DISTRIBUTION LIST

B1800/B1700 SOFTWARE PRODUCT SPECIFICATIONS

DETROIT

J. Garren - Prod. Mgmt.
P. Gonzales - Prod. Mgmt.
H. Woodrow - Int'l Group P
C. Kunkelmann - BMG
J. McClintock - CSG
D. Dahm - Corp. Eng.
Dir., Pgmg. - SSG
M. Dowers - Int'l FE
D. Hill - TC, BM, & SS

U.S. AND EUROPE

D. Cikoski - (Plymouth)
J. H. Pedersen (Plymouth)
W. E. Feeser (Austin)
J. Berta (Downingtown)
W. Minarcik (Paoli)
G. Smclnik (Paoli)
M. E. Ryan (Tredyffrin)
T. Yama - F&SSG (McLean)
J. Peterack - F&SSG (McLean)
A. Kosla - F&SSG (McLean)
A. LaCivita - F&SSG (McLean)
L. Guell - F&SSG (McLean)
R. Sutton - F&SSG (McLean)
L. DeBartelo - WADC (Irvine)
R. Cole (Pasadena)
H. M. Townsend (Pasadena)
N. Cass - Pat. Atty. (Pasadena)
D. C. Swanson (Mission Viejo)
J. Lowe (Mission Viejo)
J. C. Allan (Glenrothes)
W. McKee (Cumbernauld)
B. Higgins (Livingston)
Mgr, NPSGrp (Ruislip)
E. Norton (Middlesex)
B. Hammersley (Croydon)
J. Gerain (Pantin)
J. Cazanove (Villers)
J. C. Wery (Liege)
R. Bouvier (Liege)
G. LeBlanc (Liege)
C. J. Tooth - SSG (London)
J. Dreystadt (Wayne)

SANTA BARBARA PLANT

S. C. Schmidt
J. Hale
R. Shobe
K. Meyers
A. van der Linden
T. Cardona
R. Bauerle
J. Henige
E. Yardi
C. Stover
L. Sweeney - 2
G. Hammond - 3
J. Morrison - 6

Distribution list current as of 04/09/81

DOCUMENT TRANSMITTAL

PAGE ____ OF ____

CONTROL DOC. None

SYSTEM B1800/B1700

PL FILE NO. ____

PROJ. NO. ____ COORD. ____ CHKR. J Hurley TO RECORDS ____ DISTRIBUTION 4-16-71

DOC PFX	DOCUMENT NUMBER	REV	UNIT NUMBER	DL NO.	PL FILE NO.	ENG. RECORDS ONLY: QTY REV PAGE	COPIES EACH PAGE	SEPIAS	MT REQ'D	CD DK REQ'D		REMARKS
P.S.	2222 3069	B	COBOL 74 S-LANGUAGE			102	63					
DOCUMENT TRANSMITTAL												

SBP 1085 REV. 9-71

Burroughs Corporation

COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

2222 3069

COBOL74 S-LANGUAGE

PRODUCT SPECIFICATION

REV LTR	REVISION ISSUE DATE	APPROVED BY	REVISIONS
A	1/10/80		Original Issue - MARK 9.0 Release
B	4/14/81		Changes for MARK 10.0 Release 3-3 Updated "MISCELLANEOUS" list. 3-54 Added "5-10" RS.RMSG.P2 Values. 3-56 Added "WAIT COMMUNICATE REPLY" instruction. 3-60 Updated "FILE STATUS CODES" for "FILE STATUS" instruction. Added "5-10" Communicate Reply Values. 3-62 Added "BINARY SEARCH ASCENDING LOW" instruction. 3-63 Added "BINARY SEARCH ASCENDING HIGH" instruction. 3-64 Added "BINARY SEARCH DESCENDING LOW" instruction. 3-65 Added "BINARY SEARCH DESCENDING HIGH" instruction.

TABLE OF CONTENTS

ADR	NAME	PIC
0	SW1	9 CMP
:	:	:
7	SW8	9 CMP
49	DM-STATUS	9 (2)

Figure 1.2 Special Registers

CONTAINER SIZE

Container size is a field size (in number of bits) necessary to contain the maximum value required for that field. For example a container size of 5 bits allows a field value to house 32-bit addresses (0-31). All container sizes in the COBOL74 S-machine are of fixed length and are:

OP	opcode	9 bits
LEN	data length op	14 bits
DISP	data displacement	20 bits
SEG	data segment number	10 bits
BADDR	branch address	21 bits
DADDR	data address	20 bits

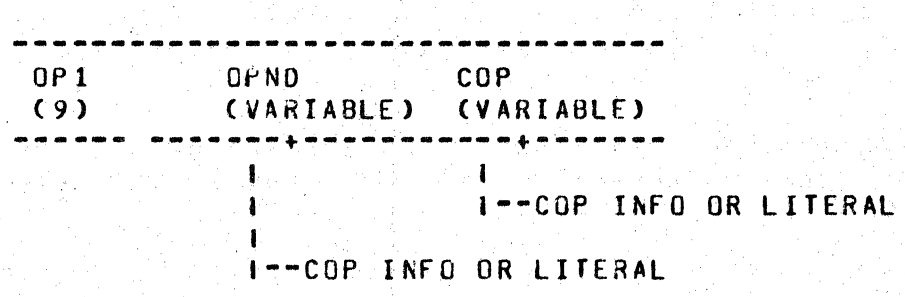
PROGRAM PARAMETERS

The parameters pertaining to a particular program are listed below:

1. Base relative address of Data Segment Zero
2. Data Segment Zero address of Collate Table
3. Base relative address of Perform Stack
4. Maximum size of Perform Stack

S-INSTRUCTION FORMAT

Each COBOL74 S-instruction consists of an S-operator followed by arguments consisting of a variable number of bits. The format and interpretation of these arguments is specified by the S-operator and is described in detail by the specification of the individual operators. An example of one such instruction format is illustrated below.



S-OPERATORS

All S-operators are encoded in a 9-bit S-operator denoted as OP1.

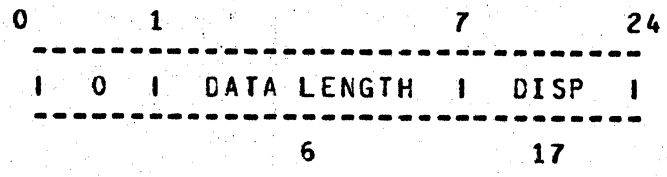
COP AND OPND

A COP is an inline descriptor pointing to a data item or a literal. An operand (OPND) may be an inline COP or an inline literal. All data items are placed in data segment zero.

There are three types of descriptors: a short COP, a long COP with segment numbers, and a long COP without segment numbers.

Short COP

Format:

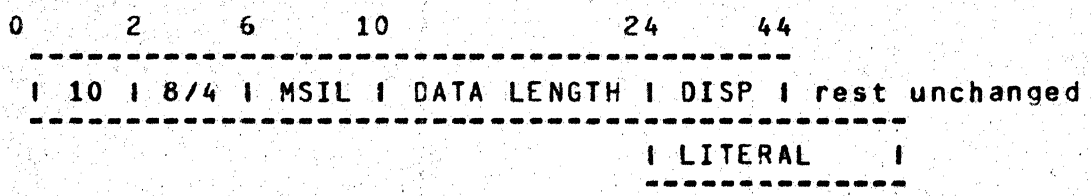


A short COP cannot be generated for subscripts, indexes, IPC data, or literals. Because a short COP does not contain information on data types, it can be used only with the following S-operators.

-MVA	-CPN
-MVN	-CPZ
-MVZ	-INC
-CPA	-INC1

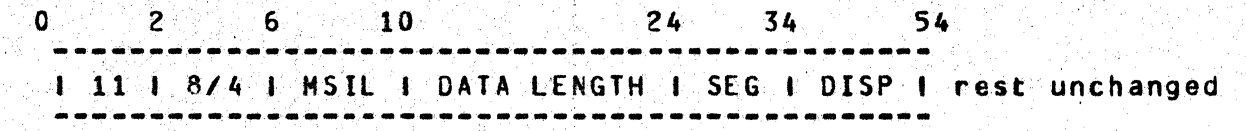
Long COP With No Segment Number

Format:



This COP can be used for any data except IPC data. A description of the fields is found in the following section on TABLE HANDLING.

Long COP With Segment Number



This COP cannot be used for literals. It is only for IPC data. Descriptions of the fields are provided in the following section on INLINE DESCRIPTORS.

COBOL74 INLINE DESCRIPTORS

An inline descriptor must provide for the following capabilities.

- Multidimensional tables with unlimited subscripting/indexing (the real limit is 48).
- Runtime bounds-checking on each dimension of a table.
- Variable length data items with an OCCURS DEPENDING clause.
- Shared data addressing.
- The specification of CODE-SET.

A variable length data item may be declared in COBOL74 by using an OCCURS DEPENDING clause.

For example:

```
01 A.
  02 B PIC X.
  02 C OC 5 TO 10 TIMES DEPENDING ON V.
   03 D PIC X OC 20 TIMES.
```

In this data structure, A is a variable length item whose length is determined as follows:

LENGTH [A] = LENGTH [B] + (V * LENGTH [C])

where 5 LEQ V LEQ 10

When A is referenced, only that part of it defined by the calculated length is used in any operation.

When C or D are referenced, their length is not variable -- but V must be bound-checked to verify that the requested table element exists.

The restrictions:

- a data description entry that contains an OCCURS DEPENDING clause may be followed only by entries that are subordinate to it.
- The OCCURS DEPENDING clause cannot be specified in a data description entry that describes an item whose size is variable.

These restrictions ensure that the description of the first subscript/index associated with a variable is the only one that contains an OCCURS DEPENDING clause.

IMPLEMENTATION STRATEGY

COBOL74 S-language employs a continue flag to tell the interpreter that more COP information is specified.

The same bounds-checking information is needed when referencing either a variable length item or the first dimension of a subscripted variable contained in a variable length item. Dimension information required for the subsequent dimensions is different from that which is required for the first dimension.

When a variable length item is referenced, the COP of the OCCURS DEPENDING operand is provided following the COP of data-name. The operand is bound-checked to verify that it is within the specified integer range.

DATA LENGTH

The data length field is 14 bits wide and contains the length of the data in digits. If this field is the length of a variable length item, the field contains the length of that part of the table which is not variable. The actual length is calculated at runtime by the interpreter (see DEPENDING ATTRIBUTES).

MULTIPLE-ENTRY-FLAG

The MULTIPLE-ENTRY-FLAG is true when indicating that multiple entry attributes are associated with this entry and false when indicating otherwise. When true, the next entry(s) contain(s) the necessary OCCURS DEPENDING and/or subscripting or indexing attribute information.

SHARED-DATA-FLAG

The SHARED-DATA-FLAG is true when indicating that this entry was a data item passed by a CALL statement of another program and false when indicating otherwise. When true, IPCO-INDEX specifies the location of the IPC.DICTIONARY descriptor for this entry.

LITERAL-FLAG

If this flag is set, then a literal follows after the DATA LENGTH field.

DEPENDING-FLAG

The DEPENDING-FLAG is true when indicating that an OCCURS DEPENDING operand is associated with this variable and false when indicating otherwise. When true, the bounds-checking and factor information follow in the entry. This is followed by any subscripting or indexing information that may be necessary.

DEPENDING ATTRIBUTES

When the DEPENDING-FLAG is true, the attributes of the DEPENDING operand are encoded in LOWER-BOUND-0, UPPER-BOUND-0, and FACTOR-0, which are binary fields. LOWER-BOUND-0 and UPPER-BOUND-0 contain the integer range of the DEPENDING operand while FACTOR-0 contains the digit displacement between elements of the table.

