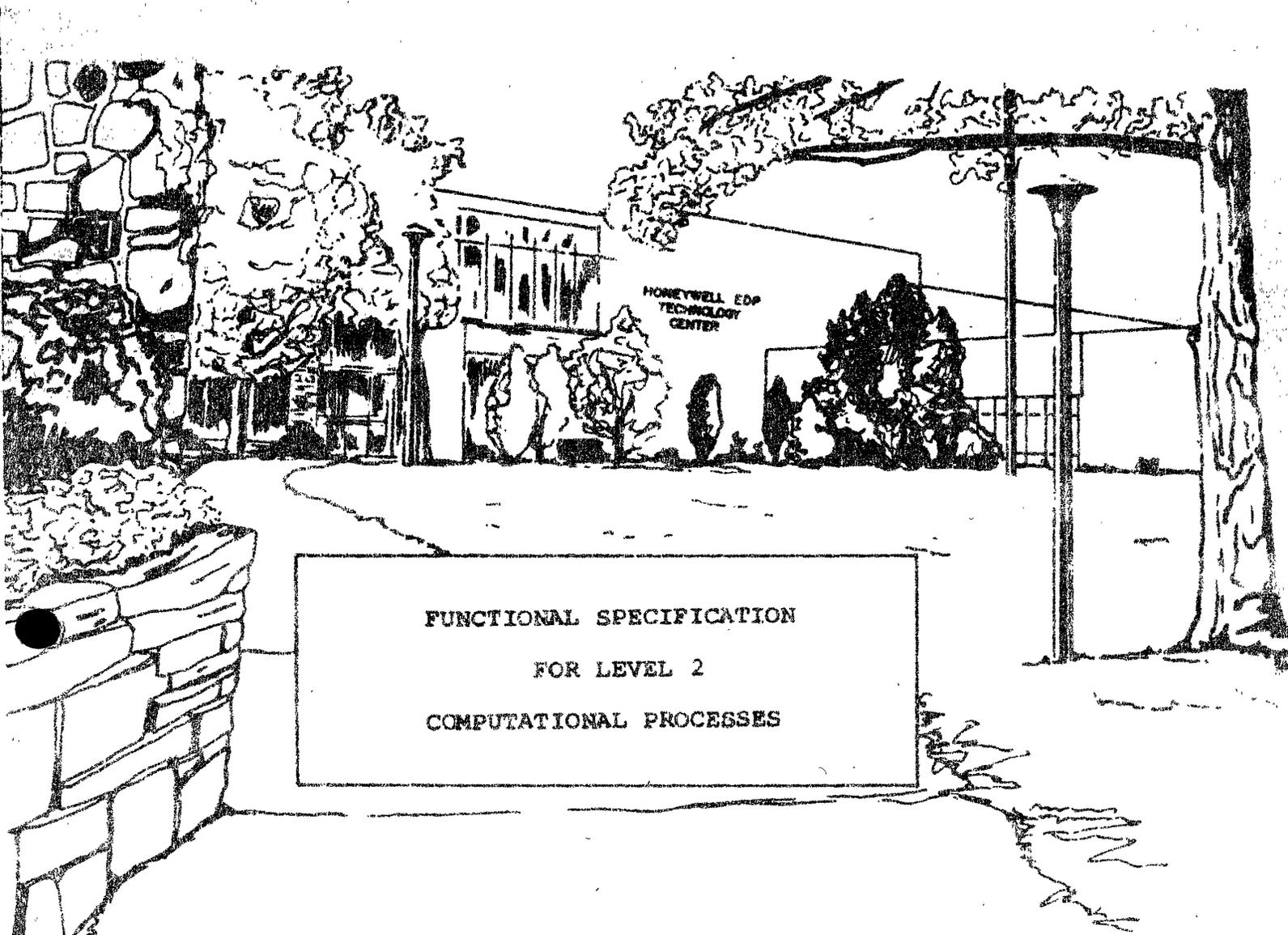


Preliminary Copy
Circulating For COMMENTS

Date: June 13, 1969
Doc. No: SY2F-00
Rev. Draft



FUNCTIONAL SPECIFICATION
FOR LEVEL 2
COMPUTATIONAL PROCESSES

Prepared by W. Bean
SYSTEMS GROUP

M. Firdman
SYSTEMS GROUP

G. Holt
SYSTEMS GROUP

E. McFaden
PUBLICATIONS

Approved by _____



PROPRIETARY NOTICE

The information and design of the system described herein were originated by and are the property of Electronic Data Processing Division, Honeywell, Inc. The content of this document is Honeywell Proprietary - Sensitive and is for internal use only. Such information may not be reproduced, disclosed to others, or used by others for any purpose without written permission from an authorized Honeywell official.

NOTE

The demands of expediency in publishing this preliminary document have precluded the editorial attention usually given to such documents. The reader should consider this if inconsistencies, redundancies, or errors become evident herein.

TABLE OF CONTENTS

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
<u>SECTION II - FORMAT REPRESENTATIONS AND DESCRIPTIONS</u>		
2.1	DATA REPRESENTATIONS AND DATA CONTROL DESCRIPTIONS	2-1
2.2	DATA REPRESENTATIONS	2-1
2.2.1	Variable - Length Data Representations	2-1
2.2.2	Fixed - Length Tagged Data Representations	2-6
2.3	DATA DESCRIPTION	2-7
2.3.1	Structors	2-8
2.3.1.1	Implicit - Length Structor	2-8
2.3.1.2	Explicit - Length Structor	2-12
2.3.2	Ministructors	2-17
2.3.2.1	Singular Ministrator	2-17
2.3.2.2	Dual Ministrator	2-18
2.3.2.3	Array Ministrator	2-19
2.3.2.4	String Ministrator	2-20
2.3.3	Microstructors	2-22
2.3.3.1	Bit/Binary String Microstructor	2-23
2.3.3.2	Byte String Microstructor	2-23
2.3.3.3	Implicit Microstructor	2-24
2.4	CONTROL INFORMATION REPRESENTATION	2-25
2.4.1	System Control Structors	2-25
2.4.1.1	Procedure Index	2-27
2.4.1.2	TSB Identifiers	2-28
2.4.1.3	Protected Elements	2-28
2.4.1.4	Trap Effectors	2-29
2.4.2	System Base	2-29
2.4.3	Task Status Block	2-31
2.4.4	Task Priority Array	2-35
2.4.5	I/O Start Array	2-35
2.4.6	External Start Array	2-36
2.4.7	Processor Status Array	2-37
2.5	I/O INFORMATION REPRESENTATION	2-38
2.5.1	I/O Structors	2-38

TABLE OF CONTENTS (Cont.)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
2.5.1.1	Device Identifier Structor	2-39
2.5.1.2	I/O Command Structor	2-39
2.5.1.3	I/O Control Command Specifier	2-40
2.5.2	I/O Operands	2-40
2.5.3	Device Specification Table	2-42
2.5.4	Traffic Registers	2-42
2.5.5	Simultaneity Table	2-43
2.5.6	Input/Output Status Word	2-43

SECTION III - INSTRUCTION FORMATS
AND INSTRUCTION EXTRACTION

3.1	GENERAL	3-1
3.2	INSTRUCTION FORMATS	3-3
3.2.1	Register-Register (RR) Format	3-3
3.2.2	Register-Selector (RS and RL) and Selector-Register (SR and LR) Formats	3-3
3.2.3	Selector-Selector (SS, SL, LS, LL) Formats	3-4
3.2.4	Relative Displacement (RD) Format	3-5
3.2.5	Control Variant (CV) Format	3-6
3.3	INSTRUCTION EXTRACTION	3-6
3.3.1	Register-Register (RR) Format Extraction	3-6
3.3.2	Register-Selector (RS, RL) and Selector-Register (SR, LR) Format Extraction	3-8
3.3.2.1	Literal Value Case	3-8
3.3.2.2	Implicit-Length Base Reference Case	3-8
3.3.2.3	Explicit-Length Base Reference Case	3-9
3.3.2.4	Indexed Implicit-Length Base Reference Case	3-11
3.3.2.5	Indexed Explicit-Length Base Reference Case	3-13
3.3.2.6	Auto-indexed Implicit-Length Base Reference Case	3-16
3.3.2.7	Auto-indexed Explicit-Length Base Reference Case	3-17
3.3.2.8	Auto-Qualified Base Reference Case	3-21
3.3.2.9	Multiple Selection Case	3-23

TABLE OF CONTENTS (Cont.)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
3.3.3	Selector-Selector (SS, SL, LS, LL) Format Extraction	3-30
3.3.4	Relative Displacement (RD) Format Extraction	3-30
3.3.5	Control Variant (CV) Format Extraction	3-35

SECTION IV - AUTOFETCH/AUTOSTORE

4.1	GENERAL	4-1
4.2	AUTOFETCH/AUTOSTORE CONVERSION	4-1
4.2.1	Autofetch/Autostore Conversion For Tagged Doublewords	4-2
4.2.2	Autofetch/Autostore Conversion For Ministructors	4-2
4.2.2.1	Autofetch of Ministructor Operands	4-3
4.2.2.2	Autostore Into Ministructor Operands	4-5
4.2.3	Autofetch Conversion For Microstructure	4-9
4.2.3.1	Autofetch of Microstructure Operands	4-9
4.2.4	Autofetch/Autostore Conversion for Deferred Selection Structures	4-11
4.2.5	Autofetch/Autostore Conversion for Bit Strings	4-12
4.2.6	Autofetch/Autostore Conversion for Binary Strings	4-14
4.2.7	Autofetch/Autostore Conversion for Floating Point Strings	4-16
4.3	AUTOFETCH EVALUATOR	4-18
4.4	AUTOSTORE EVALUATOR	4-21

SECTION V - AUTOCONVERSION

5.1	INTRODUCTION	5-1
5.2	CONVERSION CONVENTIONS	5-2
5.2.1	Tagged Logical Word to Tagged Binary Integer	5-2
5.2.2	Tagged Binary Integer to Tagged Floating Point Number	5-3
5.2.3	Tagged Floating Point to Decimal String	5-4
5.2.4	Decimal String to Tagged Floating Point	5-6

TABLE OF CONTENTS (Cont.)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
5.2.5	Floating Point Number to Tagged Binary Integer	5-7
5.2.6	Tagged Binary Integer to Tagged Logical Word	5-8

SECTION VI - INSTRUCTIONS

6.1	DATA MANIPULATION ISTRUCTIONS	6-1
6.1.1	Add/Subtract	6-2
6.1.1.1	Logical Binary Addition/Subtraction	6-3
6.1.1.2	Twos Complement Binary Addition/Subtraction	6-4
6.1.1.3	Hexadecimal Floating Point Addition/Subtraction	6-4
6.1.1.4	Decimal String Addition/Subtraction	6-6
6.1.2	Multiply	6-10
6.1.2.1	Logical Binary Multiplication	6-11
6.1.2.2	Twos Complement Binary Multiplication	6-12
6.1.2.3	Hexadecimal Floating Point Multiplication	6-13
6.1.2.4	Decimal String Multiplication	6-16
6.1.3	Divide	6-18
6.1.3.1	Logical Binary Division	6-19
6.1.3.2	Twos Complement Binary Division	6-20
6.1.3.3	Hexadecimal Floating Point Division	6-21
6.1.3.4	Decimal String Division	6-24
6.1.4	Compare	6-27
6.1.4.1	Logical Binary Comparison	6-27
6.1.4.2	Twos Complement Binary Comparison	6-28
6.1.4.3	Hexadecimal Floating Point Comparison	6-28
6.1.4.4	Decimal String Comparison	6-28
6.1.4.5	Byte String/Translated Byte String Comparison	6-29
6.1.5	Move	6-31
6.1.5.1	Byte String/Translated Byte String Move	6-31
6.1.5.2	Decimal String to Decimal String Move	6-32
6.1.5.3	Autostore Moves	6-33
6.1.6	And/Or/Exclusive OR	6-33
6.1.6.1	Logical Word And/Or/Exclusive OR	6-33

TABLE OF CONTENTS (Cont.)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
6.1.6.2	Byte String/Translated Byte String AND, OR, EXCLUSIVE OR	6-34
6.1.7	Shift	6-35
6.1.7.1	Single Precision Logical Left Shift	6-36
6.1.7.2	Single Precision Logical Right Shift	6-36
6.1.7.3	Single Precision Arithmetic Left Shift	6-37
6.1.7.4	Single Precision Arithmetic Right Shift	6-37
6.1.7.5	Double Precision Logical Left Shift	6-38
6.1.7.6	Double Precision Logical Right Shift	6-38
6.1.7.7	Double Precision Arithmetic Left Shift	6-39
6.1.7.8	Double Precision Arithmetic Right Shift	6-39
6.1.7.9	Single Precision Rotational Shift	6-40
6.1.8	Load Positive/Load Negative	6-40
6.1.8.1	Twos Complement Binary Loading	6-40
6.1.8.2	Hexadecimal Floating Point Loading	6-41
6.1.9	Load Complement	6-41
6.1.9.1	Logical Binary Negation	6-42
6.1.9.2	Twos Complement Binary Negation	6-42
6.1.9.3	Hexadecimal Floating Point Negation	6-42
6.1.10	Load and Test	6-42
6.1.10.1	Logical Binary Testing	6-43
6.1.10.2	Twos Complement Binary Testing	6-43
6.1.11	Edit	6-43
6.2	GENERAL REGISTER LOADING/STORING INSTRUCTIONS	6-43
6.2.1	Copy	6-44
6.2.2	Load	6-44
6.2.3	Fetch	6-44
6.2.4	Convert to Logical	6-45
6.2.5	Convert to Binary	6-45
6.2.6	Convert to Floating	6-46
6.2.7	Dump Multiple	6-46
6.2.8	Undump Multiple	6-48
6.2.9	Dump	6-49

TABLE OF CONTENTS (Cont.)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
6.2.10	Dump	6-50
6.2.11	Store	6-50
6.2.12	Deposit	6-50
6.3	BRANCHING INSTRUCTIONS	6-51
6.3.1	Test and Branch	6-51
6.3.2	Conditional Branch	6-52
6.3.3	Branch and Link	6-53
6.3.4	Branch On Incremented Count/Branch On Decrementd Count	6-53
6.4	STRUCTOR MANIPULATION INSTRUCTIONS	6-54
6.4.1	Select	6-54
6.4.2	Lower Subarray	6-55
6.4.3	Upper Subarray	6-55
6.4.4	Point	6-55
6.4.5	Initial Substring	6-56
6.4.6	Terminal Substring	6-57
6.5	TASK CONTROL INSTRUCTIONS	6-58
6.5.1	Stop	6-58
6.5.2	Start	6-59
6.5.3	Suspend	6-60
6.5.4	Conditional Stop	6-61
6.5.5	I/O External Conditional Stop	6-61
6.5.6	Load Status	6-64
6.5.7	Test and Set	6-65
6.5.8	Set Mode - Reset Mode	6-66
6.5.9	Field Extract	6-67
6.5.10	Field Substitute	6-68
6.6	INPUT/OUTPUT INSTRUCTIONS	6-70
6.6.1	Initial Device Operation	6-70
6.6.2	Halt Device Operation	6-70

SECTION VII - TASK MULTIPLEXING

1.1	INTRODUCTION	7-1
-----	--------------	-----

HONEYWELL PROPRIETARY - SENSITIVE

TABLE OF CONTENTS (Cont.)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
7.2	LOCK AND UNLOCK FUNCTIONS	7-1
7.3	DISPATCH OPERATION	7-2
7.4	I/O INITIATED STARTS	7-4
7.5	EXTERNALLY INITIATED STARTS	7-5

SECTION VIII - TRAPPING

8.1	INTRODUCTION	8-1
8.2	TRAPPING INFORMATION STRUCTURE	8-1
8.3	TRAP CAUSES	8-1
8.4	TRAP MECHANIZATION	8-4

SECTION IX - TIMING FACILITIES

9.1	SYSTEM CLOCK	9-1
9.2	SYSTEM TIMER	9-1
9.3	TASK TIMER	9-1

SECTION X - INPUT/OUTPUT FACILITIES

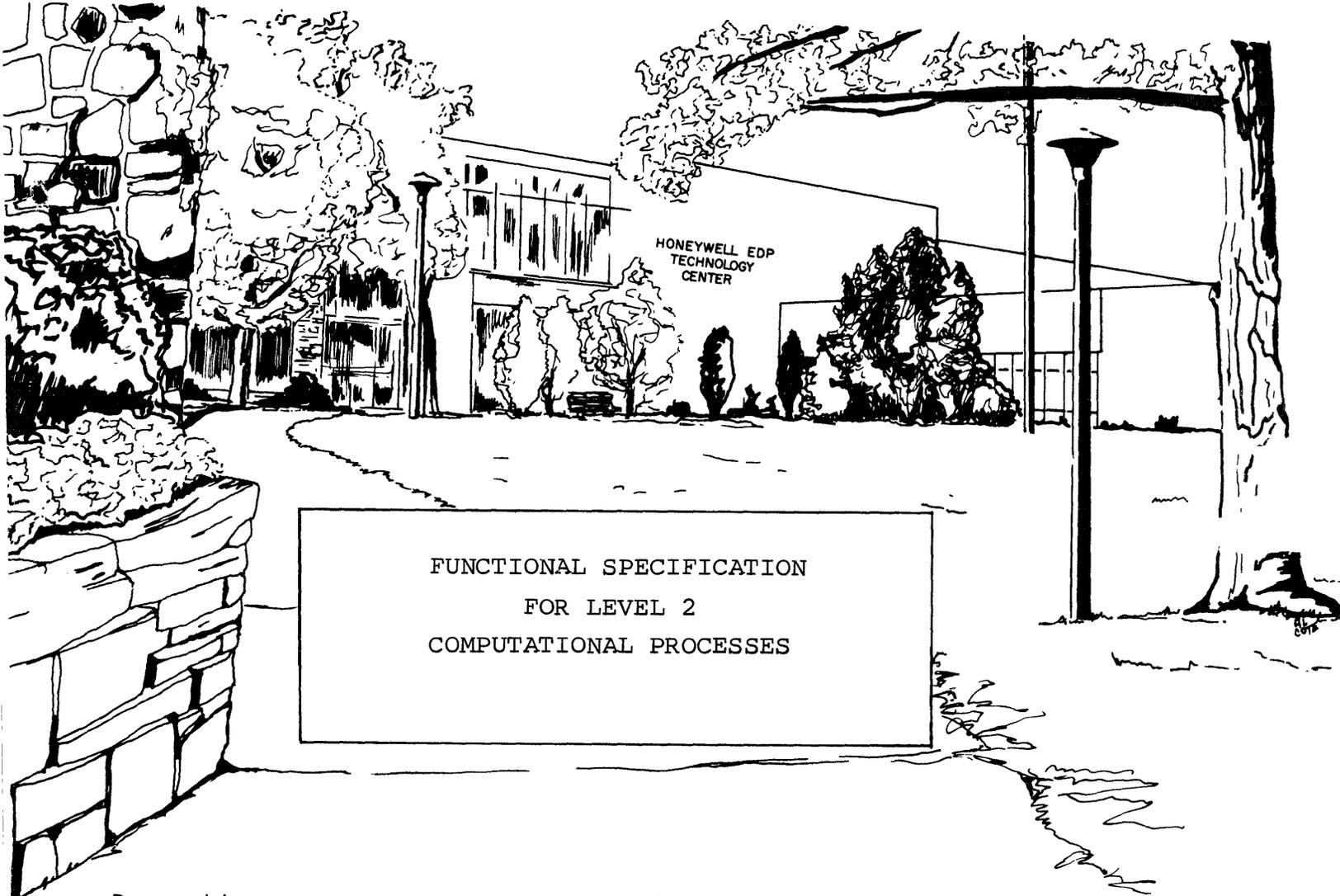
10.1	INPUT/OUTPUT OPERATIONS	10-1
10.2	CENTRAL PROCESSOR I/O INSTRUCTIONS	10-3
10.2.1	Initiate Device Operation Instruction	10-4
10.2.1.1	Extraction of IDO Orders Specifying Command Array	10-9
10.2.1.2	Extraction of IDO Orders Specifying A Single Control Command	10-14
10.2.1.3	The I/O Status Information	10-15
10.2.2	Halt Device Operation (HDO) Instruction	10-15
10.2.3	Allowable Structure Formats for I/O Instruction	10-17
10.3	INPUT/OUTPUT INTERRUPTS	10-18
10.3.1	The I/O Start Array	10-18
10.3.2	The I/O Status Word	10-18
10.3.3	Execution of I/O Interrupts	10-21

TABLE OF CONTENTS (Cont.)

<u>Page</u>	<u>Title</u>	<u>Page</u>
	<u>APPENDIX</u>	
A-1	DECIMAL STRING SIGN CODES	A-1

EM:11 Copy No. 43
No. of Pgs. _____
Issued to _____
The content of this document is
HONEYWELL PROPRIETARY SENSITIVE
and is not to be reproduced.

Doc. No: FTL-003
Date September 8, 1969
Rev. Draft 2



FUNCTIONAL SPECIFICATION
FOR LEVEL 2
COMPUTATIONAL PROCESSES

Prepared by:

Approved by:

W. Bean
SYSTEMS GROUP
M. Ferdman
SYSTEM GROUP
G. Holt
SYSTEM GROUP
E. McFaden
PUBLICATIONS



ELECTRONIC DATA PROCESSING
TECHNOLOGY CENTER

PROPRIETARY NOTICE

The information and design of the system described herein were originated by and are the property of Electronic Data Processing Division, Inc. The content of this document is Honeywell Proprietary - Sensitive and is for internal use only. Such information may not be reproduced, disclosed to others, or used by others for any purpose without written permission from an authorized Honeywell official.

SECTION II
FORMAT REPRESENTATIONS AND DESCRIPTIONS2.1 DATA REPRESENTATIONS AND DATA AND CONTROL DESCRIPTIONS

The following sections describe the data representation formats, as well as the formats for data and control structures. Structures are used to describe collections of data elements, control sequential and parallel instruction sequencing, and initiate input/output operations.

It should be noted that throughout the following sections any field designated RESERVED must contain binary zeros.

2.2 TAGGED INFORMATION REPRESENTATIONS

Tagged information consists of self-descriptive data representations and structures. This information is self-descriptive in the sense that its format includes a 4-bit field that specifies the interpretation appropriate to the remainder of the format. This 4-bit field is called a TAG field and is assigned the interpretations specified in Table 2-1.

All tagged information is 64-bits in length, with the leftmost 4 bits assigned to the TAG field. The format of the remaining 60 bits depends on the particular type of information being represented. These formats are specified in succeeding subsections.

All information stored in general registers is tagged information. Tagged information may also be stored in main storage. In the latter case, each tagged information unit must originate at a storage address that is a multiple of 8 (doubleword boundary alignment). Arrays containing tagged items are called tagged doubleword arrays.

TABLE 2-1
TAG ASSIGNMENTS

TAG	INTERPRETATION
0	Tagged Logical Word
1	Tagged Binary Integer
2	Tagged Hexadecimal Floating Point Number
3	Unassigned*
4	Explicit-Length, Modifier, Alterable Structor
5	Explicit-Length, Modifier, Nonalterable Structor
6	Explicit-Length, Specifier, Alterable Structor
7	Explicit-Length, Specifier, Nonalterable Structor
8	Implicit-Length, Baselink, Structor
9	Implicit-Length, Baselink, Structor
A	Implicit-Length, Data link, Alterable Structor
B	Implicit-Length, Data link, Nonalterable Structor
E	Unassigned*
F	System Control Structor

*Unassigned TAG codes are reserved for future functional extensions.

2.2.1 Tagged Data Representations

The tagged data representations consist of a TAG field and a fixed-length data field. The available tagged data representations are: tagged logical words, tagged binary integers, and tagged hexadecimal floating point numbers.

2.2.1.1 Tagged Logical Word

A tagged logical word is a 64-bit quantity, consisting of a TAG field (which is hexadecimal 0), a 28-bit reserved field, and a 32-bit value that is treated as a bit string of fixed-length (see Figure 2-1). This quantity must be aligned on a doubleword boundary in storage. It may also appear in a general purpose register.

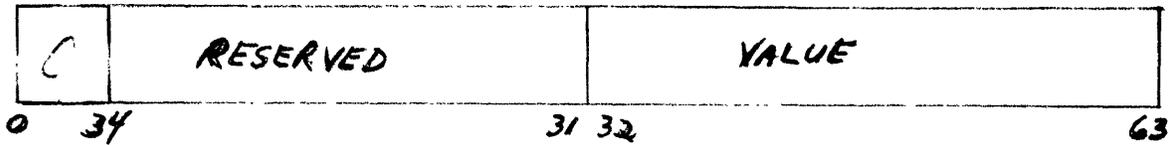


FIGURE 2-1. TAGGED LOGICAL WORD

The value field of a tagged logical word may be interpreted as either a 32-bit logical quantity or as a 32-bit unsigned binary integer.

2.2.1.2 Tagged Binary Integer

A tagged binary integer is a 64-bit quantity, consisting of a TAG field (which is hexadecimal 1), a 28-bit reserved field, and a 32-bit value that is treated as a binary two's complement integer of a fixed-length (see Figure 2-2). This quantity must be aligned on a doubleword boundary in storage. It may also appear in a general purpose register.

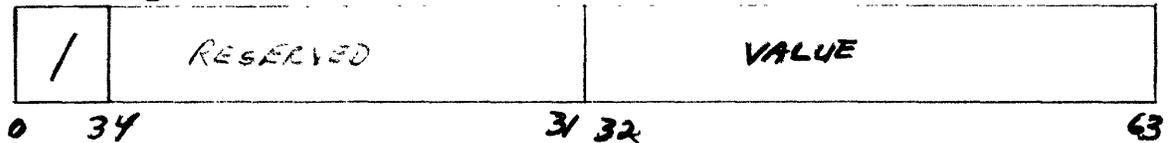


FIGURE 2-2. TAGGED BINARY INTEGER

2.2.1.3 Tagged Hexadecimal Floating Point Number

A tagged hexadecimal floating point number is a 64-bit quantity, consisting of a TAG field (which is 2), a sign bit, a seven-bit excess 64 exponent field, and a 13 digit hexadecimal mantissa (see Figure 2-3). A sign bit of 0 indicates that the mantissa is positive. This quantity must be aligned on a double word boundary in storage. It may also appear in a general purpose register.

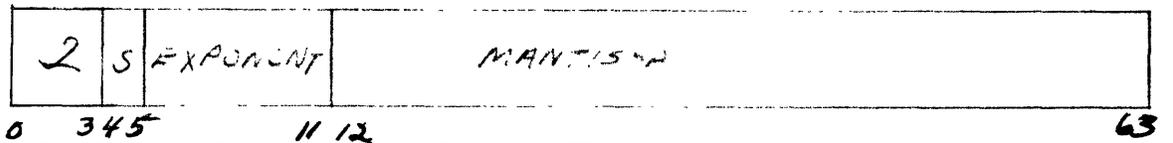


FIGURE 2-3. TAGGED HEXADECIMAL FLOATING POINT NUMBER

2.2.2 Data Structors

Data Structors are entities used for structural description. There are two basic forms of data structors, called implicit-length structors and explicit-length structors, that are used to describe arrays of fixed-length items and arrays of variable-length items, respectively. Data structors are tagged quantities and may be stored in general purpose registers or in storage. The number of storage accesses required to access an operand described by a data structor can be minimized by placing the structor in a general purpose register.

Explicit-length structors can assume one of two forms, called the modifier form and the specifier form. A modifier/specifier (M/S) indicator is included in the explicit-length structor TAG field to distinguish between these two forms. The M/S indicator is used to signal whether the structor specifies a particular data structure at a fixed main storage location or is used only to modify descriptions of areas of storage to conform to a desired data structure specification. Implicit-length structors can also assume one of two forms, called the baselink form and the data link form. The implicit-length structor TAG field distinguishes between these two forms and is used to specify whether the structor can be used as an indirection link for base reference creation or for effective operand formation, respectively.

Data structors specify several other important attributes of an information structure. In particular, the type of information described, the alterability of the information, and the number of items of information of the designated type are described by structors. Other attributes are given for certain types of information units. The attributes given in the data structor apply to all items in the array described by the structor. One type of array, called

2.2.2 tagged doubleword, allows any tagged quantity to be assigned
 (Cont.) to any item in the array, so that some of the attributes
 of this array item are associated with it, independent of
 the attributes of other array items.

The location field in data structures is 24 bits in length
 and identifies one of as many as 1,048,576 (2^{20}) 8-bit
 bytes of storage. The addressing resolution required to
 locate a particular bit is achieved by use of an offset
 field contained in certain data structures. The location
 field, together with this offset field, always identifies
 the position of the leftmost (lowest numbered) bit of a
 particular array of items in storage.

The general format for data structures is presented in
 Figure 2-4.

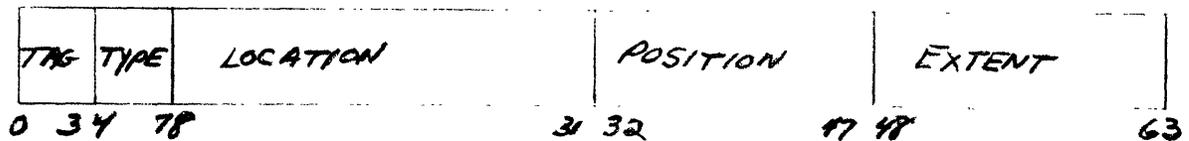


FIGURE 2-4. DATA STRUCTOR (GENERAL FORMAT)

The descriptions of the individual fields of data structures
 are specified as follows:

- a. TAG field -- this 4 bit field distinguishes between
 the implicit -- and explicit modifier and specifier
 or baselink and data link forms, and alterable and
 nonalterable cases of data structures. (See Table 2-1).
 These cases are specified as follows:
 - i. Implicit-length/Explicit-length-Implicit-Length
 items always have a fixed bit length, which is
 determined by their TYPE field. Explicit-length
 items have a specified bit length, which is
 derived from the POSITION field of their describing
 data structure.

2.2.2
(Cont.)

- ii. Modifier/Specifier and Baseline/Data link -- For explicit-length modifier structures, the LOCATION field is interpreted as a relative byte displacement from an implied base location. For explicit-length specifier structures, the LOCATION field is interpreted as an absolute storage location at which the associated data structure originates. An implicit-length baseline or data link structure LOCATION field is always interpreted as the absolute storage location of a tagged doubleword or ministrator array origin. The modifier/specifier and baseline/datalink indicator is the third bit of the TAG field of a data structure.
- iii. Alterable/Nonalterable -- An alterable structure specifies that items in its associated array are alterable when the structure is used to access them, while a nonalterable structure prevents alteration of items in its associated array. Alterability indicator is the fourth bit of the TAG field of a data structure.
- b. TYPE Field -- This 4-bit field is used to identify the type of implicit -- or explicit -- length array items described by the structure. For implicit-length structures, the length of each item is specified by this field. The interpretation of the POSITION Field, which contains the item length, is specified by the TYPE Field for explicit-length structures. The available TYPE codes are presented in Table 2-2 and Table 2-3.
- c. LOCATION Field -- This 24-bit field specifies the byte location (0 - 1,048,575 with values greater than 1,048,575 illegal) in which the first array item has its origin. For explicit-length modifier structures, this field specifies the number of bytes of relative displacement from an implied base reference location as the location of the array described by the structure.

For explicit-length specifier structors and for implicit-length structors, this field specifies the absolute storage address of the byte in which the array associated with the structor originates. For bit and binary strings, the bit offset subfield of the POSITION field of the structor is also required to establish the array origin.

- d. POSITION Field -- The interpretation of this 16-bit field depends on the TAG and TYPE field codes. The specific interpretations are considered in subsection 2.2.2.1 below.
- e. EXTENT Field -- This 16-bit field specifies the number of items of the designated TYPE in the array described by the data structor. Extents 1 to 65,536 are associated with values of all zero bits to all one bits in this field.

Data structors must be aligned on doubleword boundaries. The entire array of items described by the data structor must be placed in contiguous storage locations. Every item in the array is a data representation with the same attributes, except for storage location.

The modifier form of explicit-length structor cannot be used to access an operand in storage, since it does not describe any particular collection of items in storage. The use of data structors to fetch or store operands is discussed in Subsections 3.2 and 3.3 Autofetch/Autostore.

2.2.2.1 POSITION Field of Data Structors

The interpretation of the POSITION field of data structors depends on the values of the TAG and TYPE fields of the structor. For explicit-length structors, the interpretations are specified in the subsections defining the explicit-length items. For implicit-length structors, the following

TABLE 2-2
TYPE CODES FOR EXPLICIT-LENGTH STRUCTORS

TYPE CODE	ITEM DESCRIPTION	UNIT SIZE (BITS)
0	Bit String	1
1	Binary String	1
2	Hexadecimal Floating Point String	8
3	Unassigned*	-
4	Zoned Decimal String	8
5	Unsigned Zoned Decimal String	8
6	Packed Decimal String	8
7	Unsigned Packed Decimal String	8
8	Byte String	8
9	Translated Byte String	8
A	Unformatted Region	8
B	Edit Control String	8
C-D	Unassigned*	8
E-F	Software Assignable*	

*An attempt to use an explicit-length structor with this TYPE code will normally result in a trap.

TABLE 2-3
TYPE CODES FOR IMPLICIT-LENGTH STRUCTORS

TYPE	ITEM DESCRIPTION	ITEM SIZE (BITS)
0	Tagged Doubleword	64
1	Tagged Doubleword, LIFO Access	64
2	Tagged Doubleword, FIFO Access	64
3	Ministructor	32
4-D	Unassigned*	n.a.
E-F	Software Assignable	n.a.

*An attempt to use an implicit-length structor with this TYPE code will normally result in a trap.

interpretations are specified:

- a. Tagged Doubleword -- the POSITION Field is RESERVED.
- b. Tagged Doubleword, LIFO Access -- the POSITION field is interpreted as an unsigned binary integer in the range 0 to 65,535. This integer, multiplied by 8 and added to the value of the LOCATION field of the structor, selects a particular tagged doubleword in the array of tagged doublewords described by the structor. The POSITION field value must be not greater than the EXTENT field value. When an item in a tagged doubleword, LIFO access array is retrieved or updated, the value of the POSITION field in the associated data structor may be decremented or incremented, respectively.
- c. Tagged Doubleword, FIFO Access -- The POSITION field is interpreted as a pair of 8-bit unsigned binary integers in the range 0 to 255. These integers multiplied by 8 and added to the value of the LOCATION field of the structor, select a pair of tagged doublewords in the array of tagged doublewords described by the structor. Each of these integers must be not greater than the EXTENT field value. When an item in a tagged doubleword, FIFO array is retrieved, the leftmost 8-bits of the POSITION field, called the FIFO tail, may be used to select an item and may be incremented. When an item in a tagged doubleword, FIFO array is updated, the rightmost 8-bits of the POSITION field, called the FIFO head, may be used to select an item and may be incremented.
- d. Ministructor -- the POSITION field is RESERVED.

2.2.3 System Control Structors

System control structors are tagged information items used for system control operations. These structors are specified in Sections 2.5 (Control Information Representation) and Section 2.6 (I/O Information Representation).

2.3 COMPACT INFORMATION REPRESENTATIONS

The compact information representations are used to minimize the amount of storage required to store arrays of information containing items with certain homogeneous attributes. Each compact representation is equivalent to one of the tagged representations specified in Section 2.2.

2.3.1 Compact Data Representations

There are three compact data representations: bit strings, binary strings, and hexadecimal floating point strings. These data representations possess values over the same range as tagged logical words, tagged binary integers, and tagged hexadecimal floating point numbers, respectively.

2.3.1.1 Bit Strings

A bit string consists of a sequence of bits of specified length treated as a variable precision logical word or unsigned binary integer (see Figure 2-5). The maximum length for bit strings is 32 bits. The leftmost bit of a bit string may be any bit position in any byte of storage. Alignment and string length for bit strings affect their access time.



FIGURE 2-5. BIT STRING

Bit strings are described by explicit-length structors. The interpretation of the POSITION field of bit string structors is as follows. (See Figure 2-6).

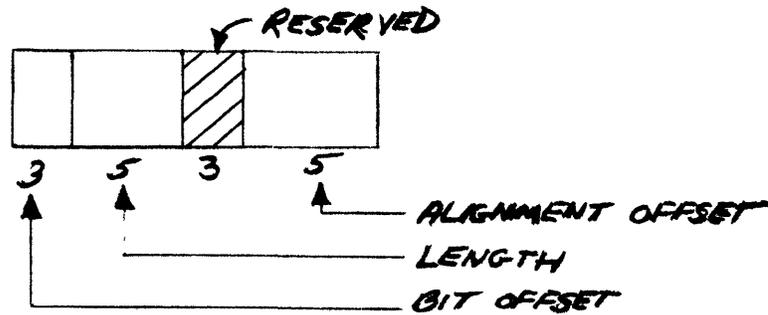


FIGURE 2-6. POSITION FIELD FOR BIT AND BINARY STRINGS

The bit offset field specifies the bit position within the byte addressed by the LOCATION field of the structor that is associated with the leftmost bit of an array of bit strings. The bit offset has a range 0 to 7. The length field specifies the number of bits in each bit string item in the array. The length has a range 1 to 32, 1 to 31 associated with binary values 00001 to 11111, and 00000 associated with a length of 32 bits. The alignment offset field specifies the offset of the bit string in a tagged logical word and is used in Autofetch/Auto-store conversion for bit strings. This field has a range of 0 to 31, associated with binary values 00000 to 11111.

2.3.1.2 Binary Strings

A binary string consists of a sequence of bits of specified length treated as a variable precision two's complement binary integer (see Figure 2-7). The maximum length for binary strings is 32 bits. The leftmost bit of a binary string may be any bit position of any byte of storage. Alignment and string length for binary strings may affect their access time.

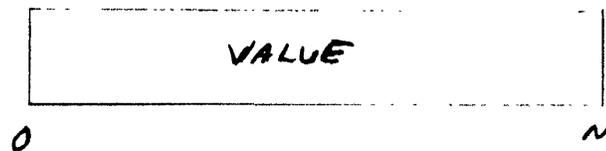


FIGURE 2-7. BINARY STRING

Binary strings are described by explicit-length structors. The interpretation of the POSITION field of binary string structors is identical to the interpretation for bit strings (see subsection 2.3.1.1) except the binary string alignment offset specifies the offset of the binary string within a tagged binary integer value.

2.3.1.3 Hexadecimal Floating Point Strings

A hexadecimal floating point string consists of an 8-bit sign exponent byte followed by a sequence of from 0 to 7 contiguous bytes, which form a 0 to 14 hexadecimal digit mantissa. (See Figure 2-8.) The leftmost bit of the string is the sign of the manissa, encoded as 0 plus and 1 minus. The next seven bits contain the exponent. The exponent is encoded as an excess 64 number with a range of -64 through +63 and is interpreted as a power of sixteen. The remainder of the string consists of the hexadecimal digits used to encode the mantissa.

The minimum length of a hexadecimal floating point string (including the sign/exponent byte) is one byte, which corresponds to a zero digit mantissa. The maximum length is 8 bytes, which corresponds to a 14 digit mantissa.

A hexadecimal floating point string must be aligned on a byte boundary in storage. Boundary alignment and string length for hexadecimal floating point strings may affect their access time.

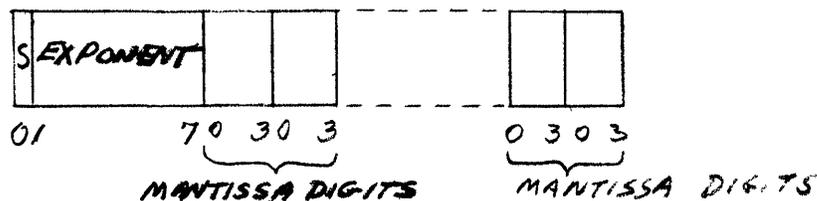


FIGURE 2-8. FLOATING POINT HEXADECIMAL STRING

Hexadecimal floating point strings are described by explicit-length structors. The interpretation of the

POSITION field of hexadecimal floating point string structor is as follows. (See Figure 2-9.)

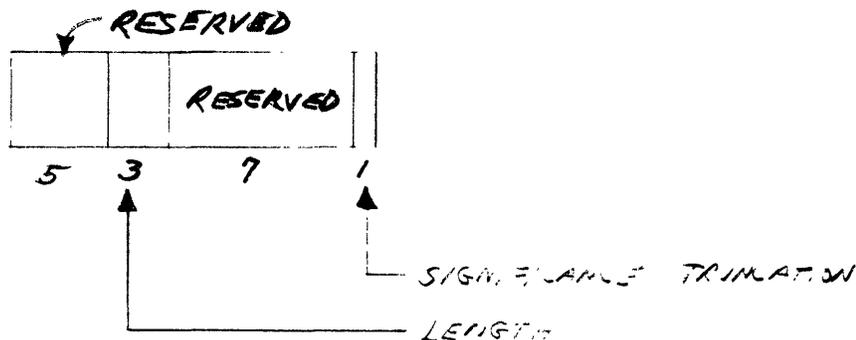


FIGURE 2-9. POSITION FIELD FOR HEXADECIMAL FLOATING POINT STRINGS

The length field specifies the number of bytes in each hexadecimal floating point string in the array described by the structor. The length has a range of 1 to 7 and 8, encoded as 001 to 111 and 000. The significance truncation field specifies the effective mantissa precision when the hexadecimal floating point string is manipulated in significance arithmetic mode. If this field is 1, significance truncation applies; otherwise, it does not.

2.3.2 Ministructors

A reduction in the number of bits required to describe a particular information structure is possible if the attributes of the structure are limited. The most important limitations are on the location and extent of arrays and on string length. Location information is reduced by requiring the data structure to be located a fixed relative displacement from the location of the descriptive quantity.

Arrays of less than a given extent and strings of less than a given length will require fewer bits in their associated descriptive quantities. The descriptive quantities assuming these abbreviated forms of description are called ministructors. Ministructors are normally stored in arrays and are converted into equivalent forms of data

2.3.2.2 Dual Ministrator

The dual ministrator is a 32-bit quantity consisting of a two-bit classifier (which is 01), a one-bit modifier/specifier or baselink/datalink indicator, a one-bit alterability indicator, a four-bit typecode field, and a 24-bit location field. (See Figure 2-11.)

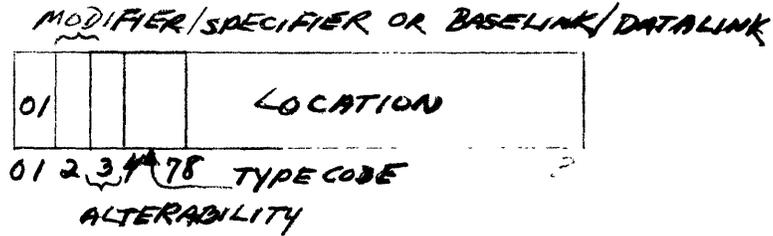


FIGURE 2-11. DUAL MINISTRUTOR

A dual ministrator is equivalent to a data structor with the following attributes:

- a. Modifier/specifier or baselink/datalink indicator and alterability indicator identical to the ministrator indicators.
- b. LOCATION field identical to the ministrator location field.
- c. EXTENT field set to one (two items).
- d. the TAG, TYPE, and POSITION fields of the data structor are determined by the typecode field of the ministrutors, according to Table 2-4.

This form of ministrator is useful in constructing binary tree structures, including chained lists and ring structures.

TABLE 2-4
MINISTRUCTOR EQUIVALENT STRUCTOR ATTRIBUTES

TYPE CODE	TAG*	TYPE	POSITION**
0	Implicit	Tagged doubleword	n.a.
1	Implicit	Ministructor	n.a.
2-7	Unassigned	n.a.	n.a.
8	Explicit	Bit String (0)	B=0, L=8, A=0
9	Explicit	Bit String (0)	B=0, L=16, A=0
A	Explicit	Bit String (0)	B=0, L=32, A=0
B	Explicit	Binary String (1)	B=0, L=8, A=24
C	Explicit	Binary String (1)	B=0, L=16, A=16
D	Explicit	Binary String (1)	B=0, L=32, A=0
E	Explicit	Hex.f.p. String (2)	L=4, S=0
F	Explicit	Hex.f.p. String (2)	L=8, S=0

*The TAG field is also generated to include modifier/specifier or baselink/datalink and alterability on indicators.

**The following abbreviations are used: B-bit offset, L-length, A-alignment offset, S-significance truncation.

2.3.2.3 Array Ministructor

The array ministructor is a 32 bit quantity consisting of a two-bit classifier (which is 10), a one-bit modifier/specifier or baselink/datalink indicator, a one-bit alterability indicator, a four-bit type field, a sixteen-bit relative displacement field, and a eight-bit extent field. (See Figure 2-12.)

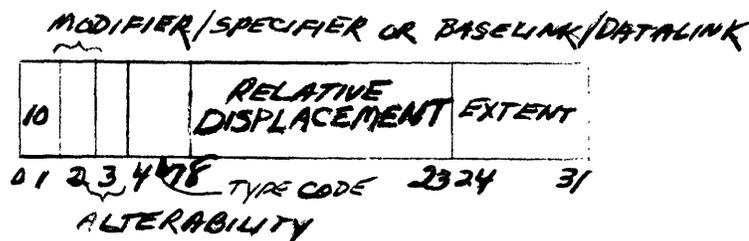


FIGURE 2-12. ARRAY MINISTRUCTOR

An array ministrator is equivalent to a data structor with the following attributes:

- a. Modifier/specifier indicator or baselink/datalink and alterability indicator identical to the ministrator indicators.
- b. LOCATION field equals the value of the relative displacement field of the ministrator in bytes, if the ministrator is an explicit-length modifier or implicit-length baselink or the sum of the storage address of the ministrator plus the relative displacement field of the ministrator, when the ministrator is an explicit-length specifier or implicit-length datalink type.
- c. EXTENT field set to the value of the extent field of the array ministrator.
- d. The TAG, TYPE, and POSITION fields of the data structor are determined by the typecode field of the ministrator, according to Table 2-4.

2.342.4 String Ministrator

The string ministrator is a 32-bit quantity consisting of a two bit classifier (which is 11), a one-bit modifier/specifier indicator, a one-bit alterability indicator, a four-bit type code field, a sixteen bit relative displacement field, and an eight-bit position field. (See Figure 2-13).

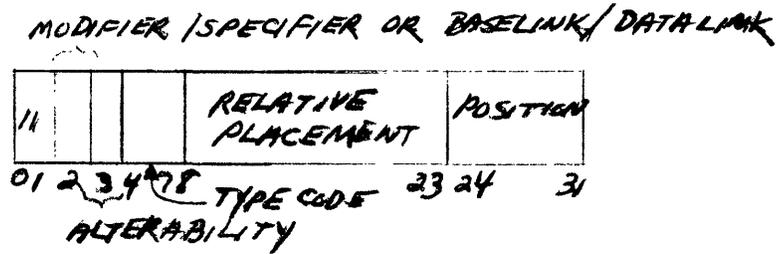


FIGURE 2-13. STRING MINISTRUTOR

2.3.2.4
(Cont.)

A string ministrator is equivalent to an explicit-length structor with the following attributes:

- a. Modifier/specifier indicator and alterability indicator identical to the ministrator indicators.
- b. LOCATION field equals the value of the relative displacement field of the ministrator in bytes, if the ministrator is a modifier, or the sum of the storage address of the ministrator plus the relative displacement field of the ministrator, when the ministrator is a specifier.
- c. EXTENT field set to zero (single item).
- d. TYPE field set to the value of the typecode field of the ministrator.
- e. POSITION field determined by the position field of the string ministrator, as a function of the typecode field of the ministrator. The manner in which this is accomplished is discussed below.

The POSITION field of the equivalent structor is determined as a function of the typecode of the string ministrator. Table 2-5 presents the alternatives.

TABLE 2-5
POSITION FIELD EQUIVALENTS FOR STRING MINISTRUCTORS

TYPECODE OF MINISTRUCTOR	EQUIVALENT TYPE	POSITION FIELD EQUIVALENT*
0	Bit String	A
1	Binary String	A
2	Hexadecimal Floating Point String	B
3	Unassigned	n.a.
4	Zoned Decimal String	B
5	Unsigned Zoned Decimal String	B
6	Packed Decimal String	B
7	Unsigned Packed Decimal String	B
8	Byte String	C
9	Translated Byte String	C
A	Unformatted Region	C
B	Edit Control String	n.a.
C-F	Unassigned	n.a.

*The alternatives for position field equivalents are as follows:

- A- The POSITION field contains the position field of the ministrator as its leftmost eight bits. The rightmost eight bits of the POSITION field are set to zero. This allows bit and binary strings with offsets 0-7, lengths 1-32, and 0 alignment offset string ministructors.
- B- The POSITION field consists of 4 zero bits, followed by the leftmost 4 bits of the position field of the ministrator, followed by 4 zero bits, followed by the rightmost 4 bits of the ministrator position field. This allows zoned decimal strings with byte lengths 1-15 and 32 and zoned decimal strings with byte lengths 1-16, and both with scale factors 0-15, and hexadecimal floating point strings with byte lengths 1-8 with or without significance truncation to be described by string ministructors.
- C- The POSITION field consists of 8 zero bits, followed by the 8 bits of the position field of the ministrator. This allows byte strings, translated byte strings, and unformatted regions with lengths 0-255 to be described by string ministructors.

2.4 BYTE SEQUENCE INFORMATION REPRESENTATIONS

The byte sequence information representations consist of a sequence of essentially identical units, each of which occupied a byte of storage. Information in the byte sequence is normally processed as an entity; that is, the entire sequence is considered to be a single operand value of variable length.

2.4.1 String Data Representations

The string data representations are used to encode data that is of highly variable length such that it is impossible to place this information in general registers.

2.4.1.1 Byte String

A byte string is a sequence of contiguous eight-bit bytes that are normally interpreted as logical values (see Figure 2-14). The maximum length for byte strings is 65,535 bytes. A byte string may originate at any byte boundary in storage. A byte string may have zero length.

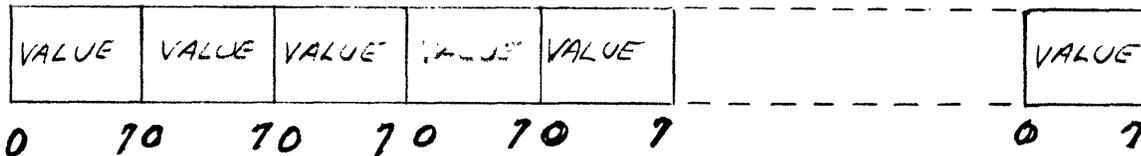


FIGURE 2-14. BYTE STRING

Byte strings are described by explicit-length structors. The interpretation of the POSITION field of byte string structors is as follows. (See Figure 2-15).

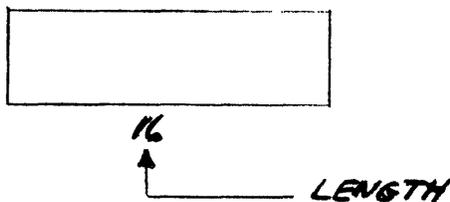


FIGURE 2-15. POSITION FIELD FOR BYTE AND TRANSLATED BYTE STRINGS

The length field specifies the number of bytes in each byte string in the array of byte strings described by the data structor. This field has a range 0 to 65,535 and is interpreted as an unsigned binary integer.

2.4.1.2 Translated Byte String

A translated byte string is a sequence of contiguous eight-bit bytes that are translated using as implicit translation table and are normally treated as logical values. (See Figure 2-16). The maximum length for translated byte strings is 65,535 bytes. A translated byte string may originate at any byte boundary in storage. The translation tables used to map from and to a translated byte string are called the load and store translation tables, respectively, and are located by means of the Task Code Map Description in the Task Status Block (see Section 2.5).

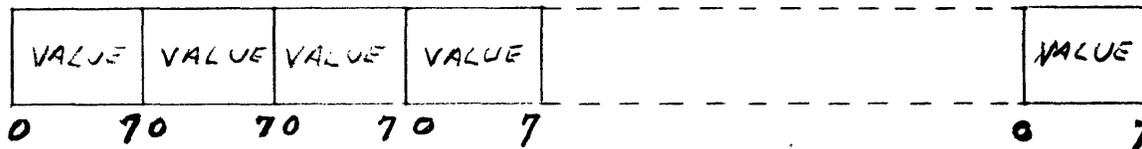
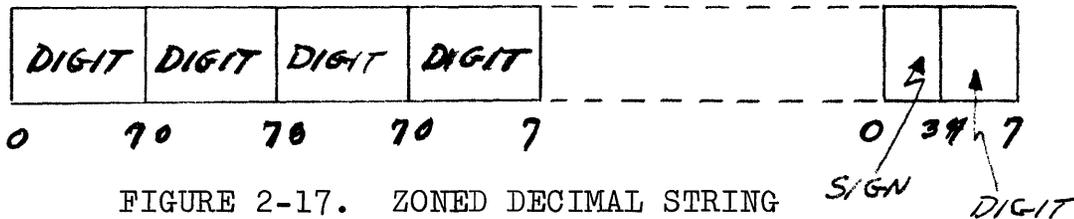


FIGURE 2-16. TRANSLATED BYTE STRING

Translated byte strings are described by explicit-length structors. The interpretation of the POSITION field of translated byte string structors is as follows. (See Figure 2-15). The length field specifies the number of bytes in each translated byte string in the array of translated byte strings described by the data structor. This field has a range 0 to 65,535 and is interpreted as an unsigned integer.

