B 1700 Systems Micro Implementation Language (MIL)

REFERENCE MANUAL

PRICED ITEM

Pri[illegible] in U S A. May 1977 [illegible]72568

B 1700 Systems Micro Implementation Language (MIL)

REFERENCE MANUAL

Copyright © 1977, Burroughs Corporation, Detroit, Michigan 48232

PRICED ITEM

Burroughs believes that the software described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special, or consequential damages. There are no warranties which extend beyond the program specification.

The Customer should exercise care to assure that use of the software will be in full compliance with laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change. Revisions may be issued from time to time to advise of changes and/or additions.

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P. O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO – West

TABLE OF CONTENTS

Section		Page
	INTRODUCTION	ix
	Background	ix
	Related Documents	ix
1	MICROPROGRAMMING CONCEPTS	1-1
	General	1-1
	Micro-Instructions	1-1
	Defined Field Concepts	1-1
	Interpretation of the Virtual Language	1-2
2	SYNTAX DIAGRAMS	2-1
	Forward Arrows	2-1
	End of Statement	2-1
	Continuation	2-2
	Keywords	2-2
	Variables	2-2
3	BASIC COMPONENTS OF MIL	3-1
	General	3-1
	Identifiers	3-2
	Labels	3-3
	Card Terminators	3-4
	Numbers	3-5
	Bit Strings	3-5
	Character Strings	3-7
	Literals	3-8
	Arithmetic Expressions	3-9
4	STRUCTURE OF A MIL PROGRAM	4-1
5	SEGMENTATION	5-1
	Introduction	5-1
	Label Addresses	5-1
	Segment Statement	5-2
	Code.Segment Statement	5-3
	Compiler - Generated Code	5-5
	Main Code Block	5-5
	External Code Block	5-6
6	DECLARATIONS	6-1
	Data Types	6-1
	Declare Statement	6-1
	Non-Structured Declarations	6-2
	Structured Declarations	6-4
	Declare Examples	6-6
	Introduction	6-6
	Non-Remap Items	6-6
	Remap Items	6-8
	General	6-8
	Reverse	6-10

TABLE OF CONTENTS (Cont)

Section		Page
7	REGISTERS AND SCRATCHPAD	7-1
	General	7-1
	Register Groups	7-1
	Alphabetical Listing of Registers and Key Concepts	7-2
	Active Registers	7-6
	X and Y Registers	7-6
	Field (F) Register	7-6
	Local (L) Register	7-6
	Transform (T) Register	7-6
	Micro-Instruction (M) Register	7-6
	Base (BR) and Limit (LR) Registers	7-6
	Address (A) Register	7-7
	A Stack (TAS)	7-7
	Top of Control Memory (TOPM) Register	7-7
	Memory Base (MBR) Register	7-7
	Control (C) Register	7-7
	Combinatorial Logic or Functional Box	7-8
	Result Registers	7-8
	XORY Result Register	7-8
	XANY Result Register	7-8
	XEOY Result Register	7-8
	CMPX Result Register	7-8
	CMPY Result Register	7-8
	MSKX Result Register	7-8
	MSKY Result Register	7-8
	SUM Result Register	7-9
	Difference (DIFF) Result Register	7-9
	Scratchpad	7-9
	Scratchpad Words - 24 Bits Each	7-9
	Double Scratchpad Words - 48 Bits Each	7-9
	Constant Registers	7-10
	Maximum Main Memory (MAXS) Register	7-10
	Maximum Control Memory (MAXM) Register	7-10
	NULL Register	7-10
	Input/Output Registers	7-10
	Console.Switches	7-10
	Console Cassette Tape Input (U) Register	7-10
	Command (CMND) Register	7-10
	Data Register	7-10
	Condition Registers	7-11
	Introduction	7-11
	Binary Conditions (BICN) Register	7-11
	XY Conditions (XYCN) Register	7-11
	XY States (XYST) Register	7-12
	Any.Interrupt Bit	7-12
	Console Interrupt (CC(3))	7-13
	Main Memory Read Parity Error Interrupt (CD(0))	7-13
	Main Memory Address Out-of-Bounds Override (CD(1))	7-13

TABLE OF CONTENTS (Cont)

Section		Page
7	REGISTERS AND SCRATCHPAD (Cont)	
	Read Address Out-of-Bounds Interrupt (CD(2))	7-13
	Write/Swap Address Out-of-Bounds Interrupt (CD(3))	7-13
	Field Length Conditions (FLCN) Register	7-13
	Interrupt Conditions (INCN) Register	7-13
	Register Designations and Areas of Application	7-14
	Organization of Fields and Subfields	7-14
8	MIL STATEMENTS	8-1
	Index to Statements	8-1
	ADD Scratchpad	8-2
	ADJUST	8-3
	AND	8-4
	ASSIGN	8-6
	BEGIN	8-7
	BIAS	8-9
	BRANCH.EXTERNAL	8-11
	CALL	8-12
	CALL.EXTERNAL	8-13
	CARRY	8-14
	CASSETTE	8-15
	CLEAR	8-16
	CODE.SEGMENT	8-17
	COMPLEMENT	8-18
	COUNT	8-20
	DEC	8-22
	DEFINE	8-23
	DEFINE.VALUE	8-24
	DISPATCH	8-25
	ELSE	8-27
	EMIT.RETURN.TO.EXTERNAL	8-29
	END	8-30
	EOR	8-31
	EXIT	8-33
	EXTRACT	8-34
	FA.POINTS	8-36
	FINI	8-37
	GO TO	8-38
	HALT	8-39
	IF	8-40
	INC	8-46
	JUMP	8-47
	LIT	8-48
	LOAD	8-49
	LOAD.MSMA	8-50
	LOAD.SMEM	8-52
	LOCAL.DEFINES	8-53
	MACRO DECLARATION	8-55

TABLE OF CONTENTS (Cont)

Section		Page
8	MIL STATEMENTS (Cont)	
	MACRO REFERENCE	8-57
	MAKE.SEGMENT.TABLE.ENTRY	8-59
	MICRO	8-60
	M.MEMORY.BOUNDARY	8-61
	MONITOR	8-62
	MOVE	8-63
	NOP	8-65
	NORMALIZE	8-66
	OR	8-67
	OVERLAY	8-69
	PAGE	8-70
	POINT	8-71
	PROGRAM.LEVEL	8-72
	READ	8-73
	REDUNDANT.CODE	8-75
	RESERVE.SPACE	8-76
	RESET	8-77
	ROTATE	8-78
	SEGMENT	8-79
	SET	8-80
	SHIFT/ROTATE T	8-82
	SHIFT/ROTATE X/Y/YX	8-84
	SKIP	8-85
	S.MEMORY.LOAD	8-87
	STORE	8-88
	SUB.TITLE	8-89
	SUBTRACT SCRATCHPAD	8-90
	SWAP	8-91
	TABLE	8-92
	TITLE	8-93
	TRANSFER.CONTROL	8-94
	WRITE	8-95
	WRITE.STRING	8-97
	XCH	8-99
9	PROGRAMMING TECHNIQUES	9-1
	Virtual-Langauge Definitions	9-1
	Source Image Format	9-1
	Program Example	9-1
Appendix A	MIL COMPILER OPERATION	A-1
	Control Cards	A-1
	General	A-1
	Dollar Cards	A-2
	Ampersand Cards	A-5
	MIL Compiler Files	A-6

TABLE OF CONTENTS (Cont)

Section		Page
Appendix B	HARDWARE INSTRUCTION FORMATS AND TABLES	B-1
	B 1700 Hardware Tables	B-1
	B 1700 Hardware Instruction Formats	B-5
	Bias	B-5
	Bit Test Branch False	B-5
	Bit Test Branch True	B-6
	Branch	B-6
	Call	B-7
	Cassette Control	B-7
	Clear Registers	B-8
	Count FA/FL	B-8
	Dispatch	B-9
	Extract From T	B-10
	Four-Bit Manipulate	B-10
	Halt	B-11
	Load F From Doublepad Word	B-11
	Monitor	B-12
	Move 8-Bit Literal	B-12
	Move 24-Bit Literal	B-13
	No Operation	B-13
	Normalize X	B-13
	Overlay Control Memory	B-14
	Read/Write Memory	B-14
	Read/Write MSM	B-15
	Register Move	B-16
	Scratchpad Move	B-17
	Scratchpad Relate FA	B-17
	Set CYF	B-18
	Shift/Rotate T Left	B-18
	Shift/Rotate XY Left/Right	B-19
	Shift/Rotate X/Y Left/Right	B-20
	Skip When	B-20
	Store F Into Doublepad Word	B-21
	Swap F with Doublepad Word	B-22
	Swap Memory	B-22
	Transfer Control	B-23
	Micro-Instruction Timing	B-24
	B 1710 Notes	B-25
	B 1720 Notes	B-25
Appendix C	RESERVED WORDS AND SYMBOLS	C-1
Index		I-1

LIST OF TABLES

Table		Page
8-1	AND Truth Table	8-4
8-2	EOR Truth Table	8-31
8-3	OR Truth Table	8-67
8-4	String Definitions	8-97
B-1	Register Addressing	B-1
B-2	Condition Registers	B-2
B-3	Microinstructions	B-3
B-4	Variant Field Definitions	B-4
B-5	Micro-Instruction Timing	B-24

INTRODUCTION

BACKGROUND

The Burroughs Micro Implementation Language (MIL) is a symbolic coding technique that makes available all the capabilities of the B 1700 Processor. The MIL compiler's machine language output is ready for execution directly upon the hardware. The user, however, must be prepared to programmatically control the total environment including bootstrap loading, interrupt servicing, and potential machine malfunctioning (e.g., parity error detection).

To use MIL properly and efficiently, the programmer must have an extensive knowledge of the available registers and their capabilities. This manual describes the registers, the syntax and the semantics of the MIL language and may be used to write programs without prior knowledge of the system.

RELATED DOCUMENTS

A description of the Input/Output subsystem and the I/O descriptors as well as more detailed information about the registers can be found in the B 1700 Systems Reference Manual (form 1057155).

1. MICROPROGRAMMING CONCEPTS

GENERAL

Microprogramming is a method for programming a computer hardware architecture. The microprogrammer is concerned with machine registers which were formerly the domain of the hardware systems designer. Strings of micro-instructions manipulate those internal registers to present an outward appearance of system hardware which is more functional for problem-oriented programming. In most machines now in the market place, read-only memories (ROM's) contain microprograms which convert the unique internal environment of several different processors into a standard assembly language. Once created, the microprograms are unalterable and may contain compromises in efficiency because of a limited hardware instruction set.

The Burroughs B 1700 system makes use of the latest technology to implement a writable control memory and has several microprograms, each optimized for the functions it will perform. The virtual system architectures chosen have been those of the standard (such as COBOL and FORTRAN), problem-oriented, compiler languages. Other microprogrammers may choose architectures and create languages optimized for other purposes.

MICRO-INSTRUCTIONS

A micro-instruction is the smallest programmable operation within the system. Each micro-instruction is fetched from memory and decoded in the (micro) register to be directly executed by the hardware.

DEFINED FIELD CONCEPTS

A defined field concept allows bit-level data addressing with lengths from 1 to 65,535 bits. There are no visible boundaries or "best" container size for any information contained in main memory. Virtual machine instruction strings (the B 1700 analog of machine object code) and their data may thus be densely packed into meaningful fields, saving considerable memory space. The programming problem of packing and unpacking data fields across hardware container boundaries is completely resolved, saving much programming effort and processor time. The microprogram fetches groups of bits in meaningful field sizes from anywhere in main memory as needed.

Special hardware, called a Field Isolation Unit, has been implemented to achieve bit addressability and variable length fields and to automatically increment addresses. This allows maximum flexibility in defining data fields and resolves the problem of packing and unpacking data fields across hardware container boundaries.

INTERPRETATION OF THE VIRTUAL LANGUAGE

The traditional approach to supporting a higher-level language is to translate the source statements as written by the programmer into another language either directly recognized by the hardware, (e.g., machine object code) or easily translatable into the machine object code (e.g., an assembly language). An alternate technique is the interpretive execution for each source statement with a logically equivalent routine in some lower-level language. A microprogrammed system offers the opportunity to combine the best of both methods. The source statements in the higher-level language are translated into a virtual system code by a compilation process. This system code, also called S-code or S-language, very closely resembles the original source language. Micro-instruction routines then interpretively execute each virtual language statement. The results are:

a. Faster compilation,

b. System architecture, as expressed in the set of microroutines, which is optimized to the source language,

c. Reduction in the processor time required to perform the logical equivalent of each source statement,

d. Reduction in the memory space required to encode each source language operation.

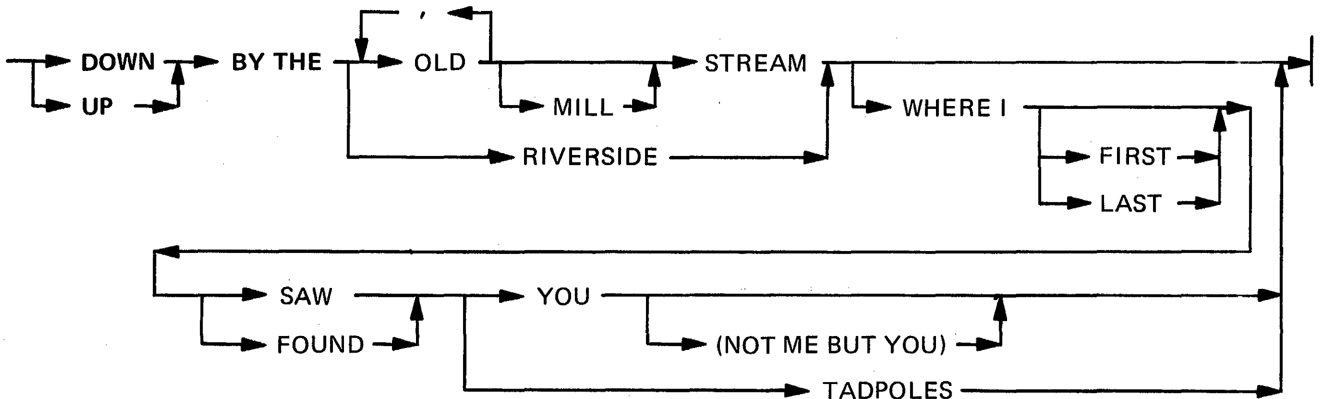
A set of microprogrammed routines is called an interpreter and effectively creates a virtual system architecture for the source language being executed. That is, when the COBOL interpreter is executing, the system is effectively a COBOL machine. When the FORTRAN interpreter is executing, the system is a FORTRAN machine, and so on for any other S-language defined.

2. SYNTAX DIAGRAMS

The principal means of describing MIL syntax is through the syntax diagram, commonly known as "rail-road" notation. The basic conventions are discussed below.

FORWARD ARROWS

Any path traced along the directional flow of the arrows will produce a syntactically valid command. The following example illustrates the technique:



Valid syntax generated from this diagram could be:

DOWN BY THE OLD MILL STREAM
UP BY THE OLD, OLD STREAM
DOWN BY THE RIVERSIDE WHERE I FOUND TADPOLES
DOWN BY THE OLD STREAM WHERE I FIRST SAW YOU (NOT ME BUT YOU)
UP BY THE RIVERSIDE WHERE I LAST FOUND YOU

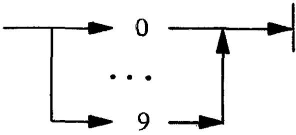
The bridge over OLD, unless otherwise specified, can be crossed any number of times.

END OF STATEMENT

The completion of a statement is indicated by the following convention:

CONTINUATION

The following convention indicates that any number from 0 through 9 is syntactically valid:

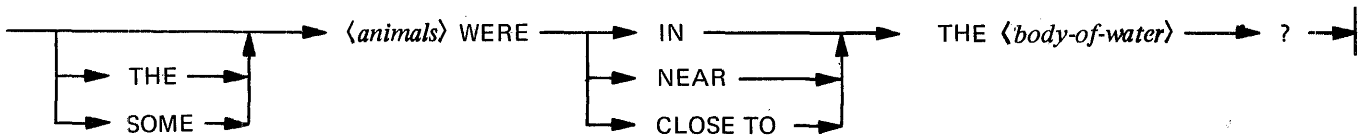


KEYWORDS

Upper-case letters indicate keywords which must literally appear in MIL statements.

VARIABLES

Lower-case letters, words, and phrases within angle brackets indicate syntactic variables which require information to be supplied by the programmer. The following example illustrates the technique:



Valid syntax generated from this diagram might be:

THE TADPOLES WERE IN THE STREAM?
COWS WERE CLOSE TO THE POND?
SOME BIRDS WERE NEAR THE OCEAN?

3. BASIC COMPONENTS OF MIL

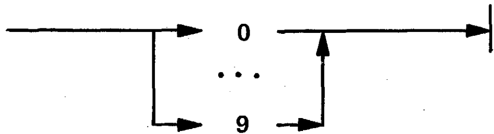
GENERAL

To understand MIL grammar the user should be familiar with the following basic elements of the MIL language.

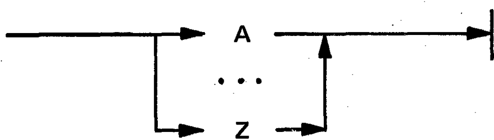
⟨*point*⟩:

⟨*underscore*⟩:

⟨*digit*⟩:



⟨*letter*⟩:



⟨special.characters⟩:

.
& ampersand
@
;
,
#
¬
$
:
>
<
_
=
(
)
*
–
/
%
"
blank (one non-visible character)

IDENTIFIERS

⟨identifier⟩:

⟨letter⟩
⟨digit⟩
.

RESTRICTIONS

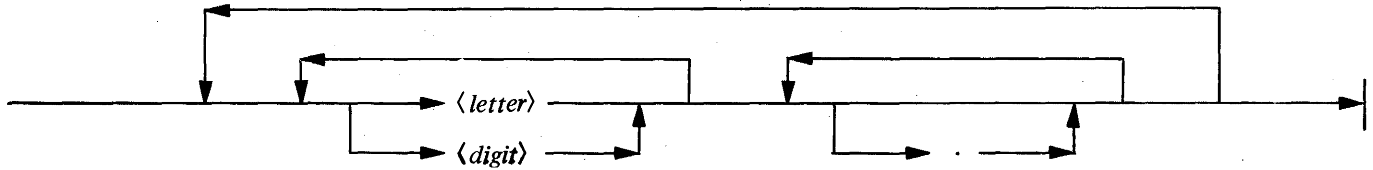
1. An ⟨*identifier*⟩ must begin with a ⟨*letter*⟩.
2. An ⟨*identifier*⟩ may not contain blanks.
3. Reserved words may not be used as ⟨*identifiers*⟩. (See Appendix C: Reserved Words and Symbols.)
4. An ⟨*identifier*⟩ is limited to a maximum of 63 characters: only the first 25 characters are used in uniqueness detection.

Examples

TEST.NAME.1   T.123.Q   ABC   LOOP . . 12

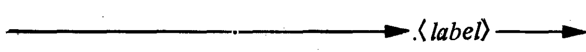
LABELS

⟨*label*⟩:

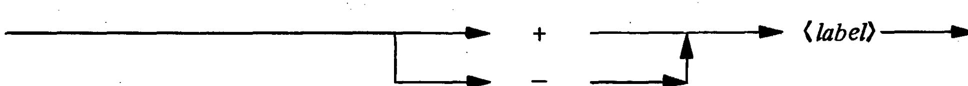


⟨*unique.label*⟩:

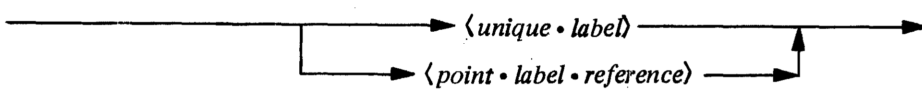
⟨*point.label.declaration*⟩:



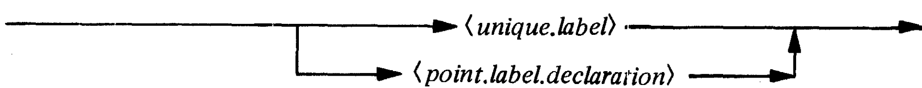
⟨*point.label.reference*⟩:



⟨*label.reference*⟩:



⟨*label.declaration*⟩:



⟨*Labels*⟩s may be declared by: (1) starting the ⟨*label*⟩ anywhere in columns 1 through 5 of a source image, or (2) starting the ⟨*label*⟩ immediately after the reserved words TABLE, SEGMENT, or CODE.SEGMENT. (See also Segmentation: Label addresses.)

RESTRICTIONS

1. A ⟨*label*⟩ must begin with a ⟨*letter*⟩ or a ⟨*digit*⟩
2. A ⟨*label*⟩ may not contain blanks.
3. A ⟨*label*⟩ is limited to a maximum of 63 characters: only the first 25 characters are used in uniqueness detection.
4. ⟨*Unique Label*⟩s may be declared only once.
5. ⟨*Point Label*⟩s may or may not be unique.

Examples

.A.POINT.LABEL   REGULAR.LABEL   LOOP   BEGINNING.OF.TEST.1

CARD TERMINATORS

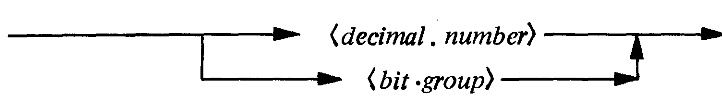
⟨*card.terminator*⟩:

RESTRICTION

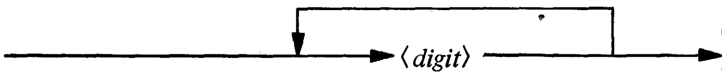
A percent sign (%) is treated as any other string character if it is contained within a ⟨*character·string*⟩. However, in all other cases, a % will cause the scanning of the current source image to terminate.

NUMBERS

⟨*number*⟩:

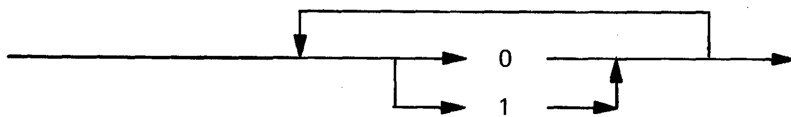


⟨*decimal·number*⟩:

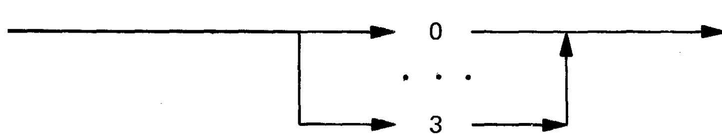


BIT STRINGS

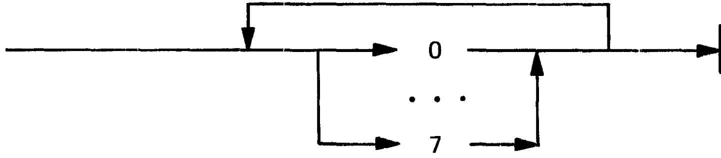
⟨*binary·string*⟩:



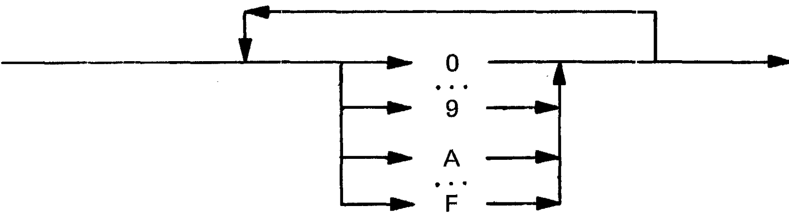
⟨*quartal·string*⟩:



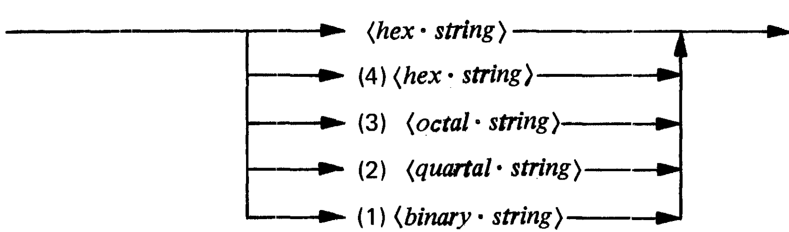
⟨octal·string⟩:



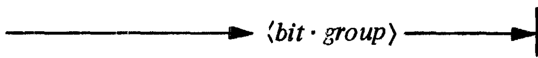
⟨hex·string⟩:



⟨bit·group⟩:



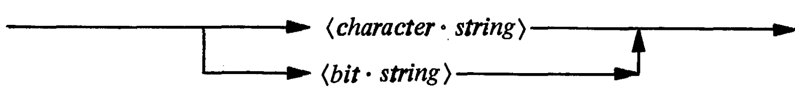
⟨bit·string⟩:



RESTRICTION

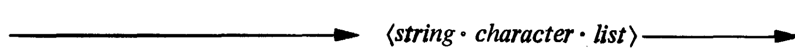
If no bit mode is specified (i.e., the indicator digit in parentheses is omitted), then "hex" is assumed.

⟨*string*⟩:

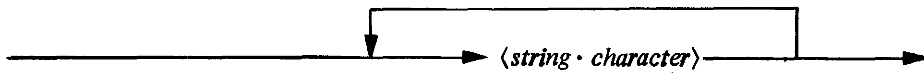


CHARACTER STRINGS

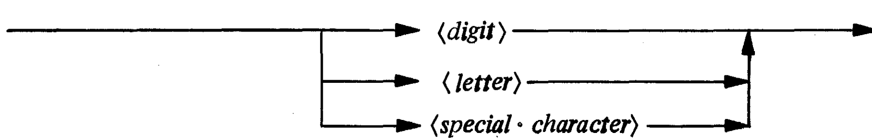
⟨*character.string*⟩:



⟨*string.character.list*⟩:



⟨*string.character*⟩:



Examples

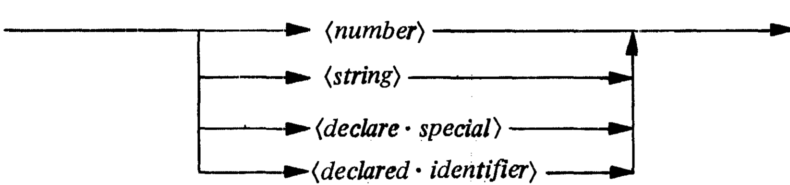
"** THIS IS AN EXAMPLE OF A CHARACTER STRING"
"   ROW THE BOAT GENTLY . . ."

RESTRICTION

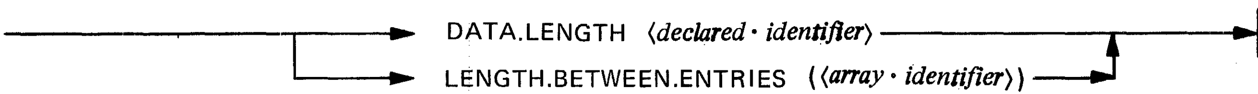
The quotation mark (") cannot be specified as a ⟨*string character*⟩. As an alternative, the programmer can specify a ⟨*hex string*⟩ instead of a ⟨*character.string*⟩.

LITERALS

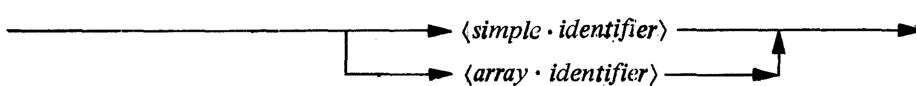
⟨*literal*⟩:



⟨*declare·special*⟩:



⟨*declared·identifier*⟩:



⟨*array·identifier*⟩:

⟨*array·index*⟩:

DATA LENGTH (⟨*declared·identifier*⟩) will supply the specified or computed length in bits of the indicated ⟨*declared·identifier*⟩. For an ⟨ *array·identifier*⟩, the length will be the length of one of the items in the array, not the length of the entire array.

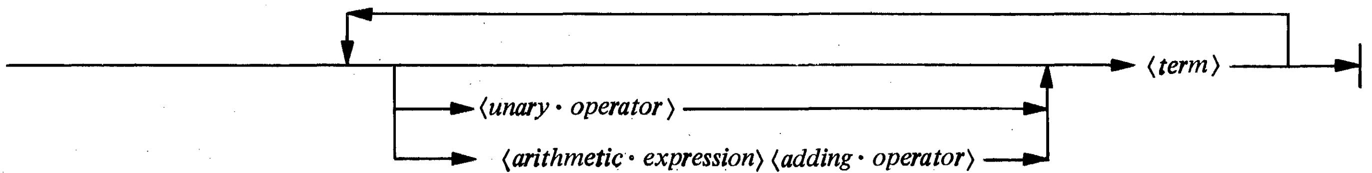
LENGTH·BETWEEN·ENTRIES (⟨*array·identifier*⟩) will supply the bit difference between the beginning of one item in the specified array and the next item in the array. Note that in the case of structured arrays (See Structured Declarations) this will not always be the same as DATA·LENGTH ⟨*array·identifier*⟩.

Examples:

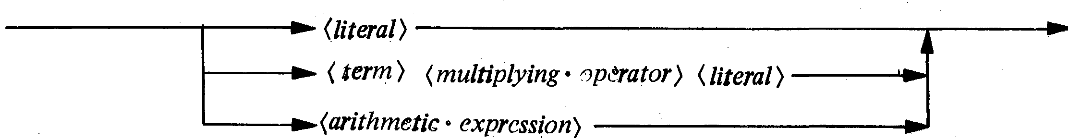
1587
"STRING"
DATA·LENGTH (AN·ITEM)
ARRAY·ELEMENT (7)

ARITHMETIC EXPRESSIONS

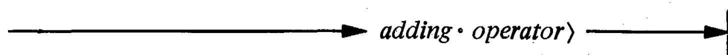
⟨*arithmetic·expression*⟩:



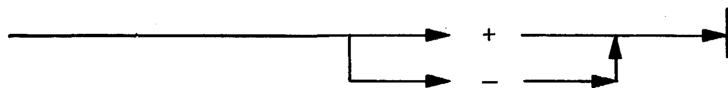
⟨*term*⟩:



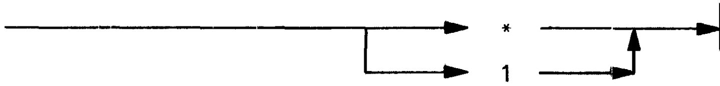
⟨*unary·operator*⟩:



⟨*adding·operator*⟩:



⟨ *multiplying·operator* ⟩:



⟨ *Arithmetic expression* ⟩s yield numerical values by combining ⟨ *literal* ⟩s in accordance with specified operations. The operators +, -, *, and / have the conventional mathematical meanings of addition, subtraction, multiplication, and division, respectively.

The sequence in which operations are performed is determined by the precedence of the operators involved. The order of precedence is:

First: * /
Second: + -

When operators have the same order of precedence, the sequence of operation is determined by the order of their appearance, from left to right. Parentheses can be used in normal mathematical fashion to override the usual order of precedence.

Parenthesized expressions are treated as ⟨ *term* ⟩s, i.e., they are evaluated by themselves and the resulting value is subsequently combined with the other elements of the ⟨ *arithmetic·expression* ⟩. Thus the normal precedence of operators may be overridden by careful placement of parentheses.

4. STRUCTURE OF A MIL PROGRAM

There are two parts or sections to a MIL program: the declarations and the body. The declarations should contain:

a. A comment description of the function of the MIL program.

b. Any global data structures (DECLARES). Note that "global" refers to use throughout the program; local refers to use restricted to a part of the program.

c. Any global DEFINEs.

d. Any MACRO definitions.

The body follows the declarations and will contain all code-producing statements. The statements should be logically grouped in labeled BEGIN . . . END blocks. Each BEGIN . . . END block may contain its own local data structures, LOCAL.DEFINES or labels. The last statement of the body should be FINI.

The following is a basic outline of a MIL program using the above general rules. For specific details on assembly coding forms and program examples refer to: Programming Techniques.

```
Declarations                                % descriptive comment
                                            DECLAREs
                                            DEFINEs
                                            MACROs

                        LABEL.A
                                            BEGIN A
                                               (code for A)
Body                                        END A
                                            BEGIN B
                                               (code for B)
                                            END B
                                            FINI
```

5. SEGMENTATION

INTRODUCTION

Segmentation in MIL is a multi-faceted and somewhat complicated subject. Because MIL is the language of the B 1700's, and because it is used for many different purposes (Diagnostics, Emulators, Interpreters, I/O Drivers, MCP Kernels, etc.), it must attempt to satisfy the needs of a wide range of users. Segmentation plays a particularly important role on the B 1700 because of the READ/WRITE access capability of the hierarchical memory structure (M-Memory, S-Memory, Disk).

LABEL ADDRESSES

To begin the discussion on segmentation, we must first identify the label types pertaining to address assignment. They are: ⟨*regular.label*⟩ and ⟨*physical.label*⟩. (These should not be confused with the two types of label representation: ⟨*unique.label*⟩ and ⟨*point.label*⟩. See Basic Components Of MIL: Labels.) The types are based on how the labels are declared which in turn determines how the address of the label is to be assigned.

A ⟨*label*⟩ which is declared by starting it in column 1-5 of a source image is always a ⟨*regular.label*⟩.

A ⟨*label*⟩ which is declared by starting it immediately after the reserved words TABLE, SEGMENT, or CODE.SEGMENT is always a ⟨*physical.label*⟩.

A ⟨*regular.label*⟩ is always given the current ⟨*segment.code.address*⟩ when the ⟨*label*⟩ is declared.

A ⟨*physical.label*⟩ is always given the current ⟨*physical.code.address*⟩ when the ⟨*label*⟩ is declared.

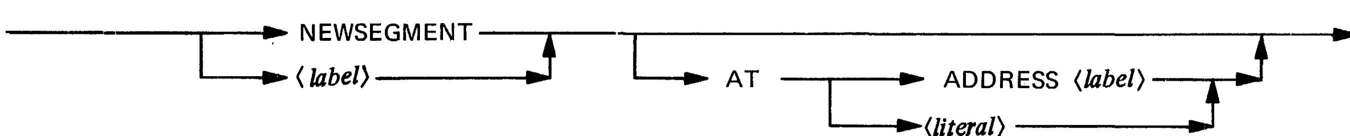
The ⟨*segment.code.address*⟩ is updated by 16 as each micro-instruction is generated and can be changed to a new value by the appearance of a SEGMENT or CODE.SEGMENT statement.

The ⟨*physical.code.address*⟩ is also updated by 16 as each micro-instruction is generated and can be changed to a new value by the appearance of an ADJUST LOCATION statement. (See MIL Statements: ADJUST).

Both the ⟨*physical.code.address*⟩ and the ⟨*segment.code.address*⟩ are initialized to 0 (zero) when a compilation begins.

SEGMENT STATEMENT

Syntax



NOTE

The ⟨*literal*⟩ must be MOD 16, meaning the last four bits must be @ (1) 0000 @.

Semantics

Through the use of the SEGMENT statement, the user has the means to divide his/her MIL program into several parts such as a single ⟨*primary.code.block*⟩ and one or more ⟨*segment.block*⟩ (s). The ⟨*primary. code.block*⟩ should provide one or more areas suitable for containing the individual ⟨*segment.block*⟩ (s). These areas are designated by declaring one or more ⟨*regular.label*⟩(s) somewhere within the ⟨*primary.code. block*⟩. Quite often there will be only one designated area for ⟨*segment. block*⟩(s), and it will begin at the end of the ⟨*primary.code.block*⟩.

The purpose of the SEGMENT statement is to inform the compiler exactly where the ⟨*segment.block*⟩ will be (relative to the ⟨*primary.code.block*⟩) when its code is executed. In this way the compiler can generate the correct branch/call displacements whenever a statement in the ⟨*primary.code.block*⟩ branches to or calls a routine in one of the ⟨*segment.block*⟩(s). In the same way, a statement in one of the ⟨*segment. block*⟩(s) may branch to or call a routine in either the ⟨*primary.code.block*⟩ or in any of the ⟨*segment. block*⟩(s). (See MIL Statements: EMIT.RETURN.TO.EXTERNAL, CALL.EXTERNAL, BRANCH. EXTERNAL.)

All code is assumed to be in the ⟨ primary.code.block⟩ until the first SEGMENT statement is encountered. From this point one, all code is assumed to be in that segment until the next SEGMENT statement is encountered, and so on.

The SEGMENT statement may also be used to specify logical breaks within a continuous stream of code. In this case, only the name of the segment needs to be specified since the code addresses are to continue linearly. The entire program and all of the ⟨*segment.block*⟩(s) are given entries in the segment dictionaries as part of the parameter blocks associated with a MIL code file. From these dictionary entries and from the segment name-to-number correspondence table the addresses and lengths of each segment are available and can be used to do sophisticated static binding prior to execution of the code. (See MIL Statements: MAKE.SEGMENT, TABLE.ENTRY).

CODE.SEGMENT STATEMENT

Syntax

```
CODE.SEGMENT ———————► <label> ———————►|
```

Semantics

Another form of segmentation in MIL is used when a microprogram is running with the MCP, or under MCP control. All of the interpreters as well as GISMO are examples of this situation. With this mechanism, a microprogrammer is able to specify which portions of the program are to reside on disk until they are actually needed for execution. This provides the programmer with the same facility normally only found in higher level languages.

In order to use this facility, the programmer must follow certain rules and remember some restrictions. First, some definitions:

⟨*main.code.block*⟩: all code generated until the first CODE.SEGMENT statement is encountered; this may encompass the ⟨*primary.code.block*⟩ and one or more ⟨*segment.block*⟩(s).

⟨*external.code.block*⟩: all code generated between a given CODE.SEGMENT statement and the next CODE.SEGMENT statement, or the end of the program, whichever comes first.

⟨*main.code.base*⟩: the M-Memory bit address of the first micro-instruction in the ⟨*main.code.block*⟩. If no part of the ⟨*main.code.block*⟩ resides in M-Memory, then the ⟨*main.code.base*⟩ should be 0.

If the processor is an S-Memory processor, then the ⟨*main.code.base*⟩ should be the memory address of the first micro-instruction in the program. (See MIL Statements: MAIN.CODE.BASE.)

⟨*mbr.topm*⟩: a 24-bit bucket containing the MBR value for the ⟨*main.code.block*⟩. In addition, since the MBR value is always a MOD 16 number, the low order 4 bits of ⟨*mbr.topm*⟩ should be the TOPM value of the ⟨*main.code.block*⟩.