The value of the DEPENDING operand is verified to be greater than or equal to LOWER-BOUND-0 and less than or equal to UPPER-BOUND-0. If the value is outside the range, then an error communicate is issued.

If CONTINUE-FLAG-0 is false, then the value of the DEPENDING operand is multiplied by FACTOR-0, is added to the data length specified in the primary COP entry, and decoding of the multiple entry attributes is terminated. In this case, the data length specified in the COP entry is the fixed part of the table.

If CONTINUE-FLAG-0 is true, then subscripting or indexing information follows. One is subtracted from the value of the DEPENDING operand and is multiplied by FACTOR-0 to form a new upper-bound. In this case FACTOR-1 and UPPER-BOUND-1 are omitted from the entry because their equivalent is represent by FACTOR-0 and NEW.UPPER.BOUND.

SUBSCRIPT-FLAG

If the SUBSCRIPT-FLAG is false, it specifies that indexing information follows and that the factor associated with each dimension of the table is omitted; if it is true, it specifies that subscripting information follows for each dimension of the table.

Factor and upper-bound fields are binary. Factor is the digit displacement between elements of the table. Upper-bound is the maximum digit displacement allowed for this dimension of the table. UPPER-BOUND-0 differs from the other upper-bound fields in that it represents the maximum value of the DEPENDING operand.

SUBSCRIPTING

Subscripting requires a factor and an upper-bound for each dimension of the variable. The continue-flag is true until the last dimension has been reached. When the continue flag is false, decoding is terminated.

Each subscript value is obtained and one is subtracted from it. If the result is less than zero, an error communicate is issued; otherwise, the result is multiplied by its associated factor and the product is compared to the corresponding upper-bound. If the product exceeds the upper-bound, an error communicate is issued; otherwise the product is added to the address displacement.

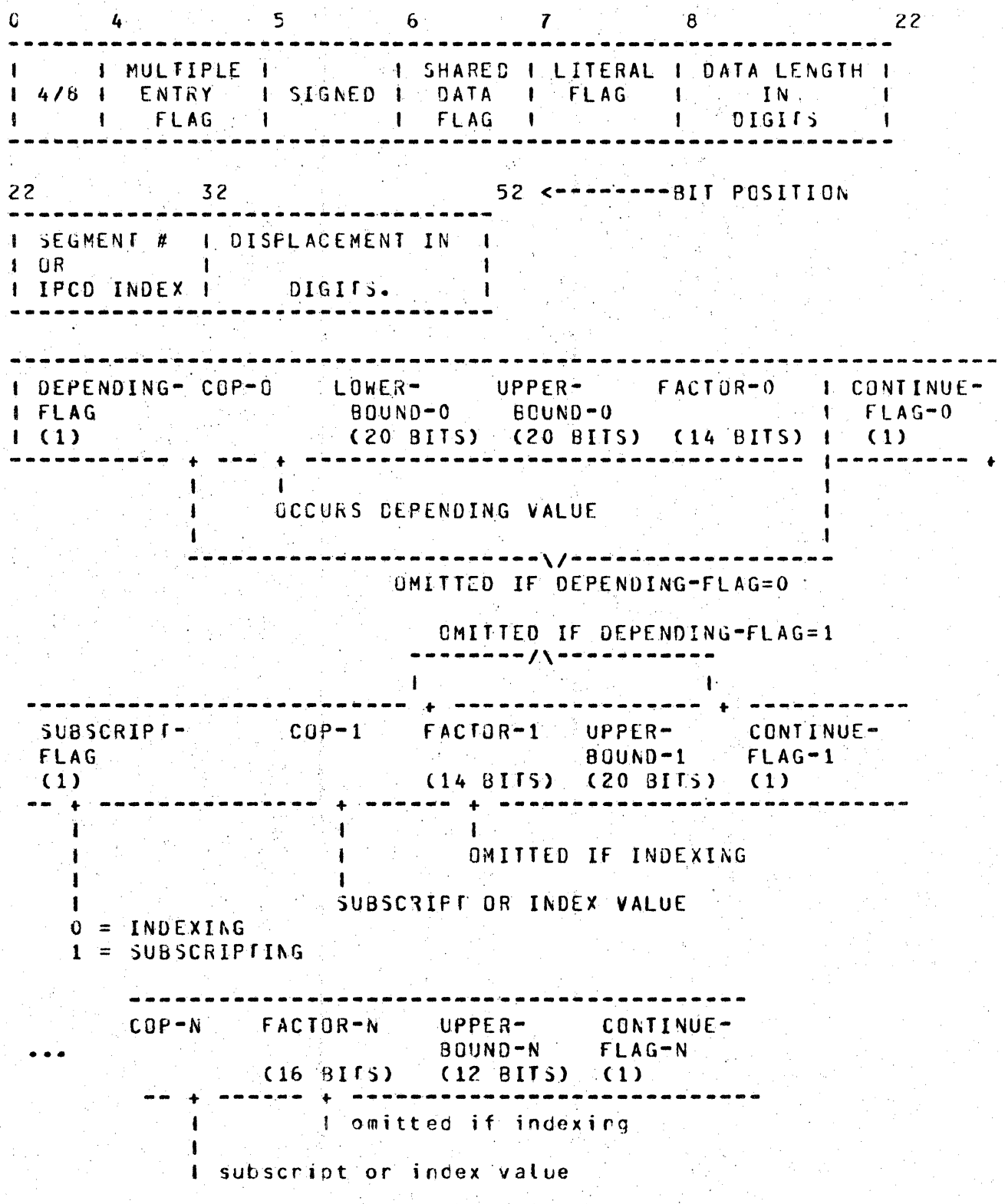
INDEXING

Because an index value is premultiplied by its associated factor, the factor required for subscripting is omitted from the attributes. An upper-bound is required for each dimension of the variable. The continue-flag is true until the last dimension is reached. When the continue-flag is false, decoding is terminated.

Each index value is obtained and if it is less than zero or greater than the upper-bound, then an error communicate is issued; otherwise the value is added to the address displacement. The format of the index consists of a 4-bit sign followed by seven 4-bit decimal digits.

MODIFIED INLINE COP ENTRY FORMAT

The format of an inline COP entry is:



SEGMENT NUMBER

Segment number is expressed in binary and specifies the data segment number of the operand. It is 10 bits in length.

DISPLACEMENT

Displacement is expressed in binary and specifies the digit displacement of the data from the base of the data segment. All data is stored beginning at an address which modulo 4-BIT must equal zero. The container size is 20 bits.

INSTRUCTION SET

ARITHMETIC

Name	Mnemonic	Arguments
INCREMENT	INC	OPND1, COP1
ADD	ADD	OPND1, COP1, COP2
DECREMENT	DEC	OPND1, COP1
SUBTRACT	SUB	OPND1, OPND2, COP1
MULTIPLY	MULT	OPND1, COP1, COP2
DIVIDE	DIV	OPND1, COP1, COP2
DIVIDE SPECIAL	DIVS	OPND1, COP1, COP2
INCREMENT BY ONE	INC1	COP1
DECREMENT BY ONE	DEC1	COP1

DATA MOVEMENT

Name	Mnemonic	Arguments
MOVE ALPHANUMERIC	MVA	COP1, OPND1
MOVE SPACES	MVS	COP1
MOVE NUMERIC	MVN	COP1, OPND1
MOVE ZEROS	MVZ	COP1
CONCATENATE	CAT	N, COP1, OPND0,..., OPNDN
EDIT	EDIT	OPND1, COP1, DADDR
EDIT WITH EXPLICIT MASK	EDTE	OPND1, COP1, MASK

BRANCHING

Name	Mnemonic	Arguments
BRANCH ON OVERFLOW	BOFL	V, BADDR
SET OVERFLOW	SOFL	V
BRANCH UNCONDITIONALLY	BUN	BADDR
PERFORM ENTER	PERF	K, BADDR
PERFORM EXIT	PXIT	K
ENTER	NTR	BADDR
EXIT	XIT	
GO TO DEPENDING	GOTD	COP1, L, DBADDR0,..., DBADDRL
ALTERED GO TO PARAGRAPH	GPAR	DADDR
ALTER	ALTR	DADDR, ACON

CONDITIONAL BRANCHING

Name	Mnemonic	Arguments
COMPARE ALPHANUMERIC	CMPA	OPND1, OPND2, R, BADDR
COMPARE NUMERIC	CMPN	OPND1, OPND2, R, BADDR
COMPARE FOR ZEROS	CMPZ	COP1, R, BADDR
COMPARE FOR SPACES	CMPS	COP1, R, BADDR
COMPARE FOR CLASS	CMPC	COP1, C, BADDR
COMPARE REPEAT	CMPR	OPND1, COP1, R, BADDR

COMPARE COLLATE	CPC	OPND1, COP2, R, BADDR

MISCELLANEOUS

Name	Mnemonic	Arguments
COMMUNICATE	COMM	COP1
LOAD COMMUNICATE REPLY	LDCR	DADDR
WAIT COMMUNICATE REPLY	WTCR	DADDR
CONVERT	CONV	COP1, DADDR, N
MAKE PRESENT	MAKP	COP1, DADDR
HARDWARE MONITOR	HMON	OPND1
FILE STATUS	FIST	RW, COP1
BINARY SEARCH ASCENDING LOW	BSAL	COP1, CPND, KEY.ADDR, COP2
BINARY SEARCH ASCENDING HIGH	BSAH	COP1, CPND, KEY.ADDR, COP2
BINARY SEARCH DESCENDING LOW	BSDL	COP1, OPND, KEY.ADDR, COP2
BINARY SEARCH DESCENDING HIGH	BSDH	COP1, OPND, KEY.ADDR, COP2

CHARACTER STRING HANDLING

Name	Mnemonic	Arguments
DESCRIPTOR SETUP	DSET	DADDR, COP1
INSPECT SETUP	ISET	V1, V2, FLAC, DADDR1, COP1, COP2, DADDR2, OPND1
INSPECT	INSP	V1, DADDR1, DADDR2
STRING	STRI	Z, OPND1, DADDR1, COP1, CPND2, BADDR

DELIMITER SETUP	DLIM	V1, V2, DADDR1, COP1
UNSTRING	UNST	F, M, Z, C, J, BADDR, DADDR1, COP1, DADDR2 ....

INTERPROGRAM COMMUNICATION

Name	Mnemonic	Arguments
IPC DICTIONARY	IPCD	V, COP1, BADDR1

OPTIMIZED OP CODES

Eight operators have been optimized for improved performance: CPA, CPN, CPZ, INC, INC1, MNA, MVN, MVZ. The optimized opcodes include the following data as a part of the opcode.

- The type of data item (8-bit or 4-bit).
- Signed or unsigned data.
- Info about the logicalsize of operands if the op has two operands.

The following notation is used to describe the opcodes.

8 - 8-bit or 4-bit data item.

5 - Signed or unsigned data item.

= True if the LOGICALSIZES of two operands are equal.

> True if the LOGICALSIZE (OPND1) is greater than the LOGICALSIZE (OPND2).

CPA

Opcode: @(1)011100@ CAT SS=

The LOGIALSIZE of the second operand is greater than the LOGICALSIZE of the first operand.

CPN

Opcode: @(1)0110@ CAT 88SS=

The LOGICALSIZE of the second operand is always greater than the LOGICALSIZE of the first operand.