2.4.1.3 Zoned Decimal String

A zoned decimal string is a sequence of contiguous eight-bit bytes, each containing an encoding of a decimal digit, the rightmost byte of which contains a signed digit (see Figure 2-17). The sign is stored as a zone field in the rightmost byte. The zone bits in the remaining digit positions are not interpreted by instructions manipulating zoned decimal strings, but are preserved by execution of these instructions. The numeric bits in each digit position are interpreted as follows: 0000 to 1001 correspond to decimal digits 0 to 9 and 1010 to 1111 are interpreted as illegal and generate a trap when encountered. The interpretation of the sign field is specified in Appendix A. Each zoned decimal string has an associated scale factor, which is used to specify the position of an implied decimal point for the string. The maximum length for zoned decimal strings is 32 digits, and the maximum scale factor is 128 digit positions to the left, and 127 digit positions to the right of the leftmost digit of the zoned decimal string. A scale factor of 0 places the implied decimal point to the left of the leftmost digit of the string. See Appendix A for sign encoding details. A zoned decimal string may originate at any byte boundary in storage.



Zoned decimal strings are described by explicit-length structors. The interpretation of the POSITION field of zoned decimal string structors is as follows. (See Figure 2-18).

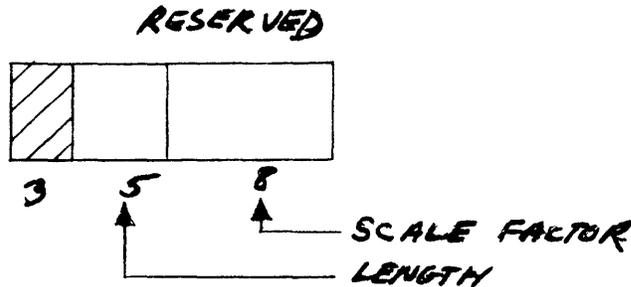


FIGURE 2-18. POSITION FIELD FOR ZONED AND UNSIGNED ZONED DECIMAL STRING STRUCTORS.

The length field specifies the number of bytes in each zoned decimal string in the array of zoned decimal strings described by the structor. This field has a range 1 to 31 and 32, which is encoded as 00001 to 11111 and 00000. The scale factor field specifies the position of an implied decimal point for the zoned decimal string. This field has a range -128 to +127 and is encoded as a twos complement binary integer.

2.4.1.4 Unsigned Zoned Decimal String

An unsigned zoned decimal string consists of a sequence of contiguous eight-bit bytes, each containing an encoding of a decimal digit. (See Figure 2-19).

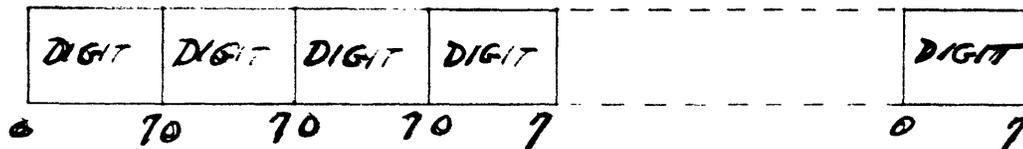


FIGURE 2-19. UNSIGNED ZONED DECIMAL STRING

The interpretation of unsigned zoned decimal strings is identical to the interpretation of zoned decimal strings, except that the zone of the rightmost byte of the string is not interpreted as a sign, but is preserved. A positive value is always implied for unsigned zoned decimal strings.

2.4.1.5 Packed Decimal String

A packed decimal string is a sequence of contiguous eight-bit bytes, each containing an encoding of a pair of decimal digits, except the rightmost byte, which contains an encoding of a decimal digit and of a sign. (See Figure 2-20). The sign code is the rightmost 4-bit field of the rightmost byte of the packed decimal string. The encoding of the sign code is specified in Appendix A. Each digit field in the string is interpreted as follows: 0000 to 1001 correspond to decimal digits 0 to 9, 1010 to 1111 are interpreted as illegal digits and generate traps when encountered. Each packed decimal string has an associated scale factor, which is used to specify the position of an implied decimal point for the string. The maximum precision for packed decimal strings is 31 decimal digits, and the maximum scale factor is 128 digit positions to the left or 127 digit positions to the leftmost digit of the packed decimal string. A packed decimal string may originate at any byte boundary in storage.

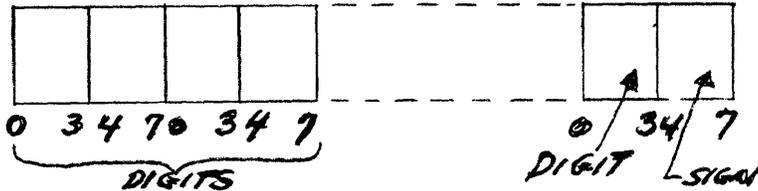


FIGURE 2-20. PACKED DECIMAL STRING.

Packed decimal strings are described by explicit-length structures. The interpretation of the POSITION field of packed decimal strings is as follows. (See Figure 2-21).

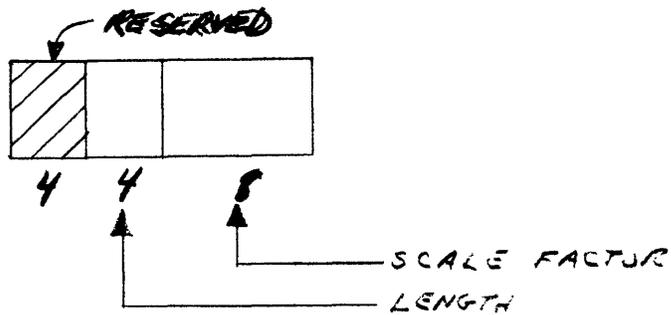


FIGURE 2-21. POSITION FIELD FOR PACKED AND UNSIGNED PACKED DECIMAL STRING STRUCTORS

The length field specifies the number of bytes in each packed decimal string in the array of packed decimal strings described by the structor. This field has a range 1 to 15 and 16, which is encoded as 0001 to 1111 and 0000. The scale factor field specifies the digit position of an implied decimal point for the packed decimal string. This field has a range -128 to +127 and is encoded as a two's complementary binary integer.

2.4.1.6 Unsigned Packed Decimal String

An unsigned decimal string consists of a sequence of contiguous eight-bit bytes, each containing an encoding of a pair of decimal digits. (See Figure 2-22).



FIGURE 2-22. UNSIGNED PACKED DECIMAL STRING

The interpretation of unsigned packed decimal strings is identical to the interpretation of packed decimal strings, except that the rightmost 4 bit field of the rightmost byte in the string is interpreted as a decimal digit. A positive sign is always implied for unsigned packed decimal strings.

2.4.2 Edit Control String

An edit control string consists of a sequence of contiguous eight-bit bytes that are interpreted as either edit control function codes or as immediate operand bytes (see Figure 2-23). The maximum length of an edit control string is 65,536 bytes. An edit control string may originate at any byte boundary in storage.

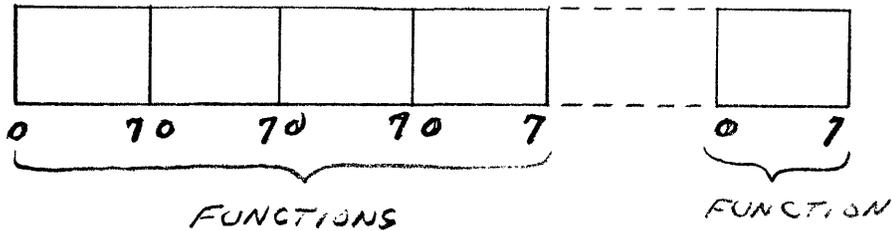


FIGURE 2-23. EDIT CONTROL STRING

The edit control string structor (explicit-length) has a format identical to that for zoned or packed decimal strings. The interpretation of the position and extent fields of this structor differ, however, from the interpretation appropriate to decimal string structors. In particular, the following interpretation applied to edit control string structors:

- a. The location field is used to identify the byte origin of the edit control string in storage.
- b. The length and scale factor fields may be used to imply the precision and scale appropriate to the destination string, if the source string is a decimal string.
- c. The extent field specifies the number of bytes in the edit control string and is used to terminate execution of the Edit instruction unless otherwise terminated.

An edit control string structor is valid as an operand only for the EDIT instruction. (See Section V).

2.4.3 Unformatted Region

An unformatted region consists of a sequence of eight-bit bytes for which no interpretation is implied. (See Figure 2-24). An unformatted region structor cannot be the effective operand of an instruction (see Section V), but can be used to represent an array of areas of storage allocated for a specific purpose. In this latter role, it is used in the Unformatted Region Qualification operation performed during instruction extraction. (See Subsection 3.5).

The maximum length for each unformatted region in an array of unformatted regions is 65,535 bytes. An unformatted region may originate at any byte boundary in storage, and may have zero length.



FIGURE 2-24. UNFORMATTED REGION

Unformatted regions are described by explicit-length structors. The interpretation of the POSITION field of unformatted region structors is as follows. (See Figure 2-25).

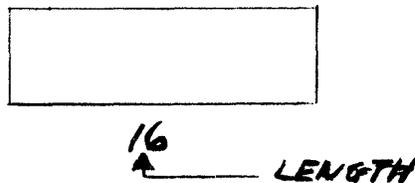


FIGURE 2-25. POSITION FIELD FOR UNFORMATTED REGION STRUCTORS

The length field specifies the number of bytes in each unformatted region in the array of unformatted regions described by the structor. This field has a range 0 to 65,535 and is interpreted as an unsigned binary integer.

2.5 CONTROL INFORMATION REPRESENTATION

The operation of the task multiplexing and control facilities is dependent on the presence in main storage of information describing the current control structure of the system. This section is devoted to a description of this required information. The instructions which operate on the control information are described in subsection 5.6 and the operation of the task multiplexing and control facility is described in Section VI. The overall organization of the control information is shown in Figure 2-26.

2.5.1 System Control Structures

The format of the structors used for system control and I/O purposes is shown in Figure 2-27.

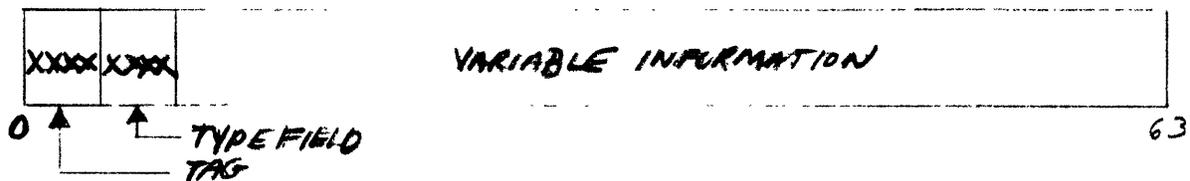


FIGURE 2-27. SYSTEM CONTROL STRUCTOR

The types applicable to system control are listed in Table 2-6.

TABLE 2-6
CONTROL STRUCTOR TYPES

TYPE	PURPOSE
0	Procedure Index
1	Relative Procedure Index
2	TSB Identifier
3	STOP Protected TSB Identifier
4	I/O Status Word
5	External Status Word
6	Type I Trap Effector
7	Type II Trap Effector
8	Reserved
9	Reserved
A	Reserved
B	Device Specifier
C	Alternate Array Specifier
D	Control Command Specifier
E	Reserved
F	I/O Command Structor

The formats and purposes of these structors are discussed in the following subsections.

2.5.1.1 Procedure Index

A Procedure Index has the format shown in Figure 2-28. Procedure indices are used to describe the state of execution of a procedure. Each task has a Current Procedure Index as part of its TSB. The Current Procedure Index describes the state of the procedure which the task is executing. There may be many procedure indices associated with a single procedure.

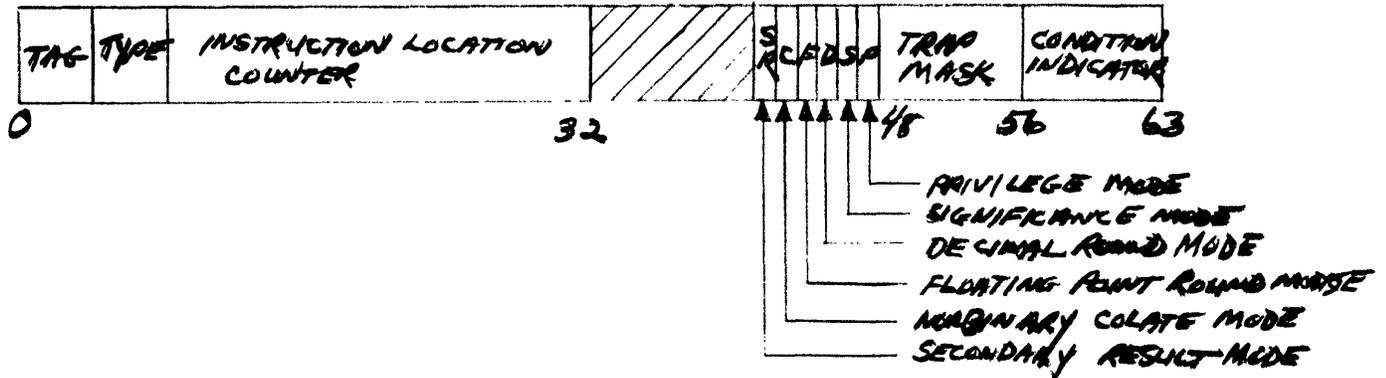


FIGURE 2-28. PROCEDURE INDEX FORMAT

2.5.1.2 Relative Procedure Index

A Relative Procedure Index is identical to a Procedure Index except that the Instruction Location Counter field contains a displacement relative to the location of the Relative Procedure Index in main storage. A Relative Procedure Index is converted into a Procedure Index whenever it is loaded into a register. (See Subsection Autofetch Conversion).

2.5.1.3 TSB Identifiers

TSB identifiers are used as operands of the task control instructions which operate on the state of tasks. They are also used as elements of the I/O and External Start Arrays. (See subsections 7.6 and 7.7).

Two types of TSB identifiers are used. A TSB identifier is normally a legitimate operand for any task control instruction. A STOP Protected TSB identifier is a legitimate operand only for the START instruction. If an attempt is made to use a STOP Protected TSB identifier as the operand of a STOP, CONDITIONAL STOP, ISTOP, or SUSPEND instruction a illegal operand trap occurs.

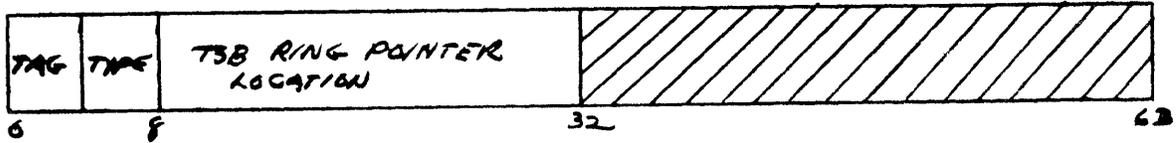


FIGURE 2-29. TSB IDENTIFIER

2.5.1.4 Status Words

There are two types of status words: I/O and External. These tagged doublewords are used to transfer information from an I/O device or External source to a task. (See subsections 7.6 and 7.7).

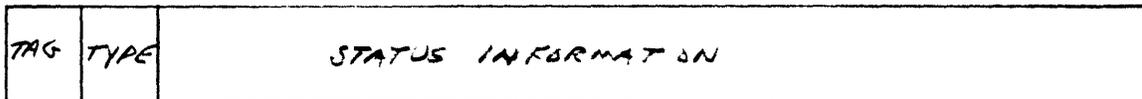


FIGURE 2-30. STATUS WORDS

2.5.1.5 Trap Effectors

Whenever an instruction references a trap effector either as an operand or during autofetch or autostore a trap occurs. The effects of these traps are described in subsection 8-1. The Trap Effector format is shown in Figure 2-31.

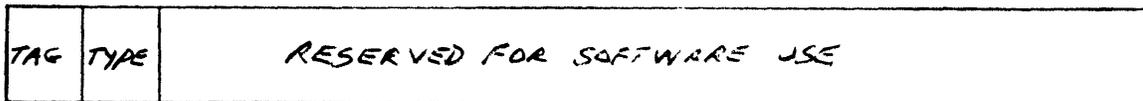


FIGURE 2-31. TRAP EFFECTORS

2.5.2 System Base

The System Base is an area in memory which contains information, or descriptions of information, concerning the state of the system as a whole. It acts as a fixed base through which the remainder of the system control

information may be accessed. It is always located in words 0-31 of main storage. The format of the System Base is shown in Figure 2-32.

0	Priority Array Structor	} T/A
2	I/O Start Array Structor	
4	External Start Array Structor	
6	Processor Status Array Structor	
8	Reserved	
10	I/O Status Array Structor	
12	Device Specification Array Structor	
14	Traffic Register Array Structor	
16	Simultaneity Table Structor	
18	<i>P</i> <i>Q</i>	
20	Table Array Structor	} Reserved
22	MAIL BOXES.	
30		

SEGMENT (handwritten on left side of table)

see (handwritten at bottom left)

FIGURE 2-32. SYSTEM BASE ORGANIZATION

The allocation of space in the System Base is as follows:

- Words 0-1: The Priority Array Structor: This explicit-length bit-string structor describes the Priority Array. (See subsection 2.5.4).
- Words 2-3: The I/O Start Array Structor: This tagged double-word array structor describes the I/O Start Array.
- Words 4-5: The External Start Array Structor: This tagged doubleword array structor describes the External Start Array.
- Words 6-7: The Processor Status Array Structor: This explicit length byte string array structor describes the Processor Status Array.
- Words 8-9: Reserved.

- Words 10-11: The I/O Status Array Structor: This explicit length byte string array structor describes the I/O Status Array.
- Words 12-17: Three array structors used to identify tables required by the I/O.
- Word 18: This word contains two lock bytes: P is the priority structure lock, and Q is the Queue lock.
- Words 20-21: The Table Array Structor: This byte string array structor describes an array of 252 byte strings, each of length 256, used for translation tables.
- Words 22-25: These words are reserved.

2.5.3

Task Status Block

The existance and current status of a task is specified by a Task Status Block (TSB). TSB's are stored in main storage and must be located on a double word boundary. They are up to 32 double words long. The exact length is determined by the operating system. Certain portions of the TSB must be accessible to the hardware. For this reason the low order portion of the TSB has a fixed format as shown in Figure 2-33.

The allocation of the TSB is as follows:

(In the following descriptions all unused bits are required to be zero).

a. Doublewords 0-15:

Sixteen general purpose registers which may be used to hold tagged data or structors.

b. Doubleword 16:

A Ring Pointer which is used to link all Tasks at a given priority level into a circular chain. (See Figures 2-33 and 2-34.)

Bits 0 and 1 specify the current state of the task. They are encoded as follows:

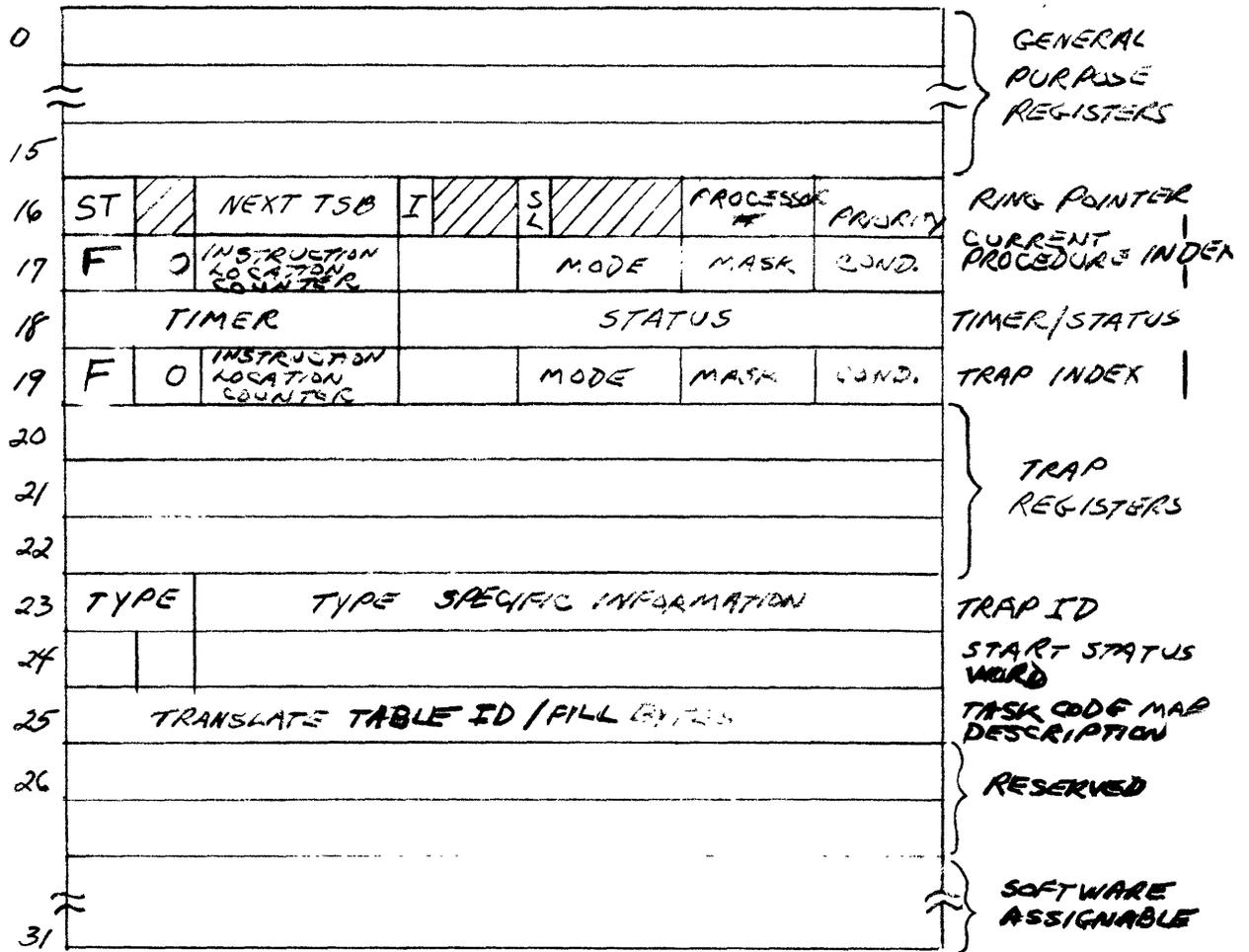


FIGURE 2-33. TSB FORMAT

- 00 Running: A running task is currently utilizing a processor.
- 01 Ready: A ready task is one which, while not currently using a processor, is prepared to do so.

10 Blocked: A blocked task is not prepared to run and is waiting for some event external to itself.

11 Available: An available task is one which is not prepared to run and which must be modified by the operating system before it may run.

Bits 8-31 contain the address of the next TSB in the chain. Bit 32 is a Start Flag which is set whenever an External of I/O Start is directed to the task. Bits 40-47 are a lock for the Start Status Word. Bits 48-55 specify the processor executing the task if it is in the running state. Bits 56-63 specify the priority of the task.

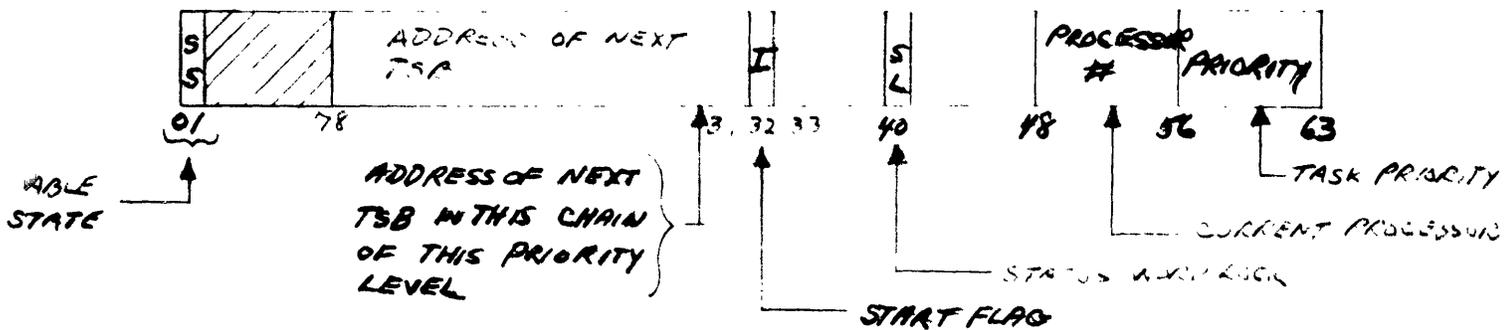


FIGURE 2-34. RING POINTER

c. Doubleword 17:

The current procedure index. This represents the state of the procedure which the task is currently executing.

d. Doubleword 18:

This word contains a 32 bit task timer and a 32 bit status field the task timer is discussed in subsection 9.3. The status field is allocated as shown in Figure 2-35.

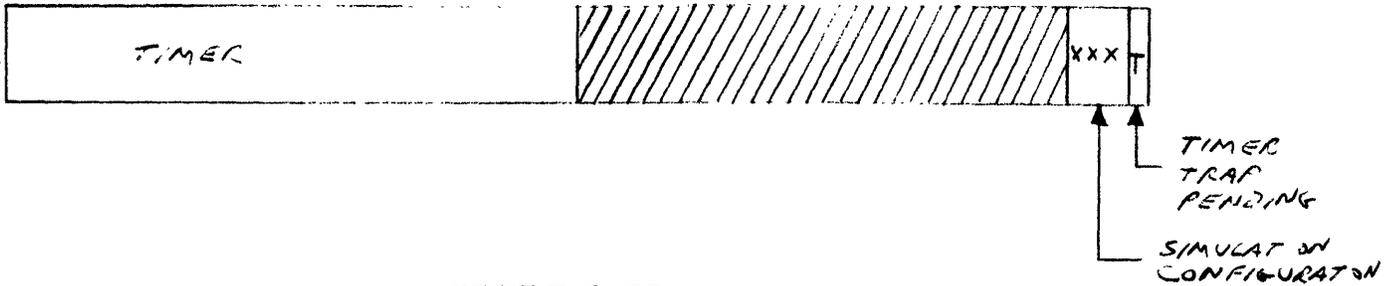


FIGURE 2-35.

The emulation configuration field (bits 27-29) are used to indicate whether or not the task requires a non-native instruction set for its execution.

e. Doubleword 19:

The Trap Index is a procedure index with the format shown in Figure 2-28. Its contents exchanged with the contents of the current procedure index when a trap occurs.

f. Doublewords 20, 21, 22:

Three general purpose register images used for trap handling.

g. Doubleword 23:

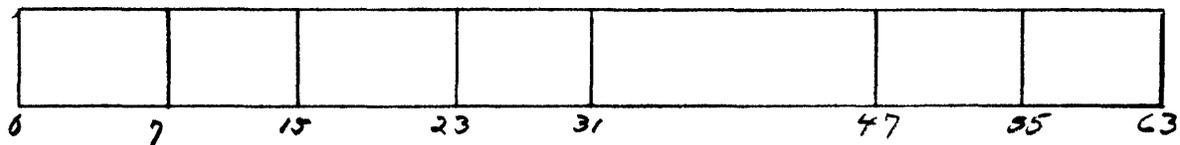
A trap ID field where information about a trap may be stored.

h. Doubleword 24:

This doubleword contains either a status field used to store an External or I/O Status Word, or a FIFO structor used to point to an area where status words may be stored.

i. Doubleword 25:

This double word contains three one-byte indices used to select the Load Translate, Store Translate, and Non-Binary Collate Tables. The Tables are selected by using the index byte to index the Table Array Structor in the system base. This doubleword also contains all numeric and alpha numeric fill bytes.



If any of these indices is set to all ones or exceed the extent of the Table Array Structor the corresponding table is not present.

FIGURE 3-36.

j. Doublewords 26-27:

These words are reserved.

k. Doublewords 28-31:

These words are assignable by the operating system.

2.5.4

Task Priority Array

The Task Priority Array (TPA) is a doubleword aligned array of 32 bit binary strings identified by the Priority Array Structor in the System Base. There is one entry in the Task Priority Array for each priority level in the system. (The maximum number of levels is 255). Each of the entries in the array points to a ring structure of all the TSBs in the system at the corresponding priority level.

The format of a TPA entry is shown in Figure 2-37. Bits 8-31 are the location field and specify the address of the ring pointer of a TSB in the ring structure.

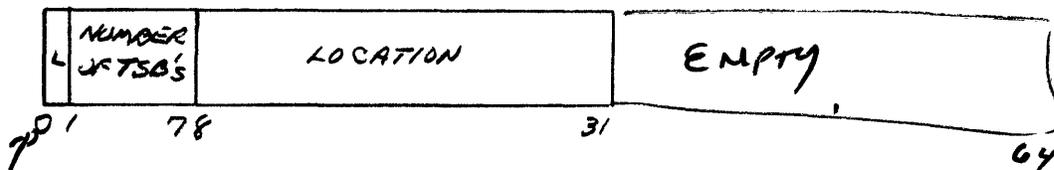


FIGURE 2-37. TASK PRIORITY ARRAY ENTRY FORMAT

Bit 0 is a Level Blocked Indicator which is set whenever the system discovers that none of the tasks in the corresponding level are in the ready state. The Level Blocked Indicator is reset whenever a task in the corresponding level makes a transition to the ready state.

Bits 1 through 7 specify the number of tasks in the ring structure at the corresponding level.

2.5.5 I/O Start Array

The I/O Start Array is an array of tagged doublewords identified by the I/O Start Array structor in the System Base. The number of entries in the I/O Start Array is a function of the I/O complement of the system and the software. Each entry may be one of the three types shown in Figure 2-38.

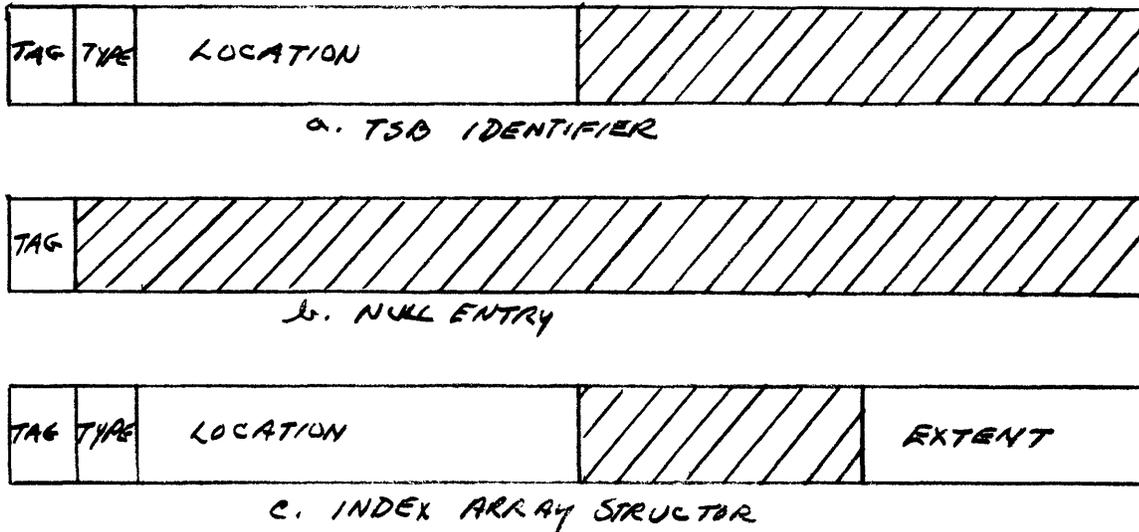


FIGURE 2-38. I/O AND EXTERNAL START ARRAY ENTRIES

The first allowable type of entry in the ISA is a TSB identifier ((b) in Figure 2-38). The location field in this entry addresses the ring pointer location in a TSB.

The second type of entry is a doubleword with a 0 tag field, ((b) in Figure 2-38). This entry is used as a null element.

The final type of entry is an implicit length tagged doubleword structor ((c) in Figure 2-38). This structor describes an Index Array which in turn may contain any one of these three types of entries. (See subsection I/O Starts).

2.5.6 External Start Array

The External Start Array is an array of tagged doublewords identified by the External Start Array Structor in the System Base. The number of entries in the array is a function of the size of the system and the operating system requirements. Each entry may be one of the three types described in subsection 2.5.5. The formats of these entries are identical with those described for the I/O Start Array in subsection 2.5.5.

2.5.7 Processor Status Array

The Processor Status Array (PSA) is a doubleword aligned array of byte strings identified by the Processor Status Array Structor in the System Base. There is a 16 byte string in the array for each processor attached to the system. The organization of these strings is shown in Figure 2-39.

The first four bytes of a PSA entry has a one-byte command field and a three-byte location field. Both of these fields are used to convey information from the system to the associated processor.

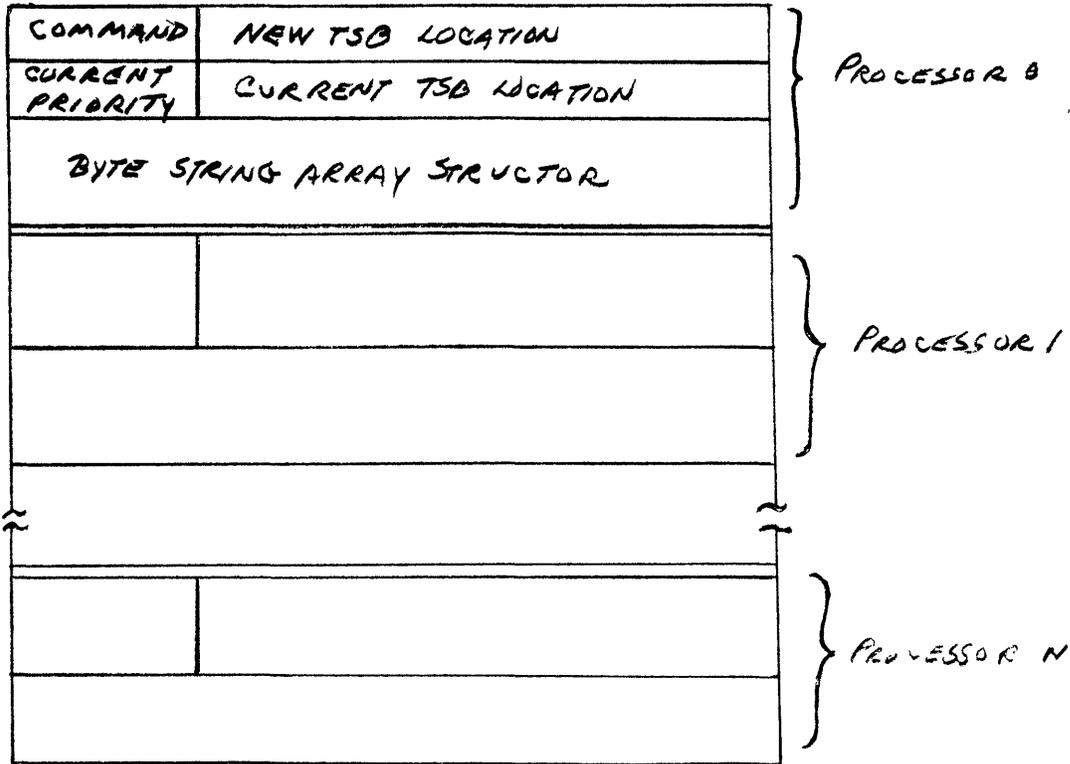


FIGURE 2-39. PROCESSOR STATUS ARRAY

The second four bytes of a PSA entry is divided into two fields: the first byte specifies the priority of the task which is being executed by the processor. The other three bytes contain the address of the Ring Pointer of the task being executed by the processor. If the processor is in the wait state then the first byte is set to all ones and the other three bytes are unspecified.

The last eight bytes of a PSA entry is a byte string array structor used to point to an area of storage reserved for the use of the associated processor. The size of this area depends on the implementation of the processor.

NOTE

The first entry in the PSA (processor zero) is reserved for the DCS. Its priority field will always be set to zero.

2.5.8 I/O Status Array

The I/O Status Array is a doubleword aligned array of byte strings identified by the I/O Status Array Structor in the System Base. Each entry in the array consists of a 16 byte string. The first eight bytes are used as a communications buffer between the processors and the I/O. The other eight bytes form a string reserved for the use of the I/O.

The number of entries in the array depends on the implementation of the I/O subsystem.

2.6 I/O INFORMATION REPRESENTATION

The execution of Input/Output instructions will require the use of information stored in main memory. Only the description and modification of this information will be specified in this section. A detailed explanation of how this information is to be used, the execution of I/O instructions and the I/O facilities, is presented in Section X.

2.6.1 I/O Structors

Tag F structors are used for system control and I/O purposes. The general format is shown in Figure 2-27. The types applicable to Input/Output structors are listed in Table 2-7.

TABLE 2-7
I/O STRUCTOR TYPES

TYPE	PURPOSE
B	Device Specifier
C	Alternate Array Specifier
D	Control Command Specifier
E	Reserved
F	I/O Command Structor

2.6.1.1 Device Identifier Structor

This structor is used to specify the logical device to be used in the peripheral operation. It is formatted as shown in Figure 2-40.

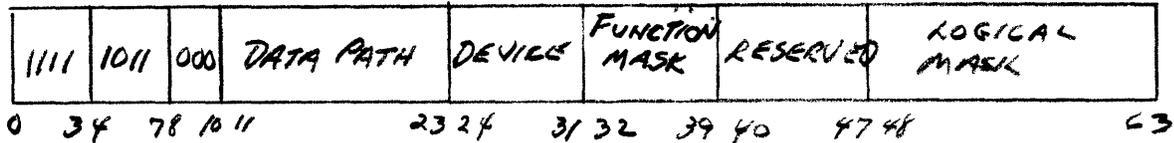


FIGURE 2-40. DEVICE IDENTIFIER STRUCTOR

The eight-bit device and 13-bit data path fields define a device and routing information. The eight-bit Function Mask and 16-bit Logical Mask fields, which are used in protection are explained in Section IX.

2.6.1.2 Alternate Array Specifier Structor

This structor will point to an array of I/O Command structors. It is to be used as the A operand in certain types of Initiate Device Operation Order. Its formats is shown in Figure 2.41,

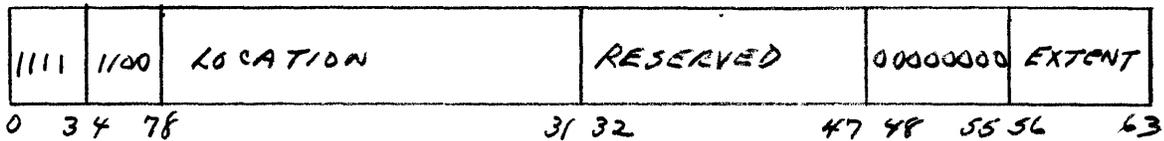


FIGURE 2-41. ALTERNATE ARRAY SPECIFIER

The 24-bit LOCATION field points to the first double word of an I/O Command Array, whose extent is specified in the EXTENT field.

This structor is used in the IDO order to initiate I/O operations on a busy device. Its application is further explained in Section X.

2.6.1.3 I/O Control Command Specifier

The I/O command defines a control operation. It is to be used as the A operand in the IDO instruction which specifies the execution of a single control command. Its format is shown in Figure 2-42.

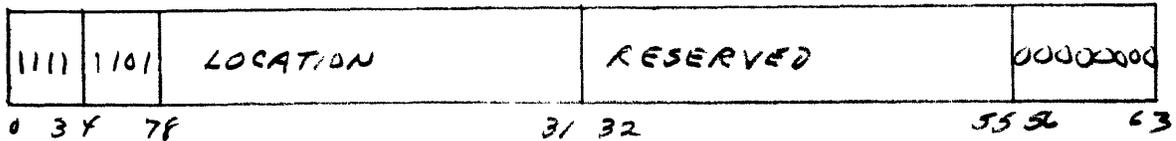


FIGURE 2-42. I/O CONTROL COMMAND SPECIFIER

It is similar to the implicit-length structor with the exception that the extent field is all zeros (it points to a single command).

2.6.1.4 I/O Command Structor

The elementary I/O operations in a chain of peripheral commands is specified by a member of an array of I/O Command Structors. These structors are formatted as shown in Figure 2-43.

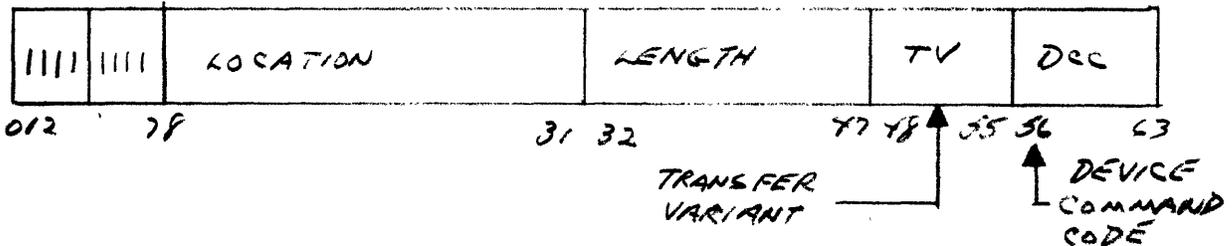


FIGURE 2-43. I/O COMMAND STRUCTOR

The 24-bit LOCATION field will specify a byte in main storage, where the data transfer will start. The 16-bit LENGTH field specifies the number of bytes to be transferred (up to 65,535 bytes). The Transfer Variant and Device Command Code specifies the function or operation to be performed, as well as information concerning the monitoring and sequencing of commands. The contents of those fields will be specified in Section X.

2.6.2 Device Specification Table

This table, whose initial address is in the System Base and contains an entry per physical device attached to the system, is formatted as shown in Figure 2-44.

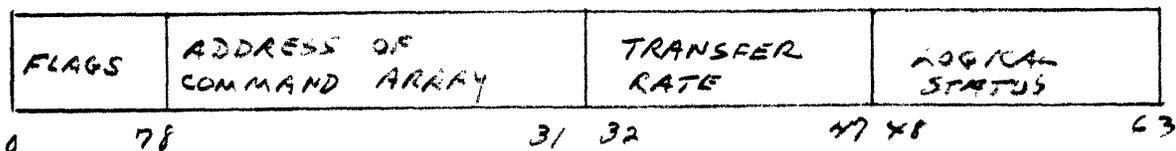


FIGURE 2-44. DEVICE SPECIFICATION TABLE FORMAT

It will contain the pertinent device status information, which is needed to effect the necessary protection and the allocation of the peripheral device, during extraction and execution of an IDO instruction. The eight-bit Flags field will be used by the central processor in determining the feasibility of execution of the order being extracted. A 24-bit field, the Address of the Command Array, points to the initial location of the Command Array currently active in the device. The device's current transfer rate is included in a 16-bit field. The 16-bit logical status will specify which logical device is currently assigned to the physical device. It is to be used with the logical mask of the device identifier structor as described in Section X.

2.6.3 Traffic Registers

This table, with an entry per system resource in which an overrun can occur, and an additional entry for the whole system is formatted as shown in Figure 2-45.

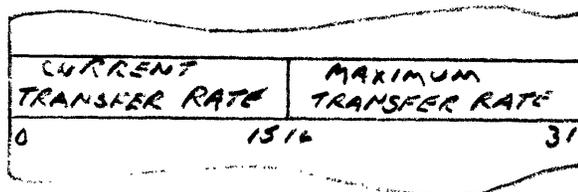


FIGURE 2-45. TRAFFIC REGISTER FORMAT

The transfer rate is measured in units, each unit being 64 transfers/second (i.e., the maximum representable number is 4,194,304 transfers/sec.). During transfer rate allocation (in the extraction of an IDO order), the traffic registers will be tested in order to anticipate a potential overrun situation.

2.6.4 Simultaneity Table

This table is used during the extraction of the IDO order to check for busy status of a resource with a fixed level of simultaneity. The table will be composed of an entry for each one of those resources. The contents of the entry will be the number of I/O operations currently being simultaneously executed.

2.6.5 Input/Output Status Word

The Input/Output Status Word (IOSW) is used in conjunction with I/O initiated starts. It contains the reason and the parameters associated with an I/O start. It is formatted as shown in Figure 2-46.

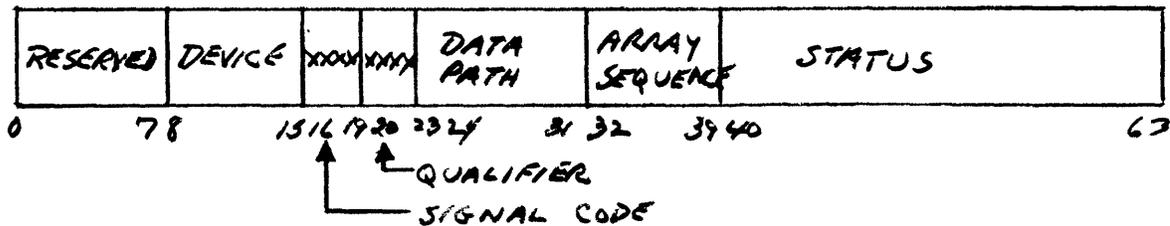


FIGURE 2-46. IOSW FORMAT

Further specification of the IOSW is provided in Section IX.

2.7 INSTRUCTION FORMATS

Instructions are two, four, or six bytes in length and must be aligned on halfword boundaries. The operation to be performed is specified by an eight-bit operation code field, the encoding of which also specifies the particular format that is applicable to the instruction. The two operands

of the instruction, which are called the A and B operands, are identified by one of several forms of operand specification. The forms of operand specification relevant to a particular instruction are characteristics of the instruction format. The formats included are discussed below. The encoding of the operation code field is specified in Appendix B.

2.7.2.1 Operand Specification Syllables

The operands of instructions are specified by instruction subfields called syllables. The available forms of syllables are considered in succeeding subsections.

2.7.2.1.1 R - Syllable

The R - Syllable consists of a 4-bit field that is interpreted as a general register address or as control information. (See Figure 2-47). When interpreted as a general register address, the field selects one of sixteen general purpose registers as the source or destination of an operand. If interpreted as control information, the field is utilized in a manner dependent upon the particular operation being performed.

2.7.2.1.2 S -Syllable

The S - Syllable consists of a 4-bit field, called the base register address, a 4-bit field, called the index register address, and a 12-bit immediate selection value field. (See Figure 2-48). The two register address fields are interpreted as general register addresses, selecting one of fifteen general purpose registers, unless the field contains a hexadecimal zero, which selects no general register. The immediate selection value field is interpreted as a twos complement binary integer in the range -2048 to +2047. The source or destination of the operand specified by the S - Syllable is determined by combination of the selected general register contents and the immediate selection value.



FIGURE 2-47. R - SYLLABLE

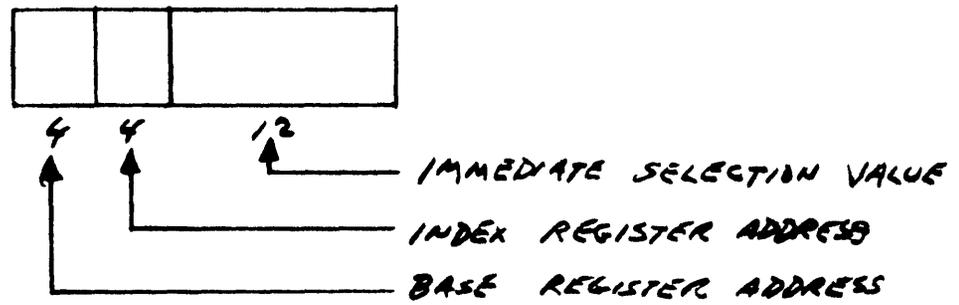


FIGURE 2-48. S - SYLLABLE

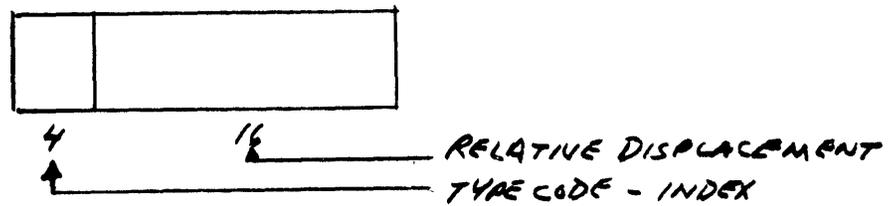


FIGURE 2-49. D - SYLLABLE

2.7.2.1.3 D - Syllable

The D - Syllable consists of a 4-bit field, called the typecode-index field, and a 16-bit field, called the relative-displacement field. (See Figure 2-49). The typecode-index field is interpreted as either: 1) a general register address, which selects one of fifteen general purpose registers, unless the field contains a hexadecimal zero, which selects no general register, or 2) a typecode, which is used to ascribe certain attributes to the associated operand. The relative-displacement field is interpreted as a twos complement binary integer.

2.7.2.1.4 V - Syllable

The V - Syllable consists of a 24-bit field that is interpreted as control information. The specific interpretation of this information depends on the particular operation being performed.

2.7.2.2 Instruction Encoding

Conceptually, an instruction consists of a sequence of operation and operand specification syllables. Instruction encoding, however, differs slightly from a linear concatenation of syllables.

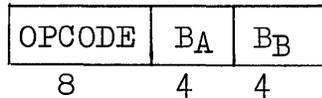
The first byte of an instruction is the operation syllable, which contains an encoding of the operation specified, and of the instruction format, as specified in Appendix B. The second byte of an instruction always consists of a pair of 4-bit fields that normally contain an encoding of a general register address or control information. For instruction formats containing S-, D-, or V-Syllables, the remaining one or two halfwords are formatted as either a concatenated 4-bit index register address and 12-bit immediate selection value or a 16-bit binary value.

The available instruction formats, presented in both conceptual and actual forms, are shown in Figure 2-50.

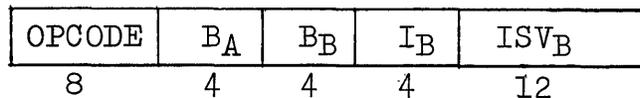
Instructions are required to be halfword boundary aligned.

Format/Bit Length/Conceptual Syllable Sequence

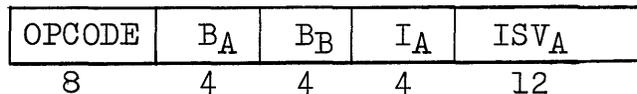
RR/16 bits/Op, R, R



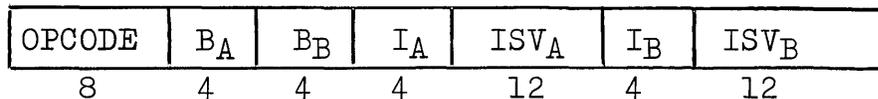
RS/32 bits/Op, R, S



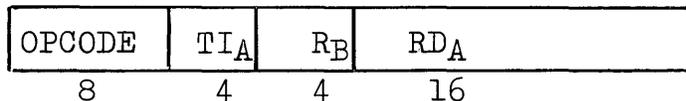
SR/32 bits/ Op, S, R



SS/48 bits/Op, S, S



RD/32 bits/Op, D, R



CV/32 bits/Op, V

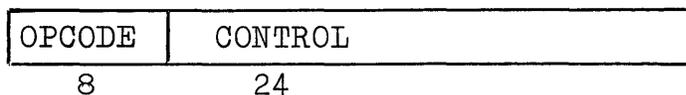


FIGURE 2-50. INSTRUCTION FORMATS

(In Figure 2-50, B_X, I_X, ISV_X, TI_X, and RD_X are abbreviations for base register, index register, immediate selection value, typecode-index, and relative displacement for the X-operand.)

--END OF SECTION--

SECTION III
SELECTION PRIMITIVES3.1 GENERAL

The selection primitives are a set of four operations that can be utilized in the process of selecting operands for instructions. These primitives are performed with one or two tagged quantities provided as inputs, one of which is always a data structor that serves as an operand description.

The four primitives are: Autofetch/Autostore Conversion, Autofetch/Autostore Evaluation, Array Indexing, and Unformatted Region Qualification. These primitives are specified in the following subsections and are utilized by operations discussed in succeeding sections.

3.2 AUTOFETCH/AUTOSTORE CONVERSION

Autofetch/Autostore Conversion is a collection of operations that are performed in order to transform operands from a compact or tagged doubleword form into a tagged form and from a tagged form into a compact or tagged doubleword form.

Autofetch Conversion requires a data structor as an input and produces a tagged quantity as a result. Autostore Conversion requires a data structor and a tagged quantity as inputs. In either case, the type of Autofetch/Autostore Conversion operation selected depends on the TYPE field of the input data structor.

In the following discussion, the data structor used by Autofetch/Autostore conversion is referred to as the operand description, while the quantity fetched from or placed in storage is called the operand.

3.2.1 Autofetch/Autostore Conversion for Bit Strings

The bit string operand is described by an explicit-length specifier structor, the TYPE field of which indicates bit string. The LOCATION field of this structor contains the storage address of the byte within which the bit string

3.2.1
(Cont.)

originates.. The bit offset subfield of the POSITION field of the structor specifies the bit position of the first bit of the bit string within the initial byte. The length subfield of the POSITION field of the structor indicates the number of bits in the bit string. The alignment offset subfield of the POSITION field of the structor indicates the desired offset of the bit string in the tagged logical word created or used.

Autofetch conversion for bit strings involves the creation of a tagged logical word. The value of the tagged logical word consists of S zero bits, followed by the L bits of the bit string, followed by 32-L-S zero bits, where S is the alignment offset field value and L the length field value from the bit string structor.

The bit string originates at bit offset F within the byte addressed by the LOCATION field of the bit string structor and occupies the succeeding L bits of storage. Autofetch Conversion for bit strings is shown in Figure 3-1.

