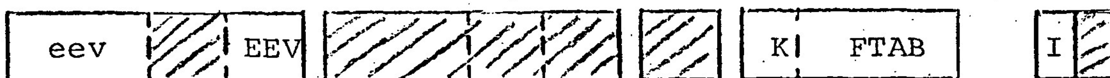
FPGE specifies a move of the read- or write-mechanism by k pages, where k is the value of the integer operand specified by C{a}. (I/O argument lists only)

FSKP: skip I/O control specifier



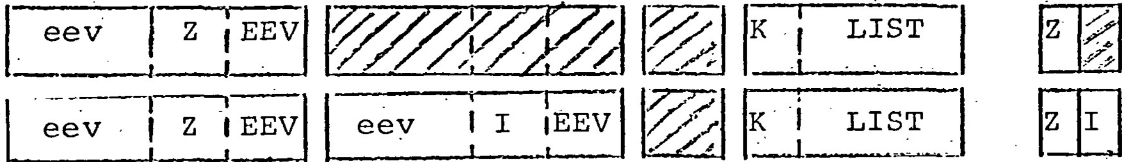
FSKP specifies a move of the read- or write-mechanism by k lines, where k is the value of the integer operand specified by C{a}. (I/O argument lists only)

FTAB: tab I/O control specifier



FTAB specifies a move of the read- or write-mechanism by k columns, where k is the value of the integer operand specified by C{a}. (I/O argument lists only)

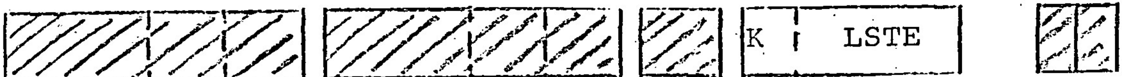
LIST: subprogram and I/O argument



The LIST instruction has two versions: the first version specifies an argument in any argument list; the second version is used only to specify repeated arguments in shaping and conversion function call lists. In both versions C{a} specifies the operand which is the list argument. C{a_t} specifies the type of the argument, except in procedure and function definition lists, when a_t is not used.

In the second version of LIST, C{b} specifies an integer operand giving the number of times the argument specified by C{a} is to be repeated. C{b_t} specifies the type.

LSTE: subprogram and I/O argument list end



LSTE indicates the end of any argument list.

2.12 I/O SPECIFIERS

I/O SPECIFIERS are used to specify read and write operations and file operations.

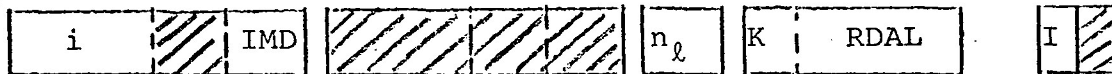
FASN: file assignment

FASN is used as an assignment instruction when in the HAL source text the 'FILE' pseudo-variable appears on the right-hand side of an assignment statement. A read is specified from the file identified by the FILE instruction indexed by C{b} into the operand specified by C{a}. If n_m = 1 then the instruction is followed by one or more further FASN instructions completing a multiple assignment. Otherwise n_m = Ø.

FILE: file I/O specifier

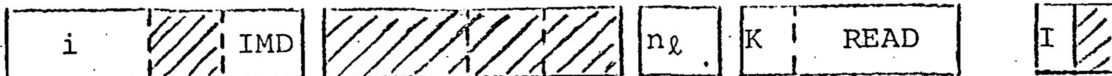
FILE specifies the device and record numbers in a file read or write operation. C{a} is the device number. The record number is the value of the integer operand specified by C{b}.

RDAL: read-all header



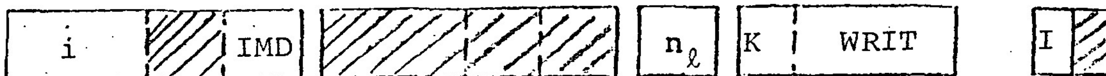
RDAL is the header of an argument list for a 'READ_ALL' statement. n_ℓ is the number of arguments specified by the following HALMAT instructions, including I/O control specifiers. C{a} is the device number of the read-mechanism.

READ: read header



READ is the header of an argument list for a 'READ' statement. n_ℓ is the number of arguments specified by the following HALMAT instructions, including I/O control specifiers. C{a} is the device number of the read-mechanism.

WRIT: write header

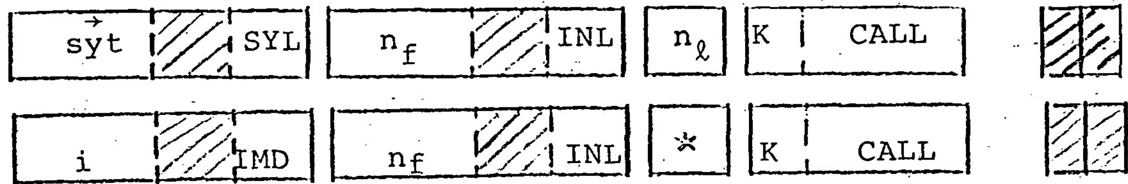


WRIT is the header of an argument list for a 'WRITE' statement. n_ℓ is the number of arguments specified by the following HALMAT instructions, including I/O control specifiers. C{a} is the device number of the write-mechanism.

2.13 SUBPROGRAM SPECIFIERS

SUBPROGRAM SPECIFIERS are instructions used to define subprograms and programs, to invoke them, and to return from them. Subprograms include functions, procedures, tasks and update blocks.

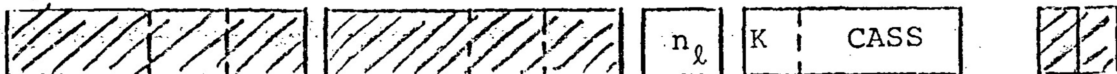
CALL: procedure call header



There are two versions of the CALL instruction, each version being used for a different purpose. The first version is used to indicate procedure invocation. $C\{a_{op}\}$ points to a procedure name in the symbol table. n_ℓ is the number of input operands specified by following LIST instructions. $C\{b_{op}\}$ is an internal flow number which together with the internal flow number $C\{b_{op}\}+1$ is used in the generation of object text.

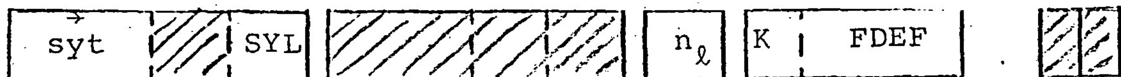
The second version of the CALL instruction specifies the invocation of a runtime error package. $C\{a_{op}\}$is an integer used to distinguish different invocations. C{b} has the same function as in the first version. If the CALL comes at the end of a user function body, C{e} = 1; otherwise C{e} = 0.

CASS: assign parameter list header



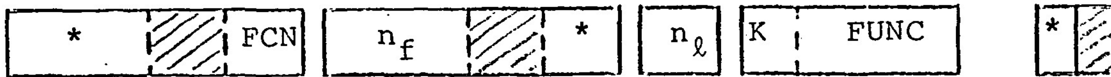
CASS is used as a header for the 'ASSIGN' output argument list specification instructions in a procedure invocation or definition. If there are also input arguments it separates the last LIST instruction specifying an input argument from the first LIST instruction specifying an output argument n_ℓ is the number of output arguments specified by the following LIST instructions.

FDEF: function definition header



FDEF is used to indicate a user function definition. $C\{a_{op}\}$ points to a function name in the symbol table. n_ℓ is the number of input parameters specified by the following LIST instructions.

FUNC: function call header



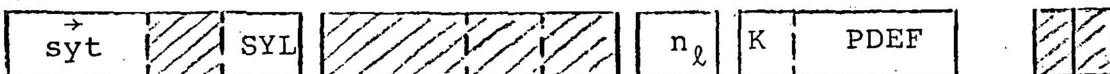
FUNC is used to specify invocations to user functions and to HAL built-in functions. n_ℓ is the number of input arguments specified by the following LIST instructions. $C\{b_{op}\}$ is an internal flow number which together with the internal flow number $C\{b_{op}\}+1$ is used in the generation of object text. Note that $C\{b_q\}$ is not used to qualify $C\{b_{op}\}$ in this instance. For all functions except 'BIT' and 'CHARACTER' subfield b_q is empty. For 'BIT' and 'CHARACTER', $C\{b_q\}$ specifies the radix of conversion, either binary, octal or hexadecimal: see Appendix E for the actual format.

The operand field a can have a number of different uses depending on the type of function specified. $C\{a_{op}\} \equiv$ FCN qualifies $C\{a_{op}\}$ as specifying a function. If $C\{a_{op}\} \leq \emptyset$ then $C\{a_{op}\}$ denotes a SHAPING FUNCTION. A table of SHAPING FUNCTIONS and their respective $C\{a_{op}\}$ values is given in Appendix E. If $C\{a_{op}\} > \emptyset$ then $C\{a_{op}\}$ points to the name of the function in the symbol table. Since NORMAL BUILT-IN FUNCTIONS as well as user functions have entries in the symbol table, all functions except SHAPING FUNCTIONS are included in this case.

MDEF: program definition header

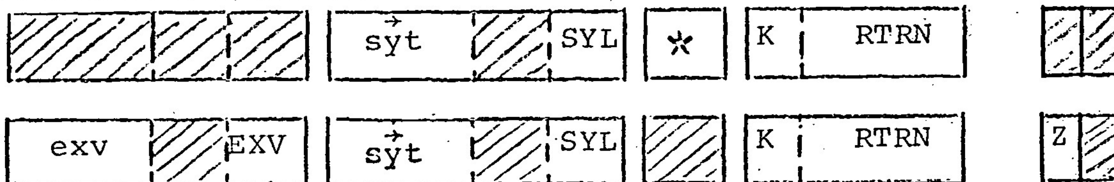
MDEF defines the name of a program. $C\{a_{op}\}$ points to the name in the symbol table.

PDEF: procedure definition header



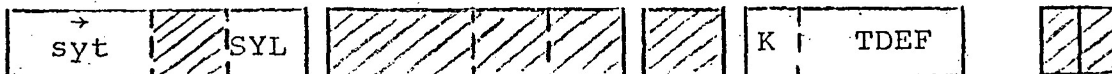
PDEF indicates the start of a procedure definition. $C\{a_{op}\}$ points to the procedure name in the symbol table. n_ℓ is the number of input parameters specified by the following LIST instructions.

RTRN: subprogram return



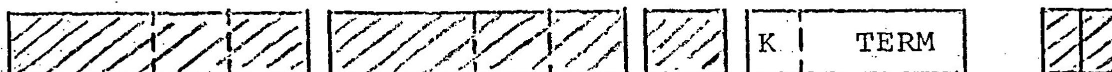
RTRN specifies a return from a subprogram. The first version is used for tasks and procedures and the second for functions. C{b_{op}} for either version points to the name of the subprogram in the symbol table. In the second version only, C{a} specifies the operand which is returned as the result of the function. This operand may be of any type. C{e} = 1 if and only if a closing return from a procedure is specified.

TDEF: task definition header



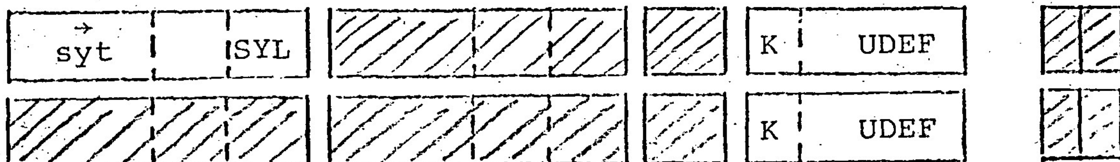
TDEF indicates the start of a task definition. C{a_{op}} points to the name of the task in the symbol table.

TERM: terminate program execution



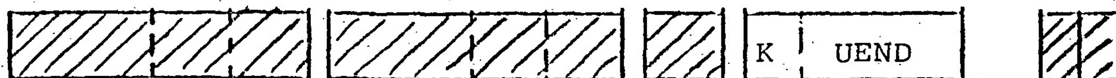
TERM indicates the termination of execution of a program.

UDEF: update block definition



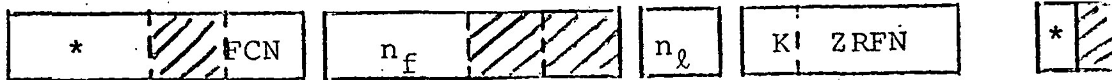
UDEF indicates the start of update block. If the update block is named, C{a_{op}} points to the name in the symbol table. If the update block is unnamed then the a field is empty.

UEND: update block end



UEND indicates the end of an update block.

ZRFN: receiver function invocation



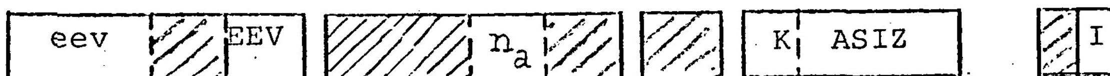
ZRFN is used to specify invocations of pseudo-variables, which are functions appearing as RECEIVERS in assignment statements. Specifically excluded from this class is the 'FILE' pseudovariable which is specified by a FILE instruction. n_ℓ is number of arguments specified by the following LIST instructions. $C\{b_{op}\}$ is an internal flow number which together with the internal flow number $C\{b_{op}\}+1$ is used in the generation of object text. $C\{a_{op}\}<\emptyset$ indicate the pseudo-variable to be one of the class of SHAPING FUNCTIONS. (all pseudo-variables are SHAPING FUNCTIONS). Appendix E gives a table of SHAPING FUNCTIONS and their respective values of $C\{a_{op}\}$.

2.14 Auxiliary SHAPING FUNCTION SPECIFIERS

SHAPING FUNCTIONS require specifications over and above that provided by the FUNC instruction and the following LIST instructions. Firstly, the function itself may be subscripted. Secondly, the function has an arbitrary number of arguments.

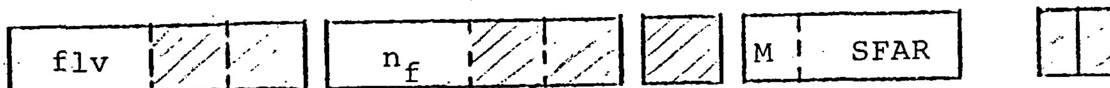
An argument of a SHAPING FUNCTION may be another SHAPING FUNCTION so that the nesting depth or CURRENT FUNCTION LEVEL needs to be specified. In certain of the SHAPING FUNCTION instructions, $C\{a_{op}\}$ gives this information.

ASIZ: arrayness of shaping function



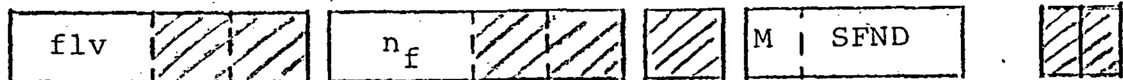
ASIZ specifies one ARRAY DIMENSION of the SHAPING FUNCTION specified by the related FUNC or ZRFN instruction. $C\{a\}$ specifies the integer operand supplying the dimension. If $C\{a\} \equiv$ IMD and $C\{a_{op}\}=-1$ then the operand supplying the dimension could not be determined during PHASE I. If the arrayness specified is the $n_a{}^{th}$ arrayness of the function (corresponding to the $n_a{}^{th}$ array subscript right to left in the HAL source text), then $C\{b_t\}=n_a$.

SFAR: shaping function argument separator



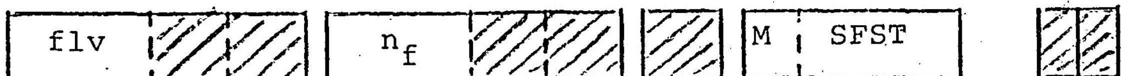
SFAR separates the HALMAT text corresponding to each argument of the SHAPING FUNCTION. flv is the CURRENT FUNCTION LEVEL. $C\{b_{op}\}$ is the same internal flow number as specified by $C\{b_{op}\}$ of the FUNC instruction for the function.

SFND: end of shaping function specification



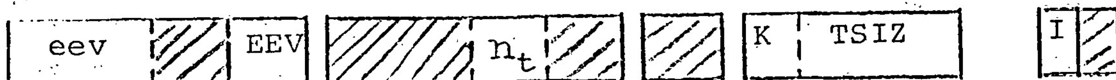
SFND marks the end of the HALMAT text specifying the invocation of a SHAPING FUNCTION. flv is the CURRENT FUNCTION LEVEL. $C\{b_{op}\}$ is the same internal flow number as specified by $C\{b_{op}\}$ of the FUNC instruction for the function.

SFST: start of shaping function specification



SFST indicates the start of the specification of a SHAPING FUNCTION invocation. flv is the CURRENT FUNCTION LEVEL. $C\{b_{op}\}$ is the same internal flow number as specified by $C\{b_{op}\}$ of the FUNC instruction for the function.

TSIZ: terminal size of shaping function



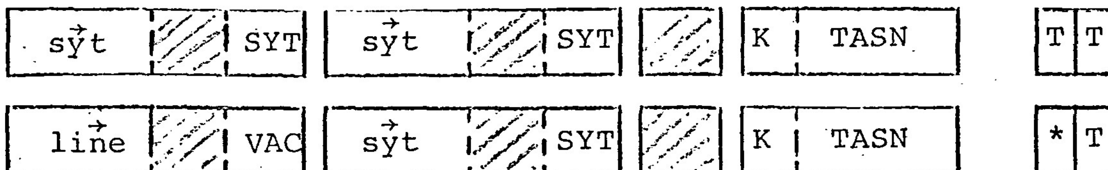
TSIZ specifies a TERMINAL DIMENSION of the SHAPING FUNCTION specified by the related FUNC instruction. C{a} specifies the integer operand supplying the dimension. If $C\{a_q\} \equiv$ IMD and $C\{a_{op}\}$=-1 then the operand supplying the dimension could not be determined during PHASE I. For the function 'BIT', and the pseudo-variable 'CONTENT' n_t=Ø. For the functions 'VECTOR' and 'CHARACTER' n_t=1. For the function 'MATRIX', n_t=2 denotes the specification of a ROW DIMENSION, and n_t=1 the specification of a COLUMN DIMENSION.

2.15 STRUCTURE OPERATIONS

STRUCTURE OPERATIONS are operations on major or minor structures. It is assumed that object text produced from the

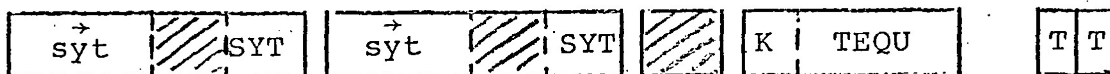
following instructions includes testing the structure templates for a match whenever such tests have to be done at run-time rather than during PHASE I.

TASN: structure assign



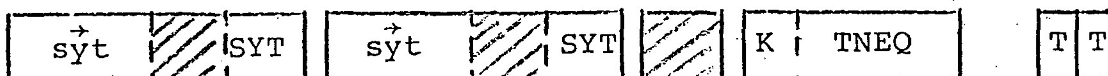
There are two versions of the TASN instruction. In both versions C{b} specifies a structure operand to be assigned. In the first version it is assigned to the structure operand specified by C{a}. In the second version it is assigned to a 'FILE' pseudo-variable: C{a_{op}} indexes a FILE instruction identifying the file.

