PART 2. STORAGE AND SYMBOL DEFINITIONS

5. STORAGE DEFINITIONS

6. SYMBOL DEFINITIONS

PART 3. BAL APPLICATION INSTRUCTIONS

7. INTRODUCTION TO APPLICATION INSTRUCTIONS

8. BRANCHING INSTRUCTIONS

9. DECIMAL AND LOGICAL INSTRUCTIONS

14. LIST PROCESSING

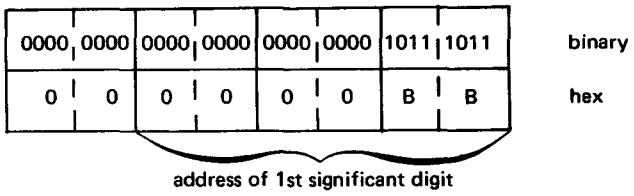
USER COMMENT SHEET

INDEX

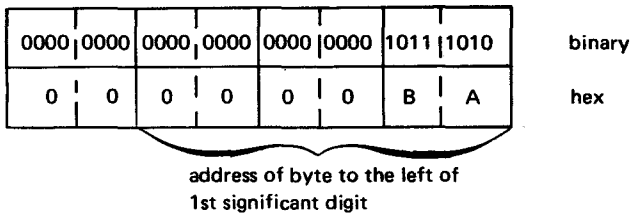
FIGURES

TABLES

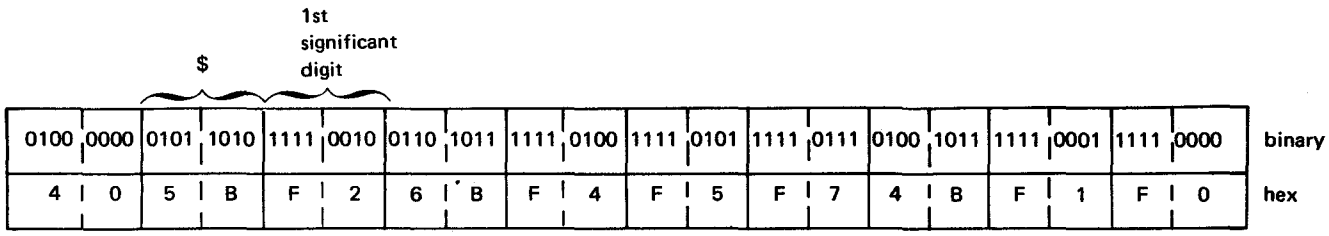
Register 1 after execution of EDMK instruction:



Register 1 after execution of S instruction:



Edited result after execution of MVI instruction:



In this example, the edit mask is moved into a 10-byte field labeled PATTERN. The address of the position where the insert character is to be placed (in the absence of significant digits before the significance starter) is loaded into register 1. Then DATA, containing the packed number, is edited and the result is placed in PATTERN. The address of the first significant byte (in this example, 2 is significant) replaces the content of register 1. Then a full word containing the decimal value 1 is subtracted from the content of register 1, therefore moving one byte to the left. The MVI instruction moves the dollar sign into the byte addressed by the content of register 1.

MSS

9.8. MODIFY STORAGE AND SKIP (MSS)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MSS	E3	SS	6

Condition Codes

- ■ SET TO 0
- ■ SET TO 1
- ■ SET TO 2
- ■ SET TO 3
- □ UNCHANGED

Possible Program Exceptions

- ■ ADDRESSING
- □ DATA (INVALID SIGN/DIGIT)
- □ DECIMAL DIVIDE
- □ DECIMAL OVERFLOW
- □ EXECUTE
- □ EXPONENT OVERFLOW
- □ EXPONENT UNDERFLOW
- □ FIXED-POINT DIVIDE
- □ FIXED-POINT OVERFLOW
- □ FLOATING-POINT DIVIDE
- ■ OPERATION
- ■ PROTECTION
- □ SIGNIFICANCE
- ■ SPECIFICATION:
  - □ NOT A FLOATING-POINT REGISTER
  - □ OP 1 NOT ON HALF-WORD BOUNDARY
  - □ OP 2 NOT ON HALF-WORD BOUNDARY
  - □ OP 2 NOT ON FULL-WORD BOUNDARY
  - ■ OP 2 NOT ON DOUBLE-WORD BOUNDARY
  - □ OP 1 NOT EVEN NUMBERED REGISTER
  - □ OP 1 NOT ODD NUMBERED REGISTER
- ■ SEE OPERATIONAL CONSIDERATIONS
- □ NONE

The modify storage and skip (MSS) instruction performs an operation that you specify by immediate operand 1 (i_1) on two operands indirectly specified by main storage operand 2. Depending on the result, program control may then pass to the next sequential instruction or to another location, called the skip location, which is offset from the instruction following the MSS instruction by a displacement value you specify in immediate operand 3 (i_3). You can put the result of the operation in the main storage location or register specified by operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MSS	$d_1(i_1,b_1),d_2(i_3,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MSS	$s_1(i_1),s_2(i_3)$

The i_1 and i_3 fields shown in the previous two formats roughly correspond to the l_1 and l_3 fields of other SS-type instructions. The i_1 value is assembled, unchanged in value, into bits 8—11 of the MSS object instruction and the i_3 value is assembled, likewise unchanged in value, into bits 12—15. The rest of the operand fields are assembled according to the rules for SS-type instructions.

9.8.1. What the Instruction Can Do

The MSS instruction can:

1. perform an arithmetic, data movement, or logical operation on two source operands, either of which can be in main storage or a register;

2. depending on the operation result, branch either to the next sequential instruction or go to the skip location as determined by the i_3 operand;

3. optionally store the operation result to a destination operand, either in main storage or a register, specified by operand 1;

4. optionally modify up to two additional registers each time the operation is performed; and

5. repeat steps 1 through 4.

At its simplest, the MSS instruction proceeds as shown in Figure 9—1.

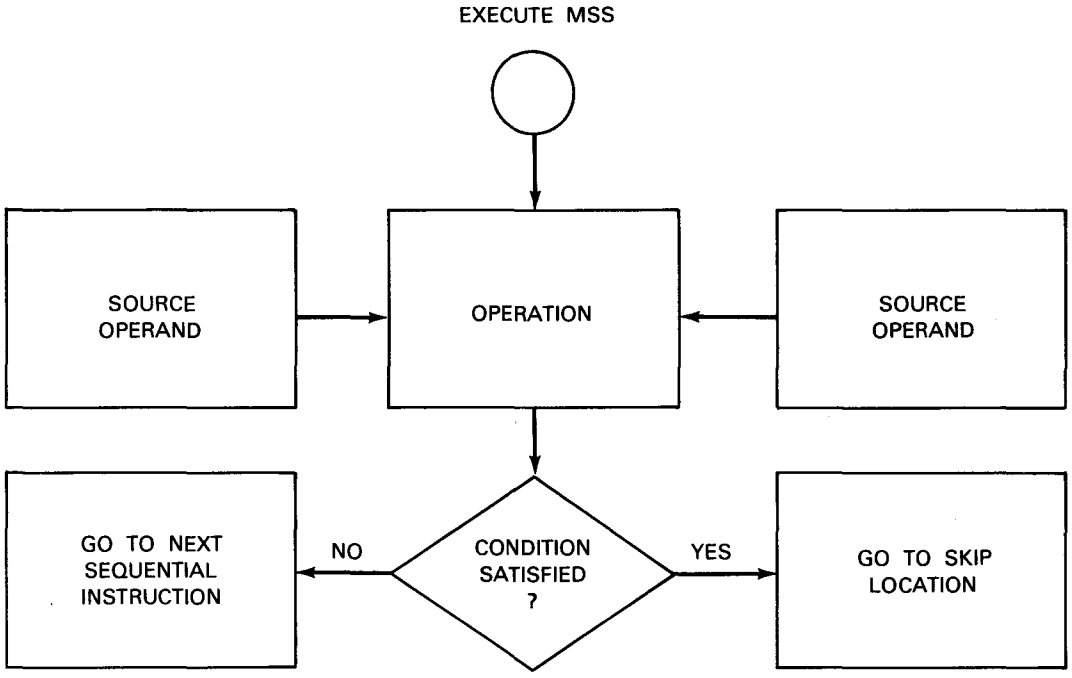


Figure 9—1. Basic MSS Execution

As Figure 9—1 shows, the MSS instruction fetches two operands and performs an operation on them that you specify by operand i_1. The operations available to you with MSS are shown in Table 9—2. Each operation tests for a condition. If that condition is not met, program control passes to the next sequential instruction after MSS. If the condition is met, however, program control skips forward a number of half words beyond the next sequential instruction, to the skip location, and continues with the instruction found there. The number of half words skipped is given by the absolute value i_3 and it can range from 0 to 15 half words (30 bytes).

Table 9—2. MSS Operations

i_1 Value	Mnemonic	Description
0	ADDZ	Add and compare for zero result
1	ADDNZ	Add and compare for nonzero result
2	SUBZ	Subtract and compare for zero result
3	SUBNZ	Subtract and compare for nonzero result
4	MCE	Move and compare for equal result
5	MCNE	Move and compare for unequal result
6	MCLE	Move and compare for less than or equal result
7	MCH	Move and compare for greater than result
8	ANDZ	AND operands and test for zero result
9	ANDNZ	AND operands and test for nonzero result
A	XORZ	XOR operands and test for zero result
B	XORNZ	XOR operands and test for nonzero result
C	ORZ	OR operands and test for zero result
D	ORNZ	OR operands and test for nonzero result

As you can see, each of the operations in Table 9—2 tests its result for a certain condition. Program flow depends on whether the result does or does not satisfy that condition. If the condition is satisfied, the i_3 displacement is added to the current program status word (PSW), in effect causing a branch to the resulting location. If the condition is not met, program control passes to the next sequential instruction. What the conditions are and how they are met is described in 9.8.4.

Besides the basic functions described so far, you can optionally put the operation result in a destination operand as shown in Figure 9—2.

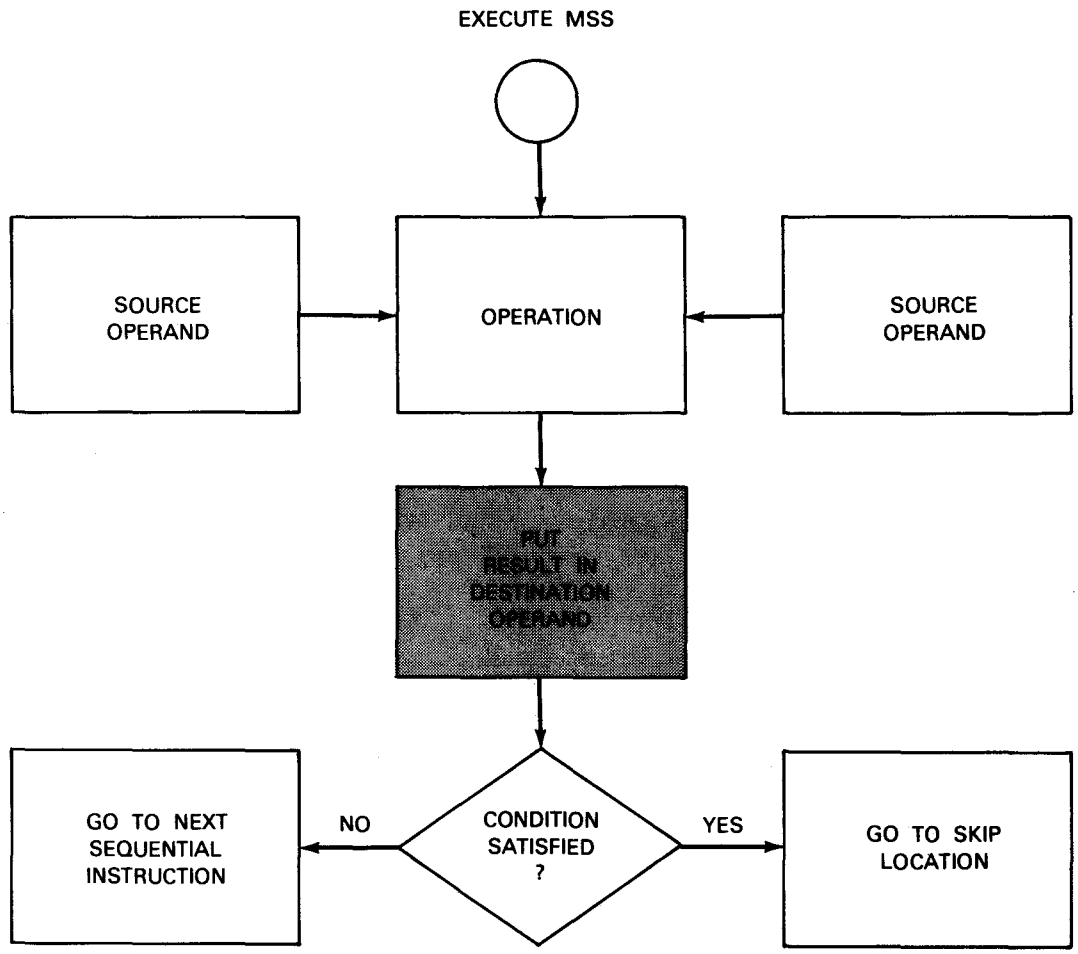


Figure 9—2. MSS Execution with Destination Feature

The destination operand of Figure 9—2 can be a register or a location in main storage. In addition to this feature, you can modify one or two additional registers (Figure 9—3):

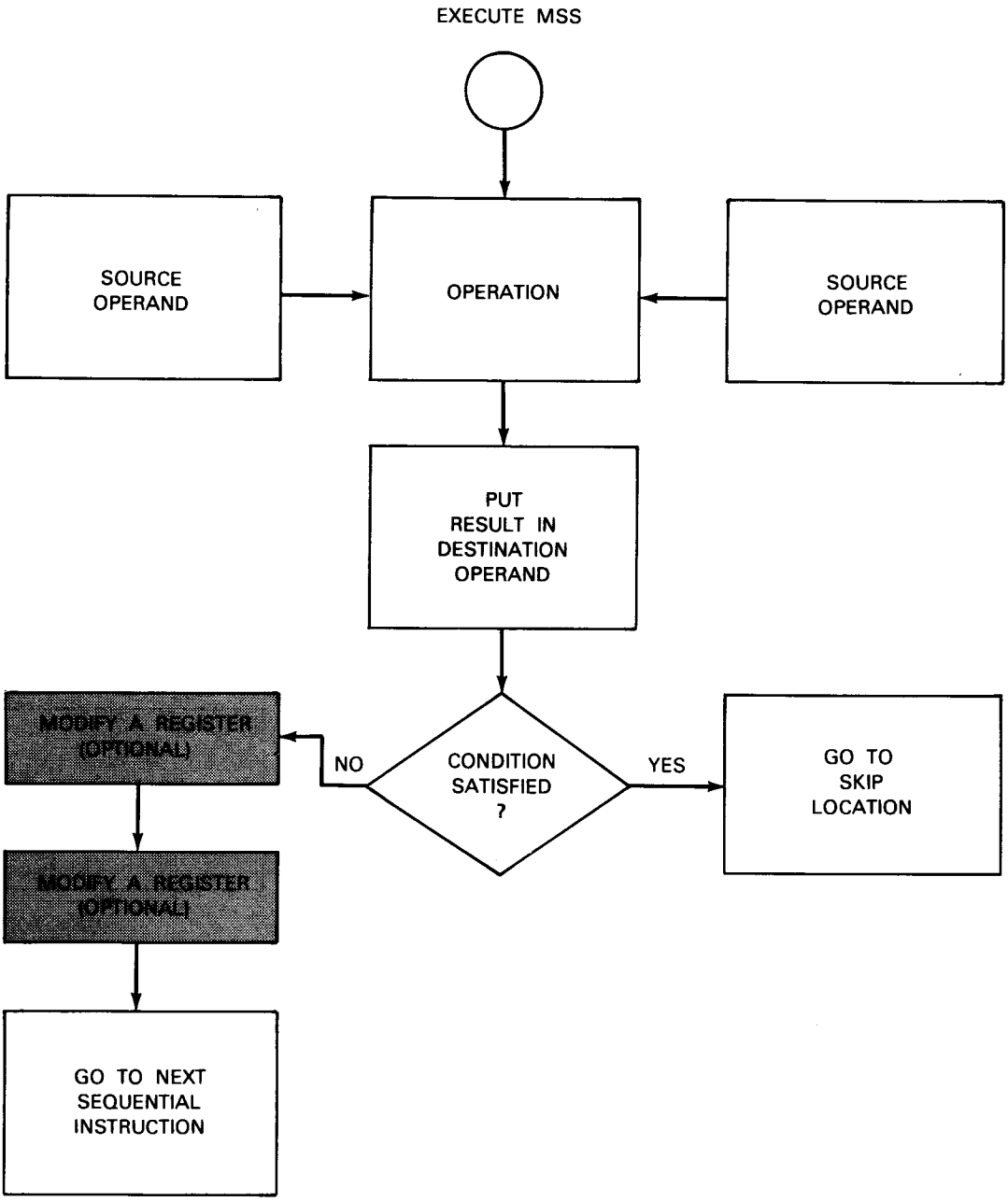


Figure 9—3. MSS Execution with Register Modification Feature

Finally, you have the option of repeating the MSS operation as shown in Figure 9—4.

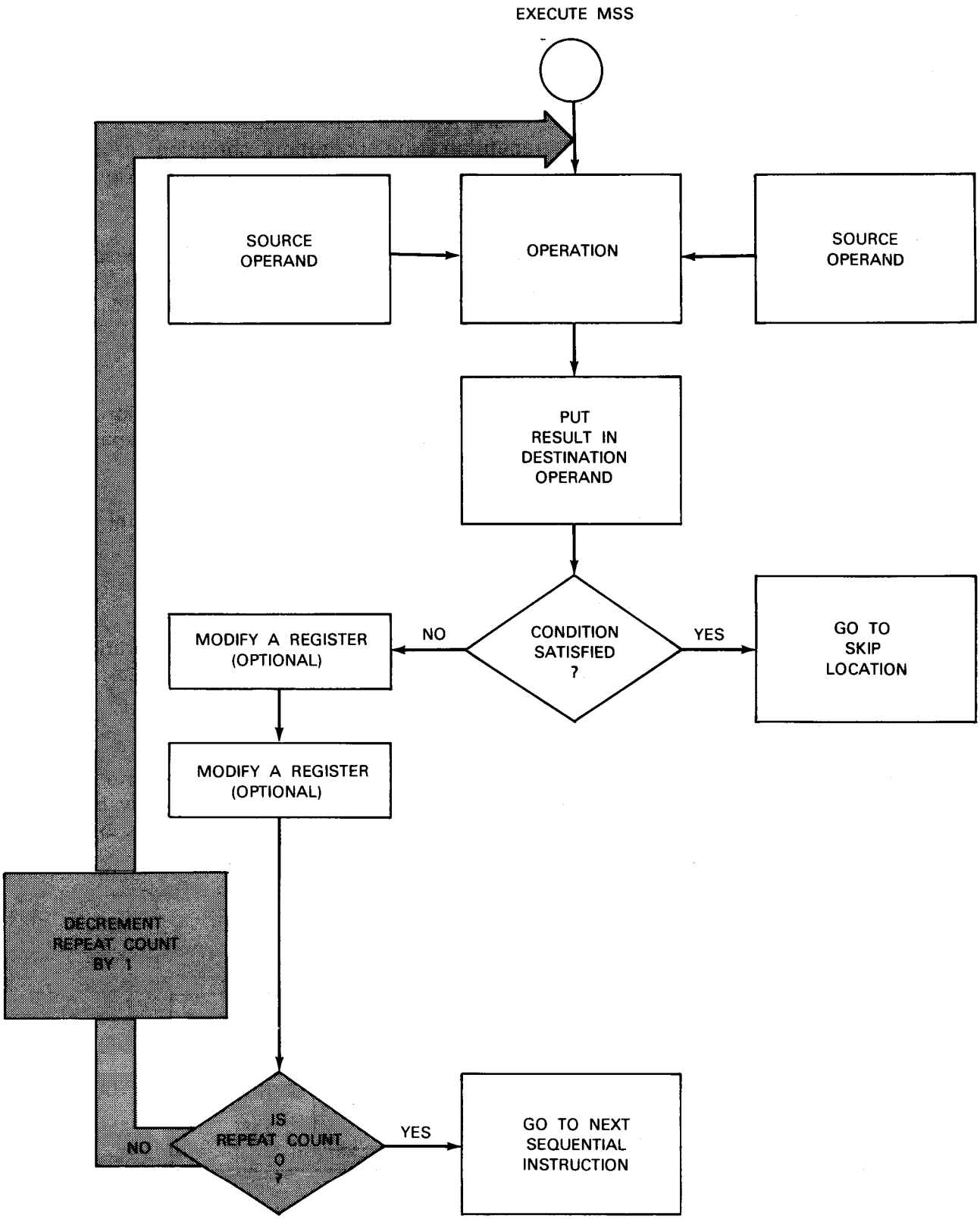


Figure 9—4. MSS Execution with Repeat Feature

If the operation condition is satisfied, the instruction skips to the main storage location specified by i_3. If the condition is not satisfied, the instruction will decrement the repeat count (whose initial value you supply) by 1. If after this the count remains nonzero, control returns to the MSS operation and the process is repeated using the updated data in the operands. If the count reaches zero, control passes to the next sequential instruction. If during a repetition, the operation condition is satisfied, regardless of the repeat count value, program control passes to the skip location specified by i_3.

You can code an MSS instruction using any combination of the features described in Figures 9—2 through 9—4.

9.8.2. What Operands You Supply

This section describes the format of the data you use in the MSS instruction. Exactly how you specify this data is described in 9.8.3; what you see here is an expansion of the MSS operations and capabilities previously described.

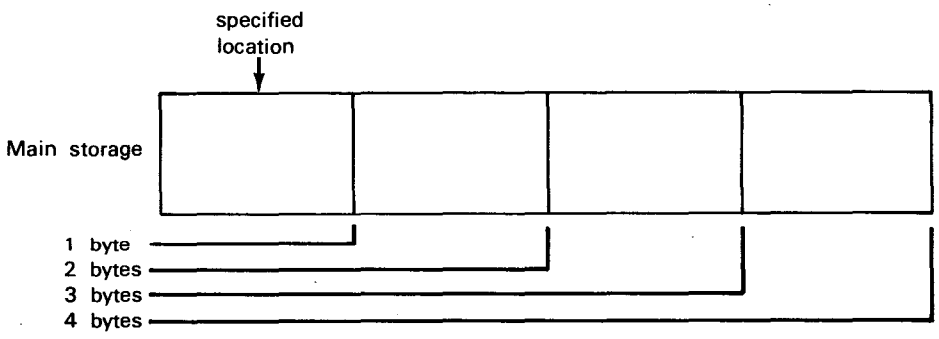
At a minimum, you need to specify the MSS operation, its two operands, and the skip location (Figure 9—1).

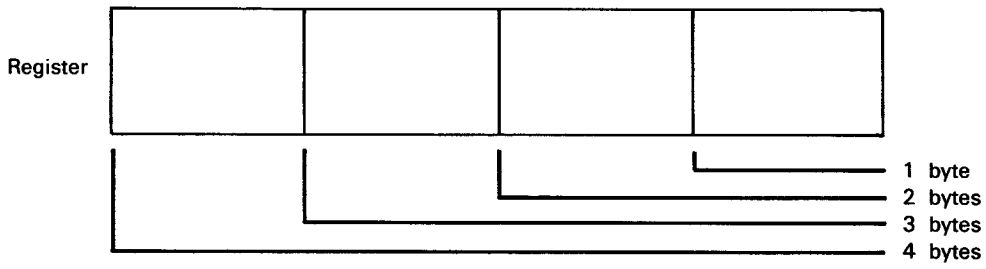
- Operation

  Fourteen operations are available as described in Table 9—2. All operations use all the bits in each operand. They are descibed in more detail in 9.8.4.

- Operands

  The MSS instruction always uses two source operands: one named the primary source data (PSD), and the other the secondary source data (SSD). They can both reside in main storage, both in registers, or one in main storage and one in a register. Both operands are equal in length, and can be one, two, three, or four bytes long. The operands are aligned as follows:





Multibyte MSS operands in main storage occupy contiguous bytes starting at the specified location and no half-word or full-word boundary alignment is necessary. In a register, the low order byte of an operand must always be aligned with the low order eight bits of the register. The two source operands may overlap.

- Skip Location

  Specified by i_3, this half byte of data represents the number of half words by which program control skips if the operation condition is met. Up to 15 half words (30 bytes) can be specified, and all skips must be in a forward direction.

If you want to keep the result of your MSS operation (Figure 9—2), you need to specify a destination data (DD) operand. You do this using the operand 1 address of your MSS source instruction. The operand can be stored in main storage, at the operand 1 address, or it can be put in a register specified by the b_1 field (in which case the d_1 value is ignored). The DD length is equal to that of the two source operands and its alignment within main storage or a register follows the same rules that they do. It can, in fact, overlap one or both of the source operands.

If you want to modify additional registers, you can specify up to two of them. For each one, you may also need to specify a register modification value (RMV), a 16-bit integer that is added to or subtracted from its modified register.

If you want to repeat the MSS operation, you need to specify an initial value for the repeat count. The count can reside in main storage, in which case it is 8 bits long, or it can occupy 32 bits in a register.

9.8.3. How You Specify Your Operands

To complement the operands of the MSS source instruction, you use a 4-word area in main storage to specify exactly how you want the instruction to execute. The area, which you must align on a double-word boundary, is addressed by operand 2. Fields within the operand 2 area specify the source operands, if and how the MSS operation is to repeat, if and how registers are to be modified, and so on. The basic format of operand 2 is shown in Figure 9—5.

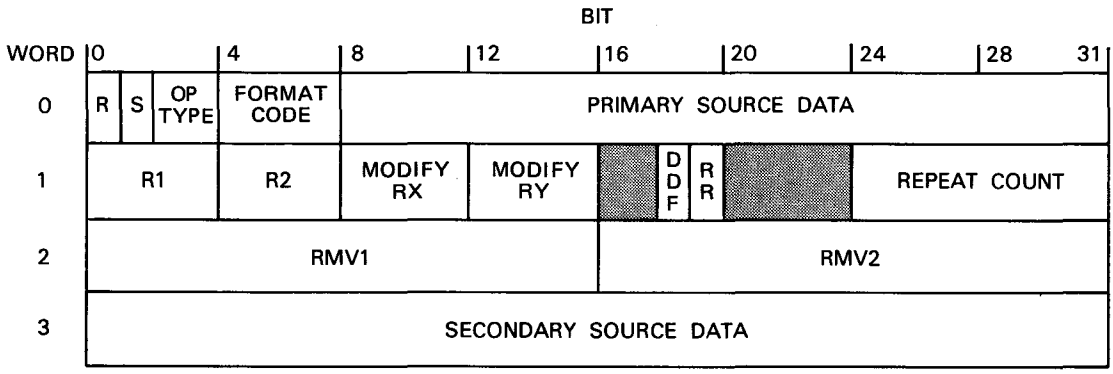


Figure 9—5. Operand 2 Format

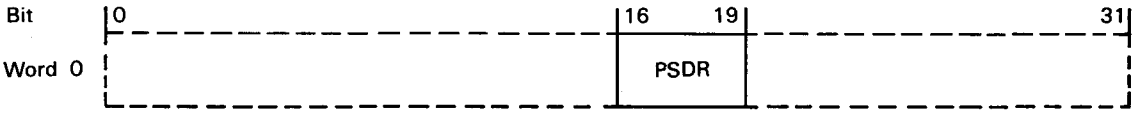
9.8.3.1. Specifying Basic Operands

To use the simplest form of the MSS instruction (Figure 9—1) you need to specify the source operands as follows:

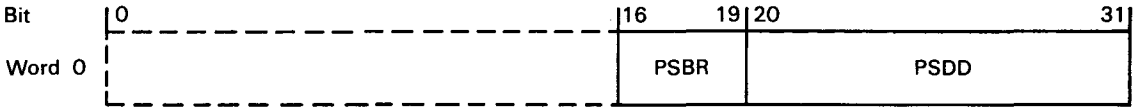
- Primary Source Operand

  You can specify the primary source operand using word 0 in one of three ways:

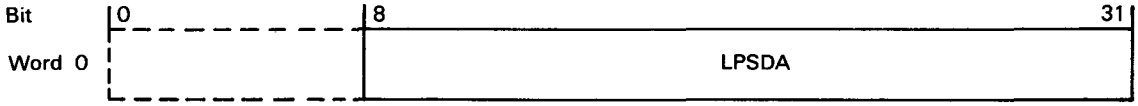
  1. Within a register, called the primary source data register (PSDR), specified in word 0:



  2. At a main storage address given in RX form by the primary source base register (PSBR) and the primary source data displacement (PSDD):



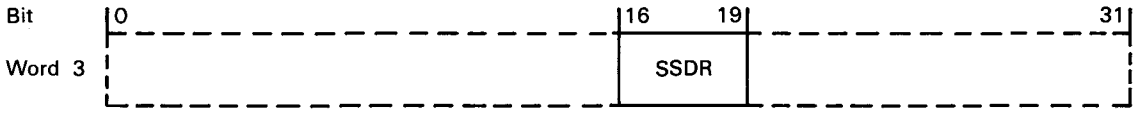
  3. Or, at a main storage location whose 24-bit address is contained in the logical primary source data address (LPSDA):



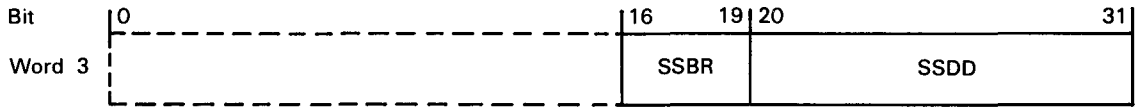
- Secondary Source Operand

You can specify the secondary source data using word 3 in one of four ways:

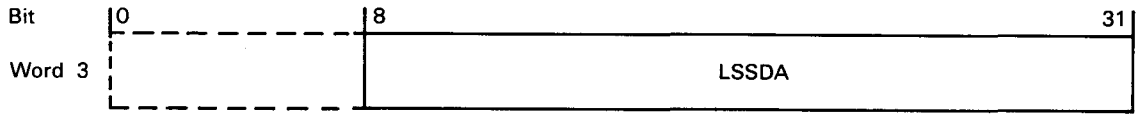
1. Within a register, called the secondary source data register (SSDR), specified in word 3:



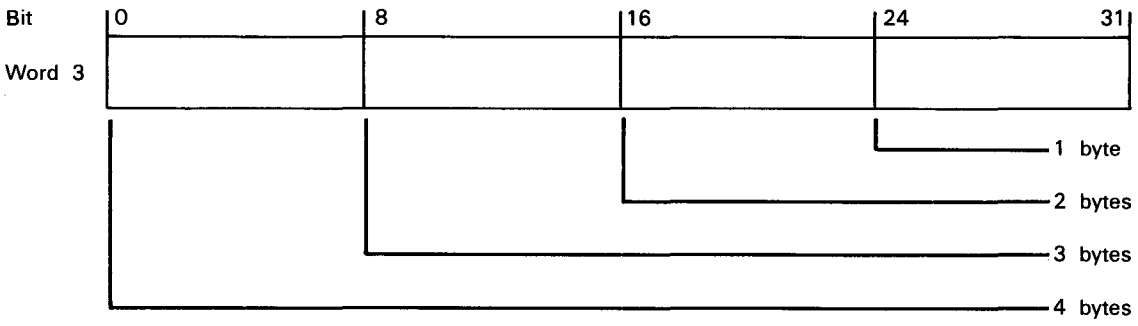
2. At a main storage address given in RX form by the secondary source base register (SSBR) and the secondary source data displacement (SSDD):



3. At a main storage location whose 24-bit address is contained in the logical secondary source data address (LSSDA):



4. Or, as immediate data, from one to four bytes in length:



Note that byte alignment for PSD immediate data follows the same rules as alignment within a register.

- Format Code

Because you can specify the PSD in one of 3 ways and the SSD in one of 4 ways, this results in 12 possible format combinations. You must, therefore, specify to the MSS instruction which one of these 12 combinations it is to operate with. You do this with the format code, bits 4—7 of word 0. Its possible settings are shown in Table 9—3.

Table 9—3. Format Code Values for Operand Types

Format Code (Hex)	PSD Type	SSD Type
0	Logical	Base/displacement
1	Logical	Logical
2	Logical	Immediate
3	Logical	Register
4	Register	Base/displacement
5	Register	Logical
6	Register	Immediate
7	Register	Register
8	Base/displacement	Base/displacement
9	Base/displacement	Logical
A	Base/displacement	Immediate
B	Base/displacement	Register

- Operand Type (Op Type)

  You use this field, located in bits 2 and 3 of word 0, to specify the length of the source operands with which the MSS instruction is to operate. If you specify a destination operand, this field specifies its length also. Its possible values are as listed in Table 9—4.

Table 9—4. Op Type Values

Binary Value	Operand Length (bytes)
00	1
01	2
10	3
11	4

9.8.3.2. Specifying Destination Data Operands

To store the result of an MSS operation, as in Figure 9—2, you need to use the following operands:

- Store Indicator (S)

  Located in word 0, bit 1, this 1-bit field lets you specify whether or not the operation result is to be stored in the destination operand. You put a 1 in this field to indicate storage, or a zero to suppress storage.

- Destination Data Format (DDF)

  You use this 1-bit field in word 1, bit 18 to indicate whether the operation result is to be put in a register or in main storage. You specify a zero to indicate result storage in the main storage location specified by the operand 1 base/displacement address. You put a 1 in this field to indicate that the result is to go directly in the register specified by b_1 in operand 1 (ignoring the d_1 field). Regardless of the setting of this field, no storage occurs if the store indicator is set to 0.

9.8.3.3. Specifying Register Modification Operands

You can specify one or two additional registers to be modified as in Figure 9—3. Each of these registers, called register X (RX) and register Y (RY), can be specified in the operand 2 area as a source operand base register or as a separate register. In operation these registers are modified once for each execution of the MSS operation that does not meet the operation's conditions. You can specify the initial values of RX and RY. You can specify the values used to modify these registers with one or two register modification value (RMV) fields. Finally, you can use the modify register (MX/MY) fields to specify exactly how modification is to take place.

- Primary Source Base Register (PSBR)

  This register, discussed earlier, can be modified using the MODIFY and RMV fields. You can use this feature to modify the effective address of the primary source field, causing it to address successive areas of main storage for each execution of the MSS operation. Base register 0 always has a zero value for address computation purposes, but the other 15 registers can be used for address modification.

- Secondary Source Base Register (SSBR)

  This register can be modified in the same way as the PSBR described earlier.

- Modification Register 1 (R1)

  You can specify that a register other than a source base register be modified. You do so by putting the register number in this 4-bit field at word 1, bits 0—3.

- Modification Register 2 (R2)

  You can specify that a second register other than a source base register be modified. You do so by putting the register number in this 4-bit field at word 1, bits 4—7.

- Register Modification Value 1 (RMV1)

  You can modify RX or RY by adding or subtracting this 16-bit field contained in word 2, bits 0—15.

- Register Modification Value 2 (RMV2)

  In addition to RMV1, you can specify a second 16-bit field to be added to or subtracted from RX or RY. This field is contained in word 2, bits 16—31.

- Format Code

  In addition to specifying source data formats, the format code specifies which two registers are to be modified, according to Table 9—5.

Table 9—5. Format Code Values for Register Modification

Format Code	Register X	Register Y
0	R1	SSBR
1	R1	R2
2	R1	R2
3	R1	R2
4	R1	SSBR
5	R1	R2
6	R1	R2
7	R1	R2
8	PSBR	SSBR
9	PSBR	R2
A	PSBR	R2
B	PSBR	R2

- Modify Register X (MODIFY RX)

  This 4-bit field in word 1, bits 8—11, permits you to specify exactly how register X is to be modified, according to Table 9—6.

Table 9—6. MX Values

MX Value	Register Modification
0	No modification
1	Increment RX by 1
2	Decrement RX by 1
3	Add RMV1 value to RX
4	Subtract RMV1 value from RX
5	Add RMV2 value to RX
6	Subtract RMV2 value from RX
7—F	Not used

- Modify Register Y (MODIFY RY)

  You use this 4-bit field at word 1, bits 12—15, to specify exactly how RY is to be modified. Register Y modifications and their values are the same as the MODIFY RX values given in Table 9—6.