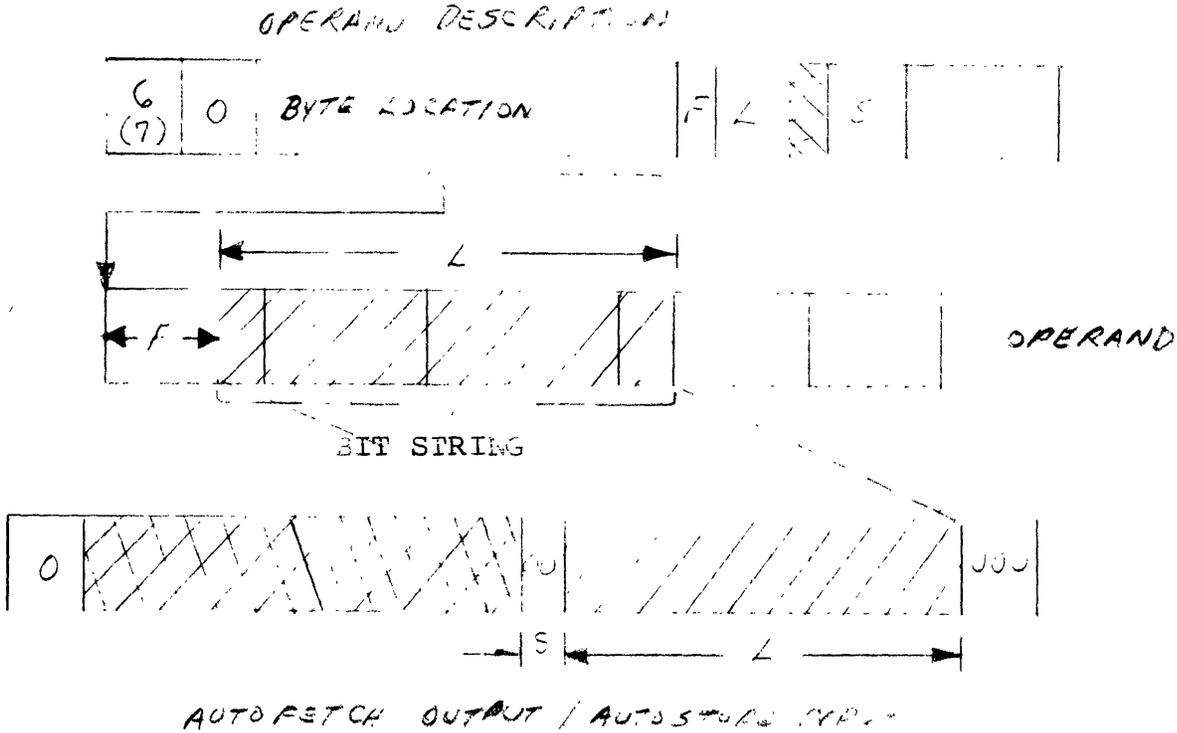


FIGURE 3-1. BIT STRING AUTOFETCH/AUTOSTORE CONVERSION

Autostore Conversion for bit strings is accomplished by substitution of bits $32 + S$ to $32 + L + S$ of a tagged logical word Autostore input into the bytes in storage containing the bit string. The bit string originates at bit offset F within the byte addressed by the LOCATION field of the structor and occupies the next L bits of storage. If $L + S$ is greater than 32, right truncation occurs. An attempt to perform Autostore Conversion with other than a tagged logical word input causes an operand selection exception trap (0104) to be generated or masked. If the bit string is nonalterable (TAG of operand description is hex 7), an attempt to perform Autostore Conversion causes an operand selection exception trap (0104) to be generated or masked. Autostore Conversion for bit strings is shown in Figure 3-1.

Autofetch/Autostore Conversion time for bit strings depends on the bit offset, length, and alignment offset of the bit string. Byte alignment, byte multiple length, and no alignment offset produces the minimum Conversion time.

3.2.2 Autofetch/Autostore Conversion for Binary Strings

The binary string operand is described by an explicit-length specifier structor, the TYPE field of which indicates binary string. The LOCATION field of this structor contains the storage address of the byte within which the binary string originates. The bit offset subfield of the POSITION field of the structor specifies the bit position of the first bit of the binary string within the initial byte. The length subfield of the POSITION field of the structor indicates the number of bits in the binary string. The alignment offset subfield of the POSITION field of the structor indicates the desired offset of the binary string in the tagged binary integer created or used.

Autofetch Conversion for binary strings involves the creation of a tagged binary integer. The value of the tagged binary integer consists of S bits identical to the leftmost bit of

3.2.2
(Cont.)

binary string (sing bit), followed by the L bits of the binary string, followed by 32-L-S zero bits, where L is the length field value and S the alignment offset field value of the binary string structor. If L + S is greater than 32, the rightmost bits are truncated.

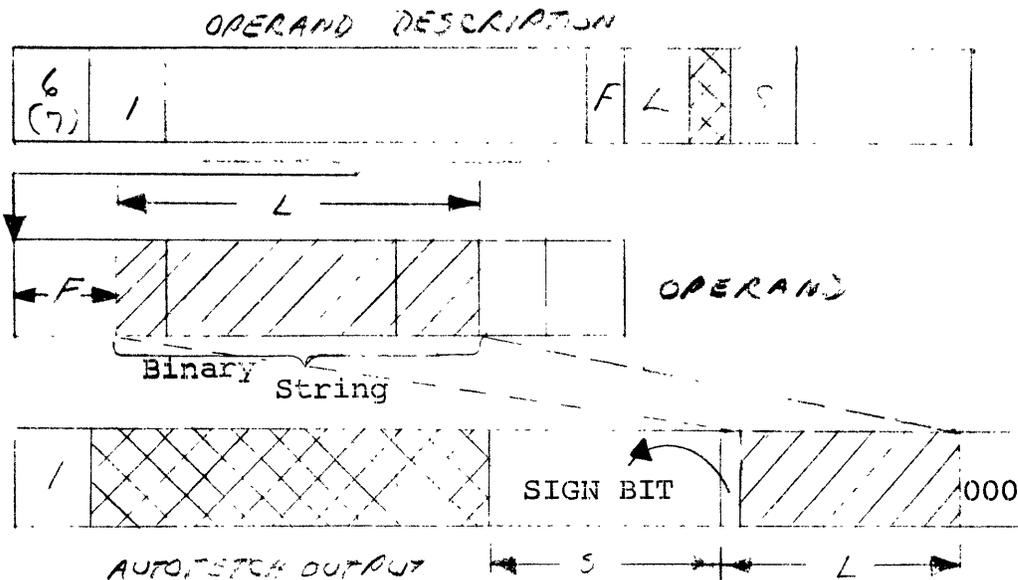


FIGURE 3-2. BINARY STRING AUTOFETCH CONVERSION

The binary string originates at the bit offset F within the byte addressed by the LOCATION field of the binary string structor and occupies the succeeding L bits of storage. Autofetch Conversion for binary strings is shown in Figure 3-2.

Autostore Conversion for binary strings is accomplished by substitution of bits 32+L to 32+L+S of a tagged binary integer Autostore input into the bytes in storage containing the binary string. The binary string originates at bit offset F within the byte addressed by the LOCATION field of the structor and occupies the next L bits of storage. If the leftmost S bits of the tagged binary integer are not equal to the bit 32 + S of the tagged binary integer, then an attempt to perform Autostore Conversion with other than a tagged binary integer input causes an operand selection exception trap (0104) to be generated or masked. Finally,

3.2.2 (Cont.) if the binary string is nonalterable (TAG of operand description is hex 7), and attempt to perform Autostore Conversion causes in an operand selection exception trap (0104) to be generated or masked. Autostore Conversion for binary strings is shown in Figure 3-3.

Autofetch/Autostore conversion time for binary strings depends on the bit offset, length, and alignment offset of the binary string. Byte alignment, byte multiple length, and no alignment offset produces the minimum Conversion time.

This description of binary string Autofetch/Autostore Conversion is consistent with the interpretation of binary strings as two's complement binary integers of specified precision and scale.

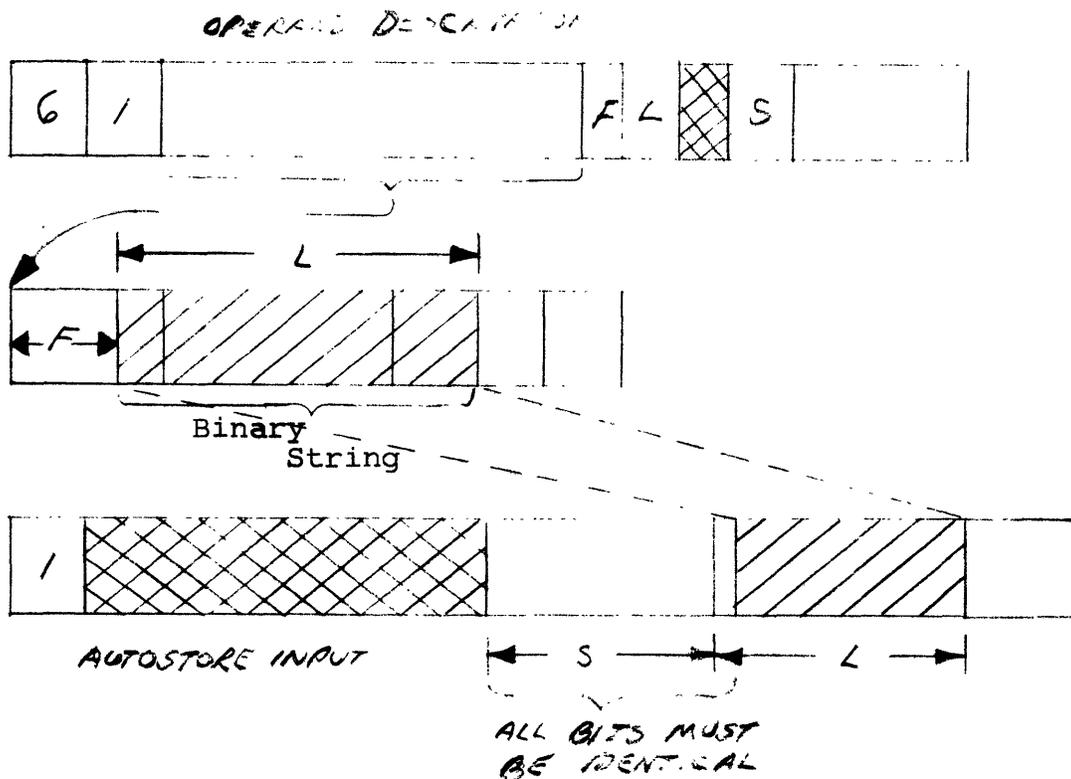


FIGURE 3-3. BINARY STRING AUTOSTORE CONVERSION

3.2.3 Autofetch/Autostore Conversion for Floating Point Strings

The floating point string operand is described by an explicit-length specifier structor, the TYPE field of which indicates floating point string. The LOCATION field of this structor contains the storage address of the byte at which the floating point string originates, The length subfield of the POSITION field of the structor indicates the number of bytes in the floating point string. The significance truncation indicator in the POSITION field is used to control significance truncation when Autofetch/Autostore Conversion for floating point strings is performed in significance mode..

Autofetch Conversion for floating point strings involves the creation of a tagged hexadecimal floating point number. This is achieved by concatenating the TAG for a tagged hexadecimal floating point number (hex 2) with the floating point string and extending the string right with zeros until the precision of the mantissa is 13 digits. If the length of the floating point string is 8 bytes, the rightmost hexadecimal digit is truncated.

When the significance mode indicator is set and the significance truncation indicator in the operand description data structor is reset, the mantissa formed above is shifted right by a number of hexadecimal digit positions equal to fifteen minus two times the length of the floating point string in bytes. If the length of the floating point string is eight bytes, no mantissa shift is performed. If the mantissa shift is performed, the number of digit positions shifted is added to the exponent. If the exponent overflows, an operand selection exception trap (0105) is shown in Figure 3-4.

Autostore Conversion for hexadecimal floating point strings is the inverse of the above process. If the significance mode indicator is set, and the significance truncation

3.2.3
(Cont.)

indicator in the data structor is reset, the mantissa is shifted left until its leftmost digit is nonzero or until $14 - (2*[L-1])$ digit positions have been shifted. The number of positions shifted is subtracted from the exponent. If the exponent under flows, an operand selection exception trap (0104) is generated or masked.

Independent of the setting of the significance mode indicator, the sign/exponent and leftmost L-1 pairs of digits of the mantissa are substituted into the byte addressed by the LOCATION field of the operand description data structor and the next L-1 bytes to its right where L is the value of the length subfield of the POSITION field of the structor. If L is 8 then the rightmost digit of the rightmost byte is zero. An attempt to perform Autostore Conversion with other than a tagged hexadecimal floating point input causes an operand selection exception trap (0104) to be generated or masked.

When the floating point round mode indicator is set, a hexadecimal eight is added to digit position $(2*[L-1]) + 1$ of the mantissa and hexadecimal zero to another digit positions of the mantissa, where L is the floating point string byte length. This action is performed after the significance shift step of Autostore Conversion. If L is 8, this step is not performed.

If the floating point string is nonalterable (TAG of operand description if hex 7), an attempt to perform Autostore Conversion causes an operand selection exception trap to be generated or masked.

Autostore Conversion for floating point strings is shown in Figure 3-5.

Autofetch/Autostore Conversion time for floating point strings depends on boundary alignment, byte length, and the significance mode and significance truncation indicator settings, and the round mode indicator setting.

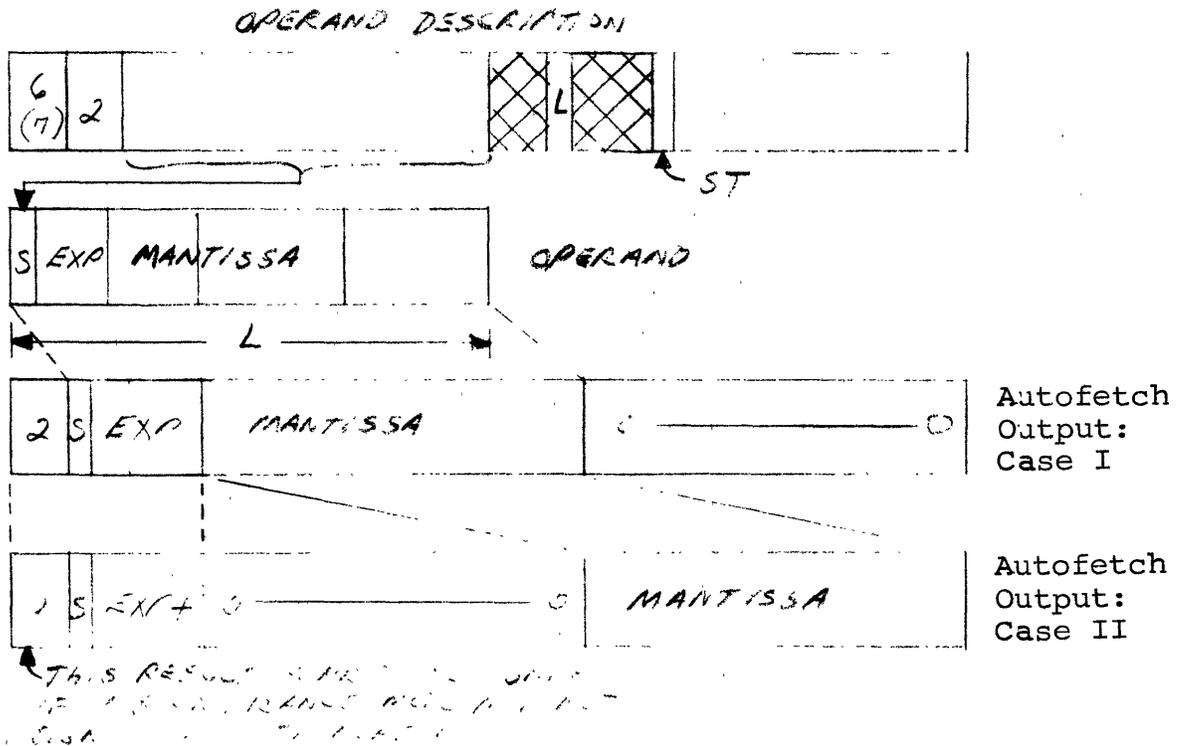


FIGURE 3-4. FLOATING POINT STRING AUTOFETCH CONVERSION

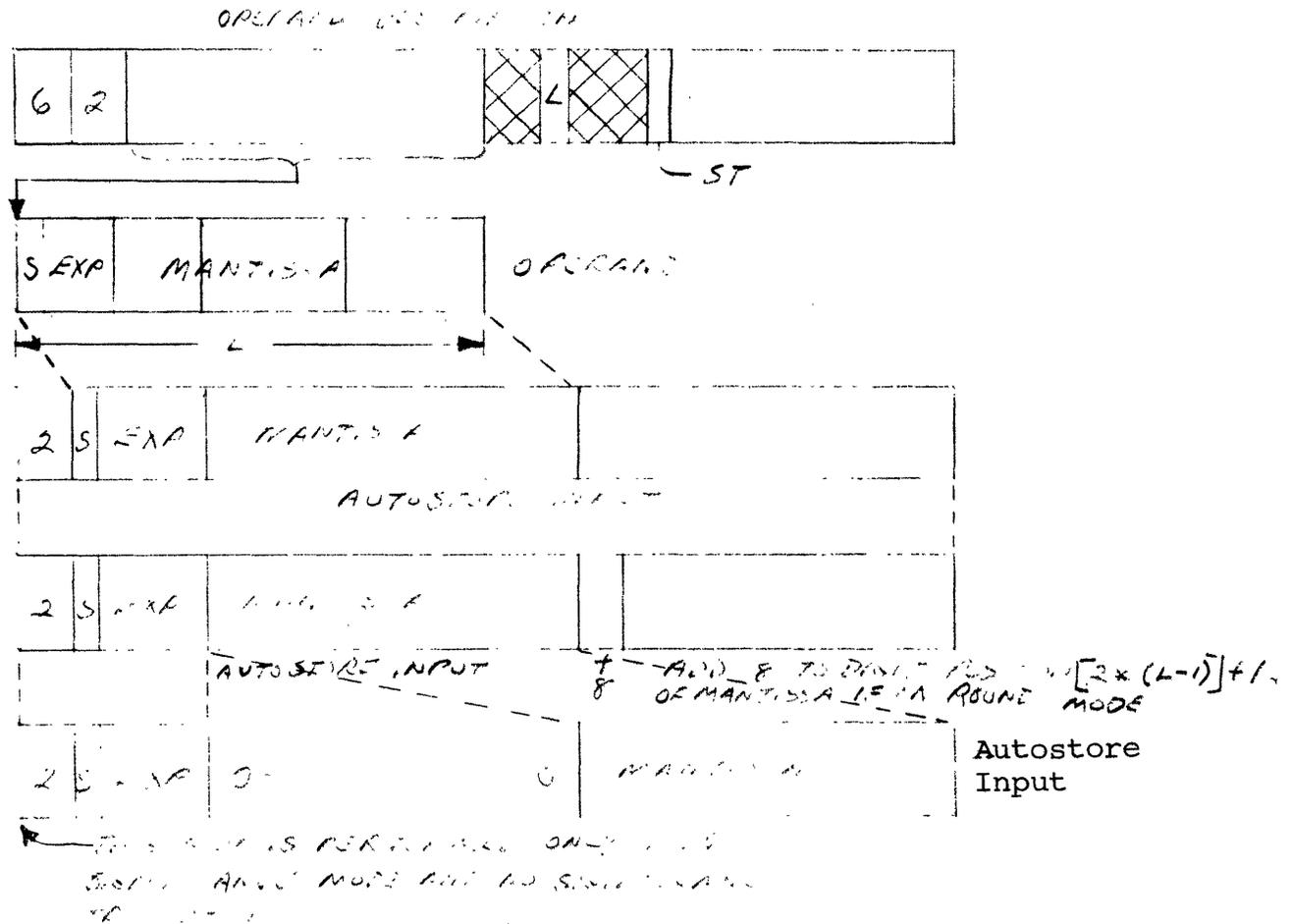


FIGURE 3-5. FLOATING POINT STRING AUTOSTORE CONVERSION

3.2.4 Autofetch/Autostore Conversion for Tagged Doublewords

The tagged doubleword operand is described by an implicit-length structor with a TYPE field specifying tagged doubleword and a LOCATION field identifying the storage address of a tagged doubleword. Any tagged quantity may be retrieved or stored into this tagged doubleword operand.

Autofetch Conversion for tagged doublewords consists of forming a tagged quantity identical to the contents of the eight consecutive bytes of storage originating at the byte addressed by the LOCATION field of the tagged doubleword structor. If the rightmost three bits of the LOCATION field are not all zero, an operand selection exception trap (0105) is generated or masked. Autofetch Conversion for tagged doublewords is shown in Figure 3-6.

Autostore Conversion for tagged doublewords consists of substituting a tagged input quantity into the eight consecutive bytes of storage originating at the byte addressed by the LOCATION field of the tagged doubleword structor. If the rightmost three bits of the LOCATION field are not all zero, an operand selection exception trap (0104) is generated or masked. When the tagged doubleword item is nonalterable (TAG of operand description is 9 or B), an attempt to perform Autostore Conversion causes an operand selection exception trap (0103) to be generated or masked. Autostore Conversion for tagged doublewords is shown in Figure 3-6.

3.2.5 Autofetch/Autostore Conversion for Ministructors

The ministructor operand is described by an implicit-length structor with a TYPE field specifying ministructor and a LOCATION field identifying the storage address of a ministructor.

Autofetch Conversion for ministructors consists of several steps, starting with retrieval of the four consecutive bytes of storage originating at the byte address specified by the

3.2.5
(Cont.)

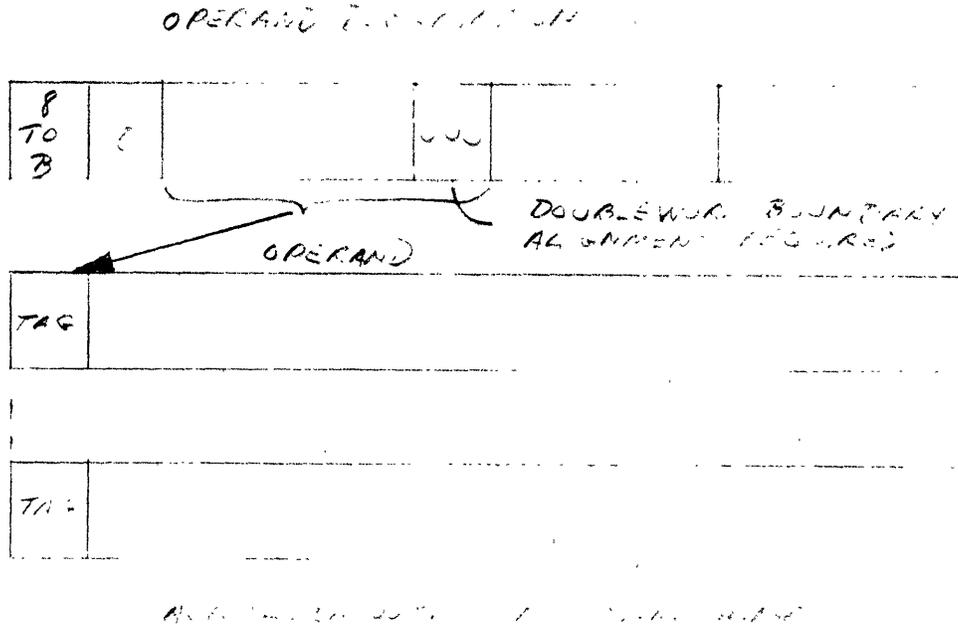


FIGURE 3-6. TAGGED DOUBLEWORD AUTOFETCH/AUTOSTORE CONVERSION

structor LOCATION field that contain the ministrator operand. If the rightmost two bits of the structor LOCATION field are not both zero, an operand selection exception trap (0105) structor identify it as either a singular, dual, array, or structor identify it as either a singular, dual, array, or string class ministrator. The next bit in the ministrator identifies the modifier/specifier and baselink/datalink forms. The next bit specifies the alterability attribute associated with the ministrator, and the next four bits constitute a typecode field. These leftmost eight bits of the ministrator control the formation of the structor that is output. The following steps describe the process for constructing the ministrator-equivalent structor:

- a. The third and fourth bits of the ministrator (modifier/specifier or baselink/datalink and alterability indicators) become the rightmost two bits of the TAG field of the structor.

3.2.5
(Cont.)

- b. When the classifier field indicates a singular or dual ministrator, the LOCATION field of the structor has the same value as the ministrator location field.
- c. When the classifier field indicates an array or string ministrator, then depending on the value of the third bit of the ministrator (modifier/specifier or baselink/datalink indicator), the LOCATION field of the structor is formed as follows:
 - i) If the third bit is reset (modifier, baselink), then the structor LOCATION field equals the value of the relative displacement field of the ministrator.
 - ii) If the third bit is set (specifier, datalink), then the structor LOCATION field equals the value of the ministrator relative displacement field plus the value of the LOCATION field of the operand description structor. The latter contains the byte address of the ministrator.
- d. When the classifier field indicates a singular, dual, or array ministrator, the typecode field of the ministrator is used to form the first two bits of the TAG field, the TYPE field, and the POSITION field of the output structor in accordance with Table 2-4.
- e. The structor EXTENT field is set to zero, one, the value of the ministrator extent field, or zero, depending on whether the classifier field indicates a singular, dual, array, or string ministrator, respectively.
- f. When the classifier field indicates a string ministrator, the leftmost two bits of the structor TAG field are set to indicate an explicit-length structor. The structor TYPE field equals the ministrator typecode field, and the typecode field value is used to control formation of the output structor POSITION field. In particular, if the typecode indicates bit, or binary string (values 0 and 1), then the position field of the ministrator becomes the leftmost eight bits of the structor POSITION field, and the rightmost eight bits of the structor POSITION field are set to zero.

3.2.5
(Cont.)

If the type code indicates floating point string, or packed or zoned signed or unsigned decimal string (values 2, 4, 5, 6, 7,), then the structor POSITION field consists of four zero bits, followed by the leftmost four bits of the ministrator position field, followed by four zero bits, followed by the rightmost four bits of the ministrator position field. Finally, if the typecode indicates byte or translated byte string or unformatted region, then the ministrator position field becomes the rightmost eight bits of the structor POSITION field, and the leftmost eight bits of the POSITION field are set to zero.

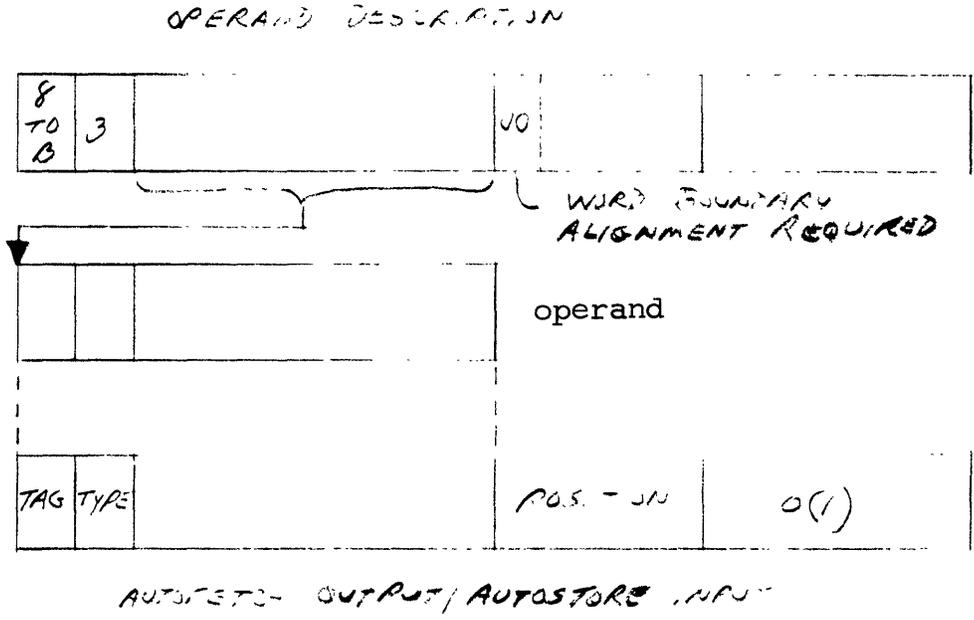
The Autofetch Conversion process for ministructors is shown in Figure 3-7.

Autostore Conversion for ministructors is also a multistep process, consisting of storage of a ministrator in the four consecutive bytes of storage originating at the byte address specified by the structor LOCATION field. If the rightmost two bits of the structor LOCATION field are not both zero, an operand selection exception trap (0104) is generated or masked. An attempt to perform Autostore Conversion into a nonalterable ministrator operand causes an operand selection exception trap (0103) to be generated or masked.

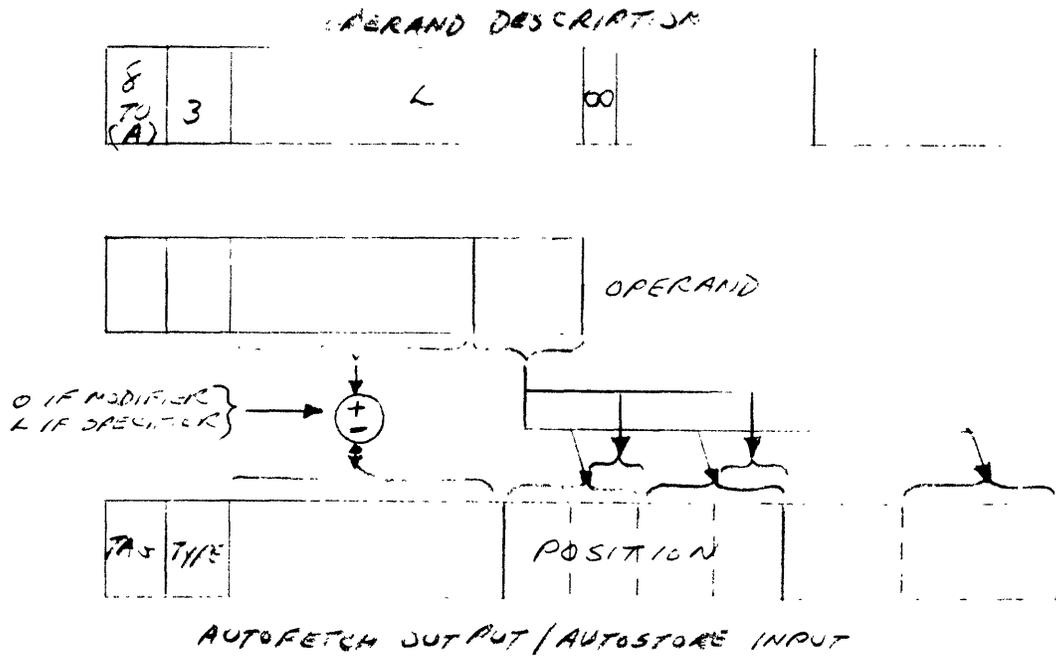
The steps performed in Autostore Conversion are controlled by various fields in the tagged quantity provided as Autostore input. These steps are specified as follows:

- a. The TAG field of the Autostore input must indicate a structor (TAG values hex 4 to B). If not, then an operand selection exception trap (0104) is generated or masked.
- b. If the TAG field indicates an implicit-length structor, then the following steps are performed.
 - i. When the EXTENT field is zero or one, the ministrator consists of a classifier value 00 (singular) or 01 (dual), baselink/datalink and alterability indicators identical to the input structor indica-

3.2.5
(Cont.)



a. SINGULAR, DUAL MINISTRUCTORS



b. ARRAY, STRING MINISTRUCTORS

FIGURE 3-7. MINISTRUCTOR AUTOFETCH/AUTOSTORE CONVERSION

3.2.5
(Cont.)

tors, a typecode field formed as specified below, and a location field value identical to the LOCATION field in the input structor.

- ii. If the EXTENT field is not zero or one, but is less than 255, the ministrator consists of a classifier value 10 (array), baselink/datalink and alterability indicators identical to the input structor indicators, a typecode field formed as specified below, and an extent field set equal to the value of the EXTENT field of the input structor. The relative displacement field of the array ministrator is computed as follows. If the TAG field indicates a baselink structor, and if the value of the structor LOCATION field is less than 65,536, then the ministrator relative displacement field is assigned this value. If the TAG field indicates a datalink structor, and if the difference between the operand description structor LOCATION field and the input structor LOCATION field is less than 65,536, then the ministrator relative displacement field is assigned the value of the difference.

The typecode field of the ministrator is set to 0, if the TYPE field of the input structor indicates tagged doubleword (TYPE value 0), or is set to 1, if the TYPE field indicates ministrator (TYPE value 3). If the constraints specified cannot be satisfied, an operand selection exception trap (0104) is generated or masked.

- c. If the TAG field indicates an explicit-length structor, the following steps are performed.
 - i. If the TYPE field indicates bit, or binary string, and if the rightmost byte of the POSITION field is zero, a string ministrator is formed with modifier/specifier and alterability indicators identical to the input structor indica-

3.2.5
(Cont.)

- tors, a typecode field identical to the TYPE field of the input structor, and a position field equal to the leftmost byte of the POSITION field of the input structor.
- ii. If the TYPE field indicates floating point string or zoned or packed, signed or unsigned decimal string, and if bits 0-3 and 8-11 of the POSITION field of the structor are all zero, a string ministrator is formed with modifier/specifier and alterability indicators identical to the input structor indicators, a typecode field identical to the TYPE field of the input structor and a position field composed of bits 4-7 followed by bits 12-15 of the structor POSITION field.
 - iii. If the TYPE field indicates byte or translated byte string or unformatted region, and if the leftmost byte of the structor position field is zero, a string ministrator is formed with modifier/specifier and alterability indicators identical to the input structor indicators, a typecode field identical to the TYPE field of the input structor, and a position field equal to the rightmost byte of the structor POSITION field.

The relative displacement field of the string ministrator is computed as follows. If the TAG field indicates a modifier structor, and if the value of the structor location field is less than 65,536, then the ministrator relative displacement field is assigned this value. If the TAG field indicates a specifier structor, and if the difference between the operand description structor LOCATION field and the input structor location field is less than 65,536, then the ministrator relative displacement field is assigned the value of the difference. If the constraints specified cannot be satisfied, an operand selection exception trap (0104) is generated or masked.

Whenever a ministrator cannot be produced by any of the above steps, an operand selection exception trap (0104) is generated or masked. Autostore Conversion for ministructors is shown in Figure 3-7.

Note that it is not always possible to perform Autofetch Conversion for a ministrator and then subsequently perform Autostore Conversion for the resulting structor.

3.3 AUTOFETCH/AUTOSTORE EVALUATION

Autofetch/Autostore Evaluation consists of a procedure for computing a structor value by means of indirection through a chain of data structors. Autofetch Evaluation must be supplied with a tagged quantity as an input and produces a tagged quantity as an output. Auto store evaluation must be supplied with two tagged input quantities. Items in the indirection chain must be data structors or ministructors, except the final item may be any tagged quantity. The maximum number of items in an indirection chain is sixteen. This limitations guarantees that the Autofetch/Autostore Evaluation process will terminate in a finite number of steps.

The steps describing Autofetch Evaluation are specified below. These steps must be performed in the order specified. The steps are defined in terms of an indirection count, which is used to record the number of indirection steps performed, and an old and current tagged value, used as working locations and initialized to zero. Autofetch Evaluation is provided with a tagged input value, and produces a tagged output value. The steps defining Autofetch Evaluation are as follows:

- a. The tagged input quantity becomes the current tagged value, and the indirection count is set to zero.
- b. The TAG field of the current tagged value is used to select an action to be performed, as specified in Table 3-1. If Autofetch Evaluation is terminated, no further

3.3
(Cont.)

steps are performed, and the current tagged value is the output of Autofetch Evaluation. If it is not terminated, step c. is performed.

- c. Increment the indirection count by one. If it is equal to sixteen, an operand selection exception trap (0106) is generated or masked. Otherwise, go to step b.

Step c. is only performed in the case where the current tagged value is an implicit-length datalink structor. This structor is used to indirectly access operands.

The steps for performing Autostore Evaluation are similar to those for Autofetch Evaluation. Autostore Evaluation may be applied in conjunction with Autofetch Evaluation or independently. If both Autofetch and Autostore Evaluation apply to an operand of an instruction, then Autostore Evaluation consists only of the action of storing a quantity in the location from which the same operand was retrieved by Autofetch Evaluation. The following two rules apply in this case:

- a. If the quantity output by Autofetch Evaluation was retrieved from a general register, then Autostore Evaluation consists of restoring the result to the same general register.
- b. If the quantity output by Autofetch Evaluation was retrieved from storage, then Autostore Evaluation consists of utilizing the data structor in the old tagged value generated by Autofetch Evaluation to perform Autostore Conversion the input quantity. The form of Autostore Conversion selected depends on the TAG and TYPE fields of the old tagged value data structor.

When only Autostore Evaluation is applied to an instruction operand, Autostore Evaluation consists of the steps presented below. The steps are defined in terms of an indirection count, which is used to record the number of indirection steps performed, and an old and current tagged value,

3.3
(Cont.)

used as working locations and initialized to zero. Autostore Evaluation is provided with a tagged input value which may serve as an operand description. The steps defining Autostore Evaluation are as follows:

- a. The tagged operand description quantity becomes the current tagged value, and the indirection count is set to zero.
- b. The TAG field of the current tagged value is used to select an action to be performed, as specified in Table 3-2.
- c. If the indirection count is zero, the current tagged value must have been retrieved from a general register, and the Autostore input is placed in this general register. Autostore Evaluation terminates.
- d. If the indirection count is nonzero, and if the current tagged value is produced by action B in Table 3-2, then the old tagged value is used as an operand description to restore the result of an instruction execution. If the TAG and TYPE fields indicate bit, binary, or floating point string, or an implicit-length type, then the result is restored using the corresponding form of Autostore Conversion. Other TAG and TYPE field combinations result in using the old tagged value to restore byte sequence results during instruction execution. Autofetch Evaluation terminates.
- e. If the indirection count is nonzero, and if the current tagged value is produced by action C in Table 3-2, then the indirection count is incremented by one. If it is equal to sixteen, an operand selection exception trap is generated or masked. Otherwise, go to step b.

Restoration of results of instruction execution takes place during instruction execution.

TABLE 3-1
 AUTOFETCH EVALUATION ACTIONS

<u>TAG</u>	<u>NAME</u>	<u>ACTION*</u>
0	Tagged Logical Word	A
1	Tagged Binary Integer	A
2	Tagged Hexadecimal Floating Point Number	A
3	Unassigned	E
4	Explicit, Modifier, Alterable Structor	A
5	Explicit, Modifier, Nonalterable Structor	A
6	Explicit, Specifier, Alterable Structor	B
7	Explicit, Specifier, Nonalterable Structor	B
8	Implicit, Baselink Structor	A
9	Implicit, Baselink Structor	A
A	Implicit, Datalink, Alterable Structor	C
B	Implicit, Datalink, Nonalterable Structor	C
C-E	Unassigned	E
F	System Control Structor	D

*The actions performed are as follows:

- A- The current tagged value is the output of Autofetch Evaluation. Autofetch Evaluation is terminated.
- B- The current tagged value is an explicit-length specifier structor. The TYPE field of this structor is examined to determine the appropriate action. If the TYPE field indicates bit, binary, or floating point string, (TYPE values 0, 1, 2), then bit, binary, or floating point string Autofetch Conversion is performed, using the current tagged value as an operand description. (See Subsection 3.2.) The Autofetch Conversion output then becomes the current tagged value. If the TYPE field indicates some other explicit-length type, then the current tagged value remains unaltered. In either case, Autofetch Evaluation is terminated.
- C- The current tagged value is an implicit-length datalink structor. The TYPE field of this structor is examined to determine the appropriate action. If the TYPE field indicates tagged doubleword, tagged doubleword - LIFO access, tagged doubleword - FIFO access, or ministrator (TYPE values 0, 1, 2, 3), then the corresponding form of Autofetch Conversion is performed, using the current tagged value as operand description. (See Subsection 3.2.) The Autofetch Conversion output then becomes the current tagged

value. The next step is then performed. Any other TYPE field value causes an operand selection exception trap (0106) to be generated or masked. The operand description for Autofetch Conversion is preserved as the old tagged value.

- D- The current tagged value is a system control structor. The TYPE field of this structor is examined to determine the appropriate action. If the TYPE field indicates relative procedure index (TYPE value 1), then the LOCATION field of the old tagged value (see C- above) is added to the LOCATION field of this relative procedure index, and its TYPE field is set to procedure index (TYPE value 0). This modified quantity then becomes the current tagged value. Other TYPE field values leave the current tagged value unaltered. In either case, Autofetch Evaluation is terminated.
- E- An operand selection exception trap (0105) is generated or masked.

TABLE 3-2
AUTOSTORE EVALUATION ACTIONS

<u>TAG</u>	<u>NAME</u>	<u>ACTION*</u>
0	Tagged Logical Word	A
1	Tagged Binary Integer	A
2	Tagged Hexadecimal Floating Point Number	A
3	Unassigned	D
4	Explicit, Modifier, Alterable Structor	A
5	Explicit, Modifier, Nonalterable Structor	A
6	Explicit, Specifier, Alterable Structor	B
7	Explicit, Specifier, Nonalterable Structor	D
8	Implicit, Baselink Structor	A
9	Implicit, Baselink Structor	A
A	Implicit, Datalink, Alterable Structor	C
B	Implicit, Datalink, Nonalterable Structor	C
C-E	Unassigned	D
F	System Control Structor	A

*The actions performed are as follows:

- A- The current tagged value is the output of Autostore Evaluation.
- B- The current tagged value is an explicit-length, specifier structor. The indirection count is incremented by one and the current tagged value becomes the old tagged value.
- C- The current tagged value is an implicit-length, datalink structor. The TYPE field of this structor is examined to determine the appropriate action. If the TYPE field indicates tagged doubleword, tagged doubleword-LIFO access, tagged doubleword FIFO access, or ministrator (TYPE values 0, 1, 2, 3), then the corresponding form of Autofetch Conversion is performed, using the current tagged value as operand description. (See Subsection 3.2) The Autofetch Conversion output then becomes the current tagged value. The next step is then performed. Any other TYPE field value causes an operand selection exception trap (0104) to be generated or masked. The operand description for Autofetch Conversion is preserved as the old tagged value.
- D- An operand selection exception trap (0104) is generated or masked.

3.4 ARRAY INDEXING

The Array Indexing operation computes a value consisting of a data structor describing the item in an array of items with a specified index value. The array of items is described by a base reference data structor, while index value consists of a tagged logical word or a tagged binary integer. The data structor resulting from the Array Indexing operation is identical to the base reference structor, with the exception that its LOCATION, EXTENT, and bit offset subfield of its POSITION Field (if appropriate) are altered. These fields are computed using the following procedures:

- a. The value of the tagged logical word or tagged binary integer index value must be not less than zero and not greater than the EXTENT field of the base reference data structor. Otherwise, an operand selection exception trap (0100) is generated or masked.
- b. The index value is multiplied by the item length specified by the base reference structor. For explicit-length structors, the item length is contained in the POSITION field of the base reference structor, as specified in Table 3-3. For implicit-length structors, the item length is determined by the TYPE field of the base reference structor. This length is 8 bytes, except for ministructors, which have a length of 4 bytes. The displacement value resulting from this multiplication specifies the number of bits or bytes the desired item is displaced from the location specified in the base reference structor.
- c. If the displacement value specifies a bit displacement, it is added to the value of the concatenated LOCATION and bit offset fields of the base reference structor, and the result becomes the value of the concatenated LOCATION and bit offset fields of the data structor resulting from the Array Indexing operation. When

TABLE 3-3
ITEM LENGTH FIELD IN EXPLICIT-LENGTH STRUCTORS

TYPE	NAME	BIT POSITIONS	
		IN DATA STRUCTOR	INTERPRETATION
0	Bit String	35 - 39	Number of bits in string*
1	Binary String	35 - 39	Number of bits in string*
2	F.P. String	37 - 39	Number of bytes in string**
3	Unassigned		N. A.
4	Zoned Decimal String	35 - 39	Number of bytes in string*
5	Unsigned Zones Decimal String	35 - 39	Number of bytes in string*
6	Packed Decimal String	35 - 39	Number of bytes in string***
7	Unsigned Packed Decimal String	35 - 39	Number of bytes in string***
8	Byte String	32 - 47	Number of bytes in string
9	Translated Byte String	32 - 47	Number of bytes in string
A	Unformatted Region	32 - 47	Number of bytes in string
B	Edit Control String	N. A.	N. A.
C-D	Unassigned	N. A.	N. A.
E-F	Software Assignable	N. A.	N. A.

*The code 00000 in the length field signifies a length of 32 bits or bytes.

**The code 000 in the length field signifies a length of 8 bytes.

***The code 0000 in the length field signifies a length of 16 bytes.

3.4
(Cont.)

the displacement value specifies a byte displacement, it is added to the value of the LOCATION field of the base reference structor, and the result becomes the value of the LOCATION field of the data structor resulting from the Array Indexing operation. In either case, the LOCATION field specifies the byte containing the leftmost bit of the item with the desired index value.

d. The EXTENT field of the result structor is set to zero.

A data structor with the above characteristics is generated as a result of the Array Indexing operation. Figure 3-8 shows the steps performed by the Array Indexing operation.

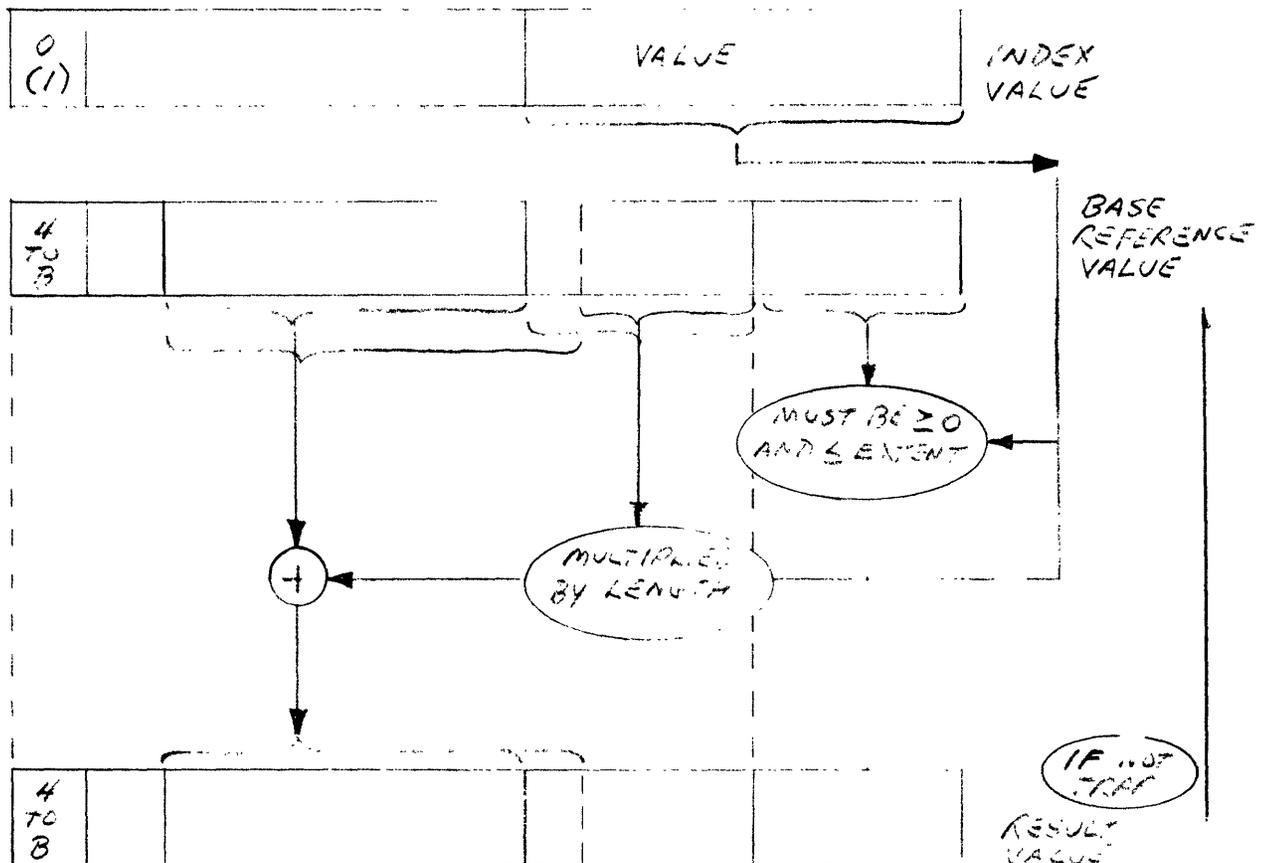


FIGURE 3-8. ARRAY INDEXING OPERATION

3.5. UNFORMATTED REGION QUALIFICATION

The Unformatted Region Qualification operation computes a value consisting of a data structor describing an item with attributes identical to a given modifier structor, called the qualitier, with the exception that the LOCATION and TAG fields of the result structor are derived by utilizing an unformatted region structor. The data structor resulting from the Unformatted Region Qualification operation is computed as follows:

- a. The TAG field of the result structor is derived from the TAG fields of the unformatted region and qualifier structors. In particular, the result structor is explicit-length, since the qualifier structor is explicit-length. The result structor is a modifier or specifier, depending on whether the unformatted region structor is a modifier or specifier, respectively. The result structor alterability indicator is set (read-only) if either of the unformatted region or qualifier structor alterability indicators is set.
- b. The TYPE field of the result structor is identical to the TYPE field of the qualifier structor.
- c. The LOCATION field value of the qualifier structor is added to the value of the LOCATION field of the unformatted region structor, and the result becomes the LOCATION field value of the result structor.
- d. The POSITION field of the result structor is identical to the POSITION field of the qualifier structor.
- e. The EXTENT field of the result structor is zero.

In the process of computing the result structor, certain checks are performed to insure that the result structor describes an item within the unformatted region.

3.5
(Cont.)

In the process of computing the result structor it is necessary to insure that the result structor describes an item within the unformatted region. This is done as follows:

- a. The maximum byte displacement of the item is calculated. For explicit length structors the item length is contained in the POSITION field of the qualifier structor, as specified in Table 3-3.

If the item length specifies a bit length, it is added to the value of the concatenated LOCATION and bit offset fields of the qualifier structor, and the result is decremented by one to produce the maximum bit displacement. The rightmost 3-bits of the maximum bit displacement are then truncated to produce the maximum byte displacement.

When the item length specifies a byte displacement, it is added to the value of the LOCATION field of the qualifier structor, and the result is decremented by one to produce the maximum byte displacement.

- b. The maximum byte displacement is then compared with the value of the POSITION (unformatted region length) field of the unformatted region structor. If it is greater than the POSITION field value, then an operand selection exception trap (0102) is generated or masked.

If the above steps do not result in a trap, the result structor computed by the previous set of steps is the result of the Unformatted Region Qualification operation. Figure 3-9 shows the steps performed by the Unformatted Region Qualification operation.

3.5
(Cont.)

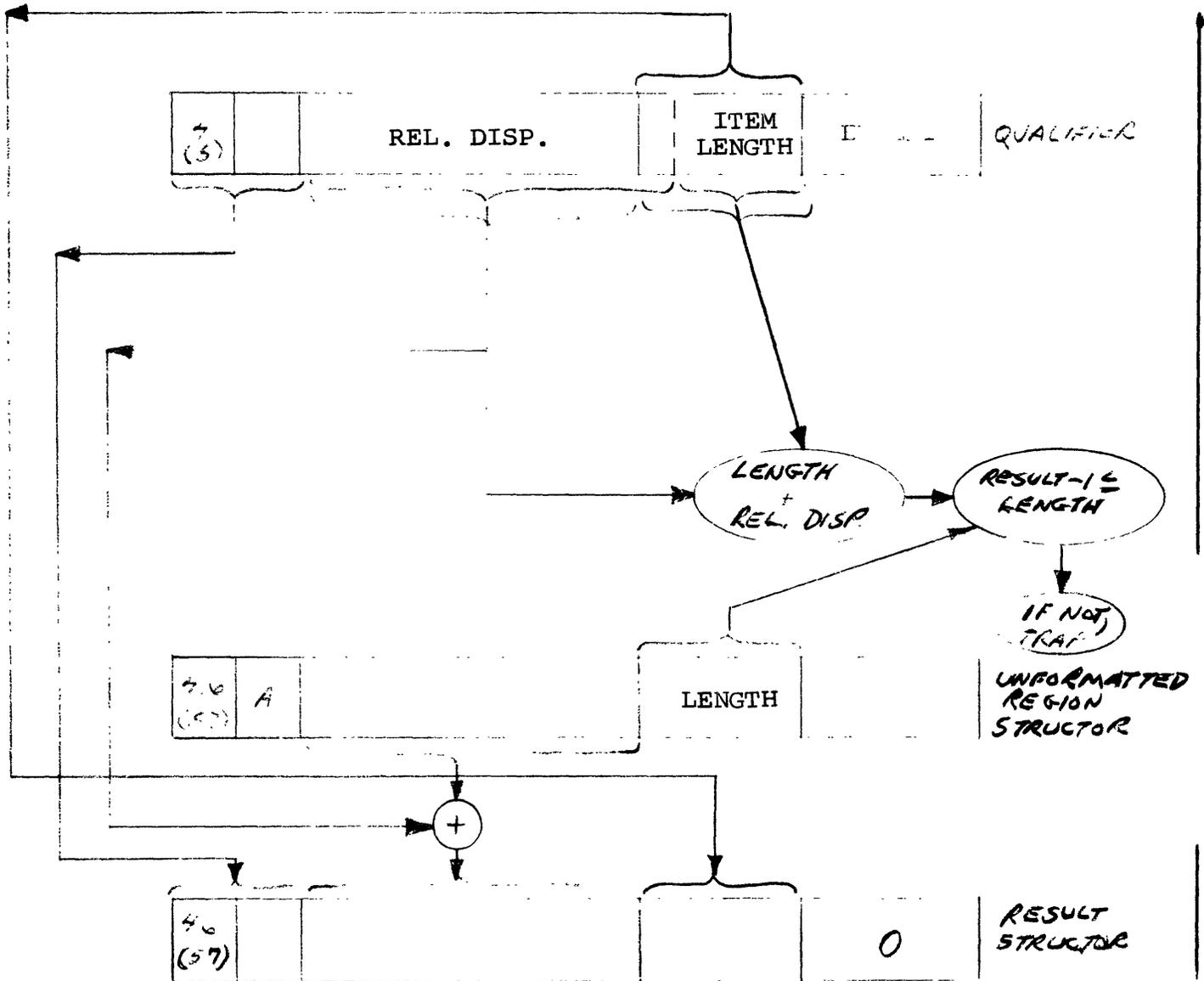


FIGURE 3-9. UNFORMATTED REGION QUALIFICATION OPERATION

--END OF SECTION--

SECTION IV
INSTRUCTION EXTRACTION4.1 GENERAL

Instruction Extraction is the process of interpreting an instruction format to determine the operation to be performed and to select the operands to be utilized.