TEQU: structure equal



The result of the TEQU instruction is TRUE if the values of the terminal variables of the structure operand specified by C{a} are equal to the corresponding terminal variables of the structure operand specified by C{b}, and FALSE otherwise.

TNEQ: structure not equal



The result of the TNEQ instruction is FALSE if the values of the terminal variables of the structure operand specified by C{a} are equal to the corresponding terminal variables of the structure operand specified by C{b}, and TRUE otherwise.

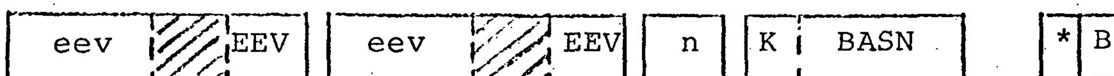
2.16 BIT STRING OPERATIONS

BIT STRING OPERATIONS are operations which have bit strings as results, with the addition of the assignment of bit strings.

BAND: bit string and

The result of the BAND instruction is the logical 'and' of the bit string operands specified by C{a} and C{b}. The shorter of the two operands is padded on the left with binary zeroes to match the length of the longer.

BASN: bit string assign



BASN assigns the bit string operand specified by C{b} to the operand specified by C{a}. The RECEIVER operand may be of bit string, integer, character, or scalar type, or it may refer to the FILE pseudo-variable.

If the RECEIVER operand is a bit string which is longer than the right-hand side operand, it is padded on the left with binary zeroes. If the RECEIVER bit string is shorter, then the most significant bits are truncated.

If the RECEIVER operand is an integer then the right-hand operand is first padded or truncated as explained to produce a bit string with the same number of bits as an integer (implementation dependent). That bit pattern then becomes the value of the integer RECEIVER.

If the RECEIVER operand is a character then the bit string right-hand side is first converted to an integer as described. The decimal representation of the integer is then converted to a character, and if the RECEIVER operand does not have the 'VARYING' attribute, padded on the right with blanks.

If the RECEIVER operand is a scalar then the bit string right-hand side is first converted to an integer as described. The integer is then converted to a scalar.

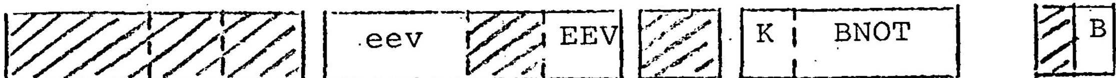
If the RECEIVER is the 'FILE' pseudo-variable then the assignment calls for the bit string operand specified by C{b} to be written on the file specified by the FILE instruction indexed by C{a_{op}}.

In any of these cases, if n=1 then the BASN instruction is followed by one or more further BASN instructions completing a multiple assignment. Otherwise n=Ø.

BCAT: bit string catenation

The result of the BCAT instruction is the left catenation of the bit string operand specified by C{a} to the bit string operand specified by C{b}.

BNOT: bit string complement



The result of the BNOT instruction is the complement of the bit string operand specified by C{b}.

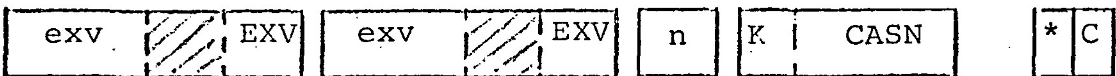
BOR: bit string or

The result of the BOR instruction is the logical 'or' of the bit string operands specified by C{a} and C{b}. The shorter of the two operands is padded on the left with binary zeroes to match the length of the longer.

2.17 CHARACTER OPERATIONS

CHARACTER OPERATIONS are those operations which have character results, with the addition of the assignment of characters.

CASN: character assign



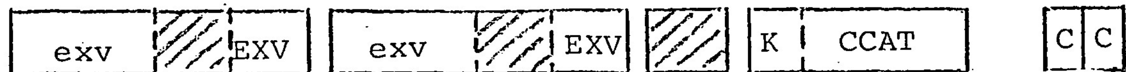
CASN assigns the character operand specified by C{b} to the RECEIVER operand specified by C{a}. The RECEIVER operand may be a character operand or the 'FILE' pseudo-variable.

If the RECEIVER operand is of character type and does not have the 'VARYING' attribute, the right-hand side is padded with blanks or truncated on the right.

If C{a} specifies the 'FILE' pseudo-variable, then the right-hand side operand is written on the file specified by the FILE instruction indexed by C{a_{op}}.

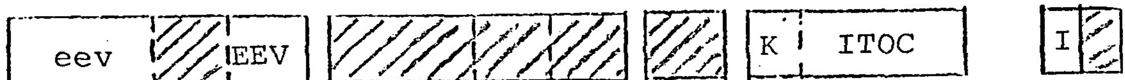
In any of the above cases, if n=1 than the CASN instruction is followed by one or more further CASN instructions completing a multiple assignment. Otherwise n=Ø.

CCAT: character catenate



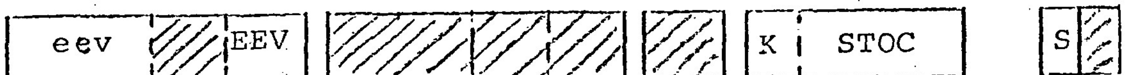
The result of the CCAT instruction is the left catenation of the character operand specified by C{a} to the character operand specified by C{b}.

ITOC: integer to character conversion



ITOC specifies the conversion of the decimal representation of the integer operand specified by C{a} to character type.

STOC: scalar to character conversion



STOC specifies the conversion of the decimal representation of the scalar operand specified by C{a} to character type.

2.18 MATRIX OPERATIONS

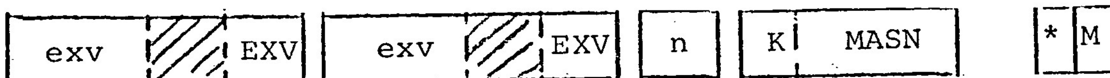
MATRIX OPERATIONS are defined as those arithmetic operations which have a result of matrix type, with the addition of the assignment of matrix operands. VARIABLE OPERANDS of matrix, vector, scalar or integer types occur. In the cases where PHASE I cannot check for the compatibility of matrix or vector operands, the object text generated for the instructions concerned must include run-time compatibility checking.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

MADD: matrix add

The result of the MADD instruction is the sum of the two matrix operands specified by C{a} and C{b}.

MASN: matrix assign



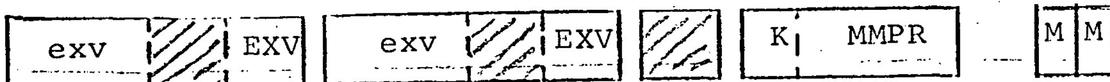
MASN assigns the matrix operand specified by C{b} to the RECEIVER operand specified by C{a}. If n=1 then the MASN instruction is followed by one or more further MASN instructions representing a multiple assignment. Otherwise n=Ø.

The RECEIVER operand may either be of matrix type, or it may specify the 'FILE' pseudo-variable in which case writing on the file specified by the FILE instruction indexed by $C\{a_{op}\}$ is indicated.

MINV: matrix inversion

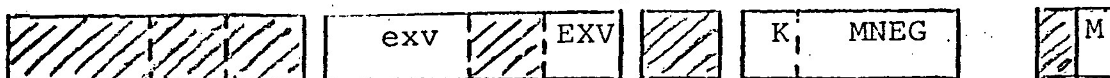
The result of the MINV instruction is the inverse of the square matrix operand specified by C{a} if the integer operand specified by C{b} has a value of -1. The format of the instruction treats inversion as if it were exponentiation.

MMPR: matrix-matrix multiply



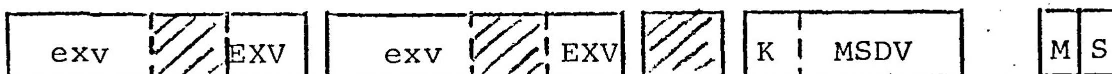
The result of the MMPR instruction is the matrix operand specified by C{a} postmultiplied by the Matrix operand specified by C{b}.

MNEG: matrix negate



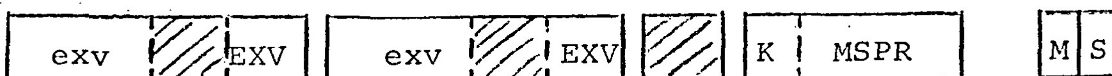
The result of the MNEG instruction is the negative of the matrix operand specified by C{b}.

MSDV: matrix-scalar divide



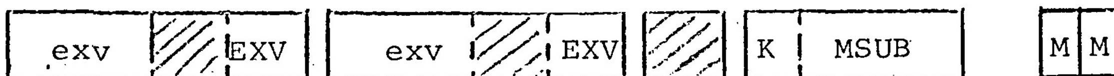
The result of the MSDV instruction is the matrix operand specified by C{a} divided by the scalar operand specified by C{b}.

MSPR: matrix-scalar multiply



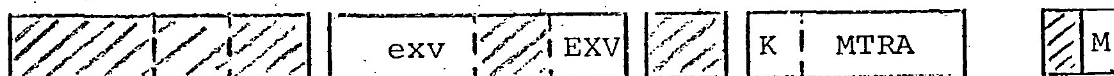
The result of the MSPR instruction is the matrix operand specified by C{a} multiplied by the scalar operand specified by C{b}.

MSUB: matrix subtract



The result of the MSUB instruction is the matrix operand specified by C{b} subtracted from the matrix operand specified by C{a}.

MTRA: matrix transpose



The result of the MTRA instruction is the transpose of the matrix operand specified by C{b}.

VVPR: vector outer product

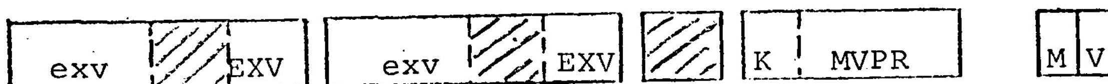
The result of the VVPR instruction is the matrix formed by the vector operand specified by C{a} postmultiplied by the vector operand specified by C{b}.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

2.19 VECTOR OPERATIONS

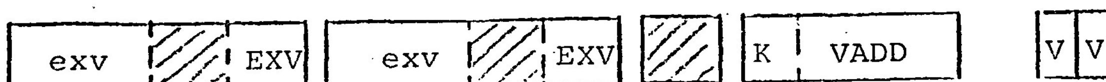
VECTOR OPERATIONS are defined as those arithmetic operations which have a result of vector type, with the addition of the assignment of vector operands. VARIABLE OPERANDS of matrix, vector or scalar type may occur. In the cases where PHASE I cannot check for the compatibility of matrix or vector operands, the object text generated for the instructions concerned must include run-time compatibility checking.

MVPR: matrix-vector multiply



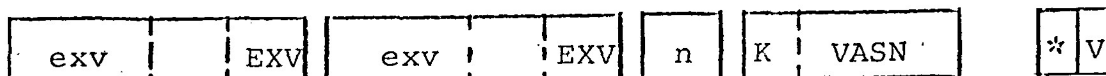
The result of the MVPR instruction is the matrix operand specified by C{a} postmultiplied by the vector operand specified by C{b}.

VADD: vector add



The result of the VADD instruction is the sum of the two vector operands specified by C{a} and C{b}.

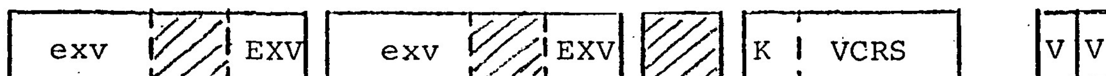
VASN: vector assign



VASN assigns the matrix operand specified by C{b} to the RECEIVER operand specified by C{a}. If n=1 then the VASN instruction is followed by one or more further VASN instructions representing a multiple assignment. Otherwise n=Ø.

The RECEIVER operand may be either of vector type, or it may specify the 'FILE' pseudo-variable in which case the file specified by the FILE instruction indexed by C{a_{op}}is indicated.

VCRS: vector crossproduct



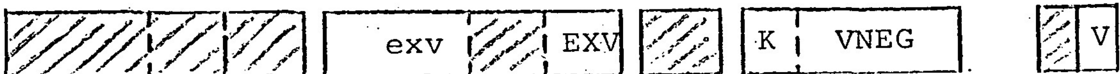
INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

The result of the VCRS instruction is the crossproduct of the two vector operands specified by C{a} and C{b}. It is defined only for 3-dimensional vector operands.

VMPR: vector-matrix multiply

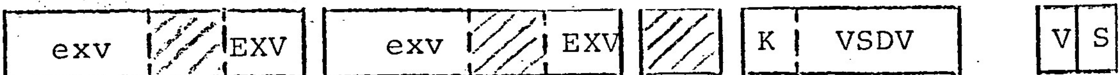
The result of the VMPR instruction is the vector operand specified by C{a} postmultiplied by the matrix operand specified by C{b}.

VNEG: vector negate



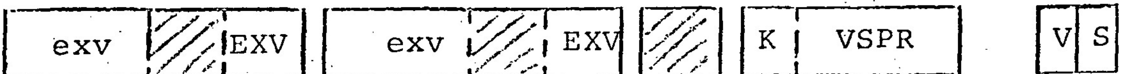
The result of the VNEG instruction is the negative of the vector operand specified by C{b}.

VSDV: vector-scalar divide



The result of the VSDV instruction is the vector operand specified by C{a} divided by the scalar operand specified by C{b}.

VSPR: vector-scalar multiply



The result of the VSPR instruction is the vector operand specified by C{a} multiplied by the scalar operand specified by C{b}.

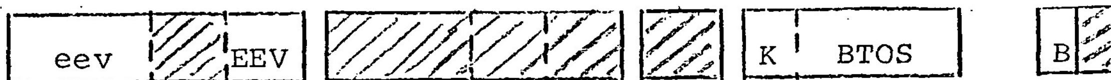
VSUB: vector subtract

The result of the VSUB instruction is the vector operand specified by C{b} subtracted from the vector operand specified by C{a}.

2.20 SCALAR OPERATIONS

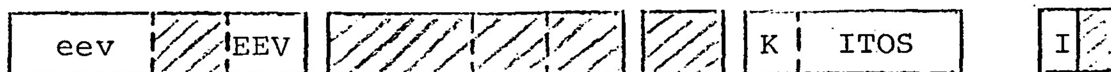
SCALAR OPERATIONS are defined as those arithmetic operations which have a scalar result type, with the addition of the assignment of scalar operands. VARIABLE OPERANDS of any type except matrix, can occur. In the cases where PHASE I cannot check for the compatibility of vector operands, the object text generated for the instructions concerned must include run-time compatibility checking.

BTOS: bit string to scalar conversion



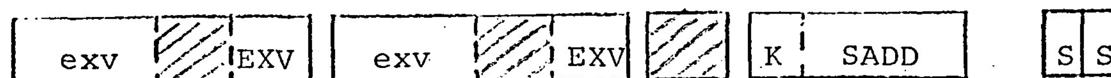
BTOS denotes the conversion of the bit string operand specified by C{a} to a scalar. The bit string is first padded with zeroes or truncated on the left to the same length as an integer (implementation dependent). The bit pattern is treated as an integer and converted to a scalar.

ITOS: integer to scalar conversion



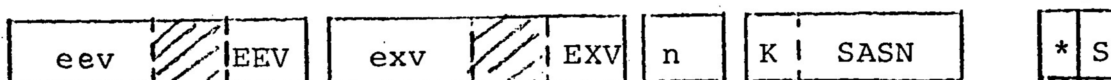
ITOS denotes the conversion of the integer operand specified by C{a} to a scalar. If $C\{a_q\} \equiv$ IMD and $C\{a_{op}\} = \emptyset$, then the ITOS instruction is part of a HALMAT subscript construct, and the operand specified is a TERMINAL DIMENSION or an ARRAY DIMENSION or a STRUCTURE-copy DIMENSION of the symbol-table variable specified by the associated ALC or RALC instruction. Chapter 3 explains this usage in detail.

SADD: scalar add



The result of the SADD instruction is the sum of the scalar operands specified by C{a} and C{b}.

SASN: scalar assign



INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

SASN assigns the scalar operand specified by C{b} to the operand specified by C{a}. The type of the RECEIVER operand may be bit string, character, integer, or scalar, or the 'FILE' pseudo-variable may be specified.

If a bit string RECEIVER operand is specified, the scalar right-hand side is first rounded to an integer, and then the bit string is assigned the same bit pattern as the integer padded with zeroes or truncated on the left as required (implementation dependent).

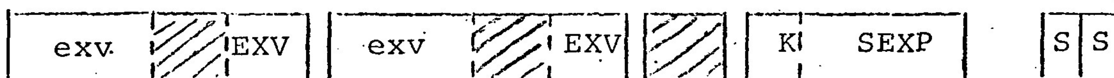
If a character RECEIVER operand is specified then the character is assigned the decimal representation of the value of the right-hand side operand, and if the character operand does not have the 'VARYING' attribute, is padded as necessary on the right with blanks.

If an integer RECEIVER operand is specified then the value of the right-hand side operand is rounded to the nearest integer before assignment.

If the RECEIVER operand specified is the 'FILE' pseudo-variable, writing on the file specified by the FILE instruction indexed by $C\{a_{op}\}$ is indicated.

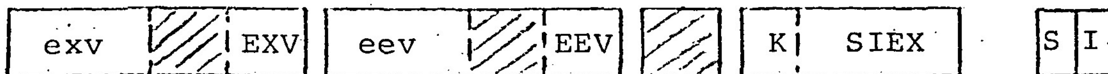
In any of the above cases if n=1 then the SASN instruction is followed by one or more further SASN instructions completing a multiple assignment. Otherwise n=Ø.

SEXP: scalar exponentiation-by-scalar



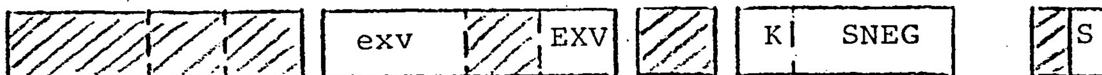
The result of the SEXP instruction is the scalar operand specified by C{a} taken to the power given by the scalar operand specified by C{b}.

SIEX: scalar exponentiation by integer



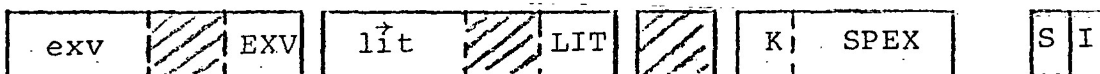
The result of the SIEX instruction is the scalar operand specified by C{a} taken to the power given by the integer operand specified by C{b}.

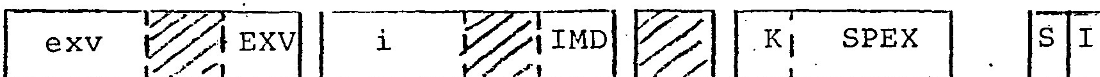
SNEG: scalar negate



The result of the SNEG instruction is the negative of the scalar operand specified by C{b}.