  Note that you can modify one register alone if you want: simply enter 0 for the MODIFY field of the other register. Entering 0 in both MODIFY fields prevents any register modification from taking place.

9.8.3.4. Specifying Repeat Operands

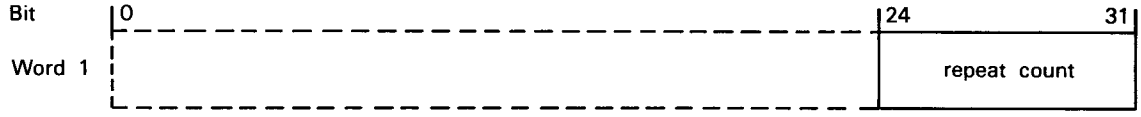
To repeat your MSS operation (Figure 9—4), you must specify the following fields:

- Repeat Indicator (R)

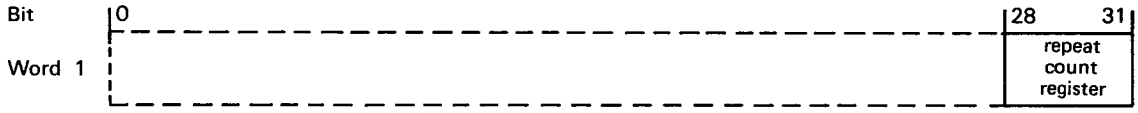
  This 1-bit field in word 0, bit 0 of the operand 2 area indicates whether or not the MSS instruction is to repeat its specified operation. Coding a 0 in this field causes the MSS operation to execute exactly once, then terminate. Coding a 1 causes the MSS operation to be repeated according to the repeat count.

- Repeat Count

  This field holds a positive binary number that is decremented by 1 after each repetition of the MSS operation that does not satisfy the operation condition. If this quantity reaches zero, the MSS instruction terminates and thereby passes control to the next sequential instruction. A repeat count of 0 causes a single execution of the MSS operation. A repeat count of n causes n+1 executions as long as no execution satisfies the operation condition. You can specify the count one of two ways: either as an 8-bit field within the operand 2 area;



  or, as a 32-bit register specified by a 4-bit field within the operand 2 area:



  If the repeat count is the 8-bit quantity contained in word 1 (bits 24—31 of operand 2) that value will not change during MSS execution. If, however, the repeat count is contained in a general register, it will be decremented by 1 for each repetition as previously described.

- Repeat Count Register Indicator (RR)

  You use this 1-bit field at word 1, bit 19 to indicate whether the repeat count is contained in the operand 2 area at word 1, bits 24—31 (RR=0) or in the general register specified in word 1, bits 28—31 (RR=1).

9.8.3.5. Format Summary

To help you select the one you want to use, all operand 2 formats and their fields are summarized in the following three illustrations. Figures 9—6 and 9—7 show the formats for the destination operand and repeat features, respectively; these can be used in all other formats. Figure 9—8 shows the 12 format codes and summarizes the fields unique to each:

- Primary Source Data (PSD)
- Secondary Source Data (SSD)
- Modification Fields (MODIFY)

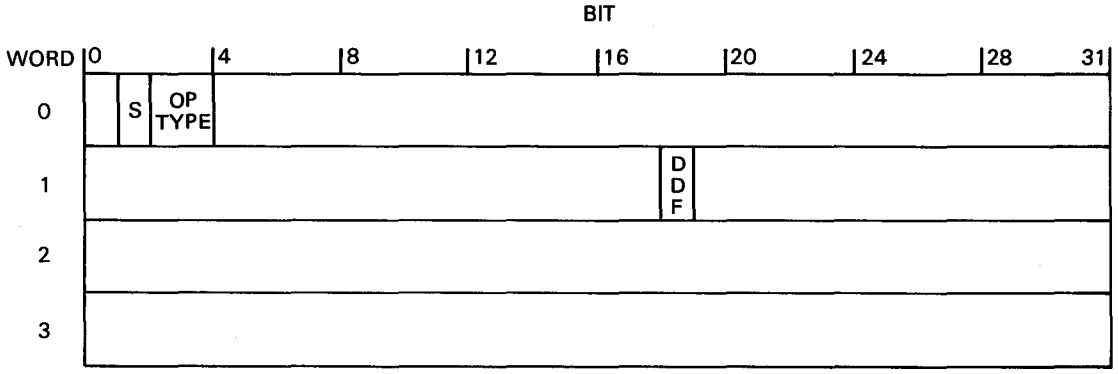


Figure 9—6. Destination Operand Fields

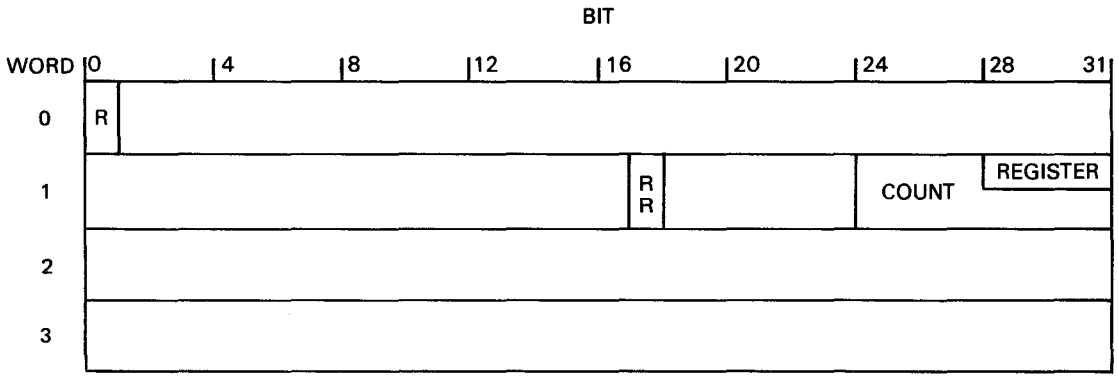


Figure 9—7. Repeat Fields

FORMAT CODE 0

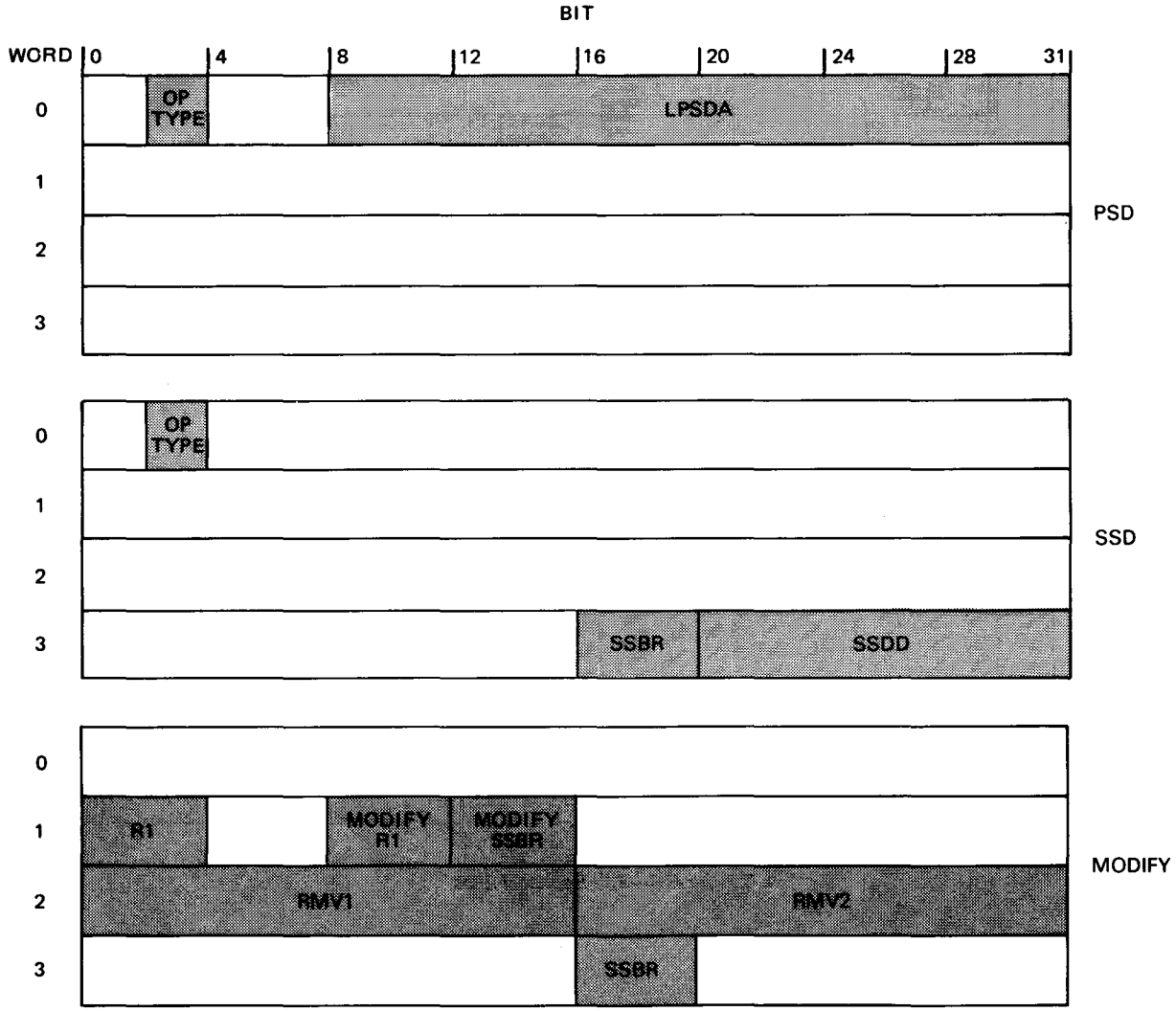


Figure 9—8. Format Fields (Part 1 of 12)

FORMAT CODE 1

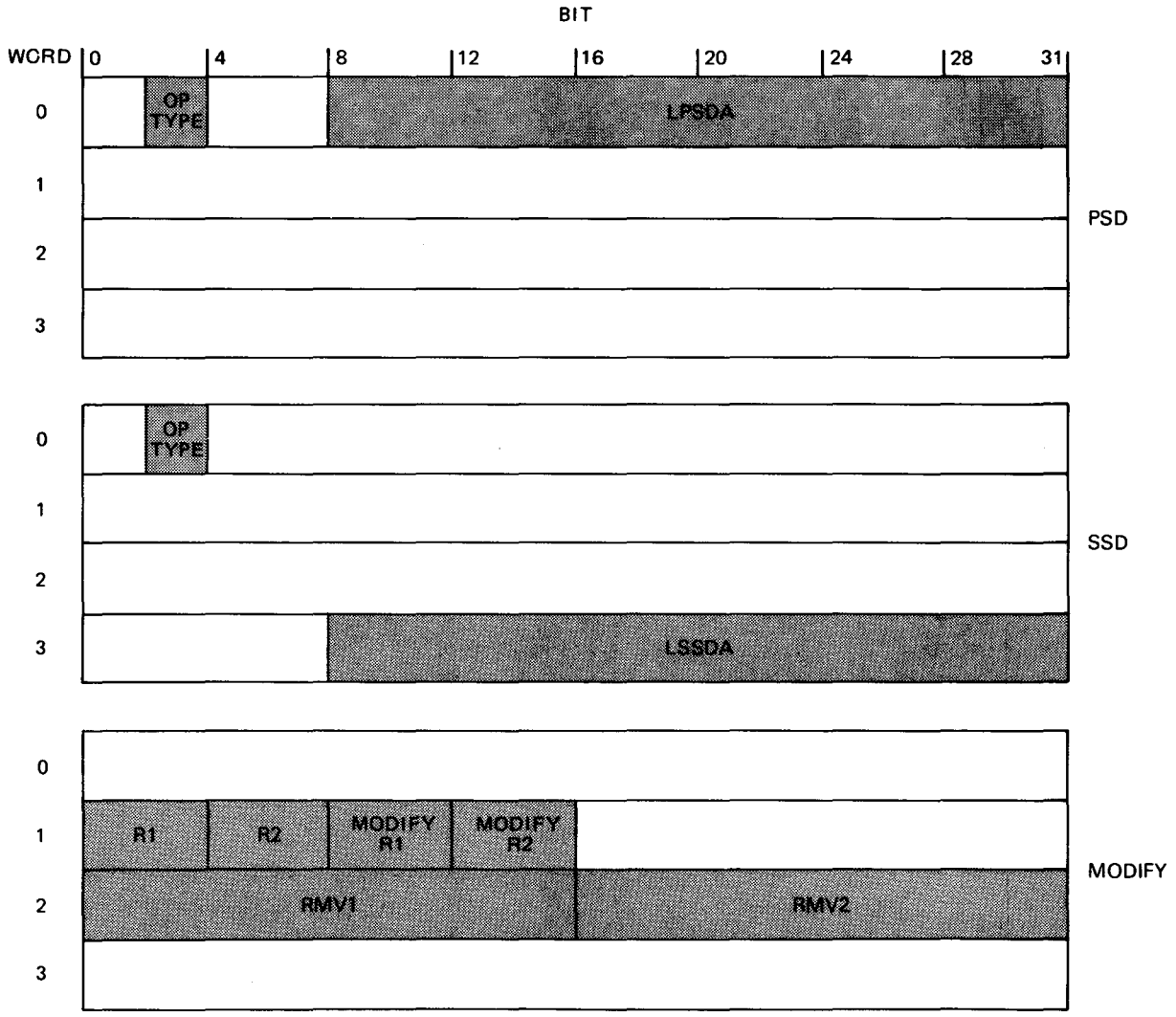


Figure 9—8. Format Fields (Part 2 of 12)

FORMAT CODE 2

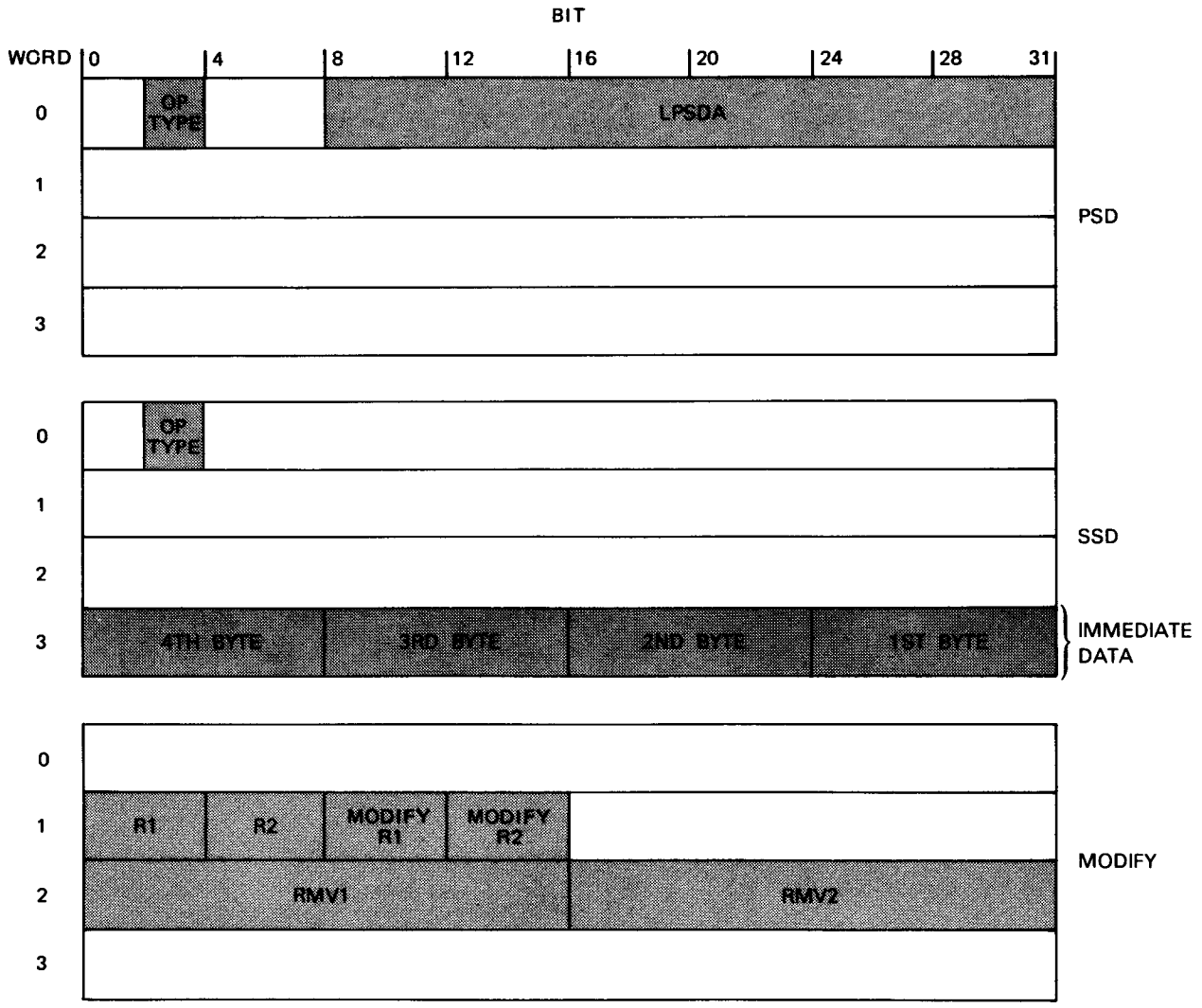


Figure 9—8. Format Fields (Part 3 of 12)

FORMAT CODE 3

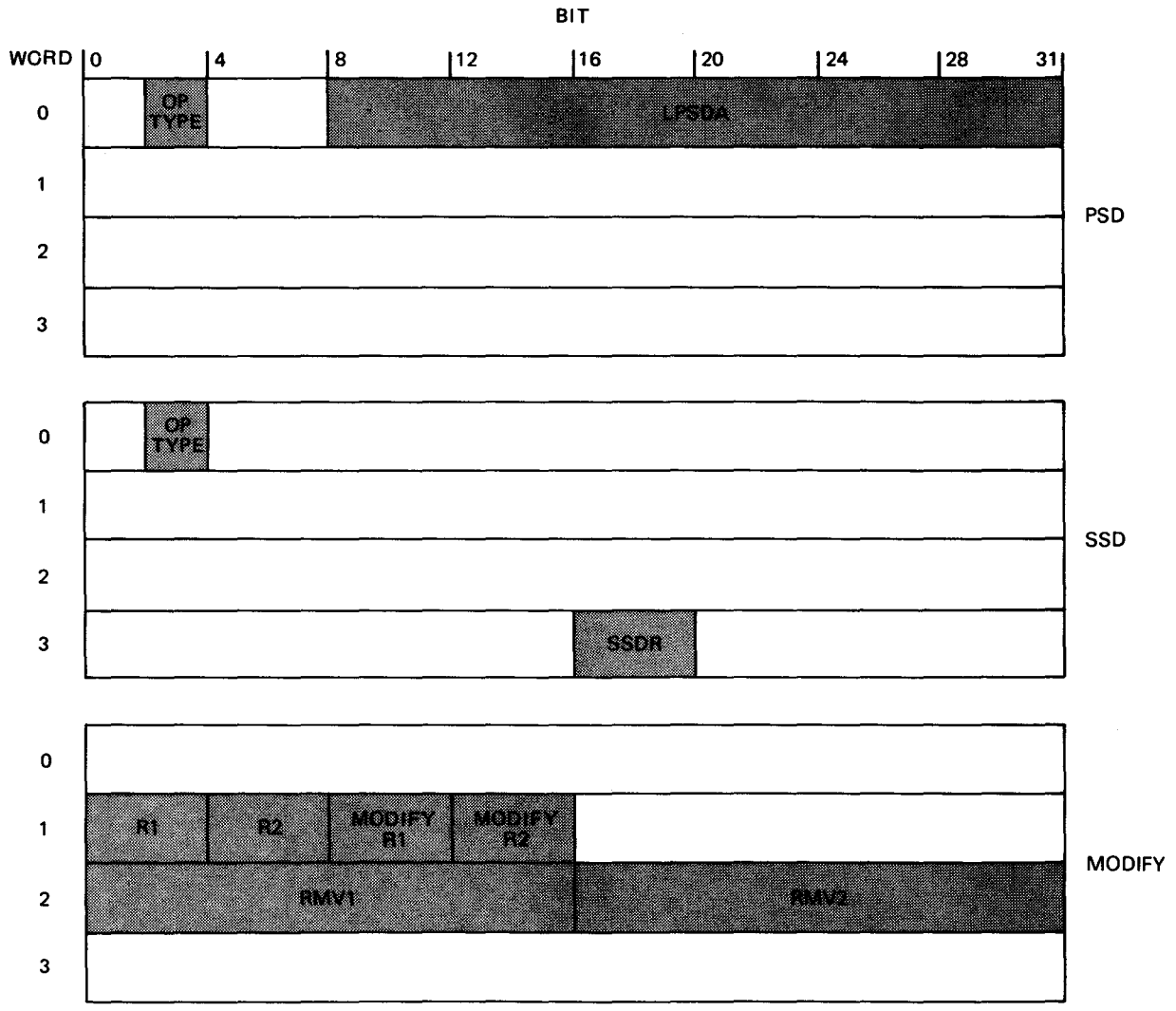


Figure 9—8. Format Fields (Part 4 of 12)

FORMAT CODE 4

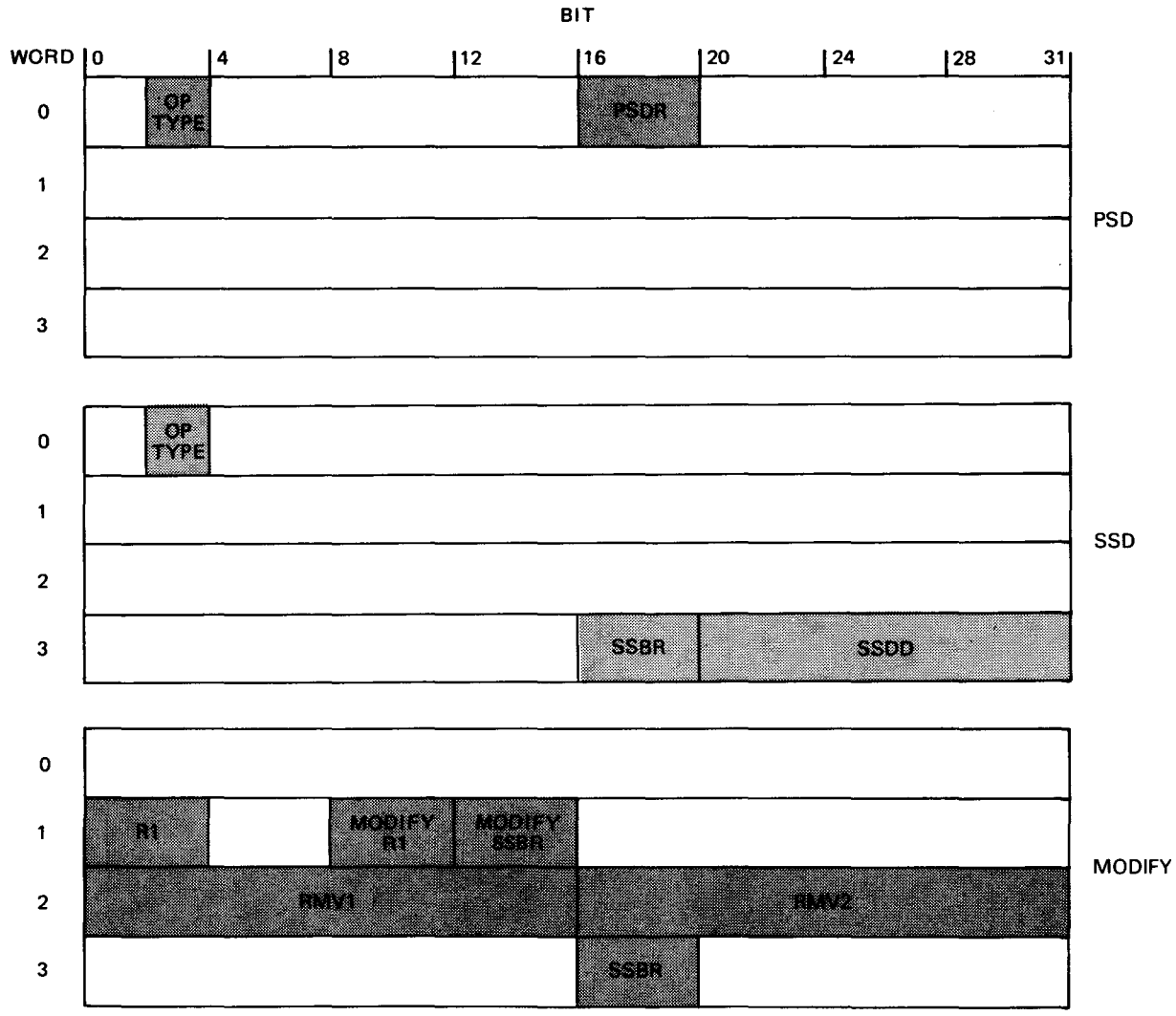


Figure 9—8. Format Fields (Part 5 of 12)

FORMAT CODE 5

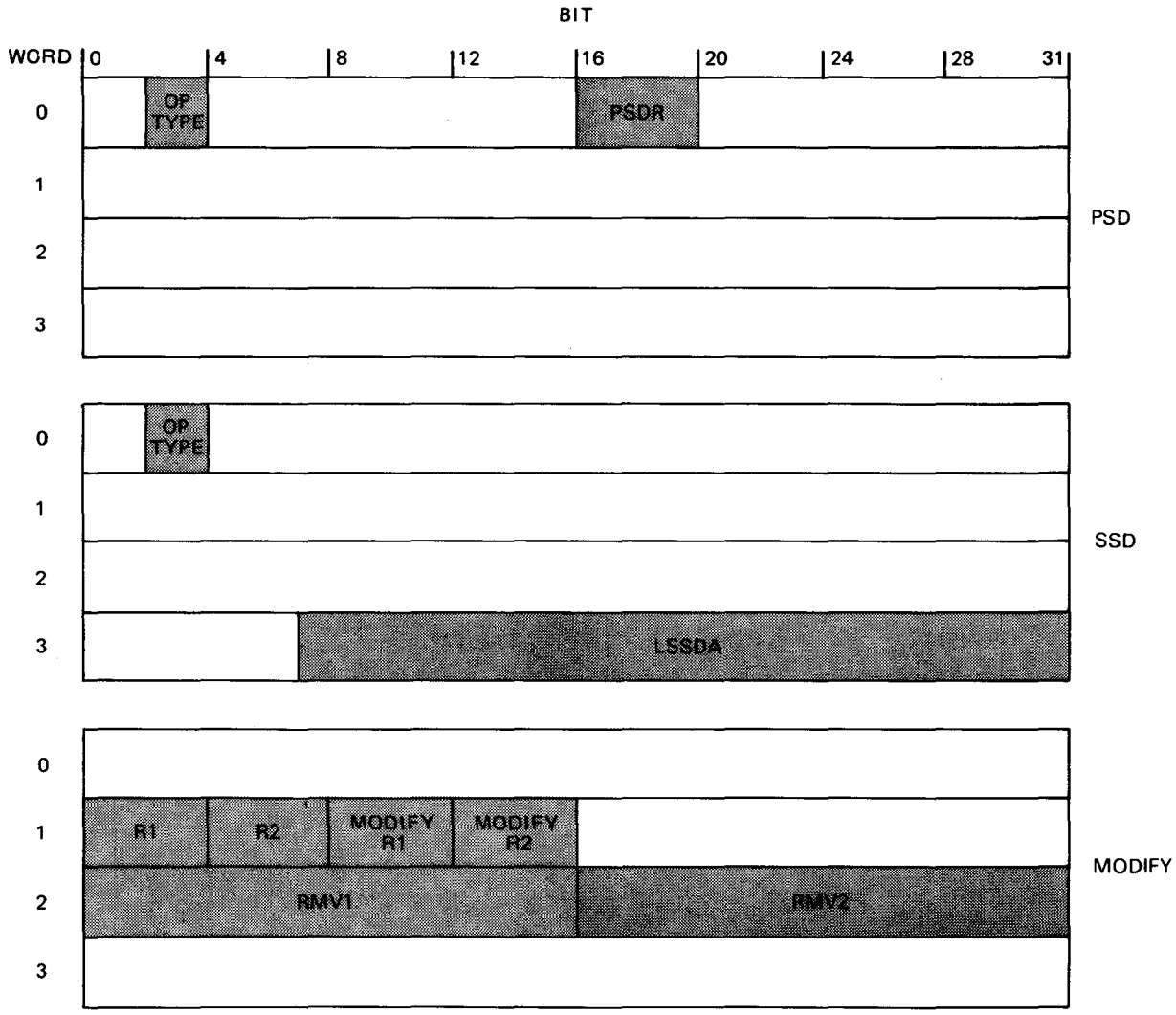


Figure 9—8. Format Fields (Part 6 of 12)

FORMAT CODE 6

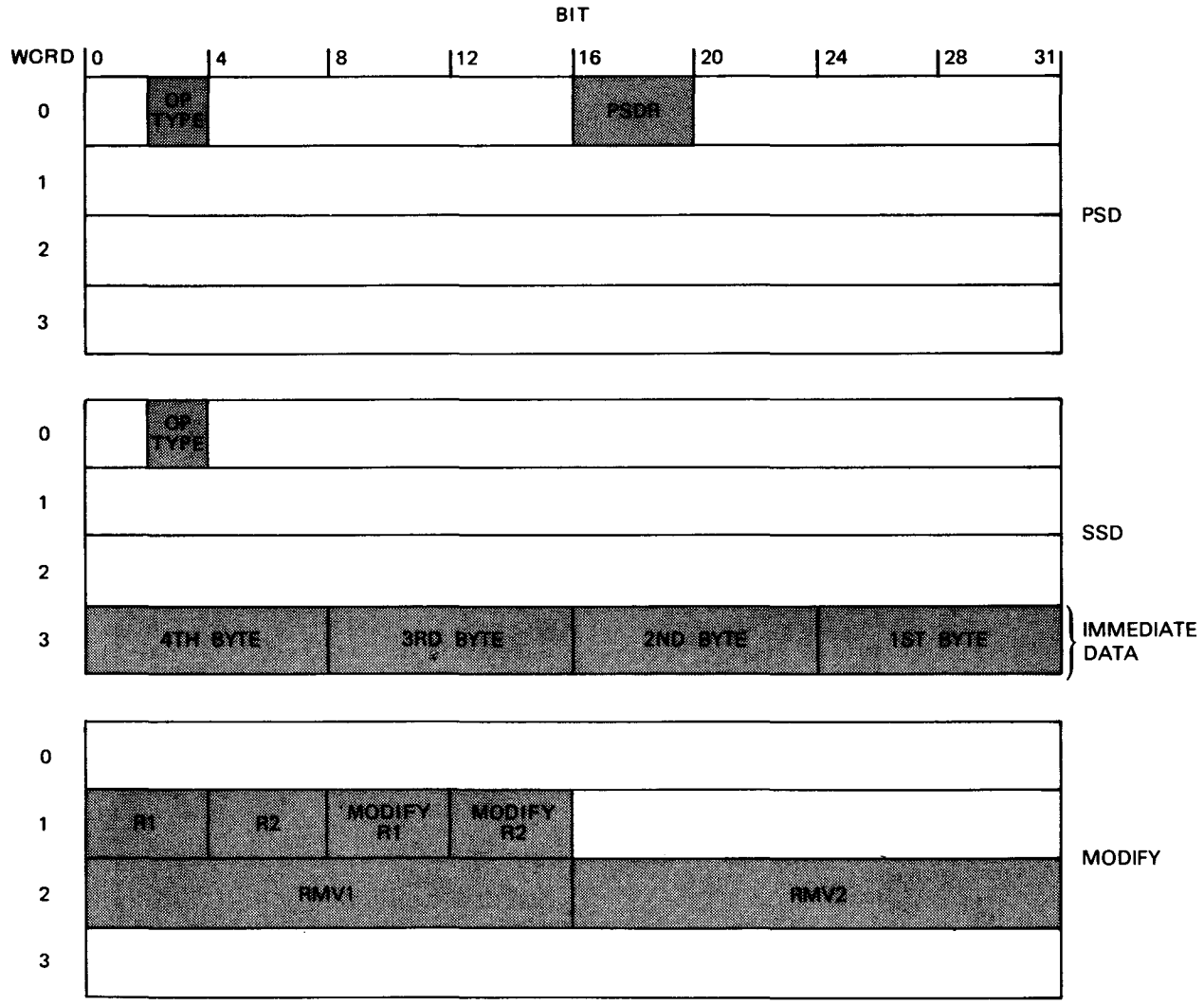


Figure 9—8. Format Fields (Part 7 of 12)

FORMAT CODE 7

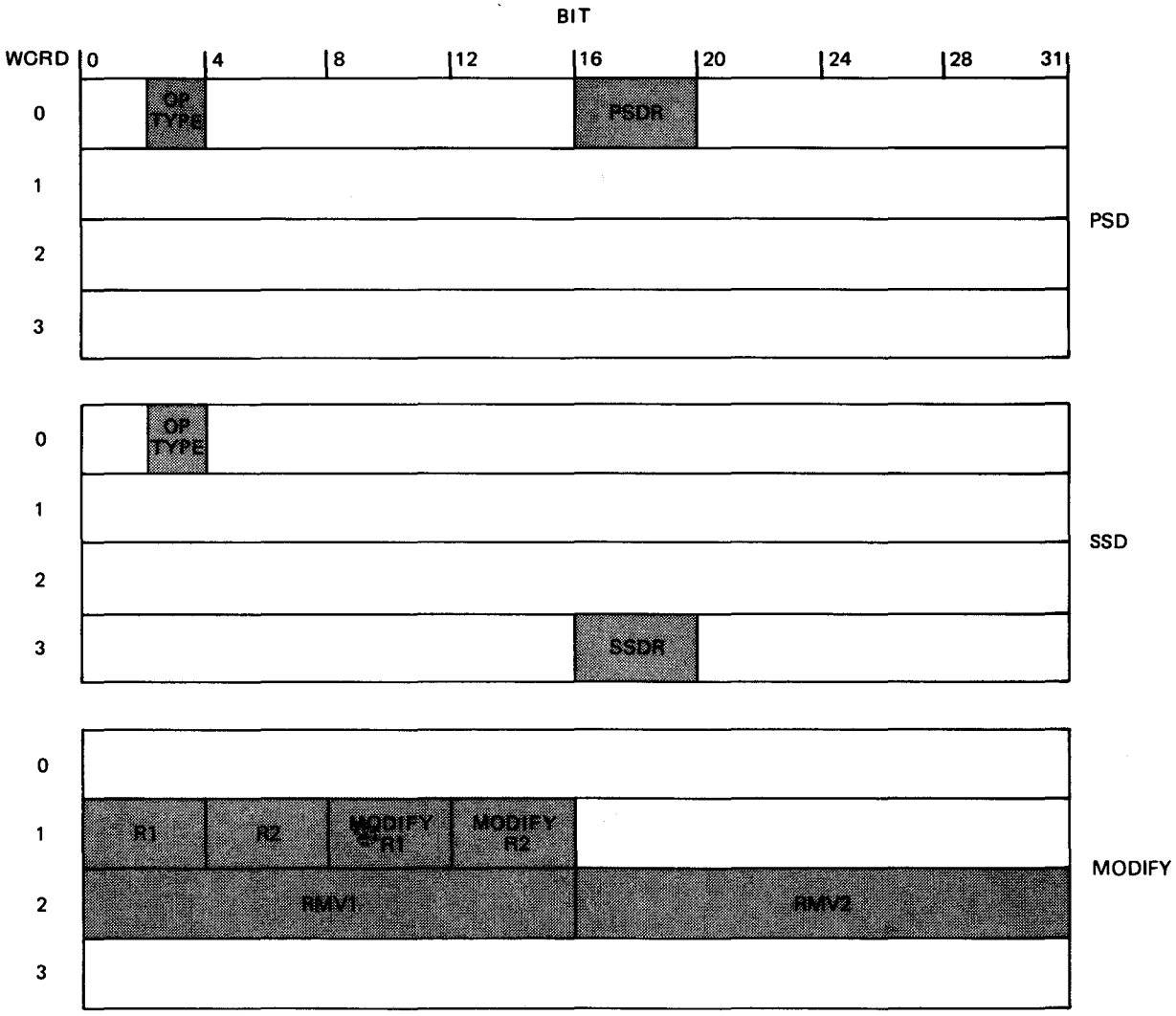


Figure 9—8. Format Fields (Part 8 of 12)