The operation syllable of an instruction identifies the instruction format (instruction syllable sequence) for each instruction. It also contains an encoding of the operation to be performed. The operand specification syllables of an instruction are used to determine the attributes of operands to which the specified operation is to be applied.

The process of Instruction Extraction consists of the following steps. The input to Instruction Extraction is the current procedure index in the TSB. The overall process is shown in Figure 4-1.

- a. The instruction location counter in the current procedure index locates an operation syllable in storage. This operation syllable is fetched and decoded.
- b. The instruction format appropriate to the instruction is determined, and the operand syllable extraction process applicable to the A operand is performed. The operand syllable extraction processes are described in succeeding subsections.
- c. The operand syllable extraction process applicable to the B operand is performed.

The result of performing Instruction Extraction is normally an operation description and two operand descriptions, called the initial A and initial B operands. In some instructions, only one operand is required or a special

interpretation is placed on an operand specification syllable.

The instruction location counter is incremented by the instruction format byte length after an instruction is executed.

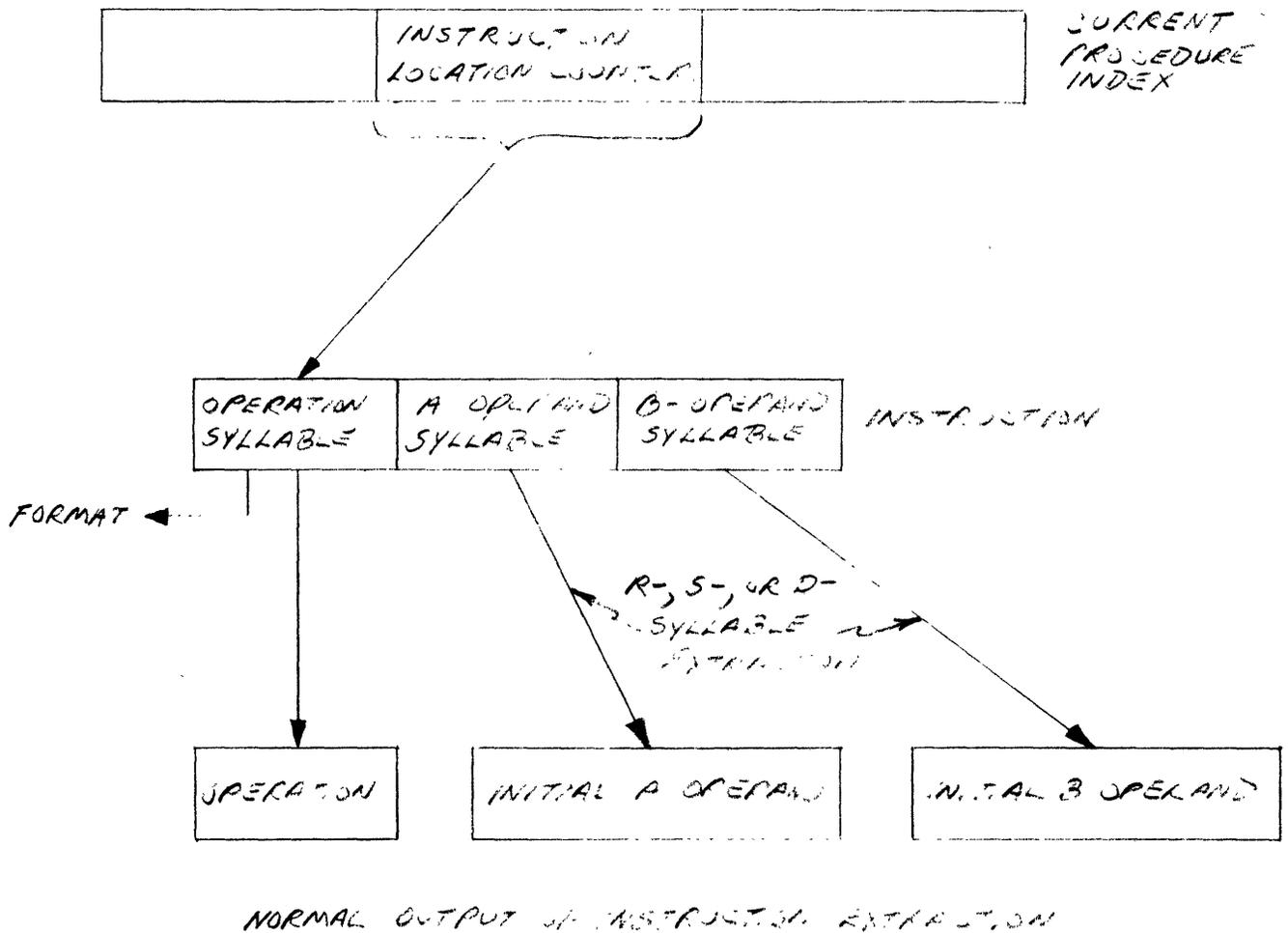


FIGURE 4-1. INSTRUCTION EXTRACTION

4.2 R-SYLLABLE EXTRACTION

The extraction sequence for R-Syllable Extraction consists of selecting the general register specified in the R-Syllable and using this register as the source or destination of an operand or as the source of a structor describing an operand in storage. If the R-Syllable is used to specify the source of an operand, the initial operand formed by R-Syllable extraction is a copy of the contents of the selected general register. If the R-Syllable is used to specify the destination of an operand, then one of the following two cases applies:

- a. When the contents of the selected general register are an explicit-length, specifier structor or an implicit-length, datalink structor and Autostore Evaluation (see section 3.3) applies to the operand described by the R-Syllable, Autostore Evaluation restores the result of instruction execution to storage.
- b. In all other cases, the result of instruction execution is restored to the selected general register.

The formation of an initial operand by R-Syllable Extraction is shown in Figure 4-2.

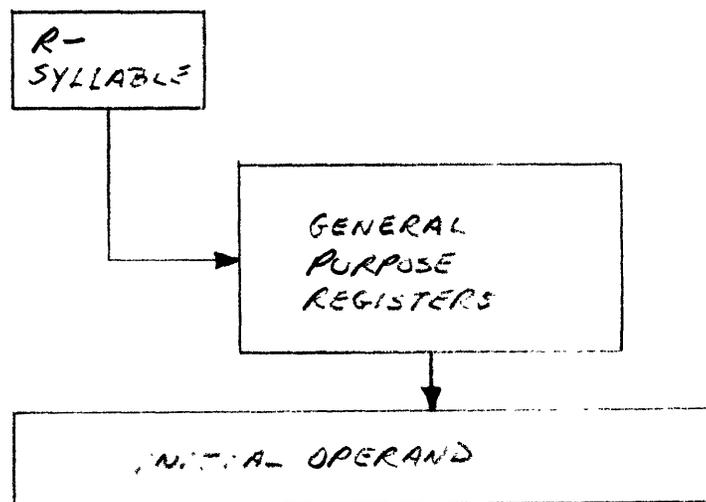


FIGURE 4-2. R-SYLLABLE EXTRACTION

4.3 S-SYLLABLE EXTRACTION

The extraction sequence for S-Syllable Extraction depends on the values of the base register address field, index register address field, and immediate selection value field of the S-Syllable. (See Figure 4-3). When either the base or index register address field is zero, the field is not used to select a general register but is used instead to perform a special type of operand selection. The flow of S-Syllable Extraction is shown in Figure 4-4.

The S-Syllable Extraction process depends primarily on the value of the base register address field. If the value of this field is zero, then the following steps are performed:

- a. If the index register address field is zero, then the immediate selection value field, interpreted as a two's complement number, is used to form a tagged binary integer of equal value, which becomes the initial operand generated by S-Syllable Extraction. This case is shown in Figure 4-5.
- b. If the index register address field is nonzero, then the contents of the addressed general register are examined. When the index register contains a tagged logical word or a tagged binary integer, its value is added to the immediate selection value, and

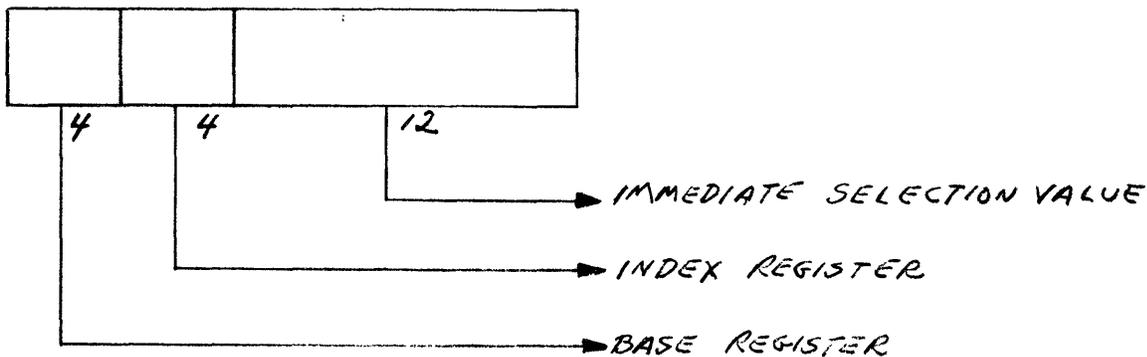


FIGURE 4-3. S-SYLLABLE FORMAT

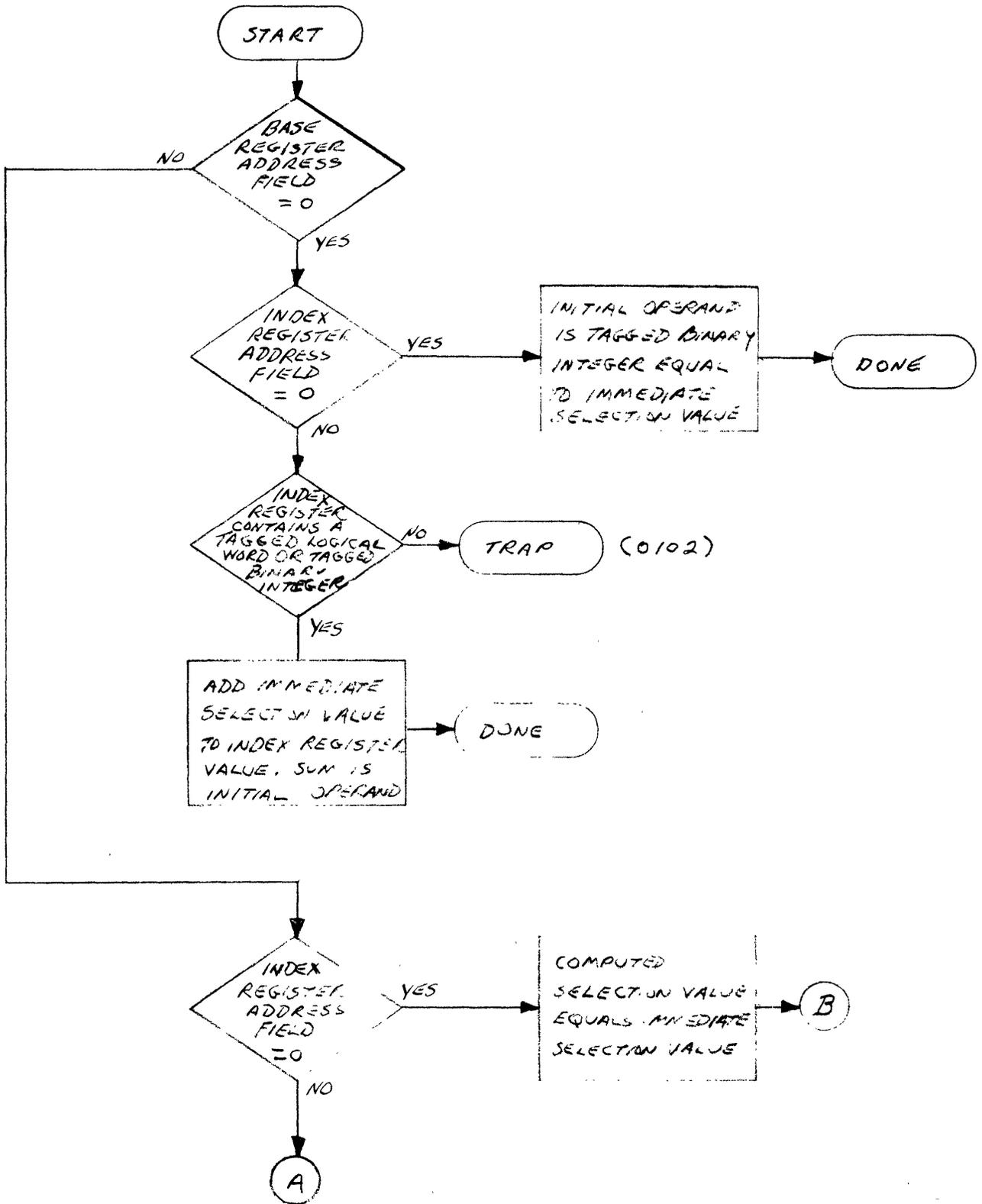


FIGURE 4-4. S-SYLLABLE EXTRACTION

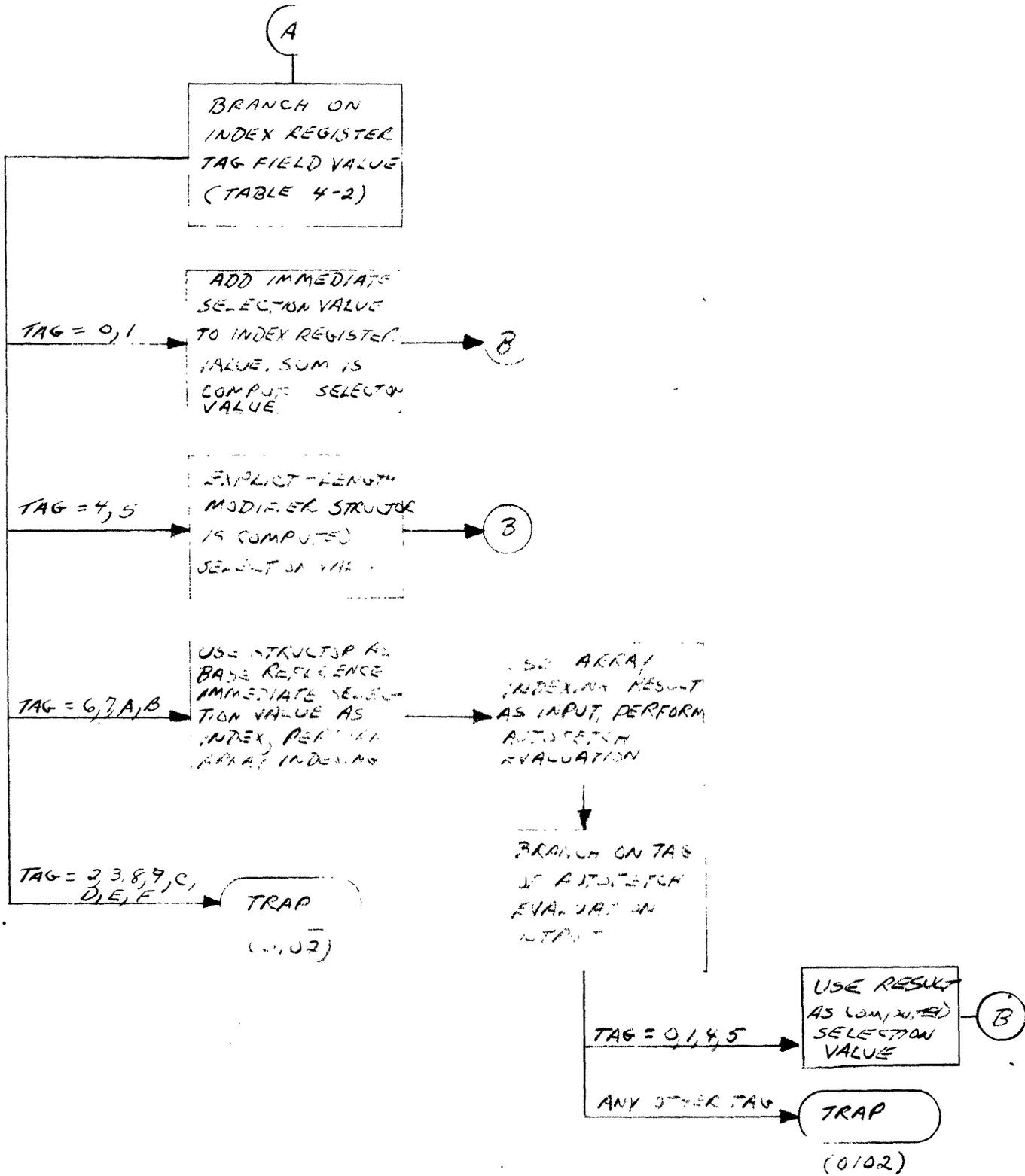


FIGURE 4-4. S-SYLLABLE EXTRACTION (Cont.)

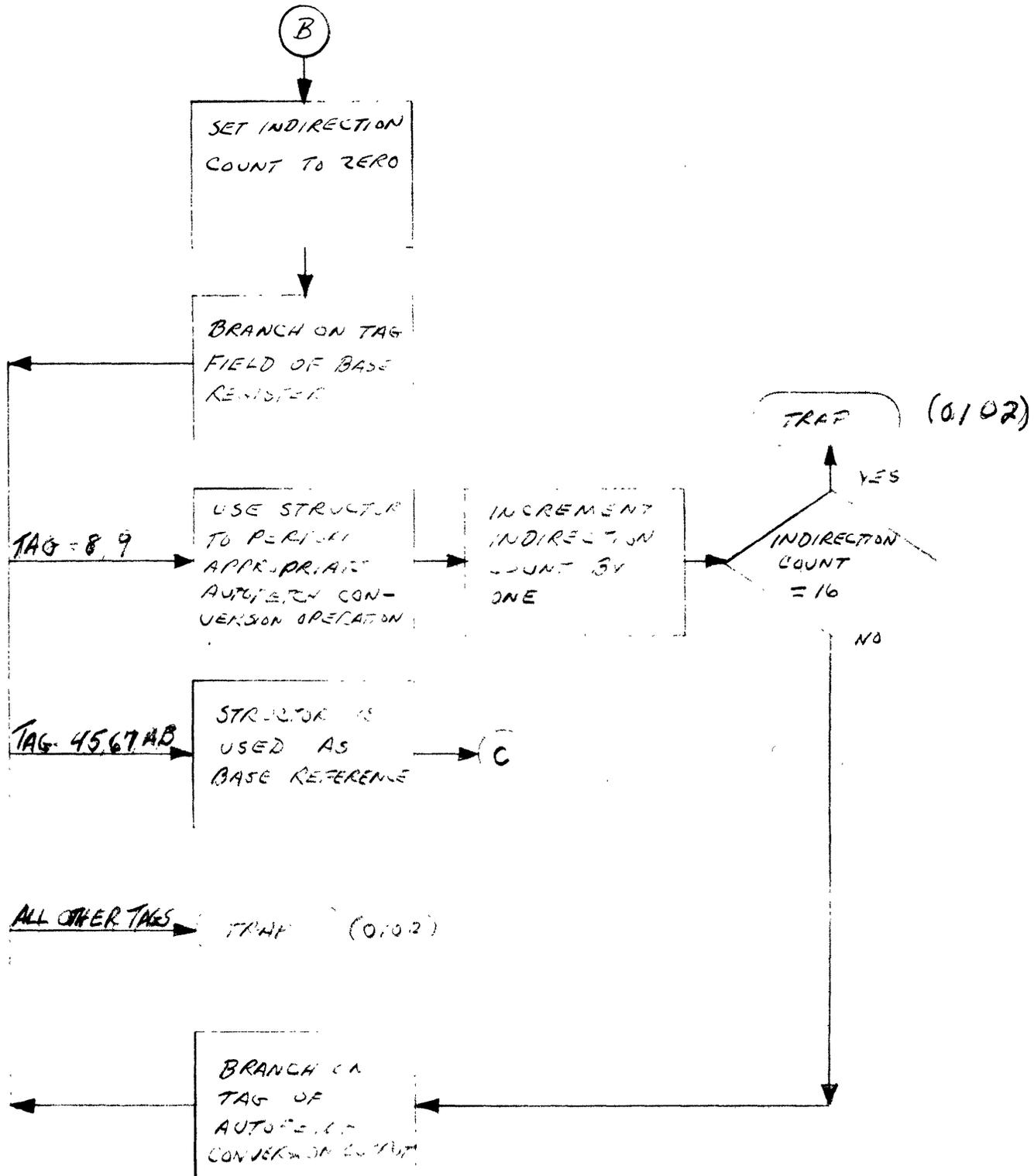


FIGURE 4-4. S-SYLLABLE EXTRACTION (Cont.)

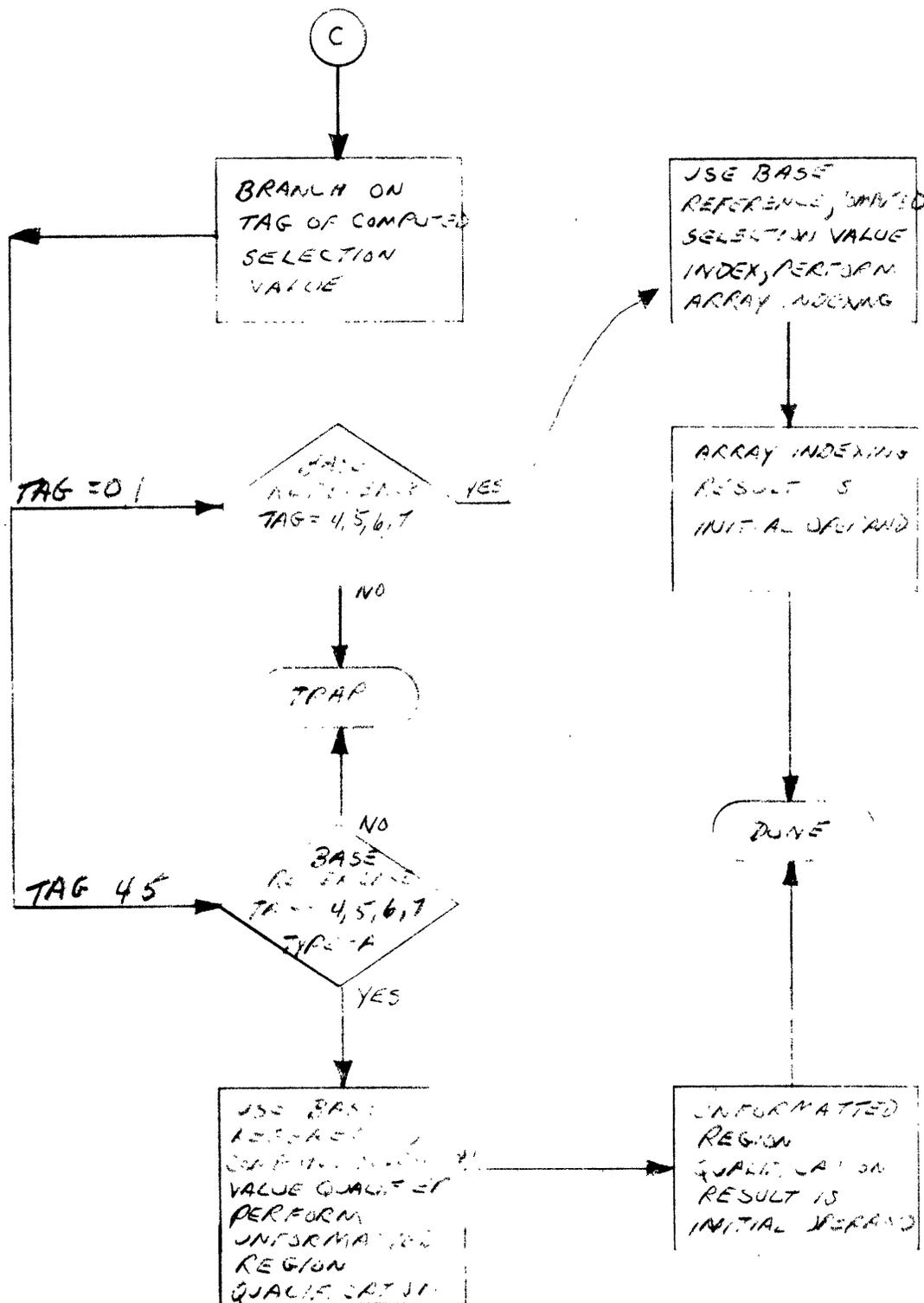


FIGURE 4-4. S-SYLLABLE EXTRACTION (Cont.)

4.3
(cont.)

the result value is used to form the initial operand derived from S-Syllable Extraction. The initial operand is a tagged binary integer. Unless the sum is within the range of tagged binary integers, an operand selection exception trap (0102) is generated or masked. When the index register contains any other tagged quantity, an operand selection exception trap (0102) is generated or masked. This case is shown in Figure 4-6.

The effect of these steps is to form an integer initial operand value from a literal value in the S-Syllable or from the sum of a literal value and the contents of a specified general register.

When the base register address field is nonzero, the S-Syllable Extraction process computes an initial operand structure using the base register contents as a base reference and the index register and immediate selection value fields as a description of a computed selection value. In this case, the extraction process proceeds according to the following steps:

- a. If the index register address field is zero, the computed selection value is a tagged binary integer equal in value to the immediate selection value field. This mode of extraction allows a known element of an array to be specified as the operand.
- b. If the index register address field is nonzero, the action taken depends on the TAG field of the selected general register. The possible actions are presented in Table 4-2. These actions always result in either a tagged logical word, tagged binary integer, or explicit-length modifier structure that is used as a computed selection value.

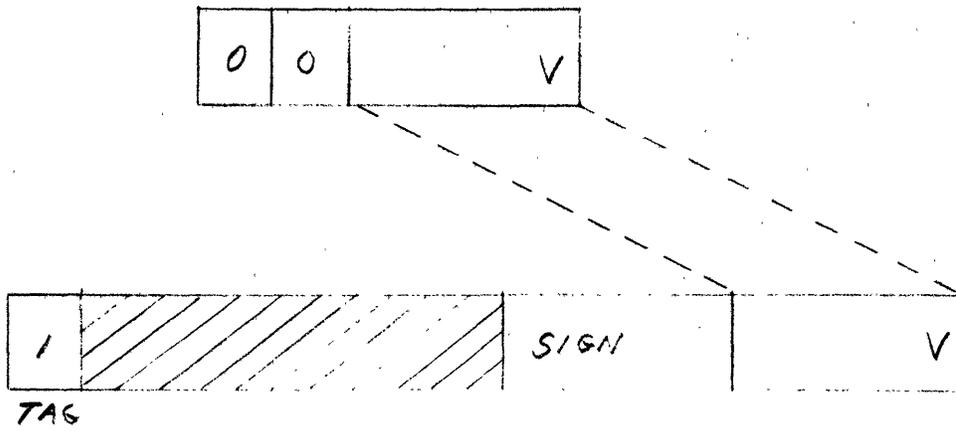


FIGURE 4-5. LITERAL VALUE EXTRACTION

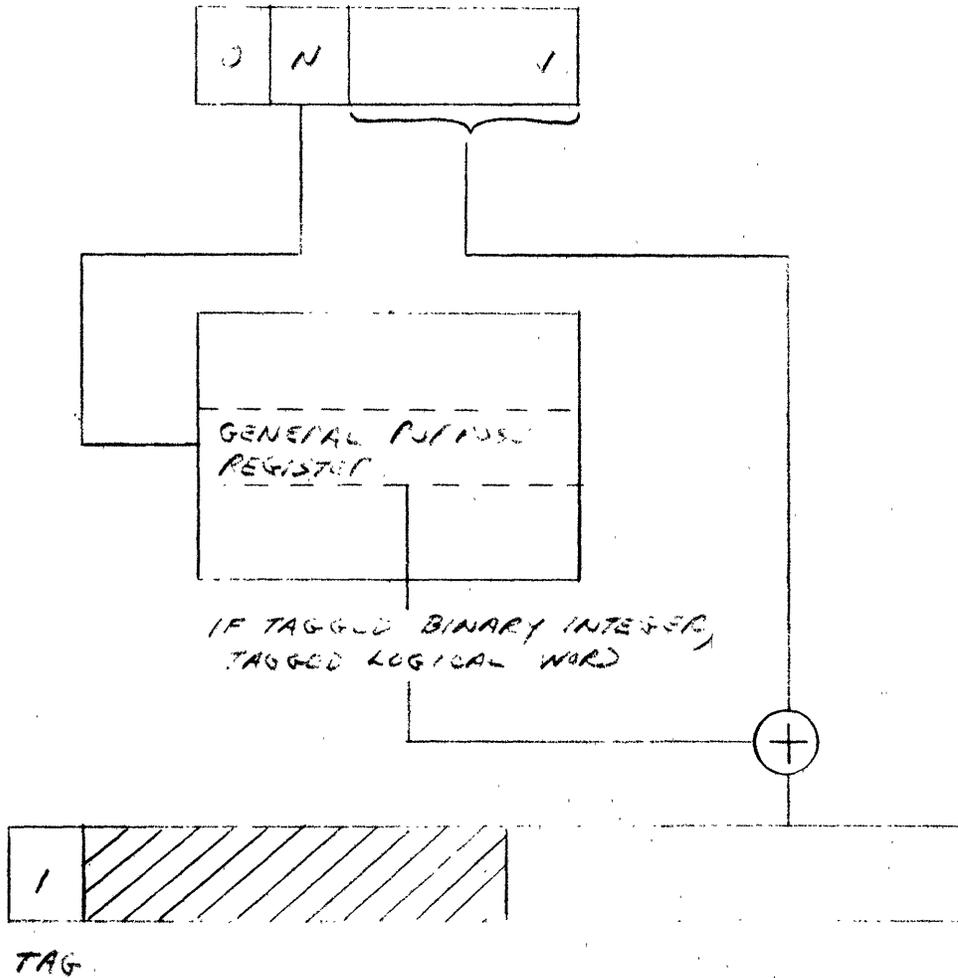


FIGURE 4-6. MODIFIED LITERAL VALUE EXTRACTION

- c. The contents of the general register specified by the base register address field of the S-Syllable are examined. The action taken depends on the TAG field of the general register as specified in Table 4-3. This action must result in either an explicit-length structor or an implicit-length datalink structor that is used as a base reference value.
- d. If the computed selection value is a tagged binary integer or tagged logical word, and if the base reference value is a data structor, then these quantities are used as index value and base reference value, respectively, and the Array Indexing operation (see Subsection 3.4) is performed. The result value becomes the initial operand formed by S-Syllable Extraction.
- e. If the computed selection value is an explicit-length modifier structor, and if the base reference value is an unformatted region structor, then these quantities are used as qualifier and base reference value, respectively, and the Unformatted Region Qualification operation (see Subsection 3.5) is performed. The result value becomes the initial operand formed by S-Syllable Extraction.

The last two steps above allow utilization of the computed selection value to derive the initial operand description from the base reference structor.

TABLE 4-2
INDEX REGISTER DETERMINED ACTIONS

TAG	NAME	ACTION*
0	Tagged Logical Word	A
1	Tagged Binary Integer	A
2	Tagged Floating Point	D
3	Unassigned	D
4	Explicit, Modifier, Alterable Structor	B
5	Explicit, Modifier, Nonalterable Structor	B
6	Explicit, Specifier, Alterable Structor	C
7	Explicit, Specifier, Nonalterable Structor	C
8	Implicit, Baselink Structor	D
9	Implicit, Baselink Structor	D
A	Implicit, Datalink, Alterable Structor	C
B	Implicit, Datalink, Nonalterable Structor	C
C-E	Unassigned	D
F	System Control, Structor	D

*The actions are specified as follows:

A- The value of the tagged logical word or tagged binary integer is added to the immediate selection value field of the S-Syllable, and the result value is used to form a tagged binary integer of equal value, which becomes the computed selection value. This mode of extraction allows indexed operand selection.

B- The explicit-length modifier structor becomes the computed selection value, and the immediate selection value field is not used. This mode of extraction allows a general register contained modifier structor to be used for Unformatted Region Qualification (see Subsection 3.5).

C- The explicit-length specifier or implicit-length datalink structor is used as a base reference structor, and the immediate selection value field of the S-Syllable is used to form a tagged binary integer index value. These two quantities are used by the Array Indexing operation (see Subsection 3.4) to compute a structor result. The result structor is then used as input to Autofetch Evaluation (see Section 3.3), which produces

(Continued)

(Table 4-2 cont.)

- C- a tagged quantity as a result. If the result is a tagged logical word, tagged binary integer, or explicit-length modifier structor, it becomes the computed selection value. If the output is any other tagged quantity, an operand selection exception trap (0102) is generated or masked. This mode of S-Syllable Extraction allows the selection value to be Autofetched from storage.
- D- An operand selection exception trap (0102) is generated or masked.

TABLE 4-3
BASE REGISTER DETERMINED ACTIONS

TAG	NAME	ACTION*
0	Tagged Logical Word	C
1	Tagged Binary Integer	C
2	Tagged Floating Point	C
3	Unassigned	C
4	Explicit, Modifier, Alterable Structor	A
5	Explicit, Modifier, Nonalterable Structor	A
6	Explicit, Specifier, Alterable Structor	A
7	Explicit, Specifier, Nonalterable Structor	A
8	Implicit, Baselink Structor	B
9	Implicit, Baselink Structor	B
A	Implicit, Datalink, Alterable Structor	A
B	Implicit, Datalink, Nonalterable Structor	A
C-E	Unassigned	C
F	System Control Structor	C

*The actions are specified as follows:

- A- The structor is used as the base reference quantity
- B- The implicit-length, baselink structor is used as input to the appropriate form of Autofetch Conversion (Sub-section 3.2), which is selected by the structor TYPE field. The output of Autofetch Conversion is a tagged

(Continued)

(Table 4-3 cont.)

- B- quantity. The indirection count is incremented by one
(Cont.) If it is equal to sixteen, an operand selection exception trap is generated or masked. Otherwise, the TAG field of the Autofetch Conversion output is used to select the next action to be performed in accordance with table 4-3. This mode of S-Syllable extraction allows a base reference value to be Autofetched from storage.
- C- An operand selection exception trap (0102) is generated or masked.

4.4 D-SYLLABLE EXTRACTION

Two forms of extraction of the D-Syllable are defined, each form being associated with a particular class of operations. The appropriate form of extraction is identified by the opcode field of the instruction. The two primary classes of instructions are data manipulation and branching instructions.

When the D-Syllable occurs in a data manipulation instruction, the initial operand structure formed by instruction extraction has the following attributes:

- a. The TAG, TYPE, and POSITION fields of the initial operand structure are determined by the typecode-index field of the D-Syllable, as specified in Table 4-4.

4.4 (Cont.)

TABLE 4-4
D-SYLLABLE INITIAL OPERAND STRUCTOR ATTRIBUTES

TYPECODE INDEX	TAG*	TYPE	POSITION**
0	B	Tagged Doubleword (0)	n.a.
1	B	Ministructor (3)	n.a.
2-7	n.a.	n.a.	n.a.
8	7	Bit String (0)	B=0, L=8, A=0
9	7	Bit String (0)	B=0, L=16, A=0
A	7	Bit String (0)	B=0, L=32, A=0
B	7	Binary String (1)	B=0, L=8, A=24
C	7	Binary String (1)	B=0, L=16, A=16
D	7	Binary String (1)	B=0, L=32, A=0
E	7	Hex.f.p. String (2)	L=4, S=0
F	7	Hex.f.p. String (2)	L=8, S=0

*The values in the TAG field correspond to the following cases:
7 - explicit-length, specifier, nonalterable; B - implicit-length, datalink, nonalterable.

**The following abbreviations are used: B - bit offset, L - length, A - alignment offset, S - significance truncation.

- b. Location field equal to the sum of the instruction location counter plus the value of the relative-displacement field of the instruction. The instruction location counter value used is the location of the opcode field of the instruction in which the D-Syllable occurs.
- c. The extent field is set to zero. This mode of extraction allows certain quantities located a fixed relative byte displacement from the instruction location to be specified as an operand of the instruction. This case is shown in Figure 4-7.

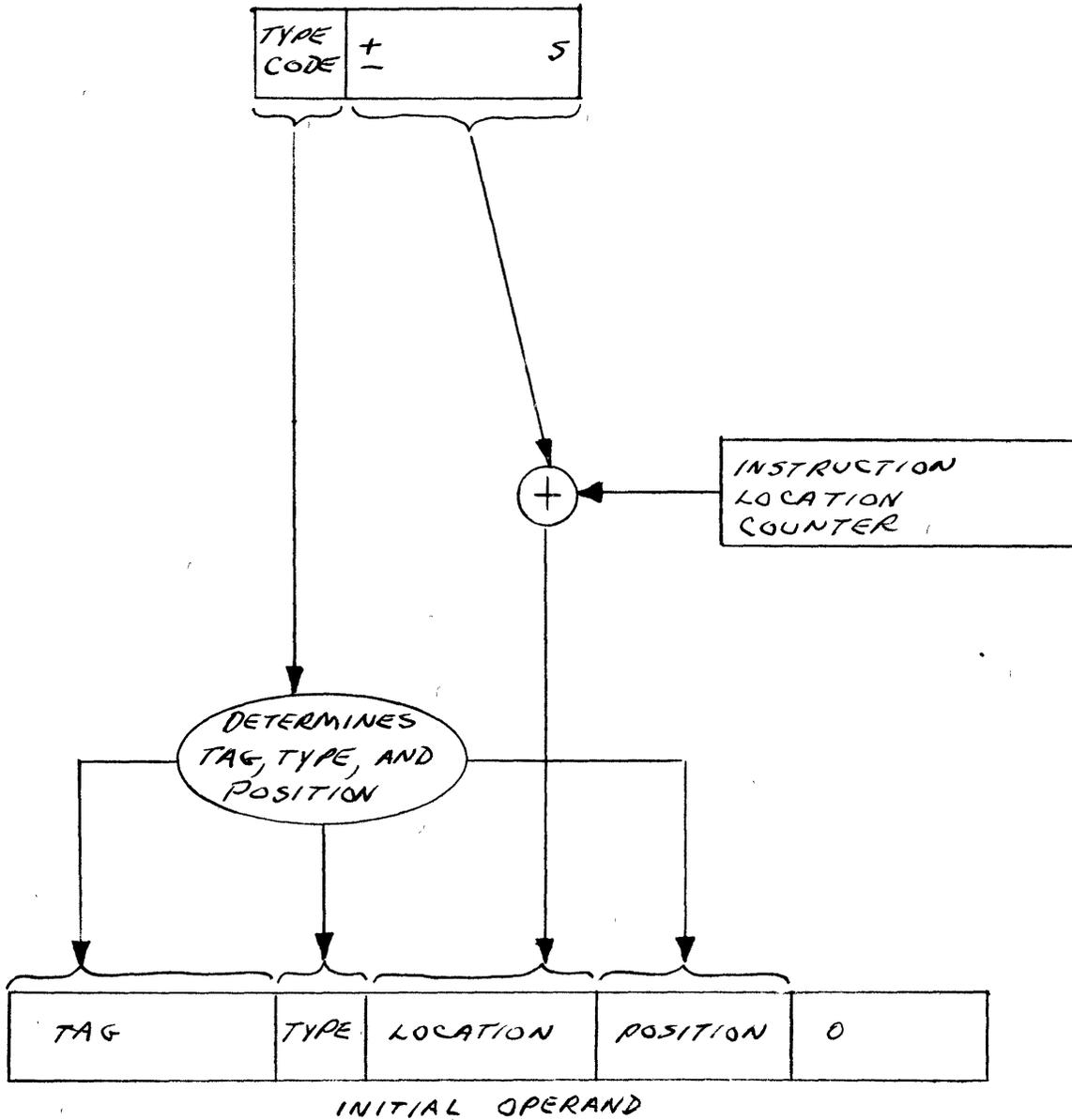


FIGURE 4-7. D-SYLLABLE EXTRACTION FOR DATA MANIPULATION INSTRUCTIONS

4.4 (Cont.)

When the D-Syllable occurs in a branching instruction, the typecode-index field and the relative-displacement field are used to specify the branch destination, which is always relative to the instruction location counter. In this case, the initial operand is a procedure index control structor identical to the current procedure index, except the instruction location counter has the following value:

- a. If the typecode-index field is a hexadecimal zero, the value equals the instruction location counter value plus the relative-displacement field value, the latter interpreted as a two's complement number. The instruction location counter value used is the location of the opcode field of the instruction in which the D-Syllable occurs.
- b. If the typecode-index field is non-zero, it is interpreted as a general register address. In this case, the contents of the selected general register are extracted and tested. If this quantity is not a tagged binary integer not less than zero or a tagged logical word, then an operand selection exception trap (0102) is generated or masked. Otherwise, the following actions take place:
 - i. The halfword at the location resulting from adding the relative-displacement field to the instruction location counter is extracted. This halfword value is used as the EXTENT field of an explicit-length specifier structor of type binary string. The LOCATION field of this structor equals the instruction location counter value plus the relative displacement field value plus two. The POSITION field describes binary strings with zero bit offset, 16-bit length, and 16-bit

4.4 (Cont.)

alignment offset.

- ii. The tagged binary integer or tagged logical word from the selected general register is used to index the binary string array described by the structor constructed in step i. This is accomplished according to the rules for Array Indexing as specified in Section 3.4.
- iii. Autofetch Conversion (see Section 3.2) is applied to the binary string structor produced by step ii. This results in a tagged binary integer, the value of which is treated as a new relative displacement.
- iv. The binary integer value derived in step iii is added to the sum of the current instruction location counter and the relative displacement field of the instruction.

The value used as location field in the new procedure index is the value computed in step iv.

This mode of extraction allows branching relative to the instruction location counter or relative to a fixed location in a procedure. If INDEX \neq 0, then the relative displacement is selected from a table under control of an index number in the general register specified by INDEX. This case is shown in Figure 4-8.

In either case, the data or control structor formed by D-Syllable Extraction becomes an initial operand of the instruction in which the D-Syllable occurs.

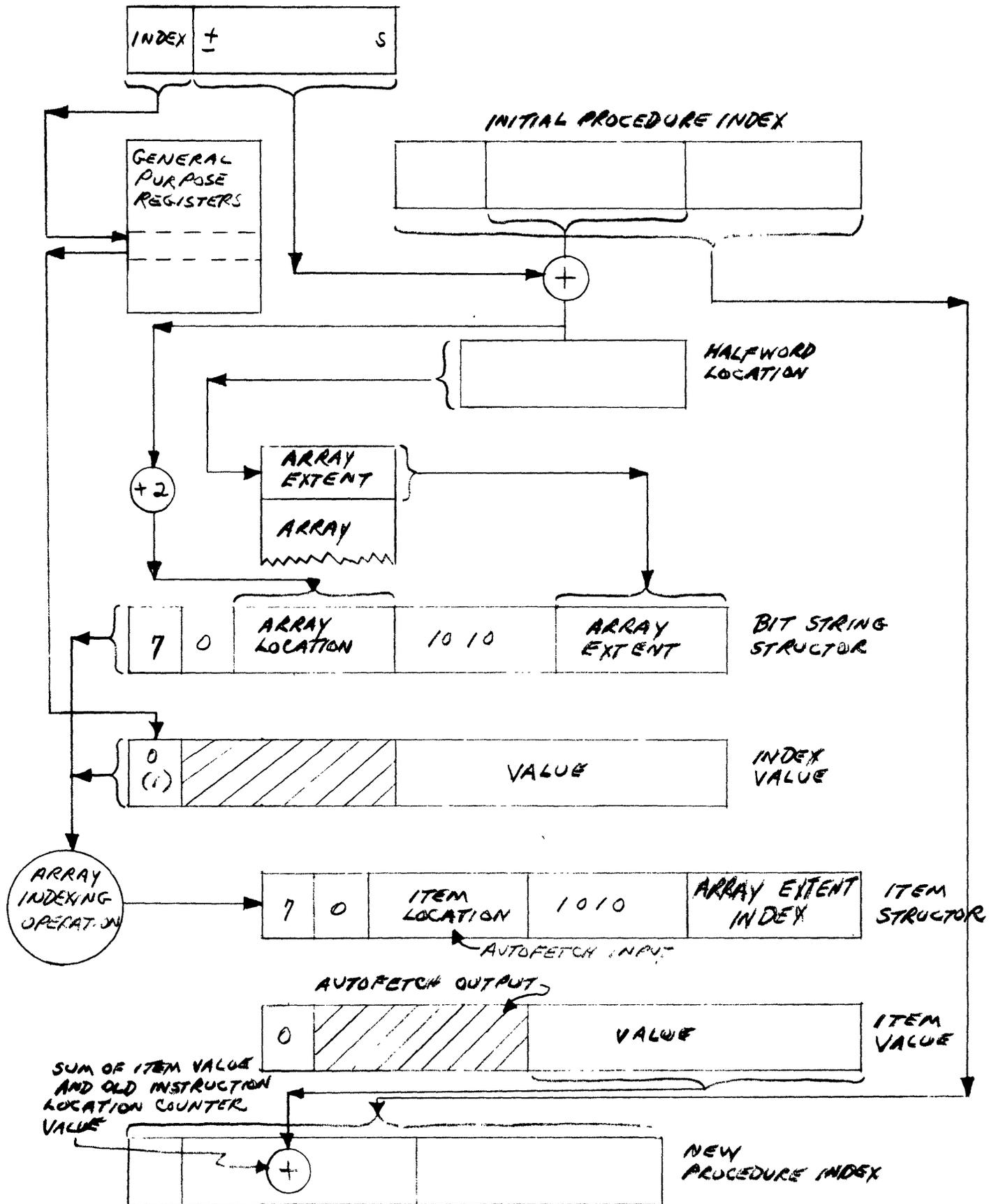


FIGURE 4-8. D-SYLLABLE EXTRACTION FOR BRANCING INSTRUCTIONS.

SECTION V
INSTRUCTIONS5.1 GENERAL

Instructions are the primitive operations performed by the processor. Instruction formats are specified in Subsection 2.7 and are referenced in this section using the mnemonics used in Subsection 2.7.

Every instruction is performed in two steps:

- a. Instruction Extraction, which is specified in Section IV, identifies the operation to be performed and the operands to which the operation is applied.
- b. Instruction Execution, which depends on the particular operation applied, consists of the actions required to perform the operation and restore a result, if necessary.

This section discusses the second step classified in terms of the operations available..

The following subsections specify the instruction set, which consists of Data Manipulation, General Register Loading/Storing, Branching, Structor Manipulation, Task Control, and Input/Output Control Instructions.

Each instruction is specified to operate on a defined set of operand types. If the operand actually presented is not one of the defined types, an illegal operand trap (0200) is generated or is masked.

5.2 DATA MANIPULATION INSTRUCTIONS

The data manipulation instructions are used to perform certain arithmetic and logical transformations on operand values. In general, the type of transformation performed depends on a function implied by the operation code and on the data attributes associated with the operands. A specific function can normally be applied to more than one type of data representation. For example, the ADD function

can be applied to two's complement binary integers, hexadecimal floating point numbers, and decimal strings. The type of operand used in a data manipulation instruction is derived either from a tag appended to the data representation or from a structor describing it.

Operations are defined in terms of compatible data types. Two operands are considered to have compatible data types if they both fall into the same class as listed below:

- a. Tagged logical words and tagged binary integers.
- b. Tagged floating point numbers.
- c. Signed and unsigned zoned and packed decimal strings.
- d. Byte and translated byte strings.

The high, low, equal, high-order truncation, low-order truncation, and overflow indicators may be set by the execution of a data manipulation instruction. These indicators are always reset prior to the execution of a data manipulation instruction. Furthermore, in any case where a result operand is not alterable, an operand selection exception trap (0103) is generated.

5.2.1 Add

Formats: RR, RS, SR, SS, RD

The effective A operand is added to the effective B operand, and the result replaces the B operand. The effective A and B operands are both derived by applying Autofetch Evaluation to both the initial A and B operands. The operand description from B operand Autofetch application is used to replace the result. The type of addition performed depends on the type of effective operands produced by Autofetch Evaluation. The valid operand combinations are discussed below. The high, low, or equal condition indicator is set, depending on the result of the ADD instruction being positive, negative, or zero, respectively.

5.2.1.1 Binary Addition

If the effective A and B operands are either tagged logical words or tagged binary integers in any combination, the value portions of these operands are treated as 32-bit binary integers, and a binary addition is performed on these values. The result value generated is used to form either a tagged logical word or a tagged binary integer, depending upon whether the effective B operand is a tagged logical word or tagged binary integer, respectively. The resulting tagged quantity is then restored to the effective B operand location.

A binary addition is performed by taking bits of equal weight (identical position) in the two operand values and computing a result bit of equal weight according to the following table:

A operand bit	0 0 0 0 1 1 1 1
B operand bit	0 0 1 1 0 0 1 1
Carry from last position	0 1 0 1 0 1 0 1
Result bit	<div style="border-top: 1px solid black; display: inline-block; width: 100%;"></div> 0 1 1 0 1 0 0 1 - <i>sum</i>
Carry to next position	0 0 0 1 0 1 1 1 - <i>carry</i>

The last position refers to the bit position of next lower weight, and next position refers to the bit position of next higher weight. The carry into the bit position of least weight is always zero.

The result of the binary addition is formed by performing the following actions. The A and B operand values are effectively extended to 33 bits, depending upon the type of value:

- a. If the operand is a tagged logical word, then the leftmost bit of the extended value is set to 0.
- b. If the operand is a tagged binary integer, then the leftmost bit of the extended value is identical to the leftmost bit of the original operand value.

c. The remaining 32 bits of the extended value are equal to the 32 corresponding bits of the original operand value.

A binary addition is then performed on the two extended operand values, generating a 33 bit result with a possible carry from the leftmost bit position. If this carry and the carry from the bit position of next lower weight are not identical, an overflow occurs, and the overflow condition indicator is set.

If the B operand is a tagged logical word, then the result is used to form a tagged logical word. If the leftmost bit of the extended result and the leftmost bit of the extended A operand are 1, then the result is not restored, and an arithmetic exception trap (0400) is generated. Otherwise the value of the resulting tagged logical word is equal to the rightmost 32 bits of the extended result.

If the B operand is a tagged binary integer, then the result is used to form a tagged binary integer. If the leftmost two bits of the extended result are not identical, then the overflow condition indicator is set. The value of the resulting tagged binary integer is equal to the rightmost 32 bits of the extended result. If the overflow condition indicator is set, an arithmetic exception trap (0400) is generated or masked, depending on the arithmetic exception trap mask indicator.

In either case, the high, low, or equal condition indicator is set as a function of the result value being greater than, less than or equal to zero.

5.2.1.2 Hexadecimal Floating Point Addition

If the effective A and B operands are tagged floating point quantities, a hexadecimal floating point add is performed. The result value is formed as a tagged floating point quantity and restored to the effective B operand location.

5.2.1.2
(Cont.)

Hexadecimal floating point addition is described in the following steps:

- a. Step One - The mantissa of the operand with the smaller exponent is shifted right by a number of digit positions equal to the difference between the larger and smaller exponent. (The rightmost shifted 13 digits are saved for the secondary result.)
- b. Step Two - The aligned mantissae of A and B operands are combined. A hexadecimal add of the two mantissae is performed if the signs of the two operands are the same. The result sign is the same as the operand sign:

If the signs of the A and B operands differ, a hexadecimal subtract of the A from the B operand is performed. The result will assume the sign of the A or B operand, depending on which is the larger.

With the floating point round mode indicator set, a hexadecimal eight is added to the digit position immediately to the right of the rightmost operand digit with the larger exponent.

When these operations result in a mantissa overflow, the result mantissa is shifted right one hexadecimal position and the exponent is increased by one. If this results in an exponent overflow, an arithmetic exception trap (0401) is generated or masked.