SPEX: scalar exponentiation by positive integer





The result of the SPEX instruction is the scalar operand specified by C{a} taken to the power given by the positive integer operand specified by C{b}, which can only have either of the two forms shown.

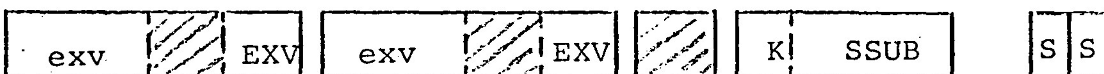
SSDV: scalar divide

The result of the SSDV instruction is the scalar operand specified by C{a} divided by the scalar operand specified by C{b}.

SSPR: scalar-scalar multiply

The result of the SSPR instruction is the product of the scalar operands specified by C{a} and C{b}.

SSUB: scalar subtract



The result of the SSUB instruction is the scalar operand specified by C{b} subtracted from the scalar operand specified by C{a}.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

VDOT: vector dot product

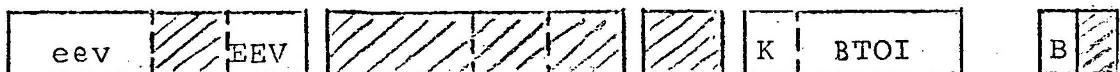
The result of the VDOT instruction is the dot product of the vector operands specified by C{a} and C{b}.

2.21 INTEGER OPERATIONS

INTEGER OPERATIONS are defined as those arithmetic operations which have integer results with the addition of the assignment of integer operands. VARIABLE OPERANDS of any type can occur.

Some of the instructions in which C{a} specifies an integer operand may have a pecularity of the field a. If $C\{a_q\} \equiv$ IMD and $C\{a_{op}\} = \emptyset$ for one of these instruction, then the instruction is part of a HALMAT subscript construct: the operand being specified is a TERMINAL DIMENSION or ARRAY DIMENSION or STRUCTURE-COPY DIMENSION of the symbol table variable specified by the associated RALC or ALC instruction of the construct. This situation is explained in detail in Chapter 3. Instructions where the situation can occur will be called SX-type instructions when they are described below.

BTOI: bit to integer conversion



BTOI indicates the conversion of the bit string operand specified by C{a} to an integer with the same bit pattern, padding with zeroes or truncating on the left as necessary (implementation dependent).

IADD: integer add

[SX-type instruction]

The result of the IADD instruction is the sum of the two integer operands specified by C{a} and C{b}.

IASN: integer assign

IASN assigns the integer operand specified by C{b} to the operand specified by C{a}. The RECEIVER operand may be of any type, and in addition may be the 'FILE' pseudo-variable.

If the RECEIVER operand is of matrix, vector or structure types then C{b} can only specify a literal zero. All of the operand is zeroed.

If the 'FILE' pseudo-variable is specified, then writing on the file specified by the FILE instruction indexed by $C\{a_{op}\}$ is indicated.

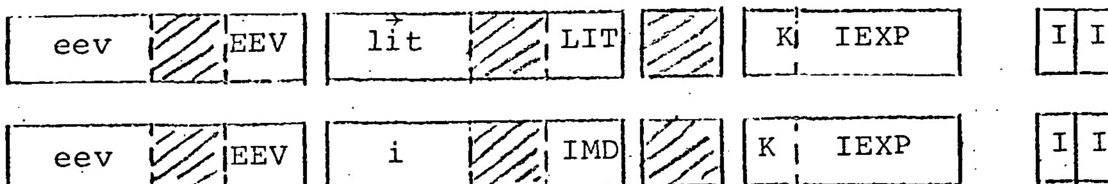
If the RECEIVER operand is of scalar type then the appropriate conversion is made.

If the RECEIVER operand is of bit string type then the bit string is assigned the same bit pattern as the integer, padded with zeroes or truncated on the left as required (implementation dependent).

If the RECEIVER operand is of character type then the decimal representation of the integer is assigned, and if the character operand does not have the 'VARYING' attribute, is padded with blanks or truncated on the right as required.

In any of the above cases if n=1 then the IASN instruction is followed by one or more further IASN instructions completing a multiple assignment. Otherwise, n = Ø.

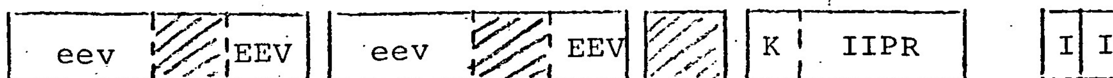
IEXP: integer exponentiation



The result of the IEXP instruction is the integer operand specified by C{a} taken to the power of the integer operand specified by C{b}, which can only take either of the two forms shown.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

IIPR: integer multiply

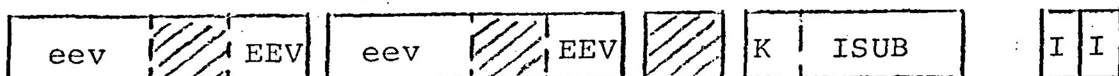


The result of the IIPR instruction is the product of the integer operands specified by C{a} and C{b}.

INEG: integer negate

The result of the INEG instruction is the negative of the operand specified by C{b}.

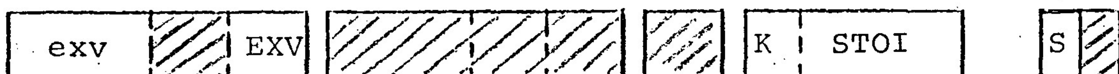
ISUB: integer subtract



[SX-type instruction]

The result of the ISUB instruction is the integer operand specified by C{b} subtracted from the integer operand specified by C{a}.

STOI: scalar to integer conversion



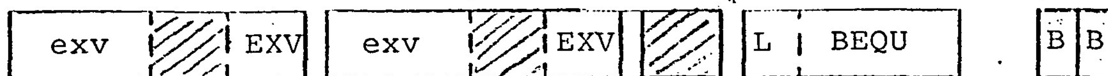
STOI specifies the rounding of the scalar operand specified by C{a} to an integer.

2.22 CONDITIONAL OPERATIONS

CONDITIONAL OPERATIONS are defined as logical and compare operations which have results which are either TRUE or FALSE. VARIABLE OPERANDS of all types can occur.

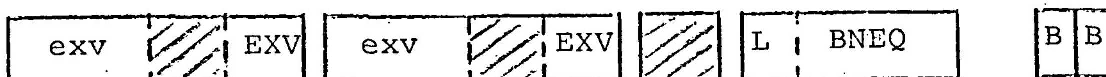
Compare operations on character operands deserve some explanation. Two character operands are 'equal' only if they are of the same length and the same value. One character operand is 'greater than' another if it is longer, or if it is the same length but the value is after the other in alphabetical order.

BEQU: bit string equal



The result of the BEQU instruction is TRUE if the bit string operands specified by C{a} and C{b} are identical, and FALSE otherwise.

BNEQ: bit string not equal

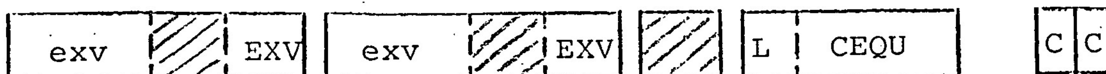


The result of the BNEQ instruction is TRUE if the bit string operands specified by C{a} and C{b} are not identical, and FALSE otherwise.

CAND: logical and

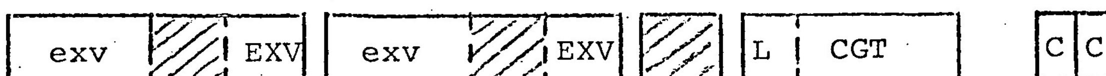
The result of the CAND instruction is the logical 'and' of the logical operands specified by C{a} and C{b}.

CEQU: character equal



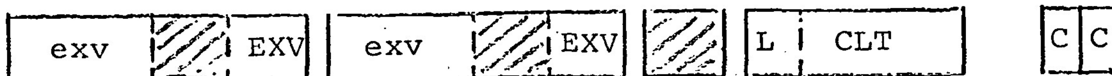
The result of the CEQU instruction is TRUE if the character operands specified by C{a} and C{b} are identical, and FALSE otherwise.

CGT: character greater than



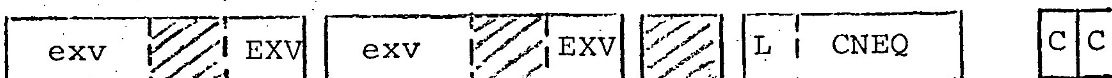
The result of the CGT instruction is TRUE if the character operand specified by C{a} is greater than the character operand specified by C{b}, and FALSE otherwise.

CLT: character less than



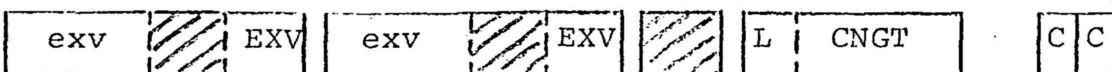
The result of the CLT instruction is TRUE if the character operand specified by C{a} is less than the character operand specified by C{b}, and FALSE otherwise.

CNEQ: character not equal



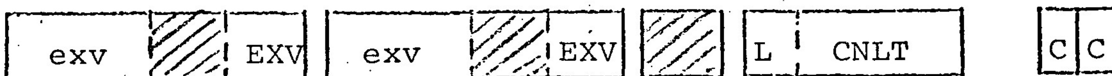
The result of the CNEQ instruction is FALSE if the character operands specified by C{a} and C{b} are identical, and TRUE otherwise.

CNGT: character not greater than



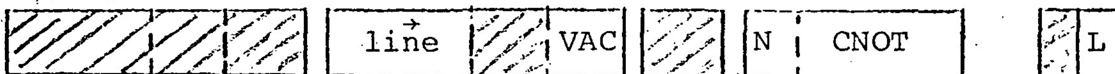
The result of the CNGT instruction is TRUE if the character operand specified by C{a} is not greater than the character operand specified by C{b}, and FALSE otherwise.

CNLT: character not less than



The result of the CNLT instruction is TRUE if the character operand specified by C{a} is not less than the character operand specified by C{b}, and FALSE otherwise.

CNOT: logical not



The result of the CNOT instruction is the complement of the logical operand specified by C{b}.

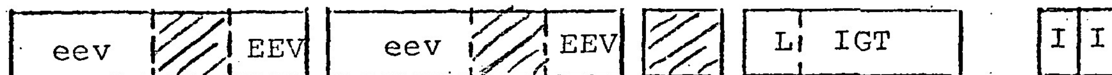
COR: logical or

The result of the COR instruction is the logical 'or' of the logical operands specified by C{a} and C{b}.

IEQU: integer equal

The result of the IEQU instruction is TRUE if the integer operands specified by C{a} and C{b} are equal and FALSE otherwise.

IGT: integer greater than

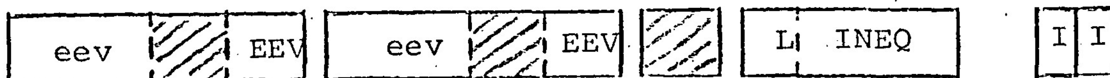


The result of the IGT instruction is TRUE if the integer operand specified by C{a} is greater than the integer operand specified by C{b}, and FALSE otherwise.

ILT: integer less than

The result of the ILT instruction is TRUE if the integer operand specified by C{a} is less than the integer operand specified by C{b}, and FALSE otherwise.

INEQ: integer not equal

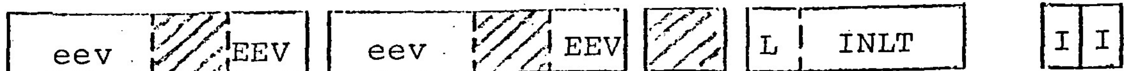


The result of the INEQ instruction is TRUE if the integer operand specified by C{a} is not equal to the integer operand specified by C{b}, and FALSE otherwise.

INGT: integer not greater than

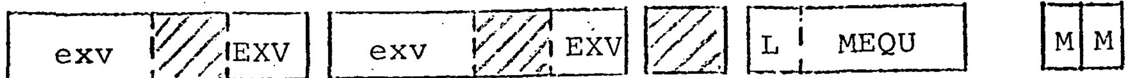
The result of the INGT instruction is TRUE if the integer operand specified by C{a} is not greater than the integer operand specified by C{b}, and FALSE otherwise.

INLT: integer not less than



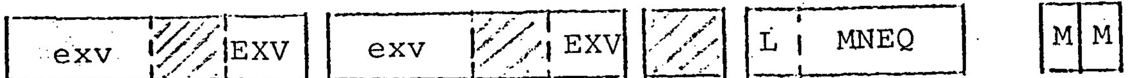
The result of the INLT instruction is TRUE if the integer operand specified by C{a} is not less than the integer operand specified by C{b}, and FALSE otherwise.

MEQU: matrix equal



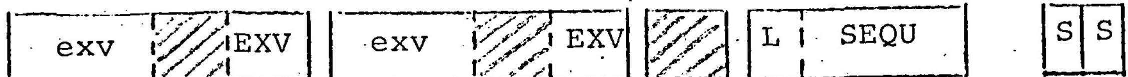
The result of the MEQU instruction is TRUE if all the corresponding elements of the matrix operands specified by C{a} and C{b} are equal, and FALSE otherwise.

MNEQ: matrix not equal



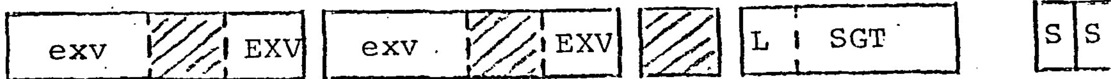
The result of the MNEQ instruction is TRUE if not all of the corresponding elements of the matrix operands specified by C{a} and C{b} are equal, and FALSE otherwise.

SEQU: scalar equal



The result of the SEQU instruction is TRUE if the scalar operands specified by C{a} and C{b} are equal and FALSE otherwise.

SGT: scalar greater than

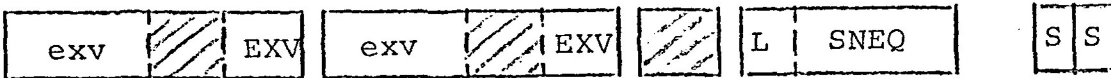


The result of the SGT instruction is TRUE if the scalar operand specified by C{a} is greater than the scalar operand specified by C{b}, and FALSE otherwise.

SLT: scalar less than

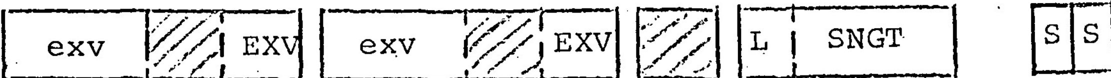
The result of the SLT instruction is TRUE if the scalar operand specified by C{a} is less than the scalar operand specified by C{b}, and FALSE otherwise.

SNEQ: scalar not equal



The result of the SNEQ instruction is TRUE if the scalar operands specified by C{a} and C{b} are unequal and FALSE otherwise.

SNGT: scalar not greater than



The result of the SNGT instruction is TRUE if the scalar operand specified by C{a} is not greater than the scalar operand specified by C{b} and FALSE otherwise.

SNLT: scalar not less than

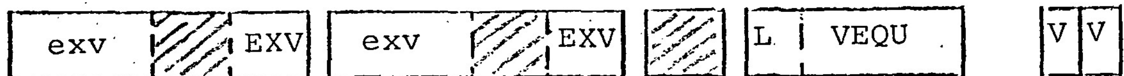
The result of the SNLT instruction is TRUE if the scalar operand specified by C{a} is not less than the scalar operand specified by C{b}, and FALSE otherwise.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

SNLT: scalar not less than

The result of the SNLT instruction is TRUE if the scalar operand specified by C{a} is not less than the scalar operand specified by C{b}, and FALSE otherwise.

VEQU: vector equal



The result of the VEQU instruction is TRUE if all the corresponding elements of the vector operands specified by C{a} and C{b} are equal, and FALSE otherwise.

VNEQ: vectors not equal

The result of the VNEQ instruction is TRUE if not all of the corresponding elements of the vector operands specified by C{a} and C{b} are equal, and FALSE otherwise.

2.23 INITIALIZATION OPERATIONS

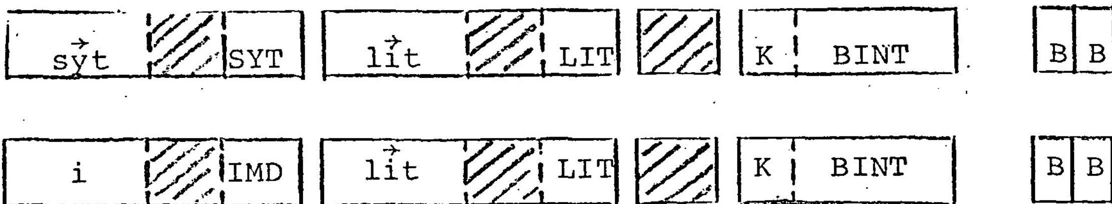
INITIALIZATION OPERATIONS assign initial values of 'STATIC' or 'AUTOMATIC' variables. Some of the instructions may make special use of the field {a}. If $C\{a_q\} \equiv$ IMD then $C\{a_{op}\}$ is the value of an OFFSET for an associated symbol table variable specified by a STRI instruction to be described.

INITIALIZATION OPERATIONS assume that the storage assigned for a symbol table variable will be arranged in a particular way when the object text is generated. To take the most complicated example possible consider an arrayed matrix M which is a structure terminal. Suppose that any element of the variable can with standard HAL notation be written

$$M_{s1,s2,s3;a1,a2,a3:r1,c1}$$

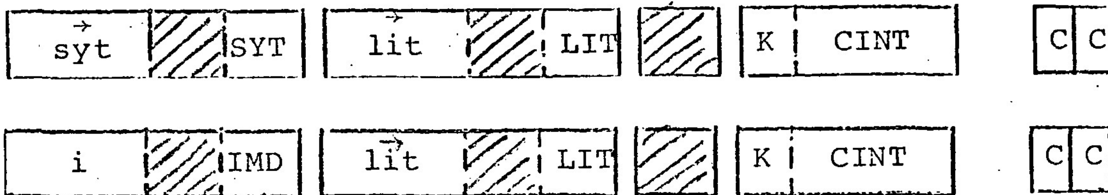
s1 through s3 are STRUCTURE-COPY SUBSCRIPTS, a1 through a3 are ARRAY SUBSCRIPTS, AND r1, c1 are the ROW and COLUMN SUBSCRIPTS. Storage is allocated in ascending positions with the value of the right most subscript c1 increasing most rapidly and the value of the left most subscript least rapidly. $M_{1,1,1;1,1,1:1,1}$ is the first element in storage. Calling this the first storage element, then any other element has a positive linear OFFSET with respect to the first element, which is determinable from knowledge of the various DIMENSIONS of the variable. It is this OFFSET which is specified by $C\{a_{op}\}$ in the cases mentioned previously.