CPZ

Opcode: @(1)01110010@ CAT 8S

INC

Opcode: @(1)001@ CAT 88SS CA @(1)11@

INC1

Opcode: @(1)01111000@ CAT 8S

MVA

Opcode: @(1)001@ CAT 88SS=>

The first operand is the destination, and the second operand is the source.

MVN

Opcode: @(1)010@ CAT 88SS=>

The first OPERAND is the destination, the second operand is the source.

MVZ

Opcode: @(1)01110011@ CAT 8S

ARITHMETIC OPERANDS AND INSTRUCTIONS

In general, operands can have any of the following formats:

- Unsigned 4-BIT.
- Unsigned 8-BIT.
- Signed 4-BIT (sign is MSD).
- Signed 8-BIT (sign over MSD).

Any restrictions concerning the types of operands permitted in an operation are specified under the description of the particular operation.

All fields are addressed by pointing to the most significant bit of the most significant unit, which, in the case of a signed field is the sign.

All fields are considered to be comprised of decimal integers.

The absolute value is stored if the receiving field is unsigned.

Unsigned fields are considered positive.

When signed format is specified for the receiving field of any arithmetic operation, the sign position is set to 1100 for a positive result and to 1101 for a negative result.

4-BIT operands are interpreted in units of 4 bits. When a signed operand is specified, the sign is interpreted as a separate and leading (leftmost) 4-BIT unit which is not included in the statement of length.

8-BIT operands are interpreted in units of 8 bits. When a signed operand is specified, the sign is interpreted as being contained in the leftmost 4 bits of the leftmost 8-BIT unit.

The length of the operand field specifies the number of 4-BIT.

When 8-BIT units are specified for the receiving field of an arithmetic operation, the leftmost 4 bits of each 8-BIT unit, except the unit carrying a sign, is set to 1111.

The value of an 8-BIT unit is carried in the rightmost 4 bits of the unit. Its value is as defined below for the 4-BIT unit. The leftmost 4 bits, except for a sign, are ignored. The value and sign interpretation of a 4-BIT unit is as follows:

Unit	Value	Sign
0000	0	+
0001	1	+
0010	2	+
0011	3	+
0100	4	+
0101	5	+
0110	6	+
0111	7	+
1000	8	+
1001	9	+
1010	UNDEFINED	+
1011	UNDEFINED	+
1100	UNDEFINED	+
1101	UNDEFINED	-
1110	UNDEFINED	+
1111	UNDEFINED	+

In addition and subtraction, results generated when the size of the result field is not sufficient to contain the result, are not specified. When the result field is longer than the length of the result, leading zero units are stored.

In three-address add, three-address subtract, and in multiply, total or partial overlap of the first two operands is permitted. Results generated when the result field totally or partially overlaps either of the operand fields, are not specified.

In two-address add and subtract, total overlap is permitted. Results generated when the result field partially overlaps the first operand field, are not specified. Note that total overlap implies that the two fields are identical.

No overlap of operands or result fields is permitted in divide. Results generated under any condition of overlap are not specified.

ADD THREE ADDRESS

```
*******
* ADD *
*******
```

OP: 08

Format:

```
***************************
* ADD OPND1, COP1, COP2 *
***************************
```

Function:

Algebraically add an addend denoted by OPND1 to an augend denoted by COP1 and store the sum in the field denoted by COP2. OPND1, COP1, and COP2 must be 4-bit items.

SUBTRACT THREE ADDRESS

```
*******
* SUB *
*******
```

OP: 10

Format:

```
****************************
* SUB OPND1, OPND2, COP1 *
****************************
```

Function:

Algebraically subtract a subtrahend denoted by OPND1 from a minuend denoted by OPND2 and store the difference in the field denoted by COP1. OPND1, OPND2 and COP1 must be 4-bit items.

ADD TWO ADDRESS

```
*******
* INC *
*******
```

Format:

```
**********************
* INC OPND1, COP1 *
**********************
```

Function:

Algebraically add an addend denoted by OPND1 to an augend denoted by COP1 and store the sum in the field denoted by COP1.

INC is an optimized S-op. OPND1 and COP1 must have the same data unit type, that is, OPND1 and COP1 both must be either 4-bit data items or 8-bit data items. The format of the opcode is "01088SS". For example, if OPND1 and COP1 both are unsigned 8-bit items, then the opcode is "010110011" or 179.

SUBTRACT TWO ADDRESS

```
*******
* DEC *
*******
```

OP: 09

```
**********************
* DEC OPND1, COP1 *
**********************
```

Function:

Algebraically subtract a subtrahend denoted by OPND1 from a minuend denoted by COP1 and store the difference in the field denoted by COP1. OPND1 and COP1 must be 4-bit items.

MULTIPLY

```
********
* MULT *
********
```

OP: 11

Format:

```
****************************
* MULT OPND1, COP1, COP2 *
****************************
```

Function:

Algebraically multiply a multiplicand denoted by COP1 by a multiplier denoted by OPND1 and store the product in the field denoted by COP2.

The result field length is the sum of the lengths of the two operands and must be denoted by COP2. OPND1 and COP1 must be 4-bit items.

The result field is always either signed 4-BIT format or unsigned 4-BIT format.

DIVIDE

```
*******
* DIV *
*******
```

OP: 12

Format:

```
**************************
* DIV OPND1, COP1, COP2 *
**************************
```

Function:

Algebraically divide a dividend denoted by COP1 by a divisor denoted by OPND1 and store the quotient in the field denoted by COP2. Store the remainder in the field denoted by COP1.

The result field length is the difference of the lengths of the two operands and must be denoted by COP2.

Results are not specified if the length of the dividend is not greater than the length of the divisor.

If the absolute value of the divisor is not greater than the absolute value of an equivalent number of leading digits of the dividend, the result is undefined.

Division by zero results in a fatal error communicate to the MCP.

OPND1, COP1, and COP2 will be 4-bit items.

The sign of the remainder is that of the original dividend.

The dividend field is always either signed 4-BIT format or unsigned 4-BIT format.

DIVIDE SPECIAL

```
********
* DIVS *
********
```

OP: 16

Format:

```
****************************
* DIVS OPND1, COP1, COP2 *
****************************
```

Function:

This operation is performed in exactly the same manner as the standard divide (DIV) operator; except that when a divisor equal to zero is encountered, an overflow toggle is set and processing is allowed to continue. The overflow toggle can be manipulated by the "SOFL" and "BOFL" S-operators.

INCREMENT BY ONE

```
********
* INC1 *
********
```

OP: 240-243

Format:

```
**************
* INC1 COP1 *
**************
```

Function:

Algebraically add the positive integer one to an augend denoted by COP1 and store the sum in the field specified by COP1.

INC1 is an optimized S-op and the opcode includes information regarding the data type of COP1. COP1 may be a short COP: the format of the opcode is "01111008S".

OP	Type of data	8S
240	4-bit unsigned	00
241	4-bit signed	01
242	8-bit unsigned	10
243	8-bit signed	11

DECREMENT BY ONE

```
********
* DEC1 *
********
```

OP: 14

Format:

```
**************
* DEC1 COP1 *
**************
```

Function:

Algebraically subtract the positive integer one from a minuend denoted by COP1 and store the difference in the field specified by COP1. COP1 must be a 4-bit item.

DATA MOVEMENT OPERANDS AND INSTRUCTIONS

In general, fields involved in data movement operations can have any of the following formats.

- Unsigned 4-BIT.
- Unsigned 8-BIT.
- Signed 4-BIT (sign is MSD).
- Signed 8-BIT (sign over MSD).

Any restrictions as to the type of fields permitted in an operation are specified under the description of the particular operation.

See arithmetic operands and instructions for a description of the four types of fields.

Totally or partially overlapped fields are not permitted, unless specified by the description of the individual instruction.

MOVE ALPHANUMERIC

```
*******
* MVA *
*******
```

OP: 64-127

Format:

```
**********************
* MVA COP1, OPND1 *
**********************
```

Function:

Move 8-BIT or 4-BIT units from the source field denoted by OPND1 to the 8-BIT or 4-BIT destination field denoted by COP1.

If the destination field is signed, it receives either the sign of the source if the source is signed, or 1100 if the source is unsigned.

If the data type of the source field is 4-BIT and the data type of the destination field is 8-BIT, each 4-BIT unit is moved to the destination with 1111.

If the data type of the source field is 8-BIT and the data type of the destination is 4-BIT, the rightmost 4 bits are moved.

If the data type of the source field is the same as the data type of the destination field, each unit is moved unchanged to the destination.

If the destination length is greater in size than the source length, the destination field is filled in on the right with trailing spaces (0100 0000) if the destination type is 8-BIT; otherwise, it is filled in on the right with zeros (0000).

If the destination length is lesser in size than the source length, the source data is truncated on the right.

Overlapping operand fields are permitted if the data type of both fields is the same. It can be assumed that the source is moved 24 bits (six digits or three characters) at a time into the destination field and that the move is from left to right.

MVA is an optimized S-op, and COP1 and OPND1 may be short COP descriptors. The opcode contains information regarding the data types. The format of the opcode is "00188SS=>". For example, if both source and destination fields are signed 8-bit items of equal length, then the op code is "001111110" or 126.

MOVE SPACES

```
*******
* MVS *
*******
```

OP: 15

Format:

```
************
* MVS COP1 *
************
```

Function:

Fill the destination field denoted by COP1 with spaces (0100 C000).

The data type of the destination field is ignored and is assumed to be unsigned 8-BIT.

MOVE NUMERIC

```
*******
* MVN *
*******
```

OP: 128-191

Format:

```
********************
* MVN COP1, OPND1 *
********************
```

Function:

Move 8-BIT or 4-BIT units from the source field denoted by OPND1 to the 8-BIT or 4-BIT destination field denoted by COP1.

If the destination field is signed, it receives either the sign of the source if the source is signed, or 1100 if the source is unsigned.

If the destination field is unsigned, the sign of the source is ignored.

If the data type of the destination field is 8-BIT, the leftmost 4 bits of each 8-BIT unit, except for the sign position, if signed, are set to 1111 regardless of the data type of the source field.

If the data type of the destination field is 4-BIT, the leftmost 4 bits of each source 8-BIT unit are ignored and only the rightmost 4 bits are moved; if the source field is a 4-BIT field, each 4-BIT unit is moved unchanged.

If the destination length is greater in size than the source length, the destination field is filled in on the left with leading zeros of appropriate type (1111 0000).

If the source length is greater in size than the destination length, the source data is truncated on the left.

Note that a sign is placed in the leftmost 4 bits of a field, whether 4-BIT or 8-BIT.

Overlapping operand fields are permitted if the data type of both fields is the same. It can be assumed that the source is moved 24 bits (six digits or three characters) at a time into the destination field and that the move is from left to right.

MVN is an optimized S-op. Consequently, COP1 and OPND1 may be short COP descriptors. The opcode contains information regarding the data types. The format of the opcode is "01088SS=>". For example, if both source and destination fields are unsigned and the source field is 4-BIT data and the destination field is 8-BIT data and the destination length is greater in size than the source length, then the opcode is "010100001" or 161.

MOVE ZEROS

```
*******
* MVZ *
*******
```

OP: 236-239

Format:

```
************
* MVZ COP1 *
************
```

Function:

Fill the destination field denoted by COP1 with zeros of the appropriate type (1111 0000 or 0000 if 4-BIT).

If the destination field is signed, 1100 is placed into the sign position.

MVZ is an optimized S-op. COP1 can be a short COP. The format of the opcode is "01110118S".