- c. Step Three - The result is normalized. This step is performed only if the significance mode indicator is reset. Normalization consists of shifting the result's mantissa left one digit at a time until the high order digit is nonzero. The exponent is decreased by one each time the mantissa is shifted. If this operation causes the exponent to underflow, an arithmetic exception trap (0401) is generated or masked.

- d. Step Four - The leftmost 60 bits (including the sign and exponent) of the result form the primary result, and are stored in the B operand location. The remaining digits of the result are the secondary result. With the secondary result mode indicator set, the secondary result is stored as a tagged floating point quantity in general purpose register zero. The exponent of the secondary result is 13 less than the exponent of the primary result. If decreasing the primary result exponent by 13 causes it to underflow and the secondary result mode indicator is set, an arithmetic exception trap (0401) is generated or masked.

5.2.1.3 Decimal String Addition

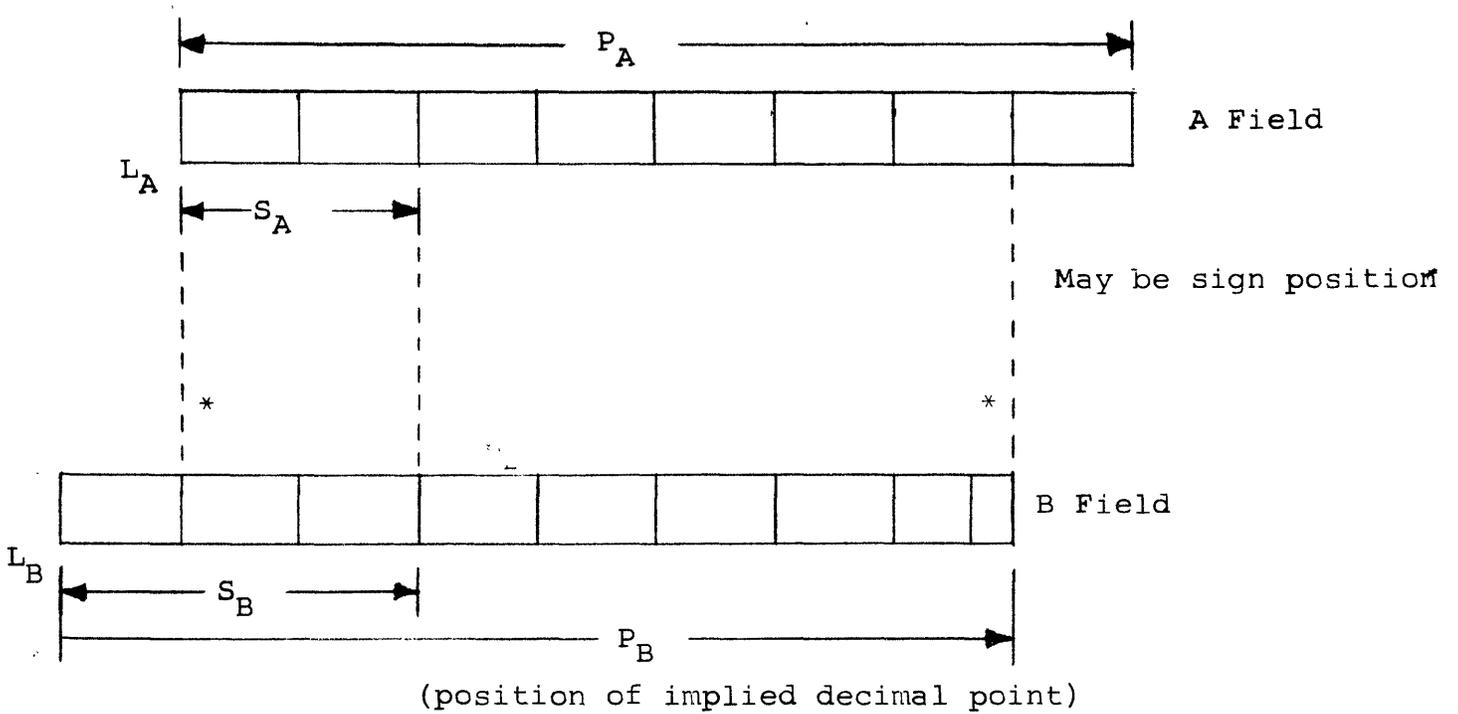
When the effective A and B operands are explicit-length specifier structures of the following types:

- a. Unsigned zoned decimal string.
- b. Zoned decimal string.
- c. Packed decimal string.
- d. Unsigned packed decimal string.

The values of the associated strings are treated as signed or unsigned decimal numbers with stated scale and precision. A scaled decimal addition may then be performed on these values. The result value generated replaces the B operand string if it is alterable.

Scaled decimal addition is performed by taking digits of equal weight in the two operand values and computing a result digit of equal weight. The result depends on the operand values and their signs. The overall sequencing of scaled decimal addition and subtraction is described in the following steps and shown in Figure 5-1.

- a. Step One - The first operation is to locate and decode the sign code of each operand. The manner in which



- L_A = location of A operand (to byte or half-byte resolution)
- L_B = location of B operand (to byte or half-byte resolution)
- S_A = scale of A operand
- S_B = scale of B operand
- P_A = length of A operand
- P_B = length of B operand

FIGURE 5-1. ADDITION OF SCALED DECIMAL STRINGS

5.2.1.3
(Cont.)

this is accomplished depends on the type of decimal string involved. The type possibilities are:

1. An unsigned zoned decimal string operand - the sign code is always positive, and the class is always one.
2. A zoned decimal string operand - the sign code is located in the zone position of the rightmost byte of the string. This is located at the byte position in storage $L_A + P_A$ or $L_B + P_B$ (refer to Figure 5-1). The sign code is decoded according to the conventions stated in Appendix A.
3. A packed decimal string operand - the sign code is located in the rightmost four-bit field of the rightmost byte of the string. This is located at the byte location in storage $L_A + P_A$ or $L_B + P_B$. The sign code is decoded according to the conventions stated in Appendix A.

The sign and class code of the B operand, and the sign of the A operand result from this step.

- b. Step Two - The second operation is to align digits in the A and B operands of equal weight. This is accomplished by allowing the digits at $L_A + S_A$ and $L_B + S_B$ to establish the alignment. Equivalently, it is necessary to align the digits at $L_A + S_A + \min(P_A - S_A, P_B - S_B)$ in the A operand with the digits at $L_B + S_B + \min(P_A - S_A, P_B - S_B)$ in the B operand, where min is the minimum of the two values in parentheses. A pairing of A and B operand digits exists from the pair established in this latter way and in digits of increasing significance up to the pair of digits at $L_A + S_A - \min(S_A, S_B)$ in the A operand, and at $L_B + S_B - \min(S_A, S_B)$ in the B operand. This alignment is shown by asterisks in Figure 5-1. Note that such a pairing will not exist if $S_A - P_A > S_B$ or if $S_B - P_B > S_A$.

5.2.1.3
(Cont.)

- c. Step Three - The third operation depends on the setting of the decimal round mode indicator. If this indicator is set and $P_B - S_B < P_A - S_A$, then the A operand digit at position $S_A + P_B - S_B$ is added to a digit value of five. The result is discarded, but the resulting carry is saved for the next step.
- d. Step Four - The fourth operation is the combining of aligned digits in the A and B operands. The digits are combined by decimally adding if the A and B operand signs are the same, and by decimally subtracting the A operand digit from the B operand digit if the signs differ. Carries from less significant digit positions and from the rounding operation in step three are taken into consideration when performing the addition, while borrows from more significant digit positions are taken into consideration when performing the subtraction. If $S_A < S_B$, the carries are made into or borrows made from the most significant B operand digits. When a carry is made out of the leftmost B operand digit, the decimal overflow indicator is set, and an arithmetic exception trap (0400) takes place if the arithmetic exception trap mask bit is not set. When a borrow is made out of the leftmost B operand digit, a recomplement cycle is initiated. If $S_A > S_B$, a high-order truncation of the A operand takes place, and the high-order truncation indicator is set. If the B operand is a signed or unsigned zoned decimal string, the zone portion of each byte processed is preserved by the decimal addition operation. The interpretation of digit codes for this step is specified in Appendix A.
- e. Step Five - The final step generates a new result sign code in the B operand, if necessary. Replacement of the sign code is necessary only if the signs of the operands were different and a recomplementation cycle

of the result was required. The sign code is generated as a function of the result sign and the class code derived from the B operand, as specified in Appendix A. If the result has a negative sign and the B operand is an unsigned zoned or packed decimal string, an arithmetic exception trap (0403) is generated or masked.

The recomplementation cycle consists of taking the ten's complement of the result computed in step four. This is accomplished by subtracting every digit from nine and adding one to the result.

5.2.2 Subtract

Formats: RR, RS, SR, SS, RD.

The effective A operand is subtracted from the effective B operand, and the result replaces the B operand. The effective A and B operands are both derived by applying Autofetch Evaluation to both the initial A and B operands. The operand description from B operand Autofetch application is used to replace the result. The type of subtraction performed depends on the type of effective operands produced by Autofetch Evaluation. The valid operand combinations are discussed below. The high, low or equal condition indicator is set, depending on the result of the SUBTRACT instruction being positive, negative, or zero, respectively.

5.2.2.1 Binary Subtraction

If the effective A and B operands are either tagged logical words or tagged binary integers in any combination, the value portions of these operands are treated as 32-bit binary integers, and a binary subtraction is performed on these values. The result value generated is used to form either a tagged logical word or a tagged binary integer, depending upon whether the effective B operand is a tagged logical

word or a tagged binary integer, respectively. The resulting tagged quantity is then restored to the effective B operand location.

A binary subtraction is performed in a fashion identical to a binary addition, with the exception that a fourth step is included in the creation of an extended A operand value (see subsection 5.2.1.1) This step is as follows:

d. The Two's complement of the operand value is taken.

Taking the two's complement of a value is accomplished by inverting every bit in the value, and then performing an addition of 1 to the bit position of least weight in the result (with carries propagated to bit positions of higher weight).

The condition indicators are set and arithmetic exception traps are generated as for binary addition.

5.2.2.2 Hexadecimal Floating Point Subtraction

If the effective A and B operands are tagged floating point quantities, a hexadecimal floating point subtraction is performed. The result value is formed as a tagged floating point quantity and is restored to the effective B operand location.

The procedure for hexadecimal floating point subtraction is identical to the procedure for hexadecimal floating point addition, with the exception that the A operand sign is inverted before step two (see subsection 5.2.1.2).

The condition indicators are set and arithmetic exception traps are generated as for hexadecimal floating point addition.

5.2.2.3 Decimal String Subtraction

When the effective A and B operands are explicit-length specifier structures of the following types:

- a. Unsigned zoned decimal string.
- b. Zoned decimal string.
- c. Packed decimal string.
- d. Unsigned packed decimal string.

the values of the associated strings are treated as signed or unsigned decimal numbers with stated scale and precision. A scaled decimal subtraction may then be performed on these values. The result value generated replaces the B operand string, if it is alterable.

Scaled decimal subtraction is accomplished in a fashion identical to decimal string addition, with the exception of a step inserted following Step One of decimal addition (see subsection 5.2.1.3). This step inverts the A operand sign value.

The condition indicators are set and arithmetic exception traps are generated as for decimal string addition.

5.2.3 Multiply

Formats: RR, RS, SR, SS, RD.

The effective B operand is multiplied by the effective A operand, and the product replaces the B operand value. The effective A and B operands are derived by applying Autofetch Evaluation to the initial A and B operands. The operand description from B operand Autofetch application is used to replace the product. The type of multiplication performed depends on the type of effective operands produced by Autofetch Evaluation. The valid operand combinations are described in the following subsections.

5.2.3.1 Binary Multiplication

When the effective A and B operands are either tagged logical words or tagged binary integers in any combination, the value portions of these operands are treated as 32-bit

5.2.3.1
(Cont.)

binary integers, and a binary multiplication is performed on these values. The result value generated is a 64-bit binary integer. If the effective B operand is a tagged logical word, then the leftmost 32 bits of the result value (high-order product) are used to form a tagged logical word. When the MULTIPLY instruction is executed in secondary result mode, the tagged logical word high-order product is placed in general register R0. Otherwise, it is discarded. The rightmost 32 bits of the result value (low-order product) are used to form a tagged logical word, which is restored to the effective B operand location. If the effective B operand is a tagged binary integer, the 64-bit result value is used to form two tagged binary integers, consisting of a high- and low-order product. These quantities are restored to general register R0 (secondary result mode) and the effective B operand location, respectively.

A binary multiplication is performed as a process of repetitive binary addition. The multiplier and the multiplicand are derived from the A and B operand values, respectively, as follows:

If the operand is a tagged binary integer and is negative, the two's complement of the operand value is taken, and the result is used. Otherwise, the unaltered value is used.

The repetitive addition process is described with reference to Figure 5-2.

The product workspace is 64 bits in length and is initially set to all zero bits. The multiplication is performed by considering the Kth bit from the left end of the multiplier, for K from 31 to 0 (bit positions 31 to 0). The multiplier bit for each K is examined. If the bit is 1, the multiplicand, offset by K + 1 bit positions from the left end of the product workspace, is added to the product workspace.

5.2.3.1
(Cont.)

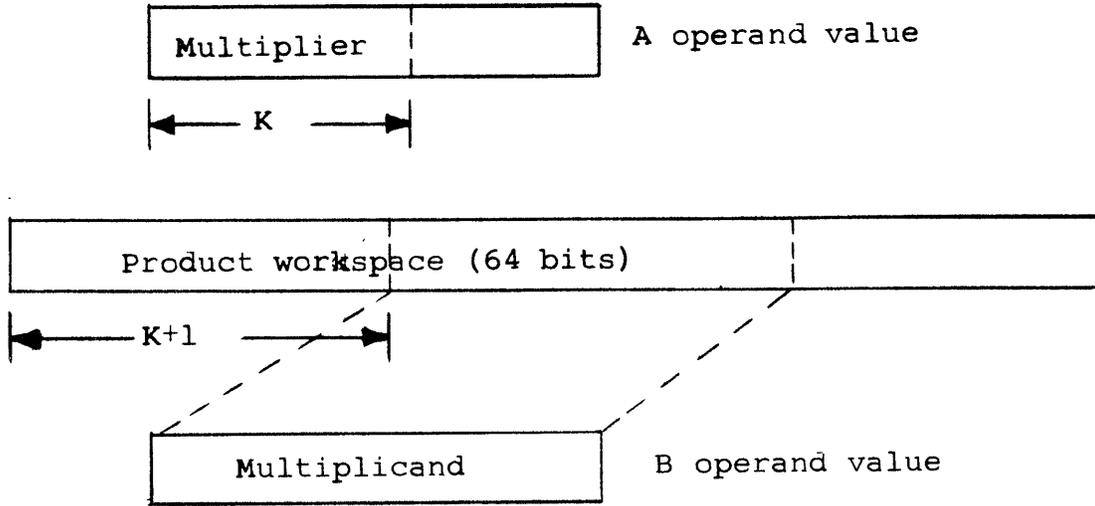


FIGURE 5-2. MULTIPLICATION OF BINARY VALUES

This addition is a binary addition as described in Subsection 5.2.1.1. The 64 bits of the product workspace form the result value, from which the high- and low-order product are formed.

If only one operand was complemented previous to the above addition process, the two's complement of the result in the product workspace is taken. In all cases, the 64 bits of the product workspace forms the result value from which the high- and low-order products are taken.

When the B operand is a tagged binary integer, the high-order product is examined to determine whether it consists of all zero or one bits. If not, the overflow and high-order truncation indicators are set. If so, but the left-most bit of the low-order product is different from the bits of the high-order product, then the overflow indicator is set. With overflow indicated and the arithmetic exception trap mask bit not set, an arithmetic exception trap (0400) is generated.

5.2.3.2 Hexadecimal Floating Point Multiplication

If the effective A and B operands are tagged hexadecimal floating point quantities a hexadecimal floating point multiplication is performed. The result has a maximum mantissa length of 26 digits. The most significant 13 digits are used to form a tagged floating point quantity which is delivered to the B operand location. The remaining digits form a secondary result which is delivered to general register R0, if the secondary result mode indicator is set. The hexadecimal floating point multiplication is described in the following steps:

- a. Step One - The first step of the multiplication is performed only if the significance mode indicator is off. It consists of prenormalizing the A and B operands by shifting them left until the high-order digit is non-zero, and decreasing the exponent by one for each hexadecimal digit position shifted.
- b. Step Two - The second step is to form the product of the mantissae of the A and B operands. The multiplication is performed in a 104-bit workspace which is initially set to zeros. The product is formed by considering the Kth bit from the rightmost bit of the A operand mantissa, for all K from 0 to 52. For each K, the A operand bit is examined and, if the bit is 1, the B operand mantissa, offset by K bit positions from the right end of the workspace, is added to the workspace (See Figure 5-3).

The exponent associated with this product is the sum of the exponents of the A and B operands less 64.

5.2.3.2
(Cont.)

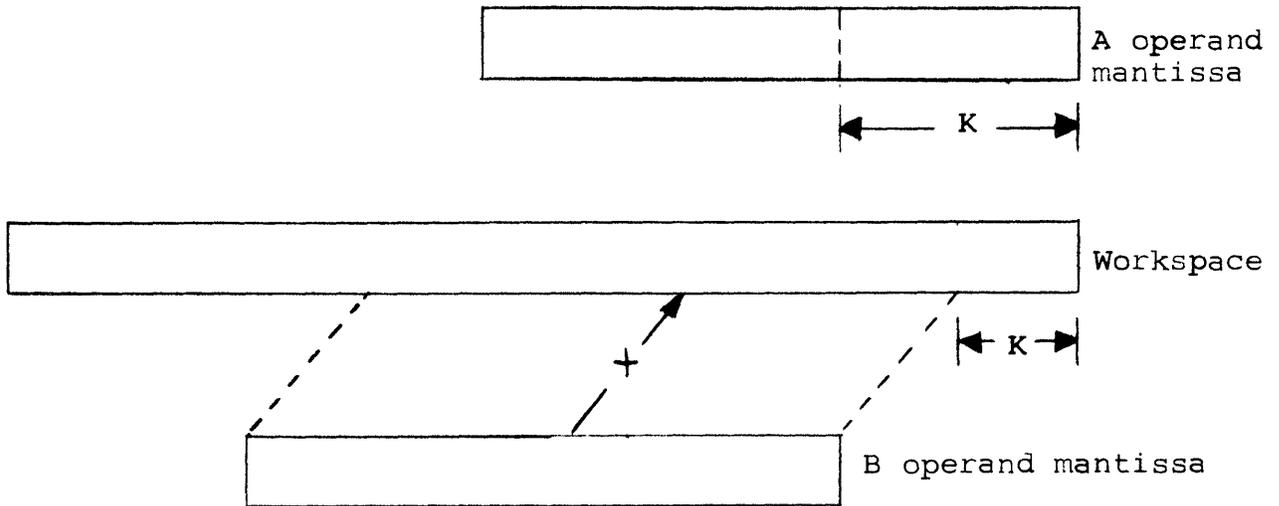


FIGURE 5-3. HEXADECIMAL FLOATING POINT MULTIPLICATION

- c. Step Three - The third step depends on the setting of the floating point mode indicators. If the Significance mode indicator is not set, the result is shifted left in the workspace until the high-order hexadecimal digit is nonzero (postnormalization). The exponent is decreased by the number of hexadecimal digit positions shifted. If the exponent underflows an arithmetic exception trap (0401) is generated. If the result is zero, the exponent and sign are also set to zero.

If the significance mode indicator is set, the result is shifted right or left until the number of significant hexadecimal digits in the high order 13 digit positions is equal to the number of significant digits originally in the operand which contained the smaller number of significant digits (significance correction). (The number of significant digits in a field is the number of digits in the field less the number of leading zero digits). The exponent is adjusted by an amount equal to the number of hexadecimal digit positions shifted.

- d. Step Four - The fourth step is performed only if the significance mode indicator is on. This process consists of shifting the result left one hexadecimal digit position if the leftmost of the significant digits of the A operand was less than hexadecimal four. If the shift takes place the exponent is decremented by one. (NOTE: the purpose of this step is to maintain a statistically correct number of significant digits during a series of operations).
- e. Step Five - The fifth step is performed only if the floating point round mode indicator is set. The process consists of adding a hexadecimal eight to the 14th digit from the left of the result field.
- f. Step Six - The sixth step delivers the result. The 13 high-order hexadecimal digits of the result together with the sign and exponent are used to form a tagged floating point quantity which is stored in the B operand location. If the secondary result mode indicator is set, the remaining 13 hexadecimal digits of the result, together with the sign and the exponent decremented by 13, are used to form another tagged floating quantity which is stored in general register R0. If decrementing by 13 causes the exponent to underflow an exponent, sign, and mantissa of zero are stored in register R0.

5.2.3.3 Decimal String Multiplication

If the effective A and B operands are explicit-length specifier structures of the following types:

- a. Zoned decimal string.
- b. Unsigned zoned decimal string.
- c. Packed decimal string.
- d. Unsigned packed decimal string.

5.2.3.3
(Cont.)

the values of the associated strings are treated as signed or unsigned decimal numbers with stated scale and precision, and a scaled decimal multiplication is performed on these values. The result value generated has a maximum precision of 64 decimal digits. A portion of the result value with precision and scale equal to the precision and scale of the B operand replaces the B operand string if it is alterable. If the MULTIPLY instruction is executed in secondary result mode, a decimal string structor must be contained in general register R0. The high-order portion of the result value not placed in the B operand string is placed in this secondary result string. If the instruction is not executed in secondary result mode, the portion of the result value not placed in the B operand string is examined to determine whether it consists of all zero digits. If not all zeros, the overflow and high-order truncation indicators are set. With a overflow indicated and the arithmetic exception trap mask bit not set, an arithmetic exception trap (0400) is generated. The low-order truncation indicator is set if least significant result digits are lost.

A scaled decimal multiplication is performed by a process of repetitive addition. The multiplier and multiplicand are the A and B operand values as shown in Figure 5-4.

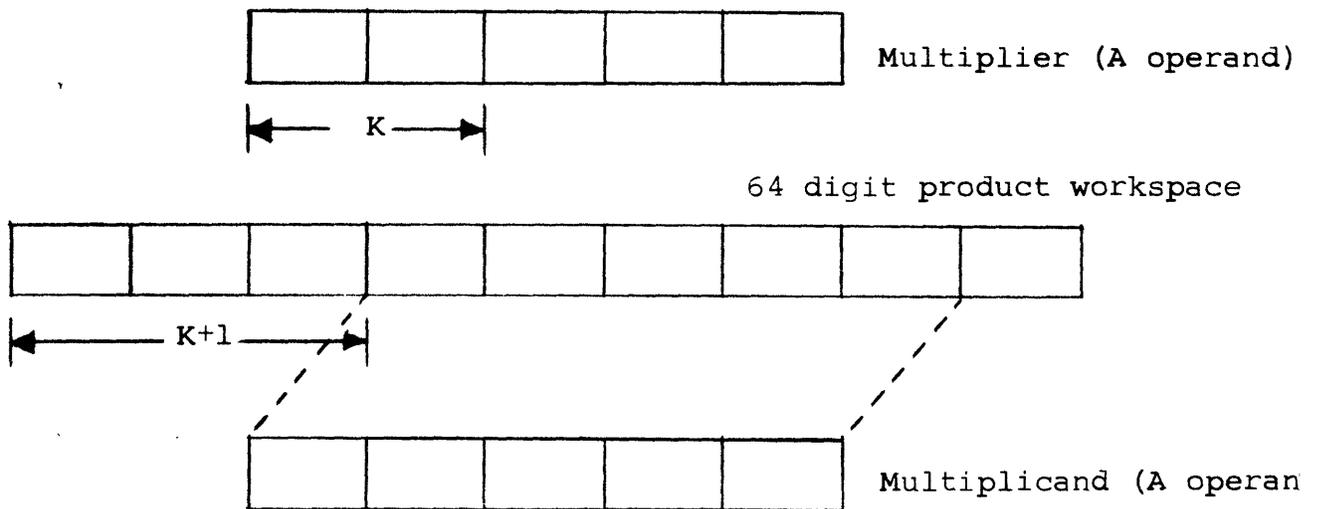


FIGURE 5-4. MULTIPLICATION OF SCALED DECIMAL STRINGS

The product workspace is 64 digits in length and is initially set to all zero digits. The multiplication is performed by considering the Kth digit from the leftmost digit of the multiplier for all K from the precision of the multiplier to 0. For each K, the multiplier digit extracted has a value N ($0 \leq N \leq 9$). The multiplicand, offset by K + 1 digit positions from the left end of the product workspace is added N times to the product workspace. The addition performed is a decimal addition, as described in Subsection 5.2.1.3.

The precision of the product is equal to the sum of the precisions of the A and B operand strings. The scale of the product is equal to the sum of the scales of the A and B operand strings. The scale and precision of the product are used to properly align and restore the result to the B operand string and to the secondary result string, if required. If the B operand is a zoned decimal string, restoration of the result preserves zones.

The result sign code is generated from the result sign and B operand sign class code as specified in Appendix A. The result sign is negative if the A and B operand signs differ. The A and B operand signs and the B operand sign class code are derived from the A and B operand sign codes as specified in Appendix A.

If the result sign is negative and the B operand an unsigned zoned or packed decimal string, the quantities in the product workspace are not placed in the B operand string and secondary result string. An arithmetic exception trap (0403) is then generated or masked, depending on the setting of the arithmetic exception trap mask bit.

5.2.4 Divide

Formats: RR, RS, SR, SS, RD.

The effective B operand is divided by the effective A operand, and the quotient replaces the B operand value. The

effective A and B operands are derived by applying the Autofetch Evaluation to the initial A and B operands. The operand description from B operand Autofetch application is used to replace the quotient. The type of division performed depends on the type of effective operands produced by the Autofetch Evaluator. An attempt to divide by zero is monitored and an arithmetic exception trap (0404) is generated if a zero divisor is detected and the arithmetic exception trap mask indicator is set.

5.2.4.1 Binary Division

If the effective A and B operands are tagged logical words or tagged binary integers in any combination, the value portions of these operands are treated as 32-bit binary integers, and a binary division is performed using these values. The result values computed are a 32-bit binary integer quotient and a 32-bit binary integer remainder. A tagged logical word or tagged binary integer equal in value to the quotient is formed, depending on the B operand type, and this quantity is restored to the effective B operand location. If the DIVIDE instruction is executed in secondary result mode, a tagged logical word or tagged binary integer equal in value to the remainder is formed, depending on the B operand type, and this quantity is placed in general register RO. If the DIVIDE instruction is not executed in secondary result mode, the remainder is discarded.

A binary division is performed by a process of repetitive subtraction. The divisor (A operand) is tested for zero, and if it is zero, an arithmetic exception trap is generated or masked. The divisor and the dividend are derived from the A and B operand values, respectively, as follows:

If the operand is a tagged binary integer and is negative, the two's complement of the operand value is taken, and the result is used. Otherwise, the unaltered value is used.

5.2.4.1 (Cont.) The repetitive subtraction process is described with reference to Figure 5-5. The quotient/remainder workspace is 64 bits in length and is initially set to all zero bits.

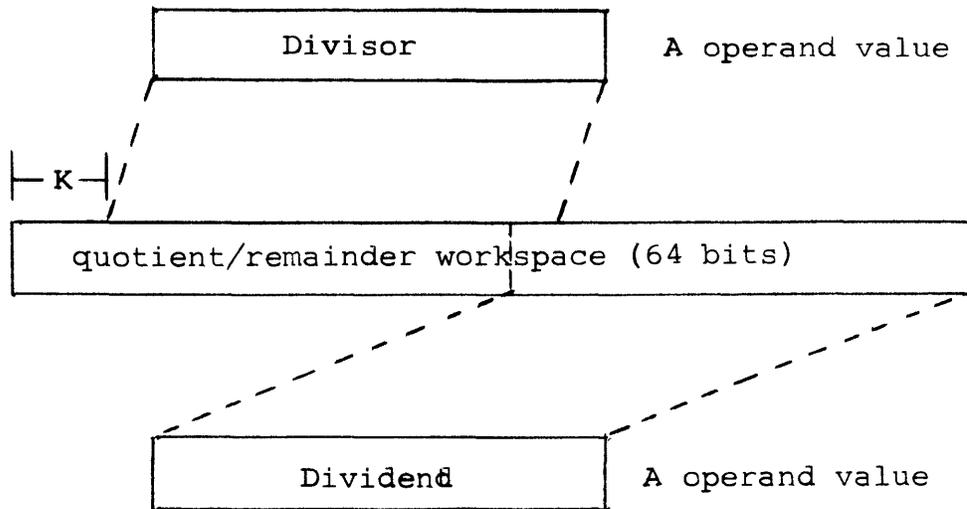


FIGURE 5-5. DIVISION OF BINARY VALUES

The dividend is placed (added) into the rightmost 32 bits of the quotient/remainder workspace. The division process is then accomplished by performing the following steps, offsetting the divisor by K bits from the left end of the quotient/remainder workspace, for K from 1 to 32.

- a. Step One - The divisor is subtracted from bits K to $K + 31$ in the quotient/remainder workspace, where the leftmost bit in the workspace is bit 0. This is equivalent to adding the two's complement of the divisor to bits K to $K + 31$ of the workspace (See Subsection 5.2.1.1).
- b. Step Two - If a carry out of bit position K is produced in step 1, then bit $K - 1$ in the workspace is set to 1, K is increased by 1, and step 1 is repeated if $K \leq 32$.
- c. Step Three - If a carry out of bit position K is not produced in step 1, then the divisor is added to bits K to $K + 31$ in the quotient/remainder workspace, K is increased by 1, and step 1 is repeated if $K \leq 32$.

When the division process is complete, the leftmost 32 bits of the quotient/remainder workspace contain the absolute value of the quotient. The rightmost 32 bits of the workspace contain the absolute value of the remainder. If only one of the A and B operands was complemented, the two's complement of the quotient value is taken to obtain the actual quotient value. Otherwise, the quotient value remains unaltered. If the B operand was complemented, the two's complement of the remainder value is taken to obtain the actual remainder. Otherwise, the remainder value is unaltered.

5.2.4.2 Hexadecimal Floating Point Division

If the A and B operands are tagged hexadecimal floating point quantities, a hexadecimal floating point divide of the A operand into the B operand is performed. The result is a quotient with a maximum length of 13 hexadecimal digits and a remainder. The quotient is stored in the B operand location and the remainder in general register RO, if the secondary result mode indicator is set. Hexadecimal floating point division is performed by the following steps:

- a. Step One - The first step is performed only if the significance mode indicator is off. It consists of pre-normalizing the A and B operands by shifting them left until the high-order hexadecimal digit is non-zero and decreasing the exponent by one for each digit shifted. If the A operand is zero an arithmetic exception trap is generated or is masked.
- b. Step Two - The second step divides the A operand mantissa into the B operand mantissa. If the significance indicator is off, then 14 hexadecimal digits of quotient are generated. If the significance mode indicator is on, then the number of hexadecimal digits generated is equal to one more than the number of significant digits

5.2.4.2
(Cont.)

in the operand which originally contained the smaller number of significant digits. (The number of significant digits in a field is the number of digits in that field less the number of leading zero digits.) The remainder from this division is preserved. The sign of the result is positive if the signs of the A and B operands were the same. Otherwise, the sign is minus. The value of the result exponent is the exponent of the B operand minus the exponent of the A operand. The exponent of the remainder is the dividend exponent minus the number of hexadecimal digits produced in the quotient.

- c. Step Three - The third step depends on the setting of the floating point mode indicators. If the significance mode indicator is not set, the quotient is shifted left until the high order hexadecimal digit is non-zero. The exponent is decreased by one for each digit position shifted. If the exponent underflows an arithmetic exception trap (0401) is generated or is masked.

If the significance mode indicator is set then the quotient is shifted right by a number of hexadecimal digit positions equal to the number of leading zero digits in the operand which originally contained the smaller number of significant digits. The exponent is increased by the number of digit positions shifted. If the exponent overflows an arithmetic exception trap (0401) is generated or is masked.

- d. Step Four - The fourth step is performed only if the significance mode indicator is set. It consists of shifting the quotient right one digit if the most significant digit of the A operand was less than hexadecimal four. (NOTE: The purpose of this step is to ensure that a statistically correct number of significant

digits are maintained during a series of operations.) If the shift takes place the exponent is increased by one. If the last significant digit is lost during this shift an arithmetic exception trap (0402) is generated or is masked.

- e. Step Five - The fifth step is performed only if the floating round mode indicator is set. The step consists of adding a hexadecimal eight to the 14th digit position of the quotient.
- f. Step Six - The sixth step delivers the result. The 13 high-order digits of the quotient together with the associated sign and exponent are used to form a tagged floating point quantity which is delivered to the B operand location. If the secondary result mode indicator is set, the remainder together with its sign and exponent are used to form a tagged floating point quantity which is stored in general register RO.

5.2.4.3 Decimal String Division

If the effective A and B operands are explicit-length specifier structures of the following types:

- a. Zoned decimal string.
- b. Unsigned zoned decimal string.
- c. Packed decimal string.
- d. Unsigned packed decimal string.

then the values of the associated strings are treated as signed or unsigned decimal numbers with stated scale and precision, and a scaled decimal division is performed on these values. The result values computed are a quotient

5.2.4.3
(Cont.)

and a remainder, each with a maximum precision of 32 decimal digits. The quotient value resulting from the division replaces the B operand string value, if it is alterable.

If the DIVIDE instruction is executed in secondary result mode, a decimal string structor must be contained in general register R0. The remainder value resulting from the division is placed in this secondary result string. The high-order and low-order truncation indicators are set if most significant or least significant quotient digits are lost, respectively, when the quotient is restored to the B operand string.

A scaled decimal division is performed by a process of repetitive subtraction. The dividend and the divisor are the A and B operand values as shown in Figure 5-6.

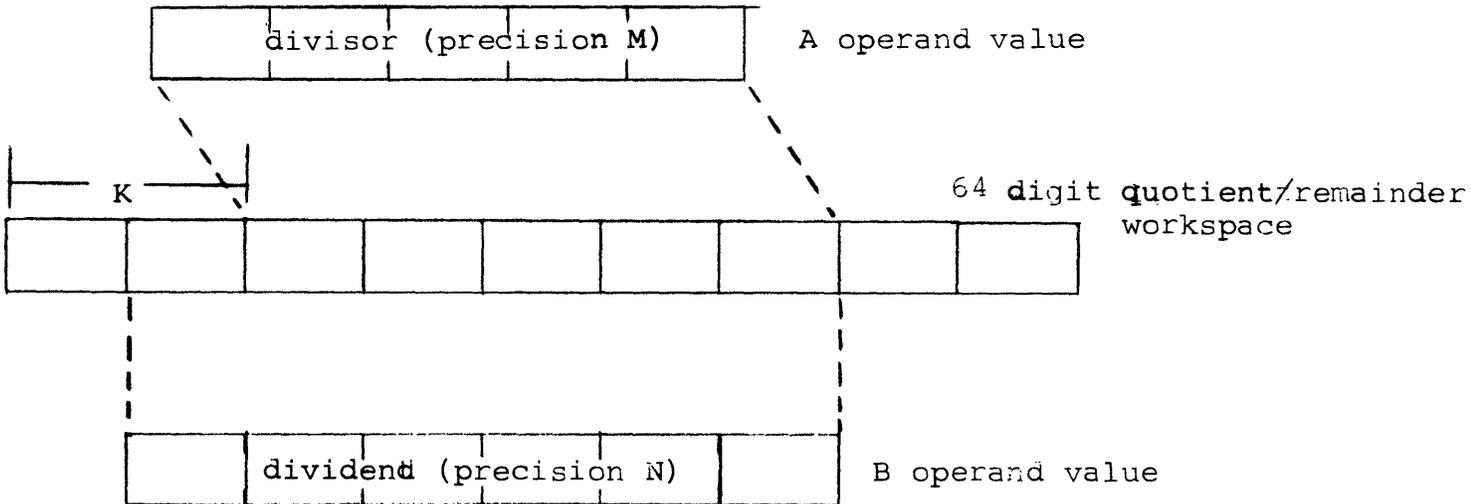


FIGURE 5-6. DIVISION OF SCALED DECIMAL STRINGS

The quotient/remainder workspace is 64 digits in length and is initially set to all zero digits. The leftmost digit in the workspace is digit 0 and the rightmost digit is digit 63. The dividend value is placed in digits 1 to N

5.2.4.3.
(Cont.)

of the workspace, where N is the stated precision of the dividend. The divisor is then tested for a zero value in which case a zero divide arithmetic exception trap (0404) is generated, or masked, depending on the setting of the trap mask bit. Simultaneously, any high-order zero digits in the divisor are detected, and the scale and precision of the divisor are decreased and its location increased appropriately. The resulting quantity is called the modified divisor. The A operand structure is not altered by this action.

The division process is then accomplished by performing the following steps, offsetting the modified divisor by K digits from the left end of the quotient/remainder workspace, for K from 1 to the dividend precision (N).

- a. Step One - The modified divisor is repetitively subtracted from digits K to K + M of the workspace, until the result is negative, which is detected by a borrow from the K - 1st digit position. The quantity M is the precision of the modified divisor. A count is maintained of the number of subtractions performed. The modified divisor is then added once to digits K to K + M of the workspace.
- b. Step Two - The count computed in step 1 is decremented by 1 and placed in digit position K - 1 of the workspace.
- c. Step Three - The workspace offset K is incremented by 1, and steps a and b are repeated if $K \leq N$. At the end of this process, the leftmost N digits of the quotient/remainder workspace contain the quotient digits, and the next M digits contain the remainder digits. The precision of the quotient equals the precision of the B operand (dividend) string, and the precision of the remainder equals the modified divisor precision. The scales of the quotient and the remainder equal the

scale of the B operand (dividend) minus the scale of the modified divisor, plus one. The scale and precision of the quotient are used to properly align and place the quotient value in the B operand string. The scale and precision of the remainder are used to properly align and place the remainder value in the secondary result string, if the division is done in secondary result mode.

The sign code for the quotient is generated from the quotient sign, and the B operand sign class code generated as specified in Appendix A. The remainder sign code is generated from the B operand sign and sign class code. The quotient sign is negative if the A and B operand signs were different. The A and B operand signs and sign class codes are derived from the A and B operand sign codes as specified in Appendix A.

If the quotient sign is negative and the B operand is an unsigned zoned or packed decimal string, the quotient value is not placed in the B operand string. An arithmetic exception error trap (0403) is then generated or masked, depending on the setting of the arithmetic exception trap mask bit. If the remainder sign is negative and the secondary result string is an unsigned zoned or packed decimal string, the remainder value is not placed in the secondary result string. An arithmetic exception error trap (0403) is then generated or masked, depending on the setting of the arithmetic exception trap mask bit.

5.2.5 Compare

Formats: RR, RS, SR, SS, RD.

The effective A and B operands are compared and depending upon the relationship between the operand values, the high, low, or equal condition indicator is set. The effective A and B operands are both derived by applying Autofetch Evaluation to the initial A and B operands. The type of

comparison performed depends on the type of effective operands produced by Autofetch Evaluation.

5.2.5.1 Binary Comparison

If the effective A and B operands are tagged logical words or tagged binary integers in any combination, the value portions of these operands are treated as 32-bit binary integers, and a binary comparison is performed on these values.

A binary comparison is accomplished by performing a binary subtraction of the A operand value from the B operand value (See Subsection 5.2.2.1) and testing, but not restoring the result. The high, low, or equal condition indicator is set, depending on whether the A operand value is less than, greater than, or equal to the B operand value. This corresponds to the result of the subtraction being positive, negative, or zero.

5.2.5.2 Hexadecimal Floating Point Comparison

If the effective A and B operands are both tagged floating point quantities, a hexadecimal floating point comparison is performed on their values.

A hexadecimal floating point comparison is accomplished by performing a hexadecimal floating point subtraction of the A operand value from the B operand value (see Subsection 5.2.2.2), without restoring the result. If the sign of the result is negative, the A operand value is greater than or equal to the B operand value. If the high order result mantissa is zero, the A and B operands are equal. The high, low, or equal condition indicator is set depending on whether the A operand value is less than, greater than, or equal to the B operand value.

5.2.5.3 Decimal String Comparison

If the effective A and B operands are explicit-length specifier structures of the following types:

- a. Zoned decimal string.
- b. Unsigned zoned decimal string.
- c. Packed decimal string.
- d. Unsigned packed decimal string.

then the values of the associated strings are treated as signed or unsigned decimal numbers with stated scale and precision. A scaled decimal comparison is then performed on these values.

A scaled decimal comparison is accomplished by performing a scaled decimal subtraction of the A operand value from the B operand value, without restoring the result (see Subsection 5.2.2.3). If the sign of the result is negative, the A operand value is greater than or equal to the B operand value. If the result is zero, the A and B operands are equal. The high, low, or equal condition indicator is set depending on whether the A operand value is less than, greater than, or equal to the B operand value.

5.2.5.4 Byte String/Translated Byte String Comparison

If the effective A and B operands are explicit-length specifier structures of type byte string or translated byte string (in any combination), the values of the associated strings are treated as bit strings of length eight times the byte length. A byte string/translated byte string comparison is then performed on these values.

The leftmost bytes in the two operand strings are aligned and establish the alignment of succeeding pairs of bytes in the two strings. This alignment establishes the pairing of bits used in performing the comparison. If the two

5.2.5.4
(Cont.)

operands are not of equal length, the shorter operand is extended to the right by appending a sequence of alphanumeric fill bytes from the Task Code Map Description (Doubleword 25) in the Task Status Block (TSB).

If either operand of the instruction is a translated byte string, then bytes from this type of string are translated using the Load Translation Table located by the Task Code Map Description in the TSB. If the Load Translation Table is not present, an operand selection exception trap is generated or masked. The byte values resulting from the translation are used in the comparison in place of the original bytes.

Operation of the comparison also depends on the non-binary collate mode indicator. If the COMPARE instruction is executed in non-binary collate mode, each byte from the A and B operand strings, or its translated value (if from a translated byte string) is translated using the Non-Binary Collating Sequence Translation Table. This is located by the Task Code Map Description in the TSB. If the Non-Binary Collating Sequence Translation Table is not present, an operand selection exception trap (010A) is generated or masked. When the COMPARE instruction is executed in binary collate mode, the above operation is not performed.

The byte values resulting from the above translation processes, if applied, are used in the comparison. The comparison is performed by considering aligned bytes from left-to-right in the two operands. Each byte is treated as an unsigned binary integer. If the two bytes are equal, then the next byte in succession is considered. If all bytes are equal, then the operands are equal. If the two bytes are not equal, the one with the larger binary value is associated with the operand of greater value. The high, low, or equal condition indicator is set depending on whether the A operand value is less than, greater than, or equal to the B operand value.

5.2.6 Move

Formats: RR, RS, SR, SS, RD.

The effective A operand value is assigned to the B operand location. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. Autostore Evaluation is applied to the initial B operand of the instruction and must produce an operand description as an effective B operand. The type of move performed depends on the type of effective operands produced by Autofetch and Autostore Evaluation.

5.2.6.1 Byte String/Translated Byte String Move

If the effective A and B operands are explicit-length specifier structures of type byte string or translated byte string (in any combination), then a byte string/translated byte string move is performed.

The move is performed by aligning the leftmost bytes in the two operand strings, together with succeeding pairs of bytes in the two strings. If the destination string is shorter than the source string, the source string is truncated on the right. With the destination string longer than the source string, the source string is extended to the right with the alphanumeric fill byte in the Task Code Map Description in the TSB.

If the A operand is a translated byte string, then bytes from this string are translated using the Load Translation Table located by the Task Code Map Description. If a Load Translation Table is not present, an Autotranslation error trap is generated or masked. The byte values resulting from the translation are then used in the move operation in place of the original bytes.

If the B operand is a translated byte string, then bytes to be placed in this string are translated using the Store

Translation Table located by the Task Code Map Description. If a Store Translation Table is not present, an operand selection exception trap (010A) is generated or masked. The byte values resulting from the translation are stored in the destination string in place of the original bytes. The move operation is performed by transmitting aligned bytes from the source to the destination string, including the above translation processes if appropriate and the alphanumeric fill if necessary.

5.2.6.2 Decimal String to Decimal String Move

If the effective A and B operands are explicit-length specifier structures of the following types:

- a. Zoned decimal string.
- b. Unsigned zoned decimal string.
- c. Packed decimal string.
- d. Unsigned packed decimal string.

the value of the A operand string, treated as a signed or unsigned decimal number with stated scale and length, is assigned to the B operand string. Consideration is given to its sign, scale, and length. The value is assigned only if the B operand string is alterable. Otherwise, an operand selection exception trap is generated or masked. A scaled decimal move is performed as a zero and add operation. In particular, the rules for decimal string addition as specified in Subsection 5.2.1.3 are followed with the exceptions noted below. The primary exception is that every digit in the B operand string is assumed to be a decimal zero initially, and the sign of the B operand string is always assumed to be positive. The high- and low-order truncation condition indicators are set, zones are set using the zone part of the numeric fill byte, in the TSB, and signs are generated as discussed in Subsection 5.2.1.3. The high, low, or equal

condition indicator is set depending on whether the value moved is positive, negative, or zero, respectively.

5.2.6.3 Autostore Moves

If the effective B operand is an implicit-length specifier structor or an explicit-length specifier structor of the following types:

- a. Bit String.
- b. Binary String.
- c. Hexadecimal floating point string.

The move is performed by Autostore Conversion.

A move by Autostore Conversion is accomplished by using the effective B operand as the operand description, and the effective A operand as the Autostore input for the appropriate type of Autostore Conversion operation. An Autostore Conversion error operand selection trap may occur while performing the Autostore Conversion operation, as specified in Subsection 3.2.

5.2.7 And

Formats: RR, RS, SR, SS, RD.

The logical product of the effective A and B operands is computed bit-by-bit, and the result replaces the B operand. The effective A and B operands are derived by applying Autofetch Evaluation to both the initial A and B operands. The operand description from B operand Autofetch application is used to replace the result. The operation performed is dependent on the type of effective operands produced by Autofetch Evaluation. The high or equal condition indicator is set depending on the result of the AND instruction being non-zero or zero, respectively.

5.2.7.1 Logical Word AND

If the effective A and B operands are both tagged logical words, the value portions of these operands are treated as 32-bit logical quantities, and the logical connective AND is applied to these logical values. The result value generated is used to form a tagged logical word of equal value. This tagged logical word is then restored to the effective B operand location.

The AND operation is performed by taking bits in identical positions in the two operand values and computing a result bit of identical position according to the following table:

A operand bit	0	0	1	1
B operand bit	0	1	0	1
Result bit	0	0	0	1

5.2.7.2 Byte String AND

If the effective A and B operands are explicit-length specifier structures of type byte string, the values of the associated strings are treated as bit strings of length eight times the byte length. The logical operation AND is then performed on these logical values. The result value generated replaces the B operand string, if it is alterable.

The leftmost bytes in the two operand strings are aligned and establish the alignment of succeeding pairs of bytes in the two strings. This alignment establishes the pairing of bits used in performing the logical operation. The AND operation performed on these bits is described in Subsection 5.2.7.1. If the A operand is shorter than the B operand, the A operand is extended on the right to the length of the B operand, filling with zero bytes. If the A operand is longer than the B operand, it is truncated to the length of the B operand.

If the B operand string is not alterable, an alterability operand selection exception trap (0103) is generated or masked, depending on the setting of the operand selection trap mask bit.

5.2.8 Or

Formats: RR, RS, SR, SS, RD.

The logical sum of the effective A and B operands is computed bit-by-bit, and the result replaces the B operand. The effective A and B operands are derived by applying Autofetch Evaluation to both the initial A and B operands. The operand description from B operand Autofetch application is used to replace the result. The operation performed is dependent on the type of effective operands produced by Autofetch Evaluation. The high or equal condition indicator is set depending on the result of the OR instruction being non-zero or zero, respectively.

5.2.8.1 Logical Word OR

If the effective A and B operands are both tagged logical words, the value portions of these operands are treated as 32-bit logical quantities, and the logical connective OR is applied to these logical values. The result value generated is used to form a tagged logical word of equal value. This tagged logical word is then restored to the effective B operand location.

The OR operation is performed by taking bits in identical positions in the two operand values and computing a result bit of identical position according to the following table:

A operand bit	0	0	1	1
B operand bit	0	1	0	1
Result bit	0	1	1	1

5.2.8.2 Byte String OR

If the effective A and B operands are explicit-length specifier structures of type byte string, the values of the associated strings are treated as bit strings of length eight times the byte length. The logical operation OR is then performed on these logical values. The result value generated replaces the B operand string, if it is alterable.

The leftmost bytes in the two operand strings are aligned and establish the alignment of succeeding pairs of bytes in the two strings. This alignment establishes the pairing of bits used in performing the logical operation. The OR operation performed on these bits is described in Subsection 5.2.8.1. If the A operand is shorter than the B operand, the A operand is extended on the right to the length of the B operand, filling with zero bytes. If the A operand is longer than the B operand, it is truncated to the length of the B operand.

If the B operand string is not alterable, an alterability operand selection exception trap (0103) is generated or masked, depending on the setting of the operand selection trap mask bit.

5.2.9 Exclusive OR

Formats: RR, RS, SR, SS, RD

The modulo-2 sum of the effective A and B operands is computed bit-by-bit, and the result replaces the B operand. The effective A and B operands are derived by applying Autofetch Evaluation to both the initial A and B operands. The operand description from B operand Autofetch application is used to replace the result. The operation performed is dependent on the type of effective operands produced by Autofetch Evaluation. The high or equal condition indicator is set, depending on the result of the EXCLUSIVE OR instruction being non-zero or zero, respectively.

5.2.9.1 Logical Word EXCLUSIVE OR

If the effective A and B operands are both tagged logical words, the value portions of these operands are treated as 32-bit logical quantities, and the logical connective EXCLUSIVE OR is applied to these logical values. The result value generated is used to form a tagged logical word of equal value. The tagged logical word is then restored to the effective B operand location.

The EXCLUSIVE OR operation is performed by taking bits in identical positions in the two operand values and computing a result bit of identical position according to the following table:

A operand bit	0	0	1	1
B operand bit	0	1	0	1
Result bit	0	1	1	0

5.2.9.2 Byte String EXCLUSIVE OR

If the effective A and B operands are explicit-length specifier structures of type byte string, the values of the associated strings are treated as bit strings of length eight times the byte length. The logical operation EXCLUSIVE OR is then performed on these logical values. The result value generated replaces the B operand string, if it is alterable.

The leftmost bytes in the two operand strings are aligned and establish the alignment of succeeding pairs of bytes in the two strings. This alignment establishes the pairing of bits used in performing the logical operation. The EXCLUSIVE OR operation performed on these bits is described in Subsection 5.2.9.1. If the A operand is shorter than the B operand, the A operand is extended on the right to the length of the B operand, filling with zero bytes. If the A operand is longer than the B operand, it is truncated to the length of the B operand.

If the B operand string is not alterable, an alterability operand selection exception trap (0103) is generated or masked, depending on the setting of the operand selection trap mask bit.

5.2.10 Shift

Formats: RR, SR

The contents of the general register specified by the B operand field of the instruction and, optionally the contents of general register R0 are shifted a number of bit positions specified by the effective A operand. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The effective A operand must be a tagged binary integer or tagged logical word. If it is not a tagged binary integer or tagged logical word, an illegal operand trap (0200) is generated. The sign (left-most bit) of the tagged binary integer is used to determine the direction of shift, with positive values associated with left shifts and negative values associated with right shifts. If the effective A operand is a tagged logical word, a left shift is performed. The contents of the general register specified by the B operand field of this instruction must be a tagged binary integer or a tagged logical word. If general register R0 participates in the shift, it must contain the same type of data representation. The type of shift performed depends on the data representation of the B operand value. If the B operand value is a tagged logical word, a logical shift is performed. If the B operand value is a tagged binary integer, an arithmetic shift is performed. After shifting, the result is restored to the general register specified by the B operand and, optionally, to general register R0.

5.2.10.1 Logical Shift

If the effective B operand is a tagged logical word, a logical shift is performed. When the SHIFT instruction is

executed in secondary result mode, general register RO must also contain a tagged logical word, the value of which is concatenated to the left of the effective B operand value. The resulting value to be shifted is either 32 bits (single precision) or 64 bits (double precision) in length. If the A operand value is positive, a logical left shift is performed; otherwise a logical right shift is performed. The number of bits shifted is equal to the magnitude of the A operand value. The leftmost (rightmost) bits shifted are truncated, and the rightmost (leftmost) bit positions are filled with zeros.

If a single precision shift is performed, the shifted value is restored to the B operand location. When a double precision shift is performed, the leftmost 32 bits of the shifted value are restored to general register RO, followed by restoration of the rightmost 32 bits to the effective B operand location. The shifted value is restored in the form of tagged logical words.

NOTE: If general register RO is specified as the effective B operand location, and the SHIFT instruction is executed in secondary result mode, the result is identical to a single precision rotational shift of the tagged logical word in general register RO.

5.2.10.2 Arithmetic Shift

If the effective B operand is a tagged binary integer, an arithmetic shift is performed. When the SHIFT instruction is executed in secondary result mode, general register RO must also contain a tagged binary integer, the value of which is concatenated to the left of the effective B operand value. The resulting value to be shifted is either a 32-bit two's complement integer (single precision) or a 64 bit two's complement integer (double precision). If the A operand value is positive, an arithmetic left shift is performed; otherwise, an arithmetic right shift is performed. The number of bits positions shifted is equal to the magnitude of the A operand value. In arithmetic left shifts, the rightmost 31

or 63 bits are shifted, preserving the leftmost (sign) bit. The leftmost bits shifted are truncated and the rightmost bits are filled with zeros. In arithmetic right shifts, all 32 or 64 bits are shifted, with the rightmost bits shifted being truncated and the leftmost bit positions being filled with the leftmost (sign) bit of the initial value.