BINT: bit string initialize



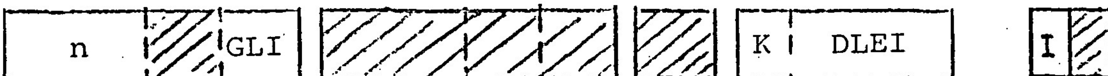
BINT initializes the bit string operand specified by C{a} with the literal specified by C{b}. There are two versions: in the first C{a} specifies a symbol table variable; in the second an OFFSET.

CINT: character initialize



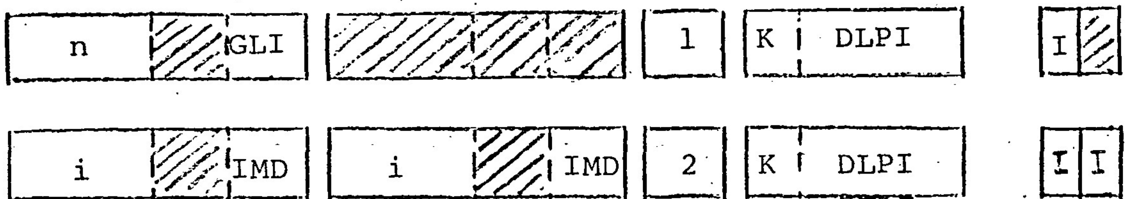
CINT initializes the character operand specified by C{a} with the literal specified by C{b}. There are two versions: in the first C{a} specifies a symbol table variable, in the second, an OFFSET.

DLEI: initialize loop end



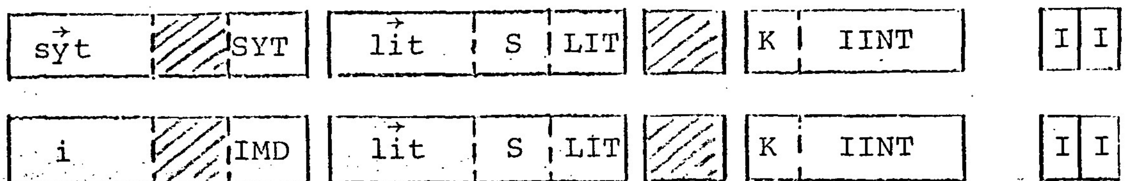
DLEI indicates the end of a repeat loop for incrementing OFFSET values. C{a} specifies an internal integer variable for use during object text generation matching that specified in the related DLPI instruction.

DLPI: initialize loop header



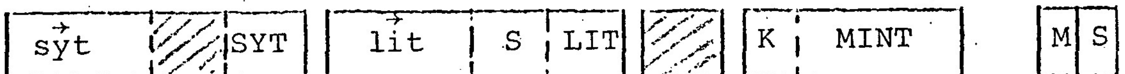
DLPI is a two-part instruction specifying the start of a repeat loop for incrementing OFFSET values. C{e} distinguishes $DLPI_1$ from $DLPI_2$. The first is invariably immediately followed by the second in the HALMAT text. C{a} of $DLPI_1$ specifies an internal integer variable used during object text generation which matches C{a} of a following DLEI instruction. $C\{a_{op}\}$ and $C\{b_{op}\}$ of $DLPI_2$ are respectively the number of times execution of the loop is required, and the amount by which the OFFSET is to be increased.

IINT: integer initialize



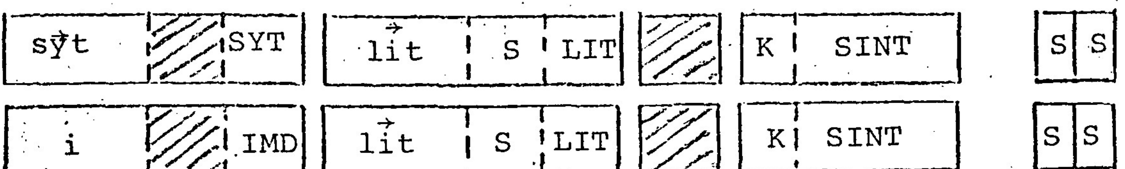
IINT initializes the integer operand specified by C{a} with the literal specified by C{b}. There are two versions: in the first C{a} specifies a symbol table variable; in the second, an OFFSET. $C\{b_t\}$ in either version is used to indicate a sign. s=1 indicates that the literal must be negated. s=∅ otherwise.

MINT: matrix initialize



MINT initializes all the elements of the matrix operand specified by C{a} with the literal specified by C{b}. $C\{b_t\}$ is used to indicate a sign. s=1 indicates that the literal must be negated. s=∅ otherwise.

SINT: scalar initialize



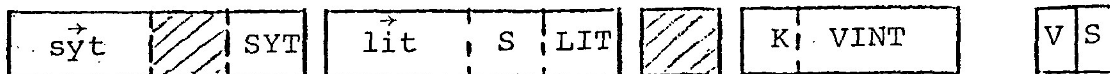
SINT initializes the scalar operand specified by C{a} with the literal specified by C{b}. There are two versions: in the first C{a} specifies a symbol-table variable; in the second, an OFFSET. $C\{b_t\}$ in either version is used to indicate a sign. s=1 indicates that the literal must be negated. s=Ø otherwise.

Note that the SINT instruction is used when matrices and vectors are to be initialized element-by-element.

STRI: repeated initialize specifier

STRI indicates that the symbol table operand specified by C{a} is to be initialized element-by-element. C{b} specifies the same internal integer variable as specified by C{a} of the related DLPI and DLEI instructions. n_a is the number of initialize loops following the instruction.

VINT: vector initialize

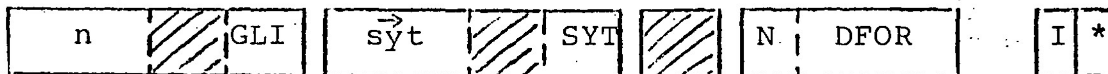


VINT initializes all the elements of the vector operand specified by C{a} with the literal specified by C{b}. $C\{b_t\}$ is used to indicate a sign. s=1 indicates that the literal must be negated. s=Ø otherwise.

2.24 DO FOR SPECIFIERS

The following instructions are used to describe HAL DO FOR statement groups. Only the logically necessary information is given in the instructions, without any specification of how incrementing or range-testing should be carved out.

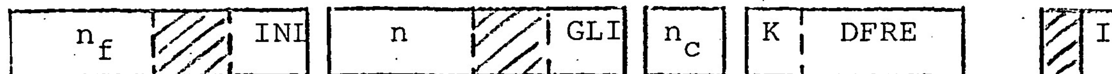
DFOR: do for header



DFOR specifies the start of a DO FOR statement group. C{b} indicates the integer or scalar range variable. C{a} indicates

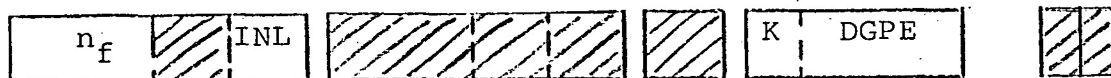
an internal integer variable used to index the DO FOR list element. Flow numbers n-2 to n+1 are reserved for the generation of object text.

DFRE: do for end



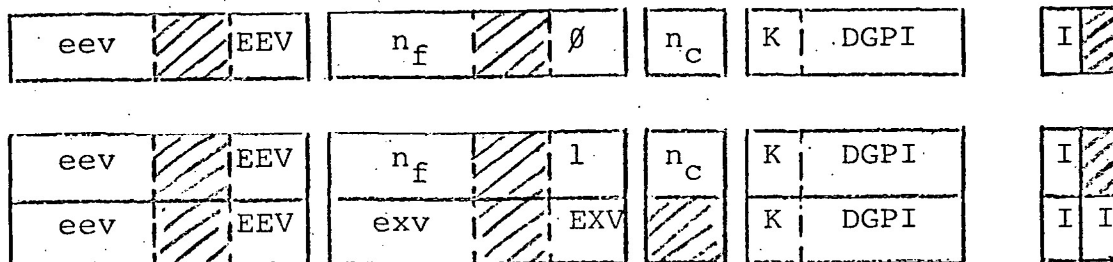
DFRE signals the end of a DO FOR statement group. n_c is the number of DO FOR list elements. C{b} corresponds with the internal integer variable specified by C{a} of the associated DFOR instruction. $n_f \equiv n$ is a reserved flow number. The intent of DFRE is similar to CBRA.

DGPE: do for list terminator



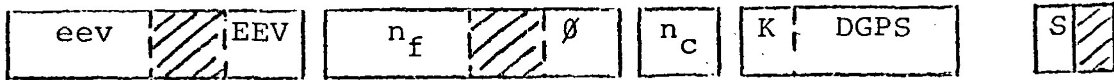
DGPE signals the end of a list of elements of a DO FOR statement. The reserved flow number n_f is always given by $n_f \equiv n-1$ where n is $C\{a_{op}\}$ of the corresponding DFOR instruction.

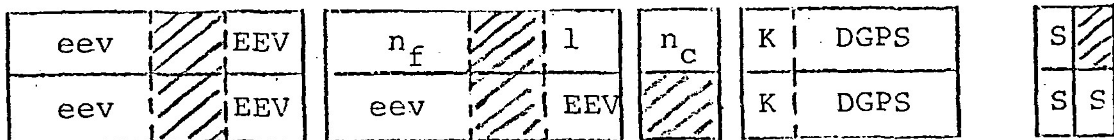
DGPI: integer do for list element



DGPI specifies a DO FOR list element. There are two versions. The single version specifies a non-iterative list element, the double version an iterative list element. The versions are distinguished by $C\{b_q\}$ of the first instruction. In each version n_f is a flow number which together with n_f+1 is reserved for code generation. n_c is the sequential number of the list element, starting from one. In the first version, C{a} is the non-iterative list element. In the second version C{a} of the first instruction of the pair is the initial value of the list element. In the second instruction, C{a} is the final value, and C{b} is the increment. In all cases the operands specified are of integer type (having been forced to the same type if necessary).

DGPS: scalar do for list element





The DGPS instructions are identical to the DGPI instructions except all the arithmetic operands specified are of scalar type.

3. HALMAT CONSTRUCTS

This chapter describes HALMAT constructs appearing in HALMAT text. A HALMAT construct is a sequence of HALMAT instructions possessing some structure characteristic of the original source text from which it was derived. For example a 'DO CASE' statement in the HAL source text would be converted during PHASE I of the compiler into a sequence of HALMAT instructions possessing certain unique structural properties: this sequence would be a particular realization of the HALMAT 'DO CASE' construct.

A HALMAT construct is rarely found in its most general form: on the other hand great variety in particular realization can occur. Rather than give definitions of the general form of different HALMAT constructs this chapter will therefore proceed by presenting annotated examples of typical HALMAT constructs. In each section the examples are graded in complexity, giving the reader a clear picture of the generality of each form.

The notation used in the examples is uniform with that already presented, with one or two obvious extensions. HAL source text is presented in its usual form. HAL statements which supply background information but are not themselves involved in the derived HALMAT construct are enclosed in braces {}. LINE numbers are given in a separate column to the left of the HALMAT text.

3.1 Macroscopic HALMAT structure

The HAL source text for an entire program consists of possibly 'COMPOOL' declarations, a program declaration, and then the body of the program followed by a 'CLOSE' statement. Each statement of a program is terminated by a semicolon. PHASE I of the compiler converts each statement to a PARAGRAPH of HALMAT text. Each PARAGRAPH of HALMAT text consists of a SMRK instruction followed by a sequence of HALMAT instructions derived from the source text. Occasionally a HAL statement may generate no HALMAT text, the derived PARAGRAPH only consisting of the SMRK instruction.

As many whole PARAGRAPHS of HALMAT text as possible are stored on each PAGE (see Section 1.7). The end of the last PARAGRAPH on each PAGE except the last is followed by a BMRK instruction. The end of the last PARAGRAPH on the last PAGE is followed by the instructions SMRK, CMRK, BMRK in that order. This situation is presented diagrammatically in Figure 3.1.1 which shows the macroscopic structure of the HALMAT text for a HAL program with no

COMPOOL declarations. The first significant instruction is the program definition head MDEF. This is matched by a closing RTRN instruction at the end of the text.

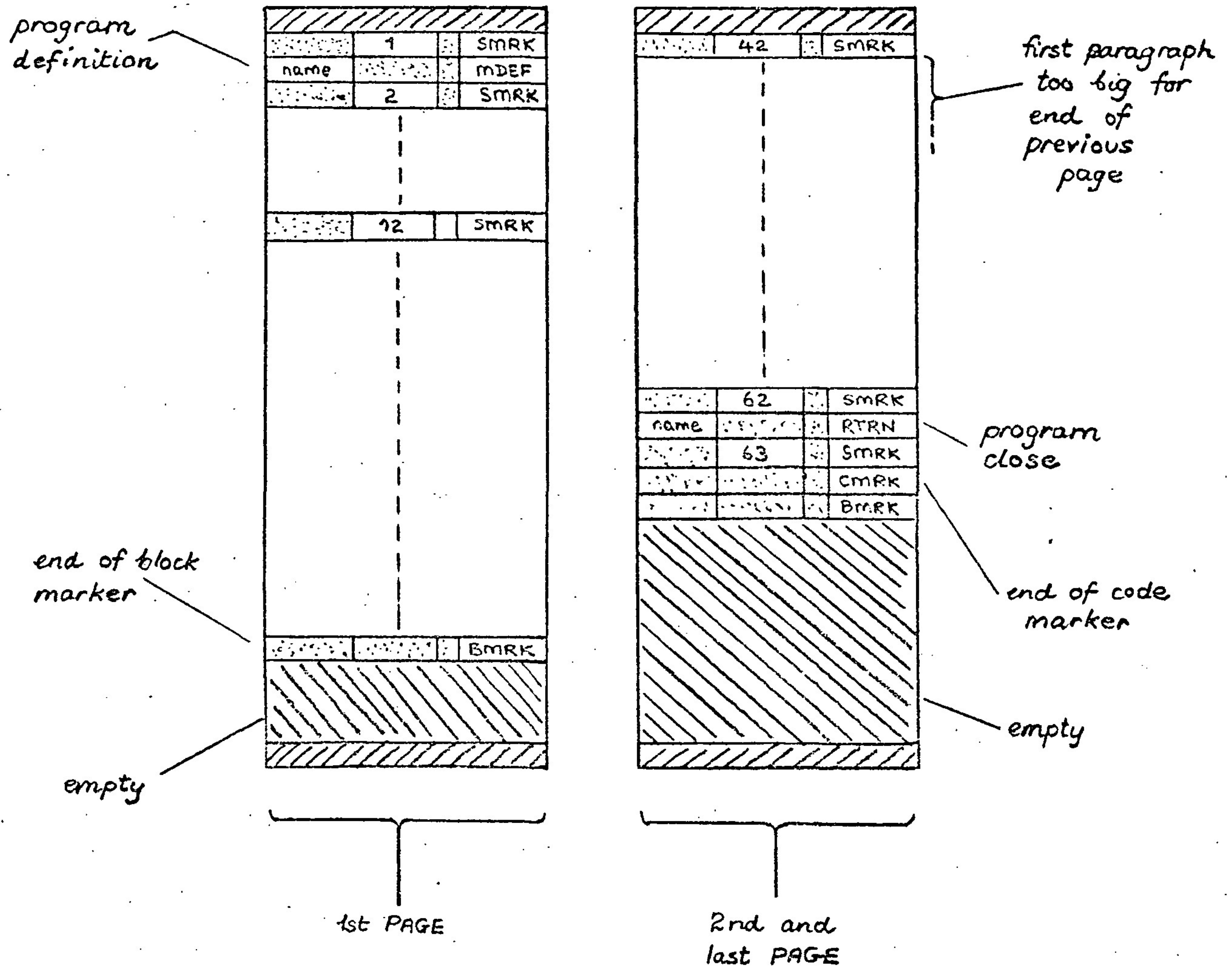


Figure 3.1.1 Macroscopic structure of HALMAT text

3.2 Arithmetic Constructs

In this section several examples of constructs corresponding to assignment statements in the HAL source text are presented.

Example 3.2.1: simple unarrayed unsubscripted assignment

```
{ DECLARE X SCALAR;             }
{ DECLARE CHARACTER, A, B;      }
  A,B = X² + 2X - 1.55||'CMS';
```

1			1∅		∅	N	SMRK	statement marker
2	$\vec{x}$	SYT	$\vec{2}$	LIT		∅	SPEX	x^2
3	$\vec{2}$	LIT				∅	ITOS	(note 1.)
4	3	VAC	$\vec{x}$	SYT		∅	SSPR	2X
5	2	VAC	4	VAC		∅	SADD	
6	5	VAC	$\vec{1.55}$	LIT		∅	SSUB	-1.55
7	6	VAC				∅	STOC	convert to character
8	7	VAC	$\vec{\text{'CMS'}}$	LIT		∅	CCAT	catenate
9	$\vec{B}$	SYT	8	VAC	1	∅	CASN	assign
10	$\vec{A}$	SYT	8	VAC	∅	∅	CASN	(note 2.)

Notes:

1. literals which are integerisable, (i.e. have integral values irrespective of whether they are written with a decimal point or not) are regarded as integers and thus force the conversion.

2. multiple assignments appear in right-to-left order of left hand sides.

Example 3.2.2: assignment with arrayed structure terminals

```
DECLARE 1 ALPHA (6),
          2 BETA (3),
            3 ARRAY (4,2) SCALAR, R;
DECLARE ARRAY (4,2) SCALAR, S;
DECLARE VECTOR (3), V, W;
```

[R] = [S] [R] + $\overline{V} \cdot \overline{W}$;

1			12		Ø	N	SMRK	
2	$\vec{S}$	SYT	$\vec{R}$	SYT		Ø	SSPR	(note 1.)
3	$\vec{V}$	SYT	$\vec{W}$	SYT		Ø	VDOT	dot product
4	2	VAC	3	VAC		Ø	SADD	
5	$\vec{R}$	SYT	4	VAC	Ø	Ø	SASN	free arrayness
6	6	IMD				D	SDLP	and structureness
7	3	IMD				D	SDLP	of
8	4	IMD				D	ADLP	statement
9	2	IMD				D	ADLP	(note 2.)
10						D	DLPE	

Notes:

1. information on the array- and structure-copy dimensions of a variable must be obtained from the symbol table.

2. the order of appearance should be carefully studied in relation to the declarations.