Opcode	Data Type	8S
236	4-BIT unsigned	00
237	4-BIT signed	01
238	8-BIT unsigned	10
239	8-BIT signed	11

CONCATENATE

```
*******
* CAT *
*******
```

OP: 32

Format:

```
************************************
* CAT N, COP1, OPND0, ..., OPNDN *
************************************
```

Function:

Move each of the N+1 fields denoted by OPND0 through OPNDN, in the order specified, into an output string starting at the field denoted by COP1.

The number of source fields is specified by the 4-BIT binary value N. The value N ranging from 0000 to 1111 is used to indicate 1 to 16 source fields.

Each field is moved according to the rules specified for MOVE ALPHANUMERIC.

If the destination length is greater in size than the combined source length, the destination field is filled in on the right with trailing spaces (0100 0000).

If the destination length is less in size than the combined source lengths, the source data is truncated on the right.

EDIT INSTRUCTIONS AND EDIT MICRO-OPERATORS

No restrictions are placed on the data type of the source field of an edit operation.

The data type of the destination field of an edit operation must be unsigned 8-BIT.

If the destination length is greater in size than the source length, the source data is assumed to have leading zero fill on the left.

If the source length is greater in size than the destination length, the source data is truncated on the left.

The operation is terminated by an edit micro-operator and not by exhaustion of either the source or destination fields.

EDIT

```
********
* EDIT *
********
```

OP: 17

Format:

```
*****************************
* EDIT OPND1, COP1, DADDR *
*****************************
```

Function:

Move data from the source field, denoted by OPND1, to the destination field, denoted by COP1, under the control of the micro-operator string contained at the location denoted by the DADDR.

The argument DADDR is an unsigned binary value which specifies the digit displacement of the micro-operator string relative to the data segment zero base. The container size of DADDR is DISP8.

EDIT WITH EXPLICIT MASK

```
********
* EDTE *
********
```

OP: 21

Format:

```
****************************
* EDTE OPND1, COP1, MASK *
****************************
```

Function:

Move data from the source field denoted by OPND1 to the destination field denoted by COP1 under the control of the micro-operator string immediately following COP1. The format of the explicit micro-operator string is the same as a literal.

EDIT MICRO-OPERATORS

The edit micro-operators used in an edit instruction are:

Operator	Mnemonic	Operation
0000 R	MVD	MOVE DIGITS
0001 R	MVC	MOVE CHARACTERS
0010 R	MVS	MOVE SUPPRESS
0011 R	FIL	FILL SUPPRESS
0100 N	SRD	SKIP REVERSE DESTINATION
0101 T	INU	INSERT UNCONDITIONALLY
0110 T	INM	INSERT ON MINUS
0111 T	INS	INSERT SUPPRESS
1000 T	INF	INSERT FLOAT
1001 T	EFM	END FLOAT MODE
1010 0000	ENZ	END NON-ZERO
1010 0001	EOM	END OF MASK
1010 0010	SZS	START ZERO SUPPRESS
1010 0011	CCP	COMPLEMENT CHECK PROTECT
OTHERS		UNDEFINED

"R" indicates a 4-BIT binary value used as a repeat count. The value 0000 represents no repeat; do it once.

"N" indicates a 4-BIT binary value used to skip over a number of destination 8-BIT units. The value 0000 represents no skip.

"T" indicates a 4-BIT binary value which is:

- used to index into a table of editing constants.
- used to indicate a conditional selection between two table constants.
- used to indicate an editing constant in line with the edit-operator string.

The next edit-operator follows the constant.

The following table indicates the normal table editing constants as well as the conditional and unconditional selection of constants associated with the value "T".

EDITING CONSTANTS

T	Table Entry EBCDIC	Mnemonic	Unconditional or Conditional Constant
0000	"+"	PLU	
0001	"-"	MIN	
0010	"*"	AST	
0011	"."	DPT	
0100	","	CMA	
0101	"$"	CUR	
0110	"0"	ZRO	
0111	" "	BLK	
1000		SPM	EITHER ENTRY 0 OR 1
1001		SBM	EITHER ENTRY 7 OR 1
1010		LIT	IN-LINE 8-BIT CONSTANT

Associated with the edit instructions are three toggles denoted as "S" for sign, "Z" for zero suppress, and "P" for check protect. Initially, the "Z" and the "P" toggles are assumed to be set to the zero state. They are set and reset as specified by the description of the individual micro-operators. The "S" toggle is set to zero if the source field sign is positive, and to one if the source field sign is negative. Unsigned fields are considered positive.

MOVE DIGIT

Set "Z" to "1", ending the zero suppress state. Move an appropriate unit (4-BIT digit or 8-BIT character) from the source field to the destination field. If a 4-BIT unit is moved, append the 4 bits 1111 to the left before storing in the destination. If an 8-BIT unit is moved, the 4 bits 1111 are substituted for the leftmost 4 bits of the 8-BIT unit.

MOVE CHARACTER

Set "Z" to "1", ending the zero suppress state. Move an appropriate unit (4-BIT digit or 8-BIT character) from the source field to the destination field. If a 4-BIT unit is moved, append the 4 bits 1111 to the left before storing in the destination. If an 8-BIT unit is moved, it is moved unchanged.

MOVE SUPPRESS

The micro-operator "MOVE DIGIT" is performed if the 4-BIT unit, or the rightmost 4 bits of the 8-BIT unit of the source field are not equal to 0000.

If the appropriate 4 bits of the source field unit are equal to 0000, the suppress toggle "Z" is inspected. If "Z" equals "1", indicating nonsuppress mode, the micro-operator "MOVE DIGIT" is performed. If the suppress toggle "Z" equals "0", the check protect toggle "P" is inspected. If "P" = "0", indicating noncheck protect mode, move the table entry containing the 8-BIT code for blank to the destination field. If "P" = "1", move the table entry containing the 8-BIT code for asterisk to the destination field.

```
        SOURCE NOT = 0  MOVE DIGIT
Z=1     SOURCE     = 0  MOVE DIGIT
Z=0 P=0 SOURCE     = 0  MOVE TABLE ENTRY 7 (BLANK)
Z=0 P=1 SOURCE     = 0  MOVE TABLE ENTRY 2 (ASTERISK)
```

FILL SUPPRESS

If "P" = "0", indicating noncheck protect mode, move the table entry containing the 8-BIT code for blank to the destination field. If "P" = "1", move the table entry containing the 8-BIT code for asterisk to the destination field.

```
P = 0   MOVE TABLE ENTRY 7 (BLANK)
P = 1   MOVE TABLE ENTRY 2 (ASTERISK)
```

SKIP REVERSE DESTINATION

Adjust the address pointer of the destination field to skip backward (lower address) "N" 8-BIT units.

INSERT UNCONDITIONALLY

Move the table entry "T" as indicated below to the destination field.

```
        T=0...7          MOVE TABLE ENTRY T
S=0     T=8              MOVE TABLE ENTRY 0 (PLUS)
S=1     T=8              MOVE TABLE ENTRY 1 (MINUS)
S=0     T=9              MOVE TABLE ENTRY 7 (BLANK)
S=1     T=9              MOVE TABLE ENTRY 1 (MINUS)
        T=10             MOVE IN-LINE TABLE ENTRY
```

INSERT ON MINUS

Move the table entry "T" as indicated below to the destination field.

```
S=1      T=0...7         MOVE TABLE ENTRY T
 *  P=0                  MOVE TABLE ENTRY 7 (BLANK)
 *  P=1                  MOVE TABLE ENTRY 2 (ASTERISK)
S=1      T=8             MOVE TABLE ENTRY 1 (MINUS)
S=1      T=9             MOVE TABLE ENTRY 1 (MINUS)
S=1      T=10            MOVE IN-LINE TABLE ENTRY
```

*: S = 0 or only source digits/characters equal to zero (minus zero) have been moved.

INSERT SUPPRESS

Move the table entry "T" as indicated below to the destination field.

```
Z=1              T=0...7    MOVE TABLE ENTRY T
Z=0 P=0                     MOVE TABLE ENTRY 7 (BLANK)
Z=0 P=1                     MOVE TABLE ENTRY 2 (ASTERISK)
Z=1      S=0     T=8        MOVE TABLE ENTRY 0 (PLUS)
Z=1      S=1     T=8        MOVE TABLE ENTRY 1 (MINUS)
Z=1      S=0     T=9        MOVE TABLE ENTRY 7 (BLANK)
Z=1      S=1     T=9        MOVE TABLE ENTRY 1 (MINUS)
Z=1              T=10       MOVE IN-LINE TABLE ENTRY
```

INSERT FLOAT

Move the table entry "T" and/or perform the micro-operator "MOVE DIGIT" as indicated below.

```
Z=1                                MOVE DIGIT
Z=0 SOURCE      =0 P=0             MOVE TABLE ENTRY 7 (BLANK)
Z=0 SOURCE      =0 P=1             MOVE TABLE ENTRY 2 (ASTERISK)
Z=0 SOURCE NOT=0 T=0..7            MOVE TABLE ENTRY T, THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=8 S=0           MOVE TABLE ENTRY 0 (PLUS) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=8 S=1           MOVE TABLE ENTRY 1 (MINUS) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=9 S=0           MOVE TABLE ENTRY 7 (BLANK) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=9 S=1           MOVE TABLE ENTRY 1 (MINUS) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=10              MOVE IN-LINE TABLE ENTRY, THEN MOVE DIGIT
```

END FLOAT MODE

Move the table entry "T" as indicated below to the destination field.

```
Z=0      T=0...7                   MOVE TABLE ENTRY T
Z=0 S=0  T=8                       MOVE TABLE ENTRY 0 (PLUS)
Z=0 S=1  T=8                       MOVE TABLE ENTRY 1 (MINUS)
Z=0 S=0  T=9                       MOVE TABLE ENTRY 7 (BLANK)
Z=0 S=1  T=9                       MOVE TABLE ENTRY 1 (MINUS)
Z=0      T=10                      MOVE IN-LINE TABLE ENTRY
Z=1 NO OPERATION
```

END NON-ZERO

Terminate the micro-operator operations if any non-zero source character/digit has been moved; otherwise continue with the next in-line operator.

END OF MASK

Terminate the micro-operator operations.

START ZERO SUPPRESS

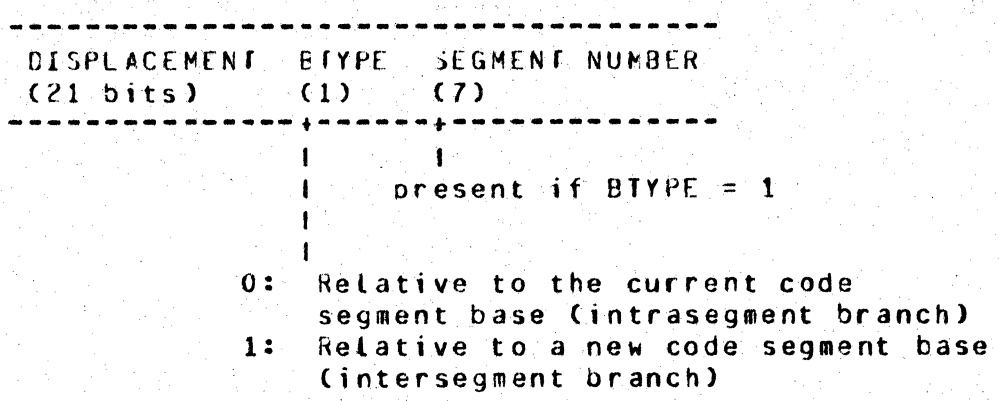
Set "Z" to the "0" state.