The microprogrammer must provide the following items in a program:

a. A define for MAIN.CODE.BASE to indicate the Scratchpad word containing ⟨*main.code.base*⟩.

Example:

DEFINE MAIN.CODE.BASE = S14B#

b. A define for MBR.TOPM to indicate the Scratchpad word containing ⟨*mbr.topm*⟩.

Example:

DEFINE MBR.TOPM = S15A

NOTE

The above defines must be included in the ⟨*main.code. block*⟩ and must not be defined within some LOCAL. DEFINE scope. In addition, the two Scratchpad locations must be initialized by the interpreter when it is given control from GISMO.

c. A routine labeled GO.TO.EXTERNAL.SEGMENT to interrogate the interpreter dictionary and generate a communicate (if necessary) to guarantee that the requested ⟨*external.code.segment*⟩ is present in S-Memory. In addition, it must perform the initial transfer to the ⟨*external.code. segment*⟩.

Example:

```
GO.TO.EXTERNAL.SEGMENT
        % T CONTAINS SEGMENT NUMBER
        % L CONTAINS BIT DISPLACEMENT WITHIN SEGMENT
    SHIFT T LEFT BY 6 BITS TO X          % T * 64
    SHIFT T LEFT BY 4 BITS TO Y          % T * 16
    MOVE SUM TO FA                       % T * 80
    ADD ADDR.INTERP.SEG.DICT TO FA
    READ 2 BITS TO X
    IF LSUX THEN                         % THE SEGMENT IS PRESENT
        BEGIN PRESENT
            COUNT FA UP BY 32
            READ 24 BITS TO X            % SEGMENT BASE ADDRESS
            IF SUBSET THEN INCLUDE       % FOR S-MEMORY PROCESSORS
          BEGIN
              MOVE L TO Y
              MOVE SUM TO A
          END ELSE
          BEGIN
              MOVE 0 TO TAS              % NECESSARY FOR
                                         % M-MEMORY SYSTEM
              MOVE L TO T                % NEW A AND TOPM VALUE
              MOVE X TO L                % NEW MBR VALUE
              TRANSFER.CONTROL
          END
        END PRESENT
    MOVE T TO L
    MOVE 58 TO T                         % COMMUNICATE NO.FOR
                                         % NON PRESENT SEGMENT
    SHIFT T LEFT BY 16 BITS
    SET L(0)                             % ONE LEVEL SEG DICT.
    GO TO GIVE.UP.CONTROL.               % SAVE STATE AND XFER TO
                                         % MCP VIA GISMO
```

NOTES

a. The initial "T" and "L" values are supplied by the compiler prior to entering the above routine.

b. Other registers may be destroyed depending on how the routine is written.

c. The routine must push a 0 (zero) onto the A stack for the M-Memory Processor. This is necessary so that an exit within an <*external.code.block*> can be trapped into a routine that will transfer control back to the <*main.code.block*>. This also implies that parameters may not be passed via the A stack when initially transferring to an <*external.code.block*>.

The compiler will provide all other routines necessary to effect the transfer to and from <*external.code.block*>(s).

The only kind of transfers allowed are calls and branches from the <*main.code.block*> to an <*external.code.block*> and from an <*external.code.block*> to the <*main.code.block*>. Transfers between <*external.code.block*>(s) are not allowed. In addition, such calls and branches must be syntactically separated from calls and branches with the same <*code.block*>. Instead of CALL, the command CALL EXTERNAL must be used. Instead of GO TO, the command BRANCH.EXTERNAL must be used. (See MIL statements: EMIT. RETURN.TO.EXTERNAL, CALL.EXTERNAL and BRANCH.EXTERNAL.)

Compiler - Generated Code

Following is the code the compiler generates when CODE.SEGMENTs are used. (All <*label*>s used in the examples are shown for clarity only: the compiler has its own internal representation for the labels.)

MAIN CODE BLOCK

a. For each different <*label*> occurring after a CALL.EXTERNAL or BRANCH.EXTERNAL statement in the <*main.code.block*>, the compiler will divert the call or branch to the following code which is generated at the end of, and part of, the <*main.code.block*>:

```
MOVE ADDRESS <label> TO L
MOVE <label.segment.number> TO T
GO TO GO.TO.EXTERNAL.SEGMENT
```

b. If the program executes on an M-Memory Processor (B 1726), the following code will be emitted in the main.code.block:

```
EXIT.TO.EXTERNAL
    MOVE TAS TO L
    MOVE TAS TO T
    MOVE LF TO TF
    MOVE 0 TO LF
    TRANSFER.CONTROL
```

EXTERNAL CODE BLOCK

a. If the program executes on an M-Memory Processor (B 1726), the following code will be emitted at the beginning of every ⟨ *external.code.block*⟩:

```
         MOVE TAS TO T
LEAVE.EXTERNAL.SEGMENT
         MOVE MBR TOPM TO L
         MOVE LF TO T
         SET LF TO 0
         TRANSFER.CONTROL
```

b. For each different ⟨ *label*⟩ occurring after a CALL.EXTERNAL or BRANCH.EXTERNAL statement in the ⟨ *external.code.block*⟩, the compiler will divert the call or branch to the following code which is generated at the end of, and part of, the ⟨ *external.code.block*⟩:

1. If the program executes on an S-Memory PRocessor (B 1712 - B 1714) the following code is generated:

```
MOVE ADDRESS (⟨ label⟩) TO X
GO TO SUBSET.BRANCH.TO.MAIN
```

2. If the program executes on an M-Memory Processor (B 1726) the following code is generated for each different ⟨ *label*⟩ in a BRANCH.EXTERNAL statement:

```
MOVE ADDRESS (⟨label⟩) TO X
GO TO BRANCH.TO.MAIN
```

3. If the program executes on an M-Memory Processor (B 1726) the following code is generated for each different ⟨ *label*⟩ in a CALL.EXTERNAL statement:

```
MOVE ADDRESS (⟨label⟩) TO X
GO TO CALL.TO.MAIN
```

c. At the end of every ⟨ *external.code.block*⟩ the following code is emitted.

1. For S-Memory Processor (B 1712 - B 1714):

```
SUBSET.BRANCH.TO.MAIN
      MOVE MAIN.CODE.BASE TO Y
      MOVE SUM TO A
```

2. For M-Memory Processors (B 1726):

```
BRANCH.TO.MAIN
      MOVE TAS TO NULL
      MOVE MAIN.CODE.BASE TO Y
      MOVE SUM TO T
      GO TO LEAVE.EXTERNAL.SEGMENT
```

```
CALL.TO.MAIN
      MOVE MAIN.CODE.BASE TO Y
      MOVE SUM TO T
      MOVE MBR TO L
      MOVE TOPM TO LF
      MOVE L TO TAS
      MOVE ADDRESS (EXIT.TO.EXTERNAL) TO X
      MOVE SUM TO TAS
      GO TO LEAVE.EXTERNAL.SEGMENT
```

NOTES

a. When branching from the ⟨ *main.code.block*⟩ to an ⟨ *external.code.block*⟩ the T and L registers are used, plus whatever registers the GO.TO.EXTERNAL.SEGMENT routine uses.

b. When calling or branching to a routine in the ⟨ *main.code.block*⟩, the X and Y registers are used: This means that they cannot also be used for passing parameters. In addition CP should be equal to 24, otherwise the transfer may not take place correctly.

   Also, on an M-Memory Processor, the T and L registers, as well as the A stack are used. Thus, a good rule of thumb is to avoid using X, Y, T, L, and TAS when passing parameters to/from the ⟨ *main.code.block*⟩ and ⟨ *external.code.block*⟩(s).

c. The code for S-Memory Processors is different than the code for M-Memory Processors. Thus, CODE.SEGMENTS cannot be used if the program is to execute interchangeably on either the B 1710 or B 1720 Series processors. (See Appendix A: $ NO EXTERNAL).

6. DECLARATIONS

DATA TYPES

Three main types of data fields may be declared in MIL:

1. BIT
2. CHARACTER
3. FIXED

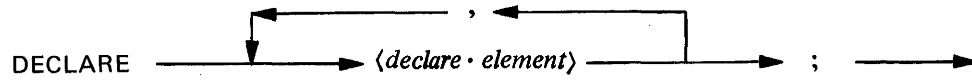
A bit field consists of a number of bits specified by a number in parentheses following the reserved word BIT.

A character field consists of a number of 8-bit characters specified by a number in parenthesis following the reserved word CHARACTER.

A FIXED data field is the same as a BIT (24) field but is allowed in order to keep declare syntax consistent with SDL.

DECLARE STATEMENT

Syntax



Semantics

The DECLARE statement specifies the addresses and characteristics of contents of memory storage areas.

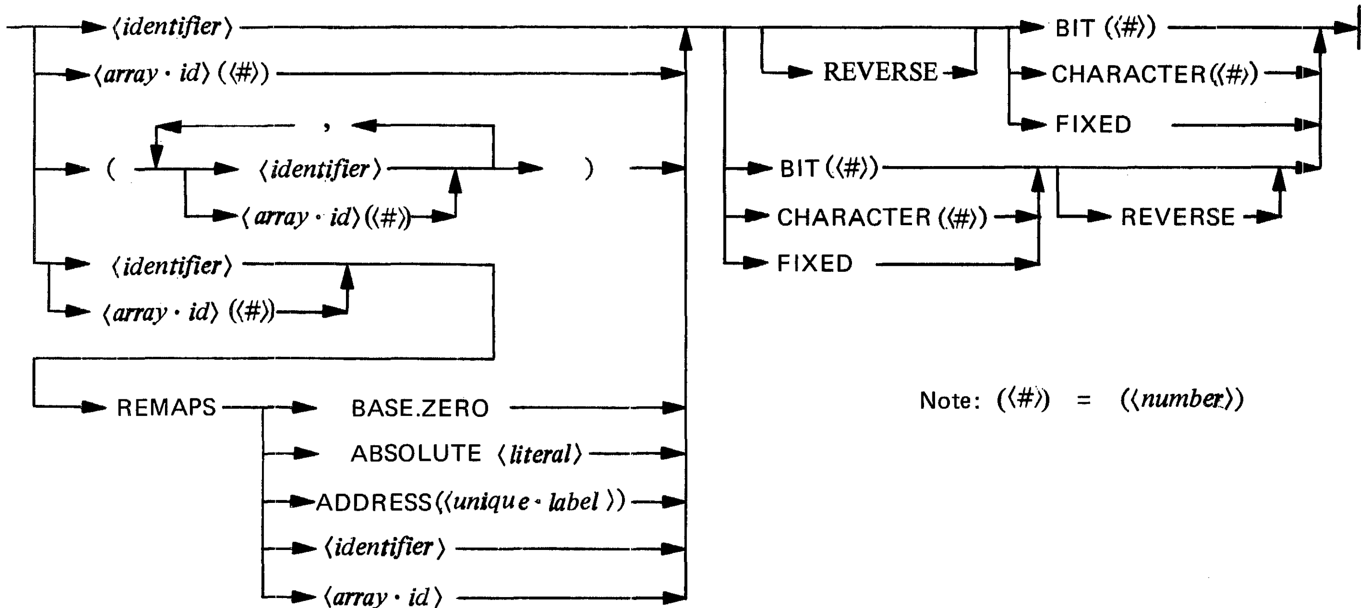
The maximum number of data elements (including fillers, dummys, and implicit fillers) allowed in one structure is 50. Any attempt to declare more will cause a table overflow error to be detected at compile time.

An *<array>* may have a maximum of 65535 elements, each being a maximum of 65535 bits (8191 characters).

The two types of *<declare.elements>* are discussed separately below.

Non-Structured Declarations

⟨*declare.element*⟩:



Data may be declared as simple, having one occurrence, or as subscripted, having as many occurrences as specified by the ⟨*array bound*⟩. In the latter case, array subscripts are considered to range from zero to ⟨*array bound*⟩-1.

BIT, CHARACTER or FIXED specifies the type of data in the field and the field size.

REVERSE specifies that an item or a structure is to be accessed in a reverse manner or in a reverse direction from some base. The easiest way to remember what is happening is to realize that the compiler will simply compute the address of a declared ⟨*identifier*⟩ normally, and then, if reverse is specified, subtract the ⟨*identifier*⟩'s length from the address to get the starting address of the ⟨*identifier*⟩.

As the syntax indicates, different data fields having the same format may be declared collectively inside parenthesis ( ).

The following example illustrates the various options available in this type of ⟨*declaration statement*⟩.

Example

```
DECLARE
     PRIDE                              FIXED
     COVETOUSNESS                       CHARACTER (10),
     GLUTTONY                           BIT (40),
     (LUST, ENVY, ANGER (5))            BIT (10),
     SLOTH (20)                         FIXED,
     WRATH (5)                          CHARACTER (6);
```

where

PRIDE is a 24-bit numeric field;
COVETOUSNESS is a 10-byte character field;
GLUTTONY is a 40-bit field;
LUST and ENVY are each 10-bit fields, as is each of the five elements
comprising ANGER;
SLOTH occurs twenty times, each element being a 24-bit numeric field;
WRATH is a six-byte character field occurring five times.

Data fields may be re-formatted by the use of the REMAPS option. Remapping is subject to the same general rules discussed above. The following example best illustrates its use:

```
B FIXED, C BIT (50),
BB REMAPS B CHARACTER (3),
CC (2) REMAPS C FIXED;
```

Note that CC specifies 48-bits (or 2 elements, 24-bits each). The last two bits will be considered as an implied filler by the compiler. A field may not be remapped larger than its original size.

There is no limit on the number of times a field may be remapped. A field which has remapped another may itself be remapped. The remap option specifies that the ⟨*identifier*⟩ on the left side of the reserved word REMAPS will have the same starting address as the ⟨*identifier*⟩ on the right side.

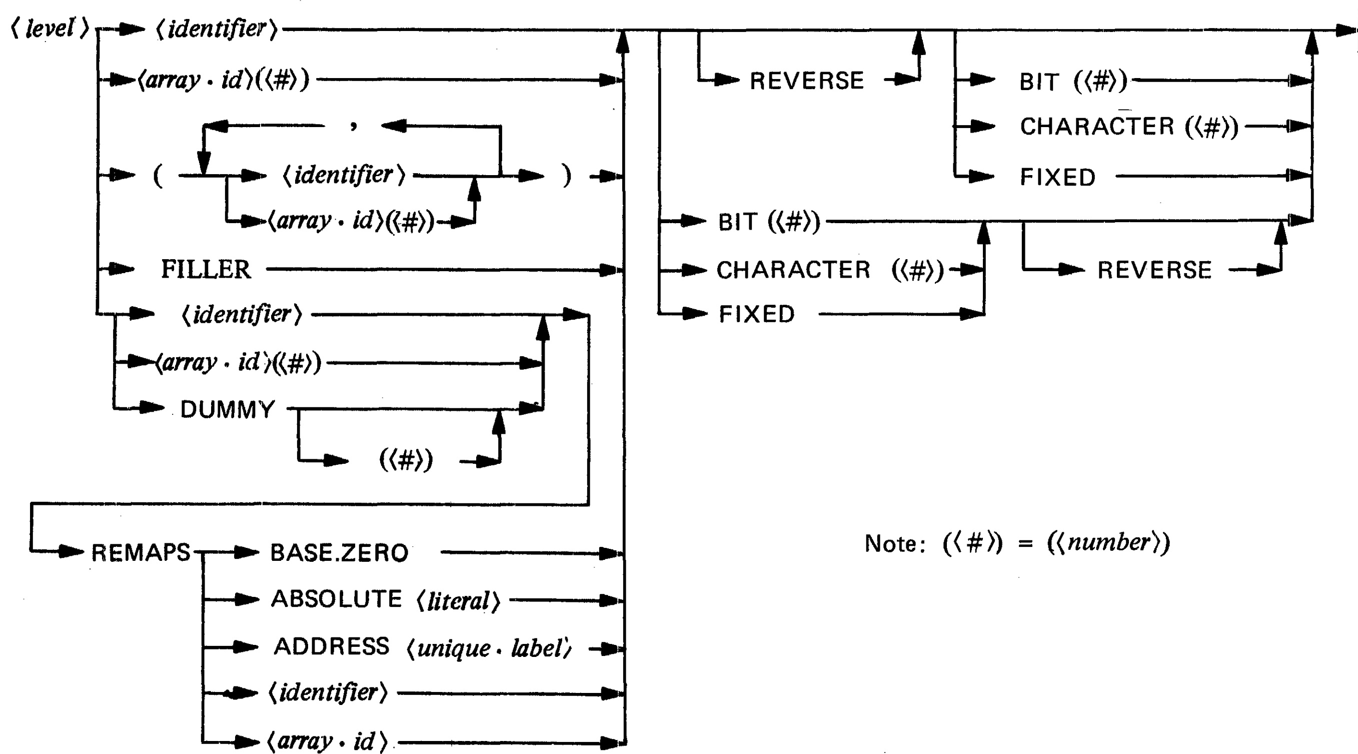
A data field may be remapped to BASE.ZERO which will give the field a relative address of zero. For example:

```
DECLARE Q REMAPS BASE.ZERO BIT (7);
```

This device is used as a free-standing declaration since it does not remap a previously declared data item.

Structured Declarations

⟨declare.element⟩:



MIL allows the structuring of data where a field may be subdivided into a number of sub-fields, each of which has its own ⟨*identifier*⟩. The whole structure is organized in a hierarchical form, where the most general declaration is a ⟨*level*⟩ 01 (or 1). No declaration may be on a ⟨*level*⟩ greater than 99. A subdivided field is called a Group Item, and a field not subdivided is known as an Elementary Item.

The type and length of data need not be specified on the group level. All Elementary Items must indicate type and length; the compiler will assume type bit and add the lengths of the components to determine the length of the Group Item. Note that the length of the Group Item is the sum of the lengths of its Elementary Items.

In the following example, both A.A and C.C are considered Group Items; A.A has a total length of 90 bits, and C.C is 50 bits in length.

Example

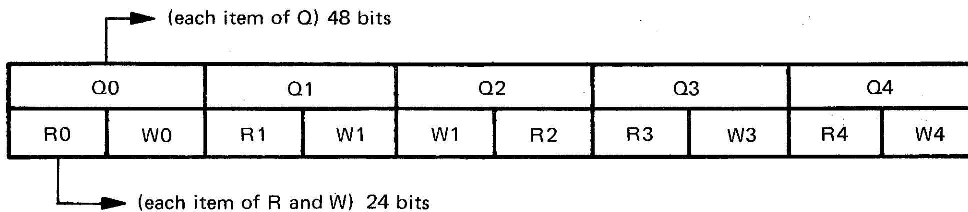
```
DECLARE
    1       A.A,
      2     C.C,
        3   D     BIT (20),
        3   E     BIT (30),
      2     H     CHARACTER (5);
```

Fillers may be used to designate certain Elementary Items which the program does not reference. If the filler is the last item in a structure, it may be omitted; the compiler will consider the Group Item to have an implied filler. A filler may never be used as a Group Item.

If the 01 level group item is an ⟨*array*⟩, it is mapped as a contiguous area in memory. However, subdivisions of this ⟨*array*⟩ are not contiguous as shown in the example structure below:

Example

```
DECLARE                                  DECLARE
     1   Q(5)   BIT(48),         or           1   Q(5)
       2 R      FIXED,                        2   (R, W) FIXED;
       2 W      FIXED;
```



If a Group Item is an ⟨*array*⟩, an ⟨*array specification*⟩ may not appear in any subordinate item; that is, only one-dimensional ⟨*array*⟩s are allowed. An ⟨*array specification*⟩ is implied for all subordinate items.

If a Group Item is declared with the REVERSE option, then REVERSE is also implied for all subordinate items in that group. Specification of the REVERSE option for subordinate items would be redundant.

Structured data may be remapped in the same manner as non-structured data. In addition, structured data may be remapped with a dummy group identifier. The purpose of this construct is to allow the user to remap data items without having to declare another Group Item which describes the same area in memory. Thus in the following example:

Example

```
DECLARE
     1   YAK            BIT(100),
       2 AARDVARK       BIT(20),
       2 SEA.OTTER      BIT(80);
```

YAK might be remapped as:

```
DECLARE                                        DECLARE
     1   AA REMAPS   YAK   BIT(100),               1    DUMMY  REMAPS YAK  BIT(100),
       2 CC                BIT(30),    or            2  CC                 BIT(30),
       2 DD                BIT(70);                  2  DD                 BIT(70);
```

Both YAK and AA refer to the same area in memory: hence AA is redundant.

If a remapped item contains the REVERSE option, then REVERSE is also implied for the remapping item.

The user should note the distinction between DUMMY and FILLER. DUMMY is used in conjunction with REMAPS to eliminate the necessity of declaring a redundant Group Item. FILLER is used if one desires to skip over a part of the structure.

The following restrictions apply to the use of DUMMY REMAPS:

1. DUMMY may only be used with ⟨*remap declarations*⟩.

2. All restrictions applying to REMAPS apply to DUMMY REMAPS.

3. DUMMY must not remap another DUMMY.

4. DUMMY Group Items must have at least one non-filler component.

DECLARE EXAMPLES

Introduction

Let us illustrate by example exactly how declarations might be used in a MIL program, and note the associated relevant points.

The DECLARE statement in MIL is one which allows the user to logically assign names to physical or relative memory address in a structured manner. This facility allows one to construct data structures in a format that is simple to understand and easy to change when the occasion arises.

Non-Remap Items

The MIL compiler maintains a variable which is initialized to 0. When an item is declared, it is assigned the current value of this variable and the variable is incremented by the bit length of the declared item.

Example

```
DECLARE
    DISPATCH.REGISTER          BIT(24),
    GLOP1                      BIT(48),
    ADDR.GISMO                 BIT(24),
    LOCN.MAKE.MCP.BE.HERE      BIT(36),
    GLOP2                      BIT(29),
    ADDR.MCP.LIMIT             FIXED;
```

Note that the DECLARE statement is completely free form, must begin with the world "DECLARE", must end in a ";", and that each element must be separated from its predecessor with a ",".

Each element thus declared is used exactly as a ⟨*literal*⟩ and most often represents a memory address.

Example

```
MOVE ADDR.GISMO TO FA
READ 24 BITS TO X
```

This would assign the literal 72 (= 24+48 = ADDR.GISMO) to register FA and would cause the contents of memory at address 72 to be read into register X.

Should the compiler encounter another DECLARE, it will merely start assigning addresses where it left off previously.

Example

```
DECLARE
    GLOP3             BIT(10)
    CHAR.SAVE.AREA    CHARACTER(8);
```

GLOP3 above would be assigned the value of the aforementioned address-counting variable, in this instance 185.

DECLARE elements may also be structured such that some names overlap pieces of memory described by other names.

Example

```
DECLARE
     1           TRACE.BITS              BIT(27),
        5        FILLER                  BIT(15),
        5        TB.FLAGS                BIT(1),
        5        TB.TYPE                 BIT(4),
           8     FILLER                  BIT(1),
           8     TB.STORES.ONLY          BIT(1),
           8     TB.BRANCHES             BIT(1),
           8     TB.THE.REST             BIT(1),
        5        TB.GET.SPACE.TYPE,
             99  (TB.STORES.ONLY,
                  TB.BRANCHES,
                  TB.REMAINDER)          BIT(1);
```

This example illustrates the following points:

1. The address picks up where the previous DECLARE leaves off. This is not true, however, where the previous item or structure is a "remap item". The compiler's internal variable used for default address assignment is maintained and incremented only for non-remap items or structures.

2. DECLAREs may be structured such that some fields are denoted as being contained within other fields.

3. "FILLER" can be used in structures as often as necessary to increment the address-counting variable past an area of memory which the programmer does not intend to reference by a symbolic name.

4. Items with the same type and length can be put into a list surrounded by parentheses, with the type and length specified only once at the end.

5. The length of an item need not be specified if it has sub-items whose lengths can be determined.

NOTE

Structures must begin with an "01" level identifier. Substructures may then have any level from 02 to 99 inclusive, with the substructure always having higher level numbers than the superstructure.

Remap Items

GENERAL

It is possible to temporarily suspend the mechanism which causes addresses to be assigned based on where the last DECLARE left of by using remaps structures.

For example, if we wish to declare a "template", where the declared addresses are added to some base prior to actual use, we would do the following:

```
DECLARE
    1     SYSTEM.DESCRIPTOR REMAPS BASE.ZERO,
      2   SY.MEDIA               BIT(2),
      2   SY.LOCK                BIT(1),
      2   (SY.IN.PROCESS,
           SY.INITIAL,
           SY.FILE)              BIT(1),
      2   FILLER                 BIT(10),
      2   SY.TYPE                BIT(4),
      2   SY.ADDRESS             BIT(36),
        3 FILLER                 BIT(12), % PORT AND CHANNEL
        3 SY.CORE                BIT(24),
      2   SY.LENGTH              BIT(24);
```

One might use the above structure as follows:

```
DEFINE SYS.DESC.BASE = S14A#
%
MOVE SY.TYPE TO FA
ADD SYS.DESC.BASE TO FA
READ DATA.LENGTH (SY.TYPE) BITS TO X
```

Note the use of a new reserved word, "DATA.LENGTH". This construct allows one to use the length of a declared item without having to define it elsewhere.

The remap structures that are permitted are:

1. REMAPS BASE.ZERO
2. REMAPS ABSOLUTE ⟨*literal*⟩
3. REMAPS ADDRESS (⟨*unique.label*⟩)
4. REMAPS ⟨*identifier*⟩
5. REMAPS ⟨*array.identifier*⟩

If one knew the absolute address of some data structure in memory, the following could be done:

```
DECLARE

    1    SAVE.AREA REMAPS ABSOLUTE 1024,
      2  SA.FIRST.ITEM    FIXED,
      2  SA.SECOND.ITEM   CHARACTER(200),
      2  SA.THIRD.ITEM    BIT(256);
```

The following technique could be used when a ⟨*label*⟩ denoting the start of a table of constants was present in a program:

```
DECLARE
    1    TRACE.TABLE(10) REMAPS ADDRESS (TRACE.MNEMONICS),
      2  ADDR.TRACE.NAME    CHARACTER (4);%

 %
TRACE.MNEMONICS
 TABLE
 BEGIN
     "LA  "
     "ALA "
     "STN "
     "STD "
     "LIT "
     "ILA "
     "STO "
     "CASE"
     "IFTH"
     "IFEL"
 END
 %
 MOVE ADDR.TRACE.NAME (2) TO FA
 READ 24 BITS TO X INC FA
 READ 8 BITS TO Y
```

Note the use of ⟨*array*⟩ in the above example. If the programmer does not know the index to use at compile time, the following could be done:

```
DEFINE TRACE.INDEX = SOB
%
MOVE TRACE.INDEX TO X
MOVE LENGTH.BETWEEN.ENTRIES (TRACE.TABLE) TO Y
CALL MULTIPLY.X.Y
% AND SO FORTH
%
MULTIPLY.X.Y
    % MULTIPLICATION CODE
    EXIT
```

The above examples have shown, among other things, two of the "specials" that are included in MIL syntax to augment usage of DECLAREd items. They are:

DATA.LENGTH (⟨*declared.identifier*⟩)
LENGTH.BETWEEN.ENTRIES (⟨*array.identifier*⟩)

Note that when ⟨*array*⟩ names are used with the specials, the subscript is not included; it is syntactically invalid to do so.

Another type of remaps is one that remaps a previously declared structure. In this case, the addresses of the remap structure will begin at the address of the remapped structure.

Example

```
DECLARE
    1       SAVE.AREA.CHARS REMAPS SA.SECOND.ITEM
       2    SA.NAME                     CHARACTER(30),
          3 (SA.PACK.ID,
             SA.FAMILY.NAME,
             SA.OFFSPRING NAME)         CHARACTER(10),
       2    SA.OWNER.NAME               CHARACTER(14);%
% THE RIGHTMOST  156 CHARACTERS OF SA.SECOND.ITEM ARE NOT REMAPPED HERE
```

REVERSE

"REVERSE" is an attribute that may be applied to a remapping simple item or structure. The presence of this reserved work causes the address associated with a ⟨*declared identifier*⟩ to be the normally-calculated address minus its declared length.

For example, suppose a programmer wishes to specify a structure that describes the top memory and wants to list the ⟨*identifier*⟩s from the top of memory downward. The following could be done:

```
DECLARE
    1         TOP.OF.MEMORY REMAPS BASE.ZERO REVERSE,
       2      FILLER                        BIT(32),
       2      ADDR.INTERRUPT.QUEUE          BIT(553),
       2      ADDR.SAVED.A.STACK            BIT(240),
       2      ADDR.GISMO.WORK.SPACE         BIT(384),
       2      ADDR.TEMP.FIB                 BIT(920),
       2      ADDR.TRACE.SPACE              BIT(2232),
          3   ADDR.TRACE.CODE               BIT(24);
```

These ⟨ *identifier*⟩s could then be used in MIL statements as follows:

```
MOVE ADDR.INTERRUPT.QUEUE TO Y
EXTRACT ADDR.TRACE.CODE FROM T TO X
```

7. REGISTERS AND SCRATCHPAD

GENERAL

This section is intended only as a brief overview of the registers within the processor. It is assumed that the reader is familiar with the contents of the B 1700 Systems Reference Manual (form 1057155). (See also Appendix B in this manual).

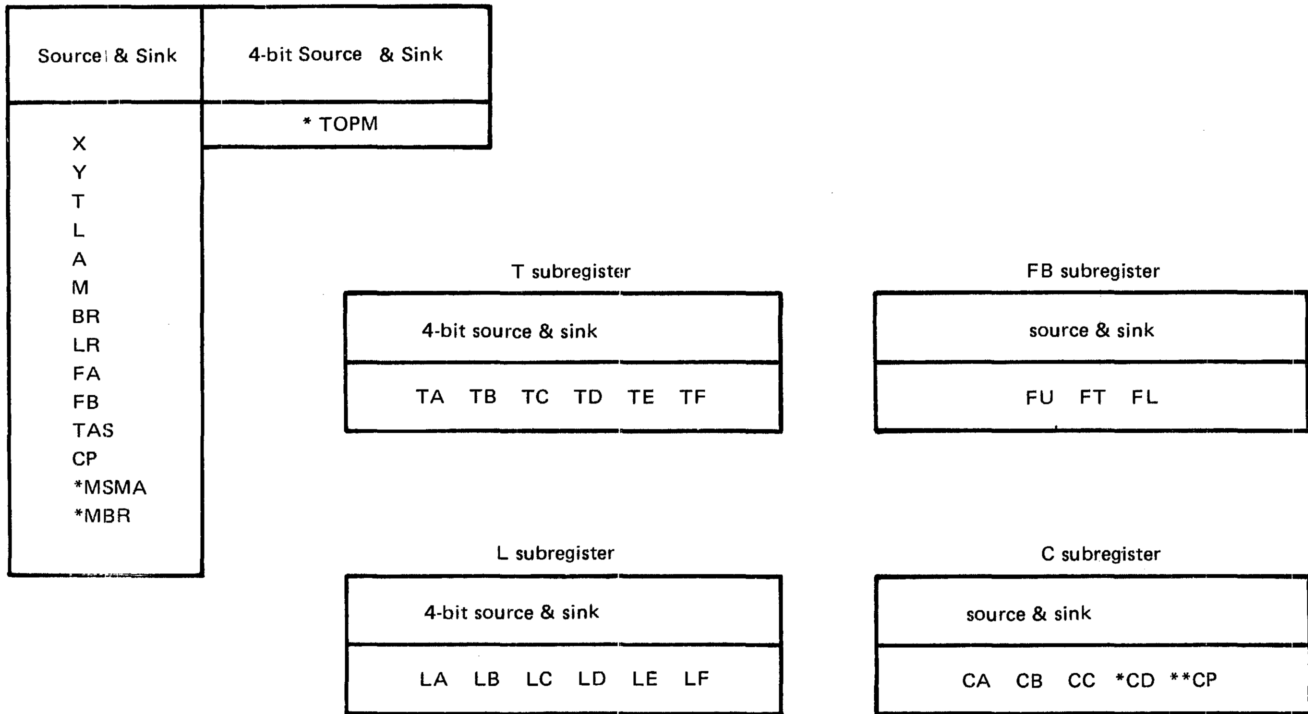
NOTE

The most-significant (left-most) bit in any register is identified in the MIL syntax as bit 0 (zero), the next most-significant as bit 1, etc. This is particularly advantageous in a bit-addressable machine since, for software purposes, it is often desirable to think of a register as being an extension of main memory. It should be noted that this convention is at variance with the hardware bit numbering convention where, generally, all bits are numbered right to left, 0 through N. This difference has particular significance when any bit data is to be OR'ed into the M register at run time.

REGISTER GROUPS

The registers briefly described in this section are divided into the following logical groups:

Active registers
Result registers
Scratchpads
Constant registers
Input/Output registers
Condition registers



* MSMA, TOPM, MBR and the low order 3 bits of CD are not physically present in the S-Memory Processor. When addressed as a source they will yield a binary value of zero. When addressed as a sink (destination) the data is lost.

** CPU, a 2-bit subregister of CP, is not addressable as a source or a sink.

RESULT REGISTERS

Source
SUM CMPX CMPY XANY XEOY MSKX MSKY OXRY DIFF

SCRATCHPAD

Single Scratchpad

Source & Sink
S0A -- S15A S0B -- S15B

Double Scratchpad

Source & Sink
S0 -- S15

CONSTANT REGISTERS

Source
MAXS MAXM

INPUT/OUTPUT REGISTERS

Source	Sink	Source & Sink
U	CMND	DATA

CONDITION REGISTERS

4-bit Source
BICN FLCN * INCN XYCN YXST

* INCN is not physically present on the S-Memory Processor. When addressed as a source it yields a binary value of 0. When addressed as a sink (destination) the data is lost.

ALPHABETICAL LISTING OF REGISTERS AND KEY CONCEPTS

Name	Length In Bits	Source Sink	Note
A	*	so & sk	Control Memory Micro-instruction Address * 24 (1726), 19 (S-1), 20 (S-2)
BICN	4	source	boolean conditions
BR	24	so & sk	Base Register or low address S-Memory protection
C	24	---	Control; not addressable as a unit
CA	4	so & sk	subfield of C; general purpose
CB	4	so & sk	subfield of C; general purpose
CC	4	so & sk	subfield of C; interrupts and flags
CD	4	so & sk	subfield of C; interrupts and flags
CMND	24	sink	I/O Command Register
CMPX	24	source	Result: complement of X; masked by CPL
CMPY	24	source	Result: complement of Y; masked by CPL
Console Switches	24	source	the 24 toggle switches located on the Console front panel
Control Memory	16-bit words	so & sk	Location of micro-instructions on M-Memory Processor
CP	8	so & sk	Control Parallel; subfield of C
CPL	5	so & sk	Control Parallel Length; subfield of CP
CPU	2	---	Control Parallel Unit; subfield of CP
CYD	1	---	Carry Difference or carry of borrow
CYF	1	---	Carry Flip-Flop; subfield of CP
CYL	1	---	Carry Level or carry of sum; masked by CPL
DATA	24	so & sk	I/O Data Register
DIFF	24	source	result of X-(Y + CYF); masked by CPL
F	48	---	Field in S-Memory; FA and FB concatenated

Name	Length In Bits	Source Sink	Note
FA	24	so & sk	Field Address in S-Memory
FB	24	so & sk	Concatenation of S-Memory Field Unit (FU), Field Type(FT), and Field Length(FL)
FL	16	so & sk	Field Length in S-Memory
FT	4	so & sk	subfield of FB
FLC	4	so & sk	subfield of FL
FLD	4	so & sk	subfield of FL
FLE	4	so & sk	subfield of FL
FLF	4	so & sk	subfield of FL
FLCN	4	source	boolean Field Length Conditions
FU	4	so & sk	S-Memory Field Unit size; subfield of FB
INCN	4	source	boolean dispatch Interrupt Conditions M-Memory Processor
L	24	so & sk	Local register also used in DISPATCH, OVERLAY, TRANSFER.CONTROL, READ/WRITE MSML AND S-MEMORY ACCESS
LA	4	so & sk	subfield of L
LB	4	so & sk	subfield of L
LC	4	so & sk	subfield of L
LD	4	so & sk	subfield of L
LE	4	so & sk	subfield of L
LF	4	so & sk	subfield of L
LR	24	so & sk	Limit Register or high address S-Memory protection
M	16	so & sk	current Micro-instruction register
MBR	24	so & sk	Main Memory Micro-instruction Base Register; not on S-Memory Processor
MAXM	24	source	hardwired Constant; number of 16-bit words of M-Memory

Name	Length In Bits	Source Sink	Note
MAXS	24	source	Constant; size in bits of available S-Memory
MSKX	24	source	Result; mask of X; length by CPL
MSKY	24	source	Result; mask of Y; length of CPL
MSMA	16	so & sk	Control Memory addressed by the A register; M-Memory Processor only
Main Memory	--	---	S-Memory
NULL	24	so & sk	always zero
PERR	4	source	Parity Error Register; reflects error conditions from S & M-Memory, and cassette
READ	24	source	Console switch position; reads S-Memory addressed by FA to Console lights (A on 1714)
SFL	16	---	subfield of SOB corresponding to FL in FB
S0-S15	48	so & sk	Double Scratchpad Words
S15A-S15B	48	so & sk	Single Scratchpad Words of S15
S-Memory	–	---	Main Memory
SU	4	---	subfield of SOB corresponding to FU in FB
SUM	24	source	Result (X + Y + CYF) length; masked by CPL
T	24	so & sk	Transform - will ROTATE, SHIFT or EXTRACT bits; used also in S-MEMORY ACCESS and TRANSFER.CONTROL
TAS	24	so & sk	Top of A Register-Stack
TA	4	so & sk	subfield of T
TB	4	so & sk	subfield of T
TC	4	so & sk	subfield of T
TD	4	so & sk	subfield of T
TE	4	so & sk	subfield of T
TF	4	so & sk	subfield of T

Name	Length In Bits	Source Sink	Note
TOPM	4	so & sk	Top of Control Memory; not on S-Memory Processor
U	16	source	cassette input only
WRIT	24	---	Console position switch; writes Console switches to address of memory contained in FA (A on 1714)
X	24	so & sk	input to Function Box
XANY	24	source	Result; X and Y; length by CPL
XEOY	24	source	Result; X EOR Y; length by CPL
XORY	24	source	Result; X OR Y; length by CPL
XY	48	source	X AND Y concatenated
XYCN	4	source	boolean XY Conditions
XYST	4	source	boolean XY States
Y	24	so & sk	input to Function Box

ACTIVE REGISTERS

The following are descriptions of the active registers:

X and Y Registers

The X and Y registers (both of which are 24 bits wide) are used as inputs into the 24-bit Function Box (see below). All functions are performed under control of the C (Control) register, which regulates the length of the operation, class of arithmetics, and least-significant carry input. The X and Y registers are capable of being shifted or rotated individually or as a unit and may receive or transmit data from or to main memory.