If a single precision shift is performed, the shifted value is restored to the B operand location. When a double precision shift is performed, the leftmost 32-bits of the shifted value are restored to general register R0, followed by restoration of the rightmost 32-bits to the effective B operand location. The shifted value is restored in the form of tagged binary integers.

5.2.11 Load Positive

Formats: RR, SR

The absolute value of the effective A operand value is placed in the general register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The operation is performed in a fashion dependent upon the type of effective A operand produced by Autofetch Evaluation. The high, low, or equal condition indicator is set depending on the value of the result being positive, negative, or zero.

5.2.11.1 Two's Complement Binary Loading

If the effective A operand is a tagged binary integer, its absolute value is placed in the general register specified by the B operand field of the instruction.

The absolute value of a two's complement binary integer is obtained by examination of the leftmost (sign) bit of its value. If this bit is 1, the value is negative, and the two's complement of the value is taken. Otherwise, the value remains unaltered. If the two's complement of the maximum negative number is taken, the binary overflow

indicator is set. Depending on the setting of the arithmetic exception trap mask bit, an arithmetic exception trap (0400) occurs or is masked.

5.2.11.2 Hexadecimal Floating Point Loading

If the effective A operand is a tagged hexadecimal floating point number, its absolute value is placed in the general register specified by the B operand field of the instruction.

The absolute value of a hexadecimal floating point number is obtained by setting its sign to plus.

5.2.12. Load Negative

Formats: RR, SR.

The negative (complement) of the absolute value of the effective A operand value is placed in the general register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The operation is performed in a fashion dependent upon the type of effective A operand produced by Autofetch Evaluation. The high, low, or equal condition indicator is set depending on the value of the result being positive, negative or zero.

5.2.12.1 Twos Complement Binary Loading

If the effective A operand is a tagged binary integer, the negative of its absolute value is placed in the general register specified by the B operand field of the instruction.

The negative of the absolute value of a twos complement binary integer is obtained by examination of the leftmost (sign) bit of its value. If this bit is zero, the value is positive, and the two's complement of the value is taken. Otherwise, the value remains unaltered.

5.2.12.2 Hexadecimal Floating Point Loading

If the effective A operand is a tagged hexadecimal floating point number, the negative of its absolute value is placed in the general register specified by the B operand field of the instruction.

The negative of the absolute value of a hexadecimal floating point number is obtained by setting its sign to minus.

5.2.13 Load Complement

Formats: RR, SR.

The negation of the effective A operand value is placed in the general register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial.

A operand produced by Autofetch Evaluation. The high, low or equal condition indicator is set depending on the result of the operation being positive, negative, or zero.

5.2.13.1 Logical Binary Negation

If the effective A operand is a tagged logical word, its negation is placed in the general register specified by the B operand field of the instruction.

The negation of a logical binary value is obtained by taking the one's complement of the value, that is, by inverting every bit in the value.

5.2.13.2 Twos Complement Binary Negation

If the effective A operand is a tagged binary integer, its negation is placed in the general register specified by the B operand field of the instruction.

The negation of a twos complement binary value is obtained by taking the two's complement of the value. If the two's

complement of the maximum negative value is taken, the overflow indicator is set. Depending on the setting of the arithmetic exception mask bit, an arithmetic exception trap (0400) occurs or is masked.

5.2.13.3 Hexadecimal Floating Point Negation

If the effective A operand is a tagged hexadecimal floating point number, its negation is placed in the general register specified by the B operand field of the instruction. The negation of a hexadecimal floating point number is obtained by inverting its sign bit.

5.2.14 Load and Test

Formats: RR, SR.

The effective A operand value is placed in the general register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The high, low, or equal condition indicator is set depending on the effective A operand value being positive, negative or zero. The test for sign and magnitude is dependent on the type of effective A operand produced by Autofetch Evaluation.

5.2.14.1 Logical Binary Testing

If the effective A operand is a tagged logical word, the value is zero if all bits are zero, and positive otherwise.

5.2.14.2 Twos Complement Binary Testing

If the effective A operand is a tagged binary integer, the value is zero if all bits are zero, negative if the left-most bit is one, and positive otherwise.

5.2.14.3 Hexadecimal Floating Point Testing

If the effective A operand is a tagged hexadecimal floating point number, the value is zero if the mantissa is zero,

positive if the sign is plus and the mantissa non-zero, and negative if the sign is minus and the mantissa non-zero.

5.2.15 CONVERT TO LOGICAL
Formats: RR, SR

The effective A operand value is converted to a tagged logical word and placed in the general register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand.

The following data types are legal effective A operands:

- a. Tagged Logical Word
- b. Tagged Binary Integer
- c. Tagged Hexadecimal Floating Point Number
- d. Zoned Decimal String
- e. Unsigned Zoned Decimal String
- f. Packed Decimal String
- g. Unsigned Packed Decimal String

An illegal operand trap (0200) is generated or masked if the effective A operand is any other type.

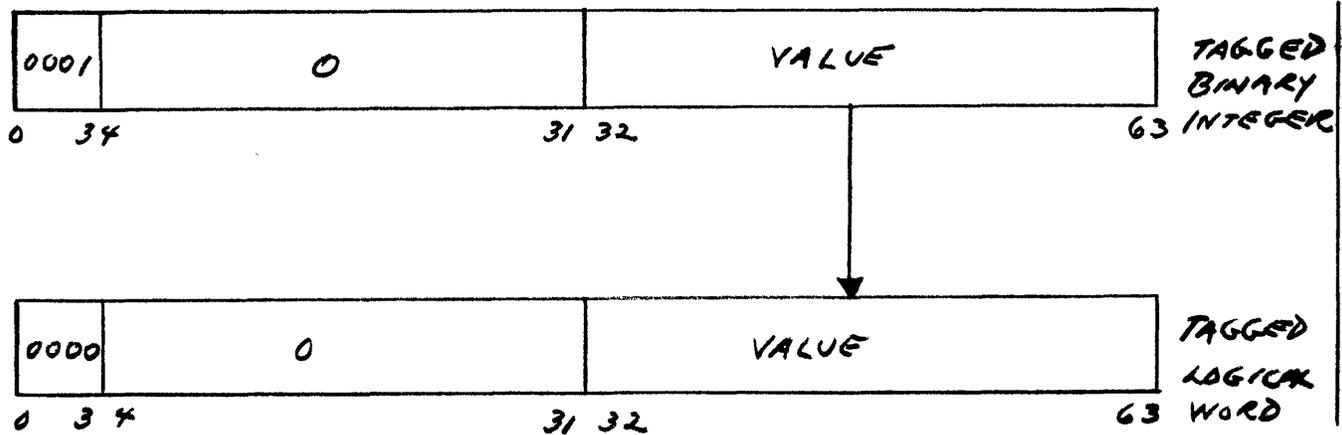
The process of conversion is defined below.

5.2.15.1 Tagged Logical Word to Tagged Logical Word Conversion

No change takes place.

5.2.15.2 Tagged Binary Integer to Tagged Logical Word Conversion

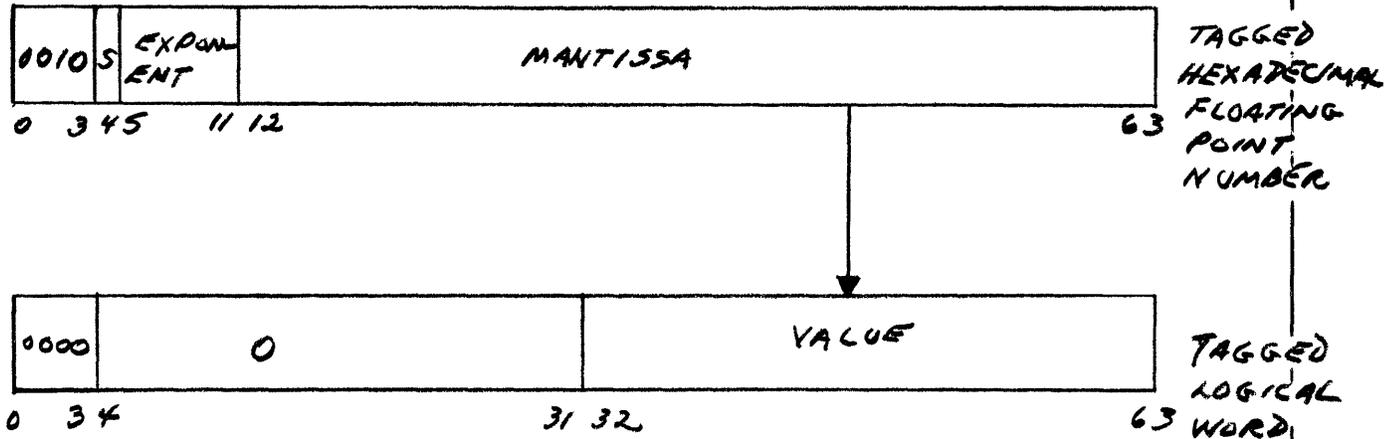
The sign bit (bit 32) of the tagged binary integer is tested. If it is one the binary integer is negative and can not be represented by a tagged logical word. An arithmetic exception trap (0403) is generated or masked. If the sign bit is zero a tagged logical word is assembled by setting the TAG field (bits 0-3) to zero (indicating a tagged logical word), setting the reserved field (bits 4-31) to zero, and by setting the VALUE field (bits 32-63) equal to bits 32-63 of the tagged binary integer as shown in the following figure.



5.2.15.3 Tagged Hexadecimal Floating Point Number to Tagged Logical Word Conversion

The sign bit (bit 4) of the floating point number is tested. If it is one, the floating point number is negative and can not be represented by a tagged logical word. An arithmetic exception trap (0403) is generated or masked. If the sign bit is zero the mantissa of the floating point number is shifted left or right one hexadecimal digit at a time increasing or decreasing the exponent by one for each right or left shift until the exponent is equal to 1001101 (implied radix point is to the right of bit 63). If any ones are shifted out of the left end of the mantissa, or if any of bit positions 12-31 of the mantissa contain ones after the shift, the floating point number cannot be represented as a tagged logical word and an arithmetic exception trap (0400) is generated or masked. If any ones are shifted out of the right end of the mantissa the number contains a fractional part and the floating point round mode indicator (bit 44 of the current procedure index) is examined. A one indicates that the mantissa should be rounded off and the last bit shifted out of the right end is added to the mantissa. In this case bit 31 must again be examined for a possible arithmetic exception trap. If the round mode

indicator is zero the bits shifted out the right end are dropped. The tagged logical word is then assembled by setting the TAG field (bits 0-3) to zero (indicating a tagged logical word), setting the reserved field (bits 4-31) to zero, and by setting the VALUE field (bits 32-63) equal to bits 32-63 of the shifted mantissa of the floating point number as shown in the following figure.



5.2.15.4 Decimal String to Tagged Logical Word Conversion

The following steps describe the process of converting a decimal string to a tagged logical word:

- a. The sign is examined. If it is negative an arithmetic exception trap (0403) is generated or masked.
- b. The scale factor is examined to determine the position of the implied decimal point. If there are any digits to the right of the decimal point the number contains a fractional part and the decimal round mode indicator (bit 45 of the current procedure index) is examined. If it is set the decimal quantity 0.5 is added to the decimal string to round off the integral portion. If the round mode indicator is not set the low-order truncation indicator (bit 59 of the current procedure index) is set.

- c. The value of the integer portion of the decimal string is converted to a positive binary integer. If the binary integer is more than 32 bits long an arithmetic exception trap (0400) is generated or masked.
- d. The tagged logical word is then assembled by setting the TAG field (bits 0-3) to zero (indicating a tagged logical word), setting the reserved field (bits 4-31) to zero, and by placing the binary result in the VALUE field (bits 32-63).

5.2.16 CONVERT TO BINARY

Formats: RR, SR

The effective A operand value is converted to a tagged binary integer and placed in the general register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand.

The following data types are legal effective A operands:

- a. Tagged Logical Word
- b. Tagged Binary Integer
- c. Tagged Hexadecimal Floating Point Number
- d. Zoned Decimal String
- e. Unsigned Zoned Decimal String
- f. Packed Decimal String
- g. Unsigned Packed Decimal String

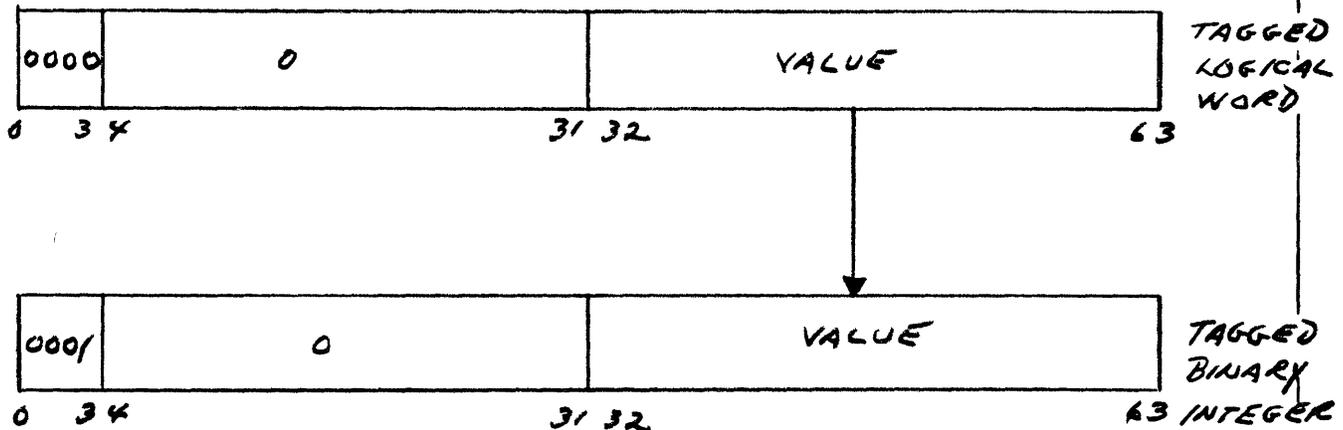
An illegal operand trap (0200) is generated or masked if the effective A operand is any other type.

The process of conversion is defined below.

5.2.16.1 Tagged Logical Word to Tagged Binary Integer Conversion

Bit 32 of the source word is tested. If it is one then the logical word is greater than or equal to 2^{31} and cannot be represented as a tagged binary integer. In this case an

arithmetic exception trap (0400) is generated or masked. If bit 32 is zero a binary integer is assembled by setting the TAG field (bits 0-3) to 0001 (indicating a tagged binary integer), setting the reserved field (bits 4-31) to zero, and by setting the VALUE field (bits 32-63) equal to bits 32-63 of the tagged logical word as shown in the following figure.



5.2.16.2 Tagged Binary Integer to Tagged Binary Integer Conversion

No change takes place.

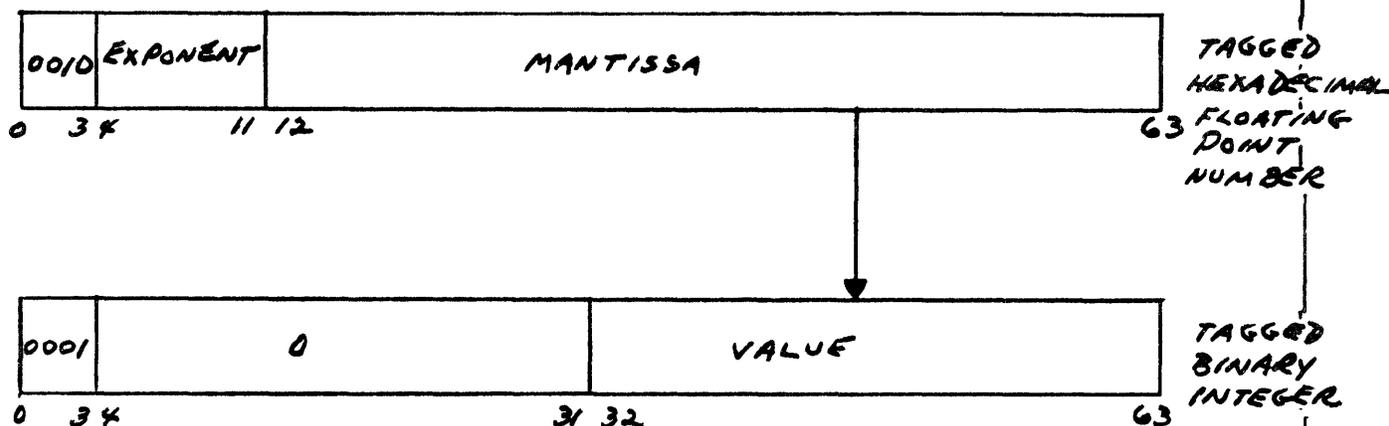
5.2.16.3 Tagged Hexadecimal Floating Point Number to Tagged Binary Integer Conversion

The mantissa of the floating point number is shifted left or right one hexadecimal digit at a time increasing or decreasing the exponent by one for each right or left shift until the exponent is equal to 1001101 (implied radix point is to the right of bit 63). If any ones are shifted out of the left end of the mantissa, or if any of bit positions 12-31 of the mantissa contain ones after the shift, the floating point number cannot be represented as a tagged binary integer and an arithmetic exception trap (0400) is generated or masked. If any ones are shifted out of the right end of the mantissa

the number contains a fractional part and the floating point round mode indicator (bit 44 of the current procedure index) is examined. A one indicates that the mantissa should be rounded off and the last bit shifted out of the right end is added to the mantissa. If the round mode indicator is zero the bits shifted out the right end are dropped.

The sign bit (bit 4) is then examined. If it is negative the mantissa is complemented and a one is added to the low-order position. Bits 12-32 should then contain all ones if the number is negative and zeros if it is positive. If not, an arithmetic exception trap (0400) is generated or masked.

The tagged binary integer is then assembled by setting the TAG field (bits 0-3) to 0001 (indicating a tagged binary integer), setting the reserved field (bits 4-31) to zero, and by setting the VALUE field (bits 32-63) to bits 32-63 of the mantissa value calculated above as shown in the following figure.



5.2.16.4 Decimal String to Tagged Binary Integer Conversion

The following steps describe the process of converting a decimal string to a tagged binary integer:

- a. The scale factor is examined to determine the position of the implied decimal point. If there are any digits to the right of the decimal point the number contains a fractional part and the decimal round mode indicator (bit 45 of the current procedure index) is examined. If it is set the decimal quantity 0.5 is added to the magnitude of the decimal string to round off the integral portion. If the round mode indicator is not set the low-order truncation indicator (bit 59 of the current procedure index) is set.
- b. The value of the integer portion of the decimal string is converted to a binary integer. The sign of the decimal string is examined. If it is negative the binary integer is complemented and a one is added to the low-order position.

The rightmost 31 bits of the binary integer are saved for the result. Every bit to the left of these 31 bits should be zero for positive numbers and one for negative numbers. If not, the magnitude of the number is too great to be represented as a tagged binary integer and an arithmetic exception trap (0400) is generated or masked.

- c. The tagged binary integer is assembled by setting the TAG field (bits 0-3) to 0001 (indicating a tagged binary integer), setting the reserved field (bits 4-31) to zero, setting the sign bit (bit 32 one if negative) and placing the above result integer in bits 33-63.

5.2.17 CONVERT TO FLOATING

Formats: RR, SR

The effective A operand value is converted to a tagged hexadecimal floating point number and placed in the general

register specified by the B operand field of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand.

The following data types are legal effective A operands:

- a. Tagged Logical Word
- b. Tagged Binary Integer
- c. Tagged Hexadecimal Floating Point Number

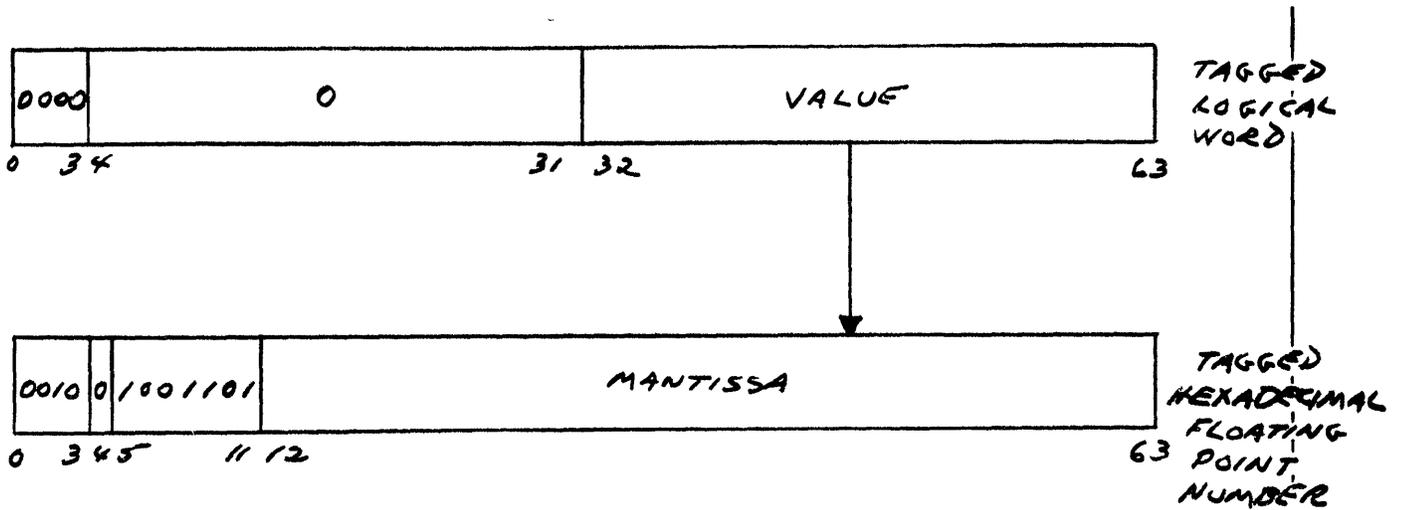
An illegal operand trap (0200) is generated or masked if the effective A operand is any other type.

The process of conversion is defined below:

5.2.17.1 Tagged Logical Word to Tagged Hexadecimal Floating Point Number Conversion

A tagged hexadecimal floating point number is assembled by setting the TAG field (bits 0-3) to 0010 (indicating a floating point number), setting the sign bit (bit 4) to zero (positive), setting the exponent (bits 5-11) to 1001101 (indicating radix point to the right of bit 63), setting bits 12-31 to zero, and setting bits 32-63 equal to bits 32-63 of the tagged logical word.

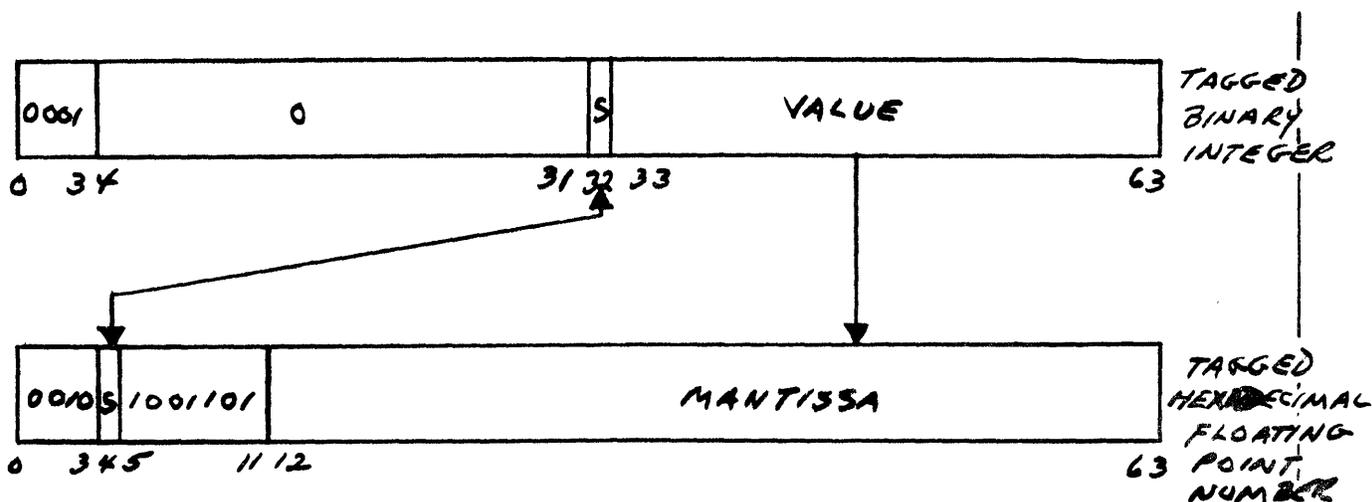
The significance mode indicator (bit 46 of the current procedure index) is examined. If it is ONE the conversion is complete. If it is zero the mantissa of the floating point number (bits 12-63) is shifted left one hexadecimal digit at a time until the high-order hexadecimal digit is nonzero. The exponent (bits 5-11) is decremented by one for each hexadecimal position shifted. Once this shift is performed the conversion is complete as shown in the following figure.



5.2.17.2 Tagged Binary Integer to Tagged Hexadecimal Floating Point Number Conversion

A tagged floating point number is assembled by setting the TAG field (bits 0-3) to 0010 (indicating a floating point number), setting the sign bit (bit 4) to the sign bit of the binary integer (bit 32), setting the exponent (bits 5-11) to 1001101 (indicating radix point to the right of bit 63), setting bits 12-32 all equal to the sign bit, and setting bits 33-63 equal to bits 33-63 of the tagged binary integer. If the sign bit is one (negative) the mantissa of the floating point number (bits 12-63) is complemented and one is added to the low-order position.

The significance mode indicator (bit 46 of the current procedure index) is examined. If it is ONE the conversion is complete. If it is zero the mantissa of the floating point number (bits 12-63) is shifted left one hexadecimal digit at a time until the high-order hexadecimal digit is nonzero. The exponent (bits 5-11) is decremented by one for each hexadecimal position shifted. Once this shift is performed the conversion is complete as shown in the following figure.



5.2.17.3 Tagged Hexadecimal Floating Point Number to Tagged Hexadecimal Floating Point Number Conversion

No change takes place.

5.2.18 CONVERT TO DECIMAL

Formats: RR, RS

The value in the general register specified by the A operand field is converted to a decimal string and stored in the effective B operand location. The effective B operand location is derived by applying Autostore Evaluation to the initial B operand.

The following data types are legal A operands:

- a. Tagged Logical Word
- b. Tagged Binary Integer

An illegal operand trap (0200) is generated or masked if the effective A operand is any other type.

The process of conversion is defined below.

5.2.18.1 Tagged Logical Word to Decimal String Conversion

The binary value contained in bits 32-63 of the logical word is converted to a decimal integer. The POSITION field of the decimal string structor is examined to determine the length and scale factor of the decimal field (see 2.4.1.3

through 2.4.1.6 for decimal field formats). The decimal integer obtained above and the destination field are aligned by decimal point. Zeros are added where necessary to fill the destination field.

If truncation will occur on the high-order end of the integer an arithmetic exception trap (0401) is generated or masked.

If truncation will occur on the low-order end the decimal round mode indicator (bit 45 of the current procedure index) is examined. If the round mode indicator is set the decimal number is rounded off to fit the field. The digit immediately to the right of the least significant digit position of the destination field is examined. A value of decimal five or more causes the value one to be added to the decimal number in the digit position corresponding to the least significant digit of the destination field. If the round mode indicator is not set the decimal number is truncated and the low-order truncation indicator (bit 59 of the current procedure index) is set.

The decimal number is then placed in the destination field in the appropriate format.

5.2.18.1 Tagged Binary Integer to Decimal String Conversion

The sign bit (bit 32) of the tagged binary integer is examined. If it is negative bits 32-63 are complemented and incremented by one (two's complement is taken) to determine the magnitude. The magnitude of the binary integer is then converted to a decimal integer.

The POSITION field of the decimal string structor is examined to determine the length and scale factor of the decimal destination field (see Paragraphs 2.4.1.3 through 2.4.1.6 for decimal field formats). The decimal integer obtained above and the destination field are aligned by decimal point. Zeros are added where necessary to fill the destination field.

If truncation will occur on the high-order end of the integer an arithmetic exception trap (0400) is generated or masked.

If truncation will occur on the low-order end the decimal round mode indicator (bit 45 of the current procedure index) is examined. If the round mode indicator is set the decimal number is rounded off to fit the field. The digit immediately to the right of the least significant digit position of the destination field is examined. A value of decimal five or more causes the value 1 to be added to the decimal number in the digit position corresponding to the least significant digit of the destination field. If the round mode indicator is not set the decimal number is truncated and the low-order truncation indicator (bit 59 of the current procedure index) is set.

The decimal number is then placed in the destination field in the appropriate format. The sign of the binary integer is placed in the sign field.

5.3 GENERAL REGISTER LOADING/STORING INSTRUCTIONS

The general register loading/storing instructions are used to transmit tagged quantities to and from the general registers.

The condition indicators are not altered by execution of these instructions.

5.3.1 Copy

Formats: RR, SR

A copy of the initial A operand is placed in the general register specified by the B operand syllable. In the RR format, this operation provides a facility for replicating the contents of one general register in another. In the SR format, it provides a facility for generating a structor from S-Syllable Extraction (See subsection 4.3) and placing it in a selected general register.

5.3.2 Fetch

Formats: RR, SR, RD.

The effective A operand is placed in the general register specified by the B operand syllable of the instruction. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The tagged quantity resulting from Autofetch Evaluation is then placed in the B operand general register.

5.3.3 Load

Formats: RR, SR, RD.

The A operand value is placed in the general register specified by the B operand syllable of the instruction.

The A operand value is derived by applying a special form of Autofetch Evaluation that terminates when the indirection count is equal to one. (See subsection 4.3). In particular, step c. of the description of Autofetch Evaluation is replaced by the following step:

Increment the indirection count by one. If it is equal to one, Autofetch Evaluation is terminated.

The tagged quantity resulting from this special form of Autofetch Evaluation is then placed in the B operand general register.

The LOAD instruction provides a facility for placing an item in an implicit-length array in a general register without interpretation of the item and without further indirection.

5.3.4 Deposit

Formats: RR, RS.

The tagged quantity in the general register specified by the A operand syllable is placed in the location specified by the effective B operand. Autostore Evaluation is applied using the initial B operand as an operand description and the initial A operand as Autostore Evaluation input.

In RR format, the B operand general register is a valid destination. In either format, the only other valid destination is a bit, binary, or floating point string or implicit-length array.

5.3.5 Store

Formats: RR, RS.

The tagged quantity in the general register specified by the A operand syllable of the instruction is placed in the B operand location. The B operand location is derived by applying a special form of Autostore Evaluation that

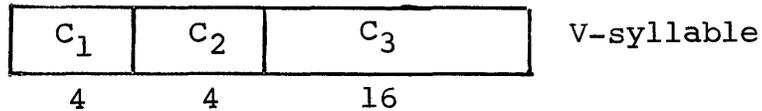
terminates when the indirection count is one. (See subsection 4.3). In particular, step e. of the description of Autostore Evaluation is replaced by the following step:

If the indirection count is nonzero, and if the current tagged value is produced by action c. in Table 3-2, then Autostore Evaluation terminates.

The initial B operand is used as operand description and the initial A operand is used as Autostore input for this special form of Autostore Evaluation.

5.3.6 Dump Multiple

Formats: CV.



The general purpose registers specified by the third control field are stored into the array specified by the first control field, which is interpreted as a general register address. The registers to be stored correspond to the bits in the third control field that are set to one, where the leftmost bit corresponds to register R0 and the rightmost bit to register RF. The second control field contains the number of bits set to one in the third control field, minus one. Register RA must contain an implicit-length structor of type tagged doubleword, LIFO array, or FIFO array.

If the structor identifies a tagged doubleword array, the first n elements of the array are used to store the corresponding general registers, where n is the number of registers to be stored.

If the structor describes a LIFO array, the position field of the structor is used to determine the location into which the first register is stored. The value of the position field, times eight, added to the value of the location field of the structor, is the desired location. Other registers are stored into succeeding higher number doublewords of storage. The position field is then incremented by n.

If the structor describes a FIFO array, the leftmost 8 bits of the position field, called the FIFO tail, are used to determine the storage address into which the first register is stored. The value of this field, times eight, added to the value of the location field of the structor is the required storage address. Other registers are stored into succeeding higher numbered doublewords of storage, until the last element in the FIFO array is used, at which point doublewords are used starting at the FIFO array origin. The FIFO tail field is incremented by n (if the extent is not exceeded) or is set to n minus the difference between the extent and FIFO tail field values (if the extent is exceeded).

For tagged doubleword arrays, the number of registers to be stored must be less than the array extent. For LIFO arrays, the difference between the extent and position field values must be greater than the number of registers to be stored. For FIFO arrays, the interval defined by the incremented value of the FIFO tail field and its old value must not include the value of the FIFO head field (rightmost 8 bits of position field).

Registers are stored into the array in the order of low-to-high numbered registers.

An operand selection exception (0107) occurs if insufficient array elements are available to execute the DUMP MULTIPLE instruction and the operand selection trap mask bit is not set.

5.3.7

Dump

Formats: RR

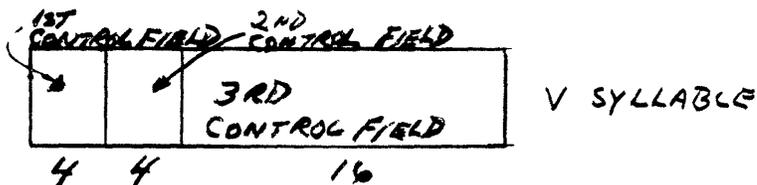
The general register specified by the A operand syllable of the instruction is stored into the array specified by a structor in the general register identified by the B operand syllable of the instruction. The action of this instruction is identical to executing a DUMP MULTIPLE

instruction with a one bit in the bit position of the third control field corresponding to the general register specified by the A operand syllable of the DUMP instruction and with a first control field identical to the B operand syllable of the DUMP instruction.

5.3.8

Undump Multiple

Formats: CV.



The general purpose registers specified by the third control field are loaded from the array specified by the first control field, which is interpreted as a general register address. The registers to be loaded correspond to the bits in the third control field of the instruction that are set to one, where the leftmost bit corresponds to register R0 and the rightmost bit to register RF. The second control field contains the number of bits set to one in the third control field, minus one. Register Ra must contain an implicit-length structor of type tagged doubleword, LIFO array, or FIFO array.

If the structor identifies a tagged doubleword array, the first n elements of the array are placed in the corresponding registers, where n is the number of registers to be loaded.

If the structor describes a LIFO array, the position field of the structor is used to select the first element to be loaded. The value of the position field, decremented by one, the quantity multiplied by eight, added to the value of the location field of the structor is the storage address of the first element. Other registers are loaded from succeeding lower numbered doublewords of storage. The position field is then decremented by n.

If the structor describes a FIFO array, the rightmost 8 bits of the position field, called the FIFO head, are used to obtain the first element to be loaded. The value of this field, times eight, added to the value of the location field of the structor is the storage address of this first element. Other registers are loaded from succeeding higher numbered doublewords of storage, until the last element in the FIFO array is fetched, at which point doublewords are obtained starting at the FIFO array origin. The FIFO head field is incremented by n (if the extent is not exceeded) or is set to n minus the difference between the extent and FIFO head field values (if the extent is exceeded).

For tagged doubleword arrays, the number of registers to be loaded must be less than the array extent. For LIFO arrays, the position field value must be greater than the number of registers to be loaded. For FIFO arrays, the interval defined by the incremented value of the FIFO head field and its old value must not include the value of the FIFO tail field (leftmost 8 bits of position field).

Registers are loaded from the array in the order of low-to-high numbered registers if the array is a tagged doubleword or FIFO array. If the array is a LIFO array, registers are loaded from the array in the order high-to-low numbered registers.

An operand selection trap (0107) occurs if insufficient array elements are available to execute the UNDUMP MULTIPLE instruction and the operand selection trap mask bit is not set.

5.3.9 Undump

Formats: RR.

The general register specified by the B operand syllable of the instruction is loaded from the array specified by a structor in the general register identified by the A operand

syllable of the instruction. The action of this instruction is identical to executing an UNDUMP MULTIPLE instruction with a one bit in the bit position of the third control field corresponding to the general register specified by the A operand syllable of the UNDUMP instruction and with a first control field identical to the B operand syllable of the UNDUMP instruction.

5.3.10 Point

Formats: RR, SR, RD

The operand description used to produce the effective A operand is placed in the general register specified by the B operand syllable. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. If the indirection count when Autofetch Evaluation terminates is zero, an operand selection exception trap (0105) is generated or masked. Otherwise the operand description utilized in the Autofetch Conversion operation that terminated Autofetch is placed in the B operand general register. This instruction is similar to the FETCH instruction, except that a structor describing an operand, rather than the operand value, is placed in the specified general register.

5.4 BRANCHING INSTRUCTIONS

The branching instructions are used to alter the current procedure index in the TSB in order to transfer control from one sequence of instructions to another. The branching methods are:

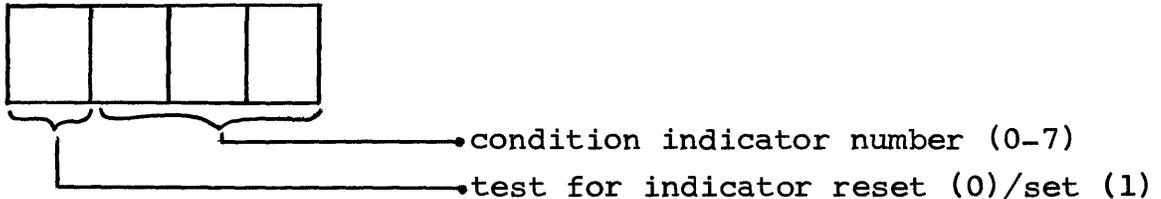
- a. Conditional Branching - based on the state of the condition indicators.
- b. Return Branching - which preserves the updated procedure index.
- c. Loop Control Branching - which uses a count value to control iteration.

The condition indicators are not altered by execution of branching instructions. In particular replacement of the current procedure index in the following subsection does not include alteration of the current condition indicators.

5.4.1 Test and Branch

Formats: RR, SR, RD

The B operand of this instruction specifies a test to be performed on one of the condition indicators. If the test is successful, a branch is performed to the A operand location. Otherwise, instructions are executed in sequence. The test indicated by the B operand field is specified by the following diagram:



In the RR, and SR formats, the effective A operand is derived by applying Autofetch Evaluation to the initial A operand. If the effective A operand is a procedure index control structor, branching is accomplished by replacing the current procedure index with the Autofetched procedure index. Otherwise, an illegal operand trap (0200) is generated or masked.

In the RD format, branching is accomplished by replacing the current value of the instruction location counter with the new instruction location value derived from RD instruction format extraction.

The condition indicators that may be tested by the TEST AND BRANCH instruction are assigned according to the following set of tables.

- a. Each Data Manipulation instruction sets the indicators according to the following table:

5.4.1
(Cont.)

<u>Indicator</u>	<u>Meaning</u>
0	both reset by Data Manipulation instructions
1	
2	High-Order Truncation Indicator
3	Low-Order Truncation Indicator
4	Overflow Indicator
5	High Indicator
6	Low Indicator
7	Equal Indicator

- b. Each Task Control instruction sets the indicators according to the following table:

<u>Indicator</u>	<u>Meaning</u>
0	Reset by Task Control Instruction
1	Set by Task Control Instruction
2	Test Condition Indicator
3	Empty Indicator
4	Full Indicator
5	Available Condition Indicator
6	S Flag
7	-

- c. Each Input/Output Control instruction sets the indicators according to the following table:

<u>Indicator</u>	<u>Meaning</u>
0	Set by I/O Control Instruction
1	Reset by I/O Control Instruction
2	-
3	Logical Protection Indicator
4	-

5	-
6	Facility Busy Indicator
7	Device Malfunction Indicator

5.4.2 Conditional Branch

Formats: RR, SR, RD.

The B operand syllable of this instruction specifies a test to be performed. If the test is successful, a branch is performed to the A operand location. Otherwise instructions are executed in sequence. The test indicated by the B operand syllable is specified by one of the following codes:

- 0000 - test never successful (no branch)
- 0001 - test successful if any condition indicators 2-7 set
- 0010 - test successful if any condition indicators 2-7 reset
- 0011 - test successful if all condition indicators 2-7 set
- 0100 - test successful if all condition indicators 2-7 reset
- 0101 to 1110 - reserved
- 1111 - test always successful (unconditional branch)

In the RR, and SR formats, the effective A operand is derived by applying Autofetch Evaluation to the initial A operand. If the effective A operand is a procedure index control structor, branching is accomplished by replacing the current procedure index with the Autofetched procedure index. Otherwise, an illegal operand trap (0200) is generated or masked.

In the RD format, branching is accomplished by replacing the current value of the instruction location counter with the new instruction location value derived from RD instruction format extraction.

The assignment of condition indicators is specified under the TEST AND BRANCH instruction. (see subsection 5.4.1).

5.4.3 Branch and Link

Formats: RR, SR, RD.

The updated value of the current procedure index (location of next instruction in sequence) is placed in the general register specified by the B operand syllable. A branch is then taken to the A operand location.

In the RR, and SR formats, the effective A operand is derived by applying Autofetch Evaluation to the initial A operand. If the effective A operand is a procedure index control structor, branching is accomplished by replacing the current procedure index with the Autofetched procedure index. Otherwise, an illegal operand trap (0200) is generated or masked.

In the RD format, branching is accomplished by replacing the current value of the instruction location counter with the new instruction location value derived from RD instruction format extraction.

5.4.4 Branch on Decrementd Count

Formats: RR, SR RS, SS, RD

The effective B operand is examined to determine whether it is a tagged logical word or a tagged binary integer. If so, the value of the tagged logical word or tagged binary integer is tested for zero. When the value is zero, no branching takes place, and instructions continue to be executed in sequence. If the value is non-zero, the value is decremented by one, a tagged logical word or tagged binary integer with this value is restored to the effective B operand location, and a branch is taken to the effective A operand location. If the effective B operand is not a tagged logical word or a tagged binary integer, or if the effective A operand is not a procedure index control structor, an illegal operand error trap (0200) is generated or masked.

The effective B operand is derived by applying Autofetch Evaluation to the initial B operand. In the RR, SR, RS, and SS formats, the effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The decremented B operand is restored by applying Autostore Evaluation to the B operand.

In the RR, RS, SR, and SS formats, branching is accomplished by replacing the current procedure index with the Autofetched A operand procedure index. In the RD format, branching is accomplished by replacing the current value of the instruction location counter with the new instruction location value derived from RD instruction format extraction.

5.5 STRUCTOR MANIPULATING INSTRUCTIONS

The structor manipulating instructions are used to form and modify structors in the non-privileged mode so that descriptions of new data structures and substructures of old ones can be generated. The condition indicators are not modified by the execution of these instructions.

5.5.1 Initial Substring

Formats: RR, SR.

The initial B operand resulting from instruction extraction must be an explicit-length structor. The length field of this structor is modified by the value of the effective A operand. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The effective A operand must be a tagged binary integer with non-negative value, or a tagged logical word. If the A operand value is not greater than the value of the length field of the B operand structor, the A operand value replaces the previous value of the length field. The modified structor is then restored to the general register specified by the B operand syllable of the instruction.

If the effective A operand is not a non-negative tagged binary integer or tagged logical word or if the B operand is not an explicit-length structor, an operand specification exception trap (0200) is generated or masked.

5.5.2 Terminal Substring

Formats: RR, SR.

The initial B operand resulting from instruction extraction must be an explicit-length structor. The location, offset, scale, and length fields of this structor are modified by the value of the effective A operand. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The effective A operand must be a tagged binary integer with non-negative value or a tagged logical word. If the type of string described by the B operand structor is byte string, translated byte string, unformatted region, or any form of decimal string, then the following steps are performed:

- a. The value of the length field of the structor must not be less than the value of the effective A operand. If it is less, then an operand selection exception trap (0100) is generated or masked.
- b. The A operand value is added to the location field and is subtracted from the length field of the B operand structor, replacing the previous values of these fields.
- c. The A operand value is subtracted from the values of the scale and length fields of the structor, replacing the previous values of these fields.

If the type of string described by the B operand structor is bit string or binary string, the following steps are performed:

- a. The value of the length field of the structor must not be less than the value of the effective A operand.

If it is less, an operand selection exception trap (0100) is generated or masked.

- b. The A operand value is aligned with the concatenated location and bit offset fields of the structor and added to the aligned bit positions in these fields, replacing their previous value. The alignment adjustment is made to account for bit offset.
- c. The A operand value is subtracted from the values of the length and alignment offset fields of the structor, replacing the previous values of these fields.

If the type of string described by the B operand structor is edit control or floating point string, an illegal operand trap (0200) is generated or masked.

The modified structor is restored to the general register specified by the B operand is not an explicit-length structor, an illegal operand trap (0200) is generated or masked.

5.5.3 Lower Subarray

Formats: RR, SR, RD.

The effective A operand is used to select the lower subarray of the array of items described by the B operand structor. The lower subarray of an array is the set of items in the array with indices running from zero to a value not greater than the extent of the array. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The effective A operand must be a tagged logical word or tagged binary integer not less than zero. If not, an illegal operand trap (0200) is generated or masked. The initial B operand must be a data structor. If not, an illegal operand trap (0200) is generated or masked.

The lower subarray is selected by subtracting the A operand value from the EXTENT field of the B operand structor. If

the result is negative, an operand selection exception trap (0100) is generated or masked. Otherwise, the result replaces the EXTENT field value.

5.5.4 Upper Subarray

Formats: RR, SR, RD.

The effective A operand is used to select the upper subarray of the array of items described by the B operand structor. The upper subarray of an array is the set of items in the array with indices running from a value greater than zero and less than the extent of the array to the extent of the array. The effective A operand is derived by applying Autofetch Evaluation to the initial A operand. The effective A operand must be a tagged logical word or tagged binary integer not less than zero. If not, an illegal operand trap is generated or masked. The initial B operand must be a data structor. If not, an illegal operand trap (0200) is generated or masked.

The upper subarray is selected by applying the Array Indexing operation (See subsection 3.4) to the B operand structor, using the effective A operand as index value. The resulting structor is then restored to the B operand general register.

5.6 SYSTEM CONTROL INSTRUCTIONS

The Test Condition, Empty, Full Available Condition, and S Flag indicators (ref. subsection 5.4.1) maybe set by the System Control indicators. These indicators are always reset prior to the execution of a System Control instruction.

5.6.1 Processor Control (PCON)

Formats: RR

The PCON instruction causes the processor specified by the B operand to respond to a command specified by the A operand. The actual performance of the command is, in general, asynchronous with respect to the execution of the PCON instruction itself.

5.6.1
(Cont.)

Autofetch applies to both operands. Both the final A operand and the final B operand are required to be tagged binary integers or tagged logical words. The A operand is a command word encoded as shown in Figure 5-7. The B operand specifies a processor.

The PCON instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction, and the instruction terminates. (This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow).

If the test and set to the Priority Structure Lock succeeds then the A operand is stored in the first word of the PSA entry of the processor specified by the B operand. This processor (the object processor) is then notified that it should examine its PSA entry and the instruction terminates.

When the object processor recognizes that it has been signalled (see Figure 6-1 Processor Control Flow) it examines its PSA entry and executes the action specified by the command word. In all cases the object processor is responsible for resetting the Priority Structure Lock.

Action	PSA Command Field	PSA Address Field
Store Current Task and Dispatch	00	...
Store Current Task and Enter Wait State	01	...
Store Current Task and Load new Task	02	Address of Ring Pointer of new task.

(Continued)

Action	PSA Command Field	PSA Address Field
Store Current Task and Dispatch skipping current task.	03	...
Change Identity	04	Address of ROM image or other required information.

FIGURE 5-7 COMMAND WORD ENCODING

If either operand is not a tagged binary integer or a tagged logical word then an illegal operand trap (0200) occurs or is masked.

5.6.2 Stop

Formats: RR

The STOP instruction places the task identified by the second operand in the blocked state. Autofetch applies to the B operand and must result in a TSB Identifier. The A operand syllable is required to be zero.

The STOP instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction is terminated. (This has the effect of retrying the instruction after testing for any external signals; See Figure 6-1 Processor Control Flow).

If the test and set to the Priority Structure Lock succeeds then the state of the task identified by the B operand is changed as shown in Table 5-1. In the case of the transition from the Running to Blocked states the processor executing the STOP instruction (the executing processor) constructs a Store Current Task and Dispatch command (See Figure 5-7 Command Word Encoding) and stores it in the PSA

entry of the processor executing the task to be blocked. (The object processor). The executing processor then signals the object processor that it should examine its PSA entry and the instruction terminates.

In the case of the transitions from ready to blocked, and from blocked to blocked the executing processor changes the state bits of the TSB, resets the Priority Structure Lock and terminates the instruction.

In the case of the transition from available to available, the Available Condition Indicator is set, an illegal operand trap (0200) occurs or is masked, the Priority Structure Lock is reset and the instruction is terminated.

TABLE 5-1
STATE TRANSITIONS FOR STOP INSTRUCTION

Old State	New State	Comments	Condition Indicators Set
Running	Blocked	Dispatch occurs	
Ready	Blocked		
Blocked	Blocked		
Available	Available	Trap occurs	Available

If Autofetch applied to the B operand does not result in a TSB Identifier then an illegal operand trap (0200) occurs or is masked.

5.6.3 Start

Formats: RR

The START instruction places the task identified by the B operand in the ready or running state. The task is placed in the running state if it is higher in priority than the lowest priority task currently in the running state; Otherwise it is placed in the ready state.

5.6.3
(Cont.)

Autofetch applies to the B operand and must result in a TSB Identifier or a STOP Protected TSB Identifier. The A operand syllable is reserved and must be set to zero.

The START instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction is terminated. (This has the effect of re-trying the instruction after testing for any external signals; See Figure 6-1 Processor Control Flow).

If the test and set to the Priority Structure Lock succeeds then the state of the task identified by the B operand is changed as shown in Table 5-2. In the case of the transitions from Ready to Running and Blocked to Running the processor executing the START instruction (the executing processor) constructs the appropriate command word (See Figure 5-7 Command Word Encoding) and stores it in the PSA entry of the processor executing the task to be started. (The object processor). The executing processor then signals the object processor that it should examine its PSA entry and the instruction terminates.

In the case of the transition from available to available the Available Condition Indicator is set, an illegal operand trap (0200) occurs or is masked, the Priority Structure Lock is reset and the instruction is terminated.

In the case of the transitions from Running to Running, Ready to Ready, and Blocked to Ready the executing processor changes the state bits of the TSB as appropriate, resets the Priority Structure Lock and terminates the instruction.

The current priority fields of the PSA are used to determine whether the task being started is higher in priority than the tasks which are already running.

TABLE 5-2
STATE TRANSITIONS FOR START INSTRUCTION

Old State	Priority High	Priority Low	Comments	Condition Indicators Set
Running	Running	...		
Ready	Running	Ready		
Blocked	Running	Ready		
Available	Available	Available	Trap	Available

If Autofetch applied to the B operand does not result in a TSB, Identifier or a STOP Protected TSB Identifier then an illegal operand trap (0200) occurs or is masked.

5.6.4 Suspend

Formats: RR.

The SUSPEND instruction places the task identified by the second operand in the ready state.

Autofetch applies to the B operand and must fetch a TSB Identifier. The A operand syllable is required to be zero.

The SUSPEND instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction is terminated. (This has the effect of re-trying the instruction after testing for any external signals; See Figure 6-1 Processor Control Flow).

If the test and set to the Priority Structure Lock succeeds then the state of the task identified by the B operand is changed as shown in Table 5-3. In the case of the transition from Running to Ready the processor executing the

5.6.4
(Cont.)

SUSPEND instruction (the executing processor) constructs a "Store Current Task and Dispatch Skipping Current Task" command (See Figure 5-7 Command Word Encoding) and stores it in the PSA entry of the processor executing the task which is to be placed in the ready state. (The object processor). The executing processor then signals the object processor that it should examine its PSA entry and the instruction terminates.

In the case of the transition from ready to ready, the executing processor resets the Priority Structure Lock and terminates the instruction.

In the case of the transitions from Blocked to Blocked, and from Available to Available, the Available Condition Indicator is set, and illegal operand trap (0200) occurs is masked, the Priority Structure Lock is reset and the instruction is terminated.

If Autofetch applied to the B operand does not result in a TSB Identifier, an illegal operand trap (0200) occurs or is masked.

TABLE 5-3
STATE TRANSITIONS FOR SUSPEND INSTRUCTION

Old State	New State	Comments	Condition Indicators Set
Running	Ready	Dispatch Occurs	
Ready	Ready		
Blocked	Blocked	Trap Occurs	Available
Available	Available	Trap Occurs	Available

5.6.5 Conditional Stop (CSTOP)

Formats: RR.

The CSTOP instruction performs a test of a bit in main storage and, depending on the result of the test, may place the task identified by the second operand in the blocked state.

Autofetch applies to both operands. The final A operand must be a Byte String Array Structor; the final B operand must be a TSB Identifier. If either operand is incorrect an illegal operand trap (0200) occurs or is masked and the instruction terminates.

The CSTOP instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction terminates. (This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow).