COMPLEMENT CHECK PROTECT

Complement the state of "P".

BRANCHING OPERANDS AND INSTRUCTIONS

A branch address argument "BADDR" has the following format:



Displacement is an unsigned binary value which specifies the bit displacement of an instruction relative to a segment base. The container size of the displacement and BTYPE combined is a program parameter.

BRANCH UNCONDITIONALLY

```
*******
* BUN *
*******
```

OP: 03

Format:

```
*************
* BUN BADDR *
*************
```

Function:

Obtain the next instruction from the location specified by BADDR.

BRANCH ON OVERFLOW

```
********
* BOFL *
********
```

OP: 23

Format:

```
******************
* BOFL V, BADDR *
******************
```

Function:

If the overflow toggle equals V, a transfer to the address (BADDR) given in the instruction occurs; otherwise, control is passed to the next sequential instruction.

The overflow toggle is unchanged. The length of V is one bit.

SET OVERFLOW TOGGLE

```
********
* SOFL *
********
```

OP: 07

Format:

```
**********
* SOFL V *
**********
```

Function:

Set the overflow toggle to V.

The length of V is 1 bit.

NOTE: The overflow toggle is set to one if a "DIVIDE BY ZERO" is encountered in the DIVIDE SPECIAL S-operator or if a field overflow is attempted in the MICR format S-operator.

PERFORM ENTER

```
********
* PERF *
********
```

UP: 06

Format:

```
******************
* PERF K, BADDR *
******************
```

Function:

Create a stack entry with the following format:

DISPLACEMENT (24)	SEGMENT NO. (7)	K (12)

Insert a displacement value, relative to the active code segment base and pointing to the next sequential S-instruction, into the stack.

Insert the current code segment number into the stack. Insert the value of K from the instruction into the stack.

Adjust the stack pointer to point to the next possible entry.

Obtain the next instruction from the location specified by BADDR.

PERFORM EXIT

```
********
* PXIT *
********
```

OP: 34

Format:

```
**********
* PXIT K *
**********
```

Function:

Compare the K contained in the instruction to the K in the current stack entry and if unequal, proceed to the next inline S-instruction. If equal, adjust the stack pointer to point to the previous entry and obtain the next S-instruction from the information contained in the removed stack entry.

ENTER

```
*******
* NTR *
*******
```

OP: 18

Format:

```
*************
* NTR BADDR *
*************
```

Function:

Same function as "PERF". K is assumed equal to zero.

EXIT

```
*******
* XIT *
*******
```

OP: 19

Format:

```
*******
* XIT *
*******
```

Function:

Same function as "PXIT". K is assumed equal to zero.

GO TO DEPENDING

```
********
* GOTD *
********
```

OP: 39

Format:

```
*****************************************
* GOTD COP1, L, DBADDR0, ..., DBADDRL *
*****************************************
```

Function:

Compare the 10-bit binary value L with the variable specified by COP1. The variable is first converted to a binary value, MODULO 2 to the 24th power.

If the binary value of the variable is less than zero or greater than L, the next instruction is obtained from the location specified by DBADDR0. Note that the variable can be signed.

If the binary value of the variable is in the range zero through L, it is used as an index to select from the list of DBADDRs the appropriate DBADDR to be used to obtain the next instruction.

DBADDR and BADDR have the same format with the exception that DBADDR always contains the segment number. Although segment number is unneccessary for those DBADDRs with BTYPE equal to zero, in order to index into the list of DBADDRs, all of the DBADDRs must be of equal length. The container size of DBADDR is BDISPB1 + 7.

ALTERED GO TO PARAGRAPH

```
********
* GPAR *
********
```

OP: 35

Format:

```
***************
* GPAR DADDR *
***************
```

Function:

Obtain the next instruction from the location specified by the address "ACON".

The address constant "ACON" has the same format as a BADDR.

The argument DADDR is an unsigned binary value which specifies the digit displacement of the "ACON" relative to the data segment zero base.

The container size of DADDR is DISPB.

ALTER

```
********
* ALTR *
********
```

OP: 36

Format:

```
**********************
* ALTR DADDR, ACON *
**********************
```

Function:

Copy the address constant "ACON" into the data area specified by the argument DADDR.

The address constant "ACON" has the same format as a BADDR.

The argument DADDR is an unsigned binary value which specifies the digit displacement of the "ACON" relative to the data segment zero base.

The container size of DADDR is DISPB.

CONDITIONAL BRANCH OPERANDS AND INSTRUCTIONS

If the condition "A (R) B" is true a transfer to the address (BADDR) given in the instruction occurs; otherwise, control is passed to the next sequential instruction. The relation (R) is defined as follows:

```
000  UNDEFINED
001  GTR
010  LSS
011  NEQ
100  EQL
101  GEQ
110  LEQ
111  UNDEFINED
```

Overlap of fields is permitted. "A" is the first operand denoted in the instruction. If an instruction has only one operand, then the assumed field is the "A" field.

COMPARE ALPHANUMERIC

```
********
* CMPA *
********
```

CP: 224-231

Format:

```
**********************************
* CMPA OPND1, OPND2, R, BADDR *
**********************************
```

Function:

Compare the two operand fields according to their binary values.

The comparison is performed left to right with any shorter operand assumed to be right-filled with blank characters (0100 0000 if EBCDIC or 0010 0000 if ASCII).

The fields are considered equal when the equal size portions are equal and the longer field (if one is longer) has trailing blanks.

8-BIT data format is assumed for both fields with no checking to verify otherwise. Signed fields have their most significant 4 bits (their sign) modified to the appropriate numeric zone (1111 for EBCDIC, 0011 for ASCII) before being compared. This modification is not permanent and is done so that sign doesn't affect the result of an alphanumeric comparison.

CMPA is an optimized S-op. Consequently, OPND1 and OPND2 may be short COP. The LOGICALSIZE of OPND2 is always greater than or equal to the LOGICALSIZE of OPND1. The format of the opcode is "0111100SS=".

COMPARE NUMERIC

```
********
* CMPN *
********
```

OP: 192-223

Format:

```
**********************************
* CMPN OPND1, OPND2, R, BADDR *
**********************************
```

Function:

Compare the two operand fields according to the algebraic values, considering the two fields to be comprised of decimal integers.

When the field sizes are different, the longer is tested for leading zeros (0000). There is no restriction as to data type. In comparing an 8-BIT character, only the rightmost 4 bits of the character are considered; the other bits are ignored.

Two fields of all zeros are equal regardless of sign.

Unsigned fields are considered positive. Sign conventions are the same as for arithmetic operands.

Results generated by invalid digit values are undefined.

CMPN is an optimized S-op. OPND1 and OPND2 can be short COPS. The LOGICALSIZE of OPND1 is always less than the LOGICALSIZE of OPND2. The format of the opcode is "011088SS=".

COMPARE FOR ZEROS

```
********
* CMPZ *
********
```

OP: 232-235

Format:

```
************************
* CMPZ COP1, K, BADDR *
************************
```

Function:

Compare two operand fields according to their algebraic values, assuming the first field to be comprised of all zeros (0000).

There is no restriction as to data type. In comparing an 8-BIT character only the rightmost 4 bits of the character are considered. The other bits are ignored.

Two fields of all zeros are equal regardless of sign.

Unsigned fields are considered positive. Sign conventions are the same as for arithmetic operands.

Results generated by invalid digit values are undefined.

CMPZ is an optimized S-op. COP1 can be a short COP. The format of the opcode is "011101085".

Opcode	Data Type	BS
232	4-BIT unsigned	00
233	4-BIT signed	01
234	8-BIT unsigned	10
235	8-BIT signed	11

COMPARE FOR SPACES

```
********
* CMPS *
********
```

OP: 37

Format:

```
************************
* CMPS COP1, R, BADDR *
************************
```

Function:

Compare two operand fields according to their binary values, assuming the first field to be comprised of all spaces (0100 0000).

The comparison is performed left to right.

Unsigned 8-BIT format is assumed with no checking to verify otherwise.

This operator is not sensitive to collate table address and is valid only for the native collative sequence.

COMPARE FOR CLASS

```
********
* CMPC *
********
```

OP: 38

Format:

```
************************
* CMPC COP1, C, BADDR *
************************
```

Function:

Compare the operand field and determine whether the field is:

```
C=00 COMPLETELY ALPHABETIC
  01 COMPLETELY NUMERIC
  10 NOT COMPLETELY ALPHABETIC
  11 NOT COMPLETELY NUMERIC
```

If the condition being tested is true, a transfer to the address BADDR given in the instruction occurs; otherwise, control is passed to the next sequential instruction.

In the alphabetic test, each character is range-checked for 1100 0001 through 1100 1001, 1101 0001 through 1101 1001, 1110 0010 through 1110 1001 and for 0100 0000. Unsigned 8-BIT format is assumed with no checking to verify otherwise.

In the numeric test each character is range-checked for 1111 0000 through 1111 1001. Signed or unsigned 8-BIT format is permitted. The 4 bits in the sign position of a signed 8-BIT field are ignored. The sign position is the leftmost 4 bits of the most significant character.

COMPARE REPEAT

```
********
* CMPR *
********
```

OP: 45

Format:

```
*********************************
* CMPR OPND1, COP1, R, BADDR *
*********************************
```

Function:

Compare the two operand fields according to their binary value.

If the COLLATE TABLE ADDRESS is nonzero, each character in the operands must be translated before comparison. This is accomplished by using each 8-BIT character from OPND1 and COP1, multiplied by eight, as an index into the translation table located at the address given by the COLLATE TABLE ADDRESS to obtain the character to be compared.

Comparison proceeds from left to right.

The field lengths are considered equal by repeating OPND1.

Both fields are assumed to have unsigned 8-BIT data type.

The size of OPND1 must divide evenly into the size of COP1; otherwise, the results of the compare may be erroneous.

COMPARE COLLATE

```
*******
* CPC *
*******
```

OP: 04

Format:

```
*******************************
* CPC OPND1, COP1, R, BADDR *
*******************************
```

Function:

If the COLLATE TABLE ADDRESS is nonzero, this program is using a non-native collating sequence and the operands must be translated into this program's collating sequence before comparison. This is accomplished by using each 8-BIT character from OPND1 and COP1, multiplied by eight, as an index into the translation table located at the address given by the COLLATE TABLE ADDRESS to obtain the character to be compared.

The comparison is performed left to right with any shorter operand assumed to be right-filled with blank characters (the blanks being translated if this program is using a non-native collating sequence).

The fields are considered equal when the equal size portions are equal and the longer field (if one is longer) has trailing blanks.

8-BIT data format is assumed for both fields with no checking to verify otherwise. Signed fields have their most significant 4 bits, i.e., their sign modified to 1111 before translation is done (if necessary). This modification is not permanent and is done so that the sign will not affect the result of an alphanumeric comparison.

MISCELLANEOUS INSTRUCTIONS

COMMUNICATE

```
********
* COMM *
********
```

OP: 33

Format:

```
**************
* COMM COP1 *
**************
```

Function:

Move the length and address fields from the COP1 entry to the RS.COMMUNICATE.MSG.PTR field located in this program's RS.NUCLEUS, converting them enroute. The origin field is unchanged.

The length is converted from a digit or character length to a bit length. The address is stored as an absolute bit address.