Example 3.2.3: assignment with terminal subscripting

```
DECLARE V VECTOR (6),
        M MATRIX (7,4);
```

$$\overline{V}_{1\ TO\ 3} = \overline{M}_{3\ AT\ 4,2} + \overline{V}_{4\ TO\ 6};$$

1				13			Ø	N	SMRK	
2	$\vec{V}$		SYT					A	RALC	receiver subscripting
3	1		IMD	3	1	IMD		A	TTSB	
4								A	ALCE	(note 1.)
5	$\vec{M}$		SYT					A	ALC	matrix subscripting: row then column
6	4		IMD	$\vec{3}$	2	LIT		A	TASB	
7.	2		IMD		1			A	TIDX	
8								A	ALCE	
9	$\vec{V}$		SYT					A	ALC	vector subscripting
10	4		IMD	6	1	IMD		A	TTSB	
11								A	ALCE	
12	5		VAC	9		VAC		Ø	VADD	
13	2		VAC	12		VAC	Ø	Ø	VASN	assignment

Notes:

1. It is difficult to predict the order of emission of the subscript constructs for different variables in the same assignment.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Example 3.2.4: complex assignment with arrayness and subscripting

```
DECLARE INTEGER, I, J, K;
DECLARE ARRAY (3) VECTOR (7), V;
DECLARE ARRAY (4,6) MATRIX (3,8), M;
```

$$[\bar{V}]_{I-1\ TO\ \#} = [\bar{M}]_{J,\ 3\ AT\ K-4:\ \#,*};$$

#	Operand 1		Qualifier	Operand 2		Qualifier		Tag	Opcode	
1				15			Ø	N	SMRK	
2	$\vec{I}$		SYT	$\vec{1}$		LIT		SE	ISUB	receiver subscripting (note 1.)
3	$\vec{V}$		SYT					A	RALC	
4	2		VAC	7	1	IMD		A	TTSB	
5								A	ALCE	
6	$\vec{K}$		SYT	$\vec{4}$		LIT		SE	ISUB	left hand side subscripting (note 2.)
7	$\vec{M}$		SYT					A	ALC	
8	$\vec{J}$		SYT		2			A	AIDX	
9	6		VAC	$\vec{3}$	1	LIT		A	AASB	
10	3		IMD		2			A	TIDX	
11								A	ALCE	
12	3		VAC	7		VAC	Ø	Ø	VASN	
13	3		IMD					D	ADLP	arrayness specifiers
14								D	DLPE	

Notes:

1. The value of # in the source text is known so that its value is inserted.

2. No instruction appears for * subscripts since that dimension is 'free'.

Example 3.2.5: complex assignment with formal parameters

```
ZERO: PROCEDURE (A, B) ASSIGN (C);
      DECLARE ARRAY (*) VECTOR (*), A, C;
      DECLARE ARRAY (*,*) MATRIX (*,*), B;
      DECLARE INTEGER, I, J;
```

$[\bar{C}]_{1\ TO\ \#-1} = [\bar{A}]_{2\ TO\ \#} + [B]_{*,\#-3:\ I,\ 4\ AT\ \#+J-5}$;

#										
1				18			Ø	N	SMRK	
2	Ø		IMD	$\vec{1}$		LIT		SE	ISUB	receiver subscripting (note 1.)
3	$\vec{C}$		SYT					A	RALC	
4	1	2	IMD	2	1	VAC		A	TTSB	
5								A	ALCE	
6	$\vec{A}$		SYT					A	ALC	subscripting of A. (note 1.)
7	2	2	IMD	Ø	1	IMD		A	TTSB	
8								A	ALCE	
9	Ø		IMD	$\vec{3}$		LIT		SE	ISUB	subscripting of B. (note 1.)
10	Ø		IMD	$\vec{J}$		SYT		S	IADD	
11	1Ø		VAC	$\vec{5}$		LIT		SE	ISUB	
12	$\vec{B}$		SYT					A	ALC	
13	9	1	VAC		1			A	AIDX	
14	$\vec{I}$		SYT		2			A	TIDX	(note 2.)
15	1Ø	1	VAC	$\vec{4}$	1	LIT		A	TASB	
16								A	ALCE	
17	6		VAC	12		VAC		Ø	VADD	
18	3		VAC	17		VAC	Ø	Ø	VASN	
19	$\vec{C}$	1	AST					D	ADLP	arrayness (note 3.)
20								D	DLPE	

Notes:

1. The value of the # is known at execution time only. It is referenced as the 1st array dimension of the variable being subscripted.

2. The value of # is known at execution time only. It is referenced as the column dimension of the matrix being subscripted.

3. The arrayness is known only at execution time. It is referenced as the 1st array dimension of the symbol table variable C.

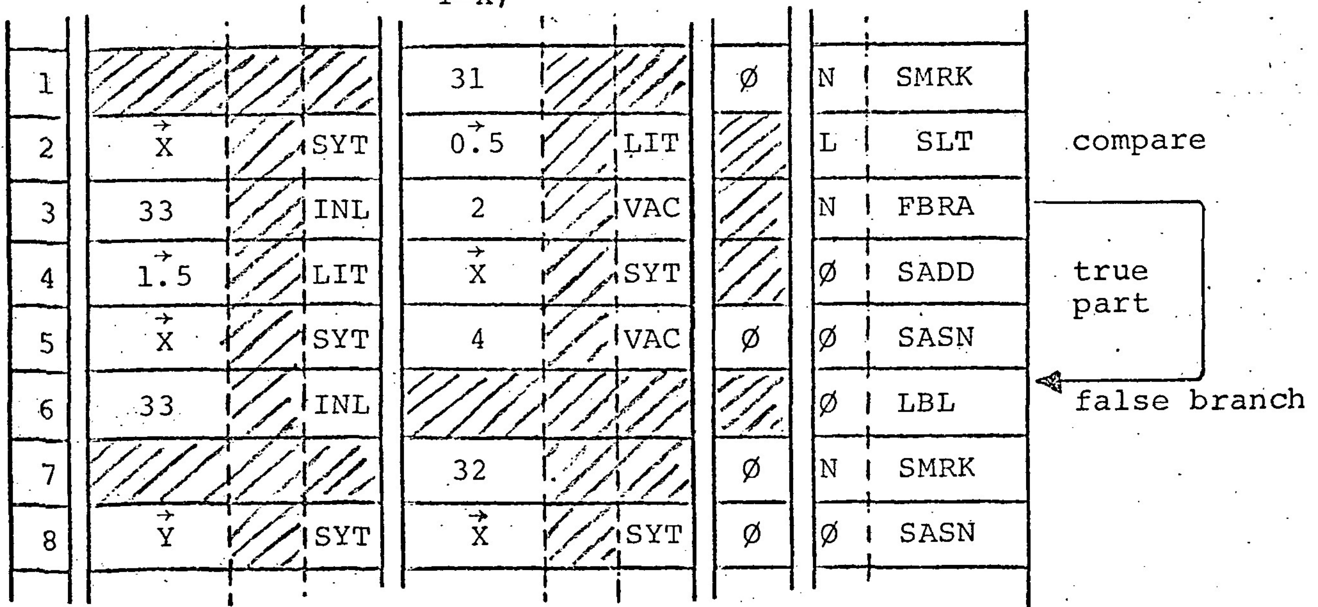
Subscripted character variables with the 'VARYING' attribute have # values which are never known until execution time whether or not they are formal parameters. Their constructs will therefore always be of the same form as the ones for the formal parameters presented in the above example.

3.3 Flow-Control Constructs

In this section examples of HALMAT constructs representing HAL flow-control statements are presented. Flow-control statements include 'IF' statement, 'DO FOR' and 'DO CASE' statements among others.

Example 3.3.1: simple IF statement

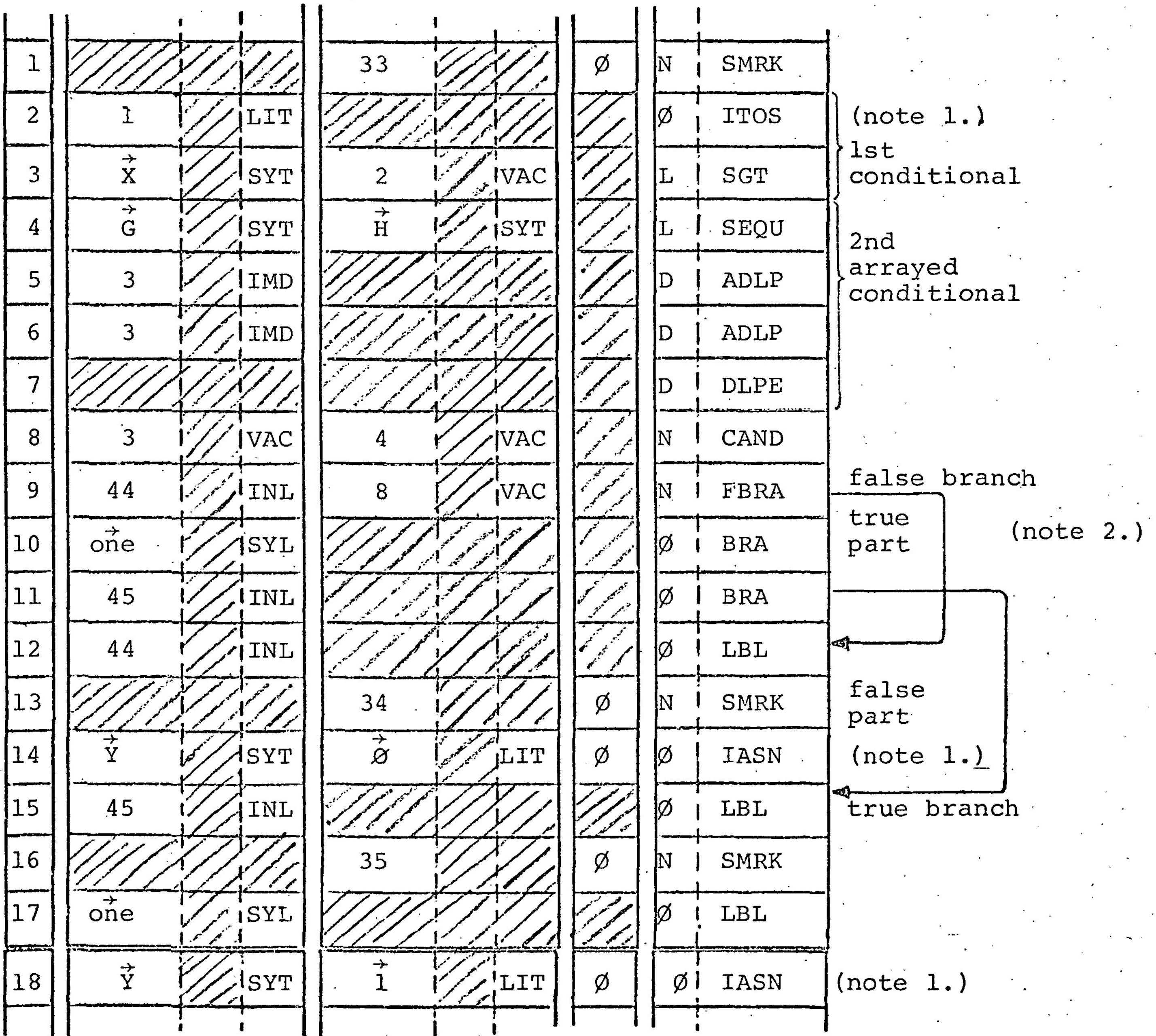
```
{DECLARE SCALAR, X, Y;}
 IF X<0.5 THEN X=1.5+X;

 Y=X;
```



1			31		Ø	N	SMRK	
2	$\vec{X}$	SYT	$\vec{0.5}$	LIT		L	SLT	compare
3	33	INL	2	VAC		N	FBRA	
4	$\vec{1.5}$	LIT	$\vec{X}$	SYT		Ø	SADD	true part
5	$\vec{X}$	SYT	4	VAC	Ø	Ø	SASN	
6	33	INL				Ø	LBL	false branch
7			32		Ø	N	SMRK	
8	$\vec{Y}$	SYT	$\vec{X}$	SYT	Ø	Ø	SASN	

Example 3.3.2: complex IF THEN ELSE statement

```
        { DECLARE SCALAR, X, Y,          }
        {    ARRAY (3,3) SCALAR, G, H;   }
          IF(X>1)&([G]=[H]) THEN GO TO ONE;
          ELSE Y=0;
     ONE: Y=1;
```



1			33		Ø	N	SMRK	
2	1	LIT				Ø	ITOS	(note 1.) 1st conditional
3	$\vec{X}$	SYT	2	VAC		L	SGT	
4	$\vec{G}$	SYT	$\vec{H}$	SYT		L	SEQU	2nd arrayed conditional
5	3	IMD				D	ADLP	
6	3	IMD				D	ADLP	
7						D	DLPE	
8	3	VAC	4	VAC		N	CAND	
9	44	INL	8	VAC		N	FBRA	false branch
10	$\vec{one}$	SYL				Ø	BRA	true part (note 2.)
11	45	INL				Ø	BRA	
12	44	INL				Ø	LBL	
13			34		Ø	N	SMRK	false part
14	$\vec{Y}$	SYT	$\vec{Ø}$	LIT	Ø	Ø	IASN	(note 1.)
15	45	INL				Ø	LBL	true branch
16			35		Ø	N	SMRK	
17	$\vec{one}$	SYL				Ø	LBL	
18	$\vec{Y}$	SYT	$\vec{1}$	LIT	Ø	Ø	IASN	(note 1.)

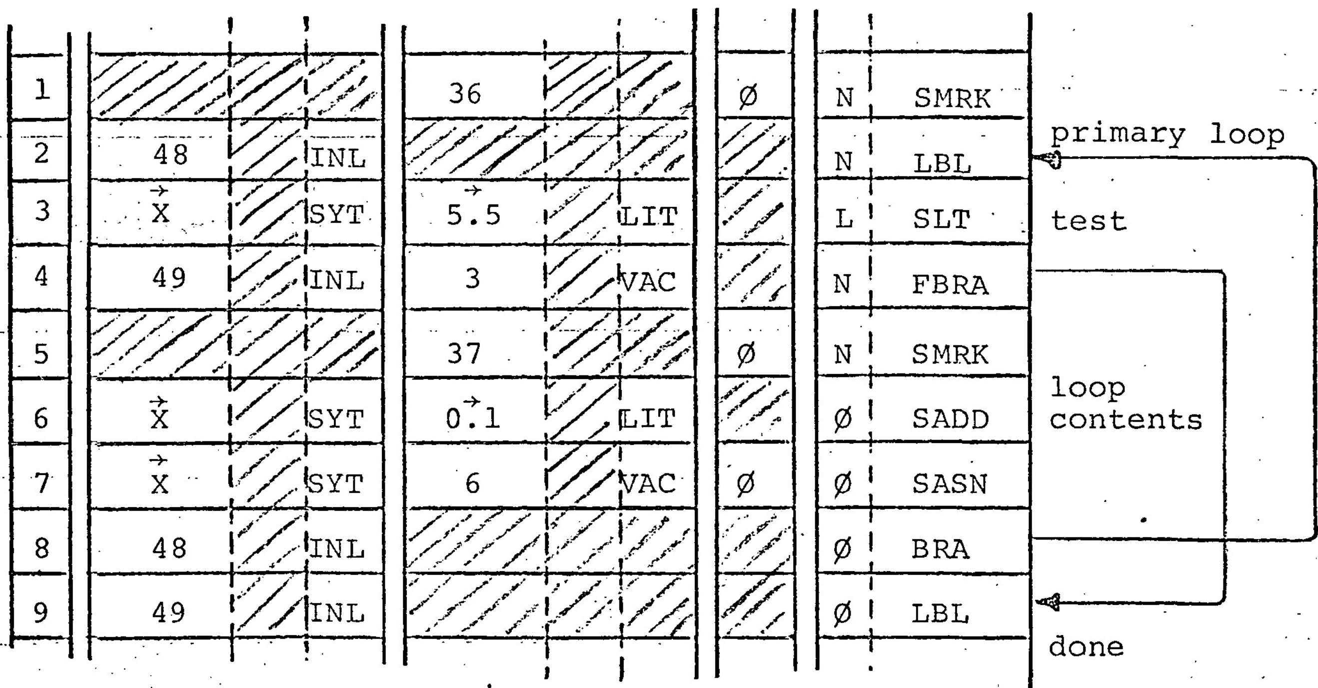
Notes:

1. Note the consequence of integerisable literals.

2. In this case the branch round the true part is redundant because the true part is itself a branch.

Example 3.3.3: DO WHILE statement

```
{ DECLARE X SCALAR;}
DO WHILE X<5.5;
   X = X + 0.1;
END;
```



1			36		Ø	N	SMRK
2	48	INL				N	LBL
3	$\vec{X}$	SYT	$\vec{5.5}$	LIT		L	SLT
4	49	INL	3	VAC		N	FBRA
5			37		Ø	N	SMRK
6	$\vec{X}$	SYT	$\vec{0.1}$	LIT		Ø	SADD
7	$\vec{X}$	SYT	6	VAC	Ø	Ø	SASN
8	48	INL				Ø	BRA
9	49	INL				Ø	LBL

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Example 3.3.4: DO CASE statement

```
{ DECLARE SCALAR, X, Y, Z; }
DO CASE 2 Z;
  X = Y;
  ;
END;
```

#	Operand 1	Qual. 1	Operand 2	Qual. 2	Tag	Flag	Opcode	Notes
1			38		Ø	N	SMRK	
2	$\vec{2}$	LIT				Ø	ITOS	case variable assignment
3	2	VAC	$\vec{Z}$	SYT		Ø	SSPR	
4	Ø	GLI	3	VAC	Ø	Ø	SASN	
5	52	INL				Ø	BRA	(note 1.)
6	52	INL	54	INL	Ø	Ø	CLBL	(note 2.) 1st case
7			39		Ø	N	SMRK	
8	$\vec{X}$	SYT	$\vec{Y}$	SYT	Ø	Ø	SASN	
9	51	INL				Ø	BRA	
10	52	INL	56	INL	1	Ø	CLBL	(note 2.) 2nd case
11			4Ø		Ø	N	SMRK	
12	51	INL				Ø	BRA	
13	52	INL	58	INL	2	Ø	CLBL	(note 2.)
14			41		Ø	N	SMRK	
15	52	INL				Ø	LBL	
16	52	INL	Ø	GLI	2	Ø	CBRA	computed branch to case
17	51	INL				Ø	LBL	end destination

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

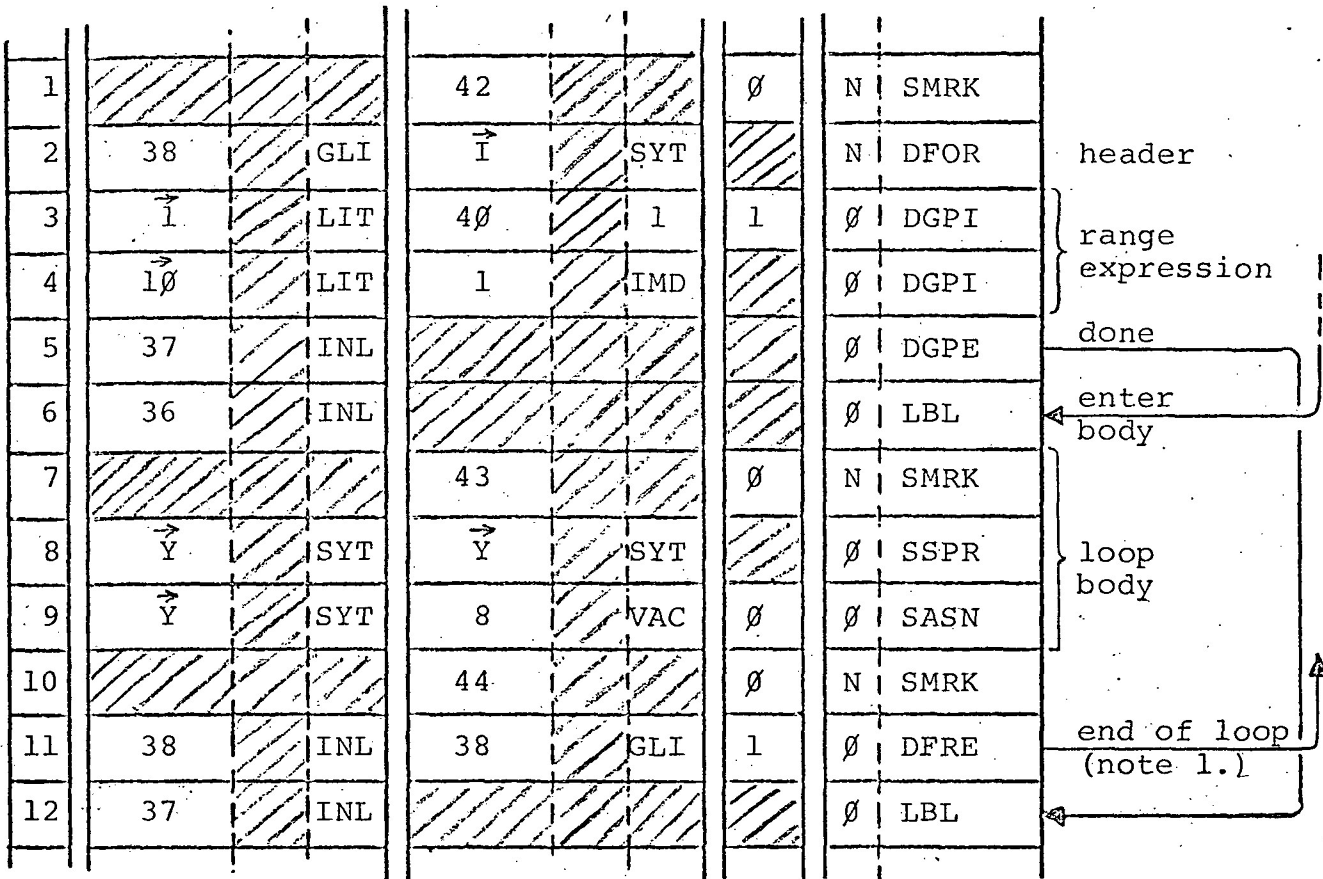
Notes:

1. The next successive internal flow number 53 is reserved for object text generation purposes.

2. The next successive internal flow numbers to C{b_{op}} are reserved for object text generation purposes. Note that the last CLBL instruction is redundant and does not indicate the start of a case.