FORMAT CODE 8

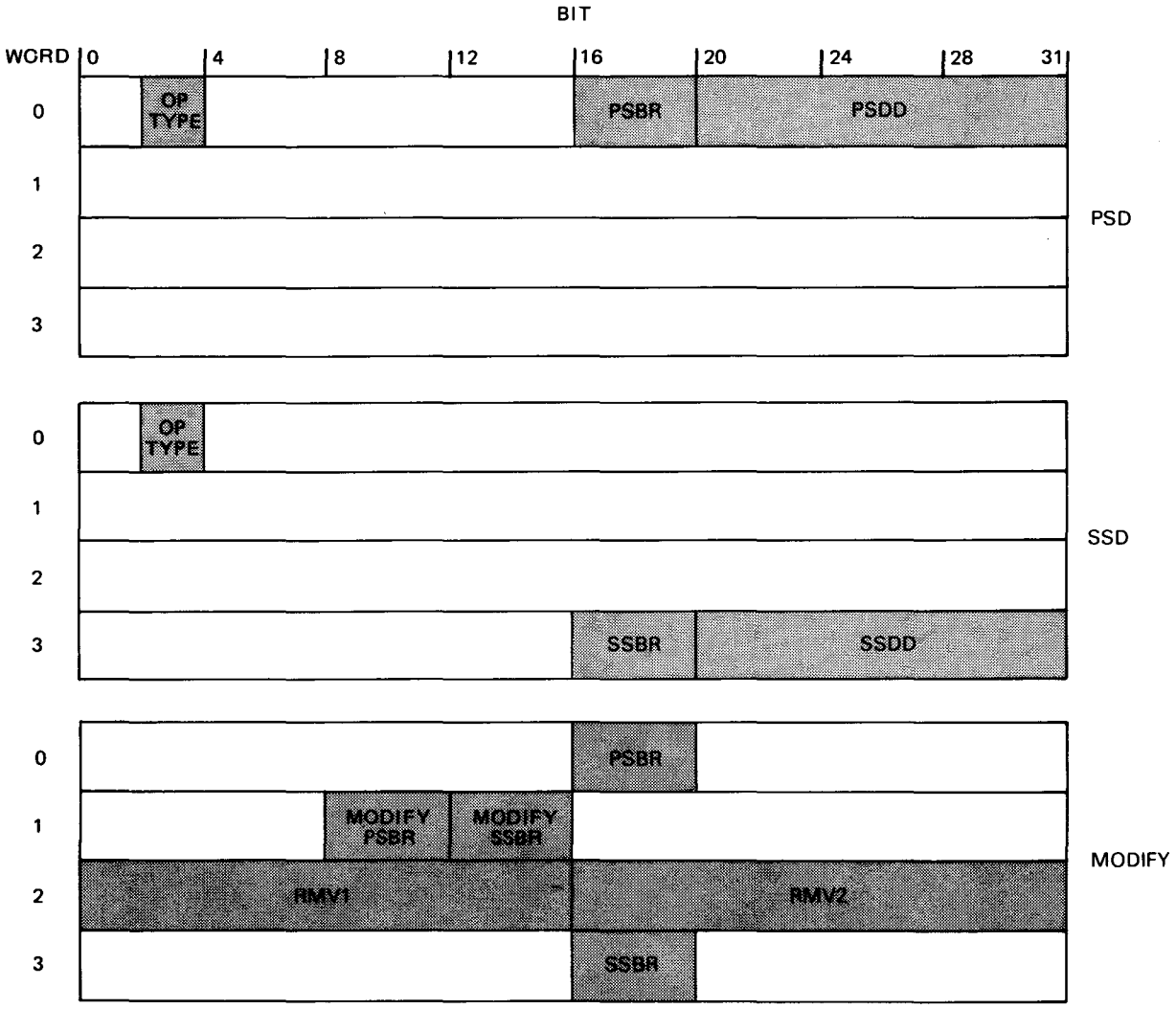


Figure 9—8. Format Fields (Part 9 of 12)

FORMAT CODE 9

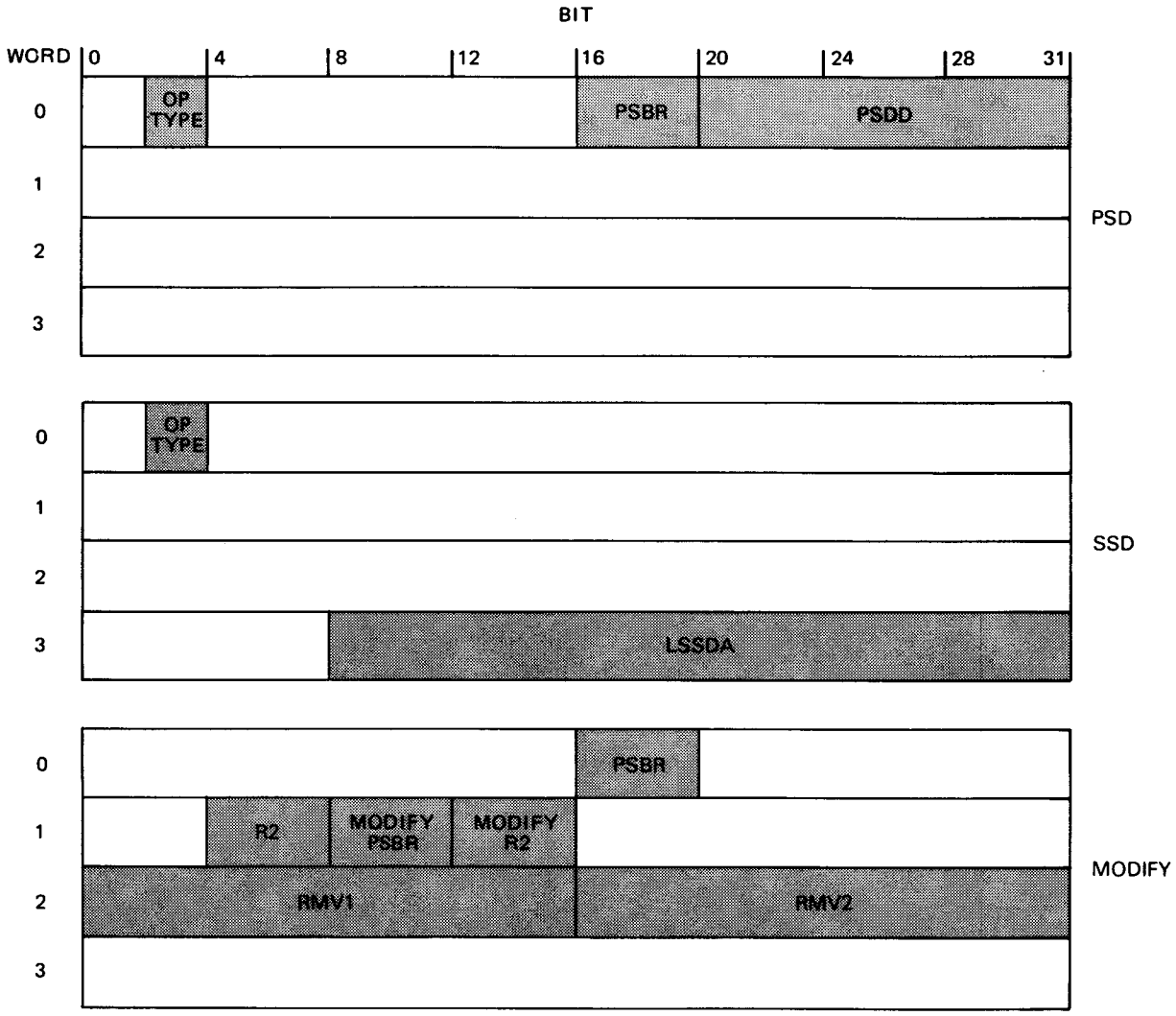


Figure 9—8. Format Fields (Part 10 of 12)

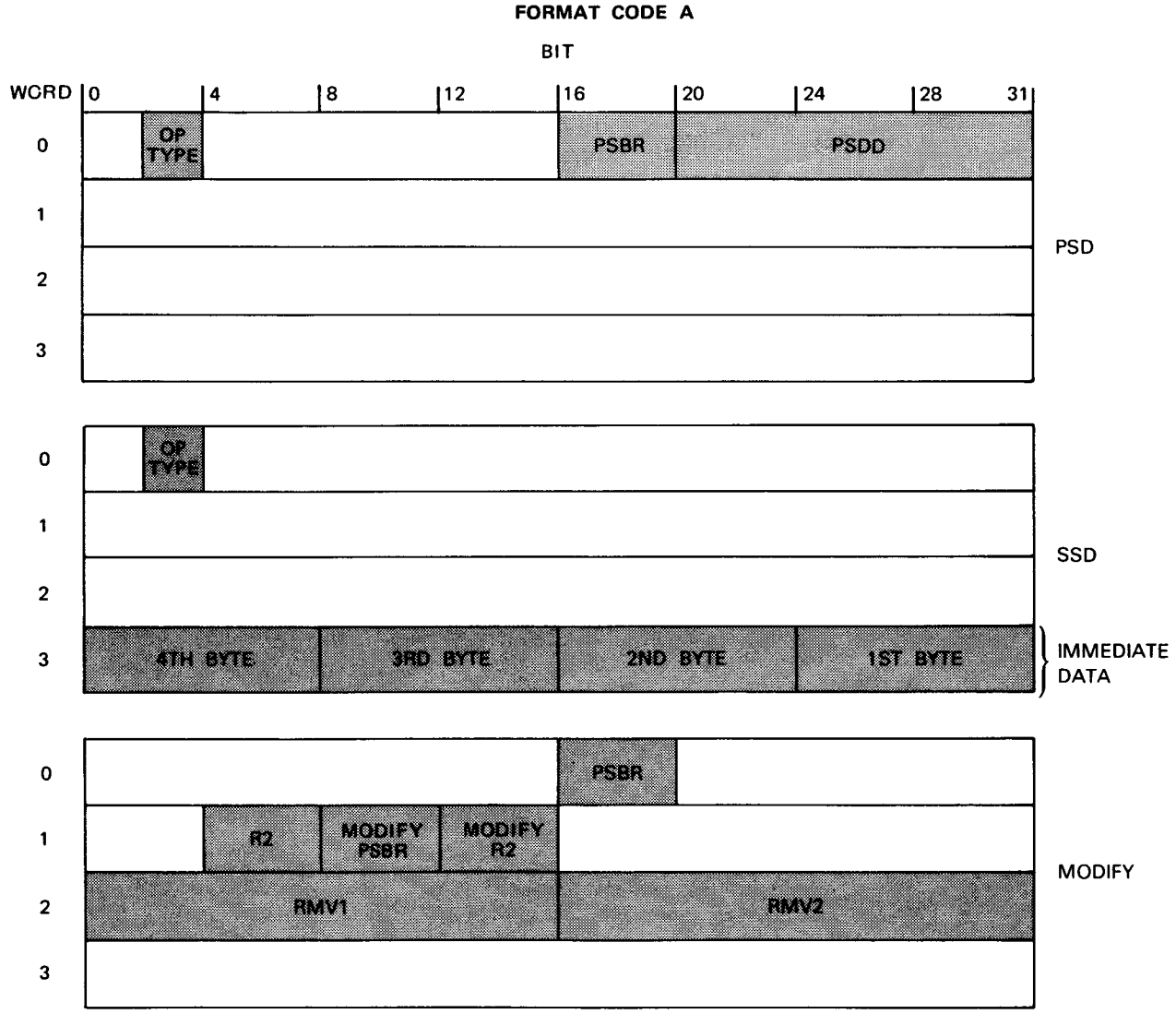


Figure 9—8. Format Fields (Part 11 of 12)

FORMAT CODE B

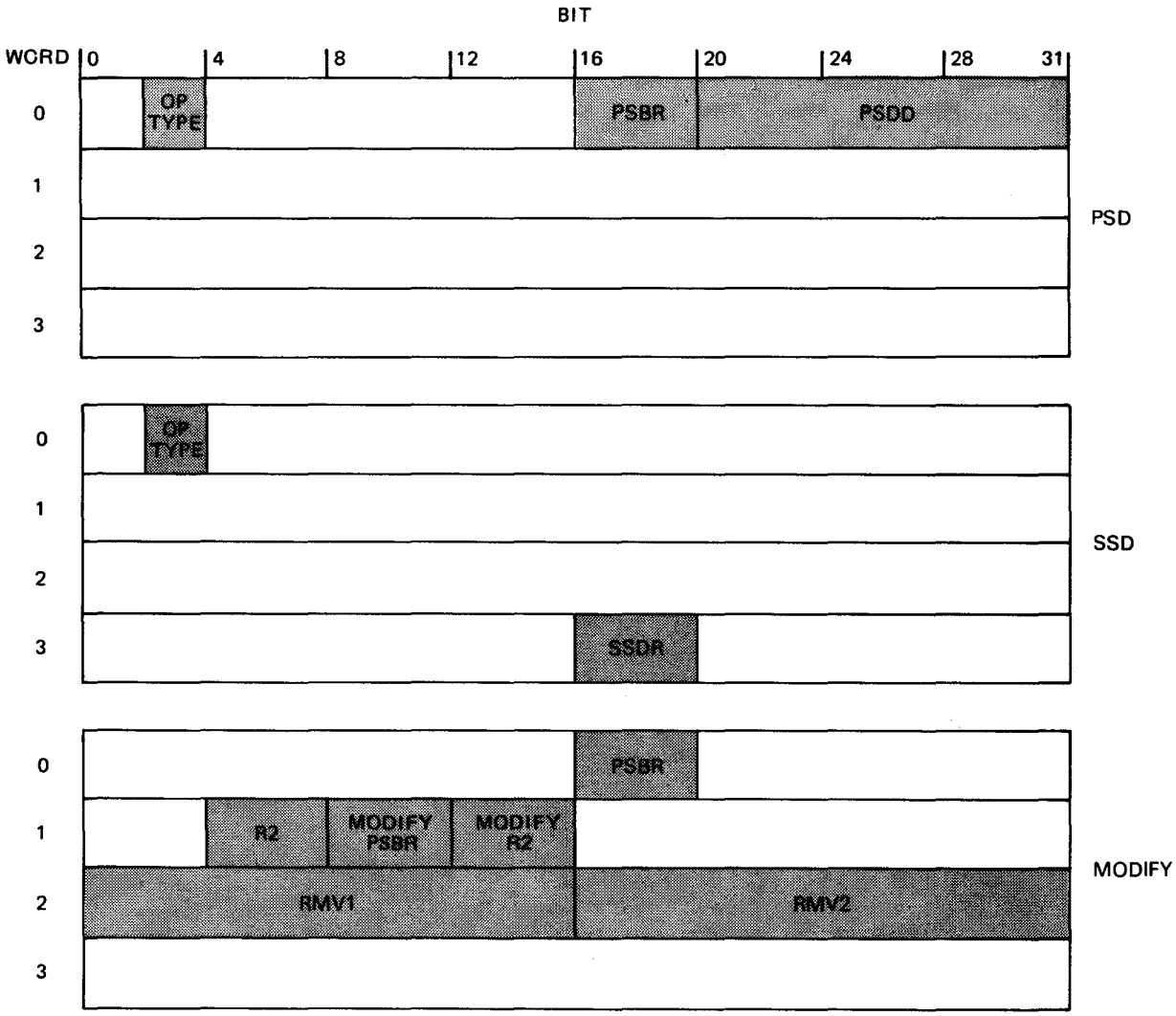


Figure 9—8. Format Fields (Part 12 of 12)

9.8.4. MSS Operation Conditions

The result of each MSS operation is tested to determine if it satisfies a condition implicit in the operation itself. The test uses the condition code, which each operation always sets according to the result. MSS operations are divided into three classes: arithmetic, move, and logical operations. The result of an arithmetic instruction is tested for a zero or nonzero result and for a carry out of its high order digit. The operands of a move operation are compared, and the PSD operand is moved to the DD operand if that option is enabled. The result of a logical operation is compared for zero or nonzero and, if enabled, is moved to the DD operand. Table 9—7 lists all MSS operations together with their possible results, condition code settings, and next instruction locations.

Table 9—7. MSS Operations and Conditions (Part 1 of 2)

ARITHMETIC OPERATIONS				
This operation performs . . .	**. . . this function.**	**If you get this result . . .**	**. . . this condition code is set . . .**	**. . . and the next instruction location is:**
ADDZ	Adds PSD and SSD, then tests for zero result.	VALUE / CARRY		
		zero / no	0	Skip location
		zero / yes	2	Skip location
		not zero / no	1	Next sequential location
		not zero / yes	3	Skip location
ADDNZ	Adds PSD and SSD, then tests for nonzero result.	zero / no	0	Next sequential location
		zero / yes	2	Next sequential location
		not zero / no	1	Skip location
		not zero / yes	3	Next sequential location
SUBZ	Subtracts SSD from PSD, then tests for zero result.	zero / yes	2	Skip location
		not zero / no	1	Skip location
		not zero / yes	3	Next sequential location
SUBNZ	Subtracts SSD from PSD, then tests for nonzero result.	zero / yes	2	Next sequential location
		not zero / no	1	Next sequential location
		not zero / yes	3	Skip location

Table 9—7. MSS Operations and Conditions (Part 2 of 2)

MOVE OPERATIONS				
Operation	**Function**	**Result**	**Condition Code**	**Instruction Location**
MCE	Compares PSD and SSD for equality, then optionally moves PSD to DD.	PSD=SSD PSD<SSD PSD>SSD	0 1 2	Skip location Next sequential location Next sequential location
MCNE	Compares PSD to SSD for inequality, then optionally moves PSD to DD.	PSD=SSD PSD<SSD PSD>SSD	0 1 2	Next sequential location Skip location Skip location
MCLE	Compares PSD to SSD for PSD≤SSD, then optionally moves PSD to DD.	PSD=SSD PSD<SSD PSD>SSD	0 1 2	Skip location Skip location Next sequential location
MCH	Compares PSD to SSD for PSD>SSD, then optionally moves PSD to DD.	PSD=SSD PSD<SSD PSD>SSD	0 1 2	Next sequential location Next sequential location Skip location
LOGICAL OPERATIONS				
ANDZ	Performs logical AND for PSD and SSD, then tests for zero result.	zero not zero	0 1	Skip location Next sequential location
ANDNZ	Performs logical AND for PSD and SSD, then tests for nonzero result.	zero not zero	0 1	Next sequential location Skip location
ORZ	Performs logical OR for PSD and SSD, then tests for zero result.	zero not zero	0 1	Skip location Next sequential location
ORNZ	Performs logical OR for PSD and SSD, then tests for nonzero result.	zero not zero	0 1	Next sequential location Skip location
XORZ	Performs logical XOR for PSD and SSD, then tests for zero result.	zero not zero	0 1	Skip location Next sequential location
XORNZ	Performs logical XOR for PSD and SSD, then tests for nonzero result.	zero not zero	0 1	Next sequential location Skip location

Note that when the repeat option is enabled, program control may not immediately pass to the next sequential instruction even if the operation condition is not satisfied. If the repeat count is greater than zero, program control passes right back to the MSS operation as in Figure 9—4.

9.8.5. Operational Considerations

- The operand 2 area must lie on a double-word boundary or else a specification exception will result.
- i_1 must be specified as a self-defining term within the range 0—D_{16}, or else an operation exception will result.
- The format code (word 0, bits 4—7 of the operand 2 area) must be within the range 0—B_{16} or else a specification exception will result.
- The MODIFY RX and MODIFY RY fields (word 1, bits 8—15) must both be within the range 0—6 or else a specification exception will result.
- Due to all these possible exceptions, you should put binary 0's in all operand 2 fields that are not used.

9.8.6. Example

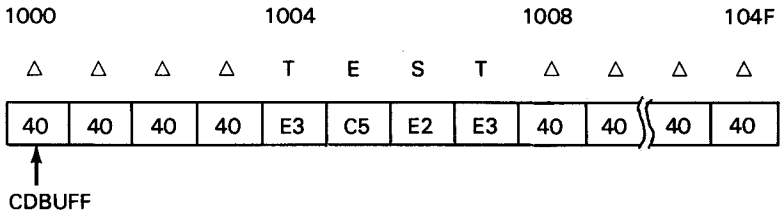
```
   LABEL    ΔOPERATIONΔ                          OPERAND
   1        10       16

 1           LA       8,CDBUFF                   CDBUFF: 8Ø-BYTE BUFFER.
 2           MVI      MSSCNTRL+7,79              SET REPEAT COUNT TO 79 (80-1).
 3           MSS      0(5),MSSCNTRL(2)
 4 * MSS OP  1: FUNCTION: MCNE -- WILL REPEAT UNTIL NONBLANK IS FOUND.
 5 *     OP  2: MSSCNTRL CONTROLS MSS EXECUTION.
 6 *     OP  3: SKIP LOCATION: 2 HALF-WORDS PAST NEXT SEQ. INST.
 7 ALLBLANK  B        LOC1                       THIS BRANCH TAKEN IF ENTIRE
 8 *                                              BUFFER IS BLANK.
 9 NOTBLANK  B        LOC2                       THIS BRANCH TAKEN IF A NONBLANK
10 *                                              CHARACTER IS FOUND IN BUFFER;
11 *                                              R8 CONTAINS ADDRESS OF FIRST
12 *                                              NONBLANK.
13             .
14             .
15           DS       ØD
16 MSSCNTRL  DC       B'1ØØØ1Ø1Ø'       WORD Ø:ENABLE REPEAT AND FMT. CODE A.
17           DC       X'ØØ'                     NOT USED.
18           DC       X'8ØØØ'                   PSD ADDRESS IN Ø(8) FORM.
19           DC       X'8Ø'              WORD 1:RX REGISTER IS R8.
20           DC       X'1Ø'                     MODIFY RX: INCREMENT R8 BY 1.
21           DC       X'ØØØØ'                   USE BITS 24-31 AS REPEAT COUNT.
22           DC       F'Ø'               WORD 2:NOT USED.
```

```
     LABEL    ΔOPERATIONΔ                    OPERAND
     1        10        16
 23          DC        X'ØØØØØØ4Ø'      WORD 3:IMMEDIATE BYTE (BLANK) IN
 24 *                                          LOW-ORDER BYTE; Ø'S IN
 25 *                                          REMAINING BYTES.
 26 CDBUFF   DS        CL8Ø
```

In this example, we use a MSS instruction to scan an 80-byte field in main storage starting at CDBUFF. If the scan finds a nonblank character, MSS execution terminates immediately, with register 8 containing the address of the nonblank character and program control passing to the branching instruction at line 9. If the 80-byte CDBUFF area contains all blanks, program control passes to the branch instruction at line 7.

Let's assume that CDBUFF is located at address 1000 and contains the following:



The four full words at MSSCNTRL control MSS execution. Looking at lines 16—23, you can see that we are enabling the repeat option using as the repeat count the data in word 1, bits 24—31. Other fields indicate that:

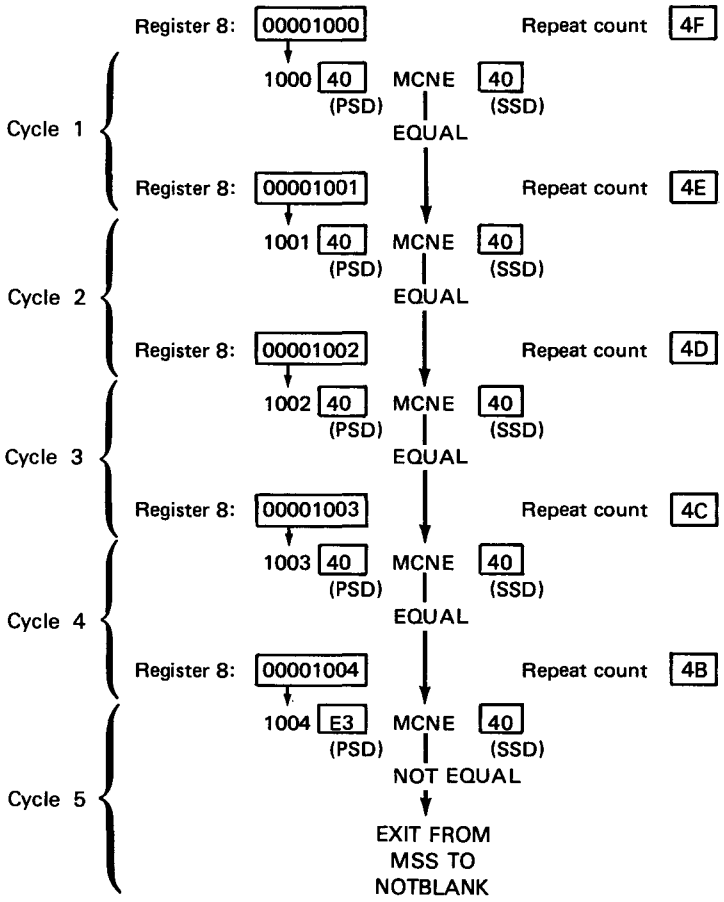
- we are using single bytes as operands;
- the PSD operand is in main storage, addressed using zero displacement and a base register of 8;
- the SSD operand is a byte of immediate data that has a value of X'40'; and
- each unsuccessful execution of the MSS operation causes register 8 to be incremented by 1.

In line 1, we load register 8 with the CDBUFF address and in line 2 set the repeat count to 79 (X'4F') by inserting that value in the MSSCNTRL repeat count field (during processing the repeat count field in MSSCNTRL will not change, but the repeat count itself will be decremented):

Register 8: 00001000 Repeat count: 4F

At line 3, we execute the MSS instruction, using the MCNE function to compare the PSD and SSD operands, and branching to skip location NOTBLANK if the two operands are not equal.

The following illustration shows the action of MSS:



For four cycles (1—4), the MSS instruction compares its immediate byte SSD with the PSD addressed by register 8, and finds them equal. Because this does not satisfy the condition imposed by MCNE, the instruction, at each cycle, increments the register 8 contents by 1 and reduces the repeat count, also by 1. This way, MSS addresses successive bytes CDBUFF, CDBUFF+1, CDBUFF+2, etc.

At cycle 5, however, the PSD byte is found unequal to the SSD byte, thus satisfying the MCNE condition. Both register 8 and the repeat count are left unchanged and program control passes to skip location NOTBLANK. At this time, register 8 points to CDBUFF+4, the first nonblank character MSS found.

Note that if all bytes were blank up to CDBUFF+79 (at address 104F), MSS would then reduce its repeat count to zero thus causing program control to pass to the next sequential instruction, ALLBLANK. This is the mechanism we use to limit the MSS scan to just the 80 bytes required.

14. List Processing

14.1. INTRODUCTION

Certain System 80 instructions allow you to create and manipulate linked lists. With these instructions you can add new members or *elements* to a list, remove existing elements, and even scan an entire list, performing logical or data movement operations as you go. You do not have to develop extensive software to handle lists because many list processing functions are now available to you in System 80 hardware. All you need to do is specify the parameters within which list processing is to take place, and the System 80 instructions do the rest. In this section we give a general overview of what types of lists you can use and the rules that govern their use. In 14.4 and 14.5 we explain in detail how to use the System 80 instructions that manipulate these lists.

All lists processing instructions deal with linked lists. These types of lists each consist of zero, one, or more than one element, which in turn contain data and one or two pointers. A pointer is a full word that contains the address of a logically adjacent element; we say that it points to that element. Each element points to at least one other element, and a separate structure called a list head points to either end of the list, thus defining and controlling the list. In the discussion that follows we use the terms *forward* and *backward* pointers to distinguish between the two pointers an element may have. For those lists having only one pointer it is considered a forward pointer.

14.1.1. LIFO List

One type of list is called a last-in-first-out (LIFO) list (Figure 14—1), sometimes called a stack.

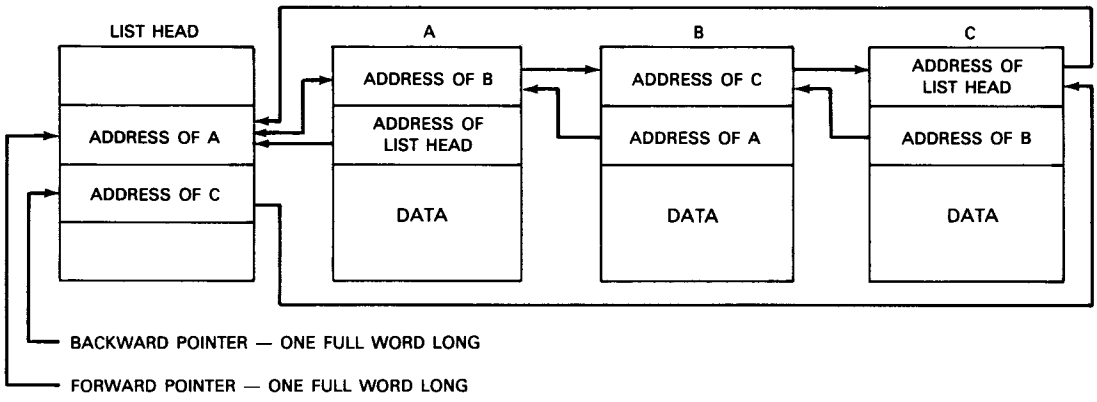
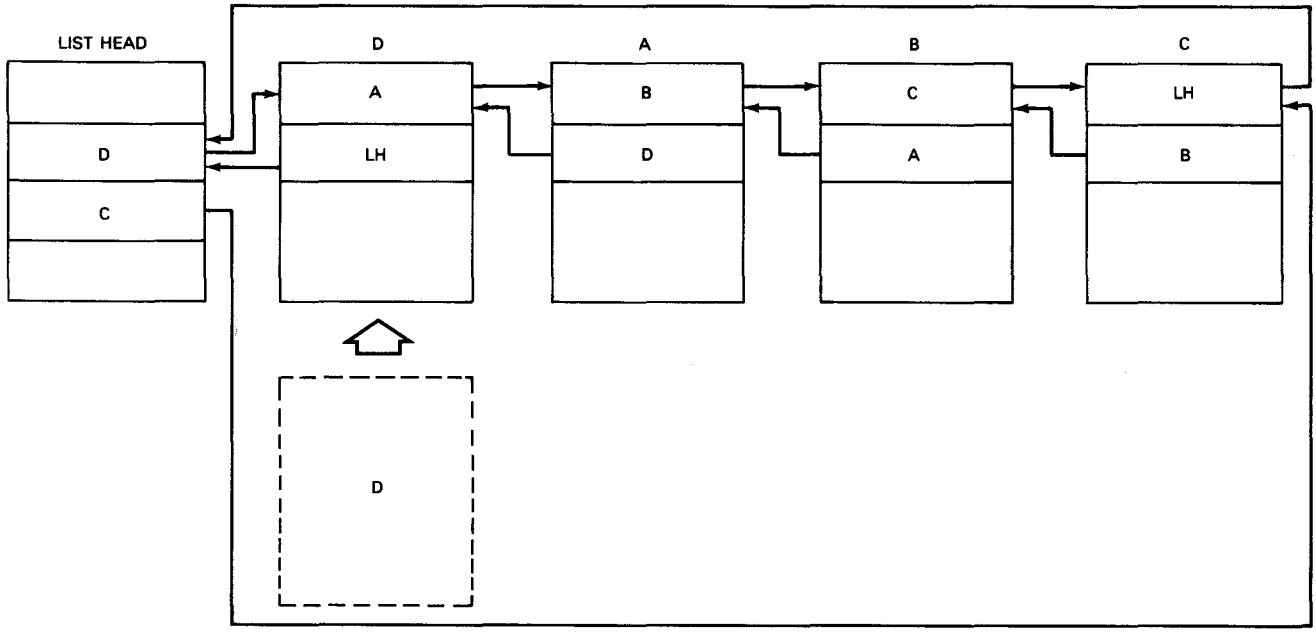


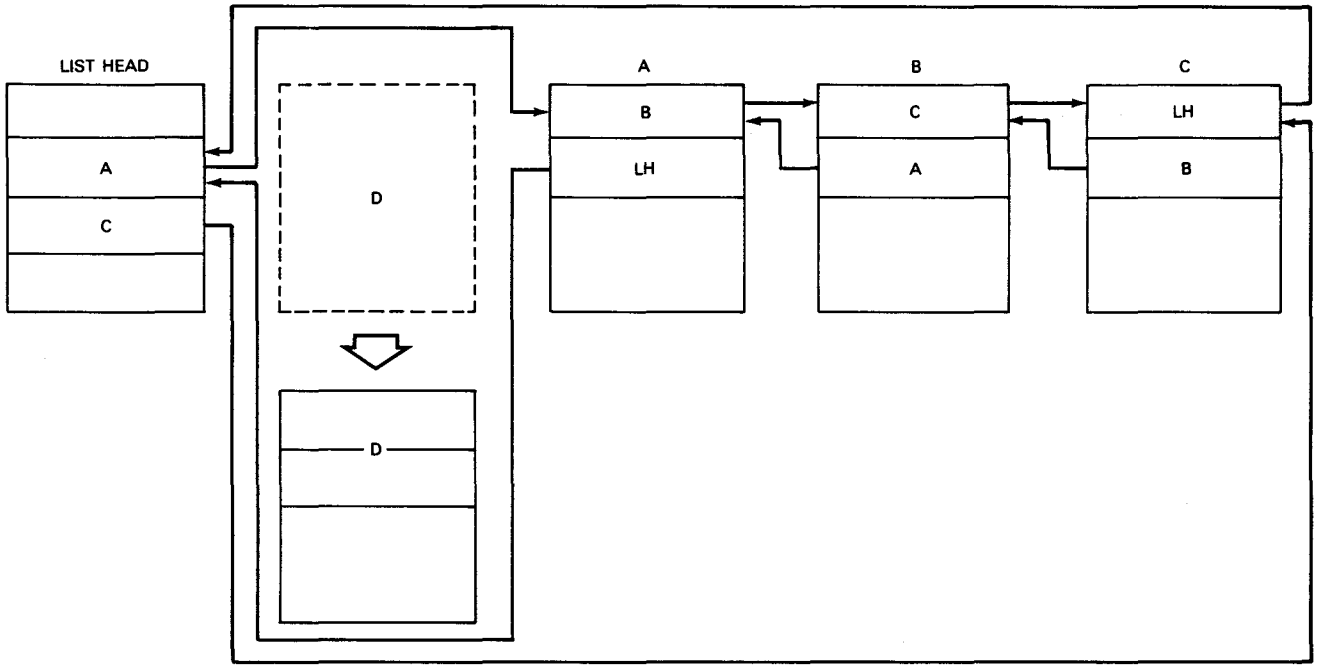
Figure 14—1. LIFO List

Each element has two pointers: the forward pointer which points to the next element in logical sequence, and the backward pointer which points to the previous element. All pointers to an element generally point to the forward pointer field of that element. Note that following the forward pointers from element to element takes us all the way around the list in one direction; following the backward pointers takes us around in the opposite direction. Note also that the DATA field in each element plays no role in organizing a list. We drop the DATA label from subsequent illustrations with the understanding that a data field of your choosing can be present in any element. We also, in subsequent illustrations, label pointers simply with the name of the element to which they point with the understanding that they actually contain that element's address.