LOAD COMMUNICATE REPLY

```
********
* LDCR *
********
```

OP: 41

Format:

```
***************
* LDCR DADDR *
***************
```

Function:

The LDCR does the mapping of RS.RMSG.P2 and the last logical I/O status values as follows:

RS.RMSG.P2 Value	Value Stored at DADDR
0 (Good Complete)	0
1 (AT END)	1
2 (I/O Error)	2
3 (Incomplete I/O)	3
4 (Duplicate Record OK)	0
5-10	1

LAST LOGICAL I/O STATUS	Value Stored at DADDR
bit 0 (Exception Descriptor Follows)	
value of 0	2
value of 1	examine following bits
1 (Boundary Violation)	1
2 (Duplicate Key)	1
3 (Sequence Error)	1
4 (Variable Length Record Err)	2
5 (Invalid Key)	1
6 (Reserved)	2
7 (Parity Error)	1
8 (Reserved)	1
9 (AT END / EOP)	1
10 (Short Block)	2
11 (Reserved)	1
12 (Reserved)	2

13 (Break On Ouput)	2
14 (Reserved)	1
15 (Time Out)	1
16 - 23 (Reserved)	1

WAIT COMMUNICATE REPLY

****
* WTCR *
****

OF: 56

****
* WTCR DADDR *
****

Function:

The WTCR returns the value of RS.RMSG.P2 at DADDR.

CONVERT

```
********
* CONV *
********
```

OP: 40

Format:

```
**********************
* CONV COP1 DADDR N *
**********************
```

Function:

Convert the operand denoted by COP1 from a decimal value to an unsigned 24-bit binary value, truncating or zero-filling on the left if necessary. Place the result at the location specified by DADDR. N represents the number of bits of converted value to store at DADDR.

The operand must be either unsigned 4-BIT or unsigned 8-BIT units.

See 'MAKE PRESENT' for definition of DADDR.

MAKE PRESENT

```
********
* MAKP *
********
```

UP: 42

Format:

```
**********************
* MAKP COP1, DADDR *
**********************
```

Function:

Load the data segment specified by COP1 and place the base relative address of the data area specified by COP1 into the 24-bit location specified by DADDR.

DADDR is an unsigned binary value which specifies a digit displacement from the data segment zero base.

The container size of DADDR is DISPB.

HARDWARE MONITOR

```
********
* HMON *
********
```

OP: 43

Format:

```
***************
* HMON OPND1 *
***************
```

Function:

The low-order 8 bits of the field described by OPND1 are used as the input to the monitor micro-operator described in the following product specifications:

M-Memory Processor	#1913 1747
S-Memory Processor	#2201 6760

The length of the field described by OPND1 must be greater than or equal to 8 bits.

FILE STATUS

****
* FSTA *
****

OP: 57

Format:

****
* FSTA, WRIT, COP *
****

Function:

Place in COP the translated value of the status of the last IO.WRIT indicates whether the last IO was a READ OR WRITE.

The translation from COMMUNICATE REPLY (RE.RMSG.P2) or the last logical I/O status values to file status codes is according to the following table:

COMMUNICATE REPLY Values returned by MCP	FILE STATUS CODES
0 (Good Complete)	00
1 (AT END / EOP)	10 (READ) 34 (WRITE)
2 (I/O Error)	Examine I/O status
3 (Incomplete I/O)	94 (READ), 95 (WRITE)
4 (Duplicate Record OK)	02
5-10	99

LAST LOGICAL I/O STATUS	FILE STATUS CODES
bit 0 (Exception Description Follows)	
value of 0	99
value of 1	examine following bits
1 (Boundary Violation)	24
2 (Duplicate Key)	22
3 (Sequence Error)	21
4 (Variable Length Record Err)	92
5 (Invalid Key)	23
6 (Reserved)	--
7 (Parity Error)	30

8	(Reserved)	--
9	(AT END / EOP)	not requested
10	(Short Block)	91
11	(Reserved)	--
12	(Reserved)	--
13	(Break On Output)	97
14	(Reserved)	--
15	(Time Out)	96
16 -23	(Reserved)	--

For any COMMUNICATE REPLY or last logical I/O status value that does not have a File Status code, the condition will be masked out with the result mask specified by the communicate. Note that an EOP will result in a "34" instead of the expected (and correct) "00".

BINARY SEARCH ASCENDING LOW

```
********
* BSAL *
********
```

OP: 56

Format:

```
****************************************
* BSAL  COP1, OPND, KEY.OFFSET, COP2 *
****************************************
```

Function:

Use a binary search to find the lowest index value of a key in a table that matches a search argument. If no match occurs then the index will describe the value of the next highest key in the table.

COP1 is the index descriptor. OPND is the search argument descriptor. KEY.OFFSET is a 20 bit binary value that denotes the beginning of a key relative to the beginning of a table element. COP2 is the table descriptor. At the beginning of this op, COP1 describes the largest zero-relative index value of the table for this search. The table is supposed to be an ascending ordered table such that table [i].key <= table [i + 1].key.

The algorithm for this binary search is characterized as follows:

```
VAR     i, high, low : integer;
        search_argument : character (key_length);
        table : array [0..COP1_value] of table_elements;

begin   low:= 0; high:= COP1_value; i:= low;
        repeat  {low <= i <= high}
        if search_argument >= table [i].key then low:= i
        else high:= i;
        i:= (high + low) div 2;
        until  (i = low);
        COP1_value:= high; {search_argument <= table (high)}
end
```

BINARY SEARCH ASCENDING HIGH

```
********
* BSAH *
********
```

OP: 29

Format:

```
****************************************
* BSAH  COP1, OPND, KEY.OFFSET, COP2 *
****************************************
```

Function:

Use a binary search to find the highest index value of a key in a table that matches a search argument. If no match occurs then the index value will describe the next lowest key in the table.

COP1 is the index descriptor. OPND is the search argument descriptor. KEY.OFFSET is a 20 bit binary value that denotes the beginning of a key relative to the beginning of a table element. COP2 is the table descriptor. At the beginning of this op, COP1 describes the largest zero-relative index value of the table for this search. The table is supposed to be an ascending ordered table such that table [i].key <= table [i + 1].key.

The algorithm for this binary search is characterized as follows:

```
VAR     i, high, low : integer
        search_argument : character (key_length);
        table : array [0..COP1.value] of table_element;

begin   low:= 0; high:= COP1_value; i:= high;
        repeat  {low <= i <= high}
        if search_argument >= table [i].key then low:= i
        else high:= i;
        i:= (high + low + 1) div 2;
        until (i=high);
        COP1_value:= low; {search_argument >= table (low)}
end
```

BINARY SEARCH DESCENDING LOW

```
********
* BSDL *
********
```

OP: 30

Format:

```
*****************************************
* BSDL  COP1, OPND, KEY.OFFSET, COP2 *
*****************************************
```

Function:

Use a binary search to find the lowest index value of a key in a table that matches a search argument. If no match occurs then the index value will describe the next lowest key in the table.

COP1 is the index description. OPND is the search argument descriptor. KEY.OFFSET is a 20 bit binary value that denotes the beginning of a key relative to the beginning of a table element. COP2 is the table descriptor. At the beginning of this op COP1 describes the largest zero-relative index value of the table for this search. The table is supposed to be a descending ordered table such that table [i].key => table [i + 1].key.

The algorithm for this binary search is characterized as follows:

```
VAR     i, high, low : integer;
        search_argument : character (key_length);
        table : array [0..COP1_value] of table_elemets;

begin   low:= 0; high:= COP1_value; i:= low;
        repeat  {low <= i <= high}
        if search_argument < table [i].key then low:= i;
        else high:= i;
        i:= (high + low) div 2;
        until   (i = low);
        COP1_value:= high;  {search_argument >= table [high].key}
end
```

BINARY SEARCH DESCENDING HIGH

```
********
* BSDH *
********
```

OP: 31

Format:

```
*****************************************
* BSDH  COP1, OPND, KEY.OFFSET, COP2 *
*****************************************
```

Function:

Use a binary search to find the highest index value of a key in a table that matches a search argument. If no match occurs then the index value will describe the next highest key in the table.

COP1 is the index descriptor. OPND is the search argument descriptor. KEY.OFFSET is a 20 bit binary value that denotes the beginning of a key relative to the beginning of a table element. COP2 is the table descriptor. At the beginning of this op, COP1 describes the largest zero-relative index value of the table for this search. The table is supposed to be a descending ordered table such that table [i].key => table [i + 1].key.

The algorithm for this binary search is characterized as follows:

```
VAR     i, high, low : integer;
        search_argument : array [i..key_length] of char;
        table : array [0..COP1_value] of table_elements;

begin   low:= 0; high:= COP1_value; i:= high;
        repeat  {low <= i <= high}
        if search _argument < table [i].key then low:= i
        else high : i;
        i:= (high + low + 1) div 2;
        until   (i = high);
        COP1_value:= low; {search_argument => table [low].key}
end
```

CHARACTER STRING S-OPS.

DESCRIPTOR SETUP

```
********
* DSET *
********
```

OP: 50

Format:

```
**********************
* DSET DADDR, COP *
**********************
```

Function:

Build a descriptor of COP at the location specified by DADDR.

The format of a descriptor (DESC) is that of a COP entry after the OCCURS DEPENDING value has been applied and after subscripts or indexes have been evaluated to derive an ultimate displacement.

The current B1800 format for a descriptor is as follows:

DESC (64)

DATA-TYPE (4)	MULTIPLE-ENTRY-FLAG (1)	SIGNED (1)	SHARED-DATA-FLAG (1)	LITERAL-FLAG (1)

LENGTH-IN-DIGITS (14)	SEGNO-OR-IPCD-INDEX (10)	DISPLACEMENT IN DIGITS (20)	FILLER (12)

The MULTIPLE.ENTRY.FLAG and LITERAL.FLAG are equal to zero in the above format.

INSPECT SETUP

```
********
* ISET *
********
```

OP: 51

Format:

```
**************************************************************
* ISET V1, V2, FLAC, DADDR1, COP1, COP2, DADDR2, OPND *
**************************************************************
```

Function:

Build an INSPECT.TABLE entry at DADDR1.

V1 specifies whether this is the last entry (V1=1) or not. V2 indicates whether BEFORE, V2=1, or AFTER, V2=2, or both, V2=3, were specified. FLAC indicates the mode of INSPECTION.

```
FLAC = 0 indicates FIRST
FLAC = 1 indicates LEADING
FLAC = 2 indicates ALL
FLAC = 3 indicates CHARACTERS
```

DADDR1 indicates the location for this INSPECT TABLE entry. COP1 indicates the number of characters to be tallied or replaced when FLAC is not equal to 3.

COP2 is the count field for tallying or the replacement string. DADDR2 is the location of the source filed description if BEFORE/AFTER is specified.

OPND is the delimiter if BEFORE/AFTER is specified.

Set INELIGIBLE.FLAG=0.

If V2=0, set AFTER.FLAG =0 and examine the source field for the delimiter specified by OPND1. If a match is found, set ELIGIBLE.POSITION to the character position of the beginning of the first occurrence of the delimiter. If a match is not found, set ELIGIBLE.POSITION=0 and AFTER.FLAG=1. This, in effect, makes the entry eligible for all characters of the source field.