If the test and set to the Priority Structure Lock succeeds then the leftmost bit of the leftmost byte of the first string in the array specified by the A operand is used to set the Test Condition Indicator. If the test fails then the Priority Structure Lock is reset and the instruction is terminated.

If the test to the A operand succeeds then the state of the task specified by the B operand is transformed according to Table 5-4.

In the case of the transition from the Running to Blocked states the processor executing the CSTOP instruction (the executing processor) constructs a "Store Current Task and Dispatch" command (See Figure 5-7 Command Word Encoding) and stores it in the PSA entry of the processor executing the task to be blocked. (The object processor). The executing processor then signals the object processor that

it should examine its PSA entry, and terminates the instruction.

In the case of the transitions from ready to blocked, the executing processor changes the state bits of the TSB, resets the Priority Structure Lock and terminates the instruction.

In the case of transition from blocked to blocked the executing processor resets the Priority Structure Lock and terminates the instruction.

In the case of the transition from available to available the Available Condition Indicator is set, an illegal operand trap (0200) occurs or is masked, the Priority Structure Lock is reset and the instruction terminates.

TABLE 5-4
STATE TRANSITIONS FOR CSTOP INSTRUCTION

Old State	New State	Comments	Condition Indicators Set
Running	Blocked	Dispatch Occurs	
Ready	Blocked		
Blocked	Blocked		
Available	Available	Trap Occurs	Available

5.6.6 I/O and External Conditional Stop (ISTOP)

Formats: RR.

The ISTOP instruction performs a raceless test of the Start Flag in the TSB of the task identified by the second operand. Depending on the result of the test, the instruction may place the task in the blocked state.

Autofetch applies to the B operand and must result in a TSB Identifier. If it does not, an illegal operand trap (0200) occurs or is masked and the instruction terminates. The A operand syllable must be zero.

The ISTOP instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction terminates. This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow.

If the test and set to the Priority Structure Lock succeeds then the Start Flag of the task identified by the B operand is used to set the Test Condition Indicator. If the Start Flag is set then the Priority Structure Lock is reset and the instruction terminates. If the Start Flag is reset then the state of the task is transformed according to Table 5-5.

In the case of the transition from the Running to blocked states the processor executing the ISTOP instruction (the executing processor) constructs a "Store Current Task and Dispatch" command (See Figure 5-7 Command Word Encoding and stores it in the PSA entry of the processor executing the task to be blocked (the object processor). The executing processor then signals the object processor that it should examine its PSA entry, and terminates the instruction.

In the case of the Transitions from ready to blocked the executing processor changes the state bits of the TSB, resets the Priority Structure Lock and terminates the instruction.

In the case of the transition from blocked to blocked the executing processor resets the Priority Structure Lock and terminates the instruction.

In the case of the transition from available to available the Available Condition Indicator is set, an illegal operand trap (0200) occurs or is masked, the Priority Structure Lock is reset and the instruction terminates.

TABLE 5-5
STATE TRANSITIONS FOR ISTOP INSTRUCTION

Old State	New State	Comments	Condition Indicators Set
Running	Blocked	Dispatch Occurs	
Ready	Blocked		
Blocked	Blocked		
Available	Available	Trap Occurs	Available

5.6.7 Set Priority (SETP)

Formats: RR.

The SETP instruction stores a new value in the priority field of the Processor Status Array entry of the processor executing the instruction. This modifies the effective priority of the task executing the instruction until it is swapped out by the processor. Autofetch applies to the A operand and must result in a tagged binary integer. The B operand syllable is required to be zero.

The SETP instruction performs a test and set to the Priority Structure Lock. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction is terminated. (This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow).

If the test and set to the Priority Structure Lock succeeds then the rightmost eight bits of the effective A operand are stored in the current priority field of the Processor Status Array entry of the processor executing the instruction. The Priority Structure Lock is reset and the instruction terminates.

If the effective A operand is not a tagged binary integer then an illegal operand trap (0200) occurs or is masked.

This instruction may only be executed in the privileged mode. If an attempt to execute it in non-privileged mode an operand selection (0108) trap occurs.

5.6.8 Undump Interlocked

Formats: RR.

The UNDUMP interlocked instruction performs a raceless load of a tagged doubleword from a FIFO or LIFO array.

Autofetch does not apply to either operand. The A operand must be a tagged doubleword array structor. The first element of the array identified by the A operand must be either a FIFO or a LIFO structor. If the A operand fails to conform to these requirements then an illegal operand trap (0200) occurs or is masked.

The UNDUMP INTERLOCKED instruction performs a test and set to the Queue Lock in the System Base. If the test fails then the sequence counter is set to the location of the op-code of the instruction and the instruction terminates. (This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow).

If the test and set to the Queue Lock succeeds then a tagged doubleword is loaded from the array described by the structor identified by the A operand and placed in the B operand register. The load conventions conform to those described for the UNDUMP MULTIPLE instruction. Once the A operand has been loaded the Queue Lock is reset and the instruction terminates.

If the FIFO or LIFO Array is empty then the Empty Condition Indicator is set.

5.6.9 Dump Interlocked

Formats: RR.

The DUMP INTERLOCKED instruction performs a raceless store of a tagged doubleword into a FIFO or LIFO array.

Autofetch does not apply to either operand. The A operand must be a tagged double word and the B operand must be a tagged doubleword array structor. The first element of the array identified by the B operand must be either a FIFO or a LIFO structor.

The DUMP INTERLOCKED instruction performs a test and set to the Queue Lock in the System Base. If the test fails then the sequence counter is set to the location of the op-code of the instruction and the instruction terminates. (This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow).

If the test and set to the Queue Lock succeeds then the A operand is stored in the array described by the structor identified by the B operand. The store conventions conform to those described for the DUMP MULTIPLE instruction. Once the A operand has been stored the Queue Lock is reset and the instruction terminates. If the FIFO or LIFO array is full the Full Condition Indicator is set. If any of the operands fail to conform to the requirements described above an illegal operand trap (0200) occurs or is masked.

5.6.10 Load Status

Formats RR.

The LOAD STATUS instruction loads an external or I/O status word from the status word in the TSB of the task executing the instruction or from the FIFO array identified by the status word in the TSB, into the B operand register.

The A operand syllable must be zero.

The LOAD STATUS instruction performs a test and set to the Status Word Lock in the TSB. If the test fails the sequence counter is set to the location of the op-code of the instruction and the instruction terminates. (This has the effect of re-trying the instruction after testing for any external signals; see Figure 6-1 Processor Control Flow.)

If the test and set to the Status Word Lock succeeds then the status word of the TSB is accessed. If it is either an I/O or an External Status word it is placed in the B operand register, the Start Flag in the TSB is reset, the Status Word Lock is reset, and the instruction terminates.

If the status word in the TSB contains a FIFO array structure, then the top element in the array is accessed and stored in the B operand register. If the array is empty then the Empty Condition Indicator is set, and an operand selection trap occurs or is masked. In either case the Start Flag in the TSB is reset, the Status Word Lock is reset, and the instruction terminates.

5.6.11 Test and Set

Formats: RR, RS.

The TEST AND SET instruction performs a raceless test of a bit in main storage.

Autofetch applies to the initial B operand and must result in a byte string array structure. The A operand syllable is required to be zero.

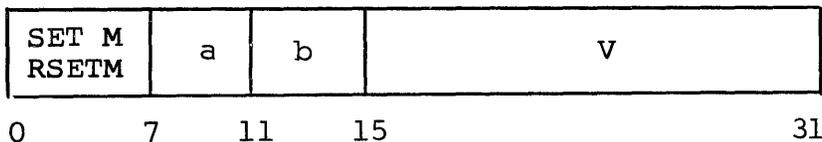
The TEST AND SET instruction accesses the leftmost byte of the first string in the array described by the B operand. It uses the high-order bit of this byte to set the Test Condition Indicator, sets the entire byte to ones and restores it to main storage. No access is permitted to the byte between the time the instruction accesses it and the time it is restored.

If Autofetch does not result in a byte string array structor then an illegal operand trap (0200) occurs or is masked.

Note that if this instruction is used to establish locks to protect access to asynchronously shared data structures, then the boundary between the structures must not occur in the middle of a word in storage.

5.6.12 Set Mode - Reset Mode

Formats : CV



When the Set Mode instruction is executed, bits 42-55 of the Current Procedure Index of the task executing the instruction are set under control of the V field. The A and B fields are ignored and should be set to zero.

If the instruction is executed in the privileged mode, bits 2-15 of the V field are aligned with bits 42-55 of the current procedure index. Each bit in the V field containing a one causes the corresponding bit in the procedure index to be set to one. The remaining bits in the procedure index retain their old values. Bits 0 and 1 of the V field should be zero.

If the instruction is executed in the normal mode then its operation is the same except that the following bits in the procedure Index field will not be set:

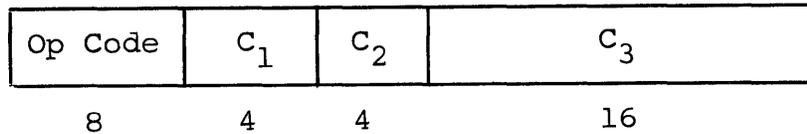
- a. Privilege mode.
- b. Instruction Exception Mask
- c. Operand Selection Exception Mask
- d. Illegal Operand Exception Mask

- e. Timer Mask
- f. Program Controlled Type I Mask
- g. Program Controlled Type II Mask

The operation of the Reset Mode is the same except that the bits in the procedure index field are reset rather than set.

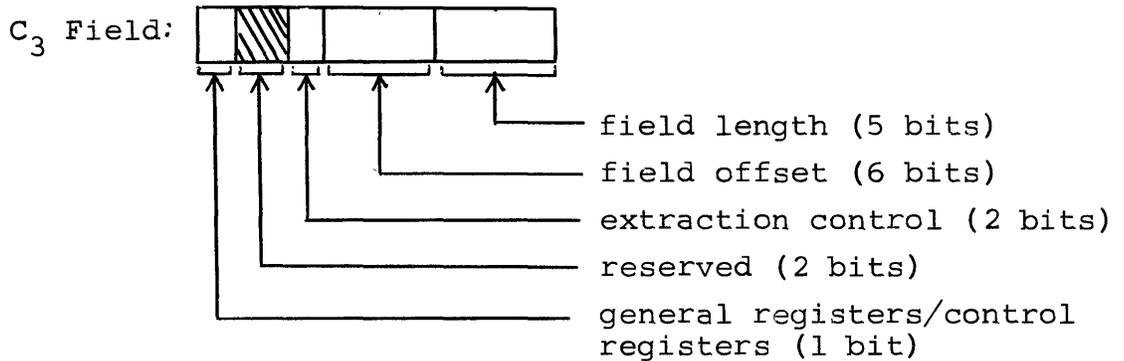
5.6.13 Field Extract

Formats: CV



A field up to 32 bits in length or a 64 bit tagged doubleword is placed in the general register specified by the first control field of the instruction (C₁) in the form of a tagged doubleword. The source of the field or doubleword is any one of the 32 doublewords of the Task Status Block associated with the task executing the FIELD EXTRACT instruction.

The interpretation placed in the third control field (C₃) of the instruction is as shown in the following diagram:



5.6.13
(Cont.)

The first bit in the C_3 field identifies the first sixteen (general purpose) registers or the second sixteen (control) registers of the Task Status Block as the source of the desired field or doubleword. The C_2 field of the instruction selects one of sixteen doublewords within the portion of the Task Status Block identified by the first bit of the C_3 field. The next two bits of the C_3 field are reserved. The following two bits are an extraction control field. The interpretation of the extraction control field is as follows:

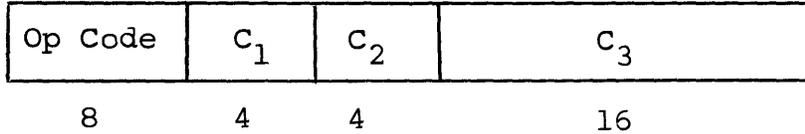
<u>Code</u>	<u>Meaning</u>
00	Form a tagged logical word with a value consisting of the desired field, left justified, and zero filled on the right.
01	Form a tagged logical word with a value consisting of the desired field, right justified, and zero filled on the left.
10	Form a tagged binary integer with a value equal to the value of the field, interpreted as a twos complement integer, filling the appropriate sign bit.
11	Form a tagged doubleword identical to the selected doubleword of the Task Status Block.

The following two subfields of the C_3 field are used to specify the offset (6 bits) and length (5 bits) of the desired Task Status Block field. These two subfields are used only for extraction control codes 00, 01, and 10.

The use of extraction control code 11 (doubleword extraction) is available only in privileged mode. An operand selection trap (0108) is generated if this control code is used in non-privileged mode and the operand selection trap mask bit is not set.

5.6.14 Field Substitute

Formats: CV



The contents of the general register specified by the first control field of the instruction (C₁) may replace the value of a field up to 32 bits in length or a 64 bit doubleword. The destination field or doubleword is any one of the 32 doublewords in the Task Status Block associated with the task executing the FIELD SUBSTITUTE instruction.

The interpretation of the third control field (C₃) of the instruction is similar to the interpretation for the FIELD EXTRACT instruction. The C₂ field, together with the first bit of the C₃ field selects the destination in the Task Status Block. The interpretation of the substitution control field (extraction control field for FIELD EXTRACT) is as follows:

<u>Code</u>	<u>Meaning</u>
00	If the general register specified by the C ₁ field contains a tagged logical word, the leftmost n bits of its value are placed in the designated field, where n is the length of the field.
01	If the general register specified by the C ₁ field contains a tagged logical word, the rightmost n bits of its value are placed in the designated field, where n is the length of the field.
10	If the general register specified by the C ₁ field contains a tagged binary integer, the

rightmost n bits of its value are placed in the designated field, where n is the length of the field.

- 11 The contents of the general register specified by the C_1 field replace the contents of the designated doubleword of the Task Status Block.

The last two subfields of the C_3 field specify the offset and length of the desired Task Status Block field for substitution codes 00, 01, 10.

This instruction may be executed only in privileged mode. An operand selection trap (0108) is generated if this instruction is executed in non-privileged mode, and the operand selection trap mask bit is not set.

5.6.15 Trap Return

Formats: RR

The TRAP RETURN instruction causes a trap to occur regardless of the setting of the trap mask bits in the Current Procedure Index. Both the A operand syllable and the B operand syllable are required to be zero.

The TRAP RETURN instruction causes the contents of General Purpose registers zero, one, and two to be exchanged with the contents of Trap Registers zero, one, and two. The sequence counter in the Current Procedure Index is set to the address of the next instruction and the contents of the Current Procedure Index are exchanged with the contents of the Trap Index. The Trap ID is not altered.

5.6.16 Read Clock

Formats: RR

The READ CLOCK instruction causes the current value of the system clock to be converted to a tagged floating point

number and stored in register Rb. The A operand syllable is required to be zero.

5.6.17 Set Clock

Formats: RR

The SET CLOCK instruction is used to insert a value into the system clock. Autofetch does not apply to either operand. The initial A operand must be a tagged floating point number. The B operand syllable is reserved and must be set to zero.

The mantissa of the A operand is used to set the clock. The sign and exponent fields of the A operand are ignored. This instruction may only be executed in the privileged mode. If an attempt is made to execute it in non-privileged mode an operand selection trap occurs (0108) or is masked.

5.6.18 Set Timer

Formats: RR

The SET TIMER instruction is used to insert a value into the VR of the system timer (see Subsection 8.2). The initial A operand is required to be a tagged floating point number; The B operand syllable is reserved and must be set to zero. Autofetch does not apply to either operand.

Bits 20-35 of the mantissa of the A operand are used to set the value register of the system timer. The sign and exponent fields of the A operand are ignored. This instruction may only be executed in the privileged mode.

If an attempt is made to execute it in non-privileged mode an operand selection trap (0108) occurs or is masked.

5.6.19 Set and Zero Timer

Formats: RR

The SET AND ZERO TIMER instruction is used to insert a value into the VR of the system timer (see Section 9.3) and simultaneously to set the CR to zero. The initial A operand is required to be a tagged floating point number; the B operand syllable is reserved and must be set to zero. Autofetch does not apply to either operand.

Bits 20-35 of the mantissa of the A operand are used to set the value register of the system timer. The clock register is set to zero. The sign and exponent fields of the A operand are ignored. This instruction may only be executed in the privileged mode. If an attempt is made to execute it in non-privilege mode an operand selection trap (0108) occurs or is masked.

5.7 INPUT/OUTPUT INSTRUCTIONS

There are two Input/Output (I/O) instructions: Initiate Device Operation (IDO) and Halt Device Operation (HDO).

5.7.1 Initial Device Operation

Formats: RR, RS, SR, SS

The B operand resulting from application of Autofetch must be Device Identifier Structor. This structor is used to identify the peripheral device to or from which the data transfer will occur, or to which the control operation will be derected.

The A operand must be one of three structors: A tagged doubleword array structor, (which should point to an array or I/O command structor), a Single Control Command specifier structor, or an Alternate Array specifier structor.

This instruction will initiate the execution of the I/O Command Array specified in A operand, using the peripheral device specified in B operand.

The Condition Code in Procedure index is set depending whether the initiation was successfully performed or not.

A more complete discussion of the execution of the IDO instruction is included in Subsection 9.3.2.1.

5.7.2 Halt Device Operation

Formats: RR, RS

The B operand resulting from application of Autofetch must be a Device Identifier Structor. The A operand is not used in this instruction. This instruction will cause a halt of any peripheral operation currently in progress in the device specified in the B operand.

--END OF SECTION--

SECTION VI
TASK MULTIPLEXING6.1 INTRODUCTION

Tasks are time multiplexed on physical processors. This multiplexing is a hardware function and is controlled by the Task Control instructions (see Subsection). The creation and deletion of tasks from the system is a software function.

The two situations which cause the task multiplexing mechanism to be used are:

- a. When a processor is free. In this case the multiplexing mechanism selects the next task for the processor to run. This situation can only occur after the execution of a STOP, SUSPEND, CONDITIONAL STOP, ISTOP, or PROCESSOR CONTROL instruction.
- b. The occurrence of an I/O or External start (see Subsections 6.6 and 6.7).

6.2 LOCK AND UNLOCK FUNCTIONS

In order to prevent race conditions during system control operations a number of locks have been defined. Each of these locks consists of a byte in main storage and is associated with a data structure or process which it is designed to protect. If the high order bit of the lock is one then access is prohibited to the associated structure except by the processor which set the lock. This is the locked state. If the bit is zero then the associated structure is unlocked and a processor is free to lock it by setting the bit to one and then using the structure.

In order to set the lock bit a processor must access it, test to determine if it is zero and, if it is,

6.2 (Cont.)

set it to one and restore it to memory. During the time the processor is performing these operations, access to the byte of main storage containing the lock bit is prohibited. The process of setting the lock bit is referred to as "locking the structure". This process is the same as the one which is available to the programmer through the TEST AND SET instruction.

It is possible for one processor to lock a facility and then to transfer its privileged status with respect to the facility to another processor.

The locks used for system control are:

- a. The Priority Structure Lock: This lock controls the System Base and all of the arrays associated with it. It ensures that only one processor will be able to perform a task control instruction at a time.
- b. The Status Word Lock: There is one such lock in each TSB. It controls the Status Word in the TSB.
- c. The Queue Lock: This lock is used to ensure that only one processor will execute a Dump Interlocked or Undump Interlocked instruction at a time.

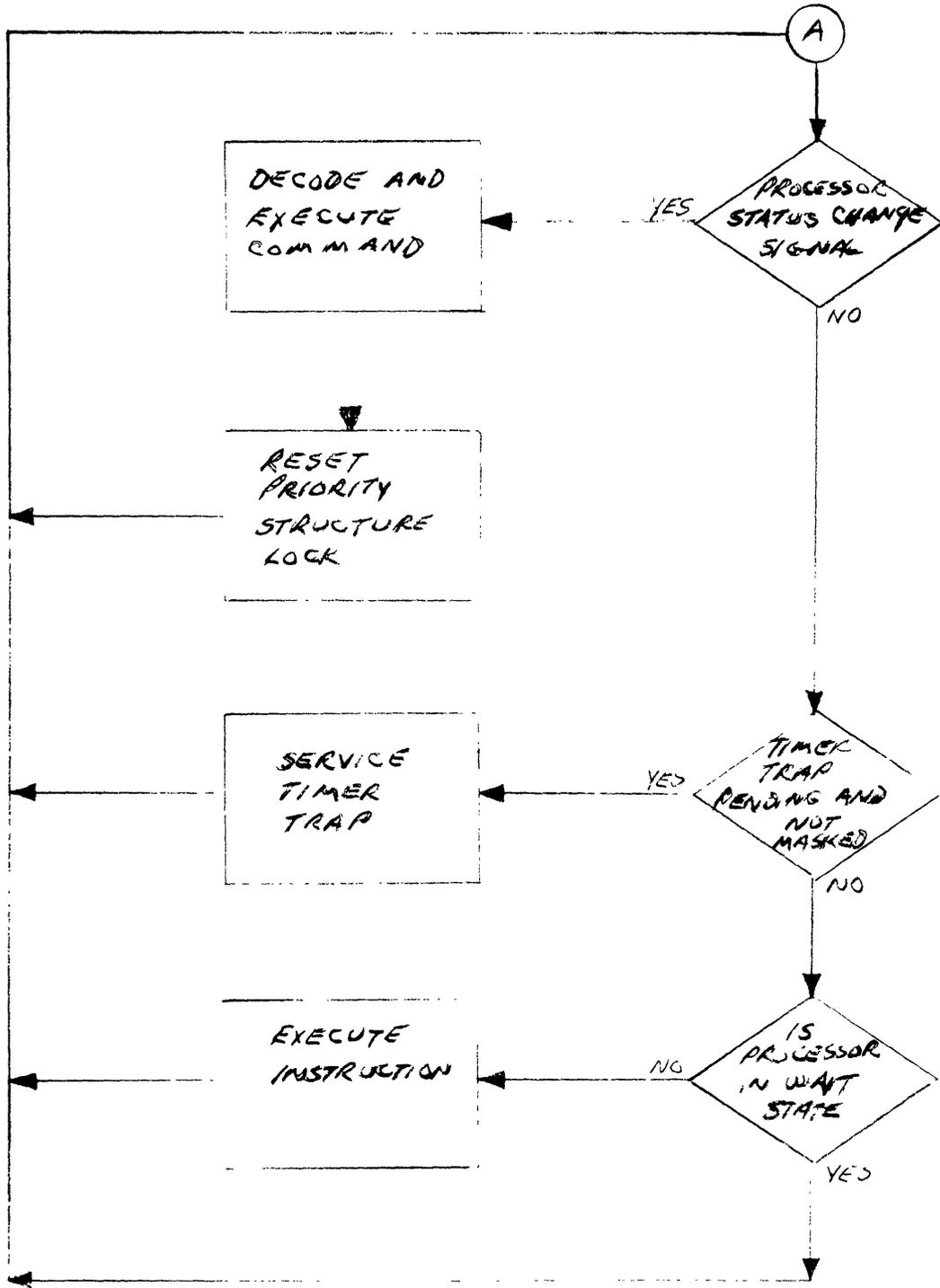


FIGURE 6-1. PROCESSOR CONTROL FLOW

6.3 TASK LOAD

When a processor determines that it is to run a task it loads some portions of the TSB for the task into its local registers. The portions of the TSB which are loaded may vary from one implementation to another. For this reason the contents of the TSB in memory can only be relied on in the following cases:

- a. The facility accessing the TSB must first have successfully performed a Test and Set of the Status Word lock.
- b. The Ring Pointer and Start Status Word are always correct in memory with one exception: The current processor field of the Ring Pointer is correct only when the task is in the running state.
- c. If the Task is in the ready, blocked, or available states the memory copy of the TSB is correct (except the current processor field of the Ring Pointer).
- d. Information obtained from the TSB by means of the Field Extract instruction is always correct.

If, while loading a task a processor finds that the task requires an identity which it does not possess, it restores the TSB to memory and generates an external start to location one of the External Start Array. The processor then places itself in the wait state.

6.4 DISPATCH OPERATION

When a processor discovers that it has no task to run, it selects a task by searching the Priority Array Structure. This search is known as the Dispatch Operation. Note that since the STOP, CSTOP, ISTOP, and SUSPEND instructions all lock the Priority Structure, it is always locked when the need for the Dispatch operation is recognized. The search proceeds as follows:

- a. A register in the processor is set to zero for use as a Level Index Counter.
- b. The Level Index Counter is used in conjunction with the Priority Array Structure in the System Base to select an entry in the Priority Array.

6.4 (Cont.) c. The Level Blocked Indicator of the entry selected from the Priority Array is tested. If it is set, the Level Index Counter is incremented and the process returns to step b. If the Level Blocked Indicator is reset, a register is loaded with the field of the Priority Array entry which specifies the number of tasks in the ring for use as a Task Counter. The location field of the Priority Array entry is used to access the Ring Pointer of the first TSB in the ring.

d. If the status bits of the task whose ring pointer was accessed in step c. are set to ready then:



1. The location field of the Ring Pointer is placed in the location field of the Priority Array Entry.
2. The address of the Ring Pointer and the priority of the task are placed in the second word of the processor's entry in the Processor Status array.
3. The status bits of the Ring Pointer are set to running.
4. The processor's number is placed in the current processor field of the Ring Pointer.
5. The necessary portions of the TSB are loaded into the processor. (See Section 6.3)
6. The Priority Structure is unlocked, and the Dispatch Operation terminates.

If the status bits are set to any other value, the Task Counter is decremented and tested for all ones. If it is all ones the process goes to step e. If it is not all ones then the location field of the Ring Pointer is used to access the Ring Pointer of the next TSB in the field and step d. is repeated.

- e. The last used entry in the Priority Array is accessed and its Level Blocked Indicator is set. The Level Index Counter is incremented and the process returns to step b.

If the process overruns the Priority Array then the processor sets the first byte of the second word of its entry in the Processor Status Array to ones, unlocks the Priority Structure, and places itself in the Wait state.

6.5 PROCESSOR CONTROL FLOW

Each central processor follows the flow shown in Figure 6-1. A single cycle of this flow may be considered to begin at point A and to terminate at point A. Starting at point A the processor determines whether any of three classes of operation are required. These classes are:

- a. Processor Control Operations: Processor Control operations are described in subsection 6.8. The need for a Processor Control Operation is recognized by the presence of a Processor Status Change Signal.
- b. Timer Trap: The Timer Trap is described in subsection 7.3. The need for a timer trap is indicated by the setting of the timer trap pending flag in the TSB.
- c. Instruction Execution: If the processor is running a task, and none of the above operations are required then it will execute the next instruction for the task.

The need for each of these operations is tested in sequence; if none of these operations are required the processor starts again at point A.

If the Processor Status Change Signal is present then

a processor Control Operation is required. The processor examines the command field of its PSA entry and performs whatever action is specified there. Once this has been done the Priority Structure Lock is reset and the process returns to point A.

If a timer trap is pending and not masked then it is serviced as described in subsection 7.3.

If a task is being run then the next instruction is executed for it.

6.6 I/O INITIATED STARTS

An I/O start is a process whereby an I/O device may transfer a double word of status information to a task and cause the task to be placed in the ready or running state.

The sequence of operations performed by the I/O in order to store a status word is shown in Figure 6-2. The operation is as follows:

- a. The Priority Structure Lock is set.
- b. I/O resources are deallocated, if required (see Sub-section 9.6).
- c. Bits 8-15 of the I/O Status Word are used to index the I/O Start Array. If the index value exceeds the extent of the array the last entry in the array is used. The entry in the array determines what further action is to be taken; if the I/O Start Array entry is a TSB Identifier the process continues with step f; if the I/O Start Array entry is a tag zero structor then the process terminates; if the I/O Start Array entry is a tagged doubleword array structor then the array which it identifies is called an Index Array and the process continues with step d.

6.6 (Cont.)

- d. Bits 16-19 of the I/O Status Word are used to index the Index Array Structor. If the index value exceeds the extent of the array the last entry in the array is used. The entry in the array determines what further action is taken; if the Index Array entry is a TSB Identifier the process continues with step f; if the Index Array entry is a tag zero structor then the process terminates; if the Index Array entry is a tagged doubleword array structor then the array which it identifies is also called an Index Array an the process continues with step e.
- e. Bits 20-23 of the I/O Status Word are used to index the second Index Array Structor. If the index value exceeds the extent of the array the last entry in the array is used. The entry in the array determines what further action is taken; if the Index Array entry is a TSB Identifier the process continues with step e. If the Index Array entry is a tag zero structor then the process terminates. No other entries are allowed in the second Index Array.
- f. A test and set is performed to the Status Word Lock in the TSB. If the test fails then step f. is repeated. If the test succeeds the Start Flag in the TSB is set and the Start Status Word is accessed. If the Start Status Word is a FIFO Array Structor then the I/O Status Word is stored in the array. If the array is full then the last word in the array is overwritten. If the Start Status Word is not a FIFO Array Structor then the I/O Status Word is deposited in the Start Status Word.
- g. In either case the Status Word Lock is reset and the state of the task is transformed as shown in Table 6-1.

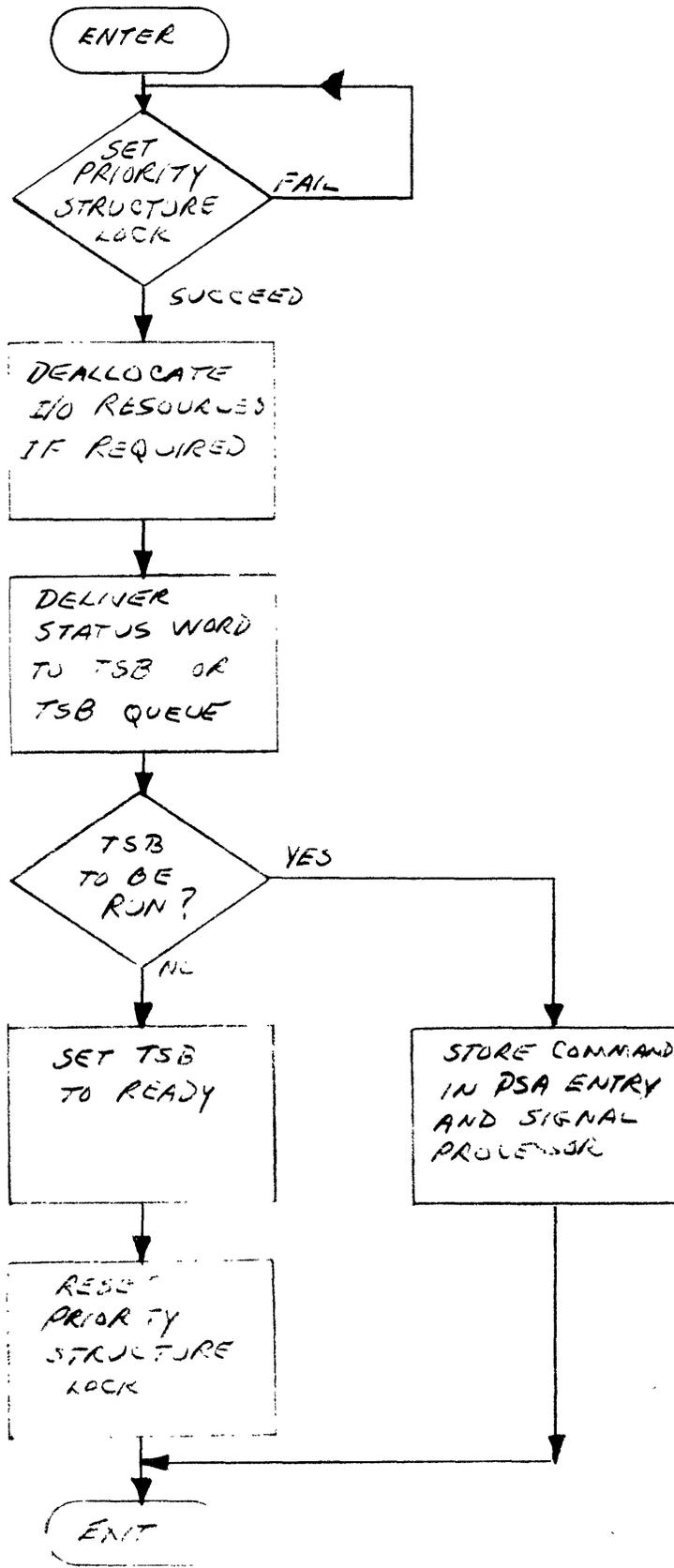


FIGURE 6-2. I/O START

The Level Blocked Indicator in the corresponding Priority Array entry is reset, and the Priority Structure is unlocked.

TABLE 6-1
TASK STATE TRANSFORMATION FOR I/O AND EXTERNAL STARTS

OLD STATE	NEW STATE	
	PRIORITY HIGH*	PRIORITY LOW*
Running	Running	...
Ready	Running	Ready
Blocked	Running	Ready
Available	Available	Available

*Priority High indicates that the priority of the task being started is higher than that of the lowest priority task currently in the running state. Priority Low indicates the reverse situation.

6.7 EXTERNALLY INITIATED STARTS

An External Start is process where by a source external to the system may transfer a doubleword of status information to a task and cause the task to be placed in the ready or running state. The mechanization for external starts is identical to that for I/O starts except that the External Start Array is substituted for the I/O Start Array, and an External Status Word is formed in place of an I/O Status Word.

6.8 PROCESSOR CONTROL OPERATIONS

Individual central processors are controlled by means of a communications buffer in their PSA entries, and an interprocessor signalling facility. Every processor has the capability of signaling each of the other processors in order to notify them that there is a message in their

6.8 (Cont.)

communications buffer.

When a processor (the command processor) wishes to cause another processor (the object processor) to perform some action it locks the Priority Structure, stores a command word in the PSA entry of the object processor and signals the object processor. The command processor does not attempt to reset the Priority Structure Lock.

When the object processor recognizes that it has been signalled (see Section 6.5) it examines its PSA entry and performs whatever action is required. The object processor is responsible for resetting the Priority Structure Lock.

The encoding of the command words is shown in Figure 5-7. The action taken in response to the various command words is as follows:

STORE CURRENT TASK AND DISPATCH: The TSB of the task being run by the object processor (if any) is restored to memory and the processor executes a Dispatch operation.

STORE CURRENT TASK AND ENTER WAIT STATE: The TSB of the task being run by the object processor (if any) is restored to memory and the processor does not load a new task. The current priority field of it's PSA entry is set ot ones and it continues to test for Processor Control signals (see Section 6.5).

STORE CURRENT TASK AND LOAD NEW TASK: The TSB of the task being run by the object processor (if any) is restored to memcry and the processor loads a new task from the address specified in the command word.

6.8 (Cont.)

CHANGE IDENTITY: The object processor assumes a new identity. The exact mechanism depends on the individual processor implementations.

STORE CURRENT TASK AND DISPATCH SKIPPING CURRENT TASK: The TSB of the task being run by the object processor (if any) is restored to memory and the processor performs a Dispatch operation. During the course of this operation the task which was restored to memory is skipped over. That is it will not be run even if it is the highest priority task in the ready state.

This facility is made available to the programmer via the PROCESSOR CONTROL instruction.

SECTION VII
TRAPPING

7.1 INTRODUCTION

A trap is a control transfer initiated by a hardware detected exception condition. Traps are internal to the task in which they occur.

7.2 TRAPPING INFORMATION STRUCTURE

The parts of the TSB used for trapping are:

- a. Trap Index: This doubleword must contain a procedure index. When a trap occurs the contents of this location are exchanged with the contents of the updated current procedure index.
- b. Trap Registers: Three general purpose register images the contents of which are exchanged with the contents of GPR's 0, 1, and 2 when a trap occurs.
- c. Trap ID (see Figure 7-1): A doubleword which, after a trap, identifies the cause of the trap.



FIGURE 7-1. TRAP ID FORMAT

- d. Trap Mask: An eight bit status field which allows each of the types of traps to be individually masked. This field is a part of the procedure Index.

7.3

TRAP CAUSES

There are eight types of traps:

- a. Instruction Exception: This class includes all error conditions which are recognizable from examination of the instruction alone. These include illegal opcodes and instruction formats.
- b. Operand Selection Exception: This class includes all error conditions recognized during the operand extraction process and during Autofetch/Autostore.
- c. Illegal Operand Exception: This class includes all traps generated when a operand is recognized as being illegal, given the instruction and the state of the machine.
- d. Machine Check: This occurs when a hardware failure is detected.
- e. Arithmetic Exception: This class includes all traps generated during the actual manipulation of the data.
- f. Timer: This trap is generated when the task timer is decremented through zero.
- g. Program Controlled Type I: This trap is generated when a Type I Trap Effector is used as an operand or during operand selection.
- h. Program Controlled Type II: This trap is generated when a Type II Trap Effector is used either as an operand or during operand selection.

These types of traps are mutually exclusive so that only one type can occur at a time. When a trap occurs the type is used to set bits 0-7 of the Trap ID.

7.3 (Cont.)

Within each of these types there may be a number of different specific causes. These will set some or all of bits 8-63 of the Trap ID. The conditions which may cause traps are discussed along with the instructions or functions where they occur. These are also shown in Table 7-1.

TABLE 7-1
TRAP CONDITIONS

TRAP TYPE	CAUSES	SETTING OF BITS 0-7 OF TRAP ID	SETTING OF BITS 8-15 OF TRAP ID	SETTING OF BITS 16-63 OF THE TRAP ID [#] (Zero If Unspecified)
Instruction Exception	1) Illegal Opcode	00	00	Unspecified
Operand Selection Exception	1) Excess Extent during selection and substring operations	01	00	Address of Array structor
	2) Qualification		01	Address of modifier structor
	3) Extraction		02	
	4) Alterability		03	Address of nonalterable structor
	5) Autostore conversion		04	Address of structor causing trap
	6) Autofetch conversion		05	Address of structor causing trap
	7) Excess indirection		06	Address of last structor used for indirection
	8) Full/Emty LIFO/ FIFO Array		07	Address of Array structor
	9) Privilidge violation		08	Unspecified
	10) I/O		09	Unspecified

(Continued)

[#]See footnote, page 7-6.

Table 7-1 (Cont)

TRAP TYPE	CAUSES	SETTING OF BITS 0-7 OF TRAP ID	SETTING OF BITS 8-15 OF TRAP ID	SETTING OF BITS 16-63 OF THE TRAP ID# (Zero If Unspecified)
	11) Non Binary collate or translation Table not present		0A	
Illegal Operand Error	1) Incompatible Operand types	02	00	Unspecified
Machine Check	1) Hardware error	03	*	Implementation dependent
Arithmetic Exception	1) Overflow	04	00	Address of result structor or register
	2) Exponent over or underflow		01	Address of result structor or register
	3) Significance loss		02	Address of result structor or register
	4) Negative result generated for an unsigned field		03	Address of result structor or retister
	5) Zero		04	Address of zero operand or the structor describing it
Timer	1) Timer decremented through zero	05	00	Unspecified

#See footnote, page 7-6.

(Continued)

*See footnote, page 7-6.

Table 7-1 (Cont.)

TRAP TYPE	CAUSES	SETTING OF BITS 0-7 OF TRAP ID	SETTING OF BITS 8-15 OF TRAP ID	SETTING OF BITS 16-63 OF THE TRAP ID# (Zero If Unspecified)
Program Controlled Type I	1) Access of Type I trap effector	06	00	Address of trap effector
Program Controlled Type II	2) Access of Type II trap effector	07	00	Address of trap effector

*Depends on machine implementation

#Where an address is supplied it will be supplied as shown in Figure 7-1.

7-6

7.3 (Cont.)

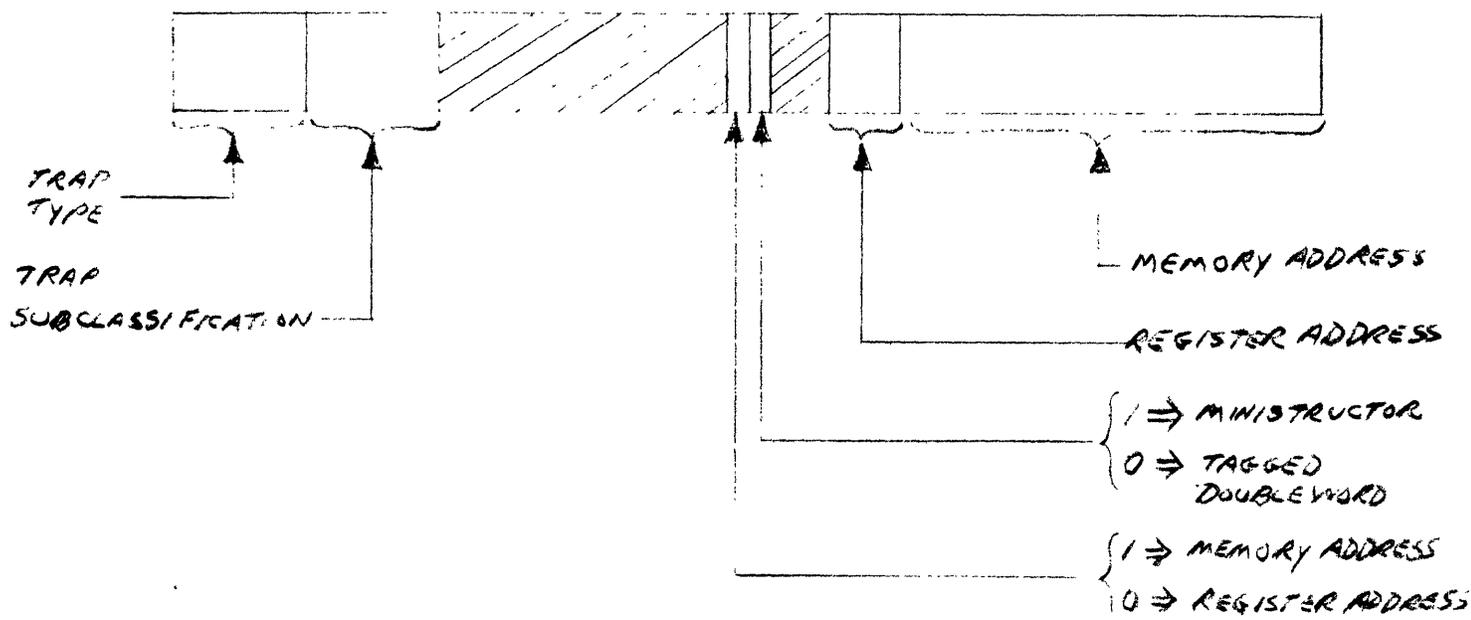


FIGURE 7-1. LAYOUT OF TRAP ID

Figure 7-1 shows the layout of the Trap ID. All unused bits are set to zero.

7.4 TRAP MECHANIZATION

When a trap condition is recognized the location counter in the current procedure index is set to point to the instruction causing the trap. The cause of the trap is stored in the Trap ID field of the TSB. The appropriate trap mask bit is tested to see if the trap is allowed. If the trap is masked the location counter is updated, the instruction terminates, and processing continues. If the trap is allowed to trap registers are swapped with GPR's 0, 1, and 2, the current procedure index is swapped with the trap index, and the instruction terminates.

NOTE

The numbers appearing in brackets after the mention of a trap are a hexadecimal representation of the setting of bits 0-15 of the trap ID after the trap occurs.

SECTION VII**I**
TIMING FACILITIES8.1 SYSTEM CLOCK

The system clock measures actual time of day. The clock is a 52 bit binary integer which is accessed by the Read Clock and Set Clock instructions (see Section 6. and 6.).

The system Clock is incremented at bit position 35 every millisecond.

8.2 SYSTEM TIMER

The system timer facility consists of two 16 bit registers: The value register (VR) and the clock register (CR). The clock register is incremented every millisecond.

The clock register is continuously compared with the value register. Whenever the contents of the CR are found to be greater than or equal to the contents of VR the clock register is reset to zero and an external start directed to location zero of the external start array occurs.

8.3 TASK TIMER

Each task has a task timer which measures the amount of time the task is in the running state. The task timer is in doubleword 18 of the TSB. It is 32 bits long and has a resolution of one microsecond. Its precision may, however, be less on some systems. The task timer is treated as a twos complement number which is decremented at the interval required by its precision, while the task is in the running state. It is not decremented if the processor is stalled waiting for a memory access and it is not decremented during any

8.3 (Cont.)

time the processor spends performing task control operations.

When the timer is decremented through zero a trap (0500) results. The trap occurs after the end of the current instruction, and before the beginning of the next instruction. If a timer trap is masked it is held pending until the mask bit is reset. During the time the trap is pending the timer continues to be decremented. This is the only trap which may be held pending.

SECTION IX
INPUT/OUTPUT FACILITIES

9.1 INPUT/OUTPUT OPERATIONS

Input/Output (I/O) operations involve the use of certain system facilities or resources. This set of resources, which is referred to as the Input/Output or Peripheral Subsystem, is defined as those system resources whose purpose is to initiate, monitor, sustain or terminate data transfers, or to execute control operations in a peripheral device.

Conceptually, the I/O subsystem can be viewed as a different area of functionality. This is illustrated in Figure 9-1.

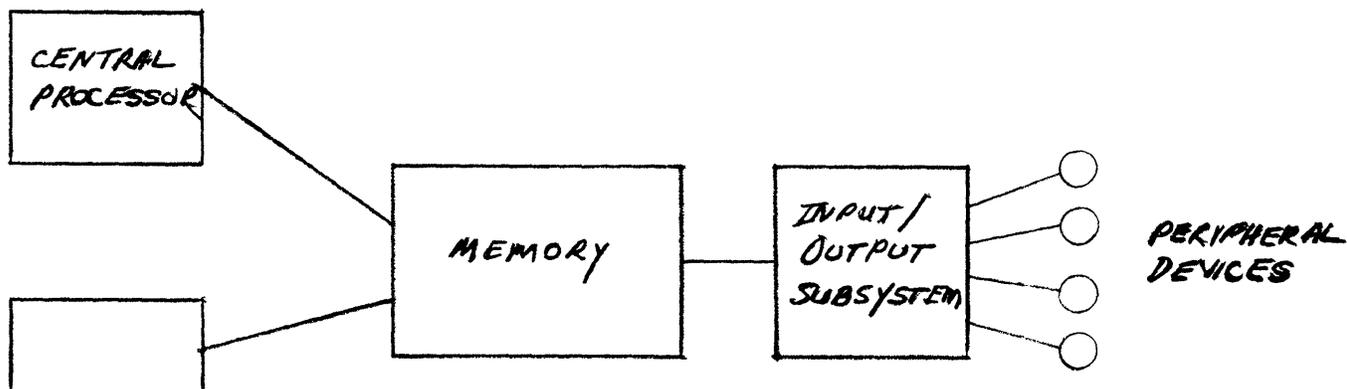


FIGURE 9-1. I/O FACILITIES

This does not imply, however, that in certain implementations or configurations the central processor and I/O subsystems cannot share common resource, e.g., control logic, control memory, etc. For the purposes of definitions, and only from a functional point of view, we will consider the I/O subsystem as a separate entity. It will be capable of executing I/O operations without intervention from the central processor task that originated them.

A peripheral device is a single addressable data source or data sink. It may be a unit producing physical media, such as a disk or tape drive or an electronic imedia such as a communications channel.

The I/O operations, as defined by the I/O commands, are classified in the following categories:

a. Data Transfer Commands:

These commands involve the transfer of data to/from peripheral devices, and from/to memory.

b. Control Commands:

These commands involve the transfer to the device of control information only, e.g., setting a parameter or condition in a peripheral device, rewinding a tape, seeking a cylinder on a disk drive, etc.

c. Inquire Commands:

These commands are used to retrieve pertinent status information from the peripheral subsystem.

The I/O commands can be grouped into an array defining a series of I/O operations to be performed by the peripheral subsystem. The execution of the array is triggered by the central processor issuing an I/O instruction.

It is important to note the difference between an I/O operation and an I/O instruction. An I/O operation is the minimum command for the peripheral subsystem. The I/O instruction, which is a central processor native mode instruction, can indicate the execution of a whole array of I/O operations. This difference is presented in Figure 9-2..

I/O operations are executed in sequence by the peripheral subsystem, until the completion of the array.

Since some of the I/O commands in the array may specify data transfer operations, and since some resources in the peripheral subsystem have a limited transfer rate capacity, the central processor will protect against the issuing of I/O

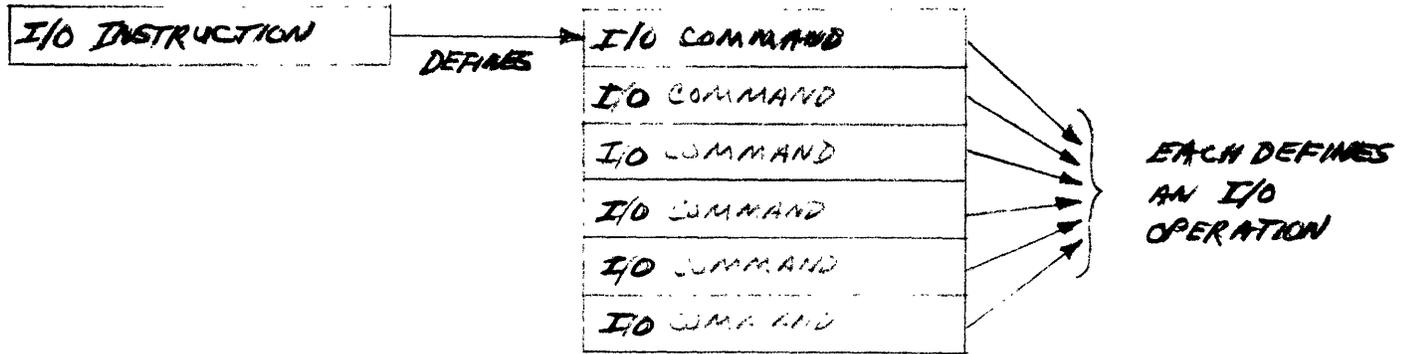


FIGURE 9-2. I/O INSTRUCTIONS AND OPERATIONS

instructions which can potentially create an "over run" condition in those resources.

Also, since many arrays can be simultaneously under execution (the maximum number determined by the number of levels of simultaneity of the peripheral subsystem) the central processor will protect the issuing of I/O instructions whose requirements exceed the number of levels of simultaneity available.

The peripheral subsystem will execute the array of I/O commands, (or sequence of I/O operations) without direct intervention of the task that originated it.

Upon completion of the array, (or during its execution if programmed to do so in the I/O commands), the Peripheral Subsystem will signal the Central Processor(s) to indicate this fact, as described in Paragraph 9.6.

9.2 INPUT/OUTPUT COMMAND STRUCTORS

The Input/Output Command Structor contains information directing a device to carry out a single operation. The structor is meaningful only to the Peripheral Subsystem. When several I/O command structors are grouped together, they form an I/O command array which in itself is a program executable by the Peripheral subsystem.

The format of an I/O command structor is as shown in Figure 9-3.

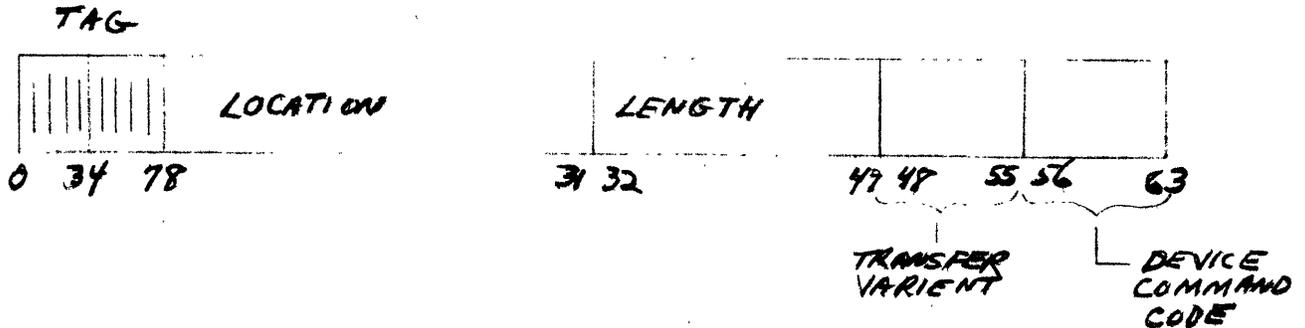


FIGURE 9-3. I/O COMMAND STRUCTOR FORMAT

The eight-bit DEVICE COMMAND CODE (DCC) field will define the particular I/O operation. Up to 255 DCC's can be specified for each type of peripheral device. One DCC (All zeros) is reserved for a No-Operation code. The particular bit configuration for the Device Command Code can be found in the corresponding Peripheral Devices/Processors specifications.

The Device Command Codes (and consequently the I/O command structors) are classified in the following categories:

- Data Transfer Commands.
- Control Commands.
- Inquiring Commands.

The Data Transfer Commands will initiate transfer of data to/from peripheral devices, and from/to memory. The Control Commands will initiate a control operation such as rewinding a tape, seeking a cylinder on a disk drive, etc. The data to be transferred (if any) will be control information only. The Inquire Commands retrieves status information.

The 24-bit LOCATION and the 16-bit LENGTH fields, will define a main memory area or buffer. If the DCC indicates a Data Transfer command the buffer will contain (or will be the destination of) the data. If the DCC implies a Control Operation command, the buffer will include all the additional control information, such as the cylinder address to

perform a seek, etc. If the DCC indicates an Inquire Command, the status information will be stored in the buffer. The LENGTH field specifies the number of bytes to be transferred. The maximum length for a single command is 65,535 bytes.

The LOCATION field is a byte address, pointing to the first byte to be transferred. Since data transfers always start in the high-order byte of a word, the lower order two bits should be zeros. There is no similar restriction in the length of the transfer.