We call the list in Figure 14—1 a LIFO list because of how we can add or remove elements. As Figure 14—2 shows, elements are added and removed from only one end of the LIFO list, the way you would put dinner plates on a stack or remove them from the stack. If the other three elements of the list had been removed after element D, it would be in this order: A B C. Note that element D is not physically inserted in, or removed from, the list. Rather, it gets new pointers to the list head and to element A. The list head and element A pointers in turn now point to the newly added element so that it occupies the first position on the list. Pointers thus make possible a wide variety of list processing operations that take little processing time because the elements themselves do not have to be shifted around in main storage; only their pointers are changed.



a. Adding an element

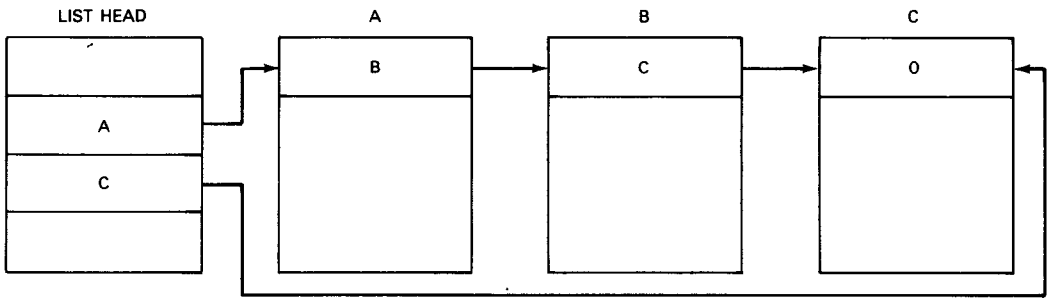


b. Removing an element

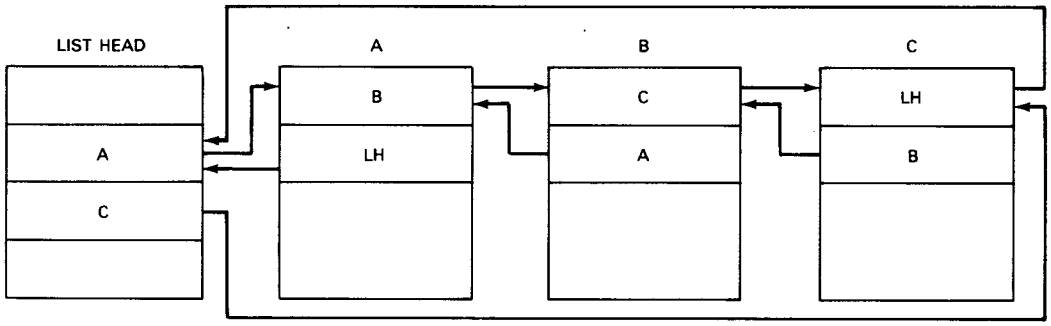
Figure 14—2. Adding and Removing Elements from a LIFO List

14.1.2. FIFO List

Other types of lists are available with System 80. One is the first-in-first-out (FIFO) list, sometimes called a queue. It comes in two forms as shown in Figure 14—3.



a. Forward-linked FIFO list

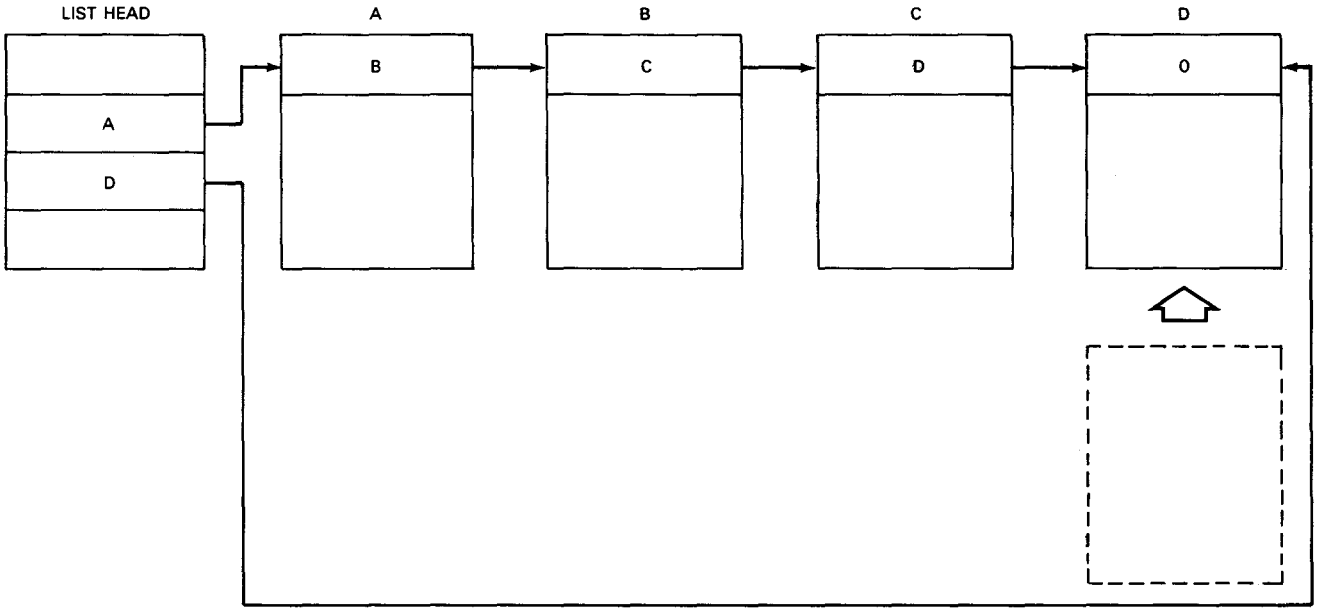


b. Double-linked FIFO list

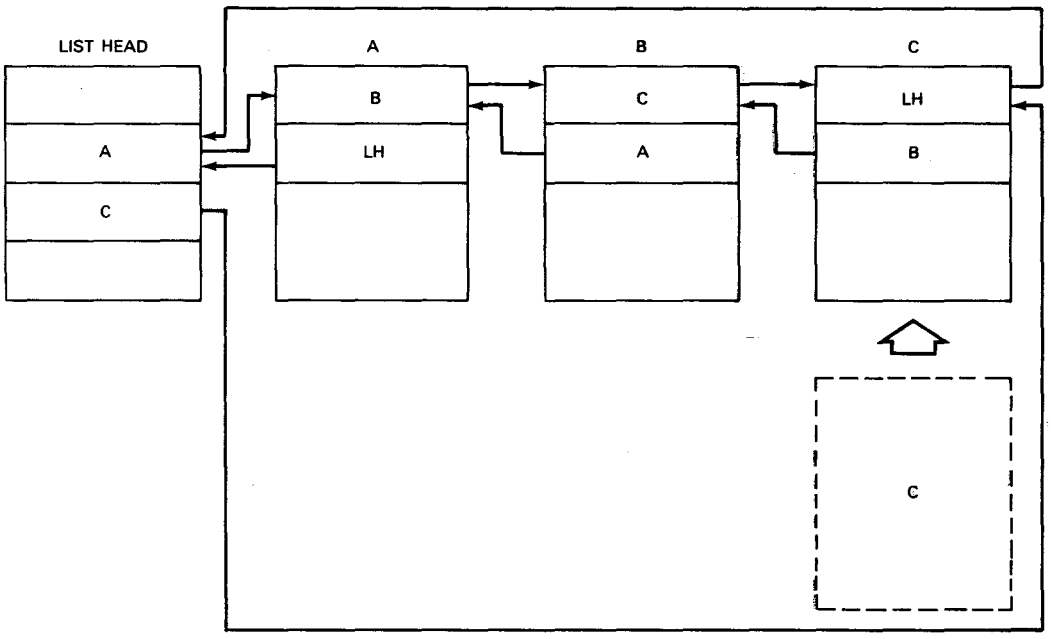
Figure 14—3. The Two Types of FIFO Lists

The forward-linked elements each point to the next element in the list except for element C; that element has a pointer of 0 which indicates that it is the last element in the list. List head pointers indicate the first and last elements. The double-linked FIFO list looks exactly like the LIFO list shown in Figure 14—1. FIFO lists, however, are handled differently.

As you can see from Figure 14—4, elements are added at one end of a FIFO list and removed from the other end, the one closest to the list head. Elements thus appear to move toward the list head much as customers move towards a teller's window in a bank.

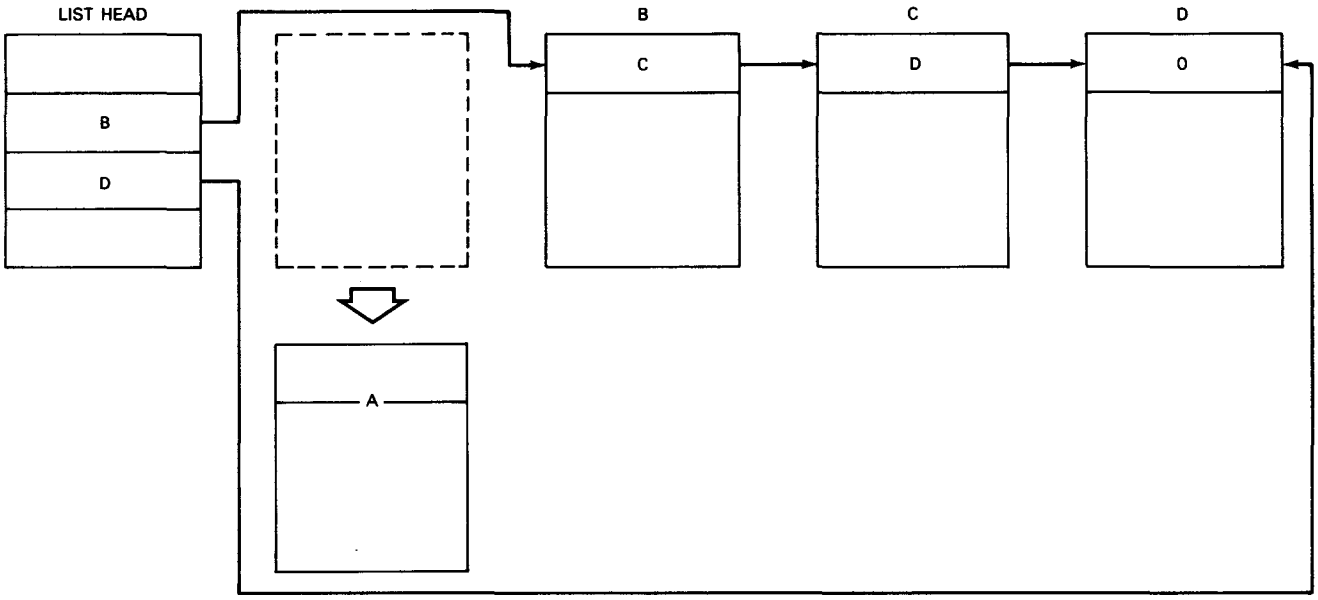


a. Adding to a forward-linked list

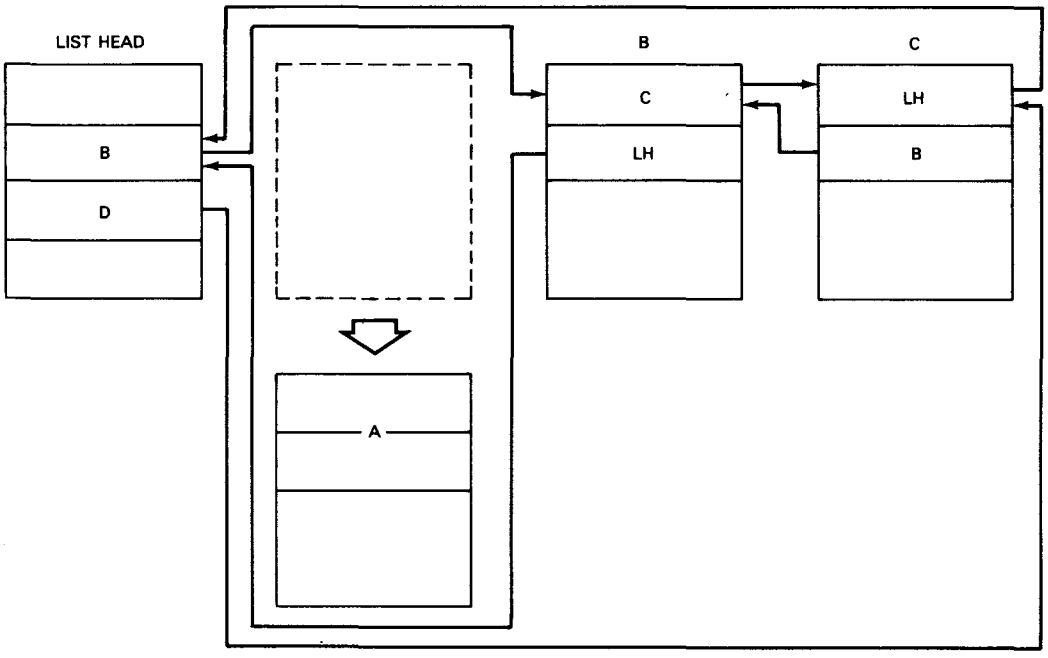


b. Adding to a double-linked list

Figure 14—4. Adding and Removing Elements in FIFO Lists (Part 1 of 2)



c. Removing from a forward-linked list



d. Removing from a double-linked list

Figure 14—4. Adding and Removing Elements in FIFO Lists (Part 2 of 2)

One variation on the FIFO list is the use of a special pointer called a station, located in the list head. In Figure 14—5 you can see that the station in the list head is pointing to the middle element of the list. While the other list head pointers must point to the end elements of a list, the station is under no such constraint. System 80 instructions can move the station back and forth, altering its value so that it points to different elements. Its only constraints are that it can move but one element position at a time and that it is limited to forward movement in a forward-linked list. Otherwise, however, you can move it from end to end in a list.

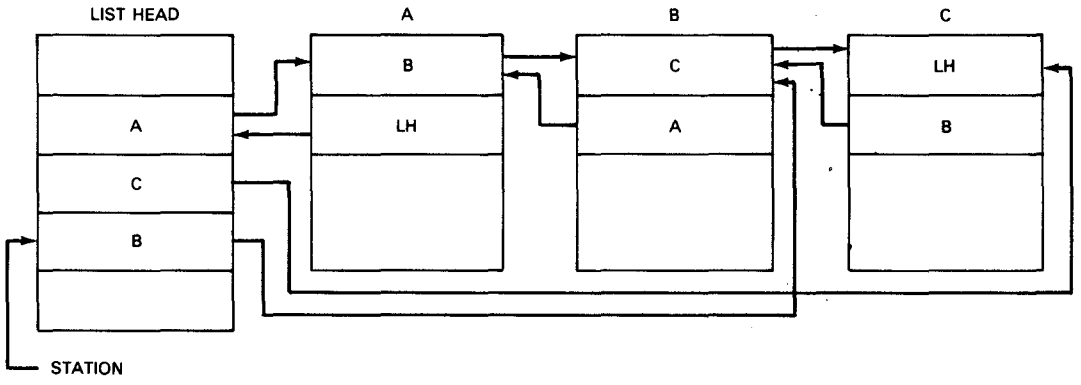


Figure 14—5. FIFO List with Station

14.1.3. Double-ended List

A double-ended list or queue is a double-linked list whose structure is similar to that of a FIFO list. Unlike a FIFO list, you can add or remove elements at either end of the list.

14.1.4. Ring with Station

Another type of list is the ring list with station as shown in Figure 14—6.

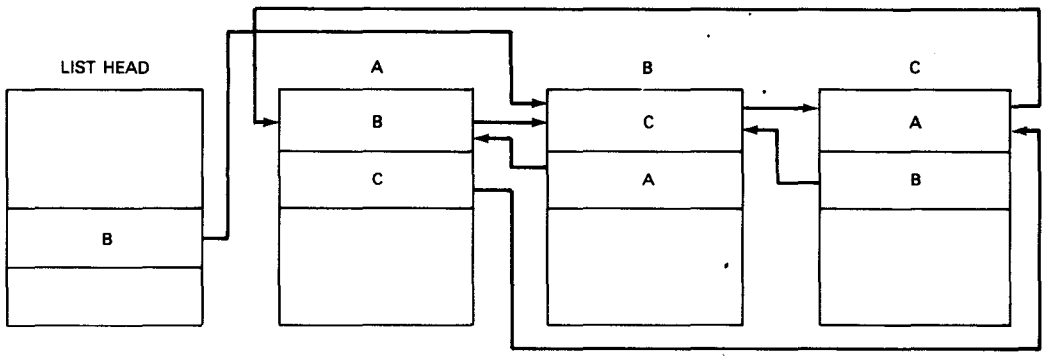
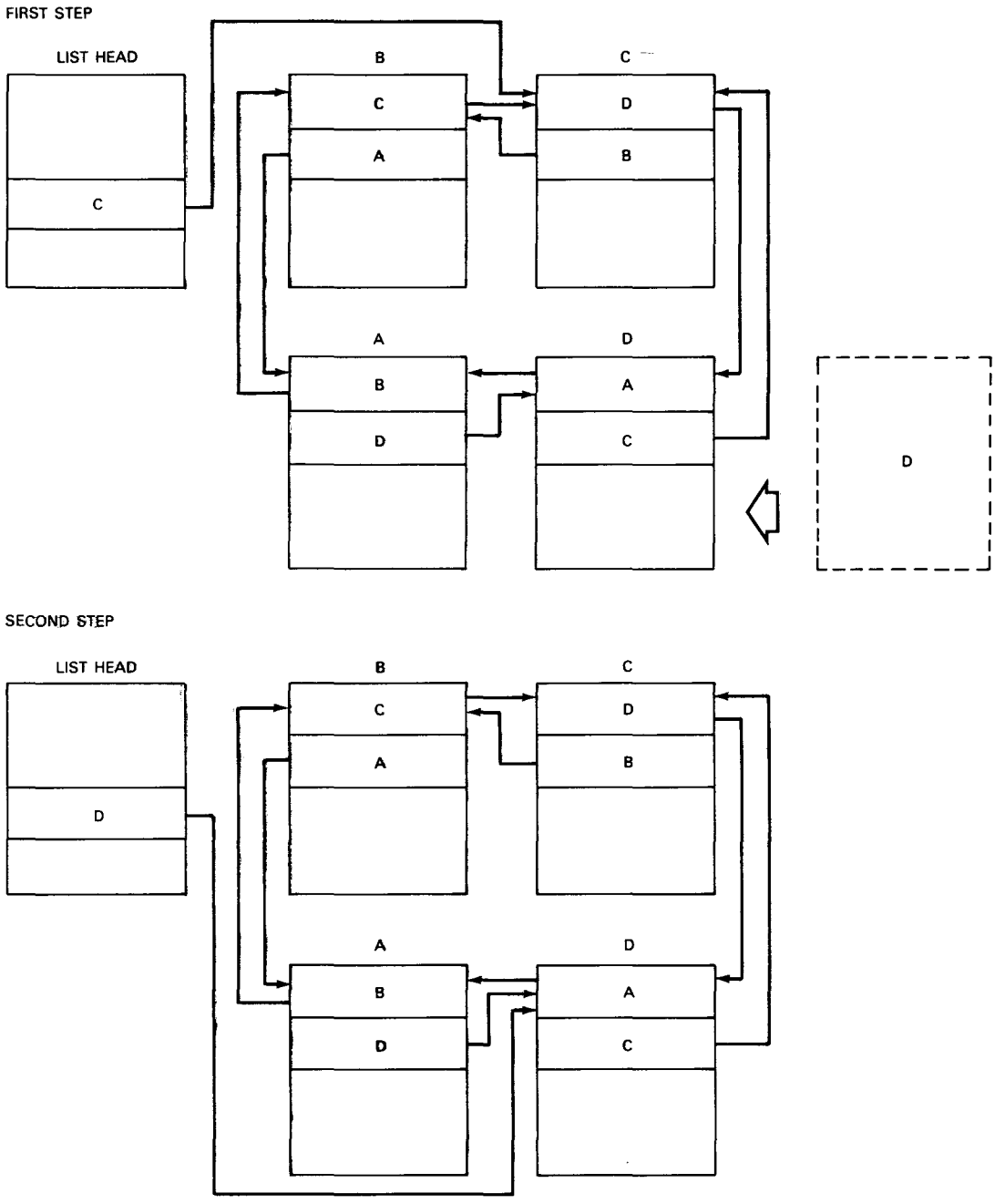


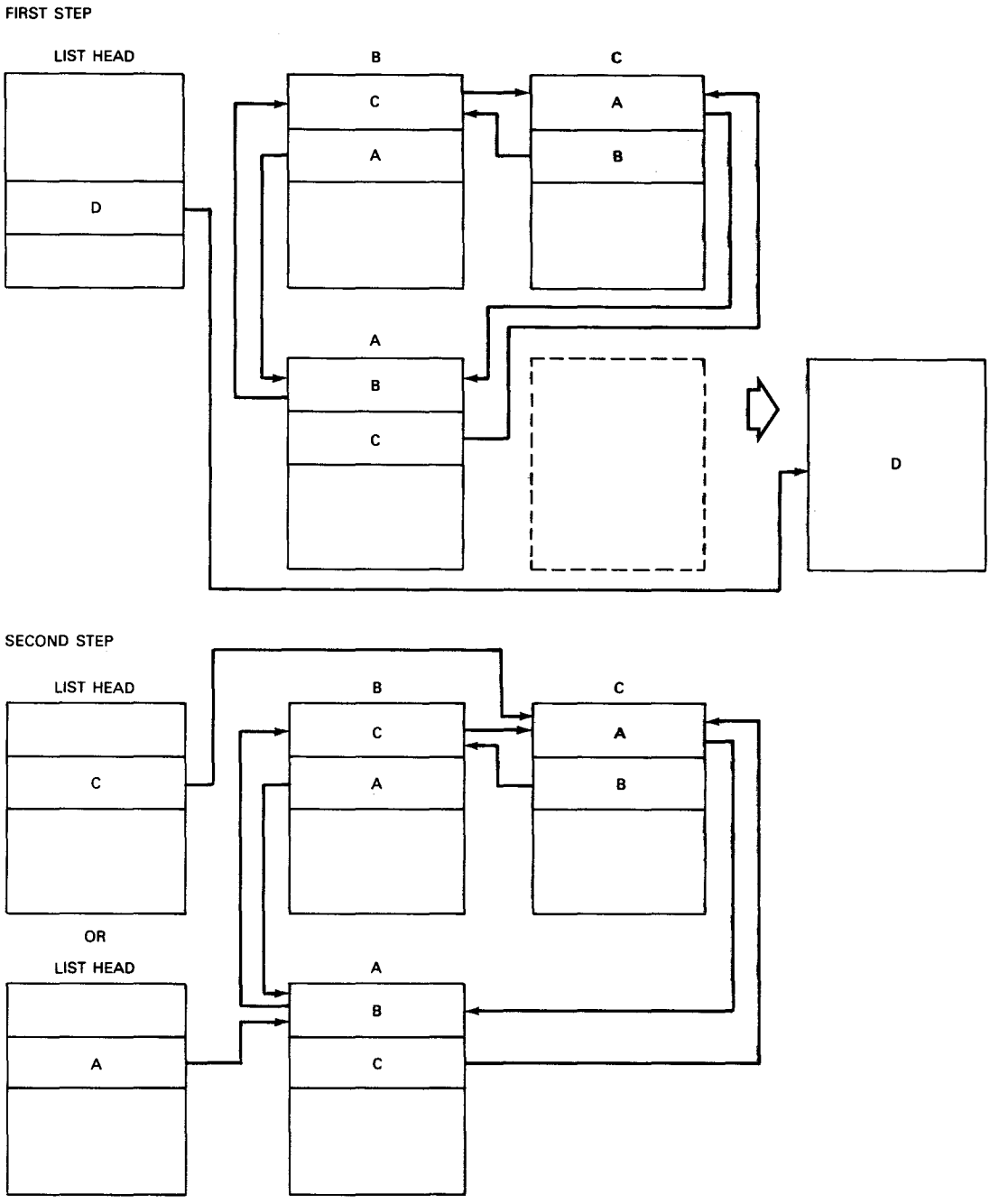
Figure 14—6. Ring List with Station

In a ring list, each element points to its two logically adjacent elements and all the elements together form an endless loop. The list head controls the ring using only its full-word station. As Figure 14—7 shows, no other control is needed.



a. Adding an element

Figure 14—7. Adding and Removing Elements from a Ring List (Part 1 of 2)



b. Removing an element

Figure 14—7. Adding and Removing Elements from a Ring List (Part 2 of 2)

The station determines where elements are added or removed. When you add a new element, it becomes the next element from the one to which the station currently points (Figure 14—7a, step 1). Then, the station is moved so that it finally points to the newly-added element (step 2). When you remove an element it is the element to which the station currently points (Figure 14—7b, step 1). After the element is removed, the station is moved either to the previous (backward pointer) element or the next (forward pointer) element (step 2) depending on how you specify the instruction that removes the element. See 14.5.2.5 for more information.

14.1.5. Priority List

If you want to arrange elements in a list according to some priority where certain elements are more readily available than others, you can use a priority list. Unlike previously described lists, this type does not consist of a single string of elements. Instead, it appears as in Figure 14—8.

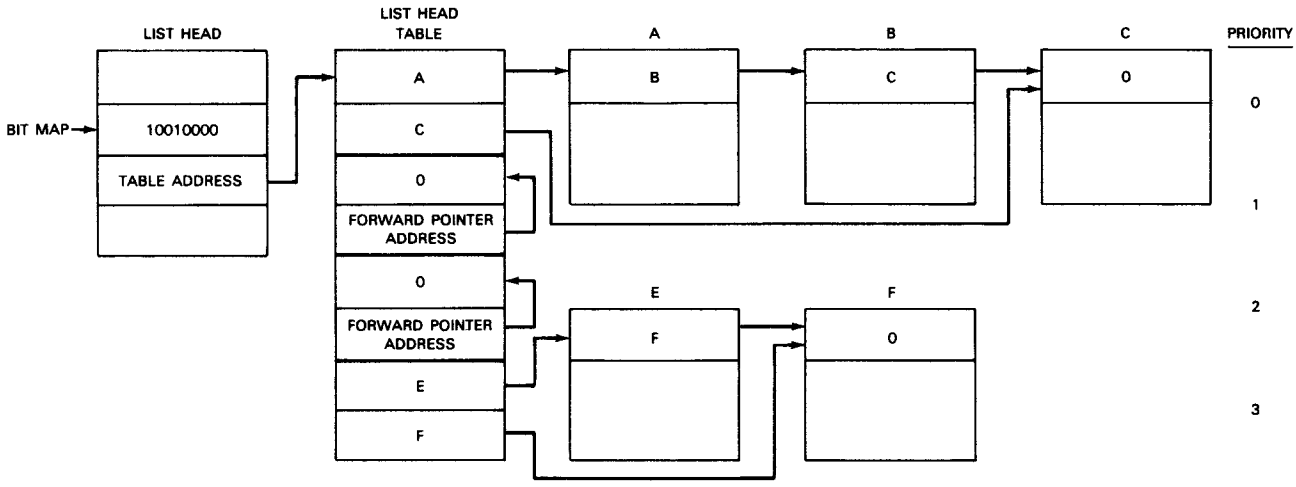


Figure 14—8. Priority List

As you can see from Figure 14—8, list control functions are split between the list head and a separate list head table, both in main storage. Entries within the table each point to a forward-linked FIFO list. The entries are arranged by priority with the first entry (priority 0) having highest priority. Within the list head, a pointer points to the first entry of the table. Above the pointer lies a bit map, a string of bits each of which corresponds to a table entry. A 1 bit means its table entry controls a list with one or more elements, an 0 bit means its table entry controls an empty list. The first bit corresponds to priority 0, the second bit to priority 1, etc. Thus, the bit map in Figure 14—8 indicates that priority levels 0 and 3 control nonempty lists, while priority levels 1 and 2 point to empty lists. As you can see, an empty list entry always has a forward pointer value of 0 and a backward pointer set to the address of the entry itself.

Because more than one priority may be available, you must specify a priority when you add an element to a priority list. If, for example, you want to add an element at priority 1 of the list in Figure 14—8, the resulting list would look like this (Figure 14—9).

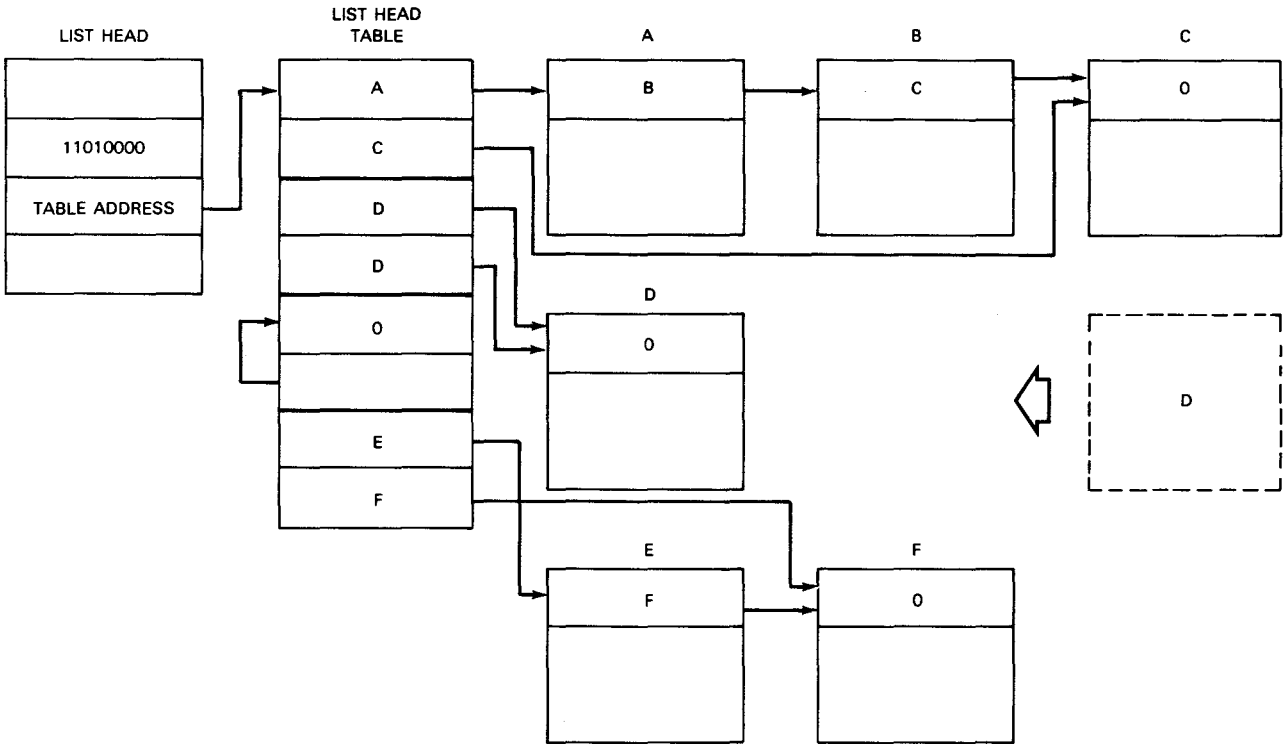


Figure 14—9. Adding to a Priority List

Note that the new element in Figure 14—9 becomes the first and only entry at priority 1 in the list head table. Note, also, that the bit map is updated from the one in Figure 14—8 to reflect the new status of priority 1.

You can remove an element from a specific priority level or you can remove it from the highest priority for which a list exists. In either case, the element pointed to by the forward element pointer in the table entry is the one removed from the priority. If, for example, you remove an element from priority level 3 in Figure 14—9, element E will be the one removed.

You can specify two level stations in the list head (Figure 14—10).

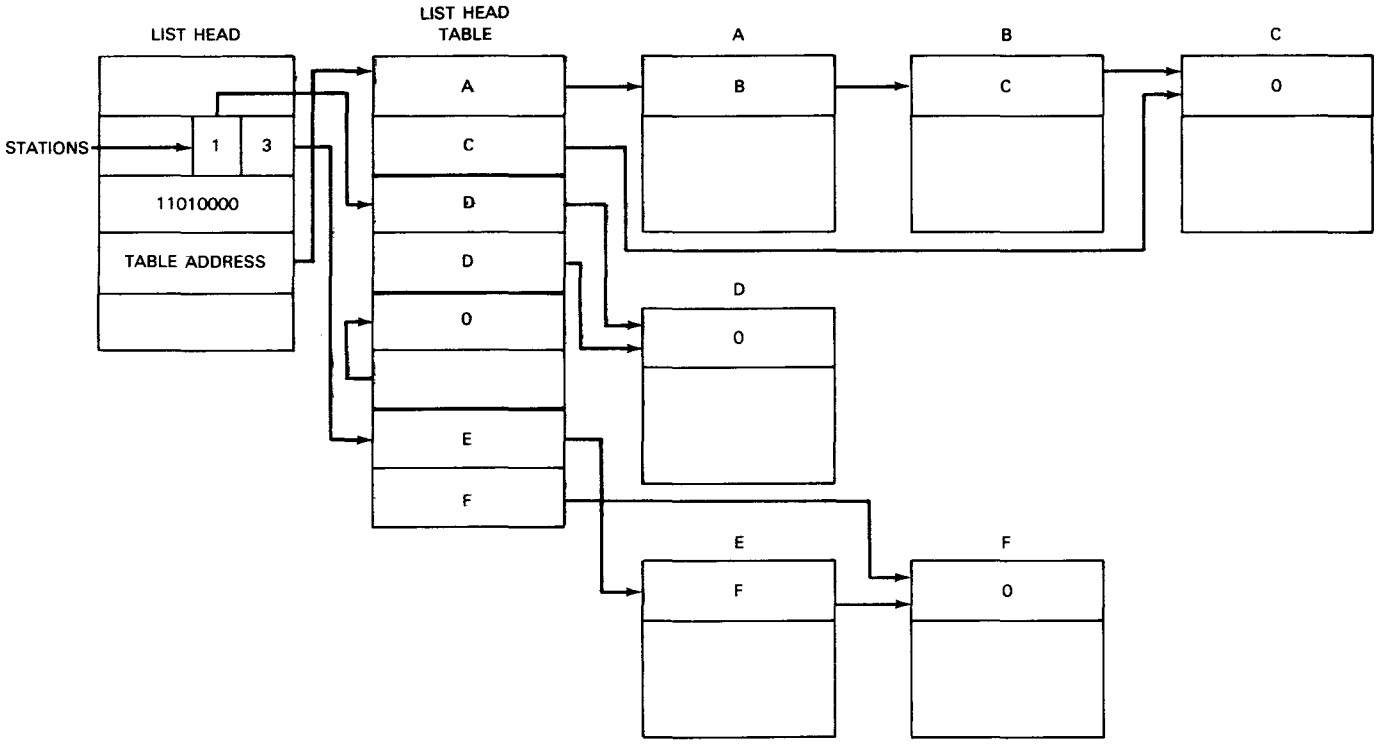


Figure 14—10. Level Stations

Level stations act much like the stations shown with FIFO and ring lists. They allow you to refer to specific parts of the list. Level stations, however, indicate only priority levels in the list head, never specific elements in the list. You can use one station or the other for they operate independently.

You can use a level station to step through a list. When you step forward, you step within a priority from the first to the last element. Then you advance to the next lower priority having a list and continue with its first element. To step backwards (available only with double-linked lists) you proceed from last element to first in a priority level, then move to the next higher priority level with a nonempty list and continue with its last element. Thus, beginning with a level station set to priority 0, you can step forward through the list in Figure 14—10 as follows: A B C D E F.

Although the lists in Figures 14—8 through 14—10 have used forward-linked FIFO lists at each priority level, you are not restricted to using that type. The list head table entries can control any other type of list we have discussed so far — even other priority lists (see 14.1.7). The only restriction is that all priority levels in a list must control the same kind of list.

14.1.6. Aged Priority List

The aged priority list is an extension of the priority list in which a hardware algorithm determines which priority level is used to add or remove an element. When you add an element, the priority you specify only indirectly determines where in the list that element goes. When you remove an element, you do not specify a priority level at all; rather, the algorithm determines which level has highest priority. That level's first element is then removed. The algorithm has been designed to, in effect, move low-priority elements to higher and higher levels as they *age* in the list. We measure an element's age in terms of list activity: the more elements are added and removed, the higher a specific element's priority rises until it becomes the element that is eligible for removal.

The list head and table for an aged priority list is initialized as shown in Figure 14—11.

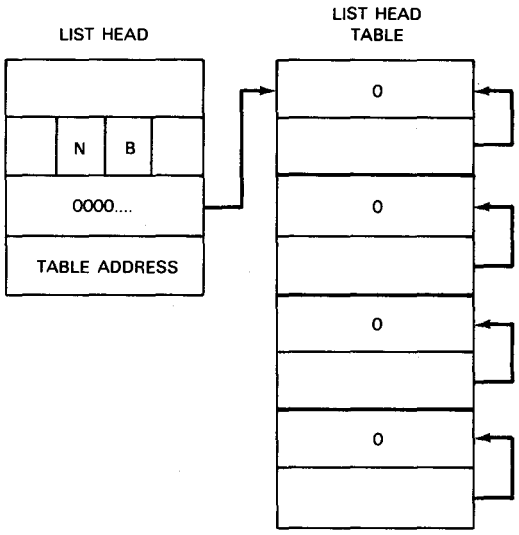


Figure 14—11. Initialized Aged Priority List

Figure 14—11 shows an aged priority list whose table entries each point to an empty forward-linked FIFO list. Two new fields distinguish this from the priority lists discussed in 14.1.5. Located in the list head, they are called the B and N fields. As elements are added and removed these two fields are updated, and they help determine what priority is used at any tiem. The algorithms used are shown, where:

— P is the priority you specify when adding an element;

— I is the list head table entry at which the new element is actually added;

— B is the B field shown in Figure 14—11, a bit map pointer; and

— N is the N field in Figure 14—11, and contains the number of priority levels currently controlling nonempty lists.