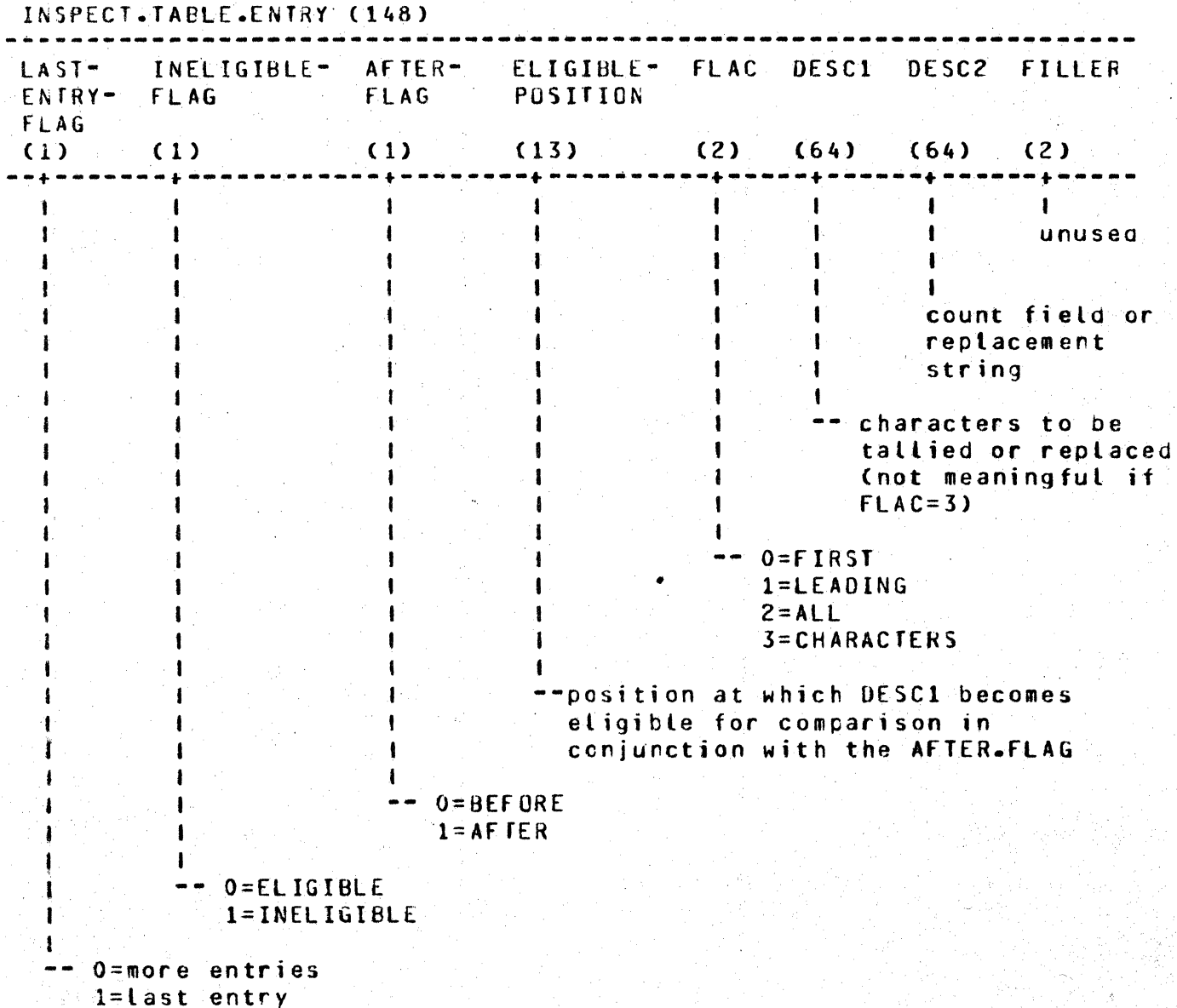
If V2=1, set AFTER.FLAG=1 and examine the source field for the delimiter specified by OPND1. If a match is found, set ELIGIBLE.POSITION to the character position to the right of the

first occurrence of the delimiter. If a match is not found, set INELIGIBLE.FLAG=1.

If V2=3, set ELIGIBLE.POSITION=0 and AFTER.FLAG=1. This, in effect, makes the entry eligible for all characters of the source field.

The description of the INSPECT TABLE is as followed:

INSPECT.POINTER is the first 24 bits of the Inspect Table which is used to save a pointer into the source string in case inspection is interrupted. This field is followed by a variable number of entries with the following format:



INSPECT

```
********
* INSP *
********
```

OP: 52

Format:

```
****************************
* INSP V, DADDR1, DADDR2 *
****************************
```

Function:

Inspect the source field specified at DADDR2, tallying, if V=0, or replacing, if V=1, a variable number of 8-BIT character(s). The location of the character(s) to be tallied or replaced and the location of count field or replacement string is specified in the Inspect Table specified by DADDR1.

Eligibility of an entry is established as follows:

1. If INELIGIBLE.FLAG=1, this entry is ineligible.

2. If AFTER.FLAG=0 and the current character position is not less than ELIGIBLE.POSITION, this entry is no longer eligible and it is made ineligible otherwise the entry is eligible.

3. If AFTER.FLAG=1 and the current character position is less than ELIGIBLE.POSITION, this entry is ineligible; otherwise the entry is eligible.

4. When looking for a match and the number of characters left in the source is less than the size of the data item specified by DESC1, this entry is made ineligible.

Note: An entry is made ineligible by setting INELIGIBLE.FLAG=1.

The comparison operation to determine the occurrences of the operands to be tallied or replaced occurs as follows:

1. The entries in the INSPECT.TABLE are considered in the order in which they are specified.

2. If the entry is eligible, DESC1 is compared against an equal number of contiguous characters, starting with the leftmost character of the source field.

3. If no match occurs or if the entry is ineligible, the comparison is repeated for each successive entry, if any, until either a match is found or there is no next successive entry. When there is no successive entry, the character position immediately to the right of the leftmost character position considered in the last comparison cycle is considered as the leftmost character position, and the comparison cycle begins with the first table entry.

4. Whenever a match occurs, tallying or replacing takes place. The character position immediately to the right of the rightmost character position that participated in the match is now considered to be the leftmost character position of the source field and the comparison cycle starts again with the first table entry.

5. The comparison operation continues until there are no eligible entries in the table or until the rightmost character position of the source filed has participated in a match or has been considered as the leftmost character position. When this occurs, the inspection is terminated.

NOTE:

It is intendec that this process be interruptable at the beginning of the cycle. INSPECT.POINTER is the first 24 bits of the Inspect Table used to save and restore the pointer into the source field in case interrupt occurs.

A detailed description of the inspection cycle is as follows:

1. Set current position P=INSPECT.POINTER.

2. Point to the first entry in the Inspect Table.

3. If the entry is ineligible, go to step 5.

4. Do the following according to the contents of FLAC:

   a. If FIRST is specified, look for a match.

      If a match is not found go to step 5.

      If a match is found, make the entry ineligible, tally or replace and go to step 7.

   b. If LEADING is specified, make the entry ineligible.

      If (AFTER.FLAG=0 AND P NOT=1) OR (AFTER.FLAG=1 AND P NOT=ELIGIBLE.POSITION), go to step 5.

      Look for a match.

      If a match is not found, go to step 5.

      If a match is found, tally or replace contiguous occurrences of DESC1 and go to step 7.

   c. If ALL is specified, look for a match.

      If a match is not found, go to step 5.

      If a match is found, tally or replace and go to step 7.

   d. If CHARACTERS is specified, tally or replace and go to step 6.

5. Get the next table entry. If there is a next entry go to step

   If INELIGIBLE.FLAG=1 for all the table entries, terminate the process.

6. Increment P by 1.

7. If P is greater than the size of the source string, terminate the process; otherwise go to step 2.

STRING

```
********
* STRI *
********
```

OP: 53

Format:

```
************************************************
* STRI Z, CPND1, DADDR1, COP1, OPND2, BADDR *
************************************************
```

Function:

Move a variable number of 8-bit characters from the source field denoted by OPND1 to the destination field specified at DADDR1 using the value specified in COP1 as a character offset into the destination field. Z indicates how OPND2 is to be used:

```
0=OPND2 is the # of characters to move
1=OPND2 is the delimiting character(s)
2=OPND2 is omitted - the # of characters
  to move is the size of OPND1
```

BADDR specifies the next statement if overflow occurs.

An overflow condition is caused if at anytime during execution the value in the pointer field is less than one or greater than the size of the destination string.

If an overflow condition exists, the operation is terminated and the next instruction address is obtained from BADDR.

The transfer of data is governed by the following rules:

1. Characters are moved from the source field to the destination field in accordance with the rules for Move Alphanumeric (MVA) except that no space fill will be provided. The contents of COP1 are incremented by one for each character transferred.

2. If V=0, characters are transferred until the end of the

source field is reached, or until the number of characters specified by OPND2 are transferred.

If V=1, characters are transferred until the end of the source field is reached, or until the delimiting character(s) specified by OPND2 are encountered. The delimiting character(s) specified are not transferred.

DELIMITER SETUP

```
********
* DLIM *
********
```

OP: 54

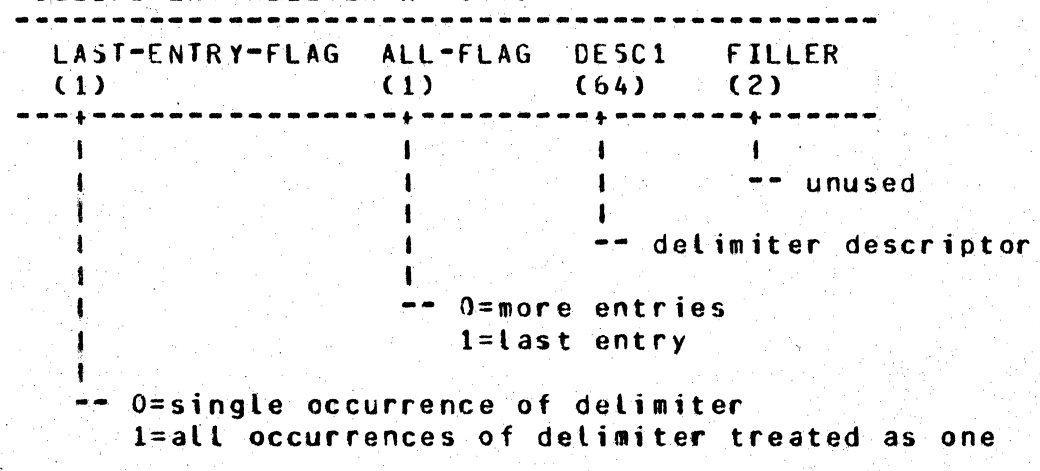
Format:

```
*********************************
* DLIM V1, V2, DADDR1, COP1 *
*********************************
```

Function:

Build a delimiter table entry at the location specified by DADDR1. V1=1 specified that this is the last entry. V2=1 indicates that ALL was specified. COP1 is the delimiter.

The table used by the Unstring S-operator contains a variable number of entries with the following format:



When Z=1, DADDR3 specifies the location of the Delimiter Table and the following comparison cycle occurs:

1. The entries in the Delimiter Table are considered in the order in which they are specified.

2. The delimiter is compared against an equal number of characters of the source field, starting with the leftmost character.

3. If no match occurs, the comparison is repeated for each successive entry, if any, until either a match is found or there is no next successive entry. When there is no successive entry, the character position immediately to the right of the leftmost character position considered in the last comparison cycle is considered as the leftmost character position, and the comparison cycle begins with the first table entry.

4. Whenever a match occurs, examination stops and the comparison cycle is discontinued.

5. The comparison operation continues until there are no more entries in the table or until the rightmost character position of the source field has participated in a match or has been considered as the leftmost character position. When this occurs, examination is terminated.