Field (F) Register

The F register is divided into FA and FB, each sub-register being 24 bits wide. The FA (Field Address) portion is used to address main memory. FB is divided into FU (Field Unit, consisting of four bits used to indicate arithmetic unit size; FT (Field Type), a general-purpose 4-bit field; and FL (Field Length), consisting of 16 bits used to indicate the length of fields in main memory. FL is subdivided into FLC, FLD, FLE and FLF, each four bits in length.

Local (L) Register

The L register is 24 bits wide and is subdivided into LA, LB, LC, LD, LE and LF, each four bits in length. L and its subdivisions are generally used to temporarily hold the contents of other processor registers. It is also used as a source and destination for main memory access and has implicit use in the DISPATCH, OVERLAY, READ/WRITE MSML and TRANSFER CONTROL micro-instructions.

Transform (T) Register

The T register is a 24-bit transformation register used extensively for interpretation of virtual-language operators. It is subdivided into TA, TB, TC, TD, TE and TF, each four bits in length. T has strong SHIFT and EXTRACT logics associated with it and is the principal formatting register of the processor. This register also has the capability of receiving or transmitting data from and to main memory.

Micro-Instruction (M) Register

The M register is a 16-bit register which holds the micro-operator for decoding and subsequent execution by the hardware. It is addressable as a source and sink register; when used as a sink register the source is bit-ORed with the upcoming M-op, except in TAPE mode.

Base (BR) and Limit (LR) Registers

The BR and LR registers are each 24 bits wide and are used to hold the main memory base and limit addresses for the currently active main memory process. The M- Memory processor hardware uses these registers to determine if addresses in the Field Address (FA) register are within the base/limit boundaries.

Address (A) Register

The A register is the microprogram address register which contains the bit address of the next micro-instruction. Values in the A register are always MOD 16; i.e., the low-order four bits are always zero. It is capable of addressing 16,384 micro-instructions located in either control memory or main memory or both. The A register is automatically incremented to the next micro-instruction before the current micro-instruction is executed. It is also capable of having any value from 0 to 4,095 added to or subtracted from it to facilitate microcode branching.

A Stack (TAS)

The A stack is a 32-element-deep, 24-bit wide, push-down, pop-up memory, i.e., a last-in-first-out (LIFO) storage structure. The A stack is used to nest microroutine linkages and allows highly shared routines, thus reducing control memory requirements. Although the A stack was intended for microcode addresses, it has been made 24-bits wide to allow for any operand storage.

NOTE

The S-Memory Processor A stack has only 16 storage elements.

Top of Control Memory (TOPM) Register

M-Processor Only

The TOPM register is four bits wide and is used to determine which memory (control or main) contains the next micro-instruction. If the A register is equal to or greater than (TOPM*512*16), the next micro-instruction will be fetched from main memory rather than control memory. The TOPM register is addressable as a source or as a sink (destination). The fetch from S-Memory takes place at address A+MBR.

Memory Base (MBR) Register

M-Processor Only

The MBR register is used with the A and TOPM registers to obtain the main memory address of the next micro-instruction. (See above formula). The MBR register is addressable as both a source and as a sink.

Control (C) Register

The C register is a 24-bit control register for the microprocessor. It contains the 24-bit Function Box controls and carry input plus some of the processor interrupts and flags. It is subdivided into CA, CB, CC, CD, each four bits wide; and CP, eight bits wide. CA and CB may be used as general-purpose registers. CC and CD represent processor interrupts and flags (see discussion under Condition Registers below). CP contains Function Box controls: CYF (0 bit of CP), CPU (1 and 2 bits of CP), and CPL (3,4,5,6, and 7 bits of CP). CYF (Carry Flip Flop) notifies the Function Box that a previous unit carry must be added to its summary results. CPU (Control Parallel Unit) notifies the Function Box of the type of unit contained in X and Y: 00 = binary, 01 = 4-bit decimal. CPL (Control Parallel Length) specifies the width, in bits, of the Function Box and Read/Write micro-instructions.

Combinatorial Logic or Functional Box

The Combinatorial Logic, often called the Function Box, produces the Result Registers. Inputs are the X register, the Y register and the Carry Flip-Flop (CYF). The inputs are combined under control of the Conrol Parallel Unit (CPU) register and the Control Parallel Length (CPL) register. When values are loaded into the X and Y registers, a large collection of output values and comparisons (called Result Registers) is made available to all subsequent micro-instructions.

RESULT REGISTERS

The Result registers are outputs from the 24-bit Function Box. Their contents are produced immediately and automatically from the inputs to the Function Box (X, Y and CYF) and cannot be changed except by changing inputs or by changing CPU (Control Parallel Unit) or CPL (Control Parallel Length). If the value of CPL is less than 24, then the (24-CPL) most-significant bits of all Result registers will be zero. These registers are source registers only and therefore cannot be used as the sink (destination) register in a MOVE or in any other instruction.

XORY Result Register

This register contains the INCLUSIVE OR of the X register combined with the Y register. This is a bit by bit operation with corresponding pairs of bits treated independently.

XANY Result Register

This register contains the AND of the X register combined with the Y register. This is the logical product of the X register and the Y register. Corresponding pairs of bits are treated independently.

XEOY Result Register

This register contains the EXCLUSIVE OR of the X register and Y register.

CMPX Result Register

This register contains the 1's complement of the X register.

CMPY Result Register

This register contains the 1's complement of the Y register.

MSKX Result Register

Masked X contains the low-order bits of the X register. The value of CPL determines the number of bits placed in MSKX. All other high-order bits are zero. If CPL is equal to 24, then MSKX is identical to the X register.

MSKY Result Register

Masked Y contains the low-order bits of the Y register. The value of CPL determines the number of bits placed in MSKY. All other high-order bits are zero. If CPL is equal to 24, MSKY is identical to the Y register.

SUM Result Register

SUM is the decimal or binary value (determined by CPU) of the X register plus the Y register plus the CYF register. Corresponding pairs of bits are grouped by CPU control, and grouping may be binary or 4-bit decimal. If the sum of (X + Y + CYF) is larger than the size specified by CPL, then the CYL, (Carry Level) will be true (one). CYL may be gated into CYF through use of the CARRY micro-instruction.

Difference (DIFF) Result Register

DIFF stores the amount resulting from the subtraction of the sum of the contents of the Y and CYF registers from the contents of the X register. The contents of the CPU register determine whether the subtraction is decimal or binary. Corresponding pairs of bits are grouped by CPU. If the difference is negative, X-(Y+CYF) 0, then Diff Result will be in 2's complement form of 10's complement form depending upon the mode, either binary or decimal respectively; and CYD (Carry Difference) will be true (one).

NOTE

The CYD register is not conditioned by CPL; it is always based on a 24-bit comparison. The programmer, therefore, must know what is in the high-order positions of the X register and the Y register if CPL is less than 24.

SCRATCHPAD

The scratchpad can be used for temporary storage of active registers. The scratchpad may be addressed as sixteen, 48-bit double words or thirty-two, 24-bit words.

Scratchpad Words - 24 Bits Each

S0A	S4A	S8A	S12A
S0B	S4B	S8B	S12B
S1A	S5A	S9A	S13A
S1B	S5B	S9B	S13B
S2A	S6A	S10A	S14A
S2B	S6B	S10B	S14B
S3A	S7A	S11A	S15A
S3B	S7B	S11B	S15B

Double Scratchpad Words - 48 Bits Each

S0	S4	S8	S12
S1	S5	S9	S13
S2	S6	S10	S14
S3	S7	S11	S15

(Sn = SnA and SnB concatenated, where n = 0 through 15)

CONSTANT REGISTERS

The following is a description of the constant registers.

Maximum Main Memory (MAXS) Register

The 24-bit MAXS register is set by the field engineer and contains the value of the maximum installed number of main memory bits. It is addressable as a source only. Main memory addresses begin at zero. The lower 15 bits are always zero, i.e., MAXS has a 4096 byte (32K bit) resolution.

Maximum Control Memory (MAXM) Register

The 24-bit MAXM register is set by the field engineer and contains the value of the maximum installed number of control memory words, each word comprising 16 bits. It is addressable as a source only. The lower 10 bits are always zero, i.e., MAXM has a 1024 word resolution. On the B 1710 series MAXM will always contain zero.

NULL Register

The NULL register is a 24-bit, addressable field of zeros. It may be addressed as source of sink; in the latter case it accepts the data but remains zero.

INPUT/OUTPUT REGISTERS

The following is a description of the Input/Output registers.

Console.Switches

M-Processor Only

This 24-bit register reflects the current state of the 24 Console Switches on the processor.

Console Cassette Tape Input (U) Register

The U register accumulates the data read from the tape cassette on the Console control panel. It is addressable as a source in the RUN mode with the MOVE REGISTER micro-instruction and in the TAPE mode with the MOVE 24-BIT LITERAL micro-instruction. (See MIL Statements: LOAD.MSMA.) It is not addressable as a sink.

Command (CMND) Register

The CMND register is used to transfer commands to the I/O controls. It is 24 bits wide and is addressable as a sink only.

Data Register

The DATA register is used to transfer data to and from the I/O controls and their peripherals. It is 24 bits wide and is addressable as a source or as a sink.

CONDITION REGISTERS

Introduction

There are five Condition registers:

Binary Conditions (BICN)
Field Length Conditions (FLCN)
Interrupt Conditions (INCN)
X AND/OR Y registers(s) Conditions (XYCN)
X AND/OR Y registers(s) States Conditions (XYST)

Each Condition register consists of four bits. The bits are identified from left to right and are assigned the position numbers 0 through 3, with 0 being the most-significant bit.

All Condition registers are source registers only. They may be moved to another register or tested, using the IF and SKIP instructions, for their current contents. They may not be the sink (destination) register of any micro-instruction.

BIT	BICN	XYCN	XYST	FLCN	INCN
0	LSUY	MSBX	LSUX	FL = SFL	NO-DEVICE
1	CYF	X = Y	ANY.INTERRUPT	FL SFL	HI-PRIORITY
2	CYD	X Y	Y NEQ 0	FL SFL	INTERRUPT
3	CYL	X Y	X NEQ 0	FL NEQ 0	LOCKOUT

Binary Conditions (BICN) Register

LSUY is true if the least-significant unit of the Y register is 1 and the Control Parallel Unit (CPU) register specifies binary (CPU = 0); or 9 and the CPU register specifies decimal (CPU = 1).

The Carry Flip-flop (CYF) register indicates the value of the carry-in in the Control Parallel (CP) register. The CYF register may be manipulated as part of the CP register and by the CARRY instruction.

The Carry Difference (CYD) register is true if X-(CYF + Y)   0. This condition is not affected by CPL, i.e., a 24-bit compare is always made.

The Carry Level (CYL) register is true if (X + Y + CYF), limited by the Control Parallel Length (CPL) register, overflows.

XY Conditions (XYCN) Register

MSBX is true if the most-significant bit of the X register, as determined by the Control Parallel Length (CPL) register, is a 1.

NOTE

The comparisons of the X register to the Y register are not affected by CPL; they are always 24-bit compares.

XY States (XYST) Register

LSUX is true if the least-significant unit of the X register is 1 and the Control Parallel Unit (CPU) register specifies binary (CPU = 0); or is 9 and the Control Parallel Unit (CPU) register specifies decimal (CPU = 1). The comparisons of the X register or the Y register to zero are not affected by CPL; all 24 bits of the X register and/or the Y register are used in the comparisons.

Any.Interrupt Bit

This bit is true if any of the following conditions in registers CC, CD, of INCN (M-Memory Processor) are true:

Event	Register (Bit Position)
MISSING DEVICE	INCN (0)
PORT INTERRUPT	INCN (2)
I/O SERVICE REQUEST INTERRUPT	CC (2)
CONSOLE INTERRUPT	CC (3)
MAIN MEMORY READ PARITY ERROR INTERRUPT	CD (0)
MEMORY WRITE/SWAP ADDRESS OUT OF BOUNDS INTERRUPT	CD (3)

The CC and CD registers are both 4-bit source and sink (destination) registers within the C register. The bits in each are numbered 0 through 3, with bit 0 being the most significant. They have been assigned the following uses and meanings:

CC (0)	STATE LIGHT
CC (1)	TIMER INTERRUPT
CC (2)	I/O SERVICE REQUEST INTERRUPT
CC (3)	CONSOLE INTERRUPT
CD (0)	MAIN MEMORY PARITY ERROR
CD (1)	MAIN MEMORY WRITE/SWAP ERROR OVERRIDE
CD (2)	MAIN MEMORY READ OUT OF BOUNDS ERROR
CD (3)	MAIN MEMORY WRITE/SWAP OUT OF BOUNDS ERROR

All bits in the CC and CD portions of the C register, once set, remain set even though the conditions that caused them to be set may no longer exist. Therefore, if it is desired to clear any of these bits to zero, this must be done explicitly. CD (1), CD (2), and CD (3) of the C register are always zero in the S-Memory Processor but still may be addressed and tested.

Console Interrupt (CC(3))

This bit is set when the interrupt toggle switch on the Console control panel is turned on. It remains set as long as the switch is on. It can be reset programmatically but not by turning the Console toggle switch off. This bit is also reported in ANY.INTERRUPT when it is on.

Main Memory Read Parity Error Interrupt (CD(0))

This bit is set when a main memory parity error is detected during a READ or a READ portion of a SWAP operation or when an attempt is made to access non-existent main memory.

Main Memory Address Out-of-Bounds Override (CD(1))

M-Processor Only

This bit is tested if the Field Address (FA) register setting is less than the Base Register (BR) setting or greater than or equal to the Limit Register (LR) setting; then WRITE or SWAP operations will be inhibited unless this bit is set (one). The state of this bit does not affect the setting of CD (2) or CD (3).

Read Address Out-of-Bounds Interrupt (CD(2))

M-Processor Only

This bit is set when a READ operation is attempted and the Field Address (FA) register setting is either less than the Base Register (BR) setting or greater than or equal to the Limit Register (LR) setting. The READ operation is not inhibited.

Write/Swap Address Out-of-Bounds Interrupt (CD(3))

M-Processor Only

This bit is set when a WRITE or SWAP operation is attempted and the Field Address (FA) register setting is either less than the Base Register (BR) setting or greater than or equal to the Limit Register (LR) setting. This bit, when on, is also reported in ANY.INTERRUPT.

Field Length Conditions (FLCN) Register

All conditions are based upon comparisons between the 16 bits of the FL register and either zero or the corresponding low-order 16 bits of the first word in the Scratchpad (S0B).

Interrupt Conditions (INCN) Register

M-Processor Only

NO DEVICE is true if an interrupt message is present in the dispatch buffer for a port or channel which does not have a device attached to it. This condition is normally cleared by the processor with a DISPATCH READ AND CLEAR instruction.

HI PRIORITY is true if there is a high-priority message present in the dispatch buffer.

INTERRUPT is true if there is a message present in the dispatch buffer for the processor. This condition is normally cleared by a DISPATCH READ AND CLEAR instruction. It is also reported in ANY.INTERRUPT.

LOCKOUT is true if the interrupt system is locked (marked as "in use").

REGISTER DESIGNATIONS AND AREAS OF APPLICATION

The following is a list, arranged by areas of application, of registers and their associated designations.

MICRO-INSTRUCTION CONTROL

A	(Micro-instruction Address)
M	(Current Micro-instruction)
TAS	(Top of Address Stack)
TOPM	(Logical Top of M-Memory)
MBR	(Micro-instruction Base Register)

S-MEMORY CONTROL

BR	(Base Register)
LR	(Limit Register)
FA	(Field Address)
FL	(Field Length)
CP	(Control Parallel)

INTERRUPT CONTROL

CC
CD
INCN

PARALLEL WIDTH CONTROL

C
CP
CPL
CPU

ORGANIZATION OF FIELDS AND SUBFIELDS

The following is a description of the organization of register fields and subfields, expressed in the notation of MIL structured data declarations.

```
1  C  BIT(24),
     2   CA  BIT(4),
     2   CB  BIT(4),
     2   CC  BIT(4),
     2   CD  BIT(4),
     2   CP  BIT(8),
           3  CYF  BIT(1),
           3  CPU  BIT(2),
           3  CPL  BIT(5);
```

NOTE: C does not exist as a composite, only as subfields.

```
1     F  BIT(48),
        2   FA   BIT(24),
        2   FB   BIT(24),
              3  FU  BIT(4),
              3  FT  BIT(4),
              3  FL  BIT(16),
                   4  FLC  BIT(4),
                   4  FLD  BIT(4),
                   4  FLE  BIT(4),
                   4  FLF  BIT(4);
```

```
1  L  BIT(24),
     2   LA  BIT(4),
     2   LB  BIT(4),
     2   LC  BIT(4),
     2   LD  BIT(4),
     2   LE  BIT(4),
     2   LF  BIT(4);
```

```
1  T  BIT(24),
     2   TA   BIT(4),
     2   TB   BIT(4),
     2   TC   BIT(4),
     2   TD   BIT(4),
     2   TE   BIT(4),
     2   TF   BIT(4);
```

8. MIL STATEMENTS

INDEX TO STATEMENTS

Following is an alphabetical list of MIL statements found in this section.

Statement	Page	Statement	Page
ADD SCRATCHPAD	8-2	MAKE.SEGMENT.TABLE.ENTRY	8-59
ADJUST LOCATION	8-3	MICRO	8-60
AND	8-4	*M.MEMORY.BOUNDARY	8-61
ASSIGN	8-6	MONITOR	8-62
BEGIN	8-7	MOVE	8-63
BIAS	8-9	NOP	8-65
BRANCH.EXTERNAL	8-11	NORMALIZE	8-66
CALL	8-12	OR	8-67
CALL.EXTERNAL	8-13	*OVERLAY	8-69
CARRY	8-14	PAGE	8-70
CASSETTE	8-15	POINT	8-71
CLEAR	8-16	PROGRAM.LEVEL	8-72
CODE.SEGMENT	8-17	READ	8-73
COMPLEMENT	8-18	REDUNDANT.CODE	8-75
COUNT	8-20	RESERVE.SPACE	8-76
DEC	8-22	RESET	8-77
DEFINE	8-23	ROTATE	8-78
DEFINE.VALUE	8-24	SEGMENT	8-79
*DISPATCH	8-25	SET	8-80
EMIT.RETURN.TO.EXTERNAL	8-29	SHIFT/ROTATE T	8-82
ELSE	8-27	SHIFT/ROTATE X/Y/XY	8-84
END	8-30	SKIP	8-85
EOR	8-31	S.MEMORY.LOAD	8-87
EXIT	8-33	STORE	8-88
EXTRACT	8-34	SUB.TITLE	8-89
FA.POINTS	8-36	SUBTRACT SCRATCHPAD	8-90
FINI	8-37	*SWAP	8-91
GO TO	8-38	TABLE	8-92
HALT	8-39	TITLE	8-93
IF	8-40	TRANSFER.CONTROL	8-94
INC	8-46	WRITE	8-95
JUMP	8-47	WRITE.STRING	8-97
LIT	8-48	XCH	8-99
LOAD	8-49		
*LOAD.MSMA	8-50		
LOAD.SMEM	8-52		
LOCAL.DEFINES	8-53		
MACRO	8-55		

*Available on B 1720 systems only.

ADD SCRATCHPAD

Syntax

ADD ——→ ⟨*scratchpad · word*⟩ ——→ TO FA ——→|

Semantics

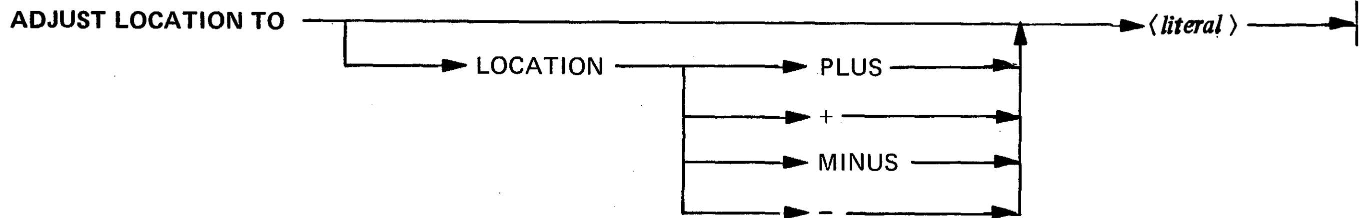
This instruction adds the left half of any scratchpad word (SOA . . . S15A) to the Field Address (FA) register. The result is placed in FA; the contents of ⟨*scratchpad.word*⟩ remain unchanged. (See also: SUBTRACT SCRATCHPAD.)

Example

ADD S9A TO FA

ADJUST

Syntax



Semantics

This pseudo-operation adjusts the ⟨*physical.code.address*⟩ of the compiler. The value of the ⟨*physical.code.address*⟩ specifies the location (control memory address) into which the next generated micro-instruction is to be placed, generally by a user-developed loader. (See also Segmentation: Label Addresses.)

LOCATION PLUS (+) OR MINUS (-) increments/decrements the ⟨*physical.code.address*⟩ by the value of the ⟨*literal*⟩. If this option is not used, the ⟨*physical.code.address*⟩ is set to the value of the ⟨*literal*⟩.

The ⟨*literal*⟩ must have a value of 0 MOD 16.

NOTE

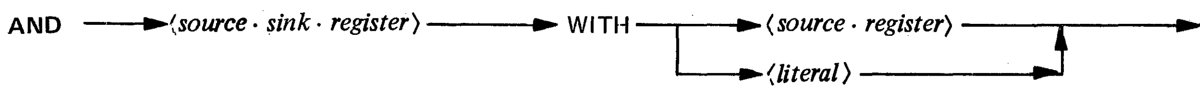
This instruction is generally used to compensate for disposable loader routines.

Examples

ADJUST LOCATION TO @100@
ADJUST LOCATION TO LOCATION + 32
ADJUST LOCATION TO LOCATION MINUS 128

AND

Syntax



Semantics

This instruction logically ANDs the contents of a 4-bit source and sink (destination) register with the bit configuration of the ⟨*literal*⟩ or the contents of a 4-bit source register. The result is placed in ⟨*source. sink.register*⟩; the contents of ⟨*source.register*⟩ remain unchanged. (See also: OR and EOR.)

The register may be any of the following:

⟨*source.sink.register*⟩	⟨*source.register*⟩
CA CB *CC *CD	⟨*source.sink.register*⟩
FT FU	BICN
FLC FLD FLE FLF	FLCN
LA LB LC LD LE LF	INCH (available on B1720 only)
TA TB TC TD TE TF	PERR (available on B1720 Model II only)
TOPM (available on B1720 only)	XYCN
	XYST

*CC and CD represent processor interrupts and flags

The ⟨*literal*⟩ has a decimal range from 0 to 15.

Table 8-1: AND Truth Table

Source . Sink . Register	⟨*Literal*⟩ or ⟨*Source.Register*⟩			Source . Sink . Register
0	AND	0	Yields	0
0	AND	1	Yields	0
1	AND	0	Yields	0
1	AND	1	Yields	1

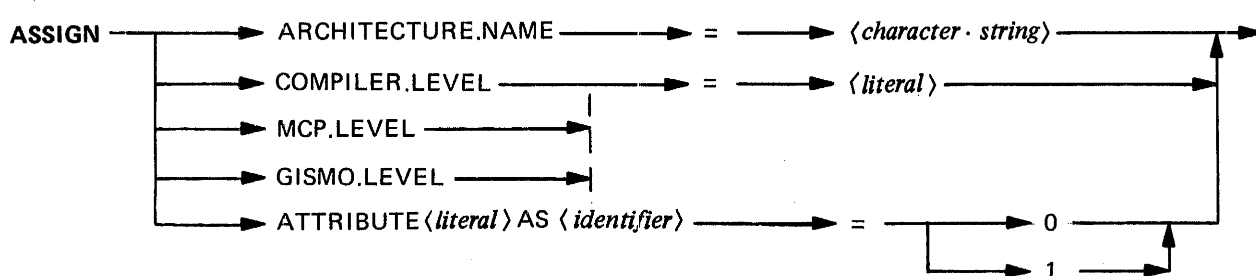
Example

AND TB WITH 3

	TA	TB	TC	TD	TE	TF	
T	0000	1010	1111	0011	0001	0010	before (0AF312)
	--	0011	--	--	--	--	literal (3)
T	0000	0010	1111	0011	0001	0010	after (02F312)

ASSIGN

Syntax



Semantics

This statement assigns values to the various interpreter verification attributes. These attributes occupy fields in the IPB (Interpreter Parameter Block) of all interpreters. They are accessed at BOJ (Beginning of Job) time by the MCP and are used to verify that the proper interpreter has been chosen.

The ⟨*character.string*⟩ for ARCHITECTURE.NAME must be a string of 10 or fewer ⟨*character*⟩s.

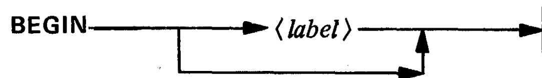
⟨*literal*⟩ has a decimal range from 0 to 255 for COMPILER.LEVEL, MCP.LEVEL, and GISMO.LEVEL; and from 0 to 79 for ATTRIBUTE.

Examples

ASSIGN ARCHITECTURE.NAME = "GISMO.26"
ASSIGN MCP.LEVEL = 197
ASSIGN ATTRIBUTE 64 AS ITEM.01=1

BEGIN

Syntax



Semantics

This statement is paired with the END statement to combine MIL statements into logical blocks. If the BEGIN/END block is labeled, the MIL program listing will reflect the first ten letters of the block name on every line of the block (See example in Programming Technique section).

The BEGIN/END block is useful in an IF statement when more than one statement is necessary following a condition.

Example

```
IF condition THEN
BEGIN TRUE.CONDITION
            :
END TRUE.CONDITION
ELSE
BEGIN FALSE.CONDITION
            :
END FALSE.CONDITION
```

BEGIN/END blocks may be nested to fifteen levels, meaning that no portion of a MIL program may reflect more than fifteen BEGIN's without matching END's.

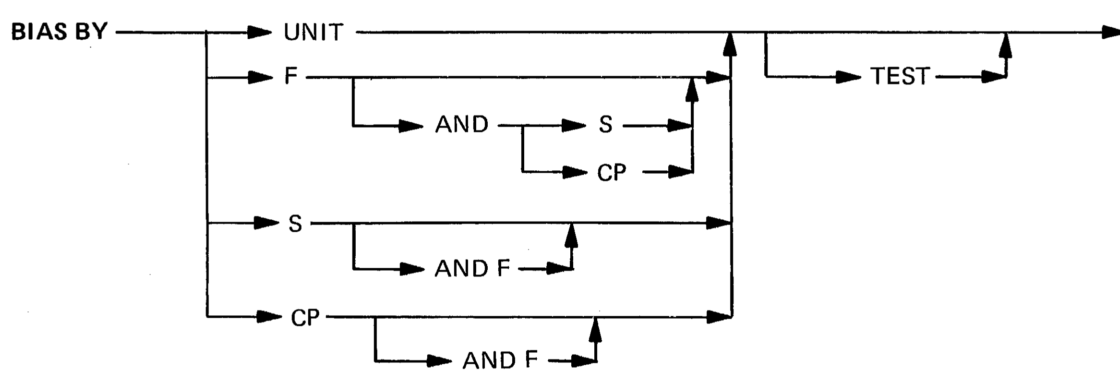
(See also: END statement and LOCAL.DEFINES statement)

Example

```
 BLOCK
NESTING
 LEVEL

   0            :
   0          BEGIN BLOCK.1
   1              :
   1              BEGIN ANOTHER.BLOCK
   2                  :
   2
   2
   2                  BEGIN INNERMOST.BLOCK
   3                      :
   3
   3                  END INNERMOST.BLOCK
   2                  :
   2              END ANOTHER.BLOCK
   1              :
   1          END BLOCK.1
                :
```

BIAS

Syntax



Semantics

This instruction sets the Control Parallel Length (CPL) register and the Control Parallel Unit (CPU) register to values calculated from the given operands.

NOTE

All references to register S refer to the SFL or SFU registers in the second half of the first scratchpad word, e.g., the SFL (low order 16 bits) part of the S0B register.

The CPU register will be set to 1 if the value of the Field Unit (FU) register is set to 4 or 8; otherwise CPU is set to 0. This is done for all variations of BIAS except BIAS BY S, which sets the CPU register from SFU rather than from the FU register.

BIAS BY . . . sets the CPL register equal to 24 or to the value in the specified register if it is less than 24. BIAS BY UNIT sets the CPL register equal to the FU register (4 for 4-bit decimal, 8 for 8-bit decimal, or any other value less than 16 for binary).

If the TEST option is used the above actions are performed, and the next micro-instruction is skipped if CPL has not been set to zero.

Examples

BIAS BY F	This instruction sets the CPL register to 24 or to the value of the Field Length (FL) register, if it is less than 24. It also sets the CPU register equal to the unit in the FU register.
BIAS BY F AND CP	This instruction sets the CPL register to 24, to the value in the FL register, or to the value in the CPL register, whichever is the smallest. It also sets the CPU register to the unit in the FU register.

BIAS BY UNIT

This instruction sets the CPL register equal to the length of the unit of the type specified by the FU register. It also sets the CPU register equal to one unit of the type specified in the FU register, i.e., 4-bit decimal, 8-bit decimal, or binary.

NOTE

In all cases except UNIT, CPU is set to 1 if FU (or SFU) is 4 or 8; otherwise CPU is set to 0. If UNIT is specified, CPL is set directly to the value in FU.

BRANCH.EXTERNAL

Syntax

BRANCH.EXTERNAL ——→ TO ——→ ⟨*label*⟩ ——→|

Semantics

This instruction transfers control to the external segment location specified by ⟨*label*⟩. (See: Segmentation.)

⟨*Label*⟩ must be associated with a run-time address that has a displacement from the BRANCH.EXTERNAL instruction of less than 4096 micro-instructions.

NOTE

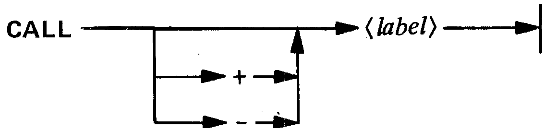
> If an external segment does not exist because $NO EXTERNAL has been specified, BRANCH. EXTERNAL is equivalent to GO TO.

Example

BRANCH.EXTERNAL TO EXTERNAL.SEGMENT.LABEL

CALL

Syntax



Semantics

This instruction stores the address of the next micro-instruction in the A stack, then branches to the location specified by ⟨ *label*⟩.

The location specified by the label may be a maximum of 4095 micro-instructions away from the CALL instruction.

Example

CALL MULTIPLICATION.ROUTINE
CALL M.IN.OUT
CALL +ABC

CALL.EXTERNAL

Syntax

CALL.EXTERNAL ——► ⟨*label*⟩ ——►|

Semantics

This instruction stores the address of the next micro-instruction in the A stack, then branches to the external segment location specified by ⟨ *label*⟩. (See: Segmentation.)

⟨ *Label*⟩ must be associated with a run-time address that has a displacement from the CALL.EXTERNAL instruction of less than 4096 micro-instructions.

NOTE

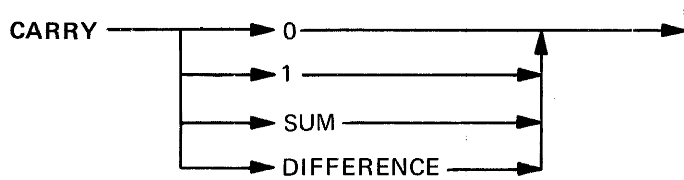
> If an external segment does not exist, becuase $NO EXTERNAL has been specified, CALL.EXTERNAL acts identically to CALL.

Example

CALL.EXTERNAL BEGINNING.OF.LOOP.1

CARRY

Syntax



Semantics

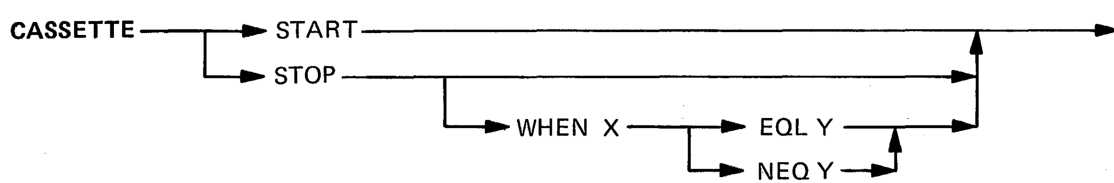
This instruction sets the Carry (CYF) register to either 0 or 1. CARRY 0 or CARRY 1 sets the CYF register to 0 or 1 respectively. CARRY SUM sets the CYF register to the value of CYL single bit. CARRY DIFFERENCE sets the CYF register to the value of the Carry Difference (CYD) register:

If:	X>Y	X=Y AND CYF=0	X=Y AND CYF=1	X<Y
CYD is set to:	0	0	1	1

The CYD register, unlike the CYL register is not conditioned by the CPL register. That is, all 24 bits of the X and Y registers are compared when setting CYF by the CYD register. The programmer should, therefore, know what is in the high-order position of the X and Y registers when using the CYD register if the CPL register is set to less than 24.

CASSETTE

Syntax



Semantics

This instruction causes the system cassette tape to start or stop a READ operation at the next inter-record gap.

The information read from the cassette is loaded into the U register and remains there for a maximum of two clock cycles before the U register is cleared.

NOTE

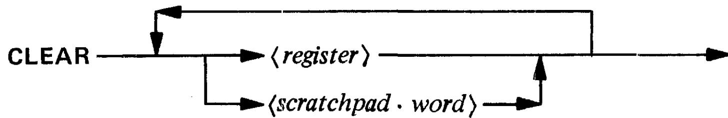
The data on the cassette is duplicated every eight bits to ensure its validity. The cassette will discriminate against parity incorrect data and, if necessary, use the duplicate eight bits. If both copies are in error, the load will be aborted. If the STEP-RUN-TAPE switch is in the TAPE position (see: LOAD. MSMA) and the START button is pushed, the successive 2-byte increments will be moved from the U register. If the instruction being executed is a 24-bit literal MOVE TO MSMA, then the next 16 bits (2 bytes) that appear in the U register are loaded into control memory at the address indicated by the A register. The A register is then incremented by 1.

Example

```
CASSETTE STOP
CASSETTE STOP WHEN X NEQ Y
```

CLEAR

Syntax



Semantics

This instruction sets the specified register(s) or 24-bit scratchpad word(s) to zero.

The following may be cleared:

⟨register⟩	*⟨scratchpad.word⟩*
A	S0A
BR	-----
CA CB *CC *CD CP CPU	S15A
FA FB FL FT FU	
FLC FLD FLE FLF	S0B
LA LB LC LD LE LF	-----
TA TB TC TD TE TF TAS	S15B
TOPM (available on B1720 only)	

*CC and CD represent processor interrupts and flags

Each register clear takes one clock cycle; each scratchpad word clear takes two clock cycles.

NOTE

MOVE NULL TO ⟨*register*⟩ will be generated for each register specified on B 1710 systems.

Example

```
CLEAR S10A
CLEAR BR  L  CB  S4B  TOPM  FU
```

CODE.SEGMENT

Syntax

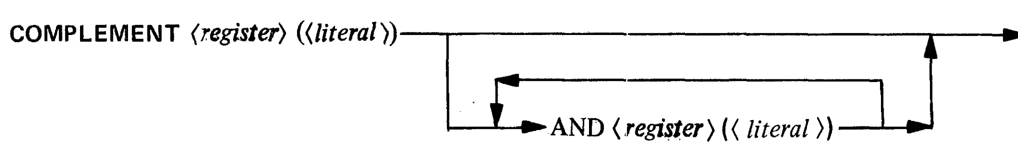
CODE.SEGMENT ——————► <*label*> ———►|

Semantics

See Segmentation: CODE.SEGMENT

COMPLEMENT

Syntax



Semantics

This instruction COMPLEMENTs (switches the state of) the specified bit. By using the options, more than one bit in any one register can be complemented with the same instruction IF ALL BITS ARE IN THE SAME 4-BIT REGISTER. (See also: SET and RESET.)

The ⟨*register*⟩ may be any 4-bit source and sink (destination) register below:

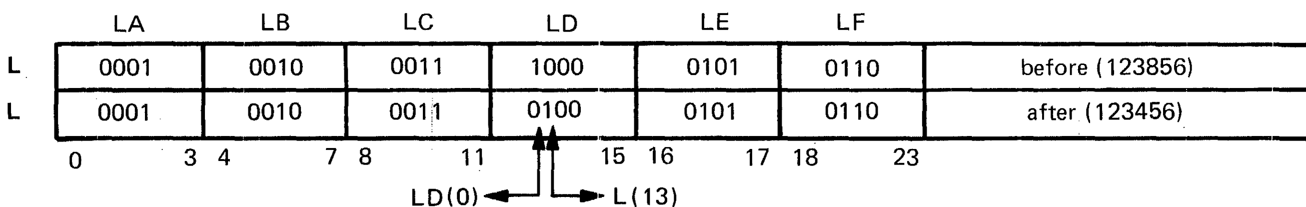
CA CB CC CD (CC and CD represent processor interrupts and flags)
FT FU
FLC FLD FLE FLF
LA LB LC LD LE LF
TA TB TC TD TE TF
TOPM (available on B1720 only)

It may also be the FL, FB, L, or T register: all bits must then be in the same 4-bit subfield.

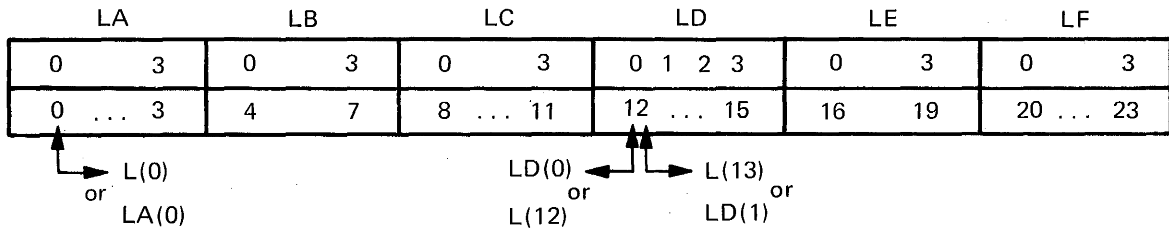
The literal has a decimal range from 0 to 3 for a 4-bit register; from 0 to 15 for the FL register; and from 0 to 23 for the FB, L, and T registers.