- Adding an Element

1. The hardware sets I equal to B+N+P.
2. If priority level I previously had no elements, the hardware increases N by 1 and sets the appropriate bit in the bit map to 1.
3. The new element is added at priority I.

- Removing an Element

1. The hardware searches the bit map from the position given by B until it finds a bit set to 1.
2. The hardware then sets B to the position of the 1-bit just found and removes the first element at that priority level.
3. If this action removes the only element at the level, the corresponding bit in the bit map is set to 0 and N is reduced by 1.

To show the practical effect of these algorithms, Table 14—1 presents a sequence of operations performed on the list of Figure 14—11. The ADD column shows the names of new elements and the priority at which we request they be added. The REMOVE column shows the elements actually removed when requested. For each time interval (t), the values of B and N are shown before and after (B_{t+1}, N_{t+1}) the operation performed during that interval. We initialize B and N to zero.

Table 14—1. Operations with Aged Priority List

TIME	ADD	REMOVE	B_t	N_t	I	B_{t+1}	N_{t+1}
t=0	A(P=2)		0	0	2	0	1
t=1	B(P=0)		0	1	1	0	2
t=2	C(P=2)		0	2	4	0	3
t=3	D(P=1)		0	3	4	0	3
t=4		B(P=0)	0	3	—	1	2
t=5	E(P=0)		1	2	3	1	3
t=6		A(P=2)	1	3	—	2	2
t=7		E(P=0)	2	2	—	3	1

Figure 14—12 shows the status of the list immediately after the ADD operation at t=5.

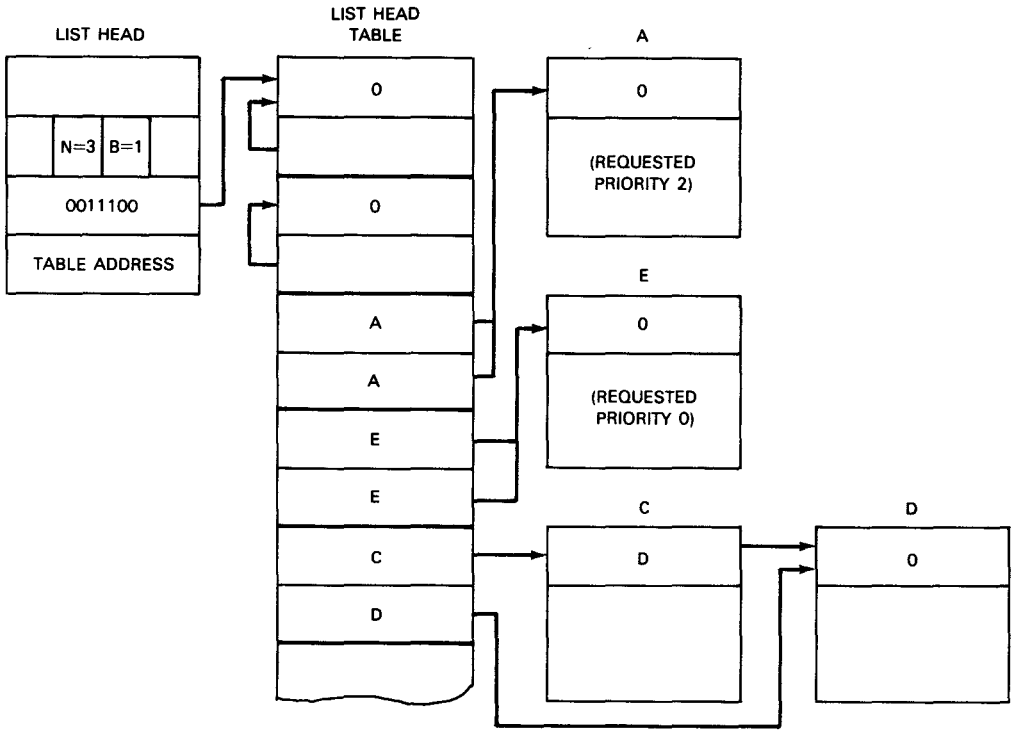


Figure 14—12. Aged Priority List after Table 14—1 Operations

In Table 14—1 we specify that elements A, B, C, and D are to be added at priorities 2, 0, 2, and 1, respectively. However, they get added at priorities 2, 1, 4, and 4, all successive values of I. Note that element A is removed before element E, even though we gave A a lower priority (2) than E (0). The difference is that we added A at t=0 but waited until t=5 to add E. The hardware algorithms ensure that if A stays in the list long enough it will become eligible for removal before E does, regardless of the relative priorities we give them. To familiarize yourself with aged priority lists you should work through the sequence of Table 14—1 to assure yourself that Figure 14—12 is correct.

Like a priority list, you can control any other type of list from an aged priority list; all such lists need only be the same type. Unlike a priority list, however, you can neither use level stations with nor step through an aged priority list.

14.1.7. Two-Level List

As discussed earlier, priority and aged priority list heads can control other lists, even other priority and aged priority lists. These are called two-level lists because each element in the list is controlled through two priority levels. A typical two-level list is shown in Figure 14—13.

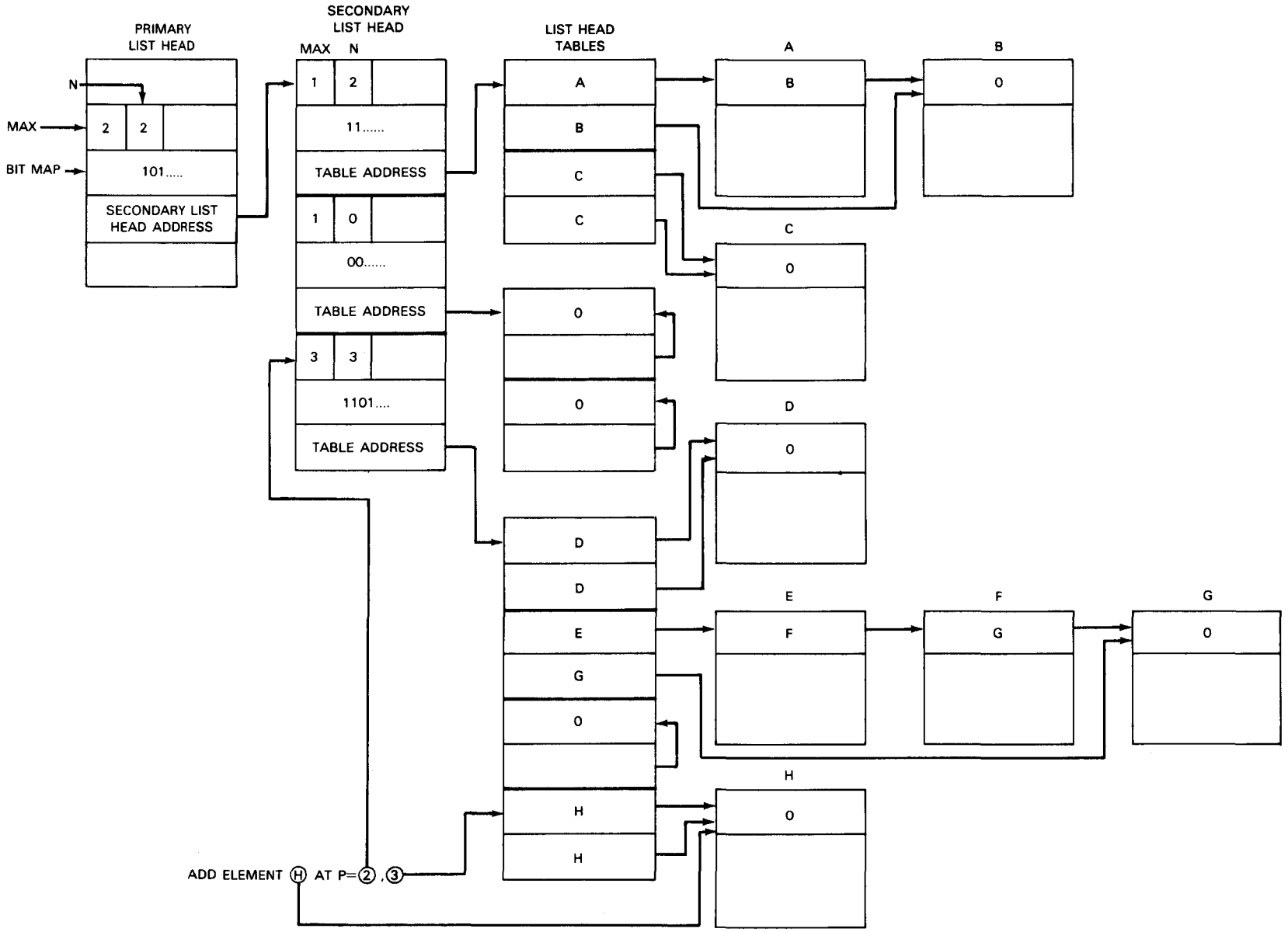


Figure 14—13. Two-Level List

Figure 14—13 shows a typical two-level priority list. The primary list head points to a three-member secondary list head instead of to the list head table seen in single-level lists. The MAX field in the primary list head defines the highest-numbered (lowest) priority the secondary list head can have. In Figure 14—13, the MAX value of the primary list head is 2, indicating a maximum of three levels in the secondary list head.

Within the secondary list head, entries are three full words long and have the same format as the three full words we highlight in the primary list head; that is, each has a MAX field, an N field, a bit map, and a pointer to its own list head table. These fields completely define their list head tables which, in turn, completely define their lists.

To add an element you specify a primary and a secondary priority. To add element H in Figure 14—13, for example, we specify a primary priority of 2 and a secondary priority of 3. As a result, H is entered in the same list head table as elements D, E, F, and G, but has its own priority.

List head tables in a two-level list must control FIFO lists and no other types. Therefore, a new element J at primary level 0, secondary level 1, would be added immediately following element C.

Because only FIFO lists are allowed with two-level priority lists, the element removed from a given priority level is always the one pointed at by the forward pointer of the priority's table entry. You can optionally specify either a primary or secondary (or both) priority for element removal. When one is not specified, the highest priority pointing to a nonempty list or secondary list head is used. Thus, if you do not specify any priority, the elements in Figure 14—13 are removed in this order: A B C D E F G H. The same sequence is followed if you step a level station forward through the list.

Figure 14—13 shows a priority list head pointing to three other priority list heads. You can construct two-level lists in which the secondary list head defines any type of list we use. In the most general case, the primary list head selects the secondary list head entry it points to according to its type (priority or aged priority) and the secondary entry then selects its list head table entry according to its type. The only restriction on a secondary list head is that all its entries must be of the same type.

14.2. WHAT IS NEEDED FOR LIST PROCESSING?

As mentioned previously, System 80 instructions give you the capability of handling list types from simple to complex. Now that you know what lists you can use, we outline here what you need to do to set up lists and list processing programs.

14.2.1. What System 80 Provides

Given proper initialization, three System 80 instructions can build and manipulate any of the list types discussed in 14.1:

- Enqueue

  Adds a new element to a list by altering pointers within the element and the list. Options include data movement and register storage.

- Dequeue

  Removes an element from a list by altering pointers within the element and the list. Options include data movement and register loading.

- Step Queue

  Uses element or level stations to move forward or backward in a list. Options include execution of a subprogram designed for element manipulation.

14.2.2. What You Must Provide

For each list you must reserve an area in main storage for its elements and for its list head. In System 80, a list head is part of a structure called a list control block (LCB). Besides the pointers, table addresses, bit maps and other fields we have discussed, LCB fields define the elements you use, determine how list operations are to execute, and enable the options available with each list processing instruction. In 14.3, we discuss the LCB format; in 14.4, the System 80 list processing instructions in detail; in 14.5, how to initialize and manipulate System 80-supported lists; and in 14.7, how to use the list processing options.

14.3. LIST CONTROL BLOCK

The format of the LCB is shown in Figure 14—14.

The LCB must be on a full-word boundary. All shaded fields must be set to zero. The following explains the use of each LCB field:

- List Type (word 0, bits 0—3)

  You use this field to specify the type of list the LCB controls. Permitted values are shown in Table 14—2.

BIT

WORD	0	4	8	12	16	20	24	28 31
0	LIST TYPE	ELEMENT TYPE	LIST CONTROLS		LIST HEAD TYPE	r_3	r_4	r_5
1	LIST IDENTIFICATION							
2	COMPONENT LINKAGE TYPE	0		0	0 0 0 0	0		0
3	r_1	r_2	0					0
4	POINTER TYPE	OFFSET TO NEXT ELEMENT POINTER			0 0 0 0	OFFSET TO PREVIOUS ELEMENT POINTER		
5	0			0	PRIORITY FIELD SIZE	OFFSET TO PRIORITY STORAGE		
6	0 0 0 0	OFFSET TO FLOATING-POINT REGISTER SAVE AREA			0 0 0 0	OFFSET TO GENERAL REGISTER SAVE AREA		
7	0 0 0 0	DATA AREA SIZE			0 0 0 0	OFFSET TO DATA AREA		
8	MINIMUM ELEMENT THRESHOLD							
9	MAXIMUM ELEMENT THRESHOLD							
10	CURRENT ELEMENT COUNT							
11	0							0
12	LIST HEAD FIELDS							
13	(SPECIFIC FIELDS DETERMINED BY							
14	TYPE LIST USED)							
15	0	0	FREE ELEMENT LCB ADDRESS					

Figure 14—14. LCB Format

Table 14—2. List Type Values

Value	List Type
0	FIFO
1	FIFO with station
2	LIFO
3	Double-ended
4	Ring with station
6	Priority
7	Aged Priority

- Element Type (word 0, bits 4—7)

  You use this field to specify the number of pointers each element in your list uses. The only permitted values are binary 0 for forward linkage and binary 1 for double linkage. Only certain combinations of list and element types are permitted in System 80; these are shown in Table 14—3.

Table 14—3. Permissible Element/List Type Combinations

List Type	Element Type	
	Forward Linkage	Double Linkage
FIFO	P	P
FIFO with station	P	P
LIFO	NP	P
Double-ended	NP	P
Ring with station	NP	P
Priority	List head table entry	
Aged priority	List head table entry	

LEGEND:

P = permitted
NP = not permitted

- List Controls (word 0, bits 8—15)

  You use this eight-bit field to control certain functions and options of the ENQUEUE, STEP QUEUE, and DEQUEUE instructions:

  — Bit 8 is a list lock. A list processing instruction sets this lock to 1, which effectively prevents other instructions from accessing the list while execution continues.

  — Bit 9 controls the accessibility of the list. When it is set to 0, only programs running under the supervisor state (program status word (PSW) bit 14 set to 0) can use the list. When bit 9 is set to 1, programs running under either supervisor or problem states can use this list. For your programs you should set this bit to 1.

  — Bit 10 determines where a station is moved after a DEQUEUE instruction. If you DEQUEUE an element to which the station is currently pointing, the instruction resets the pointer forward to the next element (bit 10=1) or backward to the previous element (bit 10=0).

  — Bit 11 is not used and must be set to 0.

  — Bit 12 controls the data movement option. For its use refer to 14.7.2.

  — Bits 13—15 are used with the register load/store option. For their use refer to 14.7.1.

- List Head Type (word 0, bits 16—19)

  When you use a priority or aged priority list you must use this field to specify what type of entry goes in the list head table (or secondary list head for two-level lists). Table 14—4 shows the permitted settings for this field, as well as the size in full words of each type of entry.

Table 14—4. List Head Type Values

Value	List Type	Table Item Length (full words per entry)
0	FIFO	2
1	FIFO with station	3
2	LIFO	2
3	Double-ended	2
4	Ring with station	1
6	Priority	3
7	Aged priority	3

Using Table 14—4 you can determine that the list head value for the priority list in Figure 14—9 would be 0 (FIFO), while the value for the priority list in Figure 14—13 would be 6 (priority list head).

- r_3 (word 0, bits 20—23)

  You use this 4-bit field to specify the even register of an even-odd register pair. If you want to add a specific element to the list, you put its address into bits 8—31 of the even-numbered register, putting zeros into bits 0—7. And when you remove an element, you can, under certain conditions, specify the element you want to remove even if it is not normally the element affected. For example, if it lies in the middle of a FIFO list, you simply put its address into the even-numbered register. For all DEQUEUE operations the address of the newly removed element is placed in the even-numbered register.

  The odd-numbered register of the pair holds the priority values you use when manipulating a priority or aged priority list. Bits 8—15 hold the primary value and bits 24—31 hold the secondary value (if one is used). The maximum values for either field is decimal 255 (FF_{16}).

- r_4 (word 0, bits 24—27)

  You use this 4-bit field to specify an even-odd register pair that you use with the data movement option. For more information refer to 14.7.2.

- r_5 (word 0, bits 28—31)

  You use this 4-bit field to specify an even-odd register pair that is primarily used in list control programs. For more information refer to 14.8.

- List Identification (word 1, bits 0—31)

  You can use this full word to identify the LCB and its list by putting any data you wish into it.

- Component Linkage (word 2, bits 0—3)

  You use this 4-bit field to specify what type of linkage you want. Its permitted values are:

  — 1 for logical address format; or

  — 2 for I/O directive format.

  For programs running in the problem state you would specify 0. In that format each pointer occupies a full word of the element with bits 0—7 containing zeros and bits 8—31 containing the 24-bit logical address of the element (or list head) to which it is pointing.

- r_1/r_2 (word 3, bits 0—7)

  You use these two fields (r_1 — bits 0—3, and r_2 — bits 4—7) with the register load/store option. For more information, see 14.7.1.

- Pointer Type (word 4, bits 0—3)

  You use this 4-bit field to specify the type of linkage you want in your list. Its value should be the same as that of the COMPONENT LINKAGE field previously discussed.

- Offset to Next Element Pointer (word 4, bits 4—15)

  You use this 12-bit field to specify the offset, in bytes, from the beginning of the element to the first byte of its next (forward) element pointer. This offset must result in a pointer that resides on a full-word boundary. The maximum permitted offset is 4095 (FFF_{16}) bytes.

- Offset to Previous Element Pointer (word 4, bits 20—31)

  You use this 12-bit field the same way as the NEXT ELEMENT POINTER offset, except that this offset specifies the location of the previous (backward) element pointer if one is used. The location must be on a full-word boundary and the maximum permitted offset is 4095 (FFF_{16}) bytes.

- Numeric Field Size (word 5, bits 16—19)

  You use this field to specify the size of the priority field (or fields) used in each priority list head. The only acceptable value is 0, indicating a size of one byte.

- Offset to Priority Storage (word 5, bits 20—31)

  You use this 12-bit field to specify the offset from the start of an element to its priority save area. Every time you add an element to a priority list, the ENQUEUE instruction puts the priority value (or values) of that element in its priority save area. The maximum permitted offset is 4095 (FFF_{16}) bytes.

- Offset to Floating-Point Register Save Area (word 6, bits 4—15)

  You use this field with the register load/store option. For more information, refer to 14.7.1.

- Offset to General Register Save Area (word 6, bits 20—31)

  You use this field with the register load/store option. For more information, refer to 14.7.1.

- Data Area Size (word 7, bits 4—15)

  You use this field to specify, in bytes, how large a data area your element is to have. The maximum permitted size is 4095 (FFF_{16}) bytes.

- Offset to Data Area (word 7, bits 20—31)

  You use this field to specify where in an element its data area is to start. You specify this as an offset, in bytes, from the start of the element. The maximum permitted offset is 4095 (FFF_{16}) bytes.

- Minimum Element Threshold (word 8, bits 0—31)

  You use this 32-bit field to specify a minimum number of elements in your list. If the CURRENT ELEMENT COUNT field drops to this value, the condition code is set to indicate an underflow condition (see 14.4.2).

- Maximum Element Threshold (word 9, bits 0—31)

  You use this 32-bit field to specify a maximum number of elements in your list. If the CURRENT ELEMENT COUNT field increases to this value, the condition code is set to indicate an overflow condition (see 14.4.1).

- Current Element Count (word 10, bits 0—31)

  The list processing instructions use this 32-bit field to indicate how many elements are currently in a list. It must be initialized to zeros.

- List Head Fields (words 12—14)

  You use these fields to point to and control your list. These fields have different formats for different types of lists. Refer to 14.5.2 for the particular format you want to use.

- Free Element LCB Address (word 15, bits 0—31)

  You use this pointer when you use a free element list (FEL). Refer to 14.6 for more information.

14.4. LIST PROCESSING INSTRUCTIONS

We discuss here the three System 80 instructions that you use for list processing. We do not discuss their options here; refer to 14.7 for more information on them.

ENQ

14.4.1. ENQUEUE (ENQ)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ENQ	**B3**	**SI**	**4**

Condition Codes

- ■ SET TO 0
- ■ SET TO 1
- ■ SET TO 2
- ■ SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

- ■ ADDRESSING
- ☐ DATA (INVALID SIGN/DIGIT)
- ☐ DECIMAL DIVIDE
- ☐ DECIMAL OVERFLOW
- ☐ EXECUTE
- ☐ EXPONENT OVERFLOW
- ☐ EXPONENT UNDERFLOW
- ☐ FIXED-POINT DIVIDE
- ☐ FIXED-POINT OVERFLOW
- ☐ FLOATING-POINT DIVIDE
- ☐ OPERATION
- ■ PROTECTION
- ☐ SIGNIFICANCE
- ■ SPECIFICATION:
  - ☐ NOT A FLOATING-POINT REGISTER
  - ■ OP 1 NOT ON FULL-WORD BOUNDARY
  - ☐ OP 2 NOT ON HALF-WORD BOUNDARY
  - ☐ OP 2 NOT ON FULL-WORD BOUNDARY
  - ☐ OP 2 NOT ON DOUBLE-WORD BOUNDARY
  - ☐ OP 1 NOT EVEN NUMBERED REGISTER
  - ☐ OP 1 NOT ODD NUMBERED REGISTER
- ☐ OTHERS, SEE TEXT

Explicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	ENQ	$d_1(b_1),i_2$

Implicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	ENQ	s_1,i_2

This instruction adds an element to the list whose LCB is addressed by operand 1. i_2 is an immediate operand, a self-defining term that is assembled into an 8-bit control field occupying bits 8—15 of the object instruction. The bits in the i_2 field have the following functions:

- Bits 8—9

  You use these bits to determine the source field of the element. You specify 00_2 to indicate that the even-numbered register of the LCB r_3 field contains the address of the element. You specify 01_2 to indicate that the element is to come from the free element list (see 14.6); in this case the instruction itself loads the even-numbered r_3 register with the address of the newly added element.

- Bit 10

  For a double-ended list, you use this bit to specify the end of the list to which you want to add the new element. To add to the left (first element) end, specify 1, to the right end, a 0. For all other list types, this bit must be set to 0.

- Bit 11

  This bit must be set to 0 or a specification exception will occur.

- Bit 12

  You can enable the data movement option (see 14.7.2) by setting both this bit and bit 12 of LCB word 0 to 1.

- Bits 13—15

  You can enable the register load/store option and specify how it is to execute by setting these three bits, on which a logical AND function is performed with bits 13—15 of LCB word 0, and the resulting three bits determine how the option is to run, as explained in 14.7.1.

Condition Code:

After execution of the ENQUEUE instruction:

- The condition code is set to 0 if the element is successfully added to the list.

- The condition code is set to 1 if the updated CURRENT ELEMENT COUNT of the LCB reaches the MAXIMUM ELEMENT THRESHOLD value. This result only sets the condition code; further enqueueing is possible until the current count reaches a value of $FFFFFFFF_{16}$ ($4,294,967,295_{10}$) elements.

- The condition code is set to 2 if the list is currently unavailable (its LOCK CONTROL bit set to 1).

- The condition code is set to 3 if the enqueue operation is unsuccessful. This can happen if you try to add an element to a list that already contains the maximum allowable number of elements ($FFFFFFFF_{16}$ or $4,294,967,295_{10}$), remove an element from an empty free element list for enqueueing, or cause an aged priority list to contain more nonempty priority levels than its maximum allows.

14.4.2. DEQUEUE (DEQ)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
DEQ	B4	SI	4

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
■ SET TO 3
□ UNCHANGED

Possible Program Exceptions	
■ ADDRESSING	□ PROTECTION
□ DATA (INVALID SIGN/DIGIT)	□ SIGNIFICANCE
□ DECIMAL DIVIDE	■ SPECIFICATION:
□ DECIMAL OVERFLOW	□ NOT A FLOATING-POINT REGISTER
□ EXECUTE	■ OP 1 NOT ON FULL-WORD BOUNDARY
□ EXPONENT OVERFLOW	□ OP 2 NOT ON HALF-WORD BOUNDARY
□ EXPONENT UNDERFLOW	□ OP 2 NOT ON FULL-WORD BOUNDARY
□ FIXED-POINT DIVIDE	□ OP 2 NOT ON DOUBLE-WORD BOUNDARY
□ FIXED-POINT OVERFLOW	□ OP 1 NOT EVEN NUMBERED REGISTER
□ FLOATING-POINT DIVIDE	□ OP 1 NOT ODD NUMBERED REGISTER
■ OPERATION	■ OTHERS – SEE TEXT

Explicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DEQ	$d_1(b_1),i_2$

Implicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DEQ	s_1,i_2

This instruction removes an element from the list whose LCB is addressed by operand 1. i_2 is an immediate operand, a self-defining term that is assembled into an 8-bit control field occupying bits 8—15 of the object instruction. The bits in the i_2 field have the following functions:

- Bits 8—9

  You use these bits to determine how to remove the element. You code a 00_2 to simply remove the element and place its address in the even-numbered r_3 register. You code a 01_2 to indicate that the newly dequeued element is to be added to the free element list (see 14.6).

- Bit 10

  For a double-ended list, you use this bit to specify the end of the list from which you want to remove the element. To remove from the left (first element) end, specify a 1, from the right (last element) end, a 0. For all other list types this bit must be set to 0.

- Bit 11

  This bit must be set to 0 or a specification exception will occur.

- Bit 12

  You can enable the data movement option (see 14.7.2) by setting both this bit and bit 12 of LCB word 0 to 1.

- Bits 13—15

  You can enable the register load/store option and specify how it is to execute by setting these three bits, on which a logical AND function is performed with bits 13—15 of LCB word 0, and the resulting three bits determine how the option is to run, as explained in 14.7.1.

Condition Code:

After execution of the DEQUEUE instruction:

- The condition code is set to 0 if the element is sucessfully dequeued.

- The condition code is set to 1 if the updated CURRENT ELEMENT COUNT of the LCB reaches the MINIMUM ELEMENT THRESHOLD value. This result only sets the condition code; further dequeueing is possible until the current count reaches a value of 0. If you use a free element list, its CURRENT ELEMENT COUNT field is increased by 1 for each element dequeued; if that value reaches the MAXIMUM ELEMENT THRESHOLD value of the free element list, the condition code is set to 1. See 14.6 for more information.

- The condition code is set to 2 if the list is currently unavailable (its lock control bit is set to 1).

- The condition code is set to 3 if the dequeue operation is unsuccessful. This can happen if you try to remove an element from an empty list or add a newly-removed element to a free element list that already contains the maximum allowable number of elements ($FFFFFFFF_{16}$ or $4{,}294{,}967{,}295_{10}$ elements).

14.4.3. STEP QUEUE (STEP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STEP	B5	SI	4

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
■ SET TO 3
☐ UNCHANGED

Possible Program Exceptions	
■ ADDRESSING	■ PROTECTION
☐ DATA (INVALID SIGN/DIGIT)	☐ SIGNIFICANCE
☐ DECIMAL DIVIDE	■ SPECIFICATION:
☐ DECIMAL OVERFLOW	☐ NOT A FLOATING-POINT REGISTER
☐ EXECUTE	☐ OP 1 NOT ON HALF-WORD BOUNDARY
☐ EXPONENT OVERFLOW	☐ OP 2 NOT ON HALF-WORD BOUNDARY
☐ EXPONENT UNDERFLOW	■ OP 1 NOT ON FULL-WORD BOUNDARY
☐ FIXED-POINT DIVIDE	☐ OP 2 NOT ON DOUBLE-WORD BOUNDARY
☐ FIXED-POINT OVERFLOW	☐ OP 1 NOT EVEN NUMBERED REGISTER
☐ FLOATING-POINT DIVIDE	☐ OP 1 NOT ODD NUMBERED REGISTER
☐ OPERATION	☐ NONE

Explicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	STEP	$d_1(b_1),i_2$

Implicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	STEP	s_1,i_2

This instruction moves a station forward or backward one position (forward only in a forward-linked list). It optionally calls a list control program (LCP) of your own design (14.8). The LCB of the list you use is addressed by operand 1. i_2 is an immediate byte, a self-defining term that is assembled into an 8-bit control field in bits 8—15 of the object instruction. The bits in the i_2 field have the following functions:

- Bits 8—9

  You determine the specific STEP QUEUE function you want with these bits. Possible values are:

  — 00_2 meaning move the selected station forward one position;

  — 01_2 meaning move the selected station backward one position;

— 10_2 meaning execute an LCP with forward station movement if necessary; or

— 11_2 meaning execute an LCP with backward station movement if necessary. If you specify 00_2 or 01_2 for a list that has no station, a specification exception will result. If you specify 10_2 or 11_2 for such a list, the instruction uses the even-numbered r_3 register as a station. See 16.8 for more information on LCPs.

- Bit 10

  If you are using a priority list you select the priority station (primary station if using a two-level list) you want modified: 0 for level station 1 or 1 for level station 2.

- Bit 11

  If you are using a two-level priority list you select the secondary priority station you want modified with this bit: 0 for level station 1 or 1 for level station 2.

- Bits 13—15

  If you are moving a station (bit 8 set to 0), you must set all these bits to 0. If you are executing an LCP (bit 8 set to 1), you can use these bits as described in 14.8.2.5 and 14.8.2.6.

There are restrictions on the use of the STEP QUEUE instruction. You cannot use it with an aged priority list. For all other list types you can step in at least one direction, forward. To step backwards, though, your list must be double-linked or a specification exception will result.

Condition Code:

After execution of the STEP QUEUE instruction the condition code is set:

- to 0 if the step is successful;
- to 2 if the list lock is set, making the list unavailable; or
- to 3 if the step is unsuccessful; this can happen because the list is empty or because the station has already moved as far as possible in the direction you want, for example, forward past the last element in a FIFO list.

If you execute a list control program it can further modify the condition code; see 14.8.

14.5. INITIALIZING AND USING SYSTEM 80 LISTS

Having outlined System 80 list and LCB formats, we show in this section how you must initialize each type of list. In addition, we show how you can use the ENQUEUE, DEQUEUE, and STEP QUEUE instructions with each.

14.5.1. Specifying Elements

For every type of list, you have to specify the format of its elements using the LCB of the list. In specifying the format of an element, you must distinguish between the element's:

- pointers;
- optional register save areas (see 14.7.1);
- priority save area (for priority lists); and
- data area.

You should put element pointers in one contiguous block, save areas in another block, and data area in a third. Doing this will help prevent the accidental alteration of data that could ruin a list. An example of laying out an element is shown in Figure 14—15.

In Figure 14—15, each area in the element to the left is specified by a corresponding field in the LCB shown to the right. These take the form of offsets from the first byte of the element. In addition, the linkage type field in word 4, bits 0—3 is set to binary 1 indicating that both pointers in the element are 24-bit logical addresses. Also, the length of the data field is shown in word 7, bits 14—15. To confirm that the LCB is initialized correctly, refer to the LCB format of Figure 14—14.

14.5.2. Specifying Lists by Type

We show here how you correctly initialize an LCB for each of the different types of System 80 lists. We also summarize how you can use the ENQUEUE, DEQUEUE, and STEP QUEUE instructions with each type.

14.5.2.1. LIFO List Usage

To initialize a LIFO list, you initialize its LCB as shown in Figure 14—16.

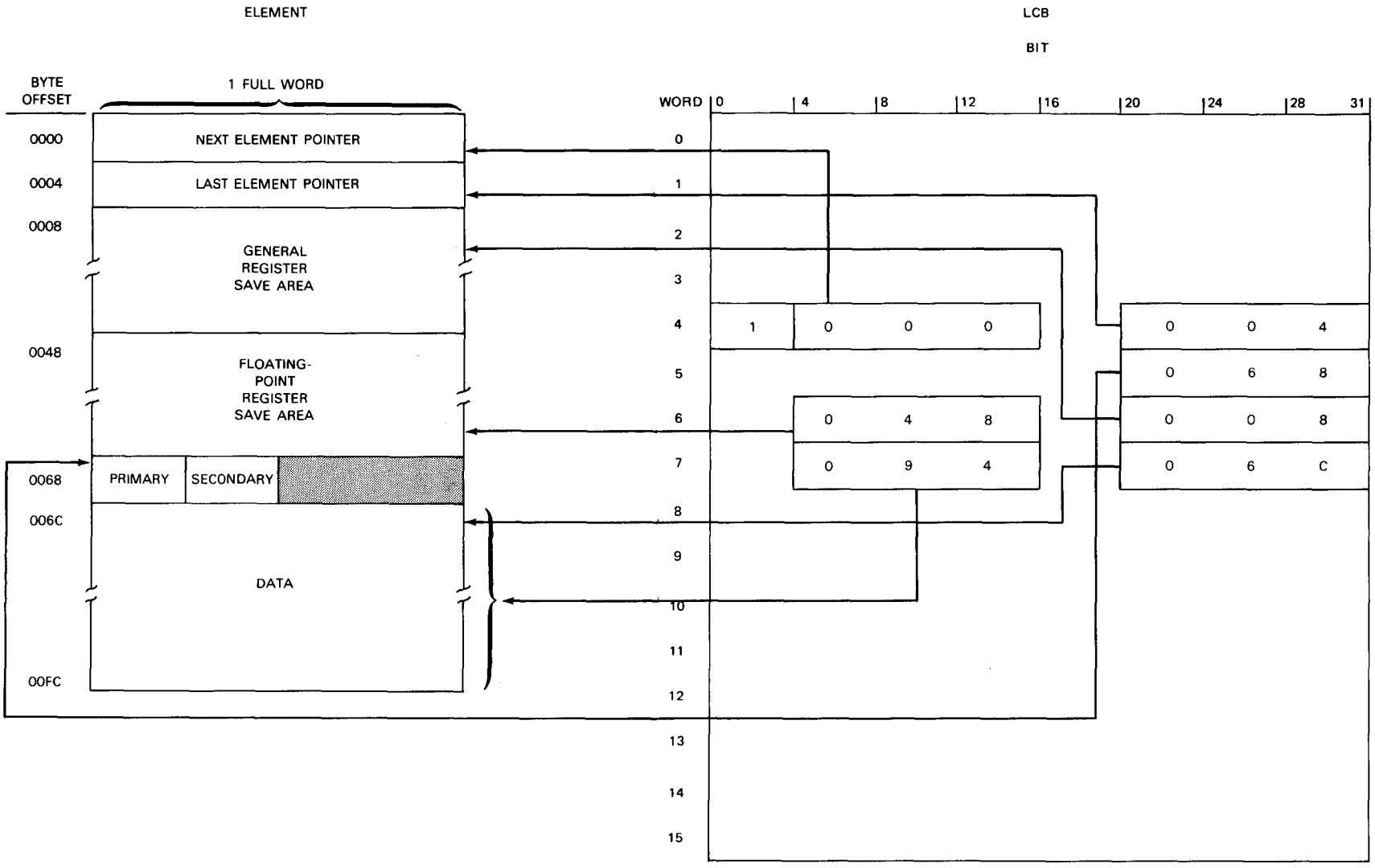


Figure 14—15. Specifying an Element in an LCB

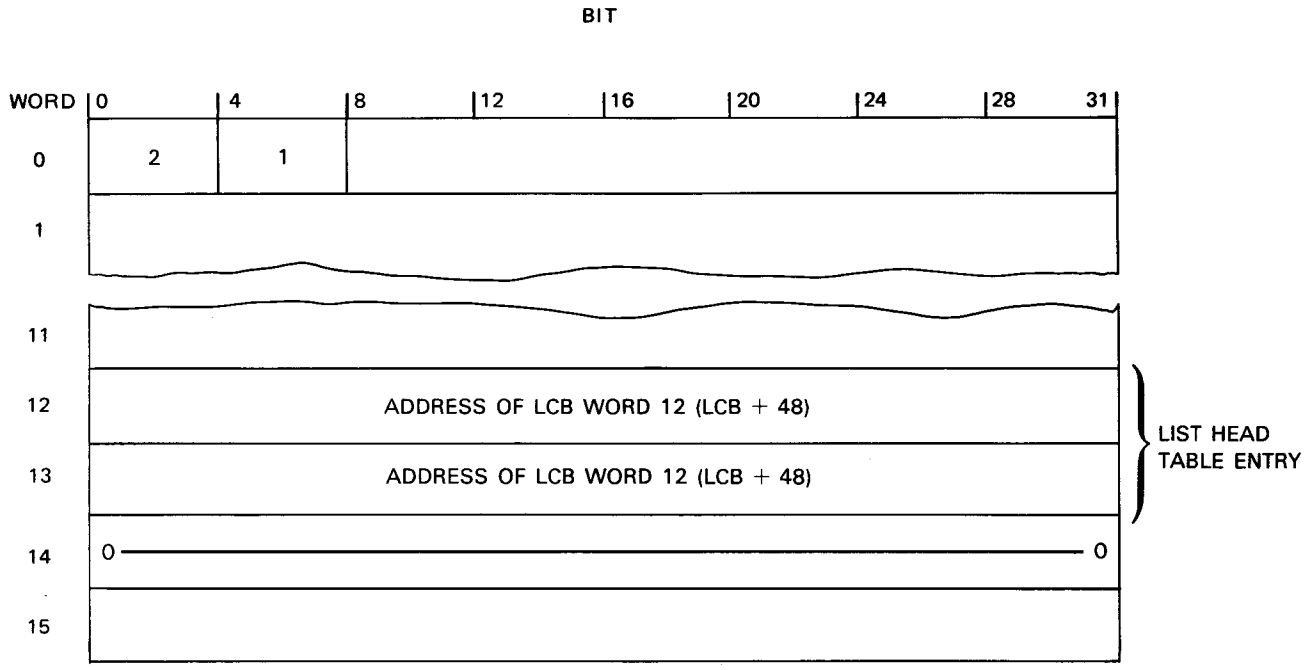


Figure 14—16. Initializing LIFO LCB

The list head type in word 0, bits 0—3 must be 2. The element type in bits 4—7 must be binary 1. The address of word 12 must be placed in words 12 and 13. Word 14 must be set to all zeros. If the LIFO list is being controlled by a list in a priority list, the first entry of its list head table, which corresponds to LCB words 12 and 13, must be initialized with the entry's address in both of its full words.