An exception to the above mentioned restriction in the location field is when the DCC indicates a Read Backward operation. If this is the case, the location field is not restricted to have its two lower order bits as zeros. I.e., the data transfer can start in any byte position in the word.

The 8-bit TRANSFER VARIANT contains information to be used by the Peripheral subsystem in maintaining the peripheral operation and in sequencing through the I/O command structures in the array. The Transfer Variant is formatted as shown in Figure 9-4. The functions to be performed are explained with the individual bit configuration.

The signals to the Central Processor(s) caused by a residue storage or programmable signal will be explained in Paragraph 9.6.

The branching capabilities as described in Figure 9-4, are not complete and they will be further discussed in next paragraph.

9.2.1 The I/O Command Array

The I/O command structures can be grouped in an array, which is executed by the Input/Output subsystem.

Upon satisfactory completion of an individual command, the Input/Output subsystem will issue the next one in sequence

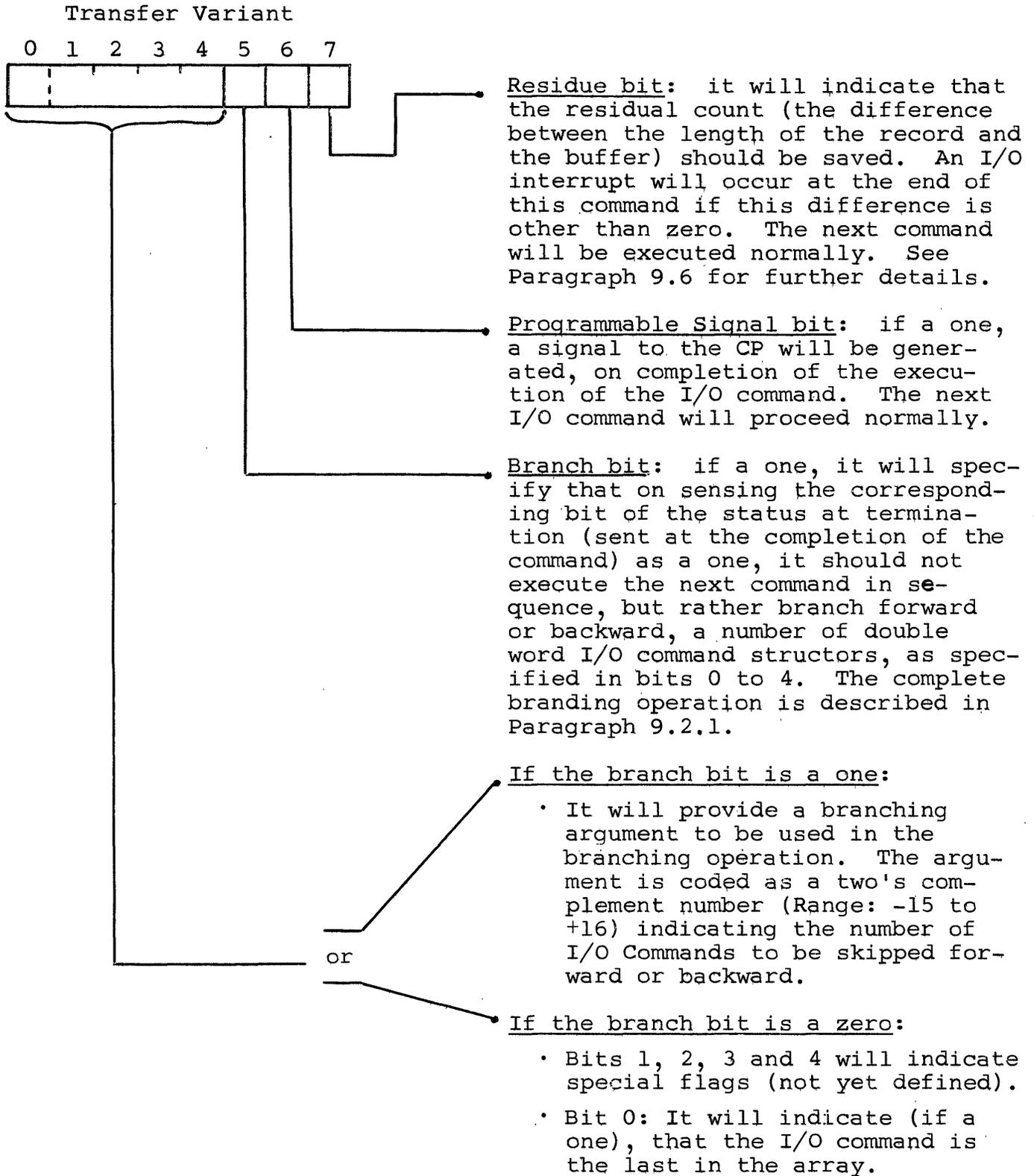


FIGURE 9-4. TRANSFER VARIANT FORMAT

until completion of the array, the last I/O command in the array being indicated by a ONE in Bit 0 of the Transfer Variant.

Normally, an I/O command will specify a peripheral operation such as reading or writing a record of date. In addition, and using bits 0 to 5 of the Transfer Variant, the I/O command is provided with a conditional branching mechanism. This branching process works as follows:

Upon completion of an I/O Command, if the Branch bit (bit 5) of the Transfer Variant is a ONE, the corresponding bit in the status send at the completion of the command is sensed, and if a 1, the branching argument of the Transfer Variant is extracted (bits 0-4) and its value added to the argument send as a second byte of status. (This later argument is an 8-bit two's complement number.) The result of the addition referents the number of I/O commands to be skipped [forward (if the resulting number is positive) or backward in in the array. The extent field is checked to protect for the issuing of commands outside the boundaries of the I/O Command Array.

At the completion of the array, a signal is generated to a pertinent task, as described in Paragraph 9.6. Abnormal situation such as errors or illegal commands will cause the immediate termination (abnormal termination) of the array and a signal generated.

9.3 INPUT/OUTPUT INSTRUCTIONS

The I/O instruction repertoire consists of the following instructions:

- a. Initiate Device Operation (IDO)
- b. Halt Device Operation (HDO)

The IDO instruction has two operands associated with it. The B operand will always identify a logical device, and the A operand which will point to an array of I/O Commands

to be executed by the specified device. Both operands will be used in the assembling of an I/O operand, which will describe to the Peripheral Subsystem, which device to use, and where the array of I/O commands is located.

The HDO instruction has only one operand, the B operand, which identifies a logical device. It will also assemble an I/O operand to be used by the peripheral subsystem. (I/O operands are described in the following subsections.)

9.3.1 Protection and Allocation Tables

In order to execute a Peripheral operation, three basic resources are needed: a peripheral device, a level of simultaneity of the I/O subsystem and transfer rate available in all those shared resourced which will transfer either control information or data.

Each one of those resources is checked by the Contral Processor before issuing an I/O instruction to the Peripheral Subsystem. The availability of the device and its logical assignment is checked in the Device Specification Table. The level of simultaneity in the Simultaneity Table and the transfer rate in the corresponding entries in the Traffic Registers. In order to avoid simultaneous accesses to the table from more than one process, there is a common lock in the system base, which should be "tested and setted" when accessing the table to allocate or deallocate resources.

a. Device Specification Table: This table is located in main storage, its starting location is contained in the System Base.

There is an entry per device and each entry is 2-word long. Its format is shown in Figure 9-5.

The first byte contains flags which indicate a series of device status bits and actions to be performed during extraction of the order that reference that physical device. Its format is shown in Figure 9-6.

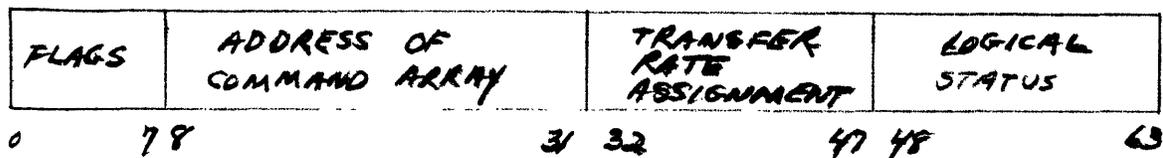


FIGURE 9-5. DEVICE SPECIFICATION TABLE ENTRY FORMAT

The Device Status Table also includes a 24-bit ADDRESS OF COMMAND ARRAY field. This field will indicate the initial address of the array under execution, if the device is currently executing a data transfer operation.

The Table also includes a 16-bit current device TRANSFER RATE ASSIGNMENT, and the 16-bit LOGICAL STATUS field which includes the logical assignment of the peripheral device, as determined by the software.

All entries in this table are maintained by software with the exception of the busy device flag and the address of the Command array, which are dynamically maintained by the hardware, upon successful initiation of an IDO order.

The detailed use of the Device Specification Table will be further clarified in describing the extraction of the IDO order in Paragraph 9.3.2.1.

- b. Simultaneity Table: This table is located in main storage with the starting location contained in the System Base.

There is an entry per each resource which can potentially have multiple levels of simultaneity. Each entry is one byte wide and it is formatted as shown in Figure 9-7.

The entry contains initially a binary number equal to 127 minus the levels of simultaneity of the resource. It is originally set up by the software and it is

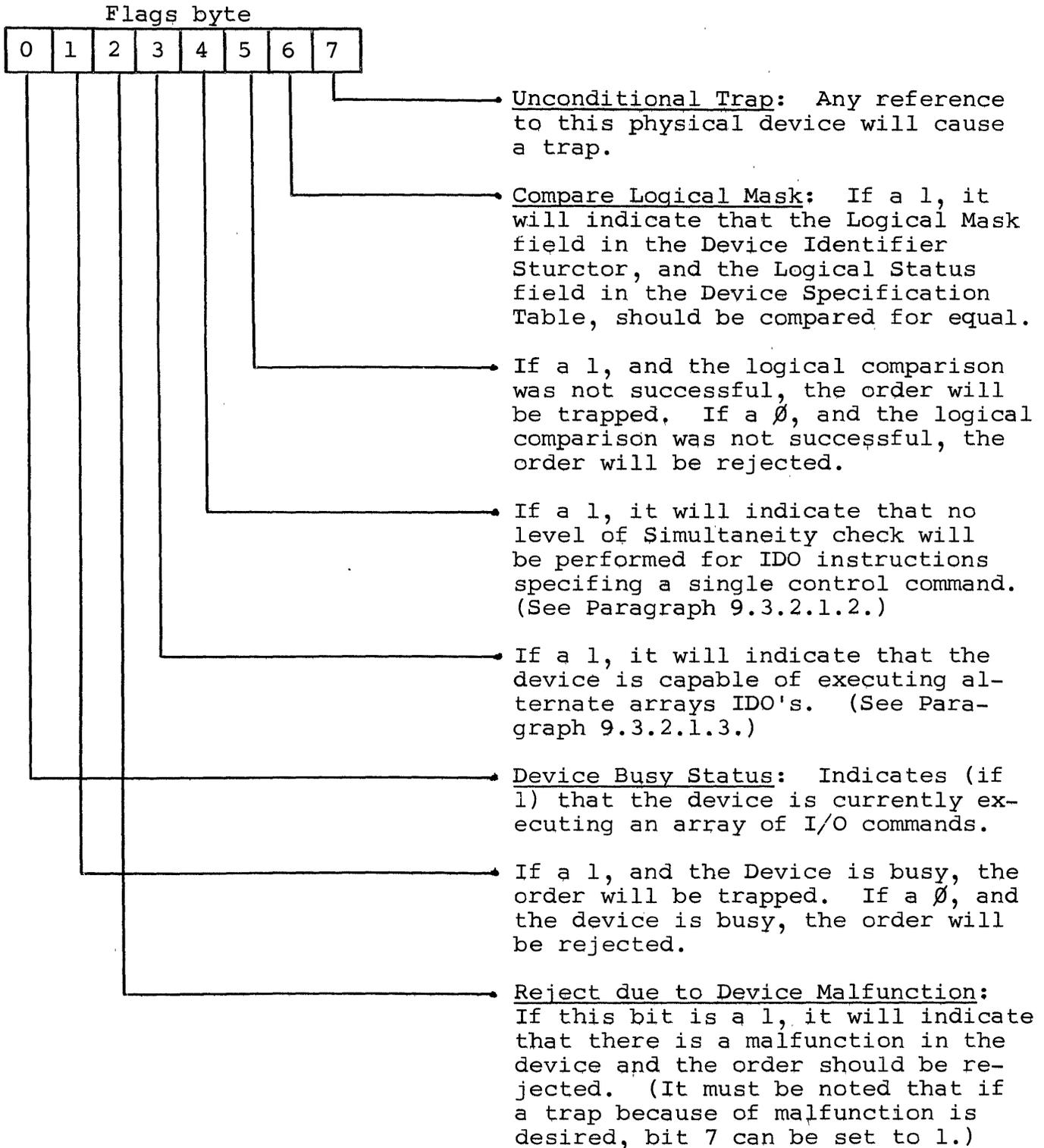


FIGURE 9-6. FLAGS IN DEVICE SPECIFICATION TABLE

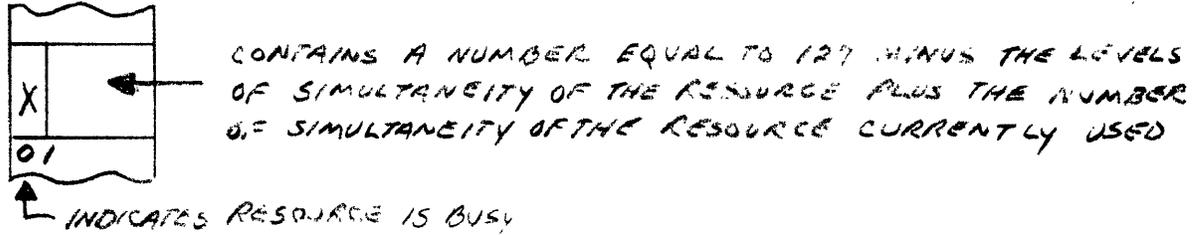


FIGURE 9-7. SIMULTANEITY TABLE FORMAT

dynamically maintained by the CP when issuing or terminating peripheral instructions.

If in attempting to initiate an IDO instruction, the indexing of the entry forces bit zero to be a ONE, it will indicate that the resource is busy, i.e., does not have any further level of simultaneity available.

The table is used during extractions of IDO orders.

- c. Traffic Registers: There is one traffic register per resource in which overrun can occur and one traffic register for the total system. The traffic registers are located in main storage starting in the location specified in the System Base.

Each entry is a word long, and its format is shown in Figure 9-8.

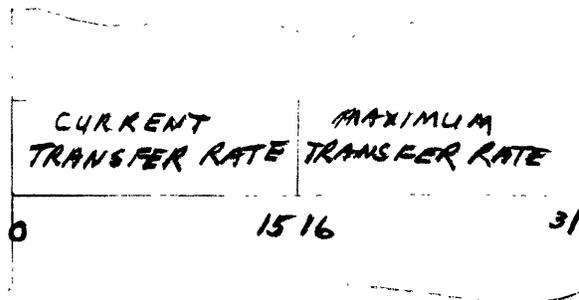


FIGURE 9-8. TRAFFIC REGISTER FORMAT

The two low order bytes contain the maximum allowable transfer rate in the resource, as set up by the software. Upon initiation of an IDO order, the hardware will add the transfer rate specified in the Device Specification Table to the two high order bytes, and then compare the result with the maximum, in order to check for potential overruns.

The transfer rates are represented as a 16-bit binary integers, in which the least significant bit corresponds to a frequency of 64 transfers/record. This implies that the maximum representable transfer rate is 4,194,304 transfer/sec.

9.3.2 Central Processor Input/Output Instructions

There are two Input/Output instructions: Initiate Device Operation (IDO) and Halt Device Operation (HDO).

9.3.2.1 The INITIATE DEVICE OPERATION Instruction

The formats of the IDO instruction are shown in Figure 9-9.

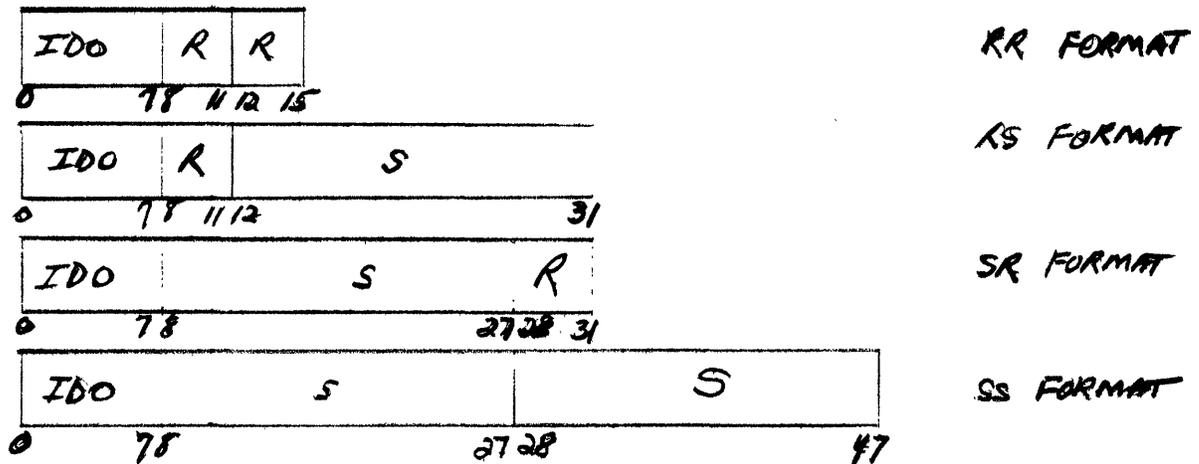


FIGURE 9-9. IDO INSTRUCTION FORMATS

(See Section III for details on Central Processor instruction formats.)

The B operand will be a Device Identifier Structor. Its format is shown in Figure 9-10.

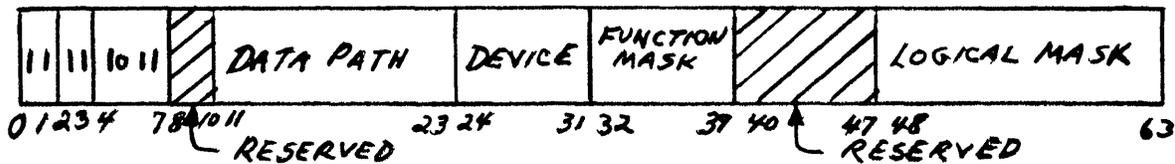


FIGURE 9-10. DEVICE IDENTIFIER STRUCTOR FORMAT

Starting from the right, the 16-bit LOGICAL MASK field will be used during extraction of the IDO order, and it will be compared for equal with a 16-bit Logical Status field in the Device Specification Table. The IDO orders will then be restricted to access the device, only if its logical status is the same as the Device Identifier Structor Logical Mask.

The 8-bit Function Mask field defines restrictions in the use of the device, as specified in the peripheral device specifications (e.g., no seek allowed, no control operations, etc.)

The eight-bit DEVICE field specifies the actual device address to be used.

The 13-bit DATA PATH field describes any required routing information.

Upon extraction of the Device Specifier Structor, (or B operand), the protection and allocation of the peripheral resources will be effected, using the corresponding table entries.

Information defining which resources to protect or allocate is included in the A operand, which can specify different types of command arrays, each with different protection requirements.

There are three types of A operand, which will define three kinds of IDO instructions as follows:

A Operand = Implicit length. tagged double word structor: IDO's initiating a Data Transfer Array. (DTA).

- A Operand = Type Hex D Control Structor: IDO is initiating a Single Control Command. (SCC).
- A Operand = Type Hex C Control Structor: IDO is initiating an Alternate Array. (ALT).

Each type will be separately described including the extraction of the IDO instruction in each case.

- Data Transfer Array: The A operand consist of an inplicit length, tagged double word structor, whose format is as shown in Figure 9-11.

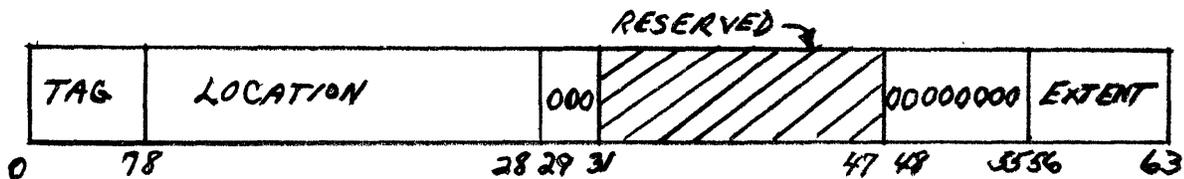


FIGURE 9-11. IMPLICIT LENGTH STRUCTOR FORMAT

The 24-bit LOCATION field will point to an array of I/O command structors or "chain" of I/O commands, and the extent of the array is limited to 256 commands (the high-order eight bits of the extent field must be zeros). The position field is not used.

The I/O command structor array is the actual list of the I/O operations or "I/O Program". An individual I/O Command structor is formatted as shown in Figure 9-3.

The complete set of facilities or resources needed to perform a Data Transfer operation will be checked and allocated during the extraction of the IDO instruction with this A operand, and they will remain busy during the execution of the a array. Those facilities are: a peripheral device, a level of simultaneity of the I/O subsystem and transfer rate.

Control commands can be included in the array, but it should be noted that resources such as transfer rate, will be tied up during the execution of these commands.

The extraction path for this type of A operand will be as follows: (The complete Central Processor extraction of a Data Transfer Array IDO order is shown in Figure 9-12).

As mentioned, the central processor extracts the Device Identifier Structor, and the entry in the Device Specification table is selected. The first byte of the entry is extracted and the actions indicated by the Flags are performed. If no trapping or rejection occurs, the transfer rate is extracted and the overrun protection is effected by using the Traffic Registers.

If the Traffic Register indicate a potential overrun condition, the order is rejected; if not, the level of simultaneity is checked and allocated.

If the simultaneity table indicates that all levels of simultaneity of the resource are busy, the order will be rejected (by setting the condition code register with the corresponding information and extracting the next order).

If all the preceding checks are satisfied, the I/O operand is assembled. The I/O operand is used by the peripheral subsystem in initiating the execution of the array. I/O operands are formatted as described in Paragraph 9-4. The location field is also stored in the Device Specification Table and the busy status bit in the flags field is set to one.

The execution of the IDO instruction terminates at this point, and the next central processor instruction in sequence is extracted.

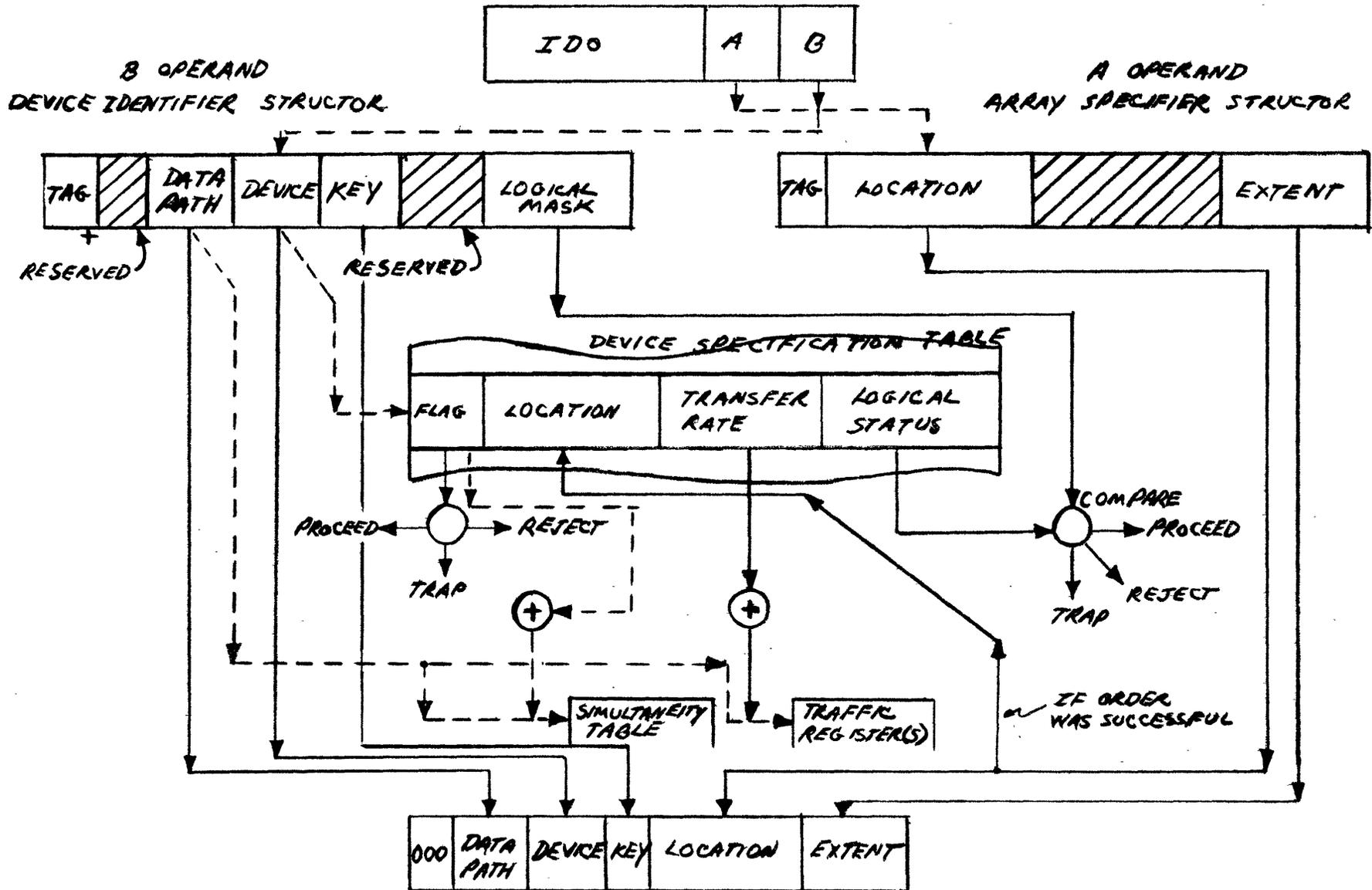


FIGURE 9-12. EXTRACTION OF AN IDO INSTRUCTION

The condition code register contents (In the Procedure Index), after the execution of an IDO order, is formatted as shown in Figure 9-13.

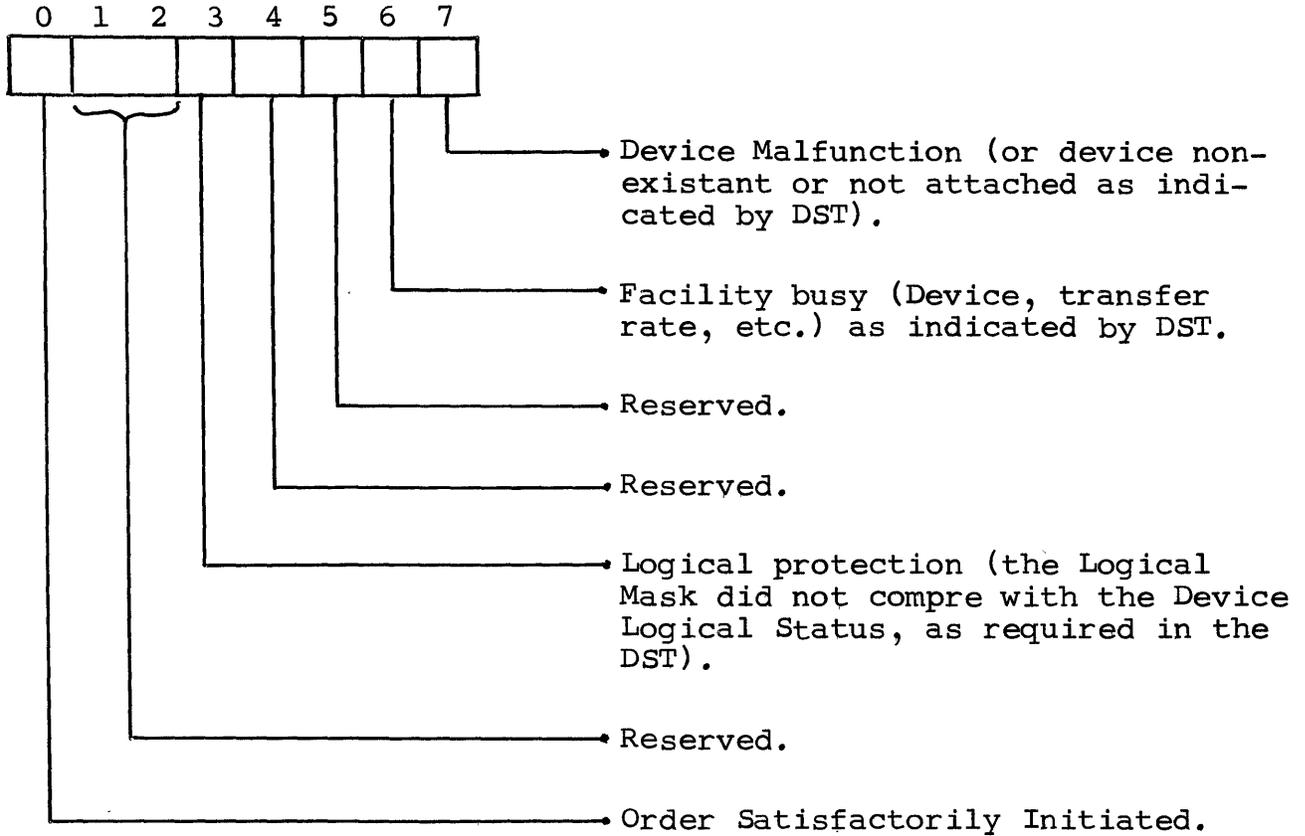


FIGURE 9-13. CONDITION CODE REGISTER FORMAT

- b. Single Control Command: The IDO instruction will specify the execution of a single control command if the A operand is a type Hex D structor, formatted as shown in Figure 9-14.

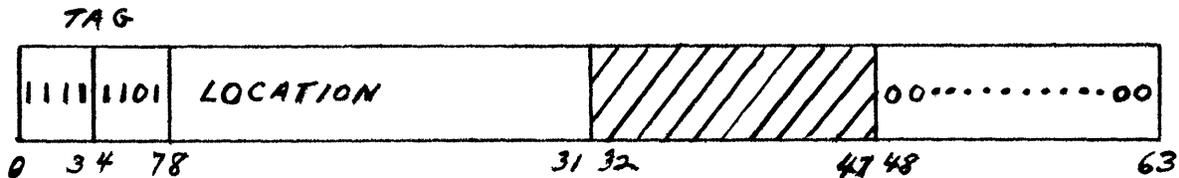


FIGURE 9-14. SINGLE CONTROL COMMAND SPECIFIER STRUCTOR

The 24-bit LOCATION field points to a single I/O command, which must specify a Control Operation. The extent field must be all zero, indicating an "array" of a single element. The position field is not used.

The extraction path is similar to the case in which the A operand was a Data Transfer array structor, with the exception that the simultaneity table check will not be made, if so specified in bit 4 of the Flags field in the Device Specification Table.

This bit, if a 1, indicates that the corresponding peripheral device is attached to facilities which can execute control commands, even in those cases in which all levels of simultaneity are busy.

An I/O operand will be generated for the I/O subsystem. A different I/O operand is used for this kind of operation because it will allow an early release of resources (transfer rate, level of simultaneity) before the actual completion of the I/O operation. All I/O operands are described in Paragraph 9-4.

After assembling the I/O operand and signalling the I/O subsystem, the execution of the IDO instruction terminates and the next central processor instruction in sequence is extracted.

The condition code contents in the Procedure Index, after execution of the IDO order, is similar to the previous case and it is shown in Figure 9-13.

- c. Alternate Array: The A operand consist of a type Hex C structor, which is formatted as shown in Figure 9-15.

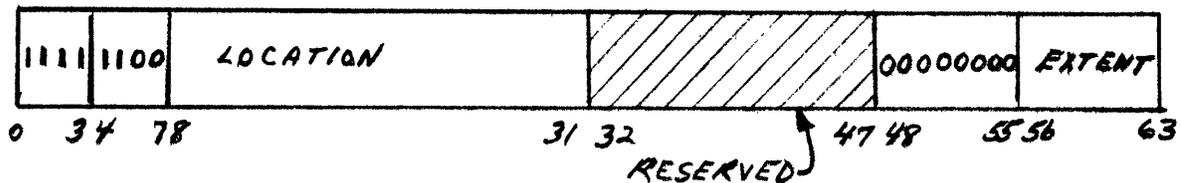


FIGURE 9-15. ALTERNATE ARRAY SPECIFIER STRUCTOR

The 24-bit LOCATION field points to an array of I/O commands similar in all respects to the Data Transfer Command array.

The difference is that an IDO with this type of A operand will be issued to a busy peripheral device. The device will stop execution of the previous array (even in those cases in which it is transferring data), it will proceed to execute the alternate array up to its completion, then resume execution of the previous array, without any loss of data or control information.

Not all the devices will be able to execute the alternate array, because it is required that the facilities to which the device is attached, be prepared to do so. Bit 3 in flags field in the Device Specification Table will indicate (if a 1), that the device is capable of executing an alternate array. This bit will be checked during the extraction of an IDO order with this type of A operand.

The extraction path is similar to the case in which the A operand specifies a Data Transfer Array, with the execution of action to take after the busy test on the device. If the device is busy, no transfer rate will be allocated (The I/O subsystem will stop execution of previous array, which used the same amount of transfer rate). If the device is not busy, the order will be automatically converted as if the A operand specified a Data Transfer Array.

I/O operands will be generated for the I/O subsystem. (All formats of I/O Operands are presented in Paragraph 9-4).

The execution of the IDO instruction terminates at this point, and the next central processor instruction in sequence is extracted.

The Condition Code in the Procedure Index, after execution of the IDO order, is similar to the previous case, (as shown in Figure 9-13), with the exception that the device busy bit can be set to one at the same time that indicating an order satisfactorily initiated.

9.3.2.2 The Halt Device Operation Instruction

The formats of the HDO instruction are shown in Figure 9-16.

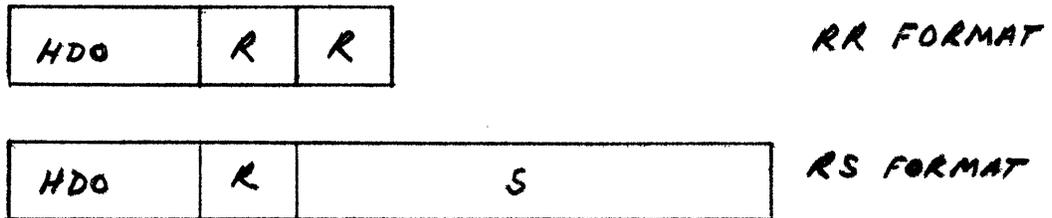


FIGURE 9-16. HDO INSTRUCTION FORMATS

(See Section III for details in Central Processor instruction formats).

The A operand is not used in an HDO order, and the contents of the register in the R syllable is not affected by its execution.

The B operand must be a Device Specifier Structor, as in the IDO instruction.

The HDO order will cause an abnormal termination to occur in the peripheral device specified in the B operand.

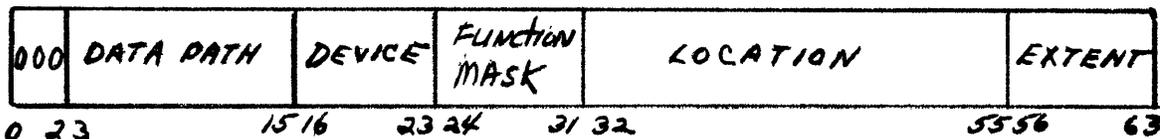
No access to the Device Specification Table will be performed. (The corresponding facilities are released on signal from the peripheral device.)

After the I/O operand is formatted, the execution of the HDO instruction terminates, and the next instruction in sequence is extracted.

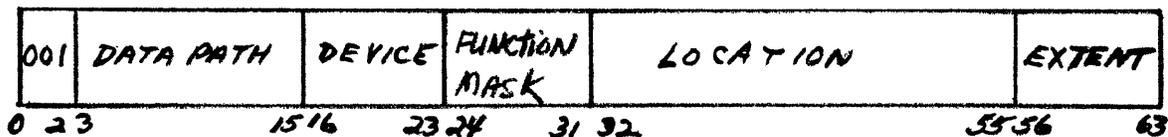
9.4 INPUT/OUTPUT OPERANDS

The Input/Operands are the result of the extraction of an I/O order, and they convey to the I/O subsystem the necessary information to execute the peripheral order. The formats of I/O operand and their corresponding codes are shown in Figure 9-17.

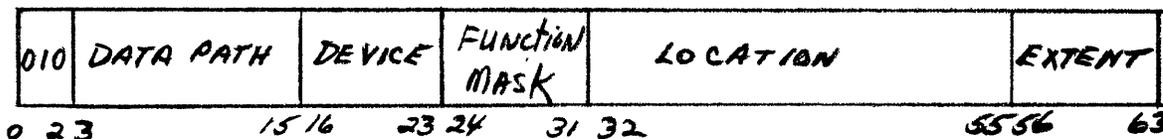
- a. I/O Operand indicating execution of a Data Transfer Array:



- b. I/O Operand indicating execution of a Single Control Command:



- c. I/O Operand indicating execution of an Alternate Array:



- d. I/O Operand indicating execution of a Halt Device Operation:

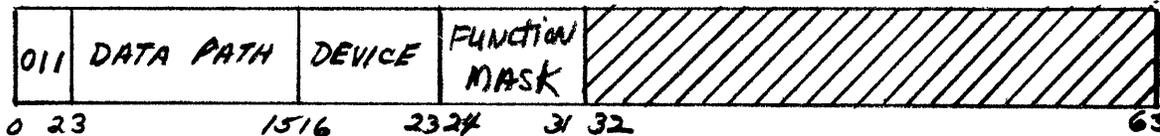


FIGURE 9-17. INPUT/OUTPUT OPERAND FORMATS

The LOCATION field points to an I/O command or array of commands.

The 4 low order bytes and the code of an I/O operand are assembled with information from the A operand of a Central Processor peripheral instruction. The high order 4 bytes contains information specified in the B operand of the same instruction.

9.5 SUMMARY OF EXTRACTION AND FORMATS

Table 9-1. contains a summary of all I/O formats, as well as the allocation features.

TABLE 9-1
VALID FORMATS RESULT OPERANDS AND
RESOURCE ALLOCATION IN I/O INSTRUCTIONS

	IDO Order (Data Transfer Array)	IDO Order (Single Control Command)	IDO Order (Alternate Array)	HDO Order
A Operand	Implicit length Double Word Structor	Type Hex D Control Structor	Type Hex C Control Structor	-
B Operand	Device Specifier Structor	Device Specifier Structor	Device Specifier Structor	Device Specifier Structor
Resulting I/O Operand	Code 0	Code 1	Code 2 (Device busy) Code 0 (Device not busy)	Code 3
Device Busy Check	Yes	Yes	Yes, but not rejection	No
Transfer Rate Allocation	Yes	Yes	No	No
Simultaneity Allocation	Yes	Conditional	Yes	No

9.6 INPUT/OUTPUT TERMINATIONS

The sequencing of I/O commands in the array will proceed until the last one in the array or "chain" is executed.

The last command is defined to be the command in which the end bit in its transfer variant is a one.

After execution of this command is completed, the I/O subsystem will send a normal termination signal, which will be directed by the Central Processor to a preassigned task.

There are other reasons for signalling the Central Processor, and they will be described in Paragraph 9.6.2.

9.6.1 The I/O Initiated Starts

The effect of any signal from the I/O subsystem is to start a prespecified task. The task to be started is selected on the basis of which device originated the signal and on the various types of signal available to the device and the I/O subsystem.

This hardware steering mechanism is implemented with the use of the I/O Start Array (See Section II). This array contains a potential entry representing a specific device, a specific type of signal (termination, residue storage, attention etc.) and a third level of indexing qualifying the type of start. Each I/O start will cause transfer of the I/O Status Word (See Paragraph 9.6.2). It contains the device address and signal code (indicating type of start) as well as the qualifier.

When the signal code of the I/O Status word indicates a termination condition, the resources (transfer rate, level of simultaneity, device) will be deallocated from the corresponding tables.

The execution of I/O initiated starts will be discussed in Paragraph 9.6.3.

9.6.2 The I/O Status Word

There are five main reasons for generating an I/O start (of a CP task):

a. Normal Termination

The termination of an I/O operation will cause an I/O interrupt. A normal termination for a data transfer operation is defined to occur at that point in time in which the data and status information for the last I/O command in the array or chain are available.

The normal termination of a control operation can indicate one of two things: 1) The device is ready to execute a new order. 2) Certain facilities can be used to initiate another I/O operation to a different device. (e.g., An I/O operation indicating a seek cylinder in a disk drive will be able to release transfer rate and the level of simultaneity very early, leaving in a busy status only the disk drive).

In the later case, the termination of the seek, or device not busy signal will be indicated with an attention signal.

b. Abnormal Termination

Meaning that the I/O operation could not be completed, or it was completed with an error.

c. Programmable Signal

During execution of an I/O command array, an I/O start is generated if so specified in the transfer variant, after execution of any operation in the chain.

d. Residue Storage

The transfer variant can specify that the residual count, or the difference between the length field in

9.6.2
(Cont.)

the order and the length of the physical record, be saved, if different from zero. A signal will be generated to effect this.

e. Attention Signals

An attention signal is defined to be all those signals coming from a device not presently associated with a level of simultaneity or an I/O command array.

Each type of I/O Status Word will have an assigned code, specifying the type of signals. The IOSW's are formatted as shown in Figure 9-18.

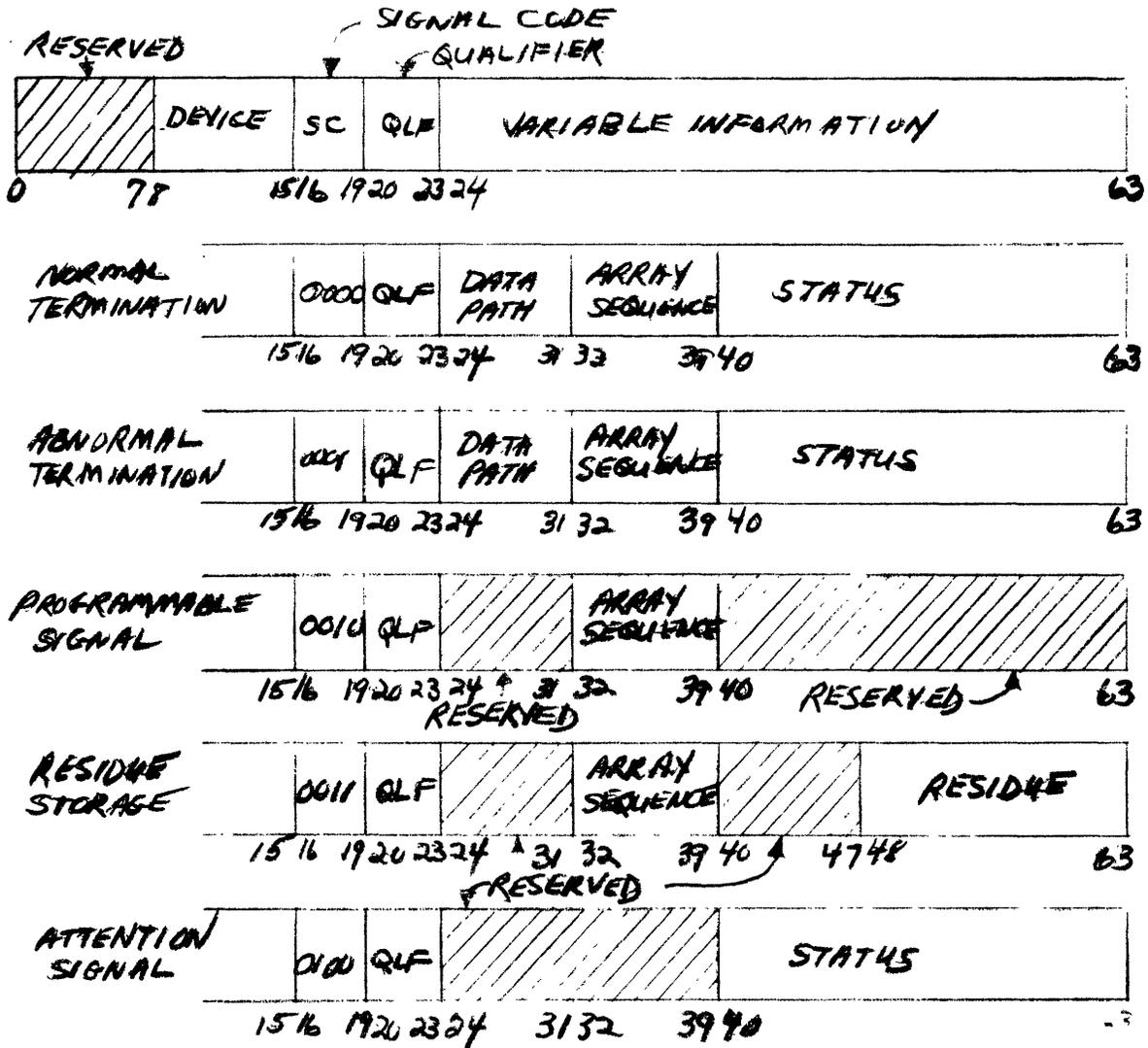


FIGURE 9-18. I/O STATUS WORD FORMATS

Bits 0-7, shown as reserved in Figure 10-18, includes de-allocation information, and it will represent which type of I/O order initiated the I/O command array.

The 4-bit QUALIFIER field will allow the peripheral device to selectively start one of up to 16 different task per each condition. (e.g., several types of attention signals).

The 8-bit DATA PATH field includes part of the routine information used in the 13-bit Data Path field associated with the I/O instruction. It is to be used to specify which resources to deallocate.

The 8-bit ARRAY SEQUENCE field, indicates the position in the array of the I/O command which caused the signal.

The 24-bit STATUS field will include pertinent status information (not yet defined).

The 16-bit RESIDUE field, will include the residual count or the difference between the length of the buffer and the length of the physical record. If the physical record was longer than the main memory buffer, it will include all zeros.

The I/O status word is stored in a queue pointed by an entry in the system base, and then moved to the location specified in the I/O start array.

9.6.3 Execution of I/O Initiated Starts

The central processor servicing the interrupt will fetch the IOSW, and sense if the cause of the interrupt is an I/O command array termination.

If the IOSW indicates a termination, normal or abnormal, of an array, the I/O subsystem will proceed to deallocate the resources used in the execution of the array, as described in Paragraph 9.6.4.

9.6.3
(Cont.)

The I/O start array entry in the System Base will indicate in which location the IOSW should be stored.

The Central Processor will sequence through the I/O start array in the following way, depending on the structors stored in the corresponding entries in the array.

The first entry can indicate one of three conditions:

- a. Ignore all I/O starts (null entry).
- b. Start a particular task.
- c. Index with device number.

If the last condition is found, the device number in the IOSW is used to extract a new entry which can specify:

- a. Ignore I/O starts to specified device (null entry).
- b. Start a particular task.
- c. Store the IOSW in a particular queue and then start a particular task.
- d. Index with signal code.

If the last condition occurs, the signal code in the IOSW is used to extract a new entry which will specify:

- a. Ignore I/O starts specifying a particular condition (null entry)
- b. Start a particular task.
- c. Store the IOSW in a particular queue and then start a particular task.
- d. Index with qualifier code.

If the last condition occurs, the qualifier code is used to extract a new entry, which will again specify a null entry, point to a task, or point to a queue and a task.

9.6.3
(Cont.)

The I/O Start will cause in most cases the change of a task status from BLOCKED to READY, and if the task is of higher priority than the task presently running, it will be swapped and its status changed to RUNNING.

If the task was already in the running state, a flag will be set in the task status block.

A complete description of the start operation appears in Section 7 (Task Multiplexing).

.6.4 Termination Conditions

Terminations can be classified in the following ways:

- a. According to the checking results in:
 1. Normal Termination
 2. Abnormal Termination
- b. According with the nature of the last or single command:
 1. Termination of data transfer.
 2. Termination of Control Operation.
- c. According with the original CP I/O instruction.
 1. Order originated in a Data Transfer Array IDO.
 2. Order originated in a Single Control Command IDO.
 3. Order originated in an Alternate Array IDO.
- d. According to which facilities should be released:
 1. Termination of transfer rate usage, level of simultaneity, but device still busy.
 2. Termination of device.
 3. Simultaneous termination of all facilities.

Those termination conditions can occur in many different combinations. (e.g., Normal termination of a control operation of an Alternate Array in which device is still busy; abnormal termination of Control Operation initiated by a Single Control Command IDO, in which all facilities simultaneously terminated, etc.).

9.6.3
(Cont.)

The releasing of facilities, during execution of the I/O start, takes in account all these different conditions. All possible combinations are presented in Table 9-2.

The entries in that table marked as CONDITIONAL refers to the fact that during initiation of an IDO order pointing to a Single Control Command, the level of simultaneity may or may not have been allocated (depending in bit 4 in the Device Specification Table). The deallocation mechanism should do the inverse operation.

Entries marked as NOT APPLICABLE refers to the fact that IDO executing Alternate Array were directed to a busy device. Consequently, termination of the array should not release the device in any case. If an abnormal situation occurs, such as a recoverable error, the device should abnormally terminate both arrays, the alternate and the original, sending two I/O starts. The second one will indicate release of the device.

TABLE 9-2
SUMMARY OF TERMINATION CONDITIONS

		Operations initiated with IDO-Data Transfer Array			Operations initiated with IDO-Single Control Channel			Operations initiated with IDO-Alternate Array		
		Transfer Rate Deallocation	Level of Simultaneity Deallocation	Device Deallocation	Transfer Rate Deallocation	Level of Simultaneity Deallocation	Device Deallocation	Transfer Rate Deallocation	Level of Simultaneity Deallocation	Device Deallocation
NORMAL TERMINATION	Level of Simultaneity End-Device still busy	Yes	Yes	No	Yes	Conditional	No	Not Applicable		
	Level of Simultaneity End-Device End	Yes	Yes	Yes	Yes	Conditional	Yes	No	Yes	No
ABNORMAL TERMINATION	Level of Simultaneity End-Device still busy	Yes	Yes	No	Yes	Conditional	No	Not Applicable		
	Level of Simultaneity End-Device End	Yes	Yes	Yes	Yes	Conditional	Yes	No	Yes	No
ATTENTION SIGNAL	Device End	No	No	Yes	No	No	Yes	Not Applicable		

ADVANCED COMPUTER SYSTEM DOCUMENT CHANGE NOTICE

DOC. NO. FTL-003 REV. DRAFT 2
TITLE FUNCTIONAL SPECIFICATION FOR LEVEL 2 COMPUTATIONAL
PROCESSES
PREPARED BY R. Keys CHANGE NOTICE NO. 4
E. McFaden DATE 2/13/70

Insert attached pages A-1 through A-3 into the subject document after Section IX.

REF: 54A

HONEYWELL PROPRIETARY

APPENDIX A
DECIMAL STRING ENCODINGS

A.1 SIGN CODES

The rightmost four-bit field of a signed packed decimal string and the zone portion of the rightmost byte of a signed zoned decimal string contain a sign code. This code specifies the sign of the value and is associated with one of four classes. The code that is generated for a result (destination) string is determined by the sign of the result value together with the class of the sign code that it replaces. Table A-1 specifies the interpretation of the sign code.

TABLE A-1
INTERPRETATION OF SIGN CODE

CODE	SIGN	CLASS
0000	+	0
0001	+	0 (preferred S-200 encoding)
0010	-	0 (preferred S-200, 1400 encoding)
0011	+	0 (preferred 1400 encoding)
0100	+	1*
0101	+	1*
0110	+	1*
0111	+	1*
1000	+	1*
1001	+	1*
1010	+	2 (preferred S360 ASCII encoding)

*If any sign code in class 1 is used, an illegal operand trap (0201) will be generated or masked following instruction execution.

(Continued)

Table A-1 (Cont.)

CODE	SIGN	CLASS
1011	-	2 (preferred S360 ASCII encoding)
1100	+	3 (preferred S360 EBCDIC encoding)
1101	-	3 (preferred S360 EBCDIC encoding)
1110	+	3
1111	+	3

Table A-2 specifies the sign code that is generated for a given sign and class:

TABLE A-2
SIGN CODE FOR GIVEN SIGN AND CLASS

SIGN	CLASS	CODE
+	0	0001
-	0	0010
+	1	0100
-	1	0100
+	2	1010
-	2	1011
+	3	1100
-	3	1101

Facilities for sign decoding/encoding must be capable of alteration to accept future sign code conventions.

A.2 DIGIT CODES

Each four-bit digit field of packed and zoned decimal strings contains an encoding for a decimal digit according to Table A-3.

HONEYWELL PROPRIETARY

TABLE A-3
INTERPRETATION OF DIGIT CODE

SOURCE OPERAND CODE	INTERPRETATION	RESULT OPERAND CODE
0000	0	0000
0001	1	0001
0010	2	0010
0011	3	0011
0100	4	0100
0101	5	0101
0110	6	0110
0111	7	0111
1000	8	1000
1001	9	1001
1010*	0	0000
1011*	0	0000
1100*	0	0000
1101*	0	0000
1110*	0	0000
1111*	0	0000

*Use of these digit codes in a decimal instruction operand will cause an illegal operand trap (0201) to be generated or masked, following instruction execution.