Example

COMPLEMENT LD(0) AND L(13)

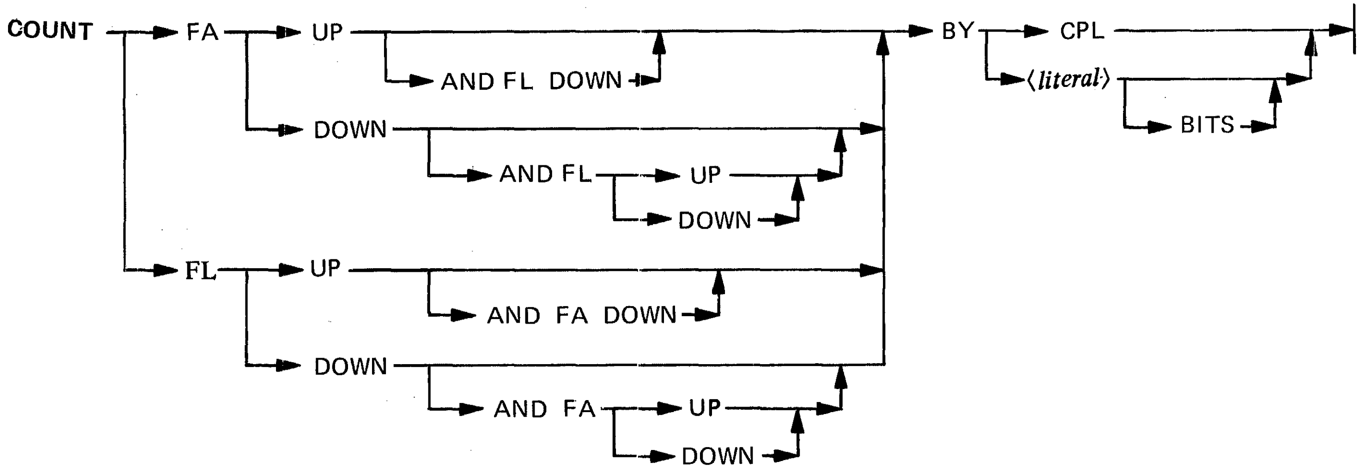


It should be noted that most registers can be addressed in either of two ways:



COUNT

Syntax



Semantics

This instruction increments or decrements the designated registers by the value of the ⟨ *literal*⟩ or the contents of the Control Parallel Length (CPL) register. If the value of ⟨ *literal*⟩ is 0, the value contained in the CPL register is used.

If the FA register is counted down, it may pass through 0 (i.e., if FA=0 and is counted down by 1, it will be set to hexadecimal FFFFFF). If the FL register is counted down, it will not become less than 0.

If either the FA or FL register overflows, wraparound to or through 0 will occur; e.g., if either is equal to the maximum value it can contain and is counted up by 1, it becomes equal to 0.

The ⟨ *literal*⟩ has a maximum decimal value of 72.

Example

Count FA Up and FL Down by 10

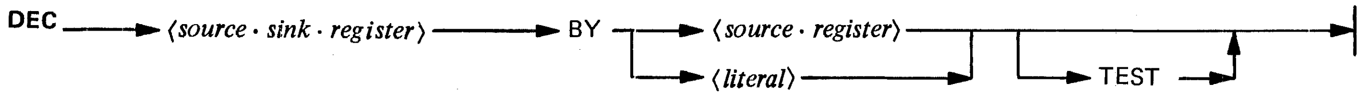
FA	0000	1001	1010	0111	1111	1011	before (09A7FB)
	--	--	--	--	--	1010	⟨*literal*⟩ +A
FA	0000	1001	1010	1000	0000	0101	after (09A805)

FL	0000	0000	0000	1000	before (0008)
	--	--	--	1010	⟨*literal*⟩ -A
FL	0000	0000	0000	0000	after (0000)

FA is counted up by decimal 10 (hexadecimal A), while FL is counted down by 8 to its minimum value.

DEC

Syntax



Semantics

This instruction decrements the contents of a 4-bit ⟨*source.sink.register*⟩ by the value of the ⟨*literal*⟩ or the contents of a 4-bit ⟨*source.register*⟩. The result is placed in ⟨*source.sink.register*⟩; the contents of ⟨*source.register*⟩ remain unchanged. (see also: INC.)

The registers may be any of the following:

⟨*source.sink.register*⟩	⟨*source.register*⟩
CA CB *CC *CD	⟨*source.sink.register*⟩
FT FU	BICN
FLC FLD FLE FLF	FLCN
LA LB LC LD LE LF	INCN (available on B1720 only)
TA TB TC TD TE IF	XYCN
TOPM (available on B 1720 only)	XYST

*CC and CD represent processor interrupts and flags

The ⟨*literal*⟩ has a decimal range from 0 to 15.

If the TEST option is used and ⟨*source.sink.register*⟩ underflows (is decremented beyond 0, the smallest value it can contain), the next micro-instruction is skipped. If underflow does not occur or if the TEST option is not used, the next micro-instruction is executed.

NOTE

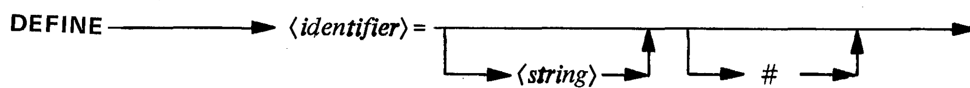
> All 4-bit registers count modulo 16; e.g., if a register contains a value of 0 and is decremented by 2, it underflows to a value of 14.

Example

```
DEC TB BY 7
DEC FLD BY LC TEST
```

DEFINE

Syntax



Semantics

This declaration assigns a name (⟨*identifier*⟩) to a string of characters. Any subsequent reference to the ⟨*identifier*⟩ is replaced by the ⟨*string*⟩. The pound sign merely clarifies the end of the DEFINE ⟨*string*⟩ if present.

⟨*String*⟩ may be a scratchpad name (24 or 48-bit); a register name; a ⟨*literal*⟩; a part of one instruction; an entire instruction, part of which may have been previously DEFINED; or empty. It may neither begin with a pound sign (#) nor contain any embedded pound signs.

The entire DEFINE declaration must be contained on one card, and all DEFINEs must be declared prior to any executable instruction.

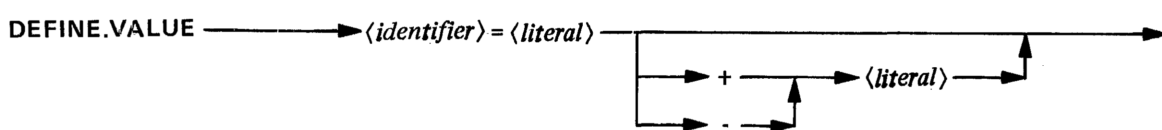
Nested DEFINEs are allowed up to 13 levels.

Example

```
DEFINE SOURCE.POINTER = S3#                 %   LOAD F FROM SOURCE.POINTER
DEFINE OP.REG = L                           %   CLEAR OP.REG
DEFINE TEST.OP = @80 0000@#                 %   MOVE TEST-OP TO OP.REG
DEFINE HINT = CC(3)                         %   RESET HINT
DEFINE IGNORE.HALT = RESET HINT             %   IGNORE.HALT
DEFINE LAZY = WRITE 21 BITS FROM X INC FA AND DEC FL#% FINGER SAVER
```

DEFINE.VALUE

Syntax



Semantics

This instruction assigns the value of the ⟨*arithmetic.expression*⟩ to the ⟨*identifier*⟩. Any occurrence of the ⟨*identifier*⟩ in the program is replaced by its assigned value.

DEFINE.VALUE creates a 24-bit ⟨*literal*⟩. Values less than zero are in 2's complement notation and are 24 bits long.

If defined ⟨*identifier*⟩ are used as ⟨*literals*⟩ in the ⟨*arithmetic.expression*⟩, they must be previously defined.

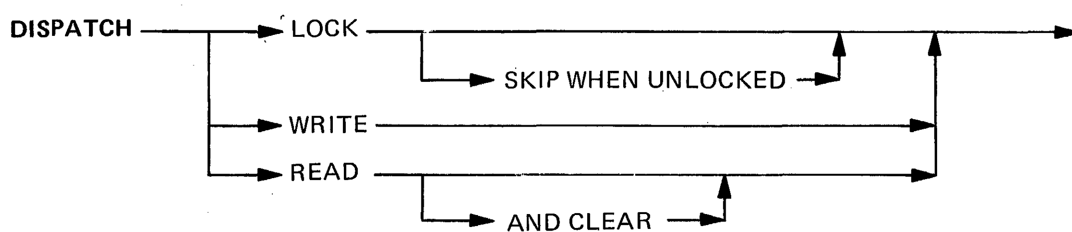
Example

```
DEFINE. VALUE AA = @50@                 % VALUE is hex 000050
DEFINE B = AA + 1                       % VALUE is hex 000051
DEFINE C = AA - 3                       % VALUE is hex 00004D
DEFINE.VALUE F03 = @(1)0010@ + 4        % VALUE is hex 000006
```

DISPATCH

(available only on B 1720 systems with port interchange)

Syntax



Semantics

This instruction sends a message (e.g. an I/O descriptor address) from the processor to a device on an I/O port.

Before sending a message to a port, the processor should first attempt to gain control of the interrupt system with a DISPATCH LOCK. This is necessary because the interrupt system is shared by all ports.

DISPATCH LOCK locks (marks as "in use") the interrupt system. If the interrupt system was already locked, the next micro-instruction is skipped.

DISPATCH LOCK SKIP WHEN UNLOCKED locks the interrupt system and skips the next micro-instruction if the interrupt system is already unlocked.

DISPATCH WRITE sends a 24-bit message to a port. Before a DISPATCH WRITE is executed, the L register must contain the 24-bit message; the seven least-significant bits of the T register must contain the destination port (bits 17-19) and channel numbers (bits 20-23). The contents of the L register are then stored in the Dispatch buffer (main memory locations 0-23), and the port and channel numbers are transferred to a hardware register (Dispatch register) in the port interchange. The contents of the L and T register remain unchanged.

DISPATCH READ transfers both a 24-bit message from the Dispatch buffer to the L register, and the source port and channel numbers to the seven least-significant bits of the T register.

NOTE

> If T (23) is found set after a DISPATCH READ and the source port is an I/O multiplexor, a main memory parity error was encountered during the fetch of an I/O descriptor address or an I/O descriptor, or during a RESULT SWAP operation. Consequently, the message transferred to the L register will be the address +24 of the parity error.

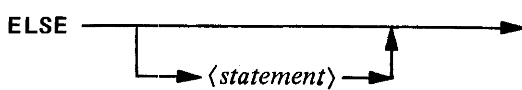
DISPATCH READ AND CLEAR does everything a DISPATCH READ will do and in addition clears the Interrupt Condition (INCN) register. That is, it resets all INCN bits to zero.

Only the least-significant seven bits of the T register are involved in any DISPATCH operation.

If the SKIP WHEN UNLOCKED option is used with any variant other than a DISPATCH LOCK, the next micro-instruction is skipped.

ELSE

Syntax



Semantics

The ELSE statement is used in conjunction with the IF statement to indicate that a ⟨*statement*⟩ is to be executed on a condition False. For example:

```
IF condition THEN
     statement
ELSE
     statement
```

The statement following the THEN clause will only be executed if the condition is true. Likewise, the statement following the ELSE clause is executed only if the condition is false.

The IF statement may also contain a BEGIN/END block following the THEN clause, in which case the ELSE clause becomes part of the END statement (see: END).

Examples

a.
```
If condition THEN
     statement
ELSE
     statement
```

b.
```
IF condition THEN
BEGIN
  :
END ELSE
     statement
```

c.
```
IF condition THEN
     statement
ELSE
BEGIN
  :
END
```

d. IF condition THEN

```
BEGIN
   ⋮

END ELSE
BEGIN
   ⋮
END
```

EMIT.RETURN.TO.EXTERNAL

Syntax

Semantics

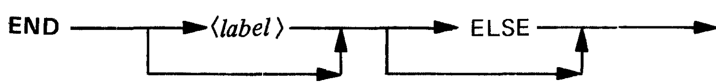
This instruction causes the compiler to emit the common code necessary to get back to the main segment from the external segment. The compiler automatically emits this code when the first CODE.SEGMENT statement is encountered. If the program requires the code to be emitted before the first CODE.SEGMENT is encountered, this statement can be used to emit the code. This code also includes the return code used when the segment is exited for the last time. (See: Segmentation.)

RESTRICTION

This statement cannot be used more than once in a program, and cannot be used after the occurrence of the first CODE.SEGMENT statement.

END

Syntax



Semantics

This statement is paired with the BEGIN statement to combine MIL statements into logical blocks. The END statement must have the same ⟨*label*⟩ as its matching BEGIN statement.

The ELSE clause is used only when needed as part of an IF statement. For example:

BLOCK NESTING LEVEL	MIL STATEMENT
0	IF condition THEN
0	BEGIN TEST.TRUE.BLOCK
1	
1	⋮
1	
1	END TEST.TRUE.BLOCK ELSE
0	BEGIN TEST.FALSE.BLOCK
1	
1	⋮
1	
1	END TEST.FALSE.BLOCK

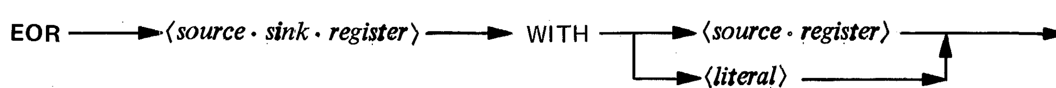
Good programming practice recommends that BEGIN's and matching END's start in the same column while the statements within the block should be indented to reflect the nesting level. (See also: BEGIN and LOCAL.DEFINES).

Example

```
IF LD(2) FALSE THEN
BEGIN EMIT.INFO
    WRITE 16 BITS FROM T INC FA AND DEC FL
    MOVE X TO S7A
    SET CB(1)
END EMIT.INFO ELSE
    MOVE Y TO S7B
```

EOR

Syntax



Semantics

This instruction logically EXCLUSIVE ORs the bits in a 4-bit ⟨*source.register*⟩ with the value of the literal or the contents of a 4-bit ⟨*source.register*⟩. The result is placed in ⟨*source.sink.register*⟩; the contents of ⟨*source.register*⟩ remain unchanged. (See also: AND and OR.)

The register may be any of the following:

⟨*source.sink.register*⟩	⟨*source. register*⟩
CA CB *CC *CD	⟨*source.sink.register*⟩
FT FU	BICN
FLC FLD FLE FLF	FLCN
LA LB LC LD LE LF	INCN (available on B1720 only)
TA TB TC TD TE TF	PERR (available on B1720 only)
TOPM (available on B1720 only)	XYCN
	XYST

*CC and CD represent processor interrupts and flags.

The ⟨*literal*⟩ has a decimal range from 0 to 15.

Table 8-2 EOR Truth Table

Source.Sink.Register		⟨*literal*⟩ Source.Register		Source.Sink.Register
0	EOR	0	Yields	0
0	EOR	1	Yields	1
1	EOR	0	Yields	1
1	EOR	1	Yields	0

Example

EOR TB WITH 3

	TA	TB	TC	TD	TE	TF	
T	0000	0101	1111	0011	0001	0010	before (05F312)
	--	0011	--	--	--	--	EOR (030000)
T	0000	0110	1111	0011	0001	0010	after (06F213)

EXIT

Syntax

```
EXIT ──────────►|
```

Semantics

This instruction returns program control to the calling routine by causing the compiler to generate a MOVE TAS TO A operation.

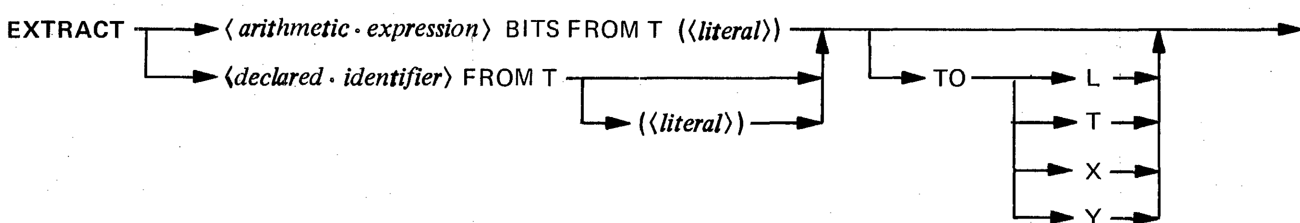
The top of the A stack (TAS) is moved to the ADDRESS (A) register, which is used by the hardware logic as the address of the next micro-instruction to be fetched. The stack is decremented automatically by the hardware after the move.

NOTE

MOVE TAS TO A may be used instead of EXIT with the same result.

EXTRACT

Syntax



Semantics

This instruction isolates the specified bits from the T register and moves them to a distination register (L, T, X, Y). If a destination register is not specified, T is assumed.

The value of the following combinations may not exceed 24 bits:

⟨arithmetic.expression⟩ +*⟨literal⟩*.
DATA.LENGTH of *⟨declared.identifier⟩*
DATA.LENGTH of *⟨declared.identifier⟩*+*⟨literal⟩*
DATA.LENGTH of *⟨declared.identifier⟩* + DATA.ADDRESS of *⟨declared.identifier⟩*

NOTE

If the starting bit for *⟨declared.identifier⟩* is not specified, its DATA.ADDRESS is used.

Examples

```
EXTRACT   4  BITS  FROM   T(20)   TO   L
DECLARE
    1      STUFF  REMAPS    BASE.ZERO     BIT        (24),
       2   ITEM.1                         BIT         (4),
       2   ITEM.2                         BIT       (127),
       2   ITEM.3                   CHARACTER         (1);
MOVE STUFF TO FA
ADD BASE TO FA
READ DATA.LENGTH (STUFF) BITS TO T
EXTRACT ITEM.2 FROM T TO X
EXTRACT ITEM.1 FROM T(0) TO T
```

EXTRACT 4 BITS FROM T(20) TO L

	TA	TB	TC	TD	TE	TF	
T	0000	0001	0011	1000	1110	0100	before (0138E4
						T(20)	

	LA	LB	LC	LD	LE	LF	
L	1001	1110	0011	1001	1111	1100	before (1E39FC)
L	0000	0000	0000	0000	0000	0100	after (000004)

Register T remains unchanged while its four extracted bits are placed in the L register. The bits are right-justified; leading zeroes are added.

NOTE

EXTRACT 0 BITS FROM T(23) TO a destination register may be specified, but the programmer must OR into the M register the number of bits to be extracted. Caution must be exercised, however, when ORing into the M register: the machine hardware instruction requires the right-bit pointer for the extraction field, not the left. The hardware also indexes the T register from 1 to 24, left to right, not 0 to 23; the compiler performs this conversion.

FA.POINTS

Syntax

FA.POINTS TO ———► ⟨*arithmetic expression*⟩ ———►|

Semantics

This pseudo-operation does not generate any code. It merely informs the compiler of the current contents of FA. This information is then used when compiling the POINT constructs in the READ, WRITE and POINT instructions.

The FA.POINTS and POINT constructs are provided so that the user may symbolically reference the memory structures declared in a declaration statement. Such references will show up in a cross-reference listing and can often result in automatic code changes when the declaration needs to be changed.

Example

```
DECLARE
        1  STRUCTURE,
                2    DATA.A BIT(10),
                2    DATA.B CHARACTER(20),
                2    DATA.C FIXED;

FA.POINTS TO DATA.A
READ DATA.LENGTH (DATA.A) BITS TO X POINT FA TO DATA.B
POINT FA TO STRUCTURE
MOVE DATA.C TO FA
WRITE DATA.LENGTH (DATA.C) BITS FROM Y POINT FA TO DATA.B
```

FINI

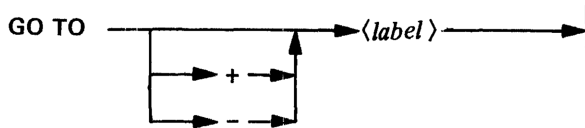
Syntax

Semantics

This instruction signals the compiler that the end of the file of source images has been reached. It should be the last statement in the source program.

GO TO

Syntax



Semantics

This instruction transfers control to the location specified by ⟨ *label*⟩. ⟨ *Label*⟩ must be associated with a run time address that has a displacement from the GO TO instruction of less than 4096 micro-instructions.

Example

GO TO SORT.ROUTINE
GO TO -LOOP.1
GO TO +LOOP.2

HALT

Syntax

```
HALT ──────────────►|
```

Semantics

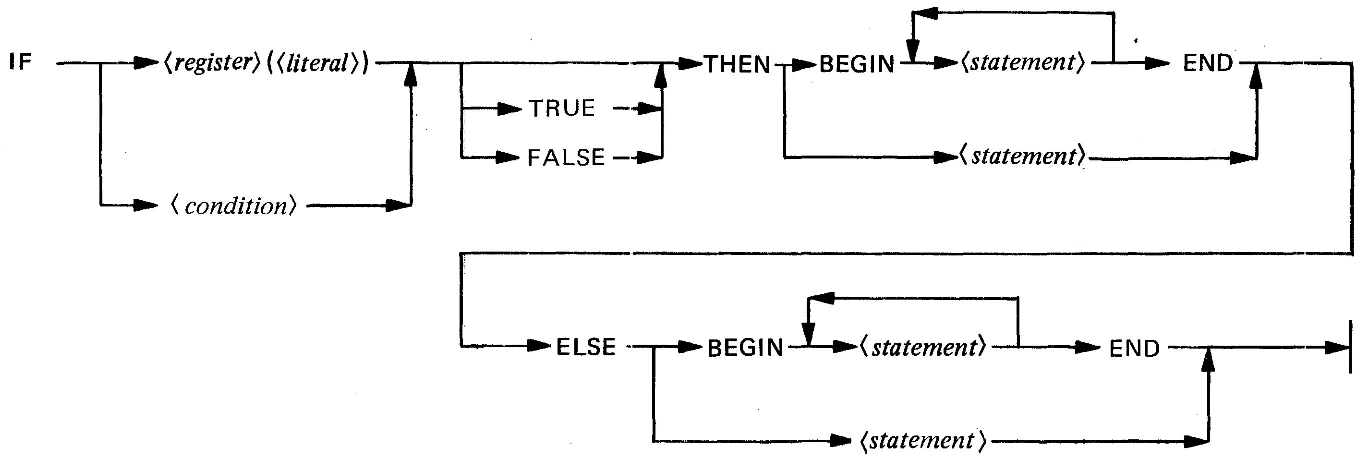
This instruction brings the processor to an orderly halt. The settings of the register select dials determine the register displayed.

Pressing the START pushbutton on the system Console will cause the processor to again begin executing micro-instructions. If the STEP/RUN switch is in the STEP position, only one micro-instruction is executed each time the START pushbutton is depressed.

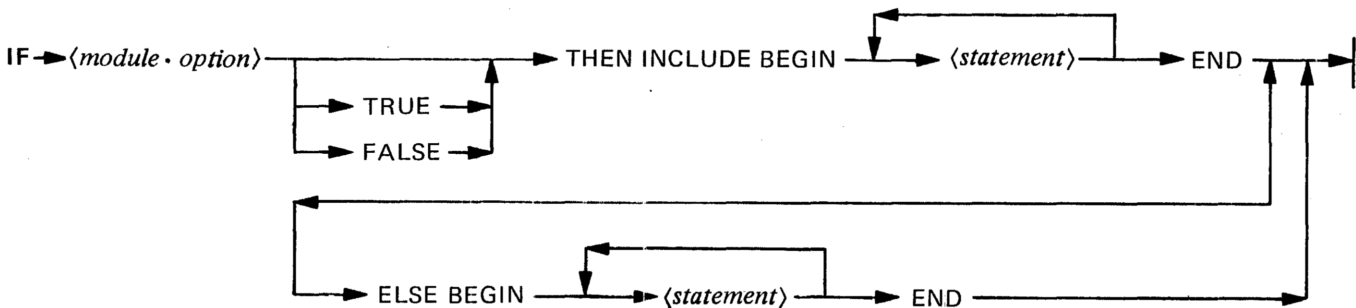
IF

Syntax

FORMAT 1: CONDITIONAL PROGRAM CONTROL



FORMAT 2: CONDITIONAL COMPILATION CONTROL



Semantics

FORMAT 1: CONDITIONAL PROGRAM CONTROL

This instruction tests a bit(s) for TRUE (one) or FALSE (zero), If the test condition is met, either the specified single statement or the specified BEGIN/END statement(s) is executed. If the test condition is not met, a branch around the first BEGIN/END pair is taken, and the ELSE BEGIN/END statement(s) is executed. Logical operators are valid on the registers immediately following the IF, enabling more than one bit to be tested at the same time, but only if all of the bits are in the same 4-bit register. (See also: COMPLEMENT, SET and RESET.)

Logical operators are valid on the registers immediately following the IF, with the following restrictions:

1) All registers logically related must be within the same 4-bit group: IF T(0) and T(3) is valid, IF T(2) and T(4) is not.

2) Only two register elements may be logically related: IF T(2) or T(0) is valid, IF T(2) and T(1) and T(0) is not.

3) NOT logic may be applied anywhere: IF NOT (L(3) or NOT L(0)) is valid.

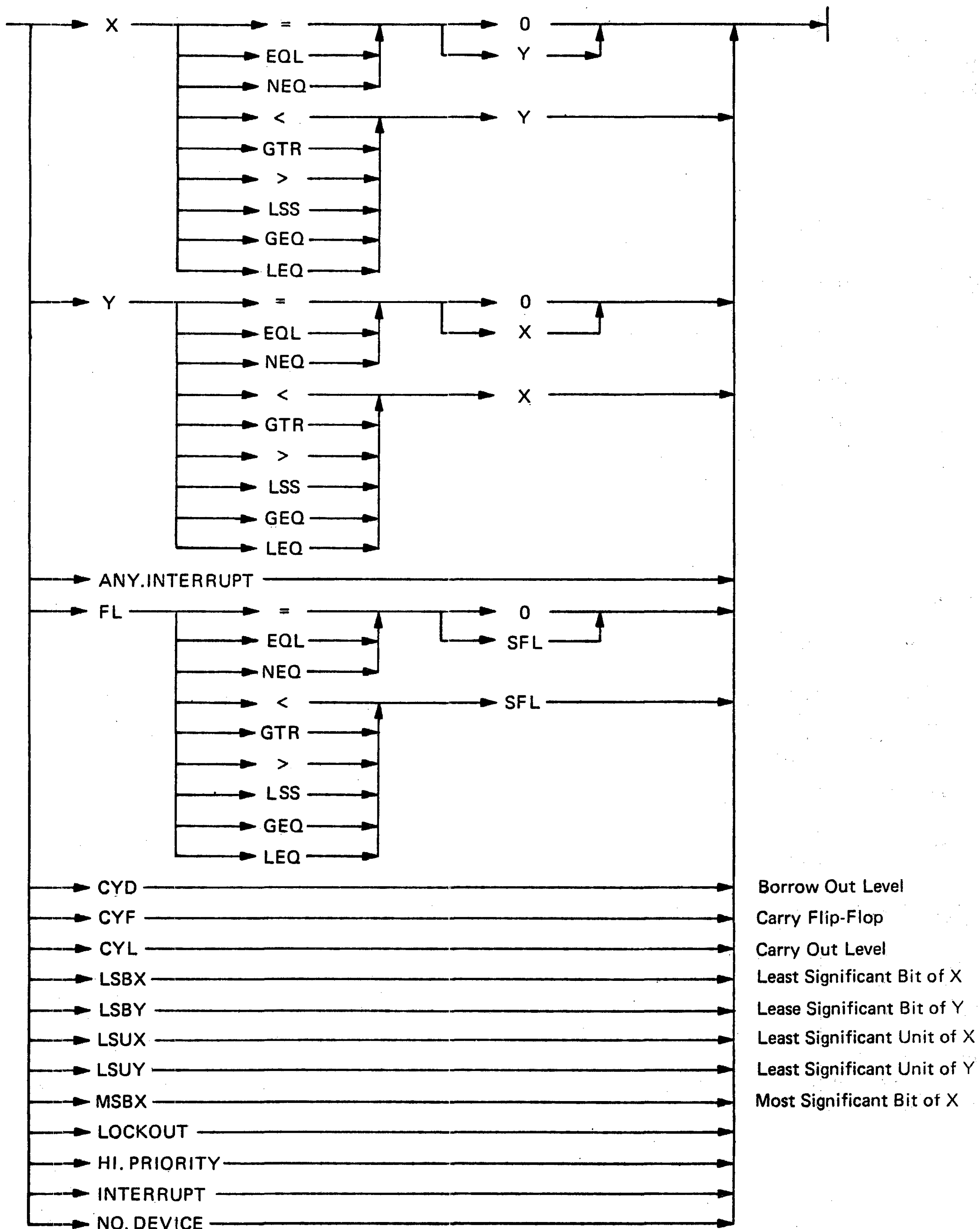
The ⟨*register*⟩ may be any 4-bit source and sink (destination) register below:

CA CB CC CD (CC and CD represent processor interrupts and flags)
FT FU
FLC FLD FLE FLF
LA LB LC LD LE LF
TA TB TC TD TE TF
TOPM (available on B1720 only)

The ⟨*register*⟩ may also be the FL, FB, L, or T register: all bits must then be in the same 4-bit subfield.

The ⟨*literal*⟩ points to the bit position which is to be tested. It has a decimal range from 0 to 3 for a 4-bit ⟨*register*⟩; from 0 to 15 for the FL register; and from 0 to 23 for the FB, L and T registers.

The condition may be any of the following conditions available from the condition registers:



X
=
EQL
NEQ
<
GTR
>
LSS
GEQ
LEQ
0
Y
Y
Y
=
EQL
NEQ
<
GTR
>
LSS
GEQ
LEQ
0
X
X
ANY.INTERRUPT
FL
=
EQL
NEQ
<
GTR
>
LSS
GEQ
LEQ
0
SFL
SFL
CYD
CYF
CYL
LSBX
LSBY
LSUX
LSUY
MSBX
LOCKOUT
HI. PRIORITY
INTERRUPT
NO. DEVICE
Borrow Out Level
Carry Flip-Flop
Carry Out Level
Least Significant Bit of X
Lease Significant Bit of Y
Least Significant Unit of X
Least Significant Unit of Y
Most Significant Bit of X

Any combination of conditions that is contained in one condition register can be tested using AND/OR logic if all bits can be tested for TRUE (on) or FALSE (off). For example, the following are valid conditions:

```
CYL AND LSUY
CYL OR CYD
```

EXAMPLE:
```
IF CYL AND LSUY TRUE THEN GO TO END.OF.ROUTINE
IF CYL OR CYD FALSE THEN GO TO LOOP
```

If TRUE or FALSE is not specified, TRUE is assumed.

EXAMPLE:
```
IF TD(2) THEN GO TO LABL7
```

Register TD	Branch to LABL7
0101	NO (bit position two is OFF)
1101	NO (bit position two is OFF)
0111	YES (bit position two is ON)
0011	YES (bit position two is ON)

Note: TD(2) could have been referred to as T(14)

Example

The following examples illustrate Format 1: Conditional Program Control:

```
IF  X = Y   THEN GO TO +A
IF  TB(1)   OR TB(3)  THEN EXIT
IF  LF(2)   THEN
            MOVE X TO Y
IF  FU(1)   FALSE THEN
            COMPLEMENT T(10)
ELSE
            RESET FL(5)
```

FORMAT 2: CONDITIONAL COMPILATION CONTROL

This instruction should be used for conditional inclusion of code, depending upon the setting of a user-defined, <*module.option*> toggle. This <*module.option*> toggle is declared and SET or RESET via a module option $ card. (See Appendix A: MIL Compiler Operation.)

More than one <*module.option*> toggle can be tested with the same IF statement by using AND/OR logic. If NOT is used in front of any <*module.option*> toggle, that <*module.option*> toggle is checked for the RESET state. If both TRUE and FALSE are omitted, TRUE is assumed.

NOTES

1. A conditional inclusion-block may not be used to include or exclude a BEGIN statement when the associated END statement is not part of the block.

2. Logical operators are valid on the registers immediately following the IF, with the following restrictions and capabilities:

   a. All registers logically related must be within the same 4-bit group: IF T(0) AND T(3) is valid, IF T(2) AND T(4) is not.

   b. Only two register elements may be logically related: IF T(2) or T(0) is valid, IF T(2) AND T(1) AND T(0) is not.

   c. NOT logic may be applied anywhere: IF NOT (L(3) OR NOT L(0)) is valid.

The following examples illustrate Format 1 (Conditional Program Control):

```
IF  X = Y THEN GO TO +A
IF TB(1) OR TB(3) THEN EXIT

IF LF(2) THEN
        MOVE X TO Y
SET TA(1)
IF FU(1) FALSE THEN
        COMPLEMENT T(10)
ELSE
    RESET FL(5)
SET L(6) AND L(7)

IF FLF(3) FALSE THEN
BEGIN
    RESET FB(1) AND FB(3)
    CLEAR S14A
END
XCH  S14  F  S14

IF LA(0) THEN
BEGIN
    MOVE TAS TO T
END ELSE
    MOVE FA TO T
MOVE TE TO LF

IF TD(3) THEN
    MOVE L TO X
ELSE
BEGIN
    MOVE T TO X
    MOVE SUM TO X
END
MOVE SUM TO FA
```

```
IF LA = 14 THEN
BEGIN
    MOVE 512 TO X
END
COMPLEMENT FU(0) AND FU(2)
```

The following are examples of conditional inclusion of code:

```
$ SET DEBUG,  RESET TRACE
$ SET TRACE,  RESET B1700
```

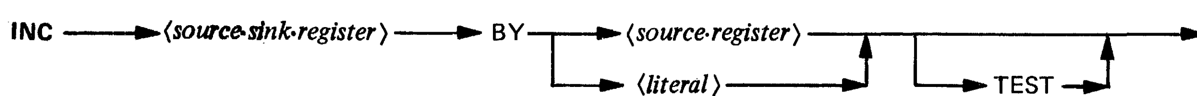
After processing these $ cards, the module options will be set TRUE or FALSE as follows:

```
DEBUG = TRUE
TRACE = TRUE
B1700 = FALSE

IF DEBUG THEN INCLUDE
        CALL DEBUG.ROUTINE
IF TRACE THEN INCLUDE
BEGIN
        CALL SAVE.REGISTERS
        CALL TRACE.ROUTINE
END
IF DEBUG AND NOT B1700 INCLUDE
BEGIN
        MOVE T TO X
END ELSE
BEGIN
        MOVE L TO X
        MOVE T TO S0A
END
IF NOT TRACE OR B1700 INCLUDE
BEGIN
        MOVE L TO X
        MOVE T TO S1A
END ELSE
BEGIN
        CALL TRACE.ROUTINE
        MOVE T TO X
END ELSE
BEGIN
        CALL TRACE.ROUTINE
        MOVE T TO X
END
```

Any of the preceding examples may be nested within any of the above BEGIN/END pairs up to a maximum of 15 levels. That is, at any given time during a compilation there may be at most 15 BEGINs that have not been paired with their respective ENDs.

INC

Syntax



Semantics

This instruction increments the contents of a 4-bit ⟨*source.sink.register*⟩ by the value of the ⟨*literal*⟩ or the contents of a 4-bit ⟨*source.register*⟩. The result is placed in the ⟨*source.sink.register*⟩; the contents of the ⟨*source.register*⟩; remain unchanged. (See also: DEC.)

The registers may be any of the following:

⟨*source.sink.register*⟩	⟨*source.register*⟩
CA CB *CC *CD	⟨*source.sink.register*⟩
FT FU	BICN
FLC FLD FLE FLF	FLCN
LA LB LC LD LE LF	INCN (available on B1720 only)
TA TB TC TD TE TF	XYCN
TOPM (available on B1720 only).	XYST

*CC and CD represent processor interrupts and flags

The literal has a decimal range from 0 to 15.

If the TEST option is used and source.sink.register overflows (is incremented beyond 15, the largest value it can contain), the next micro-instruction is skipped. If overflow does not occur or if the TEST option is not used, the next micro-instruction is executed.

NOTE

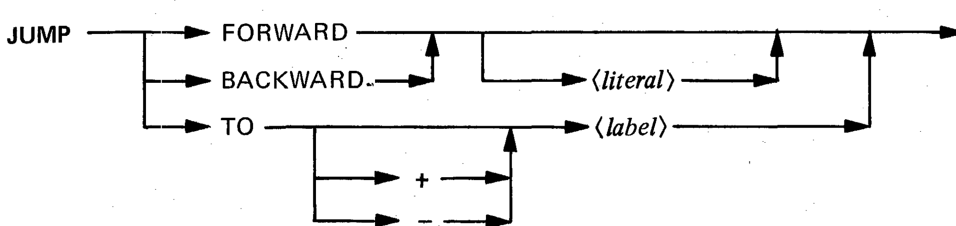
> All 4-bit registers count modulo 16; e.g., if a register contains a value of 15 and is incremented by 2, it overflows to a value of 1.

Example

```
INC LB BY 7
INC FLD BY BICN TEST
```

JUMP

Syntax



Semantics

This instruction transfers control to the designated location.

The address of ⟨*label*⟩ is limited to a maximum relative displacement of plus or minus 4095 micro-instructions.

The ⟨*literal*⟩ has a decimal range from 0 to 4095.

If ⟨*literal*⟩ is not specified, FORWARD/BACKWARD causes the compiler to generate a JUMP instruction with a displacement of zero and a direction sign of plus or minus. This is to facilitate ORing the actual displacement into the M register prior to the execution of a JUMP instruction.

Examples

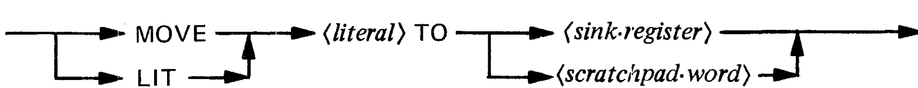
JUMP TO + LOOP.1
JUMP TO END.OF.CODE.LABEL
JUMP FORWARD
JUMP BACKWARD

NOTE

It is strongly recommended that only JUMP FORWARD and JUMP BACKWARD be used, that they be used only without a ⟨*literal*⟩, and only where necessary to generate a displacement of zero. Use the GO TO statement for all other uses.