You can only add new LIFO elements between the list head and its first element. If you put zeros into the even-numbered r_3 register and remove an element from a LIFO list, it is the first element that gets removed. If, however, the even-numbered r_3 register contains the address of any element in the list, that element is the one removed by a DEQUEUE instruction. You can use the even-numbered r_3 register as a station in STEP QUEUE operations; if you initialize it with the address of any element, STEP QUEUE replaces it with the address of the next or previous element depending on the direction of the step.

14.5.2.2. FIFO List Usage

To initialize a FIFO list you initialize its LCB as shown in Figure 14—17.

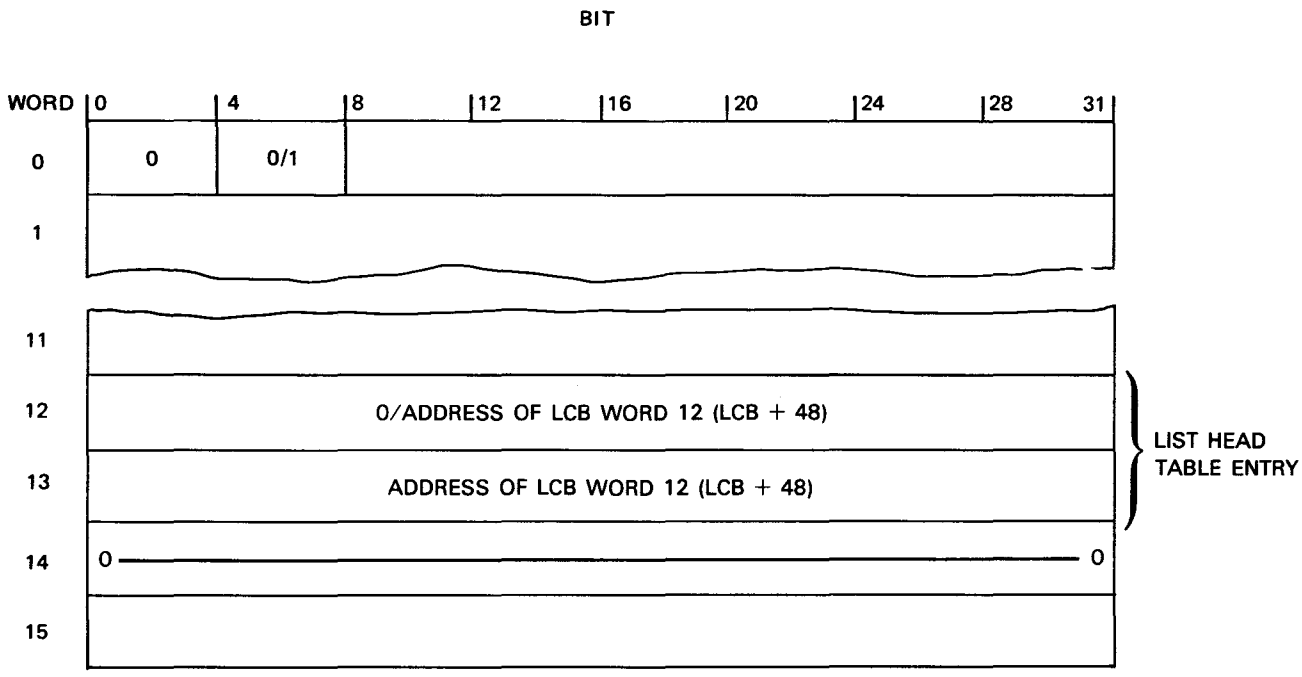


Figure 14—17. Initializing FIFO LCB

Word 0, bits 0—3 of the LCB must contain 0. Bits 4—7 of word 0 can contain either 0 (forward linkage) or 1 (double linkage). If the list is forward-linked, word 12 must contain all zeros; if double-linked, it must contain the address of word 12. Word 13 must always be initialized with the address of word 12. Word 14 must contain all zeros.

If the FIFO list is controlled by a list head table entry, that entry is two full words long; the first word must contain all zeros (for forward-linked lists) or the address of the table entry (for double-linked lists). The second word must contain the address of the first word.

You can add an element to a FIFO list only at the end of the list farthest away from the list head. When you use the DEQUEUE instruction it is usually the first element that is removed. In this case you put all zeros into the even-numbered r_3 register. If the list is double-linked, however, you can remove any element by loading the even-numbered r_3 register with the element address, then executing DEQUEUE. You can also use the even-numbered r_3 register as a station using STEP QUEUE to move the station one element forward or backward.

14.5.2.3. Double-ended List Usage

You initialize a double-ended list in much the same way as a double-linked FIFO list (Figure 14—17). The only difference is that the value you enter in word 0, bits 0—3 must contain a 3 to indicate this type of list. If the double-ended list is controlled by a list head table, its entry is two full words long. You can add or remove elements at either end of the list. You can also remove any element in the list by loading the even-numbered r_3 register with the element address before executing DEQUEUE. Additionally, you can use the even-numbered r_3 register as a station, moving it back and forth through the list, one element at a time, with STEP QUEUE.

14.5.2.4. FIFO with Station Usage

To initialize a FIFO list with station you initialize its LCB as shown in Figure 14—18.

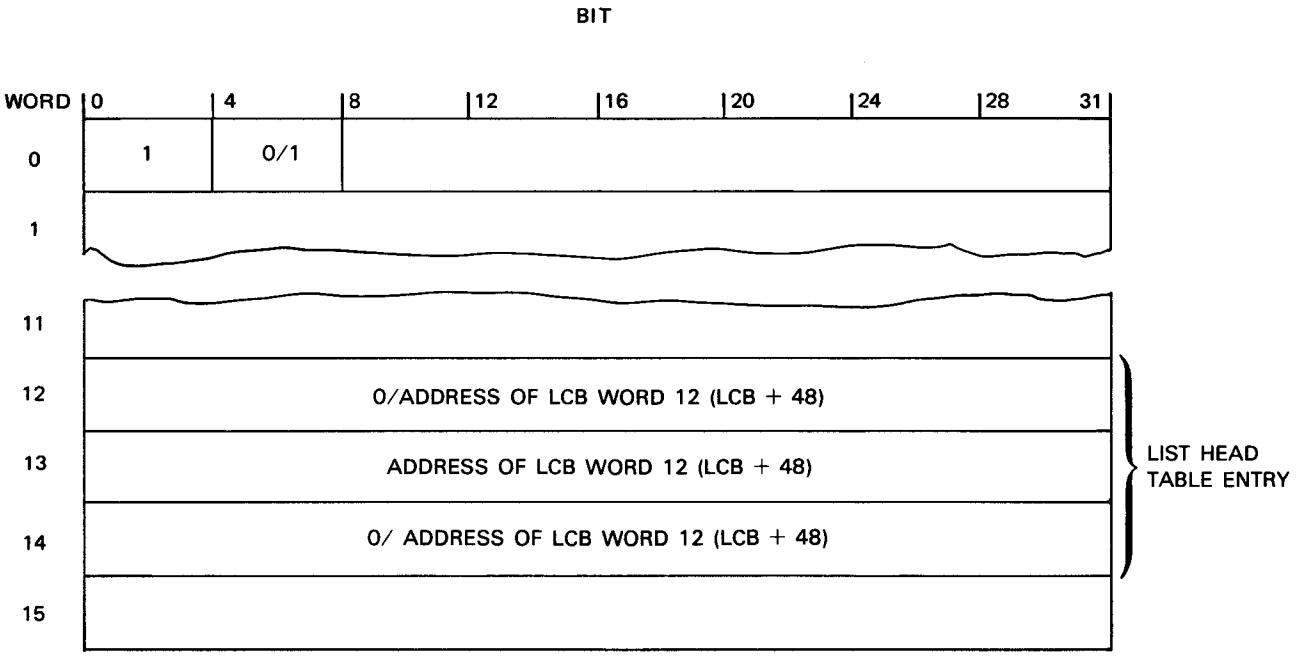


Figure 14—18. Initializing LCB for FIFO with Station

Word 0, bits 0—3 of the LCB must contain 1. Bits 4—7 can be either 0 (forward linkage) or 1 (double linkage). Word 13 is always set to the address of word 12. Depending on the element type, words 12 and 14 must both be set to zeros (forward linkage) or both to the word 12 address (double linkage). If a list head table entry controls the FIFO list it is three full words long and corresponds to words 12—14 of the LCB. If a word in the entry is not set to 0, you must set it to the address of the entry.

You add and remove elements in this type of list the same way you do for a FIFO list (14.5.2.2). Here, though, the station no longer occupies the even-numbered r_3 register, but now is included in the LCB as word 14. You can, therefore, use the STEP QUEUE instruction to move the station forward or backward in the list.

14.5.2.5. Ring with Station Usage

To initialize a ring list you initialize its LCB as shown in Figure 14—19.

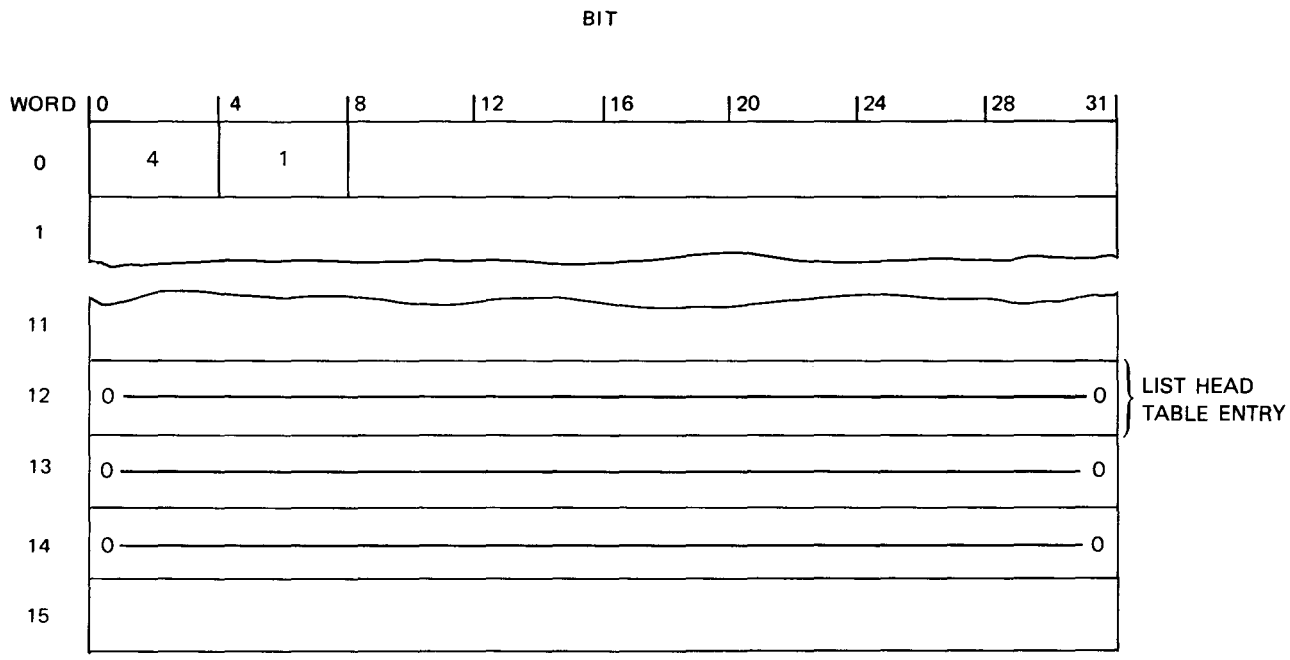


Figure 14—19. Ring List Initialization

Word 0, bits 0—3 must contain 4. Bits 4—7 must contain 1. Words 12—14 must all be initialized to zeros; if a list head table entry controls a ring list, that entry is one full word long and must also be initialized to zeros.

Word 12 of the LCB is the station a ring list uses. You add an element immediately after the one to which the station points. You usually remove the element to which the station points, but you can remove any element in the list by putting its address in the even-numbered r_3 register before executing DEQUEUE. After an ENQUEUE operation the station points to the newly-added element. If you dequeue the element the station points to, the station will be reset to the element immediately before (backward pointer) or after (forward pointer) it, depending on the setting of LCB word 0, bit 10 (0=backward, 1=forward). If you remove an element other than the one the station points to, the station remains at its present position. You can move the station forward or backward using the STEP QUEUE instruction.

14.5.2.6. Priority List Usage

To initialize a priority list, you initialize its LCB as shown in Figure 14—20.

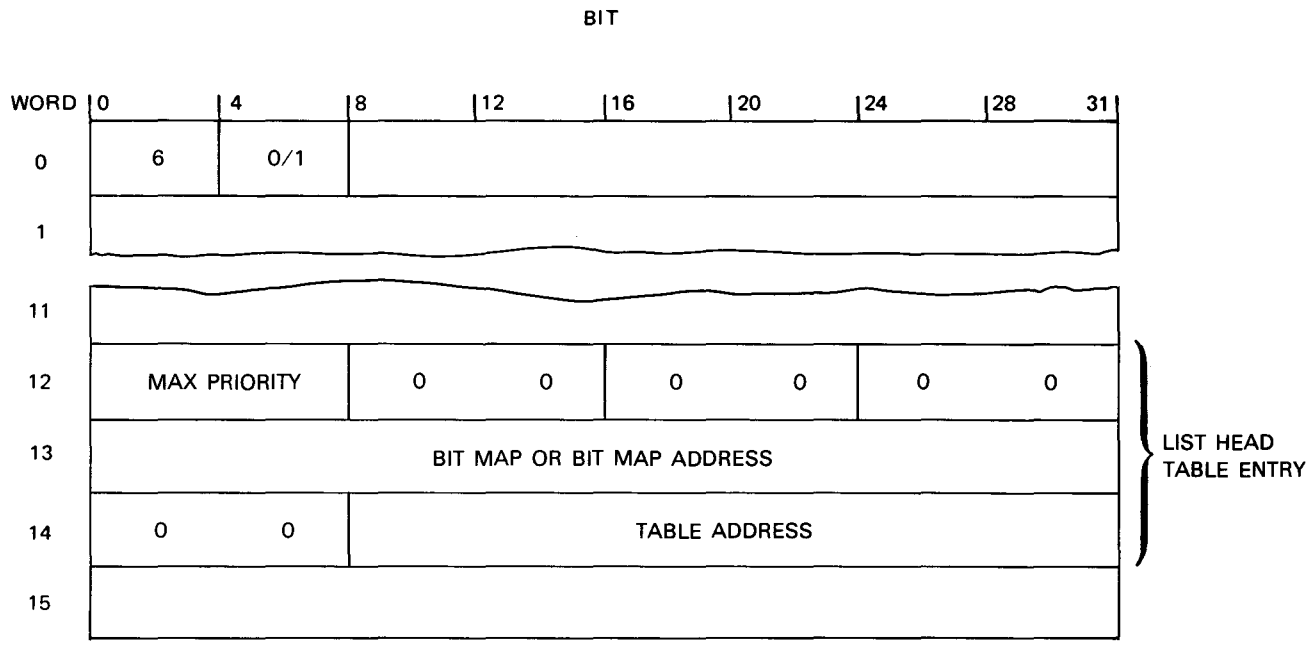


Figure 14—20. Priority List Initialization

Word 0, bits 0—3 must have a value of 6. Bits 4—7 can have a value of 0 (forward linkage) or 1 (double linkage). The rest of the LCB in Figure 14—20 is initialized as follows:

- Word 12, bits 0—7

  You put the highest-numbered priority your list can have in this field; a maximum value of 255_{10} is permitted.

- Word 12, bits 8—15

  This field is the N field, the number of priority levels currently supporting lists. Initialize this field to 0.

- Word 12, bits 16—23

  This field is level station 1, accessible from the STEP QUEUE instruction. Initialize this field to 0.

- Word 12, bits 24—31

  This field is level station 2, accessible from the STEP QUEUE instruction. Initialize this field to 0.

- Word 13, bits 0—31

  If you specify at most 32 priorities (word 12, bits 0—7 having a value between 0_{16} and $1F_{16}$) this full word comprises the bit map. The high order (leftmost) bit corresponds to priority 0, the next bit to priority 1, etc. If the number of priorities is in the range of 33—256 (word 12, bits 0—7 having a value between 20_{16} and FF_{16}) you must specify a bit map elsewhere in main storage, putting its 24-bit address in word 13. In this case the map must occupy an even number of full words and reside on a full-word boundary. In all cases the bit map must be initialized to all zeros.

- Word 14, bits 8—31

  You use this 24-bit field to contain the address of the first byte of the list head table this LCB controls. The list head table must be on a full-word boundary.

You add elements at a priority that you specify using the primary and (if used) secondary level stations contained in the odd-numbered r_3 register. Register bits 8—15 contain the primary station, bits 24—31 contain the secondary station, and the remainder of the register contains zeros. At the selected priority level, the element is enqueued at the front of the list (for LIFO lists), the back of the list (for FIFO lists), or the element station (for ring lists).

You can also remove elements from specific levels by specifying the levels in the odd-numbered r_3 register. At all priority levels, for all types of lists, it is the first element that is removed. You cannot specify that a particular element be removed, only the priority at which the element resides.

You can manipulate the two level stations of a priority list using the STEP QUEUE instruction. Unlike other types of lists, STEP QUEUE moves level stations so that they point to priority levels, never to individual elements.

14.5.2.7. Aged Priority List Usage

To initialize an aged priority list you initialize its LCB as shown in Figure 4—21.

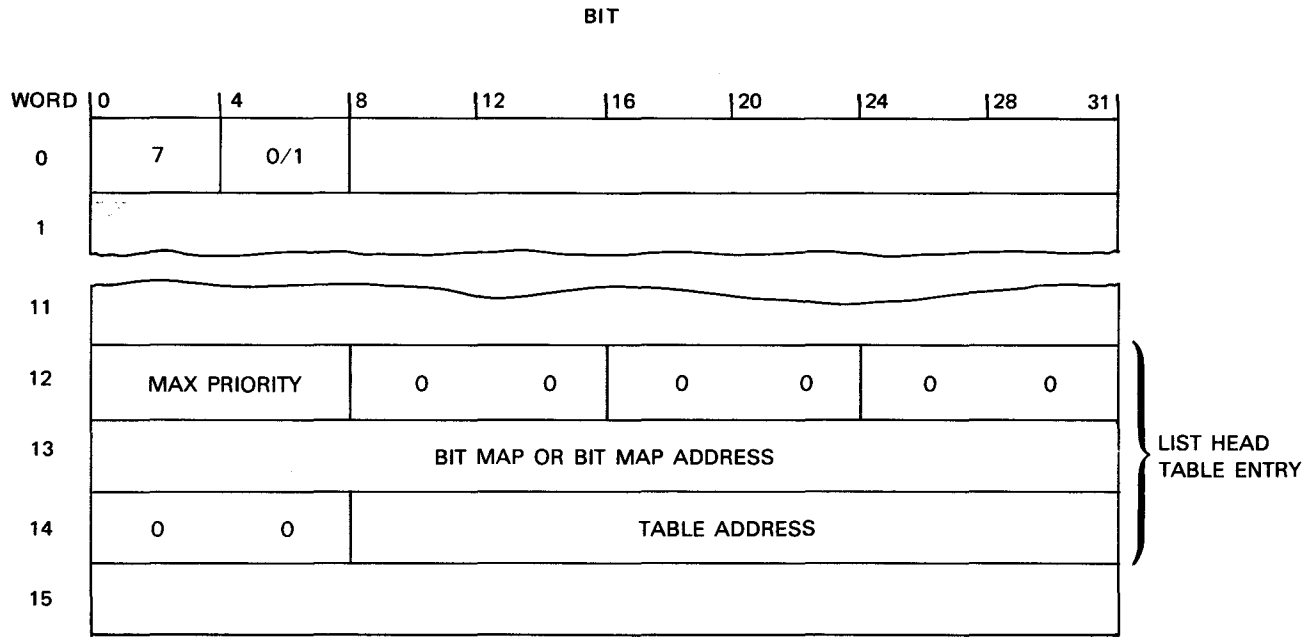


Figure 14—21. Aged Priority List Initialization

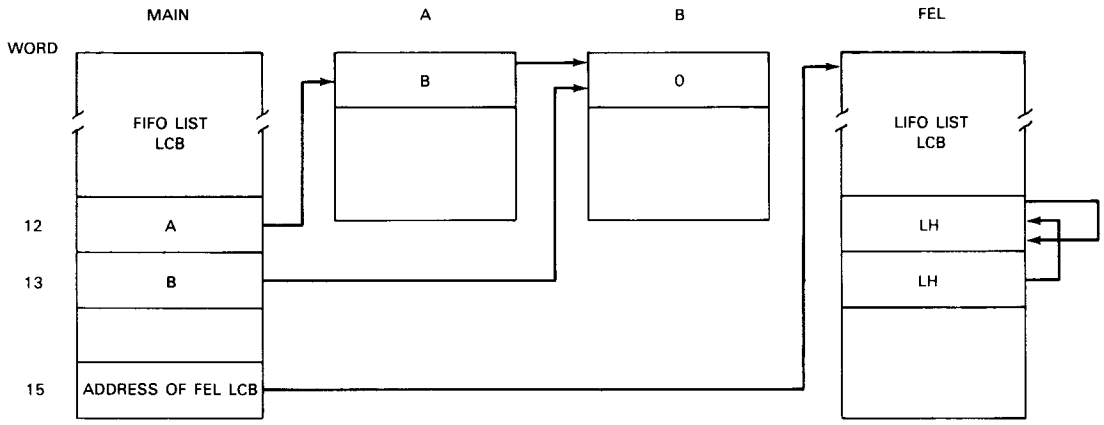
Word 0, bits 0—3 must have a value of 7. Bits 4—7 must have a value of binary 1 if the LCB controls a two-level list; otherwise, it can have a value of 0 (forward linkage) or 1 (double linkage). Words 12—14 must be initialized with the same values as a priority list (14.5.2.6). Likewise, an aged priority list can be controlled from another LCB, in which case its list head table has entries three words long; these must be initialized in the same manner as words 12—14 of the LCB in Figure 14—21. Wherever it resides, the bit map must be set to all zeros.

To add an element you specify its primary and (if used) secondary priority value in the odd-numbered r_3 register; these determine indirectly the priority at which the new element will actually be enqueued. When you remove an element from an aged priority list you do not specify a priority; the hardware algorithms included with DEQUEUE do that for you and return the address of the newly dequeued element in the even-numbered r_3 register. Likewise, you cannot use stations to point to individual priorities; you, therefore, cannot use the STEP QUEUE instruction on an aged priority list.

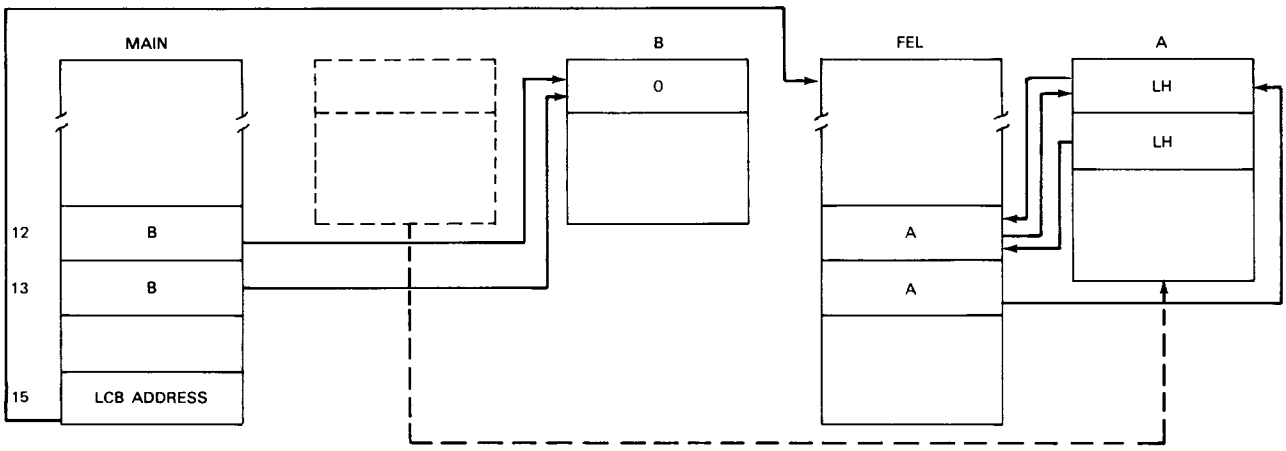
14.6. FREE ELEMENT LIST

The free element list (FEL) is a collection of elements in main storage not linked to any other list. Like other lists, it has a 16-word LCB. Unlike other lists, however, you do not directly manipulate it. Instead, it exists to provide a storage area from which other lists can add new elements and to which newly dequeued elements can go. One list can use one FEL, but a single FEL can, under certain conditions, be used by more than one list.

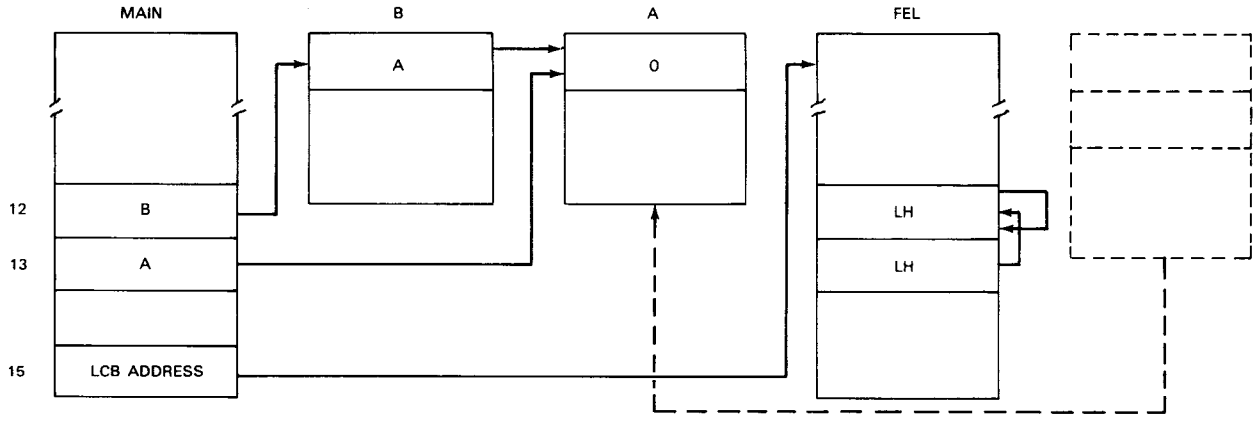
Using a FEL is simple. Figure 14—22 shows a list called MAIN and its associated FEL. MAIN is a forward-linked FIFO list, while the FEL is set up as a LIFO list.



a. List MAIN and its free element list



b. Dequeueing an element from MAIN



c. Enqueueing an element on MAIN

Figure 14—22. Enqueueing and Dequeueing with FEL

Figure 14—22a shows that MAIN has two elements, A and B, each containing a single forward pointer. In Figure 14—22b we see how an element is removed from MAIN. Here it is element A; the DEQUEUE instruction resets pointers so that element B alone occupies the MAIN list. But the section of main storage that was element A is not simply discarded. Instead, a separate ENQUEUE-type operation links A to the FEL as its first and only element. Notice that A now has two pointers, forward and backward. The FEL LCB determines where these lie within the element, and their location bears no relation to that of the single pointer earlier contained in A. The FEL simply treats element A as a section of main storage linked to other sections, all of which are available for use by list MAIN.

Figure 14—22c shows how the ENQUEUE instruction restores a FEL element to list MAIN. Again, element A is involved; a DEQUEUE-type operation removes it from the FEL and element A gets its old forward pointer back as it is linked into place behind element B (this being a FIFO list, remember). Throughout all this, the two LCBs are linked by a pointer in MAIN's LCB (word 15) that points to its related FEL LCB.

To use a FEL, you must know how to initialize it and how to call upon it during program execution.

14.6.1. FEL Initialization

You initialize the FEL much as you would any list, keeping in mind that the size of its elements must be the same between the FEL and its associated list or lists.

You can make the FEL any type of list you wish, but a LIFO or FIFO list is often best for your purposes; they permit you to specify both a maximum number of FEL elements and a limit to the main storage taken up by the FEL. In addition, you can place FEL pointers wherever you wish within its elements. The only restriction is that the FEL element size must equal that of the list or lists using the FEL.

Because FEL elements are merely blocks of main storage that other lists use, you will not likely use the data area, the register load/store option, or the data movement option. The only register you are likely to use is the even-numbered r_3 register which you initialize with the address of the first FEL element to be enqueued or dequeued. Before using the FEL, you must put the 24-bit address of its LCB in word 15, bits 8—31 of each LCB that is to use the FEL.

14.6.2. FEL Usage

You can use a FEL to add or remove an element from your list. Since the FEL determines what element is to be linked or unlinked from your list, you do not specify an element address in the even-numbered r_3 register of your list's LCB. Instead, you load that register with all zeros and set bit 9 of your ENQUEUE or DEQUEUE instruction to 1. The condition code is set as described for the instructions (14.3.1 and 14.3.2).

14.7. LIST PROCESSING OPTIONS

In addition to adding and removing elements, the System 80 list processing instructions give you the ability to manipulate registers and move main storage data whenever you add or remove an element.

14.7.1. Register Load/Store Option

By setting certain bits in your ENQUEUE or DEQUEUE instruction, you can move data between a selected set of registers and a save area in your list element. The options available to you are shown in Figure 14—23.

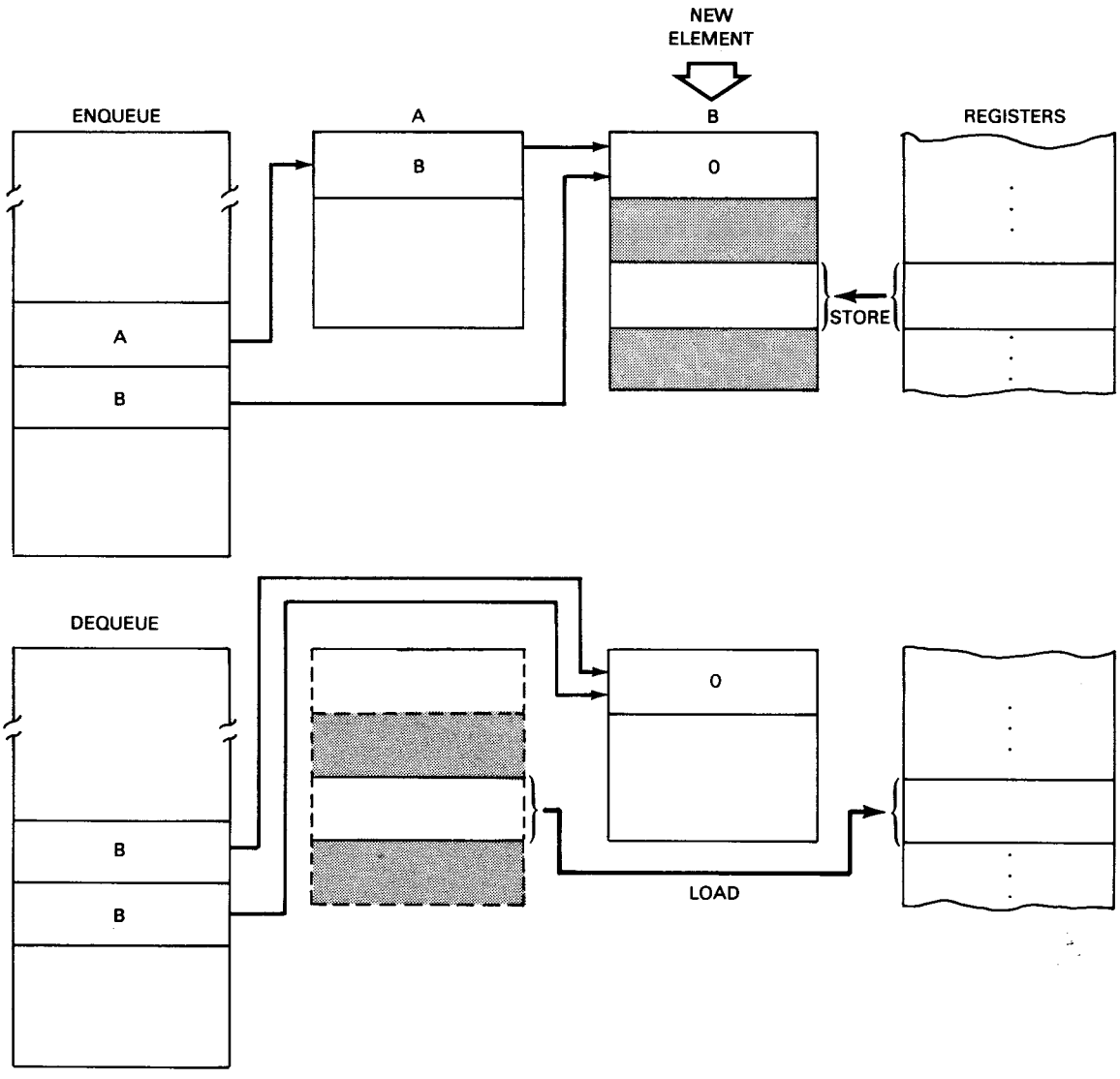


Figure 14—23. Register Load/Store Option

As Figure 14—23 shows, you can store the contents of certain registers into an element just before it is added to a list. Conversely, you can load certain registers from an element just after it is removed from the list.

The registers that can be used include:

- Some or all of the 16 problem general registers
- Some or all of the 16 supervisor general registers
- All 4 floating-point registers

To load or store general registers in an element, you must reserve a save area of 16 full words in the element, on a word boundary whose offset from the start of the element you specify in LCB word 16, bits 20—31. To load or store floating-point registers, you must reserve four double words in the element, on a double-word boundary whose offset from the start of the element you specify in LCB word 6, bits 4—15.