Example 3.3.5: simple DO FOR statement

```
{ DECLARE I INTEGER, Y SCALAR; }
DO FOR I=1 to 10;
   Y = Y Y;
END;
```



1			42		Ø	N	SMRK
2	38	GLI	$\vec{I}$	SYT		N	DFOR
3	$\vec{I}$	LIT	4Ø	1	1	Ø	DGPI
4	1Ø	LIT	1	IMD		Ø	DGPI
5	37	INL				Ø	DGPE
6	36	INL				Ø	LBL
7			43		Ø	N	SMRK
8	$\vec{Y}$	SYT	$\vec{Y}$	SYT		Ø	SSPR
9	$\vec{Y}$	SYT	8	VAC	Ø	Ø	SASN
10			44		Ø	N	SMRK
11	38	INL	38	GLI	1	Ø	DFRE
12	37	INL				Ø	LBL

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Notes:

1. The DFRE instruction indicates a branch back into the increment section of the DO FOR, as indicated by the DGPI instructions.

Example 3.3.6: complex DO FOR WHILE statement

```
{ DECLARE Y ARRAY (20) SCALAR,  }
{      INTEGER, I, J;           }

DO FOR I=1, J, 4 TO 10+J BY 2 WHILE Y_I<=0;

   Y_I=0;

END;
```

1				42			Ø	N	SMRK	
2	62		GLI	$\vec{I}$		SYT		N	DFOR	header
3	$\vec{I}$		LIT	64		Ø	1	Ø	DGPI	1st element
4	$\vec{J}$		SYT	66		Ø	2	Ø	DGPI	2nd element
5	$\vec{10}$		LIT	$\vec{J}$		SYT		Ø	IADD	3rd element of DO FOR statement
6	$\vec{4}$		LIT	68		1	3	Ø	DGPI	
7	5		VAC	$\vec{2}$		LIT		Ø	DGPI	
8	61		INL					Ø	DGPE	done
9	6Ø		INL					N	LBL	enter while test
10	$\vec{Y}$		SYT					A	ALC	while test
11	$\vec{I}$		SYT		1			A	AIDX	
12								A	ALCE	
13	$\vec{0}$		LIT					Ø	ITOS	
14	1Ø		VAC	13		VAC		Ø	SNGT	
15	61		INL	14		VAC		Ø	FBRA	fail

#	Operand 1			Operand 2							
16				43			Ø	N	SMRK	loop body (16–20)	
17	$\vec{Y}$		SYT					A	RALC		
18	$\vec{I}$		SYT		1			A	AIDX		
19								A	ALCE		
20	17		VAC	$\vec{0}$		LIT	Ø	Ø	IASN		to current DO element
21				44			Ø	N	SMRK		
22	62		INL	62		GLI	3	Ø	DFRE	end of loop (note 1.)	
23	61		INL					Ø	LBL		←

Notes:

1. If execution came from 1st or 2nd DO group, the current DO group is defined as the next in succession. If execution came from the 3rd group, that same group (because of the iterative nature) is defined as the current DO group.

3.4 I/O Constructs

In this section several examples of constructs corresponding to HAL 'READ', 'WRITE' and 'FILE' statements are presented.

Example 3.4.1: READ statement

```
{ DECLARE V VECTOR (6),           }
{         S ARRAY (3) SCALAR,     }
{         C CHARACTER;            }
```

READ(5) $\overline{V}_{1\ TO\ 3}$, S, SKIP(4), COLUMN(1), $\acute{C}$;

1				44			Ø	N	SMRK	
2	5		IMD				5	N	READ	read head
3	$\vec{V}$		SYT					A	ALC	1st list element
4	1		IMD		1			A	TIDX	
5								A	ALCE	
6	3	V	VAC					L	LIST	
7	$\vec{S}$	S	SYT					L	LIST	2nd list element
8	3		IMD					D	ADLP	
9								D	DLPE	
10	$\vec{4}$		LIT					L	FSKP	I/O control
11	$\vec{1}$		LIT					L	FCLM	
12	$\vec{C}$	C	SYT					L	LIST	
13								Ø	LSTE	end of list

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Example 3.4.2: WRITE statement

```
{ DECLARE 1 STR,
            2 MINSTR(2),
               3 TERM ARRAY(4) SCALAR; }
{ DECLARE ARRAY(3) MATRIX(4,4), P, Q;   }
        *  *
WRITE(6) P+Q, STR, LINE(1), 1.5, TERM;
```

1				45			Ø	N	SMRK	
2	6		IMD				5	N	WRIT	write head
3	$\vec{P}$		SYT	$\vec{Q}$		SYT		Ø	MADD	1st list element
4	3	M	VAC					L	LIST	
5	3		IMD					D	ADLP	
6								D	DLPE	
7	$\vec{STR}$	T	SYT					L	LIST	
8	$\vec{1}$		LIT					L	FLIN	I/O control
9	$\vec{1.5}$	S	LIT					L	LIST	(note 1.)
10	$\vec{TERM}$	S	SYT					L	LIST	5th list element
11	2		IMD					D	SDLP	
12	4		IMD					D	ADLP	
13								D	DLPE	
14								Ø	LSTE	end of list

Notes:

1. The literal is given scalar type since it is not integerisable.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Example 3.4.3: FILE statements

```
{ DECLARE 1 XX,
          2 YY(3),
            3 Z MATRIX (2,2); }
XX = FILE(4,22);
FILE(4,23) = Z*_1;
```

1	//////	//////	//////	46	//////	//////	Ø	N	SMRK	read from file into structure
2	4	//////	IMD	$\vec{22}$	//////	LIT	//////	N	FILE	
3	$\vec{XX}$	//////	SYT	2	//////	VAC	Ø	Ø	FASN	
4	//////	//////	//////	47	//////	//////	Ø	N	SMRK	
5	4	//////	IMD	$\vec{23}$	//////	LIT	//////	Ø	FILE	write matrix onto file
6	$\vec{Z}$	//////	SYT	//////	//////	//////	//////	A	ALC	
7	1	//////	IMD	//////	1	//////	//////	A	SIDX	
8	//////	//////	//////	//////	//////	//////	//////	A	ALCE	
9	5	//////	VAC	6	//////	VAC	Ø	Ø	MASN	

3.5 Procedure Constructs

In this section examples of HALMAT constructs representing procedure calls, returns and definitions are presented.

Example 3.5.1: procedure definition and return

```
TWO: PROCEDURE (A̅,B*) ASSIGN (C̅);
    DECLARE VECTOR(*), A, C;
    DECLARE MATRIX(*,*), B;
    C̅ = A̅ B*;
    RETURN;
    CLOSE TWO;
```

1			56		Ø	N	SMRK	
2	77	INL				Ø	BRA	branch round procedure body
3	$\vec{TWO}$	SYL			2	Ø	PDEF	
4	$\vec{A}$	SYT				Ø	LIST	input parameters
5	$\vec{B}$	SYT				Ø	LIST	
6					1	Ø	CASS	assign parameter
7	$\vec{C}$	SYT				Ø	LIST	
8						Ø	LSTE	
9			57		Ø	N	SMRK	markers for declarations
10			58		Ø	N	SMRK	
11			59		Ø	N	SMRK	
12	$\vec{A}$	SYT	$\vec{B}$	SYT		Ø	VMPR	procedure body
13	$\vec{C}$	SYT	12	VAC	Ø	Ø	VASN	
14			6Ø		Ø	N	SMRK	
15			$\vec{TWO}$	SYL		Ø	RTRN	
16			61		Ø	N	SMRK	
17			$\vec{TWO}$	SYL		Ø	RTRN	(note 1.)
18	77	INL				Ø	LBL	end of branch round body

Notes:

1. The close of a procedure always generates a RTRN instruction, even though the last executable statement in the procedure body was a return.

Example 3.5.2: procedure call

```
DECLARE ARRAY(4) SCALAR, A, B;
DECLARE C CHARACTER(5),
        P MATRIX(4,3),
        I INTEGER;
```

CALL THREE([A]+[B],C_3,1.5) ASSIGN($\bar{P}_{1\ \mathrm{TO}\ 3,I+1}$);

#										
1				64			Ø	N	SMRK	
2	$\vec{A}$		SYT	$\vec{B}$		SYT		L	SADD	calculation for 1st input argument
3	4		IMD		1			D	ADLP	
4								N	DLPE	(note 1.)
5	$\vec{C}$		SYT					A	ALC	calculation for 2nd input argument
6	3		IMD		1			A	TIDX	
7								L	ALCE	
8	$\vec{I}$		SYT	$\vec{1}$		LIT		SE	IADD	calculation for 1st assign argument
9	$\vec{P}$		SYT					A	RALC	
10	1		IMD	3	2	IMD		A	TTSB	
11	8		VAC		1			A	TIDX	
12								L	ALCE	
13	$\vec{\mathrm{THREE}}$		SYL	78		INL	3	Ø	CALL	(note 2.)
14	2	S	VAC					Ø	LIST	input list
15	5	C	VAC					Ø	LIST	
16	$\vec{1.5}$	S	LIT					Ø	LIST	
17							1	Ø	CASS	
18	9	V	VAC					Ø	LIST	assign list
19								Ø	LSTE	

Notes:

1. ADLP and DLPE instructions appear if an input argument is an arrayed expression, or an arrayed subscripted variable. They never appear in the case where an assign argument is an arrayed subscripted variable. Equivalent statements may be made about multiple-copy structure terminals.

2. The internal flow numbers 78 and 79 are used in the generation of object text.

3.6 Normal Function Constructs

In this section examples of HALMAT constructs representing user function definitions and invocations, and normal built-in function invocations are presented.

Example 3.6.1: function definition and return

```
FUN: FUNCTION (A̅,B̊) VECTOR;
     DECLARE VECTOR(*), A;
     DECLARE MATRIX(*,*), B;
     RETURN A̅ B̊;
     CLOSE FUN;
```

1				66			Ø	N	SMRK	
2	8Ø		INL					Ø	BRA	branch round function bodY
3	$\vec{FUN}$		SYL				2	Ø	FDEF	
4	$\vec{A}$		SYT					Ø	LIST	argument list
5	$\vec{B}$		SYT					Ø	LIST	
6								Ø	LSTE	
7				67			Ø	N	SMRK	declarations
8				68			Ø	N	SMRK	
9				69			Ø	N	SMRK	
10	$\vec{A}$		SYT	$\vec{B}$		SYT		Ø	VMPR	
11	1Ø		VAC	$\vec{FUN}$		SYL		Ø	RTRN	return
12				7Ø			Ø	N	SMRK	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

13	1		IMD	81		INL		Ø	CALL	(note 1.)
14	8Ø		INL					Ø	LBL	← end of branch round body

Notes:

1. The CALL instruction is designed to be an error exit in the case where the close of the function can be reached.

Example 3.6.2: user function invocation (using Example 3.6.1)

```
{ DECLARE U VECTOR(4), V MATRIX(4,4);
  DECLARE ARRAY(3,2) VECTOR(4), S,W;
  DECLARE ARRAY(3,2) MATRIX(4,4), T; }
```

$[\overline{W}] = FUN(\overline{U},\overset{*}{V}) + FUN(2.5[\overline{S}],[\overset{*}{T}]/[T]_{1,1});$

1				72			Ø	N	SMRK	
2	$\vec{FUN}$		SYT	83		INL	2	F	FUNC	(note 1.)
3	$\vec{U}$	V	SYT					F	LIST	first invocation
4	$\vec{V}$	M	SYT					F	LIST	
5								F	LSTE	
6	$\vec{S}$		SYT	$\vec{2.5}$		LIT		FE	VSPR	evaluation
7	$\vec{T}$		SYT					F	ALC	
8	1		IMD		2			F	TIDX	of arguments
9	1		IMD		1			F	TIDX	
10								F	ALCE	
11	$\vec{T}$		SYT	7		VAC		FE	MSDV	
12	$\vec{FUN}$		SYT	85		INL	2	F	FUNC	(note 1.)
13	6	V	VAC					F	LIST	second invocation
14	11	M	VAC					F	LIST	
15								F	LSTE	(note 2.)

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

16	2		VAC	12		VAC		Ø	VADD	assignment
17	$\vec{W}$		SYT	16		VAC	Ø	Ø	VASN	
18	3		IMD					D	ADLP	arrayness specifiers
19	2		IMD					D	ADLP	
20								D	DLPE	

Notes:

1. The internal flow number appearing in the FUNC instruction and the next consecutive one are used in the generation of object text.

2. Note that the arrayed arguments in this invocation must match the arrayness of the assignment. The implication is that the function is called the number of times specified by the arrayness, not once with an arrayed argument.

Example: 3.6.3: normal built-in functions

{ DECLARE X SCALAR, M MATRIX(4,5); }

$X = SIN(COS(TAN(X))) + DET(\overset{*}{M}_{*,2\ TO\ \#});$

1				98			Ø	N	SMRK	
2	$\vec{TAN}$		SYL	1Ø0		INL	1	FE	FUNC	functions specified inner to outer
3	$\vec{X}$	S	SYT					F	LIST	
4								F	LSTE	
5	$\vec{COS}$		SYL	1Ø2		INL	1	FE	FUNC	
6	2	S	VAC					F	LIST	
7								F	LSTE	
8	$\vec{SIN}$		SYL	1Ø4		INL	1	F	FUNC	
9	5	S	VAC					F	LIST	
10								F	LSTE	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

11	$\vec{M}$		SYT					FE	ALC	
12	2		IMD	5	1	IMD		F	TTSB	
13								F	ALCE	
14	$\vec{DET}$		SYL	1Ø6		INL	1	F	FUNC	
15	11	M	VAC					F	LIST	
16								F	LSTE	
17	8		VAC	14		VAC		Ø	SADD	
18	$\vec{X}$		SYT	17		VAC	Ø	Ø	SASN	assignment

3.7 SHAPING FUNCTION constructs

In this section examples of HALMAT constructs representing SHAPING FUNCTION invocations are presented. The examples presented are not exhaustive and are not guaranteed to show the full complexity of SHAPING FUNCTIONS.

Example 3.7.1: SCALAR and INTEGER function usage

```
DECLARE I INTEGER, S SCALAR;
DECLARE IR ARRAY(3,2) INTEGER,
        SR ARRAY(3,2) SCALAR,
        SQ ARRAY(6) SCALAR;
```

```
I = INTEGER(S);
[IR] = INTEGER([SR]);
[SR] = SCALAR([SQ]);
```

[SR] = $\text{SCALAR}_{3,2}$(4#S+2.5, 2#I-1);

#										
1				71			Ø	N	SMRK	
2	1			156		INL		M	SFST	first INTEGER shaping function
3	-5		FCN	157			1	F	FUNC	
4	$\vec{S}$	S	SYT					F	LIST	
5								F	LSTE	
6	1			156		INL		M	SFND	
7	$\vec{I}$		SYT	3		VAC	Ø	Ø	IASN	assignment
8				72			Ø	N	SMRK	
9	1			158		INL		M	SFST	
10	-5		FCN	159			1	F	FUNC	
11	3		IMD		2			F	ASIZ	
12	2		IMD		1			F	ASIZ	2nd INTEGER function
13	$\vec{SR}$	S	SYT					F	LIST	
14								F	LSTE	(note 1.)
15	1			158		INL		M	SFND	
16	$\vec{IR}$		SYT	12		VAC	Ø	Ø	IASN	assignment
17	3		IMD					D	ADLP	arrayness specifiers
18	2		IMD					D	ADLP	
19								D	DLPE	
20				73			Ø	N	SMRK	
21	1			16Ø		INL		M	SFST	
22	-6		FCN	161			1	F	FUNC	
23	3		IMD		2			F	ASIZ	
24	2		IMD		1			F	ASIZ	1st SCALAR function
25	$\vec{SQ}$	S	SYT					F	LIST	
26								F	LSTE	(note 1.)
27	1			16Ø			INL	M	SFND	

28	$\vec{SR}$		SYT	24		VAC	Ø	Ø	SASN	assignment
29	3		IMD					D	ADLP	arrayness specifiers
30	2		IMD					D	ADLP	
31								D	DLPE	
32				74			Ø	N	SMRK	
33	1			162		INL		M	SFST	2nd SCALAR function
34	$\vec{S}$		SYT	$\vec{2.5}$		LIT		F	SADD	
35	1			162		INL		M	SFAR	
36	$\vec{I}$		SYT	$\vec{1}$		LIT		F	ISUB	
37	-6		FCN	163			2	F	FUNC	
38	3		IMD		2			F	ASIZ	(note 2.)
39	2		IMD		1			F	ASIZ	
40	34	S	VAC	$\vec{4}$	I	LIT		F	LIST	
41	36	I	VAC	$\vec{2}$	I	LIT		F	LIST	
42								F	LSTE	
43	1			162		INL		M	SFND	
44	$\vec{SR}$		SYT	37		VAC	Ø	Ø	IASN	assignment
45	3		IMD					D	ADLP	arrayness specifiers
46	2		IMD					D	ADLP	
47								D	DLPE	

Notes:

1. Notice that when the shaping function has only one argument the arrayness of the argument need not match that of the assignment, and still the function does not require subscripting.