LIT

Syntax



Semantics

This instruction moves a ⟨ *literal*⟩ to any sink (destination) register (except the M register) or to any 24-bit scratchpad word. (See also: MOVE.)

The ⟨ *literal*⟩ may be any decimal integer from 0 to 16777215, a hexadecimal number from @0@ to @FFFFFF@, a binary number from @(1)0@ to @(1)111111111111111111111111@, or a ⟨ *character.string*⟩ up to three characters in length. Leading zeros are not required unless the actual value of the ⟨ *literal*⟩ is zero. The value of the ⟨ *literal*⟩ should not exceed the maximum value that the ⟨ *sink.register*⟩ can contain; if less, left zero fill occurs.

⟨ *Literal*⟩ moves to a 24-bit scratchpad word generate MOVE ⟨ *literal*⟩ TO TAS followed by MOVE TAS TO ⟨ *scratchpad.word*⟩.

PROGRAMMING NOTE

It is recommended that the MOVE instruction be used instead of LIT.

Example

MOVE 12 TO L

	LA	LB	LC	LD	LE	LF	
L	0011	0000	1001	1010	0001	0011	before (309A13)
	--	--	--	--	--	**1100**	LIT (C)
L	0000	0000	0000	0000	0000	1100	after (00000C)

LOAD

Syntax

LOAD F FROM ———► <*double·scratchpad·word*> ———►|

Semantics

This instruction moves any 48-bit double scratchpad word (S0 . . . S15) to the Field (F) register.

NOTE

The compiler will generate two MOVE instructions for B 1710 systems.

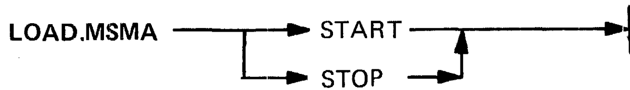
Example

LOAD F FROM S11

LOAD.MSMA

(Available on B 1720 systems only)

Syntax



Semantics

This pseudo-operation causes the compiler to either start or stop prefacing all emitted microcode with the first 16 bits of a MOVE 24 BIT LITERAL TO MSMA instruction.

The above action is required when a microprogram is to be loaded into control memory from a cassette tape while the system is in the TAPE mode. The action of the hardware while in this mode is as follows:

```
.READLOOP
        READ 16 BITS FROM THE CASSETTE TO THE U-REGISTER
        MOVE U TO M
        IF M = FIRST HALF OF 24-BIT LITERAL MOVE, THEN READ 16 BITS
             FROM THE CASSETTE TO U
        EXECUTE THE MICRO-OPERATION IN M
             (IF M= @9D00@=MOVE 24-BIT LITERAL TO THE CONTROL MEMORY
             WORD ADDRESSED BY THE A-REGISTER; THEN U, WHICH NOW
             CONTAINS THE ACTUAL MICRO-INSTRUCTION, IS MOVED TO
             CONTROL MEMORY ADDRESSED BY THE A-REGISTER AND A IS
             INCREMENTED BY 1)
        IF M = CASSETTE STOP THEN
             STOP CASSETTE AND HALT PROCESSOR
        ELSE
             JUMP TO -READLOOP
```

No statement between LOAD.MSMA START and its corresponding LOAD.MSMA STOP may reference any 〈 *label*〉 which has not been declared prior to the LOAD.MSMA STOP.

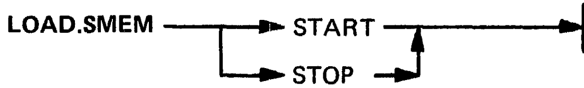
Example

The following source code could be used to enable a microprogram to be loaded from a cassette into control memory, beginning at control memory address zero:

```
MOVE 0 TO A
SEGMENT ANYNAME AT 0
LOAD.MSMA START
        ⋮
(Microprogram)
        ⋮
LOAD.MSMA STOP
MOVE 0 TO A
CASSETTE STOP
```

LOAD.SMEM

Syntax



Semantics

This pseudo-instruction causes the compiler to either start or stop appending each micro-instruction with the following instructions:

MOVE 24 BIT LITERAL TO X
WRITE (25) BITS FROM X
WRITE 16 BITS FROM X INC FA

These instructions are required when a microprogram is to be loaded into main memory from a cassette tape while the system is in the TAPE mode.

Example

```
MOVE 4096 TO FA        % START ADDRESS
LOAD.SMEM START

(microprogram)

LOAD.SMEM STOP
CASSETTE STOP
```

NOTE: The FA must start at a mod 32 value.

LOCAL.DEFINES

Syntax

```
LOCAL.DEFINES ──────────►|
```

Semantics

This statement is provided to allow the use of local definitions (see DEFINE, DEFINE.VALUE, DECLARE, MACRO). Local definitions are useful in limiting unauthorized or unnecessary access of special-purpose definitions outside of their only areas of use. LOCAL.DEFINES, however, does allow duplicate definitions with a special local meaning probably different from a more global meaning. Thus microprogrammers should be careful to avoid such potentially confusing duplications.

A definition which follows LOCAL.DEFINES has that definition only within the scope of the block in which it is defined. For example:

```
  BLOCK
 NESTING
  LEVEL          MIL STATEMENTS

    0        BEGIN LOCAL.BLOCK.1
    1            DECLARE L.1 FIXED;
    1            LOCAL.DEFINES
    1            DECLARE L.2 FIXED;
    1            BEGIN INNER.BLOCK.1
    2                DECLARE  I.1 FIXED;
    2                LOCAL.DEFINES
    2                DECLARE I.2 FIXED;
    1            END INNER.BLOCK.1
    0        END LOCAL.BLOCK.1
    0        BEGIN LOCAL.BLOCK.2
    1            DECLARE AA.1 FIXED;
```

The definition of L.1 preceeds LOCAL.DEFINES and may be referenced outside of LOCAL.BLOCK.1.

The definition of L.2 follows a LOCAL.DEFINES and may be referenced only within LOCAL.BLOCK.1.

The definition of I.1 follows LOCAL.DEFINES of LOCAL.BLOCK.1 and is within that block. Thus I.1 may be referenced only within LOCAL.BLOCK.1.

The definition of I.2 follows LOCAL.DEFINES of INNER.BLOCK.1 and is within that block. Thus I.2 is limited to use within INNER.BLOCK.1.

The definition of AA.1 is not within any block containing LOCAL.DEFINES. Thus AA.1 may be used anywhere in the statements that follow, even outside of LOCAL.BLOCK.2.

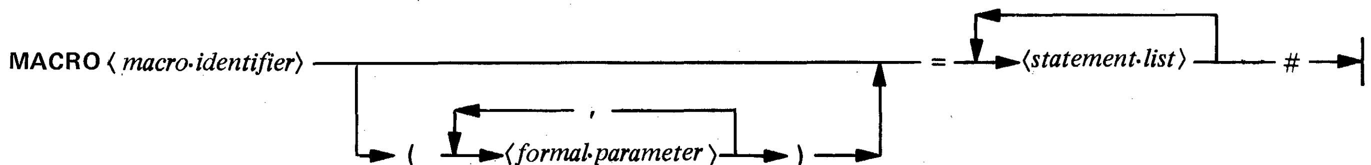
The previous example was not intended to be a model for microprogrammers. It merely demonstrates the effect of LOCAL.DEFINES. Good programming practice is to combine all global definitions at the beginning of the program and to combine all local definitions after the LOCAL.DEFINES following the BEGIN statement of the block to which they are localized.

Example

```
BEGIN LOCAL.BLOCK
    LOCAL.DEFINES
    DEFINE . . . .
    DEFINE.VALUE . . .
    DECLARE . . .
    MACRO . . .
    % MIL STATEMENTS FOLLOW
    . . .
END LOCAL.BLOCK
```

MACRO DECLARATION

Syntax



Semantics

This declaration assigns a name, the ⟨*macro.identifier*⟩, to a ⟨*statement.list*⟩ and declares any ⟨*formal.parameter*⟩*s* that is used in the ⟨*statement.list*⟩. Any subsequent reference (see MACRO reference) to ⟨*macro.identifier*⟩ will be replaced in-line by the ⟨*statement.list*⟩ and any ⟨*formal.parameters*⟩ used in these statements will be replaced by the ⟨*actual.parameters*⟩ used in the MACRO reference.

The ⟨*macro.identifier*⟩ and the ⟨*formal.parameter*⟩ list must be contained on one line, and this line must be terminated by an equal sign (=). The macro statement list must then follow, beginning on the next line, with one statement per line.

A MACRO declaration must be terminated by a pound sign (#), either at the end of the last statement or in columns 6 through 72 of the following line. For this reason, a statement within a MACRO declaration must not contain a pound sign that is not a part of a ⟨*character.string*⟩.

The ⟨*formal.parameter*⟩ list must be enclosed in parentheses. A ⟨*formal.parameter*⟩ must be a ⟨*simple.identifier*⟩. If there is more than one ⟨*formal.parameter*⟩, they must be separated by commas.

RESTRICTIONS

1. A MACRO must not reference itself although it may reference another MACRO. The maximum level to which MACROs may be nested is 10.

2. A MACRO may have a maximum of 7 ⟨*formal.parameter*⟩s.

3. A MACRO may have a maximum of 100 statement lines (records) in its ⟨*statement.list*⟩.

4. A MIL program may have a maximum of 100 MACRO declarations.

PROGRAMMING NOTE

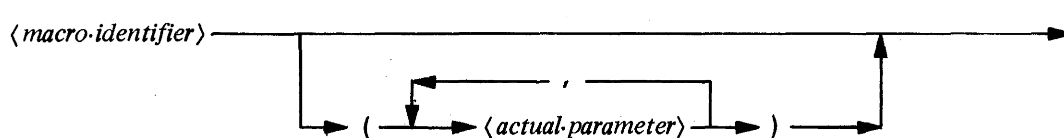
A MACRO is often used as a single statement following an IF statement. If the MACRO declaration statement list consists of more than one statement and the statement list is not bounded by a BEGIN/END pair, then a branch will be made around ONLY the first statement when the IF condition tests false. Whenever an entire MACRO ⟨*statement.list*⟩ could conceivably be used as either the THEN or ELSE part of an IF statement, the first statement in the ⟨*statement.list*⟩ should be BEGIN and the last statement should be END.

Examples

```
MACRO   EXCHANGE (DESC.1, DESC.2) =
        BEGIN
              LOAD F FROM DESC.1
.LOOP
              SWAP.THE.F.FIELD.WITH (DESC.2)
              IF FL NEQ 0 THEN GO TO -LOOP
        END#

MACRO SWAP.THE.F.FIELD.WITH (FIELD) =
        BEGIN
              BIAS BY F
              READ TO X
              XCH FIELD F FIELD
              SWAP WITH X
              COUNT FA UP AND FL DOWN BY CPL
              XCH FIELD F FIELD
              WRITE FROM X INC FA AND DEC FL
        END#

% The above MACROs could be referenced as follows:

     IF X   Y THEN
        EXCHANGE (FIELD.A, FIELD.B)
```

MACRO REFERENCE

Syntax



Semantics

A MACRO reference is replaced in-line by the statements in the ⟨*statement.list*⟩ associated with the MACRO declaration of ⟨*macro.identifier*⟩ and the ⟨*actual.parameter*⟩s replace the occurrences of ⟨*formal. parameter*⟩s used in these statements (see MACRO declaration).

There must be a one-to-one correspondence between formal and actual parameters, i.e., no ⟨*actual.parameter*⟩ may be omitted or left empty (blank), and the first ⟨*actual.parameter*⟩ replaces the first formal. parameter declared, etc.

⟨*Actual.parameter*⟩s may be ⟨*identifier*⟩, ⟨*literal*⟩s, ⟨*string*⟩s, reserved words, single line MIL statements or portions of statements. In short, they may be almost anything, with the following exceptions:

RESTRICTIONS

⟨*Actual.parameter*⟩s may not be or contain:

1. A comma, %, or unpaired parenthesis, unless contained in a ⟨*character.string*⟩.
2. An unpaired quotation mark (").
3. A ⟨*label*⟩, unless preceeded, as a part of the ⟨*actual.parameter*⟩, by a non-label token.

Example

```
MACRO GET.TABLE.DATA (TABLE.ADDRESS,ELEMENT.LENGTH, ELEMENT.IX, REG) =
      BEGIN
            MOVE ELEMENT.LENGTH TO X
            MOVE ELEMENT.IX TO Y
            CALL SET.SCRATCH.TO.X.TIMES.Y
            MOVE TABLE.ADDRESS TO FA
            ADD INTERP.MAIN.MEMORY.BASE TO FA
            ADD SCRATCH TO FA
            READ ELEMENT.LENGTH BITS TO REG
      END#
```

% Which could be referenced as:

GET.TABLE.DATA (ADDRESS (TABLE.A), 24, L, X)

Note that TABLE.A is a ⟨*label*⟩ and therefore could not be used alone as an ⟨*actual parameter*⟩.

MAKE.SEGMENT.TABLE.ENTRY

Syntax

```
MAKE.SEGMENT.TABLE.ENTRY ──┬─────────────────────────┬──►|
                           └──► VALUE ⟨literal⟩ ──────┘
```

Semantics

This instruction causes an entry to be written into the segment name-to-number correspondence table which is made a part of the final code file. This table, together with the segment dictionaries, can be used for load time binding.

The format of each entry in the correspondence table is as follows:

Level	Name	Type
1	CORRESPONDENCE.TABLE.ENTRY	BIT (44)
2	CORR.SEGMENT.NAME	CHARACTER (10),
2	CORR.EXTERNAL.SEGMENT.NUMBER	BIT (8),
2	CORR.INTERNAL.SEGMENT.NUMBER	BIT (8),
2	CORR.VALUE	BIT (8),
2	CORR.INTERNAL.SEGMENT.FLAG	BIT (1),
2	FILLER	BIT (39);

CORR.VALUE is set to zero if the "VALUE ⟨*literal*⟩" clause is not present. If present, it must not exceed 255.

CORR.ESTERNAL.SEGMENT.NUMBER is always set to the number of the current external (CODE. SEGMENT) segment. If no CODE.SEGMENT statement has been compiled at a given point, the value is zero.

If this statement is used immediately following a SEGMENT statement then the remaining fields are set as follows:

> CORR.SEGMENT.NAME is set to the name of the internal segment with right truncation or blank fill.
> CORR.INTERNAL.SEGMENT.NUMBER is set to the number of the internal segment.
> CORR.INTERNAL.SEGMENT.FLAG is set to 1.

If this statement is used immediately following a CODE.SEGMENT statement, or prior to the occurrence of either any CODE.SEGMENT or SEGMENT statement then the remaining fields are set as follows:

> CORR.SEGMENT.NAME is set to the name of the external (CODE.SEGMENT) segment, with right truncation or blank fill. This field will be blank if neither a CODE.SEGMENT nor a SEGMENT statement has yet occurred.
> CORR.INTERNAL.SEGMENT.NUMBER is set to zero, as is CORR.INTERNAL.SEGMENT.FLAG.

This statement can be used more than once within a segment if more than an 8-bit value is required.

MICRO

Syntax

MICRO ——► ⟨*literal*⟩ ——►|

Semantics

This instruction places a 16-bit constant in line. The programmer is responsible for providing any protection that may be needed to prevent a MICRO from executing; therefore, this instruction should be used with caution.

The ⟨ *literal*⟩ has a decimal range from 0 to 65535.

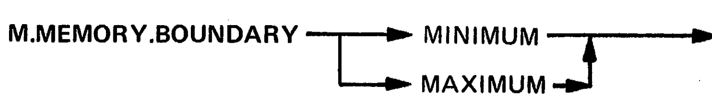
Examples

```
MICRO @83AA@      % THIS IS EQUIVALENT TO "MOVE @AA@ TO L"
MICRO "22"        % THIS IS EQUIVALENT TO @F2F2@
MICRO "HI"        % "HI" = @C8C9@
MICRO 784         % = @0310@ = "CLEAR X"
```

M.MEMORY.BOUNDARY

(Available on B 1720 systems only)

Syntax



Semantics

This instruction sets the M.MEMORY boundary fields within the IPB (Interpreter Parameter Block) of a MIL program to the current code address.

MINIMUM specifies to the operating system the number of micro-instructions that must be loaded into M-Memory before the micro-program may be given control. If, however, this value exceeds the amount of M-Memory physically present on a given system, the value will be ignored (considered = 0). The statement is generally used to ensure that the most used microcode will execute from control memory, where it executes faster than if it is executed from main memory.

MAXIMUM specifies the maximum M-Memory utilization of a micro-program. No code emitted after the occurrence of this statement will ever be loaded into, and hence executed from, M-Memory. It is generally used to keep non-executable data, such as TABLEs, from being loaded into control memory, thus being made inaccessible in main memory.

Thus at all times the portion of microcode in M-Memory will be, at the discretion of the operating system, from the beginning of a given microprogram until some point between the appearance of the M.MEMORY. BOUNDARY MINIMUM and the M.MEMORY.BOUNDARY MAXIMUM statements. The fields are ignored for stand-alone microprograms.

MONITOR

Syntax

MONITOR ——► <*literal*> ——►|

Semantics

This instruction emits the monitor micro-operator with the <*literal*> occurrence identifier. (See also Appendix B: MONITOR.)

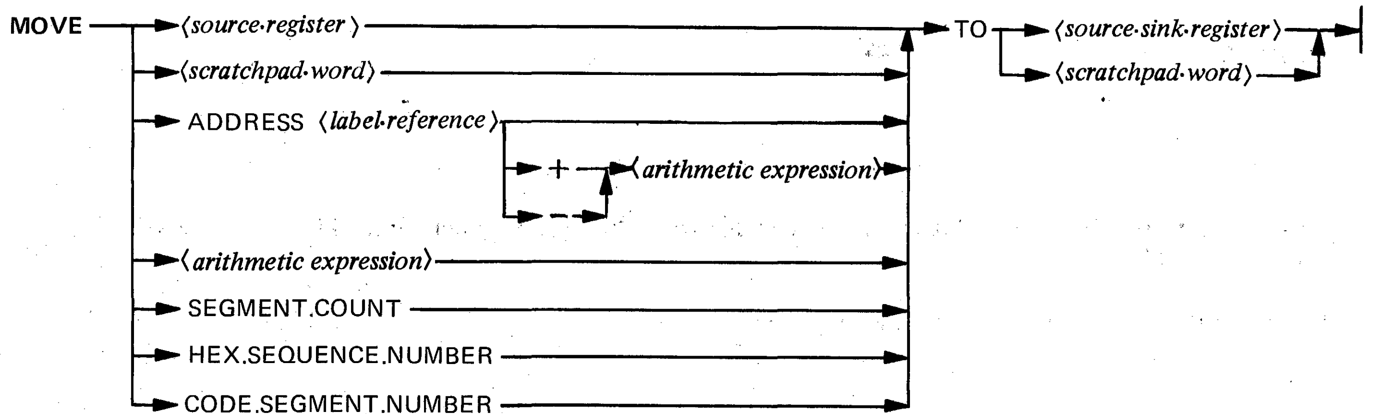
The <*literal*> has a decimal range from 0 to 255.

Example

MONITOR 5

MOVE

Syntax



Semantics

This instruction copies the specified information into a ⟨*source.sink.register*⟩ or scratchpad word.

ADDRESS (⟨*label.reference*⟩) is a literal value equal to the code address of the label reference.

SEGMENT.COUNT is a literal value equal to the number of times a Segment statement has occurred.

HEX.SEQUENCE.NUMBER is a literal value equal to the leftmost six digits of the current source statement sequence number.

CODE.SEGMENT.NUMBER is a literal value equal to the current CODE.SEGMENT number.

The following are restrictions on an S-Memory Processor.

a. If ADDRESS or ⟨*arith.expression*⟩ has a value greater than 255, and ⟨*source.sink.register*⟩ is CP, the move wll not take place.

b. If ⟨*source.register*⟩ is U, ⟨*source. sink.register*⟩ may not be TAS, M, or A.

c. If ⟨*source. register*⟩ is A, CP, M, or DATA, ⟨*source.sink.register*⟩ may not be a 4-bit register.

d. If ⟨*source.register*⟩ is SUM or DIFF, ⟨*source.sink.register*⟩ may not be CMND or DATA.

The following are restrictions on both an S-Memory and M-Memory Processor.

a. When ⟨*source.register*⟩ is DATA, ⟨*source.sink.register*⟩ may not be DATA or CMND.

b. When ⟨*source.sink.register*⟩ is M, the operation is changed to a BIT-OR which modifies the next micro-operation; it does not modify the instructions stored in memory. In tape mode no BIT-OR takes place. A literal value generated from ADDRESS, ⟨*arith.expression*⟩, or SEGMENT.COUNT may not be moved to the M register.

Examples

```
MOVE X TO Y
MOVE 48 TO S1A
MOVE ADDRESS (+ GLOP) TO T
MOVE 10 TO TA
MOVE S12A TO S10B
MOVE ADDRESS (BLAH) +16  * 8 - 1 TO FA
MOVE SEGMENT.COUNT TO T
MOVE (81+(3*10)-1)/2 TO Y
```

NOP

Syntax

Semantics

This NO OPERATION instruction does nothing except use one clock cycle and take up one word of control or main memory.

NORMALIZE

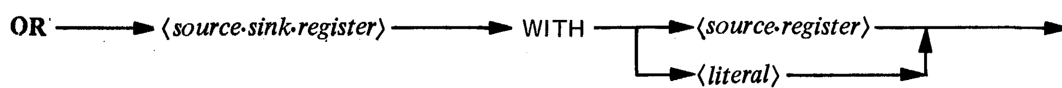
Syntax

Semantics

This instruction shifts the contents of the X register left while counting the FL register down until either the most-significant bit of X (determined by CPL) equals 1 or FL equals 0. If the most-significant bit of X is already 1, of if FL is already 0, then no shift takes place.

OR

Syntax



Semantics

This instruction is used to logically OR the contents of the 4-bit ⟨*source.sink.register*⟩ with the value of the ⟨*literal*⟩ or the contents of a 4-bit ⟨*source.register*⟩. The result is placed in ⟨*source.sink.register*⟩; the contents of ⟨*source.register*⟩ remain unchanged. (See also: AND and EOR.)

The ⟨*register*⟩s may be any of the following:

⟨*source.sink.register*⟩

CA CB *CC *CD
FT FU
FLC FLD FLE FLF
LA LB LC LD LE LF
TA TB TC TD TE TF
TOPM (available on B1720 only)

⟨*source.register*⟩

⟨*source.sink.register*⟩
BICN
FLCN
INCN (available on B1720 only)
PERR (available on B1720 only)
XYCN
XYST

*CC and CD represent processor interrupts and flags.

The ⟨*literal*⟩ has a decimal range from 0 to 15.

TABLE 8-3 OR Truth Table

Source. Sink .Register	⟨*literal*⟩ ⟨*Source.Register*⟩			⟨*Source.Sink.Register*⟩
0	OR	0	Yields	0
1	OR	0	Yields	1
0	OR	1	Yields	1
1	OR	1	Yields	1

Example

OR TB WITH 3

	TA	TB	TC	TD	TE	TF	
T	0000	0101	1111	0011	0001	0010	before (05F312)
	--	**0011**	--	--	--	--	literal **(3)**
T	0000	0111	1111	0011	0001	0010	after (07F312)

OVERLAY

(Available on B 1720 systems only)

Syntax

OVERLAY ——————►|

Semantics

This instruction overlays control memory from main memory. Before an overlay is initiated the L register must contain the first control memory overlay address, the FA register must contain the beginning main memory address, and the FL register must contain the length in bits to be overlayed. Overlay will continue until the FL register equals 0 or the A register is out of bounds. If the A register goes out of bounds, FA contains the address of the next micro-instruction in main memory; FL contains the length in bits of unfetched data.

The action of the hardware executing this instruction is as follows:

```
        MOVE A TO TAS
        MOVE L TO A
.LOOP
        READ 16 BITS TO L INC FA AND DEC FL
        MOVE L TO CONTROL MEMORY ADDRESSED BY A
        INC A
        IF FL NEQ 0 AND A NOT OUT OF BOUNDS THEN GO TO -LOOP
```

PAGE

Syntax

Semantics

This instruction causes the source listing to skip to the top of a new page at compile time. Code is not generated.

POINT

Syntax

POINT FA TO ——→ <*arithmetic.expression*> ——→|

Semantics

This pseudo-operation causes the compiler to generate an instruction that adjusts the value of FA to the value of the <*arithmetic.expression*>.

Prior to the execution of this instruction, the compiler must have been given some knowledge of the contents of FA. This can be done via:

MOVE <*arithmetic.expression*> TO FA

or

FA.POINTS TO <*arithmetic.expression*>

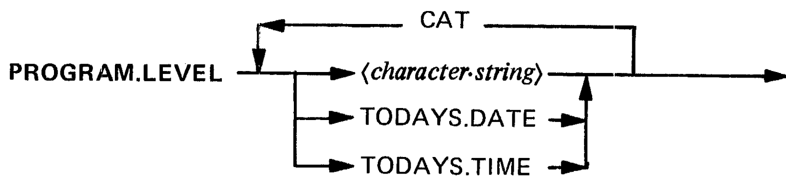
FA will be adjusted by up to 144 bits as a result of this command. (A warning message will result if the adjustment is greater than 72 bits). (See also: READ and WRITE.)

Example

```
DECLARE
        01    STRUCTURE,
                02    DATA.A BIT(10),
                02    DATA.B CHARACTER(20),
                02    DATA.C FIXED;

FA.POINTS TO DATA.A
READ DATA.LENGTH (DATA.A) BITS TO X POINT FA TO DATA.B
POINT FA TO STRUCTURE
MOVE DATA.C TO FA
WRITE DATA.LENGTH (DATA.C) BITS FROM Y POINT FA TO DATA.B
```

PROGRAM.LEVEL

Syntax



Semantics

This instruction places forty characters of information into the PROGRAM.LEVEL location of the IPB (Interpreter Parameter Block).

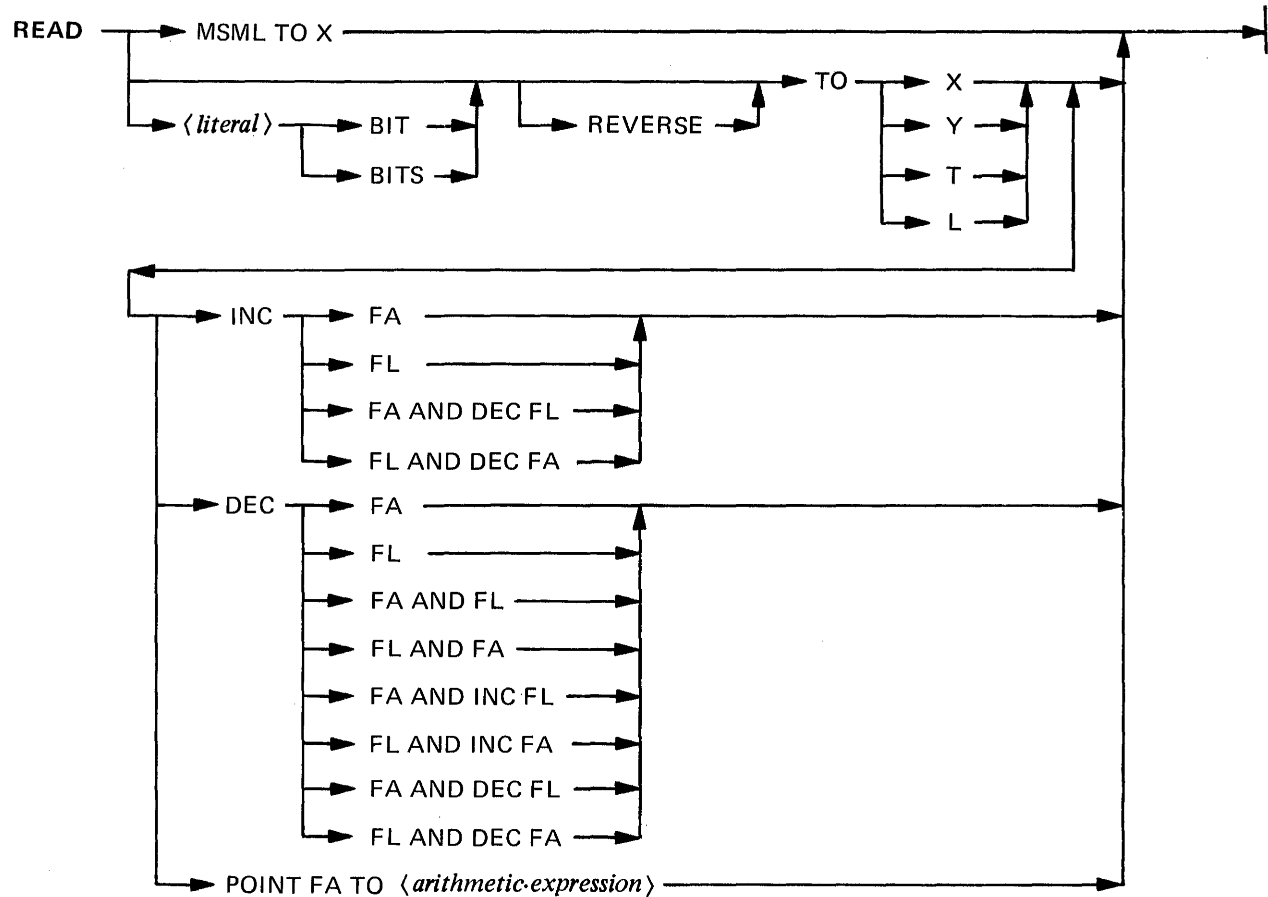
If the TITLE statement is unused, the title headings of the program listing will reflect the PROGRAM. LEVEL information.

Example

PROGRAM.LEVEL "THIS IS A SUBHEADING " CAT TODAYS.TIME

READ

Syntax



Semantics

An M-Memory READ MSML TO X instruction reads to the X register the 16 bits in M-Memory pointed to by the contents of the L register. The contents of L must be modulo 16. This facility is not available on S-Memory Processors.

An S-Memory READ instruction reads from 0 to 24 bits of information from S-Memory into one of the allowable sink (destination) registers: X, Y, T, or L.

If the ⟨*literal*⟩ is zero or is not specified, the field length is given by the contents of CPL. The read data will be right justified in the selected sink register. If the field length is zero then X, Y, T, or L will be set to zero.

Normally, on an S-Memory read, the contents of the FA register point to the first bit of the field to be read. If the REVERSE option is used, the contents of the FA register point to the last bit + 1 of the field to be read. The sink register receives the contents of this field as if it has been read in a forward direction.

INC/DEC adjusts FA/FL by the field length after the operation but in the same micro-instruction.

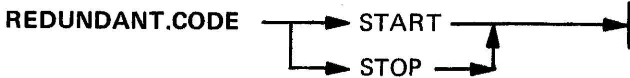
POINT FA adjusts FA by up to 144 + field length bits after the operation. (A warning message will be issued if the adjustment is greater than 72 + field length bits). The POINT FA option can be used only if ⟨*literal*⟩ BIT(S) is specified and is greater than 0. (See also: FA.POINTS and POINT.)

Examples

READ MSML TO X
READ 24 BITS TO X
READ TO Y INC FA
READ 2 BITS REVERSE TO T DEC FA AND FL
READ REVERSE TO L INC FL
READ 10 BITS TO T POINT FA TO 100

REDUNDANT.CODE

Syntax



Semantics

This REDUNDANT.CODE START pseudo-instruction causes the compiler to emit each micro twice until the occurrence of the REDUNDANT.CODE STOP pseudo-instruction. It can be used to help ensure the correct loading of a program or data from cassette.

RESERVE.SPACE

Syntax

RESERVE.SPACE FOR ——→ <*arithmetic. expression*> ——→ BITS ——→|

Semantics

This instruction causes the compiler to emit a sufficient number of NOP's (@0000@) to allow for the number of bits specified by <*arithmetic.expression*>.

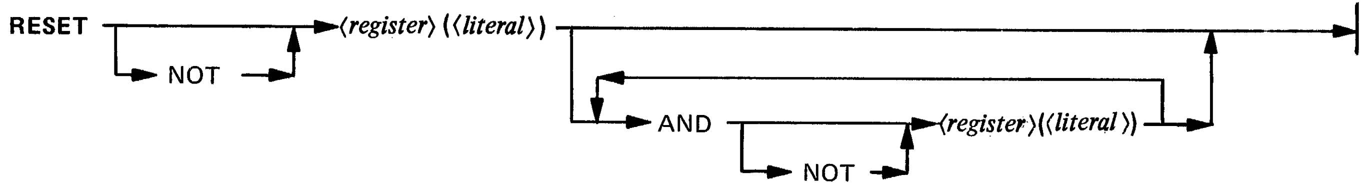
The actual amount of space reserved will always be MOD 16; therefore up to 15 bits more than that specified by the <*arithmetic.expression*> may be reserved.

Example

```
    DECLARE IO.DESCRIPTOR BIT(188);

DESC.LOCN
    RESERVE.SPACE FOR DATA.LENGTH(IO.DESCRIPTOR) BITS
```

RESET

Syntax



Semantics

This instruction RESETs (sets to zero) the bit specified by the ⟨*literal*⟩ into the register. By using the options, more than one bit in any one register can be reset with the same instruction IF ALL BITS ARE IN THE SAME 4-BIT REGISTER. (See also: COMPLEMENT and SET.)

The register may be any 4-bit source and sink (destination) register below:

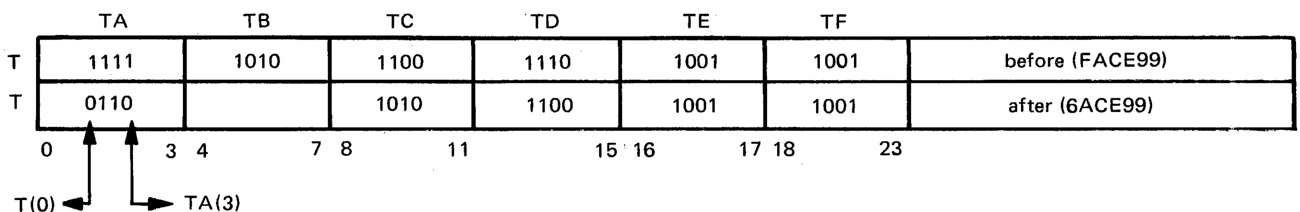
CA CB CC CD (CC and CD represent processor interrupts and flags)
FT FU
FLC FLD FLE FLF
LA LB LC LD LE LF
TA TB TC TD TE TF
TOPM (available on B 1720 only)

It may also be the FL, FB, L, or T register: all bits must then be in the same 4-bit subfield.

The ⟨*literal*⟩ has a decimal range from 0 to 3 for a 4-bit register; from 0 to 15 for the FL register; and from 0 to 23 for the FB, L, and T registers.

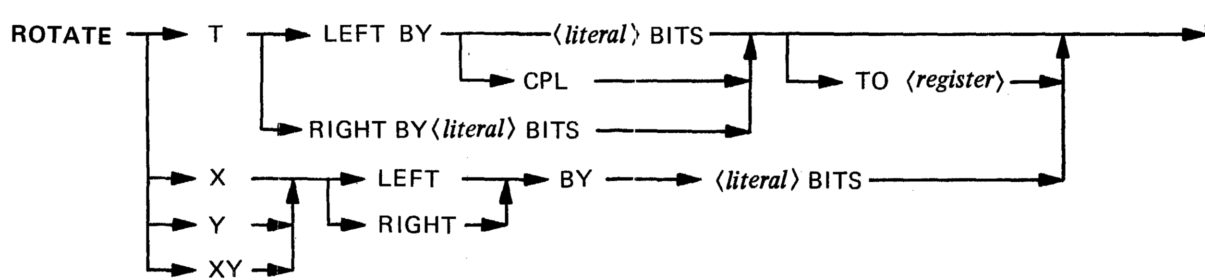
Example

RESET T(0) AND TA(3)



ROTATE

Syntax

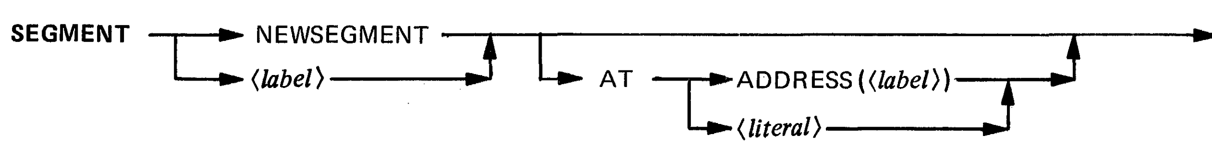


Semantics

See SHIFT/ROTATE T and SHIFT/ROTATE X/Y/XY

SEGMENT

Syntax



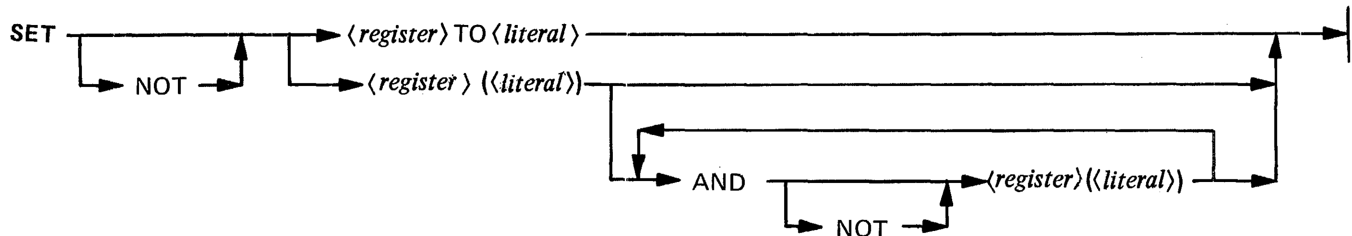
NOTE: The <*literal*> must be MOD 16.

Semantics

See: Section 5 (SEGMENTATION).

SET

Syntax



Semantics

This instruction SETs the ⟨*register*⟩ to the value of the ⟨*literal*⟩ or SETs (bit=one) the bit specified by the ⟨*literal*⟩ into the ⟨*register*⟩. By using the options more than one bit in any one register can be set with the same instruction IF ALL BITS ARE IN THE SAME 4-BIT REGISTER. (See also: COMPLEMENT and RESET.)

SET ⟨*register*⟩ TO ⟨*literal*⟩: The ⟨*register*⟩ may by any 4-bit source and sink (destination) register listed below.