If you load or store the floating-point register set, all four registers (0, 2, 4, and 6) are involved. For the general registers, though, you use the r_1 and r_2 fields in the LCB to specify which registers you use. Registers are loaded or stored starting with the r_1 register, continuing through consecutively-numbered registers, and ending with the r_2 register. If r_1 is greater than r_2 the registers are processed in this order: r_1,...,15,0,...,r_2. If r_1 equals r_2 only that register is used. Register 0, if used, is always associated with the first word of the save area, register 1 with the second word, and so on.

You determine the actual load/store function performed by the instruction you use and the bits within the instruction you specify:

ENQUEUE Instruction:

- If you set both LCB word 0, bit 13 and ENQUEUE bit 13 to 1, the general register series defined by r_1, r_2, and bit 14 is stored in the general register save area of the element to be enqueued. If either or both bits are set to 0, the general registers are not stored.
- If you set either LCB word 0, bit 14, or ENQUEUE bit 14, or both bits to 0, the general registers to be stored are taken from the problem register set. If you set both bits to 1, the registers are taken from the supervisor register set.
- If you set both LCB word 0, bit 15 and ENQUEUE bit 15 to 1, you cause all four floating-point registers to be stored to the floating-point register save area of the element to be enqueued. If either or both bits are 0, this option will not be enabled.

DEQUEUE Instruction:

- If you set both LCB word 0, bit 13 and DEQUEUE bit 13 to 1, the general register series defined by r_1, r_2, and bit 14 is loaded from the general register save area of the element just dequeued. If either or both bits are 0, the registers are not loaded.

- If you set either LCB word 0, bit 14, or DEQUEUE bit 14, or both bits to 0, the general registers to be loaded are taken from the problem register set. If you set both bits to 1, the registers are taken from the supervisor register set.

- If you set both LCB word 0, bit 15 and DEQUEUE bit 15 to 1, you cause all four floating-point registers to be loaded from the floating-point register save area of the element just dequeued. If either or both bits are 0, this option will not be enabled.

14.7.2. Data Movement Option

When you enqueue an element, you have the option of moving a contiguous block of data from elsewhere in main storage into the data area of that element. This action takes place before the element is enqueued and after it is removed from its FEL, if one is used.

When you dequeue an element, you have the option of moving a contiguous block of data from the data area of that element to a location elsewhere in main storage. This takes place after the element is dequeued and before it is added to the FEL, if one is used.

We call the contents of the element data area *element data*. We call the data located elsewhere *external data.* Fields in the LCB govern data movement as shown in Figure 14—24.

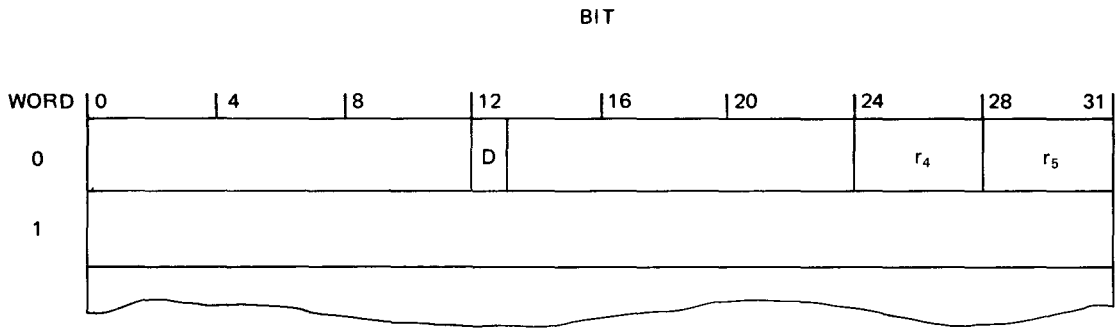


Figure 14—24. LCB Fields for Data Movement Option

Word 0, bit 12 contains the D field, which enables data movement. The r_4 and r_5 fields specify two even-odd register pairs that have the following format (Figure 14—25).

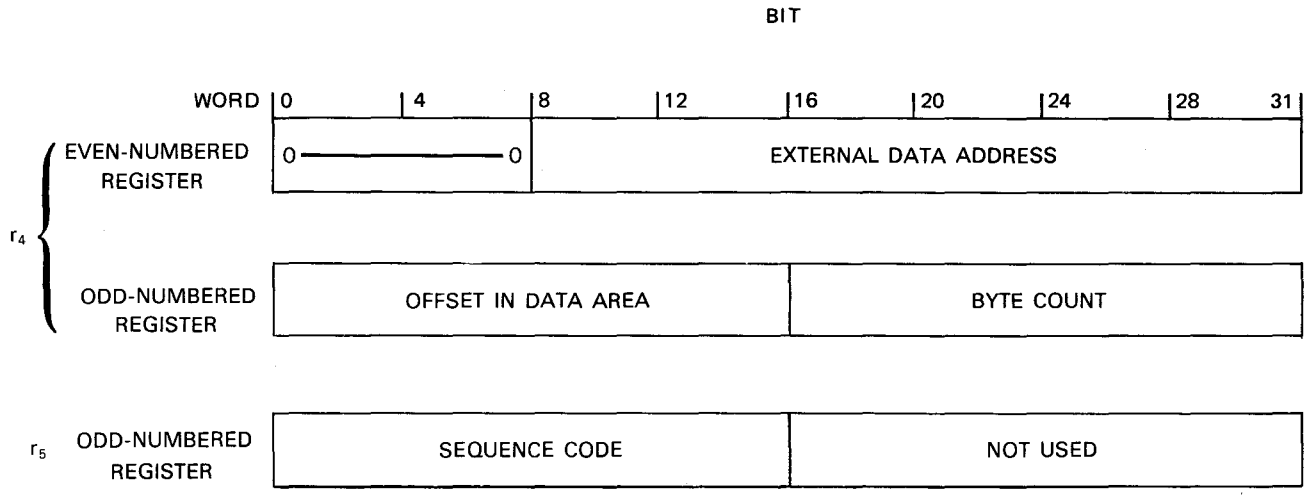


Figure 14—25. Registers for Data Movement

As Figure 14—25 shows, you put the address of the external data in bits 8—31 of the even-numbered r_4 register. You use bits 0—15 of the odd-numbered r_4 register to specify where in the element your element data lies; this value represents the offset of the first byte of your element data, in bytes, from the start of the data area. (Remember that the data area, in turn, is located as an offset within the element in word 7, bits 20—31 of the LCB). You put the number of bytes to be moved in bits 16—31 of the odd-numbered r_4 register. The data movement option uses the odd-numbered r_5 register which you must initialize to zeros.

Data movement occurs from left to right, starting with the first bytes at source and destination locations. Before movement begins, the hardware checks the element data and byte count fields to make sure you do not go beyond the limits of the element. If you do, a specification exception will occur.

To move data into an element to be enqueued, you must set both LCB word 0, bit 12 and ENQUEUE bit 12 to 1. Any other setting of these bits prevents data movement from occurring. The external data remains unchanged by the move.

To move data out of an element that is dequeued, you must set both LCB word 0, bit 12 and DEQUEUE bit 12 to 1. Any other setting of these bits prevents data movement from occurring. The element data remains unchanged by the move.

14.8. LIST CONTROL PROGRAM

In 14.4.3, we discussed how the STEP QUEUE instruction causes element or priority stations to move forward or backward in a list. Normally, STEP QUEUE moves a station from one element/priority level to the next and stops there. But a set of hardware operations is available to you which lets you use STEP QUEUE to perform logical operations, load or store registers, or move data in or out of an element. In addition, you can repeat these operations for successive elements in a list stopping only at the end of the list or when a condition you set is satisfied. These operations make up a machine instruction subset, and a sequence of them makes up a list control program (LCP) that you execute from a STEP QUEUE instruction. In the discussion that follows, we use the term *current element* to refer to the element currently pointed to by your selected station.

The list control program can cause an element or priority level station to be moved, usually after every instruction in the program is executed. Table 14—5 shows what LCB or LCB-related fields act as station for each list type.

Table 14—5. Stations Used by LCP Instructions

List Type	Station Used
FIFO	r_3 even-numbered register
LIFO	r_3 even-numbered register
FIFO with station	Station
Double-ended	r_3 even-numbered register
Ring	Station
Priority	Level station 1/Level station 2
Aged priority	No station for step queue

Note that a priority list can use one of two level stations (LCP word, 12 bits 16—31). As you will see, you select the station to be moved using the STEP QUEUE instruction that executes the list control program. Note, too, that an aged priority list does not support stations at all, so no list control program can manipulate it.

You always execute a list control program from a STEP QUEUE instruction. After the list control program finishes, program control passes to the next instruction immediately following STEP QUEUE. We first show how to put an LCP together, then how to call it with STEP QUEUE.

14.8.1. LCP Format

A list control program consists of a series of LCP instructions in main storage, each of which is four full words long. The list control program must be on a full-word boundary. Figure 14—26 shows the basic format of all LCP instructions.

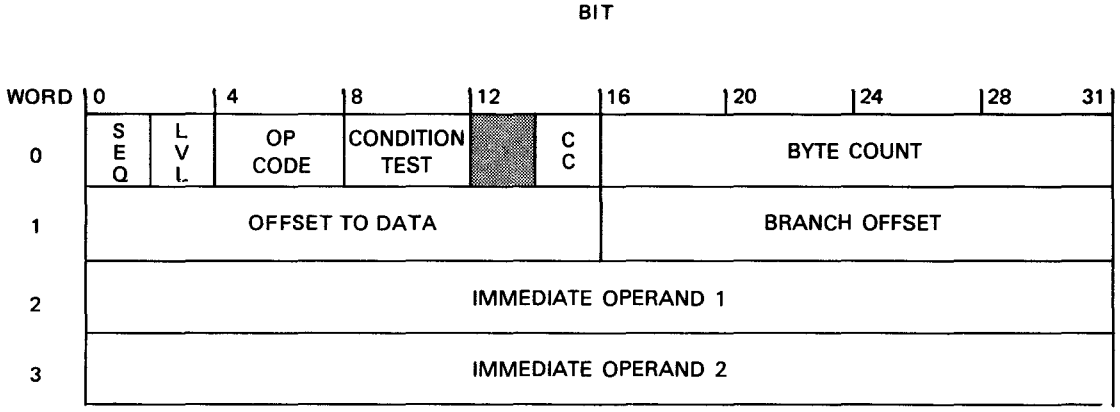


Figure 14—26. LCP Instruction Format

The following list explains the fields in Figure 14—26 by function:

- Op Code (word 0, bits 4—7)

  This 4-bit field determines which LCP instruction this is. Permitted entries, all of which are discussed in detail in 14.8.2, are:

Op Code	Instruction
00	NO-OP
01	MASKED TEST
02	LOGICAL COMPARE
03	MASK AND COMPARE
04	LOAD REGISTERS
05	STORE REGISTERS
06	MOVE DATA OUT
07	MOVE DATA IN
08	STEP STATION
09	INITIALIZE STATION
0A	SWITCH LIST SCAN

- Condition Code (word 0, bits 14—15)
  Condition Test (word 0, bits 8—11)
  Sequence Control (word 0, bits 0—1)

  When an LCP instruction completes its function, it sets its condition code (CC) field according to the result. Together with the condition test (CT) and sequence control (SEQ) fields (which you set), it determines where program control passes when the instruction finishes.

  This process takes place in two steps. First, the hardware compares the CC field to your CT field, to yield one of two results: either a CT match or a CT mismatch. Table 14—6 shows what field settings produce what results ("X" indicates a bit position whose value does not affect the result).

Table 14—6. CT Match/Mismatch Table

If the CC field is set to ...	...and you set the CT field to ...	...the result is:
00	1XXX	CT Match
00	0XXX	CT Mismatch
01	X1XX	CT Match
01	X0XX	CT Mismatch
10	XX1X	CT Match
10	XX0X	CT Mismatch
11	XXX1	CT Match
11	XXX0	CT Mismatch

  As the next step, the hardware compares the CT match/mismatch to the sequence code (SEQ) that you set. The resulting action is shown in Table 14—7.

  Do not confuse the CC field of an LCP instruction with the condition code contained in the PSW. As Table 14—7 shows, the CC field will replace the PSW condition code under certain conditions. However, there is only one PSW condition code, while a CC field exists in each instruction of your LCP.

- Branch Offset (word 1, bits 16—31)

  You use this field to direct LCP program control to the next logical instruction. This field represents the offset of the destination instruction from the start of the LCP. Together with the CC, CT, and SEQ fields, this field may be used by every LCP instruction in the set. If your program logic includes a branch from the current LCP instruction to any LCP instruction other than the next sequential one, you must use this field. Because LCP instructions are 16 bytes long, values placed in this field must be evenly divisible by 16. You can branch in a forward direction (increasing address) only. The maximum permitted offset is 4080_{10} ($FF0_{16}$) bytes.

Table 14—7. Program Control Under LCP Fields

If the SEQ value is:	...you get this result with a:	
	CT Match	**CT Mismatch**
00	The hardware replaces the condition code field in the program status word (PSW) with the CC value of the instruction. Execution of the STEP QUEUE instruction that called this list control program immediately ends.	Same result as a CT match with SEQ value 00.
01	The hardware moves the selected station in the direction you specify in STEP QUEUE bit 9. If the step is unsuccessful, control passes to the next logical LCP instruction (see BRANCH OFFSET). If the step is successful, control passes back to the first instruction in your list control program.	Program control passes to the next sequential LCP instruction without station movement.
10	Program control passes to the next logical LCP instruction (see BRANCH OFFSET) without station movement.	The hardware moves the selected station in the direction you specify by STEP QUEUE bit 9. If the step is unsuccessful, control passes to the next sequential LCP instruction. If the step is successful, control passes back to the first LCP instruction in the list control program.
11	Program control passes to the next logical LCP instruction (see BRANCH OFFSET).	Program control passes to the next sequential LCP instruction.

- Offset to Data (word 1, bits 0—15)

  Certain LCP instructions require the list control program to access data within the current element. You specify where the data is by using this field as an offset from the first byte of the element data field (LCB word 7, bits 20—31) to the first byte of the data to be accessed.

- Immediate Operands (words 2 and 3)

  Certain LCP instructions use immediate data contained within these two words. More detailed information can be found under each individual instruction.

- Byte Count (word 0, bits 16—31)

  You use this field with some LCP instructions to specify how many bytes of data are to be processed.

- Level Station (word 0, bits 2—3)

  You use this field (LVL) with some LCP instructions to specify what element or priority station they are to use in stepping operations.

14.8.2. LCP Instructions

Here we discuss the LCP instructions individually. For each instruction, we show the LCP format, describe its function, explain what condition code it sets with what results, and include other information you may need to use it.

14.8.2.1. NO-OP LCP Instruction

The NO-OP LCP instruction (Figure 14—27) performs no function by itself, serving instead as a termination instruction, a branching instruction, or a common destination point for branches from elsewhere in a list control program.

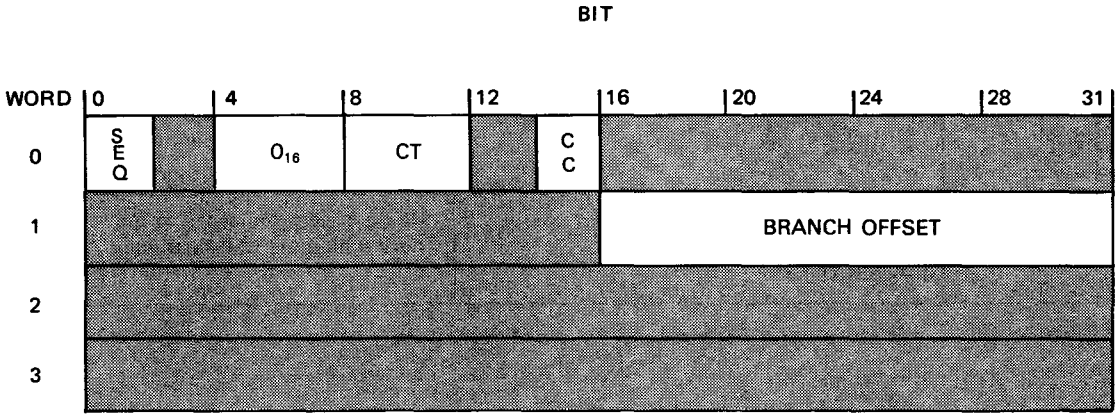


Figure 14—27. NO-OP Format

The NO-OP instruction sets its CC field to 00_2 if the operation is successful; no other CC values are used. Program control then passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields, as shown in Tables 14—6 and 14—7.

14.8.2.2. MASKED TEST LCP Instruction

The MASKED TEST LCP instruction tests one to four contiguous bytes of data in the current element against an immediate mask contained in the LCP instruction (see Figure 14—28). The instruction sets its CC field according to the result.

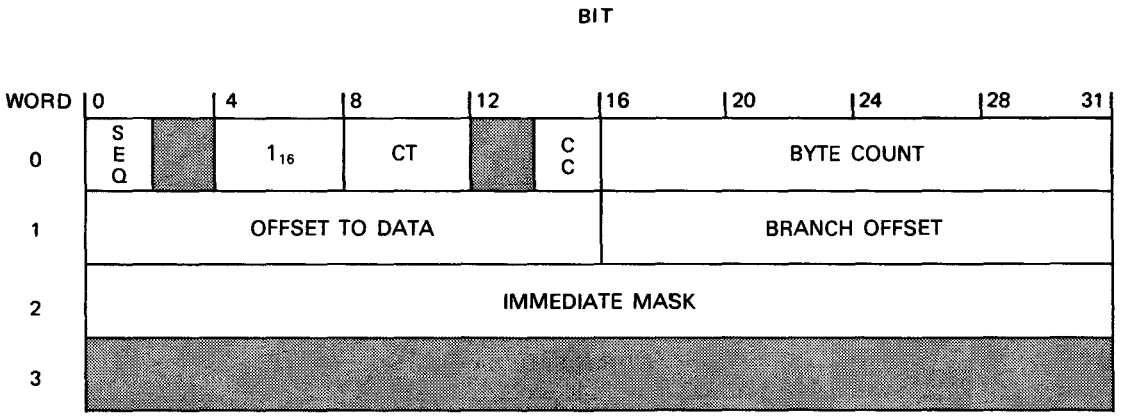


Figure 14—28. MASKED TEST Format

This instruction works much like the TEST UNDER MASK instruction. You specify the number of bytes to be tested in the BYTE COUNT field. Permitted values for this field are binary 1—4. You specify the element location of the first element byte to be tested in the OFFSET TO DATA field; this value represents the offset from the start of the element data area to the first tested byte.

The IMMEDIATE MASK field holds up to four bytes of data. Bits within the mask that are set to 1 cause their corresponding bits in the element data field to be tested. Mask bits set to 0 cause the instruction to ignore their corresponding element bits. The instruction begins with the high order (left-most) mask byte and proceeds left to right for as many bytes as you specify. After the test is finished, the instruction sets its CC field. Program control then passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields, as shown in Tables 14—6 and 14—7. The CC field is set:

- to 00_2 if all tested bits are 0 or if all bits in the mask (up to the number specified in BYTE COUNT) are 0;
- to 01_2 if some tested bits are 0 and some 1; or
- to 11_2 if all tested bits are 1.

14.8.2.3. LOGICAL COMPARE LCP Instruction

The LOGICAL COMPARE LCP instruction (Figure 14—29) logically compares one to four contiguous bytes of data in the current element against an immediate operand contained in the instruction. The instruction sets its CC field according to the result.

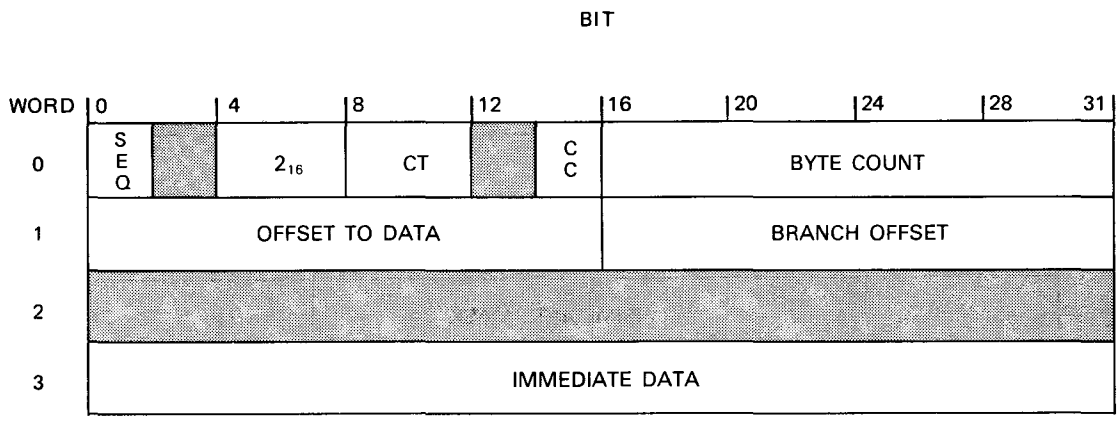


Figure 14—29. LOGICAL COMPARE Format

This instruction works much like the COMPARE LOGICAL instruction. You specify the number of bytes to be compared, from 1 to 4 bytes, in the BYTE COUNT field. You specify the location of the element operand by setting the OFFSET TO DATA field to the offset from the start of the element to the first byte of the operand.

The IMMEDIATE DATA field holds up to four bytes of operand data. Comparison starts between the first element byte and the leftmost byte of the IMMEDIATE DATA field, and proceeds left to right for as many bytes as the BYTE COUNT field specifies. For comparison purposes, all data is treated as binary and unsigned.

After the comparison operation is finished, the instruction sets its CC field. Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7. The CC field is set:

- to 00_2 if the immediate data and the element data are equal;
- to 01_2 if the immediate data is less than the element data; or
- to 10_2 if the immediate data is greater than the element data.

14.8.2.4. MASK AND COMPARE LCP Instruction

The MASK AND COMPARE LCP instruction (Figure 14—30) logically compares one to four bytes of data in the current element against an immediate operand in the instruction. The comparison is governed by an immediate mask also contained in the instruction. The instruction sets its CC field according to the result.

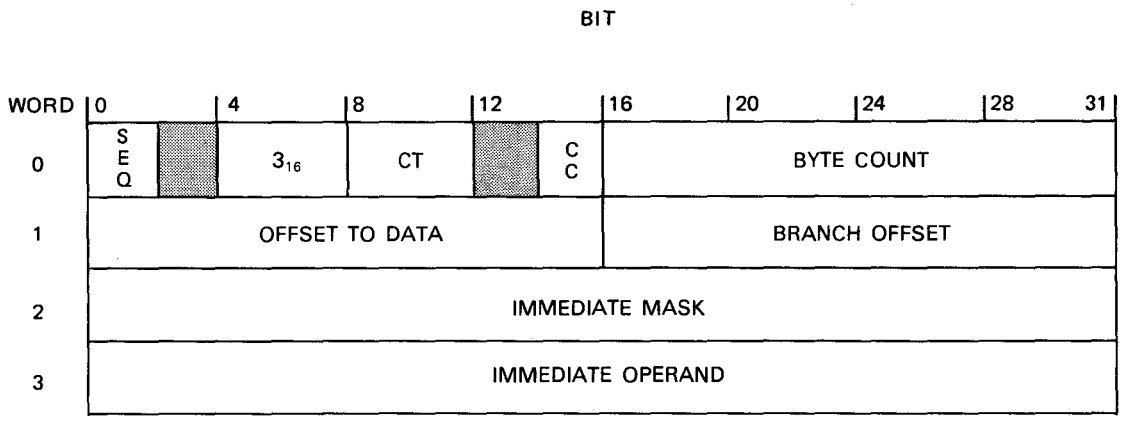


Figure 14—30. MASK AND COMPARE Format

This instruction works much like a COMPARE LOGICAL CHARACTERS UNDER MASK instruction. You specify the number of bytes to be compared, from 1 to 4 bytes, in the BYTE COUNT field. You specify the location of the element operand by setting the OFFSET TO DATA field to the offset from the start of the element data area to the first byte of the operand. You specify the operand bits that are to be compared using the IMMEDIATE MASK field. A 1 bit in the mask causes the instruction to compare the corresponding bits in the element and immediate operands. A zero bit causes the instruction to ignore the corresponding operand bits.

The IMMEDIATE OPERAND and IMMEDIATE MASK fields can hold up to four bytes each. Masking and comparison begin with the leftmost byte in each field and proceed left to right for as many bytes as the BYTE COUNT field specifies. For comparison purposes, all data is treated as binary and unsigned.

After the comparison operation is finished, the instruction sets its CC field. Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7. The CC field is set:

- to 00_2 if the masked element data is equal to the masked immediate data;
- to 01_2 if the masked immediate data is less than the masked element data; or
- to 10_2 if the masked immediate data is greater than the masked element data.

14.8.2.5. LOAD REGISTERS LCP Instruction

The LOAD REGISTERS LCP instruction (Figure 14—31) loads any or all of a selected set of registers from the register save areas of the current element.

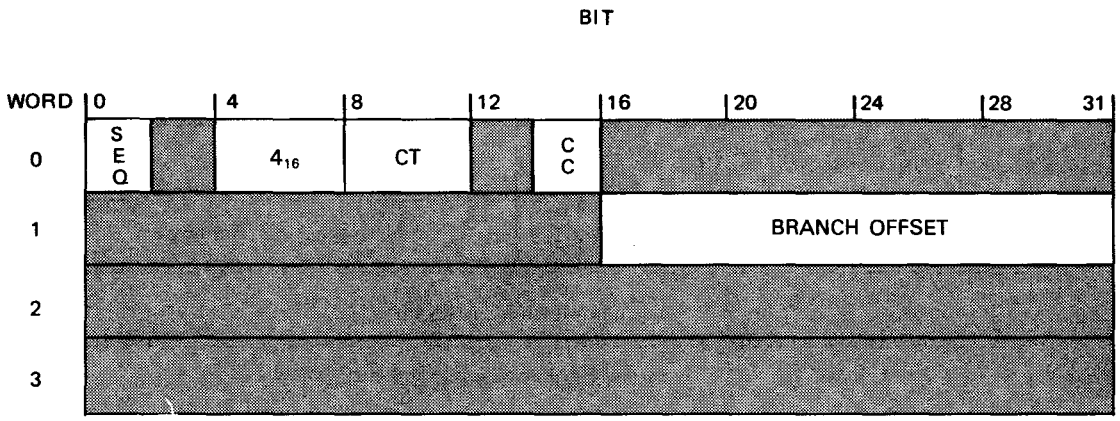


Figure 14—31. LOAD REGISTERS Format

The LOAD REGISTERS instruction acts much like the register load option of the DEQUEUE instruction. In fact, it is controlled by the same portion of the STEP QUEUE instruction (bits 13—15) that the DEQUEUE option is. It also uses the same fields of the LCB and its elements. The only difference is that with LOAD REGISTERS you can load from any element in your list, not just the one you have just removed. The following list explains how to enable and control the LOAD REGISTER instruction once you call it in your list control program:

- STEP QUEUE bit 13 — LCB word 0, bit 13

  To enable the loading of general registers from the general register save area of the current element (addressed by LCB word 6, bits 20—31) you must set both these bits to 1.

- STEP QUEUE bit 14 — LCB word 0, bit 14

  To load from the supervisor register set, specify a 1 in both bits. Otherwise, the problem register set will be used.

- STEP QUEUE bit 15 — LCB word 0, bit 15

  To enable the loading of all four floating-point registers from their save area (addressed by LCB word 6, bits 4—15) you must set both these bits to 1.

- r_1/r_2 (LCB word 3, bits 0—7)

  These two fields specify the range of general registers, supervisor or problem, to be loaded. Loading begins with the r_1 register, continues through consecutively-numbered registers, and ends with r_2. If r_1 equals r_2 only that register is loaded. If r_1 is greater than r_2 loading proceeds in this order: r_1,...,15,0...,r_2. If specified, register 0 is always loaded from the first save area word, register 1 from the second word, etc.

After the load operation is finished, the instruction sets its CC field to 00_2. Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7.

14.8.2.6. STORE REGISTERS LCP Instruction

The STORE REGISTERS LCP instruction (Figure 14—32) stores any or all of a selected set of registers to the register save area of the current element.

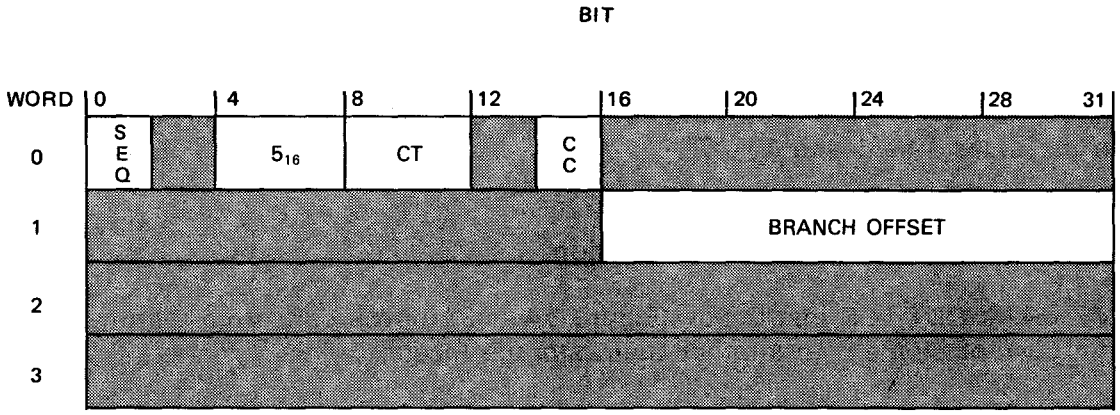


Figure 14—32. STORE REGISTERS Format

The STORE REGISTERS instruction acts much like the register store option of the ENQUEUE instruction. In fact, it is controlled by the same portion of the STEP QUEUE instruction (bits 13—15) that the ENQUEUE instruction is. It also uses the same fields of the LCB and its elements. The only difference is that with STORE REGISTERS you can store to any element in your list, not only the one just added. The following list explains how to enable and control the STORE REGISTERS instruction once you call it in your list control program.

- STEP QUEUE bit 13 — LCB word 0, bit 13

  To enable the storage of general registers to the general register save area (addressed by LCB word 6, bits 20—31) you must set both these bits to 1.

- STEP QUEUE bit 14 — LCB word 0, bit 14

  To store data from the supervisor register set, specify a 1 in both bits. Otherwise, the problem register set is used.

- STEP QUEUE bit 15 — LCB word 0, bit 15

  To enable the storage of all four floating-point registers to their save area (addressed by LCB word 6, bits 4—15) you must set both these bits to 1.

- r_1/r_2 (LCB word 0, bits 0—7)

  These two fields specify the range of general registers to be stored. Storage begins with the r_1 register, continues through consecutively-numbered registers, and ends with the r_2 register. If r_1 equals r_2 only that register is stored. If r_1 is greater than r_2 storage proceeds in this order: $r_1,...,15,0,...,r_2$. If specified, register 0 is stored at the first word of the save area, register 1 at the second word, etc.

After the storage operation is finished, the instruction sets its CC field to 00_2. Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7.

14.8.2.7. MOVE DATA OUT LCP Instruction

The MOVE DATA OUT LCP instruction (Figure 14—33) moves data from within the data area of the current element to another main storage location.

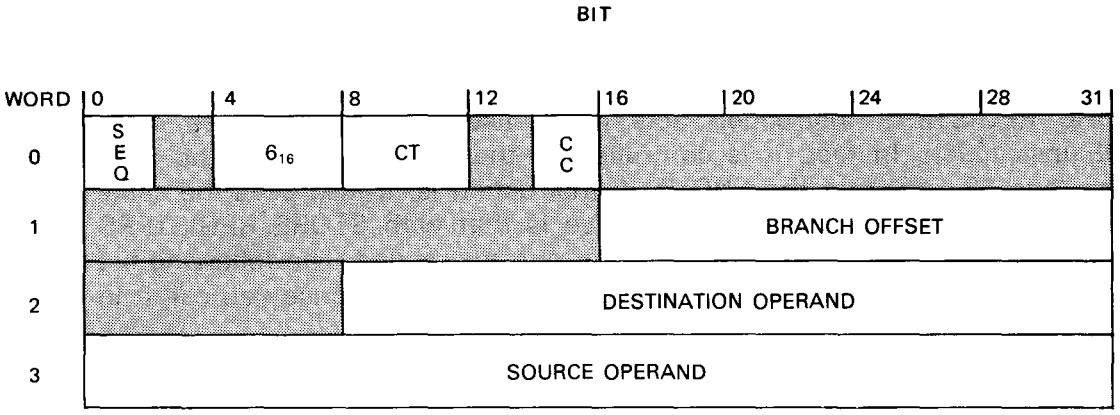


Figure 14—33. MOVE DATA OUT Format

This instruction acts much like the data movement option available with the DEQUEUE instruction. As with DEQUEUE, you specify the location of the source operand as the offset of its first byte from the first byte in the current element's data area; this offset goes into bits 0—15 of the odd-numbered r_4 register. The destination address goes into bits 8—31 of the even-numbered r_4 register. Bits 16—31 of the odd-numbered r_4 register hold the number of bytes to be moved up to a maximum of 4095 bytes.

You have the choice of controlling data movement using the current contents of the r_4 register pair or using data you load into the registers before data movement begins, data you specify in words 2 and 3 of the LCP instruction. To use the registers unchanged, put all zeros into word 2 of the instruction. To alter the registers, put the data you want to go in the even-numbered register in word 2 and the odd-numbered register data in word 3. Making word 2 a nonzero value triggers a register load routine that loads words 2 and 3 into the r_4 register pair, then uses this new data to control data movement.

After the data movement operation is finished, the instruction sets its CC field to 00_2. Program control then passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7.

14.8.2.8. MOVE DATA IN LCP Instruction

The MOVE DATA IN format is shown in Figure 14—34.

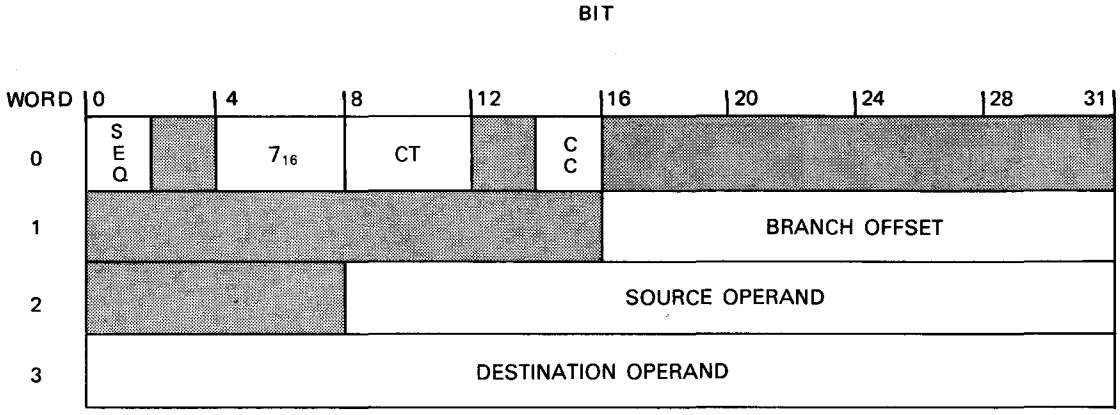


Figure 14—34. MOVE DATA IN Format

This instruction acts much like the data movement option available with the ENQUEUE instruction. As with ENQUEUE, you specify the location of the destination operand as the offset of its first byte from the first byte in the current element's data area; this offset goes into bits 0—15 of the odd-numbered r_4 register. The source address goes into bits 8—31 of the even-numbered r_4 register. Bits 16—31 of the odd-numbered r_4 register hold the number of bytes to be moved, up to a maximum of 4095 bytes.

You have the choice of controlling data movement using the current contents of the r_4 register pair or using data you load from words 2 and 3 of the instruction, just as in the MOVE DATA OUT instruction. To use the r_4 data unchanged, put all zeros into word 2. Otherwise, a nonzero word 2 value causes words 2 and 3 to be loaded to the r_4 register pair before data movement begins.

After the data movement operation is finished, the instruction sets its CC field to 00_2. Program control then passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7.