UNSTRING

```
********
* UNST *
********
```

OP: 55

Format:

```
************************************************
* UNST F, M, Z, C, J, BADDR, DADDR1, COP1,
* DADDR2, CPND1, DADDR3, COP2, COP3, DADDR4 *
************************************************
```

Function:

Move a variable number of 8-bit characters from the source field specified at DADDR1 to the destination field denoted by COP1 using the value specified at DADDR2 as a character offset into the source field.

Any of the following situations causes an overflow condition:

1. F=2 or F=3 and the value in the pointer field is less than one or greater than the size of the source string.

2. Z=0 and the value in OPND1 is less than one or greater than the size of the source string.

3. Upon completion of the operation, F=1 or F=3 and either of the following occurs:

   a. The source field contains characters that have not been examined.

   b. Z=0 and the number of characters transferred is less than the number of characters specified by OPND1.

If an overflow condition exists, the operation is terminated and the next instruction address is obtained from BADDR.

The transfer of data is governed by the following rules:

1. The string of source characters is examined beginning with the relative character position in the pointer field.

2. The number of characters examined is determined as follows:

   a. If Z=1, the examination proceeds left to right until either a delimiter in the Delimiter Table specified by DADDR3 is encountered or until the end of the string is encountered.

      If the end of the string is encountered before the delimiting condition is met, examination terminates with the last character to be examined.

      If two contiguous delimiters are encountered, the number of characters examined is zero.

   b. If Z=2, the number of characters is the size specified by COP1.

3. The characters, thus examined, are moved to the destination field as follows:

   a. If C=0, the characters are moved in accordance with the rules of the Move Numeric (MVN) S-operator.

      If the number of characters to be moved is zero, the destination field is zero-filled.

   b. If C=1, the characters are moved in accordance with the rules of the Move Alphanumeric (MVA) S-operator.

      If J=1, the characters are moved right-justified.

      If the number of characters to be moved is zero, the destination field is space-filled.

4. If the delimiter receiving field is present, the delimiting character(s) are moved to it in accordance with the rules of the Move Alphanumeric (MVA) S-operator. If the delimiting condition is the end of the string, then it is space-filled.

5. If the count field is present, the number of characters examined is moved to it in accordance with the rules of the Move Numeric (MVN) S-operator.

6. If the tallying field is present, one is added to its contents in accordance with the rules of the Increment By One (INC1) S-operator.

7. If Z=0, Z=2 or the delimiting condition is the end of the string, the pointer field is set to point to the character to the right of the last character transferred in step 3.

   If Z=1, the pointer field is set as follows:

   a. If the ALL.FLAG=0 for the matched delimiter, it points to the character to the right of the delimiter.

   b. If the ALL.FLAG=1 for the matched delimiter, the string is examined for contiguous occurrences of the delimiter and it points to the right of the last such occurrence.

INTERPROGRAM COMMUNICATION

IPC DICTIONARY

```
********
* IPCD *
********
```

Format:

```
**********************
* IPCD V COP BADDR *
**********************
```

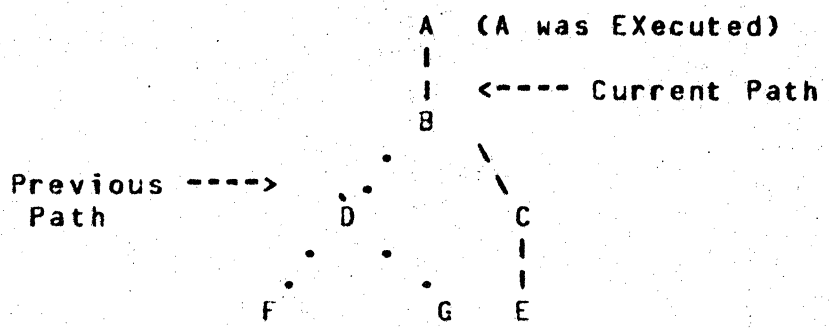
Function:

The execution of this S-op results in the construction and insertion of an 80-bit System Descriptor for the COP in the IPC.DICTIONARY at the displacement address specified by V. The base address of the IPC.DICTIONARY is adjacent to the program's RSN and can be found in RS.IPC.DICTIONARY.SPACE as an absolute address. The BADDR contains the address of the first IPCD S-op which references an operand that lies in an overlayable data segment. This address is required by the interpreter when checkerboarding has occurred and a necessary data segment does not fit into dynamic memory without overlaying another required data segment.

The Inter-Program Communication (IPC) Module provides a facility to transfer control from one program to another and the ability for both programs to have access to the same data items. Language is not a barrier in IPC. The names of the programs to which control is to be passed may or may not be known at compile time. Additionally, this module provides the ability to determine the availability of memory for the program to which control is being passed.

The definition of a 'run unit' is critical to the implementation of the CALL/CANCEL mechanism described in the ANSI Standard. The beginning-of-job (BOJ) of any program via an EXecute does not establish a run unit. A run unit is established only when an EXecuted program initiates another program (BOJ) via the CALL communicate. That CALLed program is now a member of the run unit associated with the program that was originally EXecuted. Likewise, any program CALLed by a program within the run unit

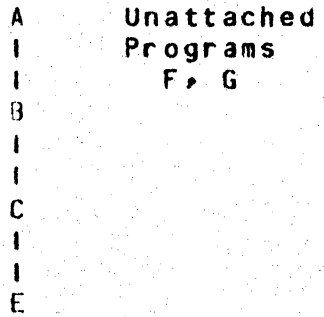
becomes part of that run unit and remains in that run unit until terminated or CANCELled. A job cannot be a member of more than one run unit. The following figure represents seven programs (A-G) which have been called within a run unit.



The connecting links are generated by and represent the last used path, and the link exists until a return (EXIT PROGRAM) is accomplished. Once a CALLed program has been exited (D, F, G), it remains suspended in its current state. The only path that is of interest is the path last traversed.

The current path is important in order to check the validity of a CALL or CANCEL statement; a program may not call or cancel itself or any of its predecessors. The other links are unimportant, as any program in the run unit can call or delete other existing programs (with the previously mentioned exceptions) or can call new programs.

If, for example, program 'E' cancels program 'D', then the run unit consists of all of the following programs and appears as:



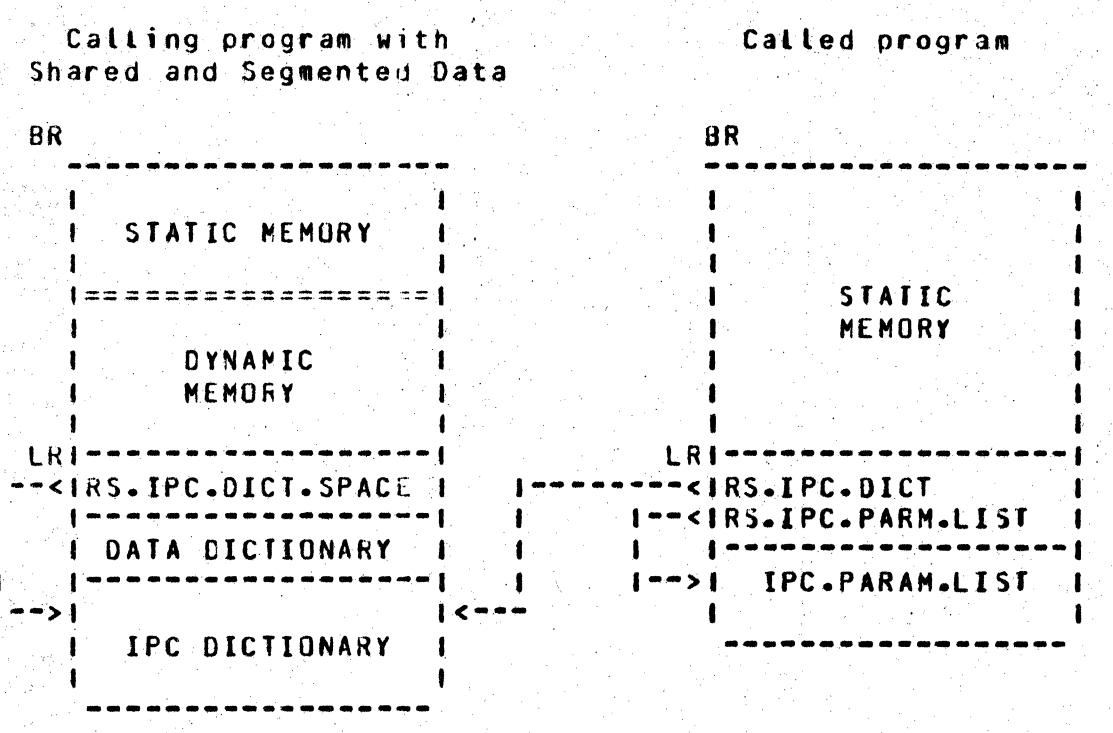
A call to any of these programs results in a transfer of control to the existing state, whereas a call to any other program (including 'D') causes an initial state copy to be invoked before control is transferred. The termination (via STOP RUN or ABORT) of any program in a run unit results in the removal of all programs in that run unit from memory.

The calling program may specify one or more data items to which the called program has access. The shared data may be any 01 or 77 level item (including items whose addresses have been received through a CALL) described in the calling program. The data items may be named and defined differently in each program. Additionally, storage for the shared data is never allocated in the called program; the address is always passed to the called program.

The IPC.DICTIONARY is a list of SYSTEM.DESCRIPTORS built by the program to describe the parameters to be passed with a CALL. This dictionary is within the space defined by RS.IPC.DICT in the RS.NUCLEUS of the CALLing program. The length of this dictionary is passed in the CALL communicate. The MMCP verifies that the number and length of parameters passed match the IPC.PARAMETER.LIST of the CALLed program.

The IPC.PARAMETER.LIST is a series of 24-bit fields that contain the length in bits of the parameters required for a given program. The original copy is generated by the compiler and resides at the end of the code file. The MCP can locate this list through a 24-bit field in the Program Parameter Block (the PROG.IPC.PTR contains the relative disk address in the code file of the IPC.PARAMETER.LIST). The number of entries in the list is obtained from a 16-bit field in the PPB (PROG.IPC.SIZE). After the program has been successfully called, the IPC.PARAMETER.LIST is appended to the RS.NUCLEUS (and the first element contains the number of entries) in order to facilitate future calls on this program.

After the MCP had verified that the number and length of parameters are correct, it updates this field to allow access to the shared data. The interpreter uses this field just as it would use the address of a Data Dictionary, that is, to obtain the absolute address of the IPC.DICTIONARY.



Accessing shared data should be almost identical to accessing segmented data. One difference is that the reading and writing takes place outside the program's base-to-limit area and therefore, the base-limit check must be inhibited. This implies that any COP for a parameter must have a flag (SHARED.DATA.FLAG) which suppresses the base-limit test and indicates that there is a following 10-bit field (IPCD.INDEX) which supplies an index into the IPC.DICTIONARY. With the absolute address obtained from the IPC.DICTIONARY, the interpreter can proceed as if it had an absolute address for a normal data item.

The execution of this S-op results in the construction and insertion of an 80-bit System Descriptor for the COP in the IPC.DICTIONARY at the displacement address specified by V. The base address of the IPC.DICTIONARY is adjacent to the program's RSN and can be found in RS.IPC.DICTIONARY.SPACE as an absolute address. The BADDR contains the address of the first IPCD S-op which references an operand that lies in an overlayable data segment. This address is required by the interpreter when checkerboarding has occurred and a necessary data segment does not fit into dynamic memory without overlaying another required data segment.

INDEX