> CA CB CC CD (CC and CD represent processor interrupts and flags)
> FT FU
> FLC FLD FLE FLF
> LA LB LC LD LE LF
> TA TB TC TD TE TF
> TOPM (available on B 1720 systems only)

It may also be the CPU register. If CPU is used, the ⟨*literal*⟩ has a decimal range from 0 to 3; otherwise the ⟨*literal*⟩ has a range from 0 to 15.

SET ⟨*register*⟩ (⟨*literal*⟩): The ⟨*register*⟩ may be any 4-bit source and sink register listed above. It may be the FL, FB, L, or T register: all bits must then be in the same 4-bit subfield. The ⟨*literal*⟩ has a decimal range from 0 to 3 for a 4-bit register; from 0 to 15 for the FL register; and from 0 to 23 for the FB, L, and T registers.

Examples

SET TA TO 3

	TA	TB	TC	TD	TE	TF	
T	1111	0100	0101	0110	0111	1000	before (F45678)
T	0011	0100	0101	0110	0111	1000	after (345678)

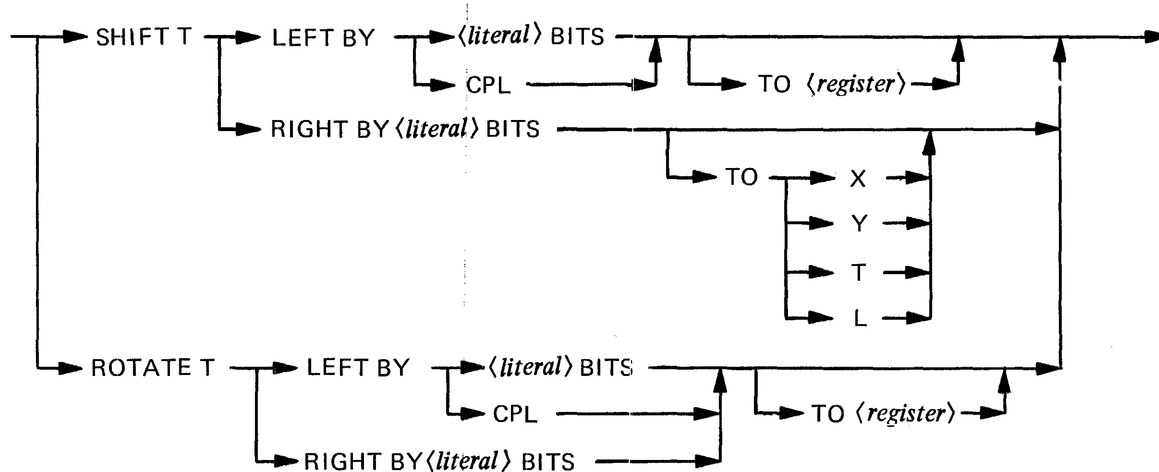
SET TC(2) AND T(11)

	TA	TB	TC	TD	TE	TF	
T	0001	0010	0000	0100	0101	0110	before (120456)
T	1001	0010	0011	0100	0101	0110	after (923456)

TC(2) T(11)

SHIFT/ROTATE T

Syntax



Semantics

This instruction SHIFTs or ROTATEs the contents of the T register and places the result either in T or in some other source and sink (destination) register. If the result is not placed in the T register, T remains unchanged. SHIFT will zero fill.

The ⟨ *literal*⟩ has a decimal range from 0 to 24.

SHIFT/ROTATE T LEFT: If 0 or CPL is used, a shift or rotation by the value of the CPL register will occur. If CPL is greater than 24, 24 is used.

TO ⟨ *register*⟩: places the shifted or rotated results in the specified source and sink register; the T register remains unchanged. If the TO register option is not used, the result is placed in the T register. The register may be any source and sink register except DATA or MBR (refer to: Registers and Scratchpad). If the ⟨ *register*⟩ is M, the result of the SHIFT/ROTATE operation is BIT-ORed into the M register and modifies the next micro-instruction.

ROTATE T RIGHT: Because the hardware can only rotate the T register to the left, the compiler converts this instruction to the proper left rotate to accomplish the same result as the rotate right.

SHIFT T RIGHT: Because the hardware can only shift the T register left, the compiler will generate an EXTRACT to accomplish the same result. Therefore, the T register may be shifted right only to the X, Y, T or L register. If the TO... option is not used, the result is placed in the T register; otherwise, the T register remains unchanged.

PROGRAMMING NOTE

It is recommended that the EXTRACT instruction be used rather than SHIFT T RIGHT.

Examples

ROTATE T LEFT BY 4 BITS

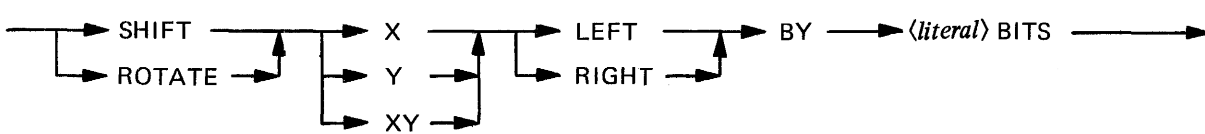
	TA	TB	TC	TD	TE	TF	
T	0110	0011	1000	0101	1111	0000	before (6385F0)
T	0011	1000	0101	1111	0000	0110	after (385F06)

SHIFT T LEFT BY 4 BITS

	TA	TB	TC	TD	TE	TF	
T	0110	0011	1000	0101	1111	0000	before (6385F0)
T	0011	1000	0101	1111	0000	0000	after (385F00)

SHIFT/ROTATE X/Y/YX

Syntax



Semantics

This instruction shifts or rotates the X, Y, or XY register (X concatenated with Y) a specified number of bits to the right or left. Zero fill will occur with the SHIFT instruction.

The ⟨ *literal*⟩ has a decimal range from 0 to 23 for the X and Y register; and from 0 to 47 for the XY register.

NOTE

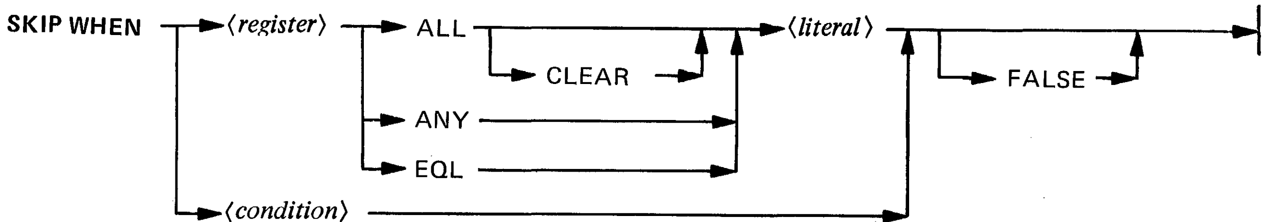
The ⟨ *literal*⟩ has a maximum value of 1 on the B 1710 systems when the concatenated XY register is specified.

Example

SHIFT X LEFT BY 5 BITS
ROTATE XY RIGHT BY 40 BITS

SKIP

Syntax



Semantics

This instruction causes one micro-instruction to be skipped if the designated ⟨*condition*⟩ is satisfied. (See also: IF.)

SKIP WHEN ⟨*registers*⟩: The ⟨*literal*⟩ contains a 4-bit mask and may be comprised of decimal, binary, or hexadecimal entries.

ALL is considered to be true only if all the bits in the ⟨*register*⟩ corresponding to one bits in the mask are true. That is, only the designated bit positions are tested to see if they contain ones. ANY is true if at least one bit in the ⟨*register*⟩ corresponding to a one bit in mask is true. EQL is true if all the ⟨*register*⟩ bits equal the corresponding bits in the mask. That is, the ⟨*register*⟩ must be exactly like the mask.

ALL CLEAR causes the masked bits of the ⟨*register*⟩ to be set to zeros after testing the ALL condition. Only the bits tested are cleared, and the clearing action always occurs whether the SKIP is taken or not. If ALL is used with a mask of 0000, the result is always false.

FALSE causes a skip when the whole ⟨*condition*⟩ is false.

SKIP WHEN condition: The ⟨*condition*⟩ may be any condition available from the condition registers. (See: IF.)

The register may be declared as follows:

FU	TA	LA	CA	BICN
FT	TB	LB	CB	FLCN
FLC	TC	LC	CC	INCN
FLD	TD	LD	CD	XYCN
FLE	TE	LE		XYST
FLF	TF	LF		

PROGRAM NOTE

The use of the IF...THEN...ELSE instruction is recommended rather than the SKIP instruction. The SKIP is limited to one, 4-bit grouping mask in one register and may only skip one micro-instruction. The IF is capable of testing any combination of bits in many registers or skipping blocks of micro-instructions and will generate a SKIP WHEN hardware micro-instruction whenever possible.

S.MEMORY.LOAD

Syntax

S.MEMORY.LOAD ——► START ——►|

Semantics

This instruction specifies the location for beginning statements in S.MEMORY. Code is not generated, but the code address of the last statement is placed in the IPB (Interpreter Parameter Block) at RESERVED. M.MEMORY.

This statement is used to specify the size of all code emitted previous to its occurrence into IPB. RESERVED.M.MEMORY in the Interpreter Parameter Block of the final code file. IPB.RESERVED.M. MEMORY can then be used by a load time binder to load previously-generated code into control memory and to make allowances for its absence in main memory.

STORE

Syntax

STORE F INTO ——► ⟨*double-scratchpad-word*⟩ ——►|

Semantics

This instruction MOVEs the Field (F) register into any double scratchpad word (S0 . . . S15); the F register remains unchanged.

NOTE

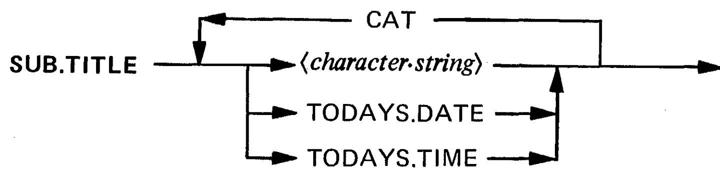
The compiler generates two MOVE instructions on B 1710 systems.

Example

STORE F INTO S6

SUB.TITLE

Syntax



Semantics

This instruction modifies program title information.

If ⟨ *character.string*⟩ exceeds 72 characters, right-hand truncation will occur.

$ HEADINGS and $ PAGE.NUMBERS must be specified if subtitles are to be listed on page headings.

Example

SUB.TITLE TODAYS.DATE CAT "PROB.A" CAT TODAYS.TIME

SUBTRACT SCRATCHPAD

Syntax

SUBTRACT ——► <*scratchpad.word*> ——► FROM FA ——►|

Semantics

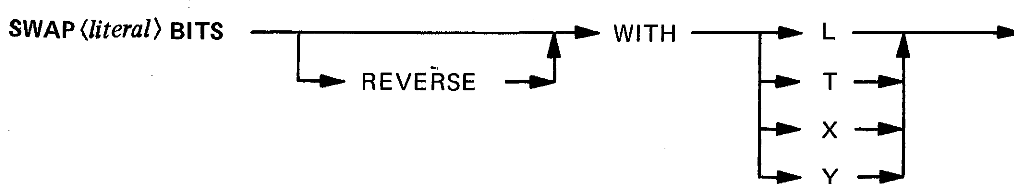
This instruction subtracts the left half of any scratchpad word (S0A . . .S15A) from the Field Address (FA) register. The result is placed in FA; the contents of <*scratchpad.word*> remain unchanged. (See also: ADD SCRATCHPAD.)

Example

SUBTRACT S3A FROM FA

SWAP

Syntax



Semantics

This instruction swaps the specified number of bits between main memory and the specified register.

The FA (Field Address) register must have been previously set to the proper main memory address.

The ⟨*literal*⟩ has a decimal range from 0 to 24. If the value of ⟨*literal*⟩ is zero, the contents of the CPL register are used. If the CPL register is also 0, the register is cleared to all zeros. If less than 24 bits are swapped, the leading bits of the register are set to zero.

Normally the contents of the FA register point to the first bit of the field to be swapped. If the REVERSE option is used, the contents of FA point to the last bit + 1 of the main memory field involved. The specified register (L, T, X or Y) receives the contents of this field as if it has been read in a forward direction.

For the B 1710 ($Subset specified) the compiler emits the following code:

```
MOVE T TO TAS
READ ⟨literal⟩ BITS (REVERSE) TO T
WRITE ⟨literal⟩ BITS (REVERSE) FROM ⟨register⟩
MOVE T TO ⟨register⟩
MOVE TAS TO T
```

PROGRAMMING NOTE

Incrementing or decrementing of the FA or FL registers is not allowed with the SWAP instruction.

TABLE

Syntax

```
                       ┌──────────────────────────────┐
TABLE ⟨label⟩ ──► BEGIN ┴──┬──► ⟨character.string⟩ ──┬──┴──► END ──►|
                           └──► ⟨hex.string⟩ ─────────┘
```

Semantics

This instruction creates in-line character strings.

Any number of strings are allowed per line, but a string cannot cross a line boundary. The⟨hex.string⟩must be enclosed within @ signs.

The BEGIN/END pair must surround all strings in the TABLE. The characters are grouped two per address, i.e., 16 bits.

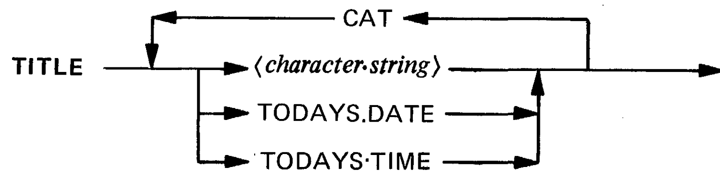
The ⟨ *label*⟩ of the table must be unique; its use references the first 16 bits of the table.

Example

TABLE REF	Code generated:
BEGIN	
"AB"	C1C2
@ABC@	C1C2
"D"	C3C4
"45"	F4F5
END	
MOVE ADDRESS (REF) TO Y	The address of the table (REF) will be loaded into the Y register

TITLE

Syntax



Semantics

This instruction modifies program title information.

If ⟨*character.string*⟩ exceeds 72 characters, right-hand truncation will occur.

$ HEADINGS and $ PAGE.NUMBERS must be specified if titles are required on following pages.

Example

TITLE TODAYS.DATE CAT "PATCHES"

TRANSFER.CONTROL

Syntax

Semantics

This instruction generates the Transfer.Control micro-instruction. (See Appendix B: Transfer.Control). On the B 1710 series it acts as a NOP.

When it is necessary to transfer control from one firmware process to another, the A, MBR, and TOPM registers may all need to be changed. Changing any one of these registers will cause a transfer of control to some micro other than the next micro in line. Consequently some means of changing all three of these registers simultaneously is required; this will be accomplished with the Transfer.Control instruction.

The action of the B 1720 hardware is as follows:

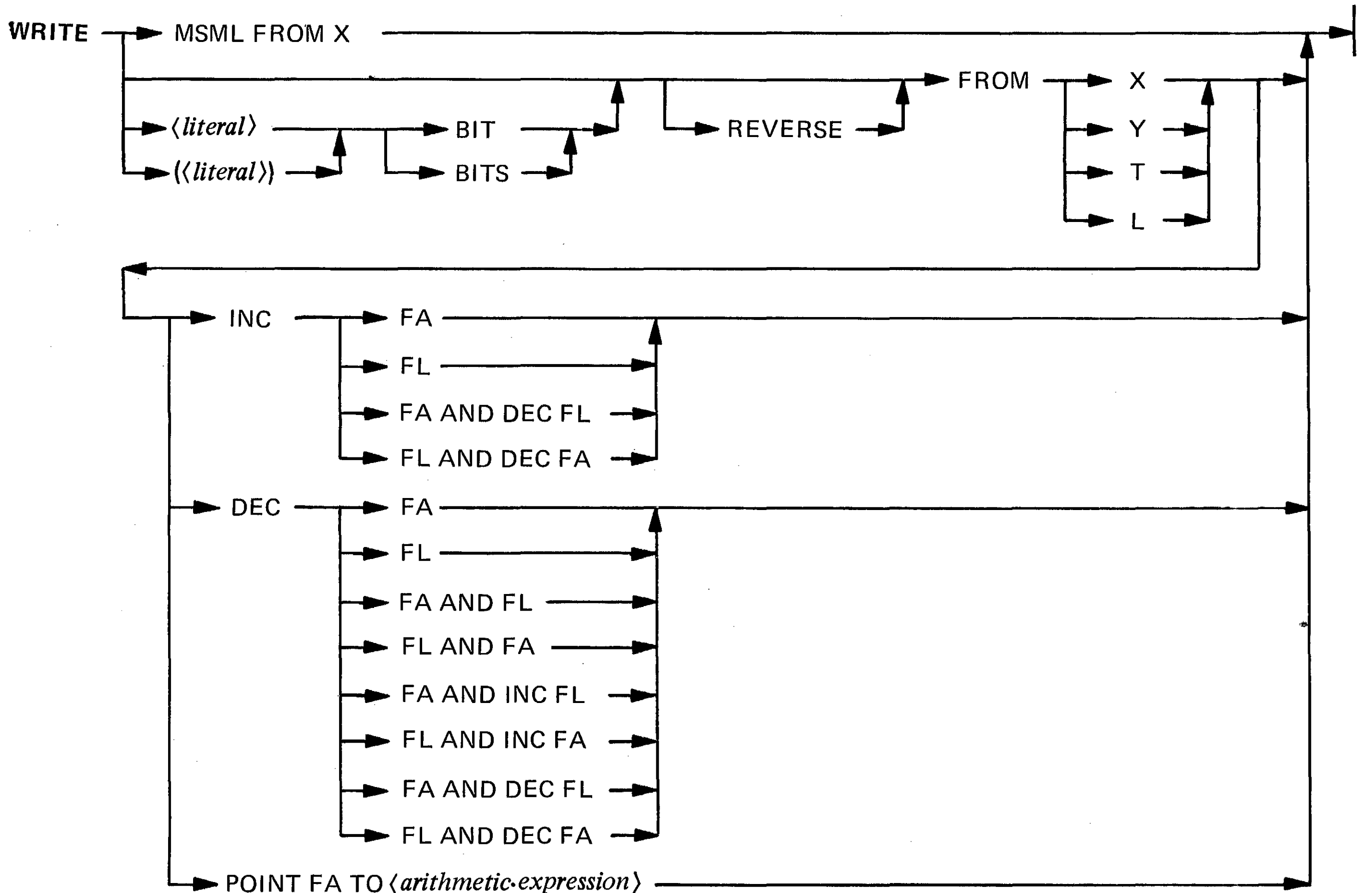
```
MOVE L TO MBR
MOVE TF TO TOPM
MOVE T(6) THRU T(19) TO A
```

Example

```
MOVE ADDRESS.OF.GISMO.IN.S.MEMORY TO L
MOVE GISMO.EVENT.ADDRESS TO T
MOVE 0 TO TF
TRANSFER.CONTROL % OFF WE GO . . .
```

WRITE

Syntax



Semantics

An M-Memory WRITE (MSML TO X) instruction writes from the X register the 16 bits in M-Memory pointed to by the contents of the L register. The contents of L must be modulo 16. This facility is not available on S-Memory Processors.

An S-Memory WRITE instruction writes from 0 to 24 bits of information into S-Memory from one of the allowable source registers: X, Y, T, or L.

The amount of data written (field length) is determined by the ⟨*literal*⟩/(⟨*literal*⟩) BIT(S) option. If this is equal to 0 or is empty, then the field length is given by the contents of CPL is right justified in the selected source register. If the field length is zero then nothing is written.

Normally the contents of the FA register point to the first bit of the field to be written. If the REVERSE option is used, the contents of the FA register point to the last bit + 1 of the field to be written to memory. Memory contains the rightmost contents of the source register as if it had been written in a forward direction.

INC/DEC adjusts FA/FL by the field length after the operation but in the same micro-instruction.

POINT FA adjusts FA by up to 144 + field length bits after the operation. (A warning message will be issued if the adjustment is greater than 72 + field length bits). This option can be used only if ⟨*literal*⟩/ (⟨*literal*⟩) BIT(S) is specified and is greater than 0. (See also: FA.POINTS and POINT.)

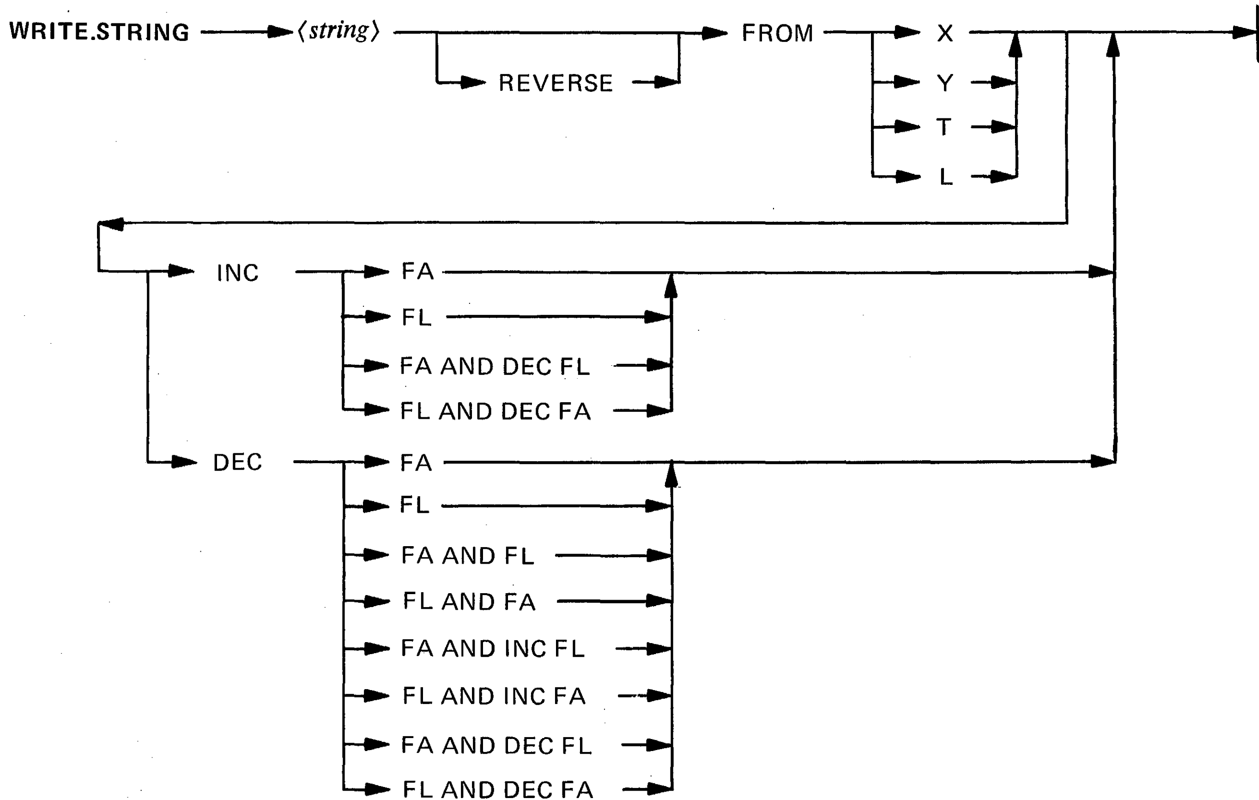
The unparenthesized ⟨*literal*⟩ has a decimal range from 0 to 24. (⟨*Literal*⟩) has a decimal range from 0 to 26: a value of 25 will cause 24 bits to be written with correct parity; a value of 26 will cause 24 bits to be written with incorrect parity.

Examples

WRITE MSML FROM X
WRITE 24 BITS FROM X
WRITE FROM Y INC FA
WRITE 2 BITS REVERSE FROM T DEC FA AND DEC FL
WRITE REVERSE FROM L DEC FL
WRITE 10 BITS FROM T POINT FA TO 25

WRITE.STRING

Syntax



Semantics

This instruction generates the necessary in-line ⟨*literal*⟩s for a ⟨*string*⟩, with moves to the indicated register. It also generates the WRITE commands to write the ⟨*string*⟩ into main memory, beginning at the address in the FA register. If ⟨*literal*⟩ exceeds 24 bits, then an INC or DEC on FA is required.

The length of the ⟨*string*⟩ is limited to the remainder of the source card image. It may be any of the data types shown in table 8-4.

Table 8-4. String Definitions

Data Type	Start-Stop Symbol	Length of Each Unit	Example
Character	-	8 bits	"APC128JKL"
Hex	@	4 bits	@124ADF@
Octal	@(3)	3 bits	@(3)123567@
Quartal	@(2)	2 bits	@(2)123321@
Unary	@(1)	1 bit	@(1)11001101@

Examples

```
WRITE.STRING "APC" REVERSE FROM X
WRITE.STRING "THIS PUTS A LITERAL INTO MEMORY" FROM T INC FA
```

XCH

Syntax

XCH ———→ ⟨*double-scratchpad-word.1*⟩ ——→ F ———→ ⟨*double-scratchpad-word.2*⟩ ———→|

Semantics

This instruction moves the Field (F) register to double scratchpad word.2 (S0 . . . S15); double scratchpad word.1 (S0 . . . S15) is then moved to the F register. The two words may be the same, causing data to be swapped between F and a double scratchpad word.

Example

```
XCH SO F SO          % equivalent to:         MOVE FA TO S0A
                     %                        MOVE FL TO S0B
                     %
                     % and simultaneously:    MOVE S0A to FA
                     %                        MOVE S0B TO FL
```

9. PROGRAMMING TECHNIQUES

VIRTUAL-LANGUAGE DEFINITIONS

A set of virtual-instructions for the virtual machine must first be defined as each being a unique string of bits. This definition may be chosen according to any relevant criteria. For example, COBOL verbs may be encoded according to their frequency of usage, the higher frequency verbs being encoded in three bits with one escape code that specifies the next eight bits as an extended code string. Another approach might be to accept directly the source language as in a time-sharing, "Line-at-a-time," interactive mode. After the S-instructions and their operand fields have been defined, any standard location or technique should be selected. For example, the base values of S-instructions and S-data might be in S4A and S5A of the scratchpad; or all routines are to be referenced with CALL and end with an EXIT instruction to facilitate subroutine usage. The microprogrammer is now ready to begin creating the microroutines needed to perform each of the events in the S-language.

SOURCE IMAGE FORMAT

The compiler accepts card images consisting of one symbolic micro-instruction per card. The source program must reflect the following format:

Column	Usage
1-5	Reserved for ⟨*label*⟩ declarations which, if used, must begin somewhere within this field. Both ⟨*point.label*⟩s and ⟨*unique.label*⟩s are allowed, with a limit of 63 characters and no embedded blanks. A blank is the separator between the label and the beginning of the micro-instruction. ⟨*Unique.label*⟩s must be unique within the first 23 characters; the remainder is considered documentation. ⟨*Point.label*⟩s must be unique within the first 25 characters.
1-72	A percent sign (%) anywhere within this field indicates that the remainder of the card image is to be ignored.
6-72	MIL statements may appear anywhere within this field. At least one blank must be used between words except in those cases where a ⟨*special.character*⟩ (e.g., a parenthesis or relational operator) is used, in which case blanks are optional.
73-80	Reserved for sequence numbers.
81-90	Reserved for patch information.

Source code maintenance as well as other compiler options may be specified by the use of either a $ (dollar sign) or & (ampersand) in column 1. (See Appendix A: MIL Compiler Operation.)

PROGRAM EXAMPLE

The following pages provide an example of a MIL program.

```
                                   BURROUGHS B1700 MIL COMPILER, MARK 6.0(06/08/76 12:25)
MODEL                                                                                      THURSDAY, DECEMBER 16, 1976, 02:41 PM.

NL BLOCK      CODE   MEMORY                                                                                        SEQUENCE   PATCH
    NAME             ADDRESS                                                                                                  INFO

0                          :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%: 00000200
0                          :%                        EXAMPLE  OF A MIL PROGRAM                    %: 00000300
0                          :%    THIS PROGRAM DEMONSTRATES SOME DESIRABLE CHARACTERISTICS OF A     %: 00000400
0                          :%  MIL PROGRAM.  THE PROGRAM "INTERPRETS" ONE BIT OP-CODE INSTRUCTIONS %: 00000500
0                          :%  FROM A FIELD IN MEMORY AND PERFORMS THE INDICATED OPERATION.  SINCE %: 00000600
0                          :%  THERE ARE ONLY TWO VALUES OF A ONE BIT OP-CODE, THERE ARE ONLY TWO  %: 00000700
0                          :%  INSTRUCTIONS TO DECODE.  (IN THIS CASE "ADD" AND "END.OF.JOB" ARE   %: 00000800
0                          :%  THE ONLY INSTRUCTIONS.)                                             %: 00000900
0                          :%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%: 00001000
0                          :%                                                                     : 00001100
0                          :%                                                                     : 00001200
0                          :%                                                                     : 00001300
0                          :     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%       : 00001400
0                          :     %%%%%%%%%%          GLOBAL DECLARES             %%%%%%%%%%       : 00001500
0                          :     %%%%%%%%%%                                      %%%%%%%%%%       : 00001600
0                          :     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%       : 00001700
0                          :       DECLARE                                                         : 00001800
0                 [000000] :                01 SYSTEM.INFORMATION            BIT(1668),            : 00001900
0                 [000000] :                   02  DEVICE.STATUS.TABLE       BIT(256),             : 00002000
0                 [000100] :                   02  AVAILABLE.SYSTEM.MEMORY   BIT(8),               : 00002100
0                 [000108] :                   02  AVAILABLE.CONTROL.MEMORY  BIT(4),               : 00002200
0                 [00010C] :                   02  OPTIONS.TABLE             BIT(48),              : 00002300
0                 [00010C] :                       03 SWITCHES               BIT(8),               : 00002400
0                          :                       03 FILLER                 BIT(40),              : 00002500
0                          :                   02  FILLER                    BIT(1352);            : 00002600
0                          :                                                                       : 00002700
0                          :                                                                       : 00002800
0                          :     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%       : 00002900
0                          :     %%%%%%%%%%          GLOBAL DEFINES              %%%%%%%%%%       : 00003000
0                          :     %%%%%%%%%%                                      %%%%%%%%%%       : 00003100
0                          :     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%       : 00003200
0                          :       DEFINE         ADDR1                      = S0#                 : 00003300
0                          :       DEFINE         ADDR2                      = S1#                 : 00003400
0                          :       DEFINE         ADDR3                      = S2#                 : 00003500
0                          :       DEFINE         NEXT.INSTRUCTION.POINTER = S4A#                  : 00003600
0                          :                                                                       : 00003700
0                          :                                                                       : 00003800
0                          :                                                                       : 00003900
0                          :                                                                       : 00004000
0                          :%%%%%%%%%%%%%%%%%%%%%                                                  : 00004100
0                          :BEGINNING.OF.PROGRAM%                                                  : 00004200
0                          :%%%%%%%%%%%%%%%%%%%%%                                                  : 00004300
0                          :%                                                                      : 00004400
0                          :%  THIS PART OF THE INTERPRETER ARBITRARILY SETS THE STARTING ADDRESS OF: 00004500
0                          :%  THE NEXT INSTRUCTION ALONG WITH THE PROGRAM BASE AND LIMIT REGISTERS.: 00004600
0                          :%  (THIS INFORMATION WOULD NORMALLY BE FOUND BY SOME OTHER METHOD.)    : 00004700
0                          :%                                                                      : 00004800
0                          :%                                                                      : 00004900
0             9800 @ [000000] :       MOVE @3000@ TO NEXT.INSTRUCTION.POINTER                      : 00005000
0             0300 @ [000010] :                                                                    :          G
0             2884 @ [000020] :                                                                    :          G
0             960C @ [000030] :       MOVE @6000@ TO BR                                            : 00005100
0             0600 @ [000040] :                                                                    :          G
0             9700 @ [000050] :       MOVE @1800@ TO LR                                            : 00005200
0             1800 @ [000060] :                                                                    :          G
0                          :%                                                                      : 00005300
```

```
0                                      :%%%%%%        THIS ROUTINE ADJUSTS THE 24 BIT FUNCTION BOX TO 24 BIT      : 00005400
0                                      :FETCH%        BINARY; READS IN A 1 BIT OP CODE; AND PERFORMS A            : 00005500
0                                      :%%%%%%        CALCULATED BRANCH TO THE INDICATED EXECUTION ROUTINE.      : 00005600
0                                      :%                                                                         : 00005700
1 FETCH                                :          BEGIN FETCH                                                     : 00005800
1 FETCH        8C18 @ [00C70]          :                MOVE 24 TO CP                                             : 00005900
1 FETCH        28A4 @ [00080]          :                MOVE NEXT.INSTRUCTION.POINTER TO FA                       : 00006000
1 FETCH        7101 @ [00090]          :                READ 1 BIT TO X INC FA                %GET OP CODE.       : 00006100
1 FETCH        2884 @ [000A0]          :                MOVE FA TO NEXT.INSTRUCTION.POINTER                       : 00006200
1 FETCH        10A5 @ [000B0]          :                MOVE X TO M                                               : 00006300
1 FETCH        C000 @ [000C0]          :                JUMP FORWARD                                              : 00006400
1 FETCH        C001 @ [000D0]          :                     GO TO ADD                        %OP _ 0.            : 00006500
1 FETCH        C015 @ [000E0]          :                     GO TO END.OF.JOB                 %OP _ 1.            : 00006600
1 FETCH                                :          END FETCH                                                       : 00006700
0                                      :%                                                                         : 00006800
0                                      :%%%%          THIS EXECUTION ROUTINE PERFORMS AN ABSOLUTE, 3 ADDRESS,     : 00006900
0                                      :ADD%          BINARY OR DECIMAL ADD. OVERLAPPING FIELDS ARE PERMITTED.    : 00007000
0                                      :%%%%          LEFT TRUNCATION OR LEFT ZERO FILL WILL OCCUR IN THE         : 00007100
0                                      :%             RESULT FIELD IF IT IS RESPECTIVELY LONGER OR SHORTER        : 00007200
0                                      :%             THAN IS REQUIRED TO CONTAIN THE RESULTANT SUM. ALL 3        : 00007300
0                                      :%             FIELDS MUST BE OF THE SAME TYPE(BINARY, 4-BIT DECIMAL       : 00007400
0                                      :%             OR 8-BIT DECIMAL) OR THE RESULT IS NOT DEFINED.             : 00007500
0                                      :%                                                                         : 00007600
1 ADD                                  :          BEGIN ADD                                                       : 00007700
1 ADD          3162 @ [000F0]          :                MOVE 2 TO FT             %INDICATES A 3 ADDRESS INSTRUCTION: 00007800
1 ADD          8890 @ [00100]          :                MOVE 144 TO FA           %LENGTH OF 3 ADDRESSES           : 00007900
1 ADD          E013 @ [00110]          :                CALL LOAD.ADDRESSES                                       : 00008000
1 ADD          0050 @ [00120]          :                LOAD F FROM ADDR1                                         : 00008100
1 ADD                                  :.LOOP                                                                     : 00008200
1 ADD          E01F @ [00130]          :                CALL GET.DATA            %GET ADDEND.                     : 00008300
1 ADD          11A0 @ [00140]          :                MOVE Y TO X                                               : 00008400
1 ADD          0701 @ [00150]          :                XCH ADDR2 F ADDR1                                         : 00008500
1 ADD          E01C @ [00160]          :                CALL GET.DATA            %GET AUGEND.                     : 00008600
1 ADD          0712 @ [00170]          :                XCH ADDR3 F ADDR2                                         : 00008700
1 ADD          0033 @ [00180]          :                BIAS BY F TEST                                            : 00008800
1 ADD          D013 @ [00190]          :                     GO TO FETCH    %END OF RESULT FIELD AND INSTRUCTION. : 00008900
1 ADD          10E2 @ [001A0]          :                MOVE SUM TO T                                             : 00009000
1 ADD          0064 @ [001B0]          :                CARRY SUM                                                 : 00009100
1 ADD          60A8 @ [001C0]          :                IF FU EQL 8 THEN     %SUPPLY EBCDIC NUMERIC ZONES IF 8-BIT : 00009200
1 ADD          C003 @ [001D0]          :                                                                          :
2 ADD                                  :                BEGIN                                                     : 00009300
2 ADD          300F @ [001E0]          :                     MOVE @F@ TO TA                                       : 00009400
2 ADD          1002 @ [001F0]          :                     MOVE TA TO TC                                        : 00009500
2 ADD          1004 @ [00200]          :                     MOVE TA TO TE                                        : 00009600
2 ADD                                  :                END                                                       : 00009700
1 ADD          7FA0 @ [00210]          :                WRITE REVERSE FROM T DEC FA AND FL   %WRITE SUM.          : 00009800
1 ADD          0720 @ [00220]          :                XCH ADDR1 F ADDR3                                         : 00009900
1 ADD          D011 @ [00230]          :                GO TO .LOOP                                               : 00010000
1 ADD                                  :          END ADD                                                         : 00010100
0                                      :%                                                                         : 00010200
0                                      :%%%%%%%%%%%                                                               : 00010300
0                                      :END.OF.JOB%                                                               : 00010400
0                                      :%%%%%%%%%%%                                                               : 00010500
0                                      :%                                                                         : 00010600
1 END.OF.JOB                           :          BEGIN END.OF.JOB                                                : 00010700
1 END.OF.JOB   C001 @ [00240]          :                HALT                                                      : 00010800
1 END.OF.JOB                           :          END END.OF.JOB                                                  : 00010900
0                                      :%                                                                         : 00011000
0                                      :%%%%%%%%%%%%%%%     THIS GENERAL ROUTINE LOADS 1, 2 OR 3 ADDRESSES TO     : 00011100
0                                      :LOAD.ADDRESSES%     SCRATCHPAD AND UPDATES THE NEXT.INSTRUCTION.POINTER.  : 00011200
0                                      :%%%%%%%%%%%%%%%     ALL ADDRESSES ARE IN THE FOLLOWING FORMAT;            : 00011300
0                                      :%                        A) A 24 BIT, BASE RELATIVE, S-MEMORY ADDRESS     : 00011400
0                                      :%                             WHICH IS CONVERTED TO AN ABSOULUTE ADDRESS  : 00011500
0                                      :%                             POINTING AT THE END OF A FIELD   1.         : 00011600
```

```
0                                  :%              B) A 4 BIT TYPE FIELD(0001 FOR BINARY, 0100     : 00011700
0                                  :%                 FOR 4-BIT DECIMAL AND 1000 FOR 8-BIT         : 00011800
0                                  :%                 DECIMAL);                                    : 00011900
0                                  :%                                                              : 00012000
0                                  :%              C) 4 UNUSED BITS                                : 00012100
0                                  :%                                                              : 00012200
0                                  :%              D) A 16 BIT FIELD LENGTH INDICATING THE NUMBER  : 00012300
0                                  :%                 OF BITS CONTAINED IN EACH FIELD.             : 00012400
0                                  :%                                                              : 00012500
1 LOAD.ADDRE                       :          BEGIN LOAD.ADDRESSES                                 : 00012600
1 LOAD.ADDRE  0804 @ [00250]       :               ADD NEXT.INSTRUCTION.POINTER TO FA  %---> END OF ADDRESSES: 00012700
1 LOAD.ADDRE  2884 @ [00260]       :               MOVE FA TO NEXT.INSTRUCTION.POINTER             : 00012800
1 LOAD.ADDRE                       :.LOOP                                                          : 00012900
1 LOAD.ADDRE  16A1 @ [00270]       :               MOVE BR TO Y                                    : 00013000
1 LOAD.ADDRE  75B8 @ [00280]       :               READ 24 BITS REVERSE TO T DEC FA    %GET TYPE AND LENGTH. : 00013100
1 LOAD.ADDRE  7538 @ [00290]       :               READ 24 BITS REVERSE TO X DEC FA    %GET ADDRESS.  : 00013200
1 LOAD.ADDRE  10EC @ [002A0]       :               MOVE SUM TO X                       %ADDRESS   BASE. : 00013300
1 LOAD.ADDRE  BC30 @ [002B0]       :               EXTRACT 16 BITS FROM T(8) TO Y      %GET LENGTH    : 00013400
1 LOAD.ADDRE  1165 @ [002C0]       :               MOVE FT TO M                        %SELECT A SCRATCHPAD : 00013500
1 LOAD.ADDRE  20C0 @ [002D0]       :               MOVE SUM TO SOA                     %ADDRESS   LENGTH : 00013600
1 LOAD.ADDRE  1165 @ [002E0]       :               MOVE FT TO M                                    : 00013700
1 LOAD.ADDRE  2290 @ [002F0]       :               MOVE T TO SOB         %TYPE AND LENGTH.         : 00013800
1 LOAD.ADDRE  31F1 @ [00300]       :               DEC FT BY 1 TEST                                : 00013900
1 LOAD.ADDRE  D00B @ [00310]       :                  GO TO -LOOP                                  : 00014000
1 LOAD.ADDRE  18A4 @ [00320]       :               EXIT                                            : 00014100
1 LOAD.ADDRE                       :          END LOAD.ADDRESSES                                   : 00014200
0                                  :%                                                              : 00014300
0                                  :%%%%%%%%%      THIS GENERAL ROUTINE WILL READ FROM ZERO TO 24 BITS INTO : 00014400
0                                  :GET.DATA%      THE M-MACHINE, SET CPU TO THE ENCODED DATA TYPE AND SET : 00014500
0                                  :%%%%%%%%%      CPL TO THE NUMBER OF BITS READ.                 : 00014600
0                                  :%                                                              : 00014700
1 GET.DATA                         :          BEGIN GET.DATA                                       : 00014800
1 GET.DATA    0032 @ [00330]       :               BIAS BY F                                       : 00014900
1 GET.DATA    7760 @ [00340]       :               READ REVERSE TO Y DEC FA AND FL                 : 00015000
1 GET.DATA    18A4 @ [00350]       :               EXIT                                            : 00015100
1 GET.DATA                         :          END GET.DATA                                         : 00015200
0                                  :          FINI                                                 : 00015300
0             C00F @ [00360]       :                                                               :          G
0             0000 @ [00370]       :                                                               :          G
0             0000 @ [00380]       :                                                               :          G
0             C000 @ [00390]       :                                                               :          G

NO ERRORS DETECTED

MICRO INSTRUCTION COUNT = 00058

CAUTION: $ SUBSET WAS NOT SPECIFIED; THEREFORE, THIS
         PROGRAM SHOULD NOT BE USED ON A B1712/B1714.
```