2. This shaping function requires subscripting to identify its arrayness because there are two arguments. Note the positions of the HALMAT text for evaluating each argument. If both arguments were symbol-table variables, a vestigial SFAR instruction would still be left. In the general case, the expressions might be arrayed, in which case arrayness specifiers would appear immediately before the SFAR separator (or before the FUNC instruction in the case of the last argument).

Example 3.7.2: CHAR and BIT function usage

```
DECLARE B BIT(42), B1 ARRAY(5) BIT(10),
        C1 CHARACTER(12),
        C2 CHARACTER(10),
        C3 ARRAY(5) CHARACTER VARYING,
        V VECTOR(2),
        T ARRAY(3) SCALAR,
        S SCALAR;
```

$\dot{B}$ = BIT$_{I\ TO\ \#}$(S);

[$\dot{B1}$] = BIT$_{@HEX}$($\acute{C}1$,4#$\acute{C}2$);

[$\acute{C}3$] = CHAR$_{5:}$([T],$\bar{V}$);

1				76			Ø	N	SMRK	
2	1			164		INL		M	SFST	1st BIT function (note 1.)
3	-1		FCN	165			1	F	FUNC	
4	32		IMD		Ø			F	TSIZ	
5	$\vec{I}$		SYT	32		IMD		F	TTSB	
6	$\vec{S}$	S	SYT					F	LIST	
7								F	LSTE	
8	1			164		INL		M	SFND	
9	$\vec{B}$		SYT	3		VAC	Ø	Ø	BASN	assignment

10				77			Ø	N	SMRK	
11	1			166		INL		M	SFST	2nd BIT function (note 2.)
12	1			166		INL		M	SFAR	
13	-3		FCN	167		4	2	F	FUNC	
14	5		IMD		1			F	ASIZ	
15	48		IMD		Ø			F	TSIZ	
16	$\vec{C}1$	C	SYT					F	LIST	
17	$\vec{C}2$	C	SYT	$\vec{4}$		LIT		F	LIST	
18								F	LSTE	
19	1			166		INL		M	SFND	
20	$\vec{B}1$		SYT	13		VAC	Ø	Ø	BASN	assignment
21	5		IMD					D	ADLP	arrayness specifiers
22								D	DLPE	
23				78			Ø	N	SMRK	
24	1			168		INL		M	SFST	CHAR function (note 3.)
25	1			168		INL		M	SFAR	
26	-2		FCN	169			2	F	FUNC	
27	5		IMD		1			F	ASIZ	
28	$\vec{T}$	S	SYT					F	LIST	
29	$\vec{V}$	V	SYT					F	LIST	
30								F	LSTE	
31	1			168		INL		M	SFND	
32	$\vec{C}3$		SYT	26		VAC	Ø	Ø	CASN	assignment
33	5		IMD					D	ADLP	arrayness specifiers
34								D	DLPE	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Notes:

1. The terminal size of the function given by the TSIZ instruction is the number of bits in the symbol table variable S (assumed single precision). The following TTSB instructions calls for truncation of those 32 bits.

2. The array size of the function given by the ASIZ instruction is the number of elements in the argument list. The terminal size of the function given by the TSIZ instruction is the maximum of the lengths of arguments, in this case 12 chars.= 48 bits.

3. The array size of the function given by the ASIZ instruction was explicit in the source text. No terminal size specification is made because the CHAR function returns a VARYING character string.

Example 3.7.3: MATRIX and VECTOR function usage

$$\left\{\begin{array}{l} \text{DECLARE B BIT, I INTEGER,} \\ \qquad\quad \text{S SCALAR, V VECTOR(12),} \\ \qquad\quad \text{M1 MATRIX(3,3),} \\ \qquad\quad \text{M ARRAY(3) MATRIX(2,2);} \end{array}\right\}$$

$$\bar{V} = \text{VECTOR}(S,\dot{B},I,\overset{*}{M}1);$$

$$[\overset{*}{M}] = \text{MATRIX}_{3:2,*}(2\#\overset{*}{M}1_{2\text{ AT }2,*});$$

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

#										
1				82			Ø	N	SMRK	
2	1			172		INL		M	SFST	VECTOR function (note 1.)
3	1			172		INL		M	SFAR	
4	1			172		INL		M	SFAR	
5	1			172		INL		M	SFAR	
6	-7		FCN	173			4	F	FUNC	
7	12		IMD		1			F	TSIZ	
8	$\vec{S}$	S	SYT					F	LIST	
9	$\vec{B}$	B	SYT					F	LIST	
10	$\vec{I}$	I	SYT					F	LIST	
11	$\vec{M1}$	M	SYT					F	LIST	
12								F	LSTE	
13	1			172		INL		M	SFND	
14	$\vec{V}$		SYT	6		VAC	Ø	Ø	VASN	assignment
15				83			Ø	N	SMRK	
16	1			174		INL		M	SFST	MATRIX function
17	$\vec{M1}$		SYT					F	ALC	
18	2		IMD	2	2	IMD		F	TASB	
19								F	ALCE	

#									
20	-8		FCN	175			1	F	FUNC
21	3		IMD		1			F	ASIZ
22	2		IMD		2			F	TSIZ
23	2		IMD		1			F	TSIZ
24	17	M	VAC					F	LIST
25								F	LSTE
26	1			174		INL		M	SFND
27	$\vec{M}$		SYT	2Ø		VAC	Ø	Ø	MASN
28	3		IMD					D	ADLP
29								D	DLPE

Rows 20–26: (note 2.)

Row 27: assignment

Rows 28–29: arrayness specifiers

Notes:

1. The terminal size of the function is the total number of data elements in the argument list. Note the three vestigial SFAR instructions.

2. Here the total number of data elements in the argument list is 12. Hence, the function subscript * is computed as 2 in PHASE I and is specified in the second TSIZ instruction.

Example 3.7.4: CONTENT pseudovariable usage

```
{ DECLARE S ARRAY(3) SCALAR,  }
{         B ARRAY(3) BIT(1);  }
CONTENT_10([S]) = [B];
```

1				86			Ø	N	SMRK	
2	1			178		INL		M	SFST	CONTENT pseud-variable (note 1.)
3	-9		FCN	179			1	F	ZRFN	
4	3		IMD		1			F	ASIZ	
5	1Ø		IMD		Ø			F	TSIZ	
6	$\vec{S}$	S	SYT					F	LIST	
7								F	LSTE	
8	1			178		INL		M	SFND	
9	3		VAC	$\vec{B}$		SYT	Ø	Ø	BASN	assignment
10	3		IMD					D	ADLP	arrayness specifiers
11								D	DLPE	

Notes:

1. This construct is exactly like any shaping function construct, except that ZRFN is used instead of FUNC because CONTENT is a RECEIVER.

3.8 Initialization Constructs

In this section are presented the constructs corresponding to constant and initial declarations.

Example 3.8.1: scalar variables

```
DECLARE S SCALAR CONSTANT(3.5),
        T ARRAY(2) SCALAR AUTOMATIC INITIAL (-4.5,1.5);
```

1				132			Ø	N	SMRK	
2	111		INL	11Ø		GLI		II	ENTS	enter static
3	$\vec{S}$		SYT	$\overrightarrow{3.5}$	Ø	LIT		I1	SINT	first variable
4	111		INL					I1	EXTS	exit static
5	$\vec{T}$		SYT	112		GLI	Ø	IW	STRI	second variable
6	1		IMD	$\overrightarrow{4.5}$	1	LIT		IW	SINT	
7	2		IMD	$\overrightarrow{1.5}$	Ø	LIT		IW	SINT	
8				133			Ø	N	SMRK	
9				11Ø		GLI		II	ENDS	end static (note 1.)

Notes:

1. This instruction appears at the end of the group of declarations in a name-scope: here the two declarations are assumed to be the only ones.

Example 3.8.2: vectors and characters

```
DECLARE C CHARACTER(4) CONSTANT('BANG');
DECLARE VV VECTOR(3) STATIC INITIAL(5.5);
DECLARE VW VECTOR(3) AUTOMATIC INITIAL(1.5,-3.0,4.0);
```

Example 3.8.3: arrays and repeated data

```
DECLARE VX ARRAY(4) VECTOR(3) CONSTANT(2#3.3,-2.5);
DECLARE VZ ARRAY(4) VECTOR(3) CONSTANT(5#0.0,6.0,6#-3.0);
```

	Operand 1			Operand 2						
1				14∅			∅	N	SMRK	
2	125		INL	124		GLI		II	ENTS	enter static
3	$\vec{VX}$		SYT	126		GLI	1	I1	STRI	first variable with one repeated loop
4	126		INL				1	I1	DLPI	
5	2		IMD	1		IMD	2	I1	DLPI	
6	1		IMD	$\vec{3}.3$	∅	LIT		I1	SINT	
7	126		INL					I1	DLEI	
8	3		IMD	$\vec{2}.5$	1	LIT		I1	SINT	
9	4		IMD					D	ADLP	(note 1.)
10								D	DLPE	
11	125		INL					I1	EXTS	exit static
12				141			∅	N	SMRK	
13	127		INL	124		GLI		II	ENTS	enter static
14	$\vec{V2}$		SYT	128		GLI	2	I1	STRI	
15	128		INL				1	I1	DLPI	

16	5		IMD	1		IMD	2	I1	DLPI	
17	1		IMD	$\vec{0.0}$	Ø	LIT		I1	SINT	second variable with two repeated loops
18	128		INL					I1	DLEI	
19	6		IMD	$\vec{6.0}$	Ø	LIT		I1	SINT	
20	129		INL				1	I1	DLPI	
21	6		IMD	1		IMD	2	I1	DLPI	
22	7		IMD	$\vec{3.0}$	1	LIT		I1	SINT	
23	129		INL					I1	DLEI	(note 1.)
24	127		INL					I1	EXTS	exit static
25				142			Ø	N	SMRK	
26				124		GLI		II	ENDS	end static

Notes:

1. In the first variable each vector of the array is initialized with the same values: hence, the arrayness specifiers. In the second variable every element of each vector is initialized individually. Hence there are no arrayness specifiers.

REFERENCES

1. 'A Guide to the HAL Programming Language', Document #MSC-01846, Intermetrics, Inc., Cambridge, Mass.

2. 'The Programming Language HAL - A Specification', Document #MSC-01848, Intermetrics, Inc., Cambridge, Mass.

APPENDIX A.

HAL360-V1 symbol and literal tables

The HAL symbol table consists of a group of arrays of various sizes, maintained by PHASE I of the HAL 360-V1 compiler. This discussion describes the set of ten of these arrays which are located in Common storage for transmission to PHASE II. Eight of these are parallel arrays, containing an entry for each location in the symbol table. Their size is SYT_SIZE, the total capacity of the table. The arrays, and their lengths in bits, are:

NAME_LENGTH	(8)
SYT_CLASS	(8)
SYT_TYPE	(8)
SYT_FLAGS	(16)
VAR_LENGTH	(16)
PRECISION	(16)
SYT_ARRAY	(16)
SYT_PTR	(16)

The important declarations for these are reproduced in Figure A.1.

HAL names are from one to thirty-two characters long. They are stored in the character array SYT, after being padded on the right with blanks to length 32. Names are stored in order of their first occurrence in the listing. Each SYT string stores up to eight HAL names. The actual length of each name is stored in NAME_LENGTH. Names of minor structure and structure terminal components of qualified structures are stored in SYT in the form by which they must be referenced by the user (e.g., A.B.C). This imposes a 32-character limit on the length of such compound names.

The array SYT_CLASS indicates whether an entry is a variable, label, function, or a 'replaced' name. The class LFUNC_CLASS applies only to certain built-in functions. The types, as indicated in SYT_TYPE, are divided into three groups. The first group is associated with variable and function names; the second group applies to labels, and the third to structures. For user-defined variables and functions, there are six types; the remaining two, BORC_TYPE and IORS_TYPE, are used only to describe parameters of certain built-in functions (see below). The type entry (and the entries in VAR_LENGTH, PRECISION, etc.) for function names describes the quantity which the function returns as the result of its invocation.

```
DECLARE SYTSIZE LITERALLY '399',
        SYTCHAR LITERALLY '49';

   COMMON  SYT(SYTCHAR) CHARACTER;
   COMMON  SYT_CLASS(SYTSIZE) BIT(8);
   DECLARE VAR_CLASS   BIT(8) INITIAL (1),
           LABEL_CLASS BIT(8) INITIAL (2),
           FUNC_CLASS  BIT(8) INITIAL (3),
           STRUC_CLASS BIT(8) INITIAL (4),
           LFUNC_CLASS BIT(8) INITIAL (5),
           REPL_CLASS  BIT(8) INITIAL (6);

   COMMON  SYT_TYPE(SYTSIZE) BIT(8);
   DECLARE BIT_TYPE        BIT(8) INITIAL (1),
           CHAR_TYPE       BIT(8) INITIAL (2),
           MAT_TYPE        BIT(8) INITIAL (3),
           VEC_TYPE        BIT(8) INITIAL (4),
           SCALAR_TYPE     BIT(8) INITIAL (5),
           INT_TYPE        BIT(8) INITIAL (6),
           BORC_TYPE       BIT(8) INITIAL (7),
           IORS_TYPE       BIT(8) INITIAL (8),

           MB_STMT_LAB     BIT(8) INITIAL ("40"),
           IND_STMT_LAB    BIT(8) INITIAL ("41"),
           STMT_LABEL      BIT(8) INITIAL ("42"),
           UNSPEC_LABEL    BIT(8) INITIAL ("43"),
           MB_CALL_LAB     BIT(8) INITIAL ("44"),
           IND_CALL_LAB    BIT(8) INITIAL ("45"),
           CALLED_LABEL    BIT(8) INITIAL ("46"),
           PROC_LABEL      BIT(8) INITIAL ("47"),
           TASK_LABEL      BIT(8) INITIAL ("48"),
           PROG_LABEL      BIT(8) INITIAL ("49"),

           MAJ_STRUC       BIT(8) INITIAL ("F0"),
           MIN_STRUC       BIT(8) INITIAL ("F1");

   COMMON  SYT_FLAGS(SYTSIZE) BIT(16);
   DECLARE LOCK_MASK      BIT(16) INITIAL ("0003"),
           DENSE_FLAG     BIT(16) INITIAL ("0004"),
           ALIGNED_FLAG   BIT(16) INITIAL ("0008"),
           IMP_DECL       BIT(16) INITIAL ("0010"),
           ASSIGN_PARM    BIT(16) INITIAL ("0020"),
           DEFINED_LABEL  BIT(16) INITIAL ("0040"),
           DEF_TYPE_FLAG  BIT(16) INITIAL ("0080"),
           AUTO_FLAG      BIT(16) INITIAL ("0100"),
           STATIC_FLAG    BIT(16) INITIAL ("0200"),
           INPUT_PARM     BIT(16) INITIAL ("0400"),
           INIT_FLAG      BIT(16) INITIAL ("0800"),
           CONSTANT_FLAG  BIT(16) INITIAL ("1000"),
           QUAL_FLAG      BIT(16) INITIAL ("2000"),
           ENDSCOPE_FLAG  BIT(16) INITIAL ("4000"),
           INACTIVE_FLAG  BIT(16) INITIAL ("8000"),
```

Figure A.1 symbol table declarations

Four types of label in the second type-group will exist after PHASE I only in the event of a source program error which leaves these labels insufficiently defined: MB_STMT_LAB, UNSPEC_LABEL, MB_CALL_LAB, and CALLED_LABEL. The types IND_STMT_LAB and IND_CALL_LAB result when a GO TO or CALL refers to a name in an outer name-scope; the content of SYT_PTR is the location index in the symbol table of the corresponding outer name.

If the PROGRAM being compiled, or any PROCEDURE, TASK, or FUNCTION defines any local variables, the SYT_PTR entry for the label of that name-scope contains the symbol table index of the first such local variable. "Local names" are those names which are known only inside the block, including names known only inside inner blocks, etc.

The third type-group applies to structures, which are described below in a separate section.

Only those flags in the SYT_FLAGS entry which are of interest beyond PHASE I will be described here. The LOCK_MASK bits contain the LOCK_TYPE for the variable involved; the all-ones value and the zero value both indicate lock-type zero. DENSE_FLAG, ALIGNED_FLAG, AUTO_FLAG, and STATIC_FLAG reflect the specified or default attributes of the entry. The DEFINED_LABEL flag is a zero when the statement, procedure, function, etc., associated with a label has not been found. The INPUT_PARM and ASSIGN_PARM flags are ones when the name appears as a formal input parameter or a formal assign parameter, respectively, of a procedure or function. The CONSTANT flag is a one when a variable has been declared to be a constant. Finally, the ENDSCOPE_FLAG is a one whenever the name is the last local variable of a name-scope (including names local to inner blocks).

VAR_LENGTH contains bit-string, character-string, and vector lengths; the entry is treated as two one-byte entries for matrices, the row-dimension being the high-order byte, and the column-dimension the other. In all of these cases, a field or subfield of all ones indicates unspecified length, to be determined at run-time. Precision is indicated in PRECISION for scalars, vectors, and matrices; values of 1 or 2 are put there for character variables to indicate fixed and varying length, respectively.

Array characteristics are kept in SYT_ARRAY, and in a subsidiary array EXT_ARRAY. If a name is not an array, SYT_ARRAY contains a zero. If the name is a one-dimensional array, SYT_ARRAY contains the size (a positive number), or all ones (-1),

indicating unspecified length (*). If the array is two or more dimensional, SYT_ARRAY contains a number less than -1. The negative of this value is an index into EXT_ARRAY, at which three or more words are stored. The first specifies the number of dimensions of the array, and that number of subsequent words contain their upper limits (-1 again indicates unspecified length).

Structures and their components all appear as STRUC_CLASS. The type entry is one of MAJ_STRUC, MIN_STRUC, or one of the six variable types. The SYT_PTR entry for each contains the declared level for that name. Array and structure-copy information is stored in SYT_ARRAY and EXT_ARRAY, as above. Structure terminals have entries like variables in SYT_TYPE, VAR_LENGTH, etc. Every major or minor structure which contains a terminal declared to be a constant has its own CONSTANT_FLAG turned on. Note, however, that the array information stored with a STRUC_CLASS name applies to that name only, and does not reflect copy-dimensionality due to higher level declarations. If a structure is the last local name in a name scope, the last structure terminal declared will be the one whose ENDSCOPE_FLAG is set.

Built-in function names and their parameter descriptions are stored in the last locations of the table. Generally, these follow the normal rules for variables and user-defined functions, but there are several exceptions, described below.

Those functions of LFUNC_CLASS have SYT_PTR entries which merely identify the function (presently, 1-5). Since LFUNC_CLASS functions may have a list of arguments of unspecified length, it is useful for technical reasons to have their DEFINED_LABEL flag zero. Functions which return a vector or square matrix of the same dimension as their argument contain hex FF00 in VAR_LENGTH. The TRANSPOSE function returns an (m,n) matrix if its argument is (n,m); this is denoted by hex 00FF in VAR_LENGTH.

The formal parameters of built-in functions are stored normally, except that they are multiply-used, where possible, to save space. Also, arguments which must be square matrices have hex FF00 in VAR_LENGTH.

The HAL literal table consists of a group of four arrays located in Common storage for transmission to PHASE II. The important declarations are reproduced in Figure A.2.

```
DECLARE LIT_CHARSIZE LITERALLY '20';
DECLARE LITSIZE LITERALLY '200';
DECLARE DW_SIZE LITERALLY '200';
DECLARE DW_TOP FIXED;
DECLARE LIT_TOP FIXED;
COMMON DW(DW_SIZE) FIXED;
COMMON LIT_INDEX(LITSIZE) FIXED;
COMMON LIT_LENGTH(LITSIZE) BIT(8);
COMMON LIT_CHAR(LIT_CHARSIZE) CHARACTER;
```

Figure A.2 Literal table declarations

The original literal character strings are stored in the character array LIT_CHAR, which is accessed by LIT_INDEX. The contents of bits 16-23 of LIT_INDEX access an entry in LIT_CHAR, and the contents of bits 24-31 of LIT_INDEX point to a literal character string in that entry. LIT_LENGTH gives the length of the character string minus one.

For appropriate literals a double-word floating point version is held in DW, which is accessed by bits 0-15 of LIT_INDEX.

For a literal which is preceded by 'BIN', 'OCT', or 'HEX' and a repeat number n, an indirect form of entry is employed. The primary character string pointed to is one which gives information on the actual literal itself. This primary character string is nine characters long and of the form:

xaabbcccc

where

x = { B for 'BIN'; O for 'OCT'; H for 'HEX'; D for 'DEC' }

bb is a 16 bit pointer to the actual literal character string

aa is a 16 bit pointer to the literal character string specified as a repeat. If there are no repeats, it is a 16 bit zero.

cc is a 32 bit integer giving the size in bits of the literal including the effect of the repeat.

LIT_LENGTH gives the length of the actual literal character string minus one as before.

APPENDIX B: The HALMAT Instruction Set

AASB	019	array at-subscript specifier
ADLP	00D	arrayness specifier
AIDX	015	array index specifier
ALC	024	allocate subscript header
ALCE	026	allocate subscript end
ASIZ	049	arrayness of shaping function
ATSB	01D	array to-subscript specifier
BAND	102	bit-string and
BASN	101	bit-string assign
BBRA	013	bit-string branch on false
BCAT	105	bit-string catenate
BEQU	722	bit-string equal
BINT	821	bit-string initialize
BMRK	005	end of block marker
BNEQ	721	bit-string not equal
BNOT	104	bit-string complement
BOR	103	bit-string or
BRA	010	unconditional branch
BTOI	621	bit to integer conversion
BTOS	521	bit to scalar conversion
CALL	029	procedure call header
CAND	7E2	conditional and
CASN	201	character assign
CASS	03B	assign parameter list header
CBRA	011	computed branch
CCAT	202	character catenate
CEQU	742	character equal
CGT	744	character greater than
CINT	841	character initialize

CLBL	009	computed-branch label
CLT	746	character less than
CMRK	006	end of code marker
CNEQ	741	character not equal
CNGT	743	character not greater than
CNLT	745	character not less than
CNOT	7E4	conditional complement
COR	7E3	conditional or
DFOR	050	do for header
DFRE	054	do for end
DGPE	053	do for list terminator
DGPI	052	integer do for list element
DGPS	051	scalar do for list element
DLEI	803	initialize loop end
DLPE	00E	end of array- and structureness specification
DLPI	802	initialize loop header
ENDS	023	end of static bypass block
ENTS	021	enter static bypass block
EXTS	022	exit static bypass block
FASN	039	file assignment
FBRA	012	false branch on condition
FCLM	03F	column i/o control specifier
FDEF	02C	function definition header
FILE	034	file i/o specifier
FLIN	041	line i/o control specifier
FLP	020	precision converter
FPGE	040	page i/o control specifier
FSKP	03D	skip i/o control specifier
FTAB	03E	tab i/o control specifier
FUNC	028	function call header
IADD	6CB	integer add
IASN	601	integer assign

IEQU	7C6	integer equal
IEXP	6CF	integer exponentiation
IGT	7C8	integer greater than
IINT	8C1	integer initialize
IIPR	6CD	integer multiply
ILT	7CA	integer less than
INEG	6D0	integer negate
INEQ	7C5	integer not equal
INGT	7C7	integer not greater than
INLT	7C9	integer not less than
ISUB	6CC	integer subtract
ITOC	2C1	integer to character conversion
ITOS	5C1	integer to scalar conversion
LBL	008	label
LIST	03C	subprogram and i/o argument
LSTE	03A	subprogram and i/o argument list end
MADD	362	matrix add
MASN	301	matrix assign
MDEF	036	program definition header
MEQU	762	matrix equal
MINT	861	matrix initialize
MINV	3CA	matrix invert
MMPR	368	matrix-matrix product
MNEG	344	matrix negate
MNEQ	761	matrix not equal
MSDV	3A6	matrix divide-by-scalar
MSPR	3A5	matrix multiply-by-scalar
MSUB	363	matrix subtract
MTRA	329	matrix transpose
MVPR	46C	matrix-vector product
PDEF	02D	procedure definition header

RALC	025	receiver allocate subscript specifier
RDAL	02F	read-all header
READ	02A	read header
RTRN	030	subprogram return
SADD	5AB	scalar add
SASB	01A	structure at-subscript specifier
SASN	501	scalar assign
SDLP	00C	structureness specifier
SEQU	7A6	scalar equal
SEXP	5AF	scalar exponentiation-by-scalar
SFAR	04B	shaping function argument separator
SFND	04C	end of shaping function specification
SFST	04A	start of shaping function specification
SGT	7A8	scalar greater than
SIDX	016	structure index specifier
SIEX	571	scalar exponentiation-by-integer
SINT	8A1	scalar initialize
SLT	7AA	scalar less than
SMRK	004	statement marker
SNEG	5B0	scalar negate
SNEQ	7A5	scalar not equal
SNGT	7A7	scalar not greater than
SNLT	7A9	scalar not less than
SPEX	572	scalar exponentiation by positive integer
SSDV	5AE	scalar divide
SSPR	5AD	scalar multiply
SSUB	5AC	scalar subtract
STOC	2A1	scalar to character conversion
STOI	6A1	scalar to integer conversion
STRI	801	repeated initialize specifier
STSB	01E	structure to-subscript specifier
TASB	018	terminal at-subscript specifier

TASN	038	structure assign
TDEF	035	task definition
TEQU	045	structure equal
TERM	032	terminate program execution
TIDX	014	terminal index specifier
TNEQ	044	structure not equal
TSIZ	048	terminal size of shaping function
TTSB	01C	terminal to-subscript specifier
UDEF	033	update block header
UEND	037	update block end
VADD	482	vector add
VASN	401	vector assign
VCRS	48B	vector cross product
VDOT	581	vector dot product
VEQU	782	vector equal
VINT	881	vector initialize
VMPR	4CD	vector matrix product
VNEG	444	vector negate
VNEQ	781	vector not equal
VSDV	4A6	vector divide-by-scalar
VSPR	4A5	vector multiply-by-scalar
VSUB	483	vector subtract
VVPR	387	vector outer product
WRIT	02B	write header
ZRFN	04D	receiver function invocation

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

APPENDIX C: Mnemonics of HALMAT operation codes

Code	Mnemonic	Code	Mnemonic
000		030	RTRN
001		031	
002		032	TERM
003		033	UDEF
004	SMRK	034	FILE
005	BMRK	035	TDEF
006	CMRK	036	MDEF
007		037	UEND
008	LBL	038	TASN
009	CLBL	039	FASN
00A		03A	LSTE
00B		03B	CASS
00C	SDLP	03C	LIST
00D	ADLP	03D	FSKP
00E	DLPE	03E	FTAB
00F		03F	FCLM
010	BRA	040	FPGE
011	CBRA	041	FLIN
012	FBRA	042	
013	BBRA	043	
014	TIDX	044	TNEQ
015	AIDX	045	TEQU
016	SIDX	046	
017		047	
018	TASB	048	TSIZ
019	AASB	049	ASIZ
01A	SASB	04A	SFST
01B		04B	SFAR
01C	TTSB	04C	SFND
01D	ATSB	04D	ZRFN
01E	STSB	04E	
01F		04F	
020	FLP	050	DFOR
021	ENTS	051	DGPS
022	EXTS	052	DGPI
023	ENDS	053	DGPE
024	ALC	054	DFRE
025	RALC	055	
026	ALCE	056	
027		057	
028	FUNC	058	
029	CALL	059	
02A	READ	05A	
02B	WRIT	05B	
02C	FDEF	05C	
02D	FDEF	05D	
02E		05E	
02F	RDAL	05F	

100	
101	BASN
102	BAND
103	BOR
104	BNOT
105	BCAT
106	
107	
108	
109	
10A	
10B	
10C	
10D	
10E	
10F	

200	
201	CASN
202	CCAT
203	
204	
205	
206	
207	
208	
209	
20A	
20B	
20C	
20D	
20E	
20F	

		2C0 2C1 2C2 2C3 2C4 2C5 2C6 2C7 2C8 2C9 2CA 2CB 2CC 2CD 2CE 2CF	 ITOC		
2A0 2A1 2A2 2A3 2A4 2A5 2A6 2A7 2A8 2A9 2AA 2AB 2AC 2AD 2AE 2AF	 STOC				

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

300				360	
301	MASN			361	
302				362	MADD
303				363	MSUB
304				364	
305				365	
306				366	
307				367	
308				368	MMPR
309				369	
30A				36A	
30B				36B	
30C				36C	
30D				36D	
30E				36E	
30F				36F	
		340			
		341			
		342			
		343			
		344	MNEG		
		345			
		346			
		347			
		348			
		349			
		34A			
		34B			
		34C			
		34D			
		34E			
		34F			
320				380	
321				381	
322				382	
323				383	
324				384	
325				385	
326				386	
327				387	VVPR
328				388	
329	MTRA			389	
32A				38A	
32B				38B	
32C				38C	
32D				38D	
32E				38E	
32F				38F	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

		3C0 3C1 3C2 3C3 3C4 3C5 3C6 3C7 3C8 3C9 3CA 3CB 3CC 3CD 3CE 3CF	 MINV		
3A0 3A1 3A2 3A3 3A4 3A5 3A6 3A7 3A8 3A9 3AA 3AB 3AC 3AD 3AE 3AF	 MSPR MSDV				

INTERMETRICS INCORPORATED · 380 GREEN STREET · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 868-1840

400				460	
401	VASN			461	
402				462	
403				463	
404				464	
405				465	
406				466	
407				467	
408				468	
409				469	
40A				46A	
40B				46B	
40C				46C	MVPR
40D				46D	VMPR
40E				46E	
40F				46F	
		440			
		441			
		442			
		443			
		444	VNEG		
		445			
		446			
		447			
		448			
		449			
		44A			
		44B			
		44C			
		44D			
		44E			
		44F			
				480	
				481	
				482	VADD
				483	VSUB
				484	
				485	
				486	
				487	
				488	
				489	
				48A	
				48B	VCRS
				48C	
				48D	
				48E	
				48F	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

4A0	
4A1	
4A2	
4A3	
4A4	
4A5	VSPR
4A6	VSDV
4A7	
4A8	
4A9	
4AA	
4AB	
4AC	
4AD	
4AE	
4AF	

500					
501	SASN				
502					
503					
504					
505					
506					
507					
508					
509					
50A					
50B					
50C					
50D					
50E					
50F					
				570	
				571	SIEX
				572	SPEX
				573	
				574	
				575	
				576	
				577	
				578	
				579	
				57A	
				57B	
				57C	
				57D	
				57E	
				57F	
520				580	
521	BTOS			581	VDOT
522				582	
523				583	
524				584	
525				585	
526				586	
527				587	
528				588	
529				589	
52A				58A	
52B				58B	
52C				58C	
52D				58D	
52E				58E	
52F				58F	

		5C0			
		5C1	ITOS		
		5C2			
		5C3			
		5C4			
		5C5			
		5C6			
		5C7			
		5C8			
		5C9			
		5CA			
		5CB			
		5CC			
		5CD			
		5CE			
		5CF			
5A0					
5A1					
5A2					
5A3					
5A4					
5A5					
5A6					
5A7					
5A8					
5A9					
5AA					
5AB	SADD				
5AC	SSUB				
5AD	SSPR				
5AE	SSDV				
5AF	SEXP				
5B0	SNEG				
5B1					
5B2					
5B3					
5B4					
5B5					
5B6					
5B7					
5B8					
5B9					
5BA					
5BB					
5BC					
5BD					
5BE					
5BF					

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

600 601 602 603 604 605 606 607 608 609 60A 60B 60C 60D 60E 60F	IASN				
620 621 622 623 624 625 626 627 628 629 62A 62B 62C 62D 62E 62F	BTOI				

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

		6C0	
		6C1	
		6C2	
		6C3	
		6C4	
		6C5	
		6C6	
		6C7	
		6C8	
		6C9	
		6CA	
		6CB	IADD
		6CC	ISUB
		6CD	IIPR
		6CE	
		6CF	IEXP
6A0		6D0	INEG
6A1	STOI	6D1	
6A2		6D2	
6A3		6D3	
6A4		6D4	
6A5		6D5	
6A6		6D6	
6A7		6D7	
6A8		6D8	
6A9		6D9	
6AA		6DA	
6AB		6DB	
6AC		6DC	
6AD		6DD	
6AE		6DE	
6AF		6DF	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

				760 761 762 763 764 765 766 767 768 769 76A 76B 76C 76D 76E 76F	MNEQ MEQU
		740 741 742 743 744 745 746 747 748 749 74A 74B 74C 74D 74E 74F	CNEQ CEQU CNGT CGT CNLT CLT		
720 721 722 723 724 725 726 727 728 729 72A 72B 72C 72D 72E 72F	BNEQ BEQU			780 781 782 783 784 785 786 787 788 789 78A 78B 78C 78D 78E 78F	VNEQ VEQU

INTERMETRICS INCORPORATED · 380 GREEN STREET · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 868-1840

		7C0 7C1 7C2 7C3 7C4 7C5 7C6 7C7 7C8 7C9 7CA 7CB 7CC 7CD 7CE 7CF	 INEQ IEQU INGT IGT INLT ILT		
7A0 7A1 7A2 7A3 7A4 7A5 7A6 7A7 7A8 7A9 7AA 7AB 7AC 7AD 7AE 7AF	 SNEQ SEQU SNGT SGT SNLT SLT				
		7E0 7E1 7E2 7E3 7E4 7E5 7E6 7E7 7E8 7E9 7EA 7EB 7EC 7ED 7EE 7EF	 CAND COR CNOT		

800				860	
801	STRI			861	MINT
802	DLPI			862	
803	DLEI			863	
804				864	
805				865	
806				866	
807				867	
808				868	
809				869	
80A				86A	
80B				86B	
80C				86C	
80D				86D	
80E				86E	
80F				86F	
		840			
		841	CINT		
		842			
		843			
		844			
		845			
		846			
		847			
		848			
		849			
		84A			
		84B			
		84C			
		84D			
		84E			
		84F			
820				880	
821	BINT			881	VINT
822				882	
823				883	
824				884	
825				885	
826				886	
827				887	
828				888	
829				889	
82A				88A	
82B				88B	
82C				88C	
82D				88D	
82E				88E	
82F				88F	

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

		8C0 8C1 8C2 8C3 8C4 8C5 8C6 8C7 8C8 8C9 8CA 8CB 8CC 8CD 8CE 8CF	IINT		
8A0 8A1 8A2 8A3 8A4 8A5 8A6 8A7 8A8 8A9 8AA 8AB 8AC 8AD 8AE 8AF	SINT				

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

APPENDIX D: Numerical codes for subfield mnemonics

1. Qualifier subfield mnemonics

Following are the numerical codes for the mnemonics to be found in subfields a_q and b_q of HALMAT instructions.

VARIABLE OPERAND:

SYT	1	IMD	4
VAC	2	GLI	5
LIT	3		

LABEL OPERAND:

INL	Ø	ABL	2
SYL	1		

SPECIAL PURPOSE:

AST, SST	6	FCN	7

2. Optimization code mnemonics

Following are the hex decimal numerical codes for the mnemonics to be found in the code optimizer subfield r_{co} of HALMAT instructions.

Ø	0	AE	9
E	1	D	A
IW	3	M	B
I1	4	S	C
II	5	SE	D
F	6	N	E
FE	7	L	F
A	8		

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

3. Operand type mnemonics

Following are the numerical codes for the mnemonics occasionally encountered in subfields a_t and b_t of HALMAN instructions.

B	1	S	5	
C	2	I	6	
M	3	T	0	(not as defined in PHASE I)
V	4			

APPENDIX E: SHAPING FUNCTION information

1. Numerical codes for radix values

Some SHAPING FUNCTIONS have conversion radix information given in subfield b_q. Following are the possible code values.

no radix information	Ø
binary	1
octal	3
hexadecimal	4

2. Numerical codes for SHAPING FUNCTIONS

The numerical codes appearing in subfield a_{op} of the FUNC instruction for SHAPING FUNCTIONS correspond to the following functions.

BIT	-1	SCALAR	-6
CHAR	-2	VECTOR	-7
$BIT_@$	-3	MATRIX	-8
$CHAR_@$	-4	CONTENT	-9
INTEGER	-5		

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

ADDENDUM

Due to incomplete implementation in some areas, several HALMAT instructions have not been included in the text. These are as follows:

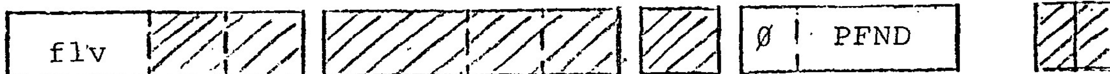
ZDLP: suppressed arrayness specifier (ZDLP = 00F)

ZDLP appears in place of arrayness specifiers if an arrayed argument of a procedure is other than a symbol-table variable. It is used because of current compiler restructions on procedure arguments.

PFST: proc/func start specifier (PFST = 04E)

PFST currently appears prior to the start of the HALMAT construct for procedure or non-shaping function invocations. flv is the nesting level of the invocation. It is not shown in the constructs of Sections 3.5 and 3.6.

PFND: proc/func end specifier (PFND = 04F)



PFND currently appears at the end of the HALMAT construct for procedure or non-shaping function invocations. flv is the nesting level of the invocation. It is not shown in the constructs of Sections 3.5 and 3.6.