APPENDIX A: MIL COMPILER OPERATION

CONTROL CARDS

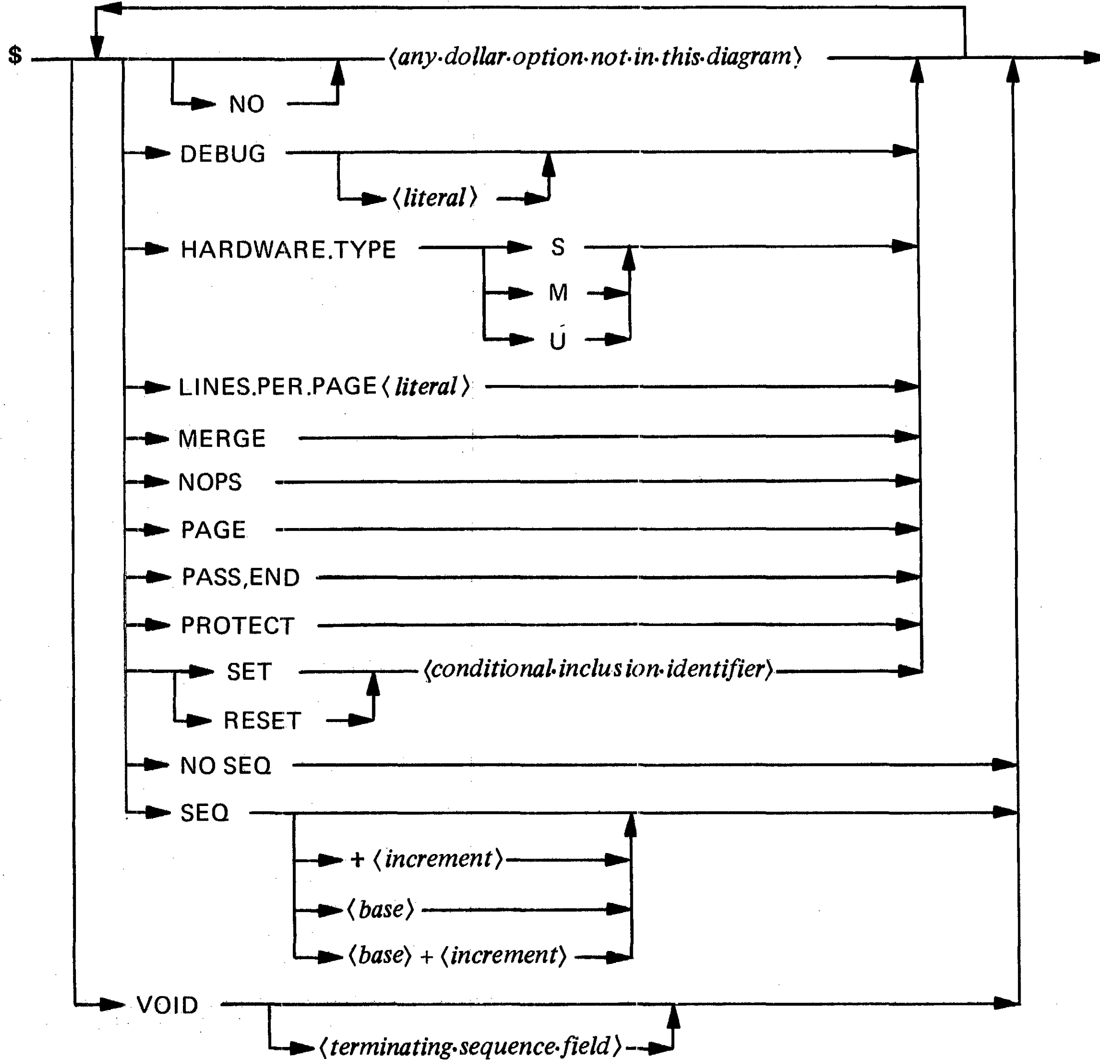
General

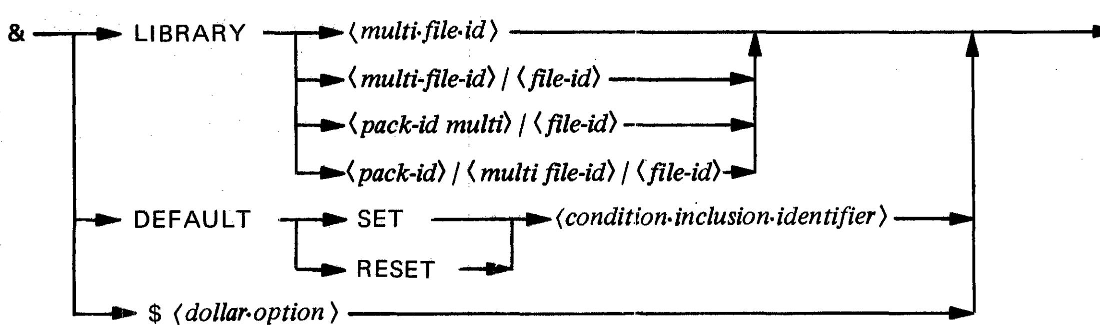
The purpose of the compiler control card is to allow the programmer to specify option settings to the compiler.

Every MIL control card has either a $ (dollar sign) in column 1 and is called a "dollar card", or has an & (ampersand) in column 1 and is called an "ampersand card". Column 73-80 may be used as a sequence field.

Dollar Cards

SYNTAX





SEMANTICS

ALLCODE	lists all code generated for each MIL statement when listing
AMPERSAND	lists all ampersand records, except &$ records, when listing (default on)
ANALYZE.CODEFILE	prints an analysis of the code file at end of source listing
CHECK	checks for sequence errors (default on)
COMPILE	when reset a fast source listing will be produced with no code generation or syntax checking (default on)
CONTROL	prints all dollar cards when listing; same as $DOLLAR
DEBUG	for compiler debugging use
DECK	punches an object deck
DOLLAR	prints all dollars cards on listing
DOUBLE	double spaces listing when printing
ERROR.FILE	lists errors and warnings on a separate printer file as well as on the main listing
EXPAND	when listing, prints all statements (including comments) within a macro when a macro is invoked
EXTERNAL	generates external segment branching code (on by default)
FORCE	generates a code file regardless of syntax errors
FRAME	lists all IF, BEGIN . . . END statements which conditionally exclude code (default on)
HARDWARE.TYPE = { U / S / M }	specifies which hardware processor type will be used: S = S-Memory; M = M-Memory; U = Universal
HEADINGS	prints all title and subtitle headings at the beginning of each page when listing
LINES.PER.PAGE	specifies the maximum number of lines per page of listing

LIST	lists all source records excluding macro records that are compiled (default on)
LISTALL	lists all unconditionally excluded records to be printed
LIST.NOW	lists source records when read; same as $LISTP
LIST.PATCHES	lists all patches from CARDS file when read
LISTP	same as $ LIST.NOW
MERGE	merges a secondary source file ("CARDS") with the primary source file ("SOURCE") replacing primary source records by secondary records with the same sequence numbers
NEW	creates a new source file ("NEWSOURCE")
NO	resets any specified dollar option if allowed
NOPS	generates NOPs in external linking code for debugging purposes
OLD.LISTING.FORMAT	produces listing in pre-V.1 compiler format
PAGE	skips to a new page before printing the next line
PAGE.NUMBERS	puts page number on each new page when listing and puts a maximum number of lines on a page (60 by default) which can be changed by $LINES.PER.PAGE
PARAMETER.BLOCK	punches a parameter block with the object deck if used with $DECK; otherwise only code is punched
PASS.END	displays compiler pass information on the SPO
PROTECT	protects SKIP when specified
RELEASE	generates a release tape with listing, code deck, code file, and new source
RESET	resets any specified conditional inclusion option
SEQ	resequences source records
SET	sets any specified conditional inclusion options
SINGLE	prints single-space listings (default on)
SUBSET	generates code for B1710 (S-Memory) Processors; same as $HARDWARE.TYPE=S

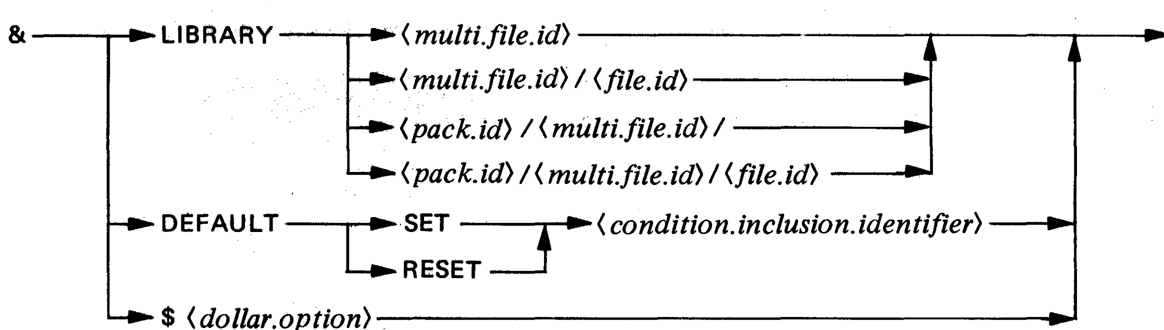
SUPPRESS	suppresses printing of warning messages
VOID	deletes a specified range of source records. The terminating sequence range must be exactly 8 characters
XREF	sets XREF.LABELS and XREF.NAMES
XREF.ALL	sets XREF.LABELS, XREF.NAMES and XREF. REGISTERS
XREF.LABELS	cross-references all labels
XREF. NAMES	cross-references all names
XREF.REGISTERS	cross-references all registers

NOTES AND RESTRICTIONS

1. Unless otherwise specified (through the MERGE option), the only source of input is the card reader. Once $ MERGE has been specified and the first non-$ record has been encountered, it is not possible to again indicate "CARDS ONLY".
2. If no dollar cards are used in the default options are: EXTERNAL, AMPERSAND, CHECK, COMPILE, FRAME, LIST and SINGLE. All input will be from the CARDS file.
3. Options are turned off only through the appearance of NO followed by the option word. Note that NO and the option word are separated by at least one blank.
4. Comments may appear on dollar cards by preceeding the comment with a % (percent sign).
5. Dollar cards are not included as part of a "NEWSOURCE" file when $ NEW is specified.

Ampersand Cards

Syntax



SEMANTICS

LIBRARY	Causes the specified file to be opened and compiled. Compilation proceeds to the end of the Library file with no contribution from any standard primary or secondary input file. At end of file, compilation is resumed from the standard input files.
DEFAULT	Specifies default settings for one or more conditional inclusion toggles. The default setting for a particular toggle will take effect only if no previous $ or & card specified a setting for that toggle. EXAMPLE: & DEFAULT SET TOG.A RESET TOB.B

NOTES AND RESTRICTIONS

1. A library file is assumed to be a disk file.
2. The last record in a library file that is to be compiled must be FINI: This record cannot be omitted.
3. All & records are included as part of a "NEWSOURCE" file when $ NEW is specified.
4. &$ records are listed only when both $ DOLLAR and $ AMPERSAND are specified.
5. LIBRARY, DEFAULT and $ statements may not be intermixed on a single & card.

MIL COMPILER FILES

Some of the internal file names in the compiler and the file uses are listed below. This information will find use in label equation at compile time.

CARDS	Input file containing control and source records. The DEFAULT bit is set for this CARD.READER file.
LINE	Output file for the compile listing. The device is PRINTER or BACKUP.
PUNCH	Output PUNCH or BACKUP file for the object deck produced when "$ DECK" is specified.
SOURCE	Secondary input file for source records when "$ MERGE" is specified. The DEFAULT bit is set on this DISK file.
NEWSOURCE	Output DISK file for new source records when "$ NEW" is specified. The file contains 90-byte records, blocked 4.
LIBSOURCE	Input DISK file for source records when "$ LIBRARY ⟨*file name*⟩ is encountered. The DEFAULT bit is set for the file.
LINESAVE	A temporary work file containing a copy of the listing.
CODE.FILE	A temporary work file containing a copy of the object code.
PARAM.FILE	A temporary work file containing parameters affecting the object code and the listing.

MILXREF	A temporary disk file containing information to be processed during the cross-referencing phase. The file is produced only if one of the "$ XREF" options is specified.
CODE	The actual generated code file. This DISK file contains a maximum of 300 180-byte records, and may contain only one area.
ERROR.LINE	An auxilliary PRINTER or BACKUP file replicating lines on file LINE that have caused syntax errors, and the actual error messages, if "$ERROR.FILE" has been specified. This allows the main listing to go to backup with an immediate indication of any syntax errors.

APPENDIX B: HARDWARE INSTRUCTION FORMATS AND TABLES

B 1700 HARDWARE TABLES

Table B-1. Register Addressing

Group (Row) Number	SELECT (Column) NUMBER			
	0	1	2	3
0	TA	FU	X	SUM
1	TB	FT	Y	CMPX
2	TC	FLC	T	CMPY
3	TD	FLD	L	XANY
4	TE	FLE	A	XEOY
5	TF	FLF	M	MSKX
6	CA	BICN	BR	MSKY
7	CB	FLCN	LR	XORY
8	LA	*TOPM	FA	DIFF
9	LB	RESERVED	FB	MAXS
10	LC	RESERVED	FL	*MAXM
11	LD	*PERR	TAS	U
12	LE	XYCN	CP	*MBR
13	LF	XYST	*MSM	DATA
14	CC	*INCN	READ	CMND
15	CD	RESERVED	WRIT	NULL

*Available on B 1720 systems only

Table B-2. Condition Registers

	Bits			
	0	1	2	3
BICN	LSUY	CYF	CYD	CYL
XYCN	MSBX	X = Y	X Y	X Y
XYST	LSUX	INT	Y NEQ 0	X NEQ 0
FLCN	FL = SFL	FL SFL	FL SLF	FL NEQ 0
*INCN	PORT DEVICE MISSING	PORT HIGH PRIORITY	PORT INTERRUPT	PORT LOCKOUT
CC	STATE LIGHT	TIMER INTERRUPT	I/O INTERRUPT	CONSOLE INTERRUPT
CD	MEMORY READ DATA PARITY ERROR INTERRUPT	MEMORY * WRITE/SWAP ADDR OUT OF BOUNDS OVER-RIDE	MEMORY * READ ADDR OUT OF BOUNDS INTERRUPT	MEMORY * WRITE/SWAP ADDR OUT OF BOUNDS INTERRUPT

*Available on B 1720 systems only

NOTES

1. BICN, FLCN, XYST, and XYCN are addressable as source registers only.

2. The TOPM, MBR, and A registers are used to determine the memory (control or main) and location of the next micro-instruction.

3. MSMA is control memory and may be addressed only from the maintenance Console during tape mode.

4. CPU is destination register only.

5. NULL always contains a value of 0. Any register or scratchpad word to which it is moved will be cleared to 0.

MICRO NAME	MC 0	1	2	3	MD 4	5	6	7	ME 8	9	10	11	MF 12	13	14	15
REGISTER MOVE	0	0	0	1	REG 1 GROUP SOURCE REGISTER				REG 1 SELECT		REG 2 SEL CT		REG 2 GROUP SINK REGISTER			
SCRATCHPAD MOVE	0	0	1	0	REGISTER GROUP SOURCE OR SINK				REG SELECT		MOV DIR	DPW 1/2	DOUBLE PAD WORD ADDRESS			
4 BIT MANIPULATE	0	0	1	1	REGISTER GROUP 4 BIT SOR & SNK				REG SEL	MANIPULATE VARIANTS			4 BIT MANIP. LITERAL			
BIT TEST REL BRANCH FALSE	0	1	0	0	REGISTER GROUP 4 BIT SOURCE				REG SEL	TESTBIT NUMBER		DSP SGN	RELATIVE BRANCH DISPLACEMENT MAG			
BIT TEST REL BRANCH TRUE	0	1	0	1	REGISTER GROUP 4 BIT SOURCE				REG SEL	TESTBIT NUMBER		DSP SGN	RELATIVE BRANCH DISPLACEMENT MAG			
SKIP WHEN	0	1	1	0	REGISTER GROUP 4 BIT SOR & SNK				REG SEL	SKIP TEST VARIANTS			4 BIT TEST MASK			
READ/WRITE MEMORY	0	1	1	1	R/W VAR	COUNT FA/FL VARIANTS			DATA REG CODE		TW SGN	DATA TRANSFER WIDTH MAGNITUDE				
MOVE 8 BIT LITERAL	1	0	0	0	REGISTER GROUP. REG SEL IS 2				ENTIRE 8 BITS OF 8 BIT LITERAL							
MOVE 24 BIT LITERAL	1	0	0	1	REGISTER GROUP. REL SEL IS 2				8 MOST SIGNIFICANT BITS OF FULL 24 BIT LITERAL							
SHIFT/ROTATE T REG	1	0	1	0	SINK REGISTER GROUP				SNK REG SELECT		S/R VAR	LEFT SHIFT/ROTATE COUNT				
EXTRACT FROM T REG	1	0	1	1	RIGHT BIT POINTER FOR EXTRACTION FLD					SNK REG CODE		EXTRACTION FIELD WIDTH				
BRANCH RELATIVE	1	1	0	DSP SGN	RELATIVE DISPLACEMENT MAGNITUDE											
CALL RELATIVE	1	1	1	DSP SGN	RELATIVE CALLED ADDRESS MAGNITUDE											
SWAP MEMORY	0	0	0	0	0	0	1	0	DATA REG CODE		TW SGN	DATA TRANSFER WIDTH MAGNITUDE				
CLEAR REGISTERS	0	0	0	0	0	0	1	1	L REG	T REG	Y REG	X REG	FA REG	FL REG	FU REG	CP REG
SHIFT/ROTATE X OR Y	0	0	0	0	0	1	0	0	S/R DIR VARIANT		X/Y VAR	LEFT OR RIGHT, X O R Y SHIFT/ROTATE COUNT				
SHIFT/ROTATE X AND Y	0	0	0	0	0	1	0	1	S/R DIR VARIANT		LEFT OR RIGHT X AND Y SHIFT/ROTATE COUNT					
COUNT FA/FL	0	0	0	0	0	1	1	0	COUNT FA/FL VARIANTS			COUNT SCALAR MAGNITUDE				
EXCHANGE DPW	0	0	0	0	0	1	1	1	SINK DPW ADDRESS				SOURCE DPW ADDRESS			
SCRATCHPAD RELATE FA	0	0	0	0	1	0	0	0	/	/	/	DSP SGN	LEFT HALF PAD WORD ADDRESS			
MONITOR	0	0	0	0	1	0	0	1	LITERAL OCCURRENCE IDENTIFIER							
DISPATCH	0	0	0	0	0	0	0	0	0	0	0	1	DISPATCH VARIANTS			SKP FLG
CASSETTE CONTROL	0	0	0	0	0	0	0	0	0	0	1	0	CASSETTE MANIP VARIANTS			
BIAS	0	0	0	0	0	0	0	0	0	0	1	1	BIAS VARIANTS			TST FLG
STORE F INTO DPW	0	0	0	0	0	0	0	0	0	1	0	0	SINK DPW ADDRESS			
LOAD F FROM DPW	0	0	0	0	0	0	0	0	0	1	0	1	SOURCE DPW ADDRESS			
CARRY FF MANIPULATE	0	0	0	0	0	0	0	0	0	1	1	0	CYF ← CYD	CYF ← CYL	CYF ← 1	CYF ← 0
HALT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
OVERLAY M-STRING	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
NORMALIZE X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
TRANSFER CONTROL	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
NO OPERATION	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

MICRO NAME	VARIANTS	0 / 00 / 000	1 / 01 / 001	10 / 010	11 / 011	100	101	110	111
REGISTER MOVE									
SCRATCHPAD MOVE	MOV DIR: 1/2 DPW:	P ← R LEFT	R ← P RIGHT						
4 BIT MANIPULATE	MANIP VARIANTS:	SET	AND	OR	EOR	INC	INC TEST	DEC	DEC TEST
BIT TEST REL BRANCH FALSE	DSP SIGN:	+	−						
BIT TEST REL BRANCH TRUE	DSP SIGN:	+	−						
SKIP WHEN	SKIP TEST VARIANTS:	ANY CLR/	ALL CLR/	EQL CLR/	ALL CLR	ANY/ CLR/	ALL/ CLR/	EQL/ CLR/	ALL/ CLR
READ/WRITE MEMORY	R/W VAR: CNT VAR:	READ NØP	WRT FA	FL↑	FA↑ FL↓	FA↓ FL↑	FA↓	FL↓	FA↓ FL↑
MOVE 8 BIT LITERAL	REG SEL: TW SIGN:	X +	Y −	T	L				
MOVE 24 BIT LITERAL									
SHIFT/ROTATE T REG	S/R VAR:	SHFT	ROT						
EXTRACT FROM T REG	SINK REG CODE:	X	Y	T	L				
BRANCH RELATIVE	DSP SIGN:	+	−						
CALL RELATIVE	DSP SIGN:	+	−						
SWAP MEMORY	TW SIGN: REG CODE:	+ X	− Y	T	L				
CLEAR REGISTERS									
SHIFT/ROTATE X OR Y	X/Y VAR: S/R, DIR:	X SFT ←	Y SFT →	ROT ←	ROT →				
SHIFT/ROTATE X AND Y	S/R, DIR VARIANTS:	SFT ←	SFT —	ROT ←	ROT →				
COUNT FA/FL	COUNT FA/ FL VAR:	NØP	FA↑	FL↑	FA↑ FL↓	FA↓ FL↑	FA↓	FL↓	FA↓ FL↓
EXCHANGE DPW									
SCRATCHPAD RELATE FA	DSP SIGN:	+	−						
MONITOR									
DISPATCH	SKP FLAG: DISP VAR:	FAIL LOCK	SUCC WRTLO	READ	R & C	WRITHI	ABSNT	UNDEF	UNDEF
CASSETTE CONTROL	CASSETTE MANIP:	START TAPE	STOP @ GAP	STOP ON X≠Y	UNDEF	UNDEF	UNDEF	STOP ON X = Y	UNDEF
BIAS	TEST FLG: BIAS VAR:	TST/ UNIT	TEST F	S	F5	NØP	FCP	NØP	NØP
STORE F INTO DPW									
LOAD F FROM DPW									
CARRY FF MANIPULATE									

Table B-3. Microinstructions

Table B-4. Variant Field Definitions

FOUR-BIT MANIPULATE (3nnn) VARIANTS

BITS 4-6	CONDITIONS
000	SET
001	AND
010	OR
011	EOR
100	INC
101	INC/TEST
110	DEC
111	DEC/TEST

SKIP WHEN (6nnn) SKIP TEST VARIANTS

BITS 4-6	CONDITIONS
000	ANY. SKIP
001	ALL. SKIP
010	EQU. SKIP
011	ALL CLR. SKIP
100	NOT ANY. SKIP
101	NOT ALL. SKIP
110	NOT EQU. SKIP
111	NOT ALL. CLR. SKIP

READ/WRITE MEMORY (7nnn) VARIANTS

BITS 6-7	CONDITIONS
00	X REG.
01	Y REG.
10	T REG.
11	L REG.

BITS 8-10	CONDITIONS
000	NOP
001	FA UP
010	FL UP
011	FA UP FL DN
100	FA DN FL UP
101	FA DN
110	FL DN
111	FA DN FL DN

EXTRACT FROM T REG. (8nnn) VARIANTS

BITS 5-6	CONDITIONS
00	X REG.
01	Y REG.
10	T REG.
11	L REG.

SWAP MEMORY (02nn) VARIANTS

BITS 6-7	CONDITIONS
00	X REG.
01	Y REG.
10	Y REG.
11	L REG.

CASSETTE CONTROL (002n) VARIANTS

BITS 3-1	CONDITIONS
000	START TAPE
001	STOP ON GAP
010	STOP ON X NEQ Y
011-111	RESERVED

COUNT FA AND FL (06nn) VARIANTS

BITS 5-7	CONDITIONS
000	NOP
001	FA UP
010	FL UP
011	FA UP FL DN
100	FA DN FL UP
101	FA DN
110	FL DN
111	FA DN FL DN

DISPATCH (001n) VARIANTS

BITS 1-3	CONDITIONS
000	DISPATCH LOCK
001	DISPATCH WRITE
010	DISPATCH READ
011	DISPATCH RD & CLR
100	RESERVED
101	RESERVED
110	RESERVED
111	RESERVED

BIAS (003n) VARIANTS

BITS 3-1	CONDITIONS
000	FU
001	24 OR FL
010	24 OR SFL
011	24 OR FL OR SFL
100	NOP
101	24 OR CPL OR FL
111	OR SFL

B 1700 HARDWARE INSTRUCTION FORMATS

Bias

OP CODE 0000 0000 0011	BIAS VARIANTS (V) 0 . . . 7	TEST CPL NEQ 0 FLAG 0 - NO TEST 1 TEST CPL RESULT
0 – 11	12 – 14	15

This instruction sets CPU to the value 1 if the value of FU is 4 or 8 and to 0 otherwise, unless V = 2. If V = 2, the value of the CPU is determined by SFU in lieu of FU. SFU is the first 4 bits of the scratchpad word S0B. (On the B 1710, FU = 8 will set CPU = 0.)

The value of CPL is also set to the smallest of the values denoted in the following table.

V	VALUES
0	FU
1	24 or FL
2	24 or SFL
3	24 or FL or SFL
4	CPL
5	24 AND CPL AND FL
6	CPL
7	CPL (not defined on the B 1710)

If the test flag equals 1 and the final value of CPL is not 0, the next micro-instruction is skipped.

Bit Test Branch False

OP CODE 0100	REGISTER GROUP # 0 . . . 15	REGISTER SELECT # 0 . . . 1	REGISTER BIT # 0 . . 3	DISPLACEMENT SIGN 0 - POSITIVE 1 - NEGATIVE	DISPLACEMENT VALUE 0 . . 15
0 – 3	4 – 7	8	9 – 10	11	12 – 15

This micro-instruction tests the designated bit within the specified register and branches (relative to the next micro-instruction) by the amount and direction of the signed displacement value if the bit is 0. If the bit is 1, a displacement value of 0 is assumed, and control passes to the next in-line micro-instruction. A displacement value indicates the number of 16-bit words from the next in-line micro-instruction. A negative sign indicates lower addresses (backward displacement). The maximum displacement is 15 micro-instructions.

NOTE

Register Bit # is read from right to left, 0 - 3 in accordance with the hardware bit numbering convention.

Bit Test Branch True

OP CODE 0101	REGISTER GROUP # 0 ..15	REGISTER SELECT # 0...1	REGISTER BIT # 0...3	DISPLACEMENT SIGN 0 POSITIVE 1 NEGATIVE	DISPLACEMENT VALUE 0...15
0 3	4 7	8	9 10	11	12 15

This instruction tests the designated bit within the specified register and branches (relative to the next instruction) by the amount and direction of the signed displacement value if the bit is 1. If the bit is 0, a displacement value of 0 is assumed, and control passes to the next in-line micro-instruction. A displacement value indicates the number of 16-bit words from the next in-line micro-instruction. A negative sign indicates lower addresses (backward displacement). The maximum displacement is 15 micro-instructions.

NOTE

Register Bit # is read right to left, 0 - 3 in accordance with hardware bit numbering convention.

Branch

OP CODE 110	DISPLACEMENT SIGN 0 POSITIVE 1 - NEGATIVE	DISPLACEMENT VALUE 0...4095
0 2	3	4 15

This instruction fetches the next micro-instruction from the location obtained by adding the signed displacement value given in the instruction to the address of the next in-line micro-instruction.

A displacement value indicates the number of 16-bit words.

Call

OP CODE 111	DISPLACEMENT SIGN 0 - POSITIVE 1 NEGATIVE	DISPLACEMENT VALUE 0 . . . 4095
0 2	3	4 15

This instruction pushes the address of the next in-line micro-instruction (already contained in A register) into the A stack and then fetches the next micro-instruction from the location obtained by adding the signed displacement value given in the instruction to the address of the next in-line micro-instruction.

A displacement value indicates the number of 16-bit words.

NOTES

1. EXIT, the opposite of CALL, is accomplished by employing the MOVE register instruction with TAS as the source register and A as the sink register.

2. When the A address is stored in the A stack, it is multiplied by 16 and stored as a bit address.

Cassette Control

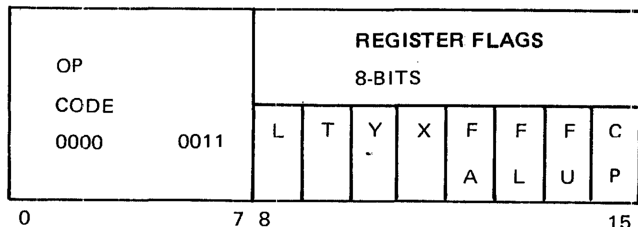
OP CODE 0000 0000 0010	CASSETTE MANIPULATE VARIANTS (V) 0 . . . 7	RESERVED FLAG BIT 0 . . . 1
0 11	12 14	15

This instruction performs the indicated operation on the tape cassette.

V = 0 Start Tape
1 Stop Tape
2 Stop Tape if X NEQ Y
3 Reserved
4 Reserved
5 Reserved
6 Reserved
7 Reserved

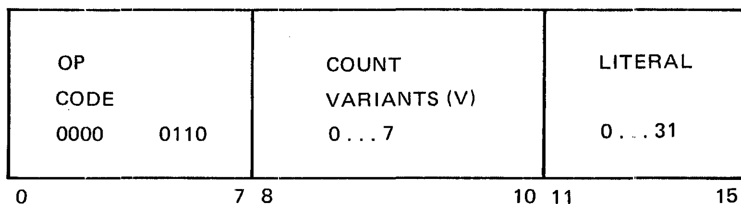
All Stop Tape variants cause the tape to halt in the next available gap.

Clear Registers



This micro-instruction clears the specified register(s) to 0 if the respective flag bit is 1.

Count FA/FL



This micro-instruction increments (decrements) binarily the designated register(s) by the value of the literal contained in the micro-instruction or by the value of CPL if the value of the literal is 0.

Neither overflow nor underflow of FA is detected. The value of FA may go through its maximum value or its minimum value and wrap around.

Overflow of FL is not detected. The value of FL may go through its maximum value and wrap around. Underflow of FL is detected and will not wrap around. The value 0 is left in FL.

Literal values (or CPL values if LIT=0) of 25 through 31 are truncated to the value 24.

Count variants are as follows:

V =	000	No Count
	001	Count FA UP
	010	Count FL UP
	011	Count FA UP and FL DOWN
	100	Count FA DOWN and FL UP
	101	Count FA DOWN
	110	Count FL DOWN
	111	Count FA DOWN and FL DOWN

Dispatch

(Requires a hardware I/O subsystem available on the B 1720 only)

OP CODE 0000 0000 0001	DISPATCH VARIANTS	SKIP VARIANT (Applies only to lockout variant)
	000 - LOCKOUT	0 - SKIP IF ALREADY LOCKED
	001 - WRITE	1 - SKIP IF NOT ALREADY LOCKED
	010 - READ	
	011 - READ AND CLEAR	
	100 - WRITE HIGH	
	101 - PORT ABSENT	
0 – 11	12 – 14	15

This micro-instruction sends/receives interrupt and interrupt information to/from other ports.

Since the interrupt system is shared by all ports, the processor should gain control of the interrupt system by successfully completing a LOCKOUT prior to a DISPATCH WRITE.

LOCKOUT sets the lockout bit in the DISPATCH register and allows, via the skip variant, skipping or not skipping the next 16-bit instruction based upon the success or failure (already set) of the LOCKOUT.

WRITE (High or Low) DISPATCH sets the Lockout and Interrupt flip flops in the port interchange. It also stores the contents of the L register into memory location 0 to 23 and the contents of the least-significant seven bits of the T register (designating the destination port# and channel#) into the appropriate port interchange register. In addition, it sets (Write High) or resets (Write Low) the high Interrupt flip flop in the port interchange.

READ DISPATCH stores the contents of memory locations 0 through 23 into the L register and the contents of the Port Channel register into the least significant 7 bits of the T register. The other 17 bits of T are unaffected.

READ AND CLEAR DISPATCH in addition to performing the READ DISPATCH operation clears the lockout flip flop, the two interrupt flip flops and the Port Device Absent flip flop in the port interchange. It does not clear any memory locations.

PORT ABSENT is executed by the processor when necessary to return a Port Device Absent Level signal to another port indicating the absence of the designated channel.

Dispatch operations in the case of Processor-2 and Processor Adapter-1 (direct connect to memory) are limited to the following:

a. LOCKOUT + SKIP-IF-NOT-ALREADY-LOCKED: always skips.

b. WRITE LOW: always sets Port Device Absent Level true (true indicates absence).

c. READ and CLEAR: always sets the Port Device Absent level false (false indicates present).

No changes occur in the T and L registers. In the INCN register only the Port Device Absent bit can change. The Lockout, the Interrupt, and High Priority bits will always be false. No other dispatch operations are defined.

Extract From T

OP CODE 1011	ROTATE BIT COUNT 0 . . . 24	DESTINATION REGISTER 00 - X 01 - Y 10 - T 11 - L	EXTRACT BIT COUNT 0 . . . 24
0 3	4 8	9 10	11 15

This micro-instruction rotates the T register contents left by the ROTATE count, extracts the bits specified and moves the result to the sink register. If the extract bit count is less than 24, the data is right-justified with the left (most-significant) zero bits supplied.

The contents of the T register are unchanged unless it is also the sink register.

A rotate value of 24 is equal to 0 and is equivalent to a NO OPERATION.

NOTE

The microprogramming language compiler uses the left-most bit to be extracted and calculates the rotate bit count to be used by the hardware circuits. The assembler addresses the bits within the T register left to right as 0 through 23; hardware addresses the bits right to left as 0 through 23.

Four-Bit Manipulate

OP CODE 0011	REGISTER GROUP # 0 . . . 15	REGISTER SELECT # 0 . . . 1	MANIPULATE VARIANTS (V) 0 . . . 7	LITERAL 0 . . . 15
0 3	4 7	8	9 11	12 15

This micro-instruction performs the operation specified by the variants on the designated register.

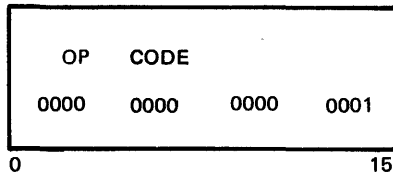
- V = 0 The register is set to the value of the literal.
- 1 The register is set to the logical AND of the register and literal.
- 2 The register is set to the logical OR of the register and literal.
- 3 The register is set to the logical EXCLUSIVE-OR of the register and literal.
- 4 The register is set to the binary sum (modulo 16) of the register and literal.

5 The register is set to the binary sum (modulo 16) of the register and literal, and the next micro-instruction is skipped if a carry is produced.
6 The register is set to the binary difference (modulo 16) of the register and the literal.
7 The register is set to the binary difference modulo 16 of the register and literal, and the next micro-instruction is skipped if a borrow is produced.

EXCEPTION

BICN, FLCN, XYCN, XYST, INCN (B1720) and CPU (B 1710) when specified as operand registers are not changed as a result of this operation. However, the carry or borrow outputs are produced and a skip can result.

Halt

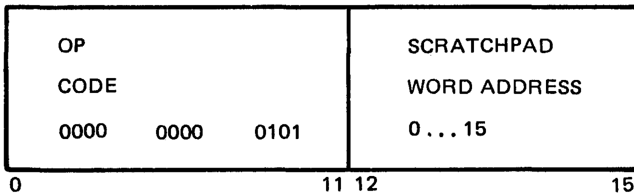


This micro-instruction stops the execution of the micro-instructions. In RUN mode the next micro to be executed is fetched and stored in the M register, and the A register points to the next following micro. In TAPE mode the next micro is not fetched and stored in the M register, but the HALT micro is left in the M register.

The register indicated by the register select switch will be displayed.

Load F From Doublepad Word

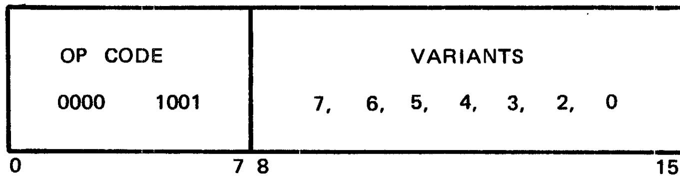
(Available on B 1720 systems only.)



This micro-instruction moves the contents of the A and B portions of the designated scratchpad word to the FA and FB registers respectively.

Monitor

(Available on B 1720 systems only.)



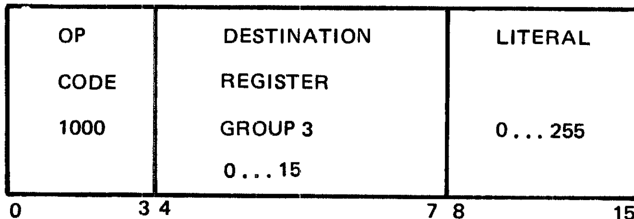
This micro-instruction skips to the next sequential instruction.

During the time this micro-operator is executing the operator and the last two bits (0 and 1) are decoded, ANDed with the system clock and are present in the backplane as follows:

MONITOR	0	True for the OP Code
MONITOR	00R0	True if last two bits are 00
MONITOR	01R0	True if last two bits are 01
MONITOR	02R0	True if last two bits are 10
MONITOR	03R0	True if last two bits are 11

At the backplane, the monitors are one-half clock from leading edge to trailing edge.

Move 8-Bit Literal



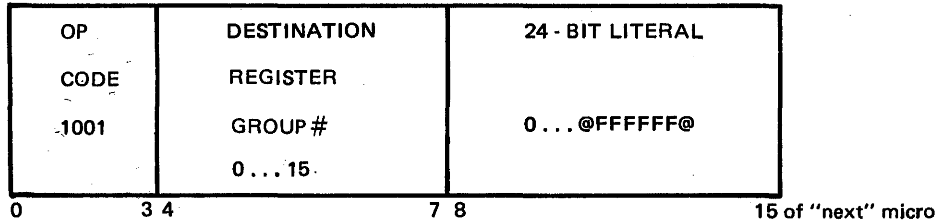
This micro-instruction moves the 8-bit literal given in the micro-instruction to the sink register. If the move is to a register of length 〉 8 bits, the data is right-justified with left (most-significant) zero bits supplied.

EXCEPTIONS

1. READ and WRITE are excluded as sinks.

2. When M is used as a sink register, the operation is changed to a bit-OR which modifies the next micro-instruction. It does not modify the micro-instruction as stored in memory.

Move 24-bit Literal

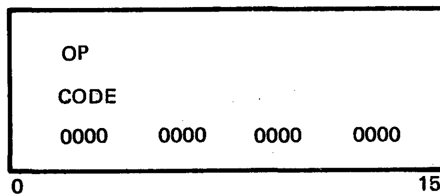


This micro-instruction moves the 24-bit literal given in the double-length micro-instruction to the sink register. If the move is between registers of length ⟨ 24 bits, the literal is truncated from the left.

EXCEPTIONS

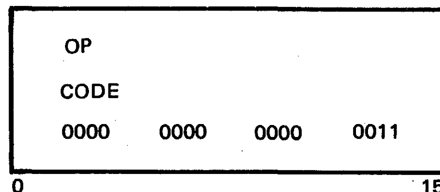
1. READ, WRITE, M and CP (B 1710) are excluded as sinks.
2. The MSMA register (available only on the B 1720) may be a sink only in the TAPE mode.

No Operation



This micro-instruction initiates a skip to the next sequential micro-instruction.

Normalize X

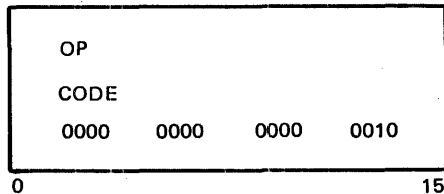


This micro-instruction shifts the X register left while counting FL down until FL = 0 or until the bit in X referenced by CPL = 1. Zeros are shifted into the right-most end of X.

CPL = 1 references the right-most bit of X while CPL = 24 references the left-most bit of X. If CPL = 0, the operation will continue until FL = 0.

Overlay Control Memory

(Available on B 1720 systems only.)



This micro-instruction overlays control memory (M-Memory) from main memory.

The starting main memory address is in the FA register; the length of the data to be overlaid, in bits, is in the FL register. The starting control memory address is in the L register.

Execution of the micro-instruction proceeds as follows:

a. The contents of the A register are moved to the TAS register.

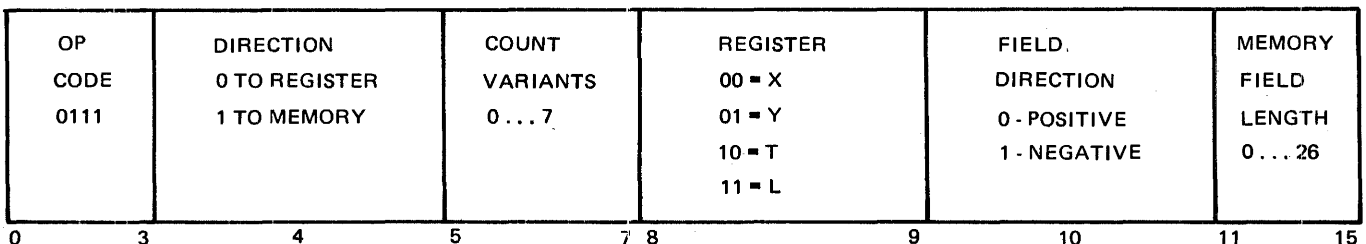
b. The contents of the L register are moved to the A register.

c. The first 16 bits of data are read from main memory and stored in the control memory via register L. Register FL is decremented by 16 bits; FA is incremented by 16 bits; and A is incremented by 1 word.

d. Step 3 is repeated until FL = 0 or A MAXM, at which point the process terminates with a move of TAS to A.

e. The operation then continues with the next micro-instruction.

Read/Write Memory



This micro-instruction moves the contents of the register (memory) to the memory (register). If the value of the memory field length is less than 24, the data from memory is right-justified with left (most-significant) zero bits supplied while the data from the register is truncated from the left.

The contents of the source is unchanged.

Register FA contains the bit address of the memory field while the memory field direction sign and memory field length are given in the instruction.

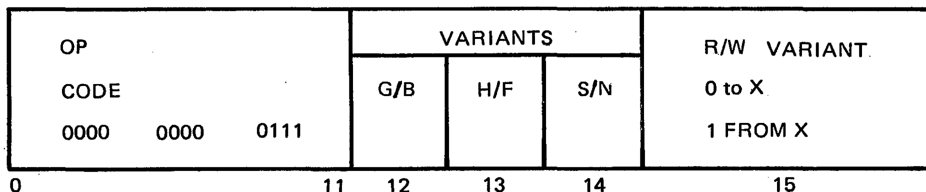
If the value of the memory field length as given in the instruction is 0, the value in CPL is used.

Memory field length values (or CPL values if Memory Field Length = 0) of 25 and 26 are truncated to the value of 24. When used on a WRITE operation, the value 25 and 26 cause odd and even parity respectively to be written into memory regardless of the parity of the read data.

For a description of the count variants, see COUNT FA/FL.

Read/Write MSM

(Available on B 1720 systems only.)



This micro-instruction (1) moves the contents of the X register to the M-Memory word specified by the address contained in the L register if the R/W variant bit = 1; data is right justified with left (most significant) bits supplied or (2) moves the contents of the M-Memory word specified by the address contained in the L register to the X register if the R/W variant bit = 0; data is right justified with left (most significant) zero bits supplied.

The lower 4 bits and the upper 8 bits of the address in L are ignored.

READ/WRITE MSM causes the A register to be moved to the TAS register and the L register to be moved to the A register before the instruction is executed. The TAS is restored to A after the READ/WRITE MSM operation is completed.

The S variant is used to enable the set/reset of the G/B and H/F flip flops. If S = 1, the G/B and H/F flip flops are set/reset by the G/B and H/F variants. If S = 0, no change is made in the G/B and H/F flip flops.

If the G/B flip flop is true, all READ/WRITE MSM operations will force bad parity in the addressed word. If the G/B flip flop is false, all READ/WRITE MSM operations will force good parity in the addressed word.

If the M/F flip flop is true, the processor upon reading an M-Memory word containing parity error will flag the error condition by setting a CD bit true. It will not halt. If the H/F flip flop is false, the processor upon detection of a parity error in reading an M-Memory word will flag the error condition by setting PERR bit 1 true and then halt. Reading an M-Memory word occurs when fetching a M-op from M-Memory or when moving an M-Memory word to any destination.

The H/F and G/B flip flops are cleared to zero (false) with the CLEAR signal. If S = 1, the G/B and H/F flip flops are set/reset prior to the execution of the READ/WRITE MSM portion of the operation.

Register Move

OP CODE 0001	SOURCE REGISTER GROUP # 0...15	SOURCE REGISTER SELECT # 0...3	DESTINATION REGISTER GROUP # 0...3	DESTINATION REGISTER SELECT # 0...15
0 3	4 7	8 9	10 11	12 15

This micro-instruction moves the contents of the source register to the sink register. If the move is between registers of unequal lengths, the data is right-justified with left (most-significant) zero bits supplied or the data is truncated from the left, whichever is appropriate.

The contents of the source register are unchanged unless it is also the sink register.

EXCEPTIONS

1. WRIT, CMND (and CPU, READ on B 1720) are excluded as source registers.

2. When the M register is used as a sink in RUN or STOP mode, the operation is changed to an bit-OR which modifies the next micro-instruction. It does not modify the instruction stored as in memory. In TAPE mode, no bit-OR takes place.

3. BICN, FLCN, XYCN, XYST, INCN, READ, WRIT, SUM, CMPX, CMPY, XANY, XEOY, XEOR, MSKX, DIFF, MAX, MAXM, and U are excluded as sink registers.

4. U is excluded as a source register in the STEP mode.

5. When DATA (and SUM, DIFF on B 1710) is designated as a source, CMND, and DATA are excluded as sinks.

6. On the B 1710 when A, M, CP, or DATA is designated as a source, all 4-bit registers are prohibited as sinks.

7. On the B 1720, when U or DATA is designated as a source and when the next micro-instruction is to be obtained from main memory, M is excluded as a sink.

Scratchpad Move

OP CODE 0010	REGISTER GROUP# 0 . . . 15	REGISTER SELECT# 0 . . . 3	DIRECTION 0 - TO SCRATCHPAD 1 - FROM SCRATCHPAD	SCRATCHPAD WORD 0 - LEFT WORD 1 - RIGHT WORD	SCRATCHPAD WORD ADDRESS 0 . . . 15
0 3	4 7	8 9	10	11	12 15

This micro-instruction moves the contents of the register (scratchpad) to the scratchpad (register). If the move is between fields of unequal lengths, the data is right-justified with left (most-significant) zero bits supplied or the data is truncated from the left, whichever is appropriate.

The contents of the source register are unchanged.

EXCEPTIONS

1. When the M register is used as a sink, the operation is changed to a bit-OR which modifies the next micro-instruction. It does not modify the micro-instruction as stored in memory.

2. BICN, FLCN, XYCN, XYST, INCN, READ, WRIT, SUM, CMPX, CMPY, XANY, XORY, XEOY, MSKX, MSKY, DIFF, MAXS, MAXM and U are excluded as sink registers.

3. WRIT, CMND (and CPU, READ on B 1710) are excluded as source registers.

4. U is excluded as a source in STEP mode.

5. On the B 1710 M as a source results in a transfer of 24 zeros.

Scratchpad Relate FA

OP CODE 0000 1000	RESERVED 000	SIGN OF SCRATCHPAD 0 = POSITIVE 1 = NEGATIVE	LEFT HALF ADDRESS OF A SCRATCHPAD WORD 0 . . . 15
0 7	8 10	11	12 15

This micro-instruction replaces the contents of the FA register by the binary sum of FA and the left half of the specified scratchpad word.

Neither overflow nor underflow of FA is detected. The value of FA may go through its maximum value or its minimum value and wrap around.

Set CYF

OP CODE 0000 0000 0110	SET VARIANTS (V) 1, 2, 4, 8
0 — 11	12 — 15

This micro-instruction sets the carry flip-flop as specified by the variants.

V = 1 Set CYF to 0
2 Set CYF to 1
4 Set CYF to CYL (carry total from sums)
8 Set CYF to CYD (carry borrow from difference)

NOTES

1. CYL is generated under the control of the length in CPL.

2. CYF is an input to the arithmetic logic along with the X and Y registers. CYF is the left-most bit of the CP portion of the C register.

Shift/Rotate T Left

OP CODE 1010	DESTINATION REGISTER GROUP # 0...15	DESTINATION REGISTER SELECT# 0...3	SHIFT/ROTATE 0 - SHIFT 1 – ROTATE	SHIFT/ROTATE BIT COUNT 0...24
0 — 3	4 — 7	8 — 9	10	11 — 15

This micro-instruction shifts (rotates) register T left by the number of bits specified and then moves the 24-bit result to the sink register. If the move is between registers of unequal lengths, the data is right-justified, with data truncated from the left.

The contents of the T register are unchanged unless it is also the sink register.

Zero fill on the right and truncation on the left occurs with the shift operation. ROTATE is an end-around shift with no truncation or fill.

If the value of the SHIFT/ROTATE COUNT as given in the instruction is 0, the value given in CPL is used.

EXCEPTIONS

1. When the M register is used as a sink register, the operation is changed to a bit-OR which modifies the next micro-instruction. It does not modify the micro-instruction as stored in memory.

2. BICN, FLCN, XYCN, XYST, INCN, READ, WRIT, SUM, CMPX, CMPY, XANY, XEOY, XORY, DIFF, MAXS, MAXM and U are excluded as sink registers.

Shift/Rotate XY Left/Right

OP CODE 0000 0101	SHIFT/ROTATE VARIANT 0 - SHIFT 1 - ROTATE	SHIFT/ROTATE DIRECTION VARIANT 0 - LEFT 1 - RIGHT	SHIFT/ROTATE BIT COUNT 0...48
0 7	8	9	10 15

This micro-instruction shifts (rotates) register X and Y left (right) by the number of bits specified. The register X is the left-most (most-significant) half of the concatenated 48-bit XY register. Only a count of one may be specified on the B1710 for the concatenated XY register.

Zero fill on the right and truncation on the left occurs with the left shift. Zero fill on the left and truncation on the right occurs with the right shift.

If the value of the SHIFT/ROTATE COUNT as given in the micro-instruction is 0, the operand is shifted/rotated by the amount determined by CPU as follows:

CPU	SHIFT/ROTATE COUNT
00	1 bit
01	4 bits
10	Undefined
11	8 bits (available only on B 1720 systems)

NOTE

The shift by CPU option is available only on B 1720 systems.

Shift/Rotate X/Y Left/Right

OP CODE 0000 0100	SHIFT/ROTATE VARIANT 0 - SHIFT 1 - ROTATE	SHIFT/ROTATE DIRECTION 0 - LEFT 1 - RIGHT	X/Y VARIANT 0 - X REG 1 - Y REG	SHIFT/ROTATE BIT COUNT 0 . . . 24
0 7	8	9	10	11 15

This micro-instruction shifts (rotates) register X or Y left or right by the number of bits specified.

Zero fill on the right and truncation on the left occurs with the left shift. Zero fill on the left and truncation on the right occurs with the right shift.

If the value of the SHIFT/ROTATE COUNT as given in the micro-instruction is 0, the operand is shifted (rotated) by the amount determined by CPU as follows:

CPU	SHIFT/ROTATE COUNT
00	1 bit
01	4 bits
10	Undefined
11	8 bits (not available on B 1710 systems)

NOTE

The shift by the CPU option is available on B 1720 systems only.

Skip When

OP CODE 0110	REGISTER ROW # 0 . . . 15	REGISTER COLUMN # 0 . . . 1	SKIP TEST VARIANTS (V) 0 . . . 7	MASK 0 . . . 15
0 3	4 7	8	9 11	12 15

This micro-instruction tests only the bits in the register that are referenced by the 1 bits in the mask and ignores all others. It then performs the actions specified below. Exception: If V = 2 or V = 6, it compares all bits for an equal condition.

V = 0 If any of the referenced bits are l's, the next micro-instruction is skipped.

1 If all the referenced bits are 1's, the next micro-instruction is skipped.

2 If the register is equal to the mask, skip the next micro-instruction.

3 This is the same as V = 1, but the referenced bits are also cleared to 0 without affecting the non-referenced bits.

4 If any of the referenced bits are 1's, the next micro-instruction is not skipped.

5 If all the referenced bits are 1's, the next micro-instruction is not skipped.

6 If the register is equal to the mask, the next micro-instruction is not skipped.

7 This is the same as V = 5, but the referenced bits are also cleared to 0 without affecting the non-referenced bits.

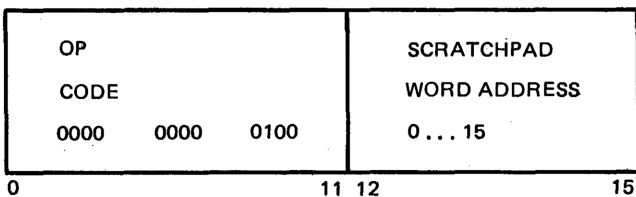
NOTES AND RESTRICTIONS

1. If the mask equals 0000 the ANY result is false. The skip is made for V = 0 and is not made for V = 4. If the mask equals 0000, the ALL result is true. The skip is made for V = 5 and V = 7 and is not made for V = 1 and V = 3.

2. BICN, FLCN, XYCN, XYST, and cannot be cleared with V = 3 or V = 7. However, they can be tested.

Store F Into Doublepad Word

(Available on B 1720 systems only.)



This micro-instruction moves the contents of the FA and FB registers to the designated scratchpad word. FA is transferred to the A half of the scratchpad word, and FB (which contains FL, FT, and FU) is transferred to the B scratchpad word.

The contents of FA and FB remain unchanged.

Swap F with Doublepad Word

OP CODE 0000 0111	DESTINATION 48-BIT SCRATCHPAD WORD 0...15	SOURCE 48-BIT SCRATCHPAD WORD 0...15
0 — 7	8 — 11	12 — 15

This micro-instruction moves the contents of the FA and FB registers to a hardware holding register. It then moves the contents of the left and right word of the source scratchpad word to the FA and FB register respectively, and moves the contents of the hardware holding register to the destination scratchpad word.

Swap Memory

(Available on B 1720 systems only)

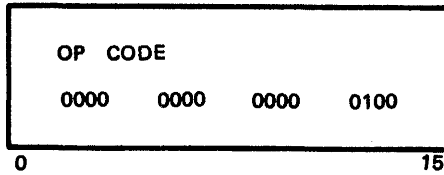
OP CODE 0000 0010	REGISTER # 00 = X 01 = Y 10 = T 11 = L	FIELD DIRECTION 0 - POSITIVE 1 - NEGATIVE	MEMORY FIELD LENGTH 0...24
0 — 7	8 — 9	10	11 — 15

This micro-instruction swaps data from main memory with the data in the specified register. If the value of the memory field is less than 24, the data from memory is right-justified with left (most-significant) zero bits supplied. The data from the register is truncated from the left before entering memory.

Register FA contains the absolute binary address of the main memory field while the field direction sign and field is given in the instruction.

If the value of the memory field length as given in the instruction is 0, the value given in CPL is used.

Transfer Control



This micro-instruction moves the 24-bit value from the L register to the MBR register; moves the least significant 4 bits from the T register to the TOPM register; and moves the most significant 20 bits from the T register to the A register, truncating the left most 6 bits of the source.

MICRO-INSTRUCTION TIMING

Table B-5: Micro-Instruction Timing

B1710 Notes	B1710 Clocks	Micro-Instructions	B1720 Clocks	B1720 Notes
	2	BIAS	1	
	1	BIND	3	
	2	BIT TEST BRANCH FALSE	1	
	2	BIT TEST BRANCH TRUE	1	
	4	BRANCH	1	1
1	5	CALL	5	
	2	CASSETTE CONTROL	1	
	-	CLEAR REGISTERS	1	
	4	COUNT FA/FL	1	
	-	DISPATCH	6/5	
	3	EXTRACT FROM REGISTER T	1	
	2	FOUR-BIT MANIPULATE	1	
	2	HALT	1	
	-	LOAD F FROM DOUBLEPAD WORD	1	
	2	MONITOR	1	
	2	MOVE 8-BIT LITERAL	1	
	6	MOVE 24-BIT LITERAL	2	
	2	NO OPERATION	1	
3	6	NORMALIZE X	1	2
	-	OVERLAY CONTROL MEMORY	5	3
	8	READ/WRITE MEMORY	5/4	4
	-	READ/WRITE MSM	6	
2	2	REGISTER MOVE	1	
2	2	SCRATCHPAD MOVE	1	
	4	SCRATCHPAD RELATE FA	1	
	2	SET CYF	1	
	3	SHIFT/ROTATE REGISTER T LEFT	1	
4	6	SHIFT/ROTATE XY LEFT/RIGHT	1	2
	3	SHIFT/ROTATE X/Y LEFT/RIGHT	1	2
	2	SKIP WHEN	1	
	-	STORE F INTO DOUBLEPAD WORD	1	
	10	SWAP F WITH DOUBLEPAD WORD	2	
	-	SWAP MEMORY	4	5

B 1710 Notes

The basic clock of the B 1710 is 4 megahertz.

1. This includes the fetch of the called micro-instruction.

2. For BCD result register moves, there are three clocks.

3. There are six clocks per bit plus one additional clock.

4. Only a value of one bit is allowed in the B 1710.

B 1720 Notes

The basic clock of the B 1720 is 6 megahertz.

1. If the relative address is not within control memory (therefore in main memory), there are two clocks.

2. There is one clock per bit.

3. There are five clocks per 16 bits (one micro-instruction) plus five clocks.

4. READ is five clocks until the processor receives the data. WRITE is four clocks until the processor is released. Some instructions may be performed during the processor READ or WRITE command times if they immediately follow the READ or WRITE commands: this is called "concurrency". Consecutive READ or WRITE commands operate at MAIN MEMORY READ cycle speed (four clocks) or WRITE cycle speed (six clocks) respectively.

5. The data is presented to the processor and is released in one MAIN MEMORY READ cycle. Concurrent execution of certain micro-instructions is performed if they immediately follow the SWAP command. The WRITE portion of the SWAP command is begun and performed in parallel to the READ portion, and main memory is not available for the duration of a WRITE cycle. For consecutive main memory commands, refer to note 4.

APPENDIX C: RESERVED WORDS AND SYMBOLS

Note: Several elements in the following list will not appear elsewhere in this manual, being in the compiler for future development or debugging purposes.

.
<
(
+
*
)
;
¬
-
/
,
–(underscore)
>
#
@
‘ (apostrophe)
=
”
{
A
ABSOLUTE
ADD
ADDRESS
ADJUST
ALL
ALLCODE
AMPERSAND
ANALYZE.CODEFILE
AND
ANY
ANY.INTERRUPT
ARCHITECTURE.NAME
AS
ASSIGN
ASTACK
AT
ATTRIBUTE
BACKWARD
BASE.LIMIT
BASE.ZERO
BEGIN
BIAS
BICN
BIT
BITS
BR
BRANCH
BRANCH.EXTERNAL
BY
B710
CA
CALL
CALL.EXTERNAL
CARRY
CASSETTE
CAT
CB
CC
CD
CHARACTER
CHECK
CLEAR
CMND
CMPX
CMPY
CODE.SEGMENT
CODE.SEGMENT.NUMBER
COMPILE
COMPILER.LEVEL
COMPLEMENT
CONSOLE.SWITCHES
CONSTANT
CONTROL
COUNT
CP
CPL
CPU
CYD
CYF
CYL
DATA
DATA.LENGTH
DATA.TYPE
DATA.USAGE
DEBUG
DEC
DECK
DECLARE
DEFINE
DEFINE.VALUE
DETAIL
DIFF
DIFFERENCE
DISPATCH
DOLLAR
DOUBLE
DOWN
DUMP
ELSE
EMIT.RETURN.TO.EXTERNAL
END
EOR
EQL
ERROR.FILE
EXIT
EXPAND
EXTERNAL
EXTRACT
F
FA
FA.POINTS
FALSE
FB
FINI
FIXED
FL
FLC
FLCN
FLD
FLE
FLF
FOR
FORCE
FORWARD
FRAME
FROM
FT
FU
GEQ
GISMO.LEVEL
GO
GTR
HALT
HARDWARE.TYPE
HEADINGS
HEX.SEQUENCE.NUMBER
HI.PRIORITY
HIPRI
IF

INC
INCLUDE
INCN
INTERRUPT
INTO
JUMP
L
LA
LANGUAGE.EXTENSION
LB
LC
LD
LE
LEFT
LENGTH.BETWEEN.ENTRIES
LEQ
LF
LINES.PER.PAGE
LIST
LIST.NOW
LIST.PATCHES
LISTALL
LISTP
LIT
LOAD
LOAD.MSMA
LOAD.SMEM
LOCAL.DEFINES
LOCATION
LOCK
LOCKED
LOCKOUT
LR
LSBX
LSBY
LSS
LSUX
LSUY
M
M.MEMORY.BOUNDARY
MACRO
MAKE.SEGMENT.TABLE
MAP
MAXIMUM
MAXM
MAXS
MBR
MCP.LEVEL
MERGE
MICRO
MINIMUM
MINUS
MOD
MONITOR
MOVE
MSBX
MSKX
MSKY
MSMA
MSML
NEQ
NEW
NEWSEGMENT
NO
NO.DEVICE
NODEVICE
NOP
NOPS
NORMALIZE
NOT
NULL
OLD.LISTING.FORMAT
OLDIPB
OR
OVERLAY
PAGE
PAGE.NUMBERS
PARAMETER.BLOCK
PASS.END
PLUS
POINT
PORT
PROGRAM.LEVEL
PROTECT
READ
REDUNDANT.CODE
RELEASE
REMAPS
RESERVE.SPACE
RESET
REVERSE
RIGHT
ROTATE
S
S.MEMORY.LOAD
SEGMENT
SEGMENT.COUNT
SEQ
SET
SFL
SFU
SHIFT
SINGLE
SKIP
SPACE
START
STOP
STORE
SUB.TITLE
SUBSET
SUBTRACT
SUM
SUPPRESS
SWAP
SO
SOA
SOB
S1
S1A
S1B
S10
S10A
S10B
S11
S11A
S11B
S12
S12A
S12B
S13
S13A
S13B
S14
S14A
S14B
S15
S15A
S15B
S2
S2A
S2B
S3
S3A
S3B
S4
S4A
S4B
S5
S5A
S5B
S6
S6A
S6B
S7
S7A
S7B
S8
S8A
S8B
S9
S9A
S9B
T
TA
TABLE
TAPE
TAS
TB
TC
TD
TE
TEST
TF
THEN
TITLE
TO
TODAYS.DATE
TODAYS.TIME
TOPM
TRACE
TRANSFER.CONTROL
TRUE
U
UNIT
UNLOCKED
UP
VALUE
VOID
WHEN
WITH
WRITE
WRITE.STRING
X
XANY
XCH
XEOY
XORY
XREF
XREF.ALL
XREF.LABELS
XREF.NAMES
XREF.REGISTERS
XREF.ZIP
XY
XYCN
XYST
Y

INDEX

Item	Page
A Register	7-7
A Stack	7-7
Add Scratchpad Micro-Instruction	8-2
Adjust Location Statement	5-1, 8-3
Ampersand Cards	A-5, A-6
AND Statement	8-4, 8-5
Any.Interrupt Bit	7-12
Architecture.Name	8-6
Arithmetic Expressions	3-9, 3-10
Array Declarations	6-2
Array Group Items	6-5
Arrays: Maximum Size	6-1
Assign Statement	8-6
Attribute	8-6
Base Register	7-6
Base.Zero	6-3
Begin Statement	8-7, 8-8
Begin/End Code Blocks	8-7
Bias Micro-Instruction	B-5
Bias Statement	8-9, 8-10
BICN Register	7-11
Bit Data Fields	6-1
Bit Strings	3-5, 3-6
Bit Test Branch False Micro-Instruction	B-5, B-6
Bit Test Branch True Micro-Instruction	B-6
BR Register	7-6
Branch Micro-Instruction	B-6
Branch.External Statement	8-11
C Register	7-7
CA Register	7-7
Call Micro-Instruction	B-7
Call Statement	8-12
Call.External Statement	8-13
Card Terminator	3-4, 3-5
Carry Micro-Instruction	7-9
Carry Statement	8-14
Cassette Control Micro-Instruction	B-7
Cassette Control Statement	8-15
CB Register	7-7
CC Register	7-7, 7-12
CD Register	7-7, 7-12
Character Data Fields	6-1
Character Strings	3-7
Clear Registers Micro-Instruction	B-8

INDEX (Cont)

Item	Page
Clear Statement	8-16
CMND Register	7-10
CMPX Register	7-8
CMPY Register	7-8
Code.Segment Statement	5-1, 5-3 through 5-7
Combinatorial Logic	7-8
Compiler Control Cards	A-1
Compiler Files	A-6, A-7
Compiler.Level	8-6
Complement Statement	8-18, 8-19
Condition Registers	7-11
Condition Registers Summary	B-2
Console Interrupt	7-12, 7-13
Console.Switches Register	7-10
Constant Registers	7-10
Correspondence Table	5-2, 8-59
Count FA/FL Micro-Instruction	B-8
Count Statement	8-20, 8-21
CP Register	7-7
CPL Register	7-7, 7-8, 8-9, 8-10
CPU Register	7-7, 7-8, 8-9, 8-10
CYD Register	7-9, 7-11
CYF Register	7-7, 7-11
CYL Register	7-11
Data Register	7-10
Data Types	6-1
Data.Length	3-8, 6-8, 6-10
DEC Statement	8-22
Declarations Maximum Number	6-1
Declare Statement	6-1, 6-6 through 6-11
Decrement Statement	8-22
Define Statement	8-23
Define.Value Statement	8-24
Diff Register	7-9
Digit	3-1
Dispatch Micro-Instruction	B-9, B-10
Dispatch Statement	8-25, 8-26
Dollar Cards	A-2, A-5
Dummy	6-6
Elementary Items	6-4
Else Statement	8-27, 8-28
Emit.Return.To.External Statement	8-29
End Statement	8-30
EOR Statement	8-31, 8-32
Exchange Scratchpads with F Register Statement	8-99

Item	Page
Exit Statement	8-33
External Dollar-Option	8-11, 8-13
Extract From T Micro-Instruction	B-10
Extract Statement	8-34, 8-35
F Register	7-6
FA.Points Statement	8-36
Field Length Conditions Register	7-11, 7-13
Filler	6-5, 6-6, 6-7
FINI Statement	8-37
Fixed Data Fields	6-1
FLCN Register	7-11, 7-13
Four-Bit Manipulate Micro-Instruction	B-10, B-11
FU Register	8-10
Function Box	7-8
GISMO.Level	8-6
GO TO Statement	8-38
Group Items	6-4
Halt Micro-Instruction	B-11
Halt Statement	8-39
I/O Service Request Interrupt	7-12
Identifier	3-2, 3-3
IF Statement	8-40 through 8-45
IF Statement	8-7
INC Clause in Read/Write Memory Statements	8-74, 8-96, 8-97
INC Statement	8-46
INCN Register	7-11, 7-13
Increment Statement	8-46
Input/Output Registers	7-10
Inter-Firmware Communications	8-94
Interrupt Conditions Register	7-11, 7-13
Jump Statement	8-47
Key Concepts Alphabetic List	7-2 through 7-5
L Register	7-6
Label	3-3, 3-4
Label Addresses	5-1
Length.Between.Entries	3-9, 6-10
Letter	3-1
Level Numbers	6-4, 6-8
Limit Register	7-6
Lit Statement	8-48

INDEX (Cont)

Item **Page**

Literals . . . 3-8
Load F From Double Scratchpad Word Micro-Instruction . . . B-11
Load Scratchpad Statement . . . 8-49
Load Statement . . . 8-49
Load.MSMA Statement . . . 8-50, 8-51
Load.SMEM Statement . . . 8-52
Local.Defines Statement . . . 8-53, 8-54
LR Register . . . 7-6
LSUX . . . 7-12
LSUY . . . 7-11

M Register . . . 7-6
M.Memory.Boundary Statement . . . 8-61
MACRO Declaration Statement . . . 8-55, 8-56
MACRO Reference . . . 8-57, 8-58
MACROS . . . 8-55, 8-58
Make.Segment.Table.Entry Statement . . . 8-59
MAXM Register . . . 7-10
MAXS Register . . . 7-10
MBR Register . . . 7-7
MCP.Level . . . 8-6
Micro Statement . . . 8-60
Micro-Instruction Addresses . . . 7-7
Micro-Instruction Decoding . . . 7-6
Micro-Instruction Timing Chart . . . B-24, B-25
Micro-Instruction Summary Table . . . B-3, B-4
MIL Statements Alphabetic List . . . 8-1
Monitor Micro-Instruction . . . B-12
Monitor Statement . . . 8-62
Move Statement . . . 8-63, 8-64
Move 24-Bit Literal Micro-Instruction . . . B-13
Move 8-Bit Literal Micro-Instruction . . . B-12
MSBX . . . 7-11
MSKX Register . . . 7-8
MSKY Register . . . 7-8

NO Operation . . . 8-65
NO Operation Micro-Instruction . . . B-13
NOP Statement . . . 8-65
Normalize Statement . . . 8-66
Normalize X Micro-Instruction . . . B-13
Null Register . . . 7-10
Number . . . 3-5

Operator Precedence . . . 3-10
OR Statement . . . 8-67, 8-68
Overlay Control Memory Micro-Instruction . . . B-14

INDEX (Cont)

Item **Page**

Page Statement 8-70
Parity Error 7-12
Parity Error Interrupt 7-13
Physical, Label 5-1
Point 3-1
Point FA Clause 8-74, 8-96
Point Statement 8-71
Point.Label 3-3
Port Device Interrupts 7-13
Program, Level Statement 8-72

Read Memory Statement 8-73, 8-74
Read Out of Bounds Interrupt 7-12, 7-13
Read/Write Memory Micro-Instruction B-14, B-15
Read/Write MSM Micro-Instruction B-15
Redundant.Code Statement 8-75
Register Addressing Table B-1
Register Bit Numbering Convention 7-1
Register Move Micro-Instruction B-16
Registers, Alphabetic List . . . . . 7-2 through 7-5
Regular.Label 5-1
Remap Items . . . . . 6-8 through 6-11
Remap/Reverse Combination 6-6
Remapping Structured Data 6-5
Remaps 6-3
Reserve.Space Statement 8-76
Reserved Word List C-1, C-2
Reset Statement 8-77
Result Registers 7-8, 7-9
Reverse 6-2, 6-5, 6-10
Rotate Statements B-18, B-29, 8-84

S.Memory.Load Statement 8-87
Scratchpad 7-9
Scratchpad Move Micro-Instruction B-17
Scratchpad Relate FA Micro-Instruction B-17, B-18
Segment Dictionary 5-2
Segment Statement 5-1, 5-2
Set CYF Micro-Instruction B-18
Set Statement 8-80, 8-81
SFL Register 8-9
SFU Register 8-10
Shift/Rotate T Left Micro-Instruction B-18, B-19
Shift/Rotate T Statement 8-82, 8-83
Shift/Rotate X/Y Left/Right Micro-Instruction B-20
Shift/Rotate X/Y/X/ Statement 8-84
Shift/Rotate XY Left/Right Micro-Instruction B-19

Item	Page
Skip Statement	8-85, 8-86
Skip When Micro-Instruction	B-20, B-21
Source Image Format	9-1
State Light	7-12
Statements: Alphabetic List	8-1
Store F Into Double Scratchpad Word Statement	8-88
Store F Into Doublepad Word Micro-Instruction	B-21
Store Statement	8-88
Structured Declarations	6-4, 6-7
Sub.Title Statement	8-89
Subtract Scratchpad Statement	8-90
Sum Register	7-9
Swap F With Doublepad Word Micro-Instruction	B-22
Swap Memory Micro-Instruction	B-22
Swap Statement	8-91
T Register	7-6
Table Statement	8-92
TAS Register	7-7
Timer Interrupt	7-12
Title Statement	8-93
TOPM Register	7-7
Transfer.Control Micro-Instruction	B-23
Transfer.Control Statement	8-94
U Register	7-10
Underscore	3-1
Unique.Label	3-3
Verlay Control Memory Statement	8-69
Write Memory Statement	8-95, 8-96
Write.String Statement	8-97, 8-98
Write/Swap Out of Bounds Interrupt	7-12, 7-13
Write/Swap Out of Bounds Override Flag	7-12, 7-13
X Register	7-6, 7-8
X/Y Conditions Register	7-11
X/Y States Register	7-11
XANY Register	7-8
XCH Statement	8-99
XEOY Register	7-8
XORY Register	7-8
XYCN Register	7-11
XYST Register	7-11, 7-12
Y Register	7-6, 7-8
24-Bit Function Box	7-8

Printed in U.S.A. May 1977 1072568