14.8.2.9. STEP STATION LCP Instruction

The STEP STATION LCP instruction (Figure 14—35) moves a station you select one position in the direction you specify. You can only use this instruction with priority lists; therefore, it has the effect of moving the selected station from one list head table entry to the next priority level above or below it.

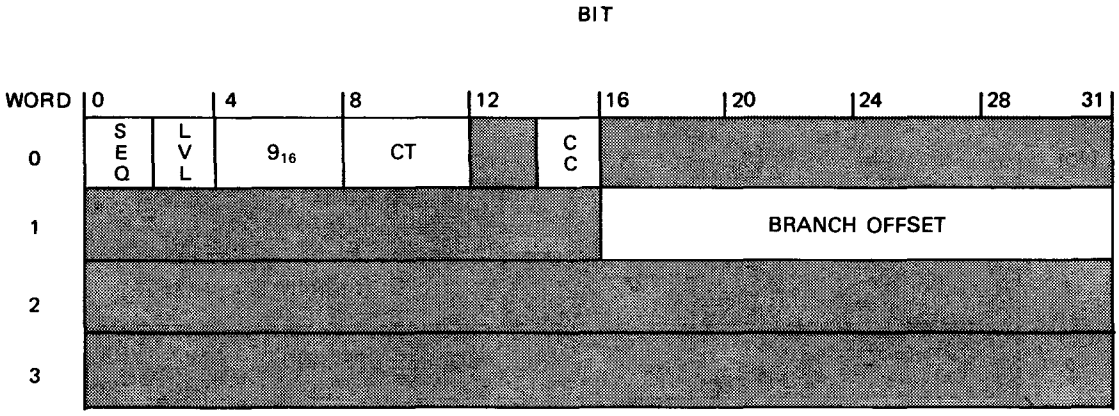


Figure 14—35. STEP STATION Format

You select the priority station you want moved by a combination of parameters in and out of the LCP instruction. You use the LVL field to specify whether you are moving a primary level or secondary level station for a two-level list. Permitted values are:

- 00_2 for a primary level station; or
- 01_2 for a secondary level station.

For a one-level station you always specify a LVL value of 00_2.

Bit 9 of the STEP QUEUE instruction that calls the list control program determines the direction of station movement. Permitted values are:

- 0_2 for forward movement (decreasing priority); or
- 1_2 for backward movement (increasing priority).

As you have seen, each priority list head has two stations you can move.

You use the STN field, bits 10 and 11 of the STEP QUEUE instruction, to specify the station you want. Permitted values are:

- $0X_2$ for primary level station 1;
- $1X_2$ for primary level station 2;

- $X0_2$ for secondary level station 1; or
- $X1_2$ for secondary level station 2.

The X character indicates a bit position whose value does not affect station selection. If you specify a LVL value of 00_2, for example, you need specify only STN values $0X_2$ or $1X_2$ as bit 11 (secondary level) is now irrelevant.

After execution of the step function the instruction sets its CC field according to the result. Possible settings are:

- 00_2 for a successful step operation; or
- 01_2 for an unsuccessful step operation, usually caused by attempting to step a station beyond one end of a list.

Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7.

14.8.2.10. INIT STATION LCP Instruction

The INIT STATION LCP instruction (Figure 14—36) causes the station you select to point to the first or last element in a list.

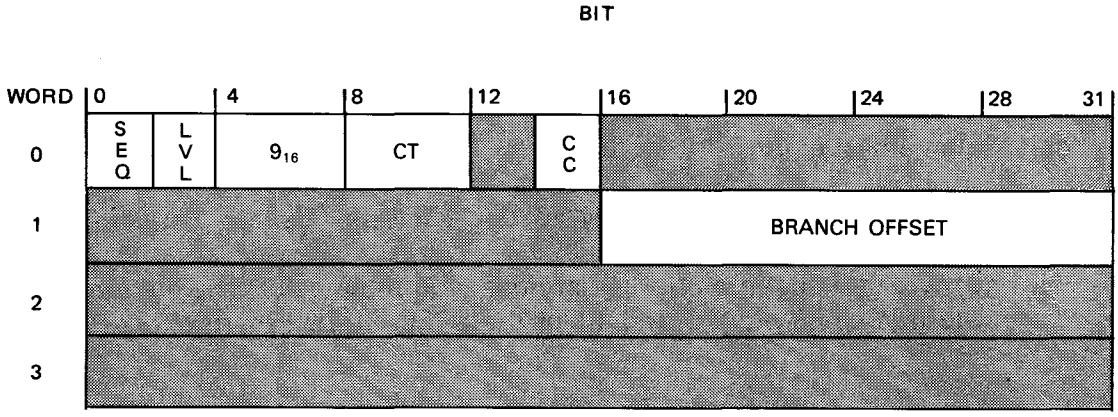


Figure 14—36. INIT STATION Format

This instruction uses some of the same fields as the STEP STATION instruction. Unlike it, however, you can use INIT STATION with every type of list except aged priority. Table 14—8 shows, for every permitted type of list, what element or priority level a station points to after this instruction is executed.

Table 14—8. INIT STATION Effects on Stations

List Type	Step Direction	
	Forward	Backward
LIFO*	First element	Last element
FIFO*	First element	Last element
Double-ended*	First element	Last element
FIFO with station	First element	Last element
Ring with station	Current element	Current element
Priority	Priority 0	Lowest priority (equal to MAX)
Aged priority	Not used	

*Even-numbered r_3 register used as station

Note that specifying a forward direction means that the station is set to point to the first element you would encounter while scanning from a list head in the forward direction. Similarly, a backward direction means the station is set to point to the first element you would encounter scanning backward pointers from the list head — even though that same element is the last one you would find in a forward scan.

For our purposes we distinguish a priority station, an 8-bit field used only with priority and aged priority lists, from an element station, a 24-bit field containing the address of an element. You select the type of station you are initializing using the LVL field. Permitted values are:

- 00_2 for a primary level priority station;
- 01_2 for a secondary level priority station; or
- 10_2 for an element station.

For a one-level priority list you must specify an LVL value of 00_2.

Because you have two stations in a priority list head, you must select the station you want to initialize. You do this using bits 10 and 11 of the STEP QUEUE instruction. Permissible values are:

- $0X_2$ for primary level station 1;
- $1X_2$ for primary level station 2;
- $X0_2$ for secondary level station 1; or
- $X1_2$ for secondary level station 2.

You choose the direction of initialization (the step direction of Table 14—8) using bit 9 of STEP QUEUE. Permitted values are:

- 0_2 for forward initialization; or
- 1_2 for backward initialization.

After the initialization operation is finished, the instruction sets its CC field:

- to 00_2 if the operation is successful; or
- to 01_2 if the operation is unsuccessful, usually caused when the instruction encounters an empty nonpriority list.

Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields as shown in Tables 14—6 and 14—7.

14.8.2.11. SWITCH LIST SCAN (SWLS) LCP Instruction

The SWITCH LIST SCAN (SWLS) instruction (Figure 14—37) is a modification of the MASK AND COMPARE LCP instruction (14.8.2.4) that is used with the supervisor switch list.

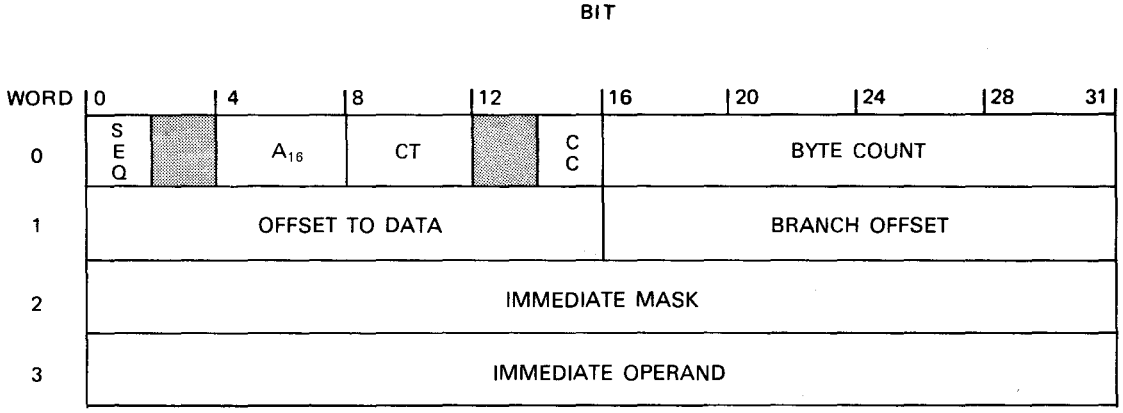


Figure 14—37. SWLS Format

The SWLS instruction acts much like the MASK AND COMPARE instruction in that it logically compares 1 to 4 bytes of data contained in the IMMEDIATE DATA field against an equal number of bytes in the current element data area, at a location specified by the OFFSET TO DATA field. Like MASK AND COMPARE, too, this instruction uses a mask in the IMMEDIATE MASK field in such a way that 1 bits in the mask cause a comparison between their corresponding bits in the immediate operand and element fields, while 0 bits in the mask cause their corresponding bits to be ignored. And like MASK AND COMPARE, the BYTE COUNT field determines the number of bytes to be compared; if fewer than four bytes are specified, the instruction begins its operation with the leftmost byte and proceeds left to right.

The differences between the MASK AND COMPARE and SWLS instructions lie in the logic each follows after the comparison operation is finished. First, the SWLS instruction sets its CC field:

- to 00_2 if the masked element operand equals the masked immediate operand, the mask equals 0, or the element operand equals 0, regardless of mask or comparison values;
- to 01_2 if the masked immediate data is less than the masked element data; or
- to 10_2 if the masked immediate data is greater than the masked element data.

Program control passes to the location determined by the CC, CT, SEQ, and BRANCH OFFSET fields. This happens as described in Table 14—6 and 14—7 — but with one exception: if a condition occurs that would cause a step to the next (or previous) element, the instruction does not pass program control back to the first instruction in the list control program. Rather, the instruction repeats itself, the station pointing now to the next element in the list. All other conditions are handled as shown in Tables 14—6 and 14—7.

14.8.3. Initializing and Calling List Control Programs

Much of the work involved in designing list control programs lies in the logic they must follow. Keep in mind that an LCP instruction is always four words long and that you can branch any number of instructions forward using the BRANCH OFFSET field; but you can branch backward to one location only, the first instruction in the list control program. Remember, too, that once you execute a list control program from a STEP QUEUE instruction, all STEP QUEUE control bits (bits 8—15) remain in effect and unchanged throughout execution.

Once you have designed your list control program, you must link it with the STEP QUEUE instruction that is to execute it. You do this using your list's LCB. Table 14—9 explains how to do this and initialize other important fields also.

Table 14—9. Initializing LCB Registers for LCP Execution

LCB Field	Initialize With
Even-numbered r_5 register	The address of the first LCP instruction, called the base address
Odd-numbered r_5 register	All zeros; this is the current LCP offset, which LCP instructions use as a location counter.
Even-numbered r_3 register	Current element address, which must be used with LIFO or FIFO lists not using a separate station. This is usually set for you by previous list processing instructions.

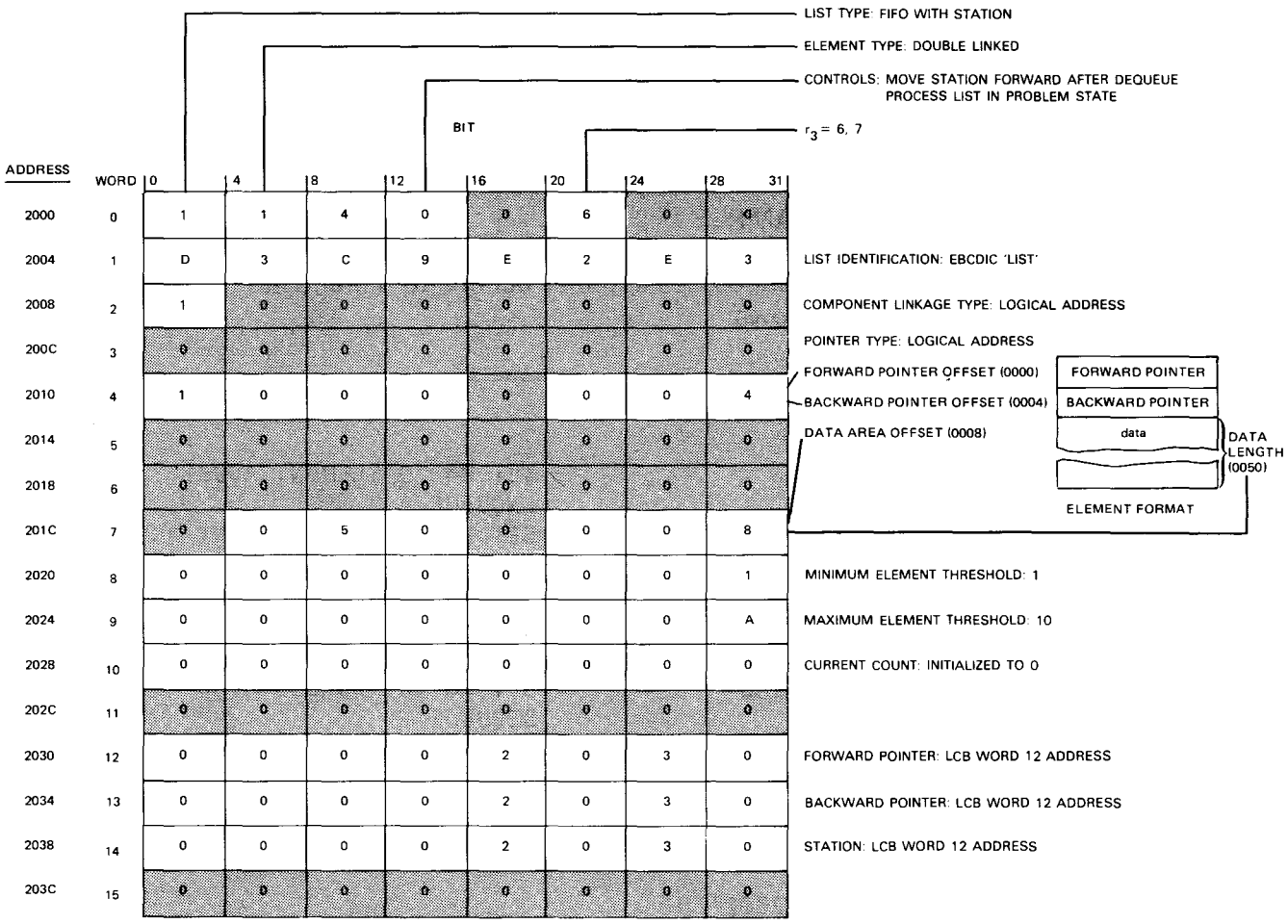
To execute your list control program, you execute a STEP QUEUE instruction with bit 8 set to 1. Other fields within STEP QUEUE are used by LCP instructions themselves, and their use is discussed for each instruction. In general, you should set those bits in STEP QUEUE and your LCP instructions to 0 that you do not otherwise need to set to 1.

14.9. LIST PROCESSING EXAMPLE

Here we show how we can use list processing instructions to manipulate a list of our own creation. The coding we use follows:

```
 1. QBEGIN  BALR 12,Ø
 2.         USING *,12
 3.         LA    6,ELEMENTS              SET R6 TO ELEMENT ADDRESS
 4.         ENQ   LCBLOCK,Ø               ENQUEUE A
 5.         LA    6,88(6)                 SET R6 TO NEXT ELEMENT ADDRESS
 6.         ENQ   LCBLOCK,Ø               ENQUEUE B
 7.         LA    6,88(6)                 SET R6 TO NEXT ELEMENT ADDRESS
 8.         ENQ   LCBLOCK,Ø               ENQUEUE C
 9.         STEP  LCBLOCK,B'Ø1ØØØØØØ'     MOVE STATION BACKWARDS TO B
10.         L     6,LCBLOCK+56            SET R6 TO ADDRESS OF B
11.         DEQ   LCBLOCK,Ø               DEQUEUE B
12.         LA      6,Ø                   CLEAR R6
13.         DEQ   LCBLOCK,Ø               DEQUEUE FIRST ELEMENT (A)
            .
            .
            .
14.         ORG   QBEGIN+X'2ØØØ'
15. LCBLOCK DC    X'114ØØ6ØØ'             WORD Ø
16.         DC    C'LIST'                 WORD 1
17.         DC    X'1ØØØØØØØ'             WORD 2
18.         DC    F'Ø'                    WORD 3
19.         DC    X'1Ø'                   WORD 4 BITS Ø-7
20.         DC    AL3(4)                  WORD 4 BITS 8-31
21.         DC    2F'Ø'                   WORD 5-6
22.         DC    H'8Ø'                   WORD 7 BITS Ø-15
23.         DC    H'8'                    WORD 7 BITS 16-31
24.         DC    F'1,1Ø'                 WORDS 8-9
25.         DC    2F'Ø'                   WORDS 10-11
26.         DC    3A(LCBLOCK+48)          WORDS 12-14
27.         DC    F'Ø'                    WORD 15
            .
            .
            .
28.         ORG   QBEGIN+X'3ØØØ'
29. ELEMENTS EQU  *
```

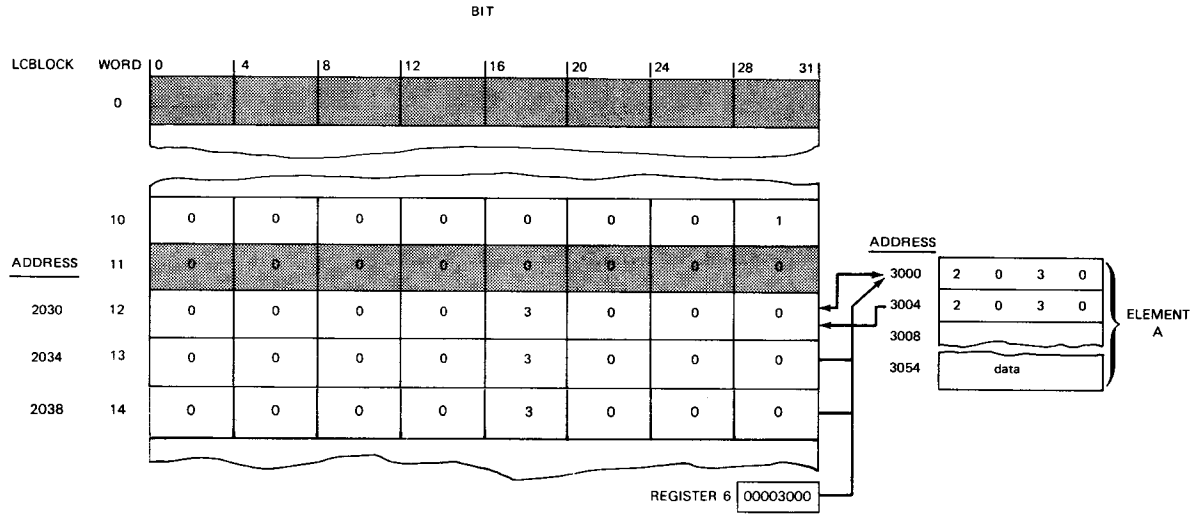
To learn what the instructions at lines 1—13 do, we should first analyze the LCB they all use. The LCB defined by lines 15—27, LCBLOCK, begins at location 2000 (the ORG directive at line 14). The following chart shows LCBLOCK, initialized according to the rules for FIFO lists; shaded areas represent unused fields.



The list elements are simple 88-byte sections of main storage, each containing one forward pointer, one backward pointer, and an 80-byte data area. The LOAD ADDRESS instruction at line 3 puts the address of ELEMENTS, the first element to be enqueued, in register 6, the even-numbered r_3 register. Assume that ELEMENTS is at location 3000; register 6 will be set as follows:

Register 6 | 00003000 |

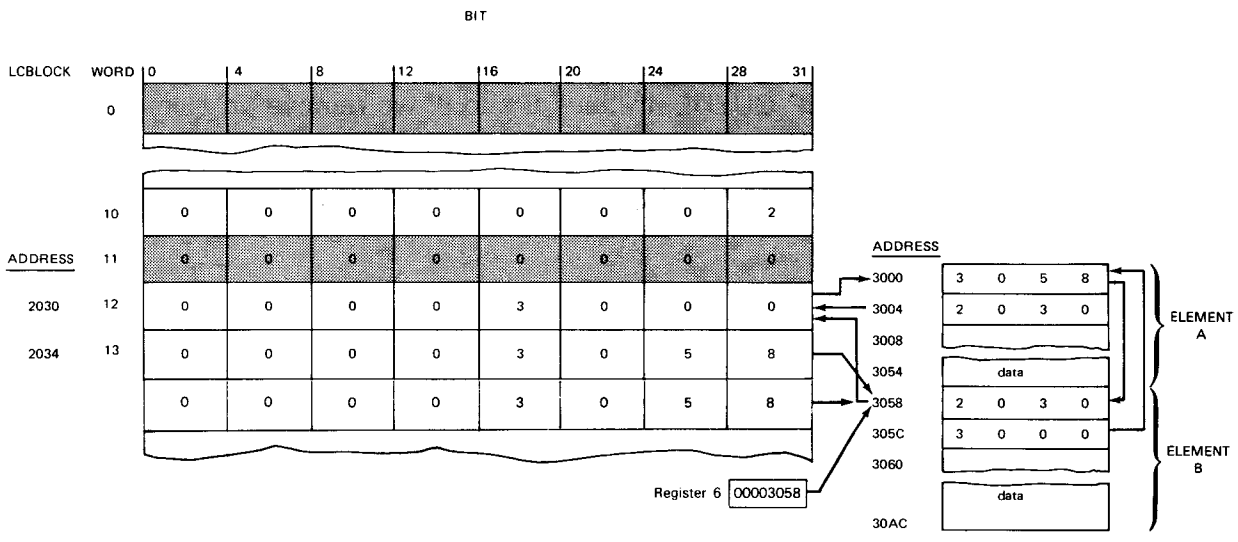
The ENQUEUE instruction at line 4 adds the element at location ELEMENTS to the list (element A). After execution of the ENQUEUE, the condition code is set to 0 and the list and element A are updated as follows:



Only LCB word 10 (CURRENT COUNT), 12 (forward pointer), 13 (backward pointer), and 14 (station) are changed by ENQUEUE. Register 6 remains set to the address of element A. In line 5, we prepare for the next list processing operation by making register 6 point to another section of main storage at ELEMENT+88, which is to be element B:

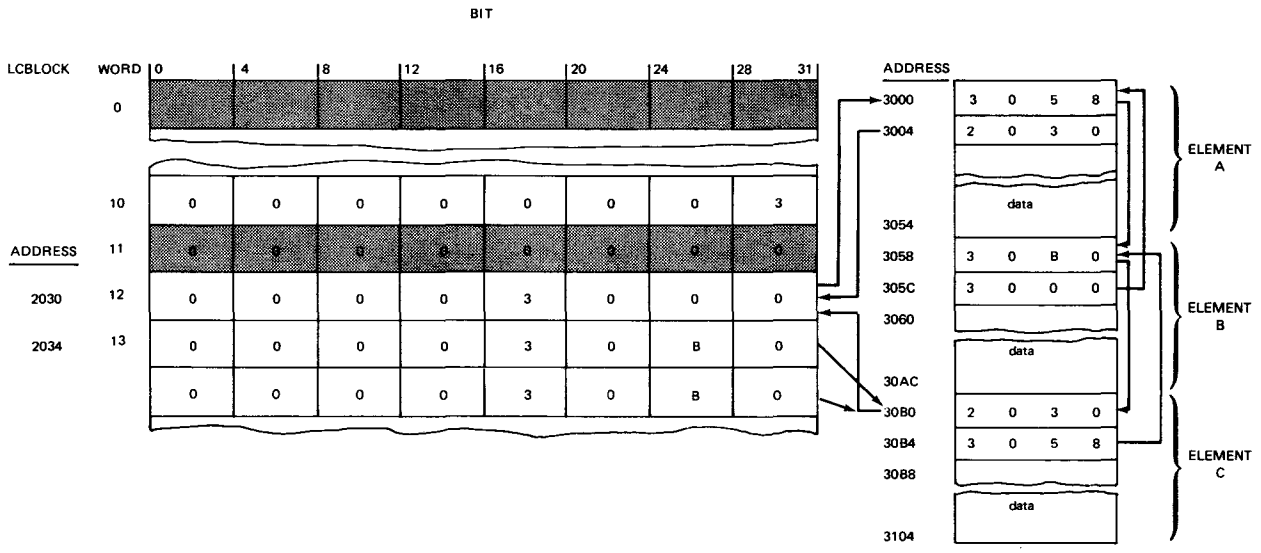
Register 6 | 00003058 |

At line 6, the next ENQUEUE instruction adds element B to the list, leaving the condition code set to 0 and updating the list as follows:



As before, register 6 remains unchanged by the operation although the instruction creates new links between LCBLOCK and its two elements. Note that element B is the last element you would encounter if you follow the forward pointers from LCBLOCK through its elements. This is true no matter where in main storage you put element B; as long as it does not overlap other list elements, it can lie anywhere in relation to the LCB or to other elements. Note also that the station in word 14 points to element B; it does so because B is the element most recently added to the list.

We wish to add a third element, called element C. To do this we set register 6 to point to address ELEMENTS+176 (line 7) and execute ENQUEUE once again (line 8). After execution, the condition code remains set to 0 and the list is updated as follows:

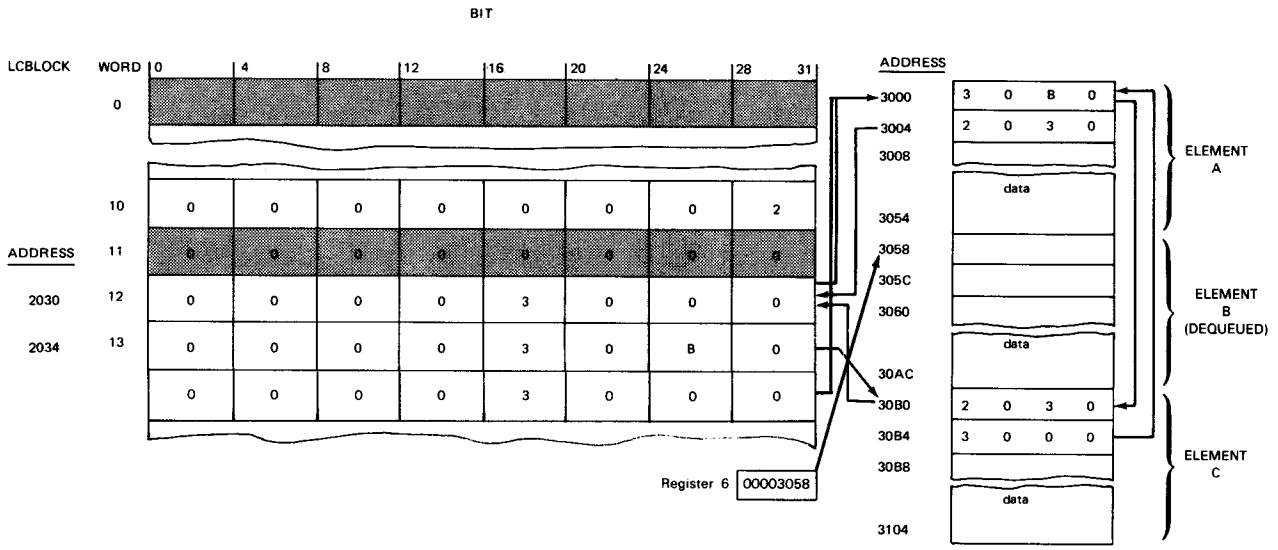


As you can see, the station points to element C when the ENQUEUE instruction in line 8 finishes. The CURRENT COUNT field (LCB word 10) holds a value of 3, representing the three elements the list now contains.

The next instruction, at line 9, is a STEP QUEUE in which we set bit 9 of its object code to 1; this indicates that the station is to be moved backward, pointing to the previous element in the list. By following the element C backward pointer shown, we can see that the station now points to element B:

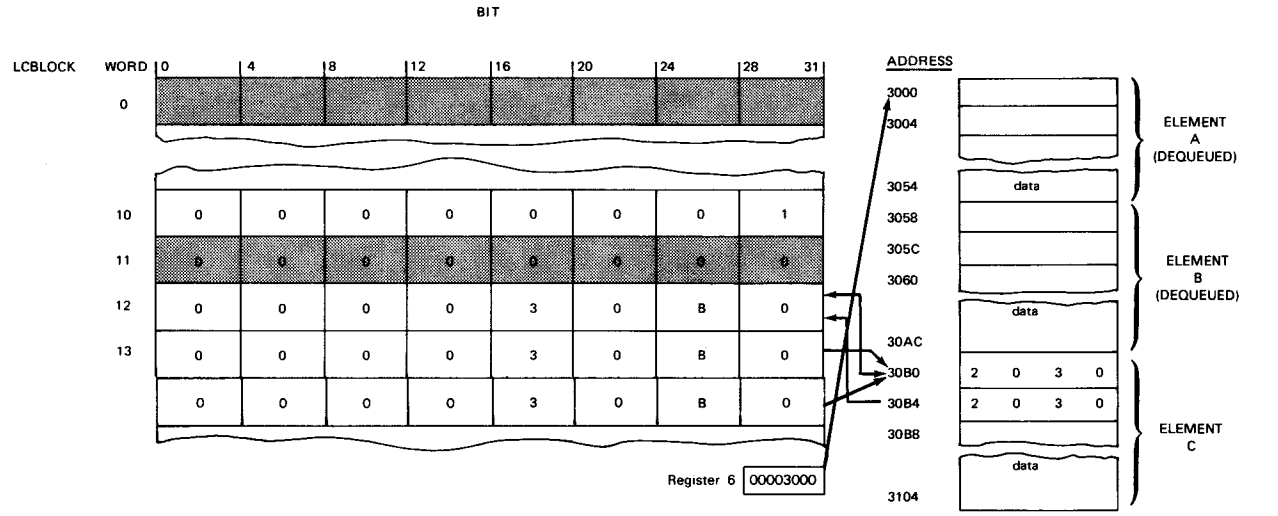
LCB word 14: 00003058

The condition code is set to 0, indicating a successful step. Except for the station, no fields in the list are changed by STEP QUEUE. In lines 8 and 9, we remove an element from the list. We first load the station pointer, currently pointing to element B in register 6. Because the even-numbered r_3 register (6) now contains a nonzero value, the instruction interprets it as the address of the element to be removed, element B. The list is updated as follows:



As you can see, register 6 continues to point to element B, now removed from the list. Note that the pointers of elements A and C now point to each other, neither of them to B. Note also that the station now points to the element preceding the one removed (as directed by LCBLOCK word 0, bit 10). The condition code is set to 0, and the CURRENT COUNT FIELD is reduced by 1.

We wish to perform one more operation, to remove the first element in the list. To do so, we first put all zeros in register 6 (using the LOAD ADDRESS instruction of line 12). This action ensures that it is the first element that is removed by the DEQUEUE instruction of line 13. That instruction updates the list as follows:



Note that register 6 is set to point to element A, the first element of the list and the one removed. All pointers now connect the LCB with element C alone. The CURRENT COUNT field is reduced to 1, which is the value we placed in the MINIMUM ELEMENT THRESHOLD field of the LCB. Consequently, the condition code is set to 1 (underflow).

Appendix E. Instruction Listings

Included in this appendix are alphabetic listings of the mnemonic codes (Table E—1) and instruction names (Table E—2) and a numeric list of the machine codes (Table E—3).

Table E—1. Mnemonic List of Instructions (Part 1 of 4)

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
A	Add	5A	4	$r_1,d_2(x_2,b_2)$	$r_1s_2(x_2)$
AD	Add Normalized, Long	6A	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
ADR	Add Normalized, Long	2A	2	r_1,r_2	r_1,r_2
AE	Add Normalized, Short	7A	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
AER	Add Normalized, Short	3A	2	r_1,r_2	r_1,r_2
AH	Add Half Word	4A	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
AI	Add Immediate	9A	4	$d_1(b_1),i_2$	s_1,i_2
AL	Add Logical	5E	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
ALR	Add Logical	1E	2	r_1,r	r_1,r_2
AP	Add Decimal	FA	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
AR	Add	1A	2	r_1,r_2	r_1,r_2
AU	Add Unnormalized, Short	7E	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
AUR	Add Unnormalized, Short	3E	2	r_1,r_2	r_1,r_2
AW	Add Unnormalized, Long	6E	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
AWR	Add Unnormalized, Long	2E	2	r_1,r_2	r_1,r_2
BAL	Branch and Link	45	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
BALR	Branch and Link	05	2	r_1,r_2	r_1,r_2
BC	Branch on Condition	47	4	$i,d_2(x_2,b_2)$	$i,s_2(x_2)$
BCR	Branch on Condition	07	2	i,r_2	i,r_2
BCT	Branch on Count	46	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
BCTR	Branch on Count	06	2	r_1,r_2	r_1,r_2
BXH	Branch on Index High	86	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
BXLE	Branch on Index Low or Equal	87	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
C	Compare Algebraic	59	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
CD	Compare, Long	69	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
CDR	Compare, Long	29	2	r_1,r_2	r_1,r_2
CE	Compare, Short	79	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
CER	Compare, Short	39	2	r_1,r_2	r_1,r_2
CH	Compare Half Word	49	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
CL	Compare Logical	55	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$

Table E—1. Mnemonic List of Instructions (Part 2 of 4)

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
CLC	Compare Logical	D5	6	$d_1,(l,b_1),d_2(b_2)$	$s_1(l),s_2$
CLCL	Compare Logical Characters Long	0F	2	r_1,r_2	r_1,r_2
CLI	Compare Logical Immediate	95	4	$d_1(b_1),i_2$	s_1,i_2
CLIS	Compare Logical Immediate and Skip	E1	6	$d_1(b_1),i_2,m_3,d_4$	s_1,i_2,m_3,s_4
CLM	Compare Logical Characters Under Mask	BD	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2
CLR	Compare Logical	15	2	r_1,r_2	r_1,r_2
CLRCH	Clear Channel	9F02	4	(Privileged)	(Privileged)
CLRDV	Clear Device	9D	4	(Privileged)	(Privileged)
CP	Compare Decimal	F9	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
CR	Compare Algebraic	19	2	r_1,r_2	r_1,r_2
CSM	Compare and Swap Under Mask	B9	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
CVB	Convert to Binary	4F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
CVD	Convert to Decimal	4E	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
D	Divide	5D	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
DD	Divide, Long	6D	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
DDR	Divide, Long	2D	2	r_1,r_2	r_1,r_2
DE	Divide, Short	7D	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
DEQ	Dequeue	B4	4	$d_1(b_1),i_2$	s_1,i_2
DER	Divide, Short	3D	2	r_1,r_2	r_1,r_2
DP	Divide Decimal	FD	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),,s_2(l_2)$
DR	Divide	1D	2	r_1,r_2	r_1,r_2
ED	Edit	DE	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
EDMK	Edit and Mark	DF	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
EIO	Enqueue I/O	E0	6	(Privileged)	(Privileged)
ENQ	Enqueue	B3	6	$d_1(b_1),i_2$	s_1,i_2
EX	Execute	44	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
EXD	Execute Diagnose	8300	4	(Privileged)	(Privileged)
HDR	Halve, Long	24	2	r_1,r_2	r_1,r_2
HDV	Halt Device	9E	4	(Privileged)	(Privileged)
HER	Halve, Short	34	2	r_1,r_2	r_1,r_2
HPR	Halt and Proceed	99	4	(Privileged)	(Privileged)
IC	Insert Character	43	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
ICM	Insert Characters Under Mask	BF	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2
ISK	Insert Storage Key	09	2	(Privileged)	(Privileged)
L	Load	58	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LA	Load Address	41	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LCDR	Load Complement, Long	23	2	r_1,r_2	r_1,r_2
LCER	Load Complement, Short	33	2	r_1,r_2	r_1,r_2
LCHR	Load Channel Register	9F03	4	(Privileged)	(Privileged)
LCR	Load Complement	13	2	r_1,r_2	r_1,r_2
LCTL	Load Control	B7	4	(Privileged)	(Privileged)
LD	Load, Long	68	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LDA	Load Directive Address	51	4	(Privileged)	(Privileged)
LDR	Load, Long	28	2	r_1,r_2	r_1,r_2
LE	Load, Short	78	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LER	Load, Short	38	2	r_1,r_2	r_1,r_2
LH	Load Half Word	48	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LIA	Load I/O Address	61	4	(Privileged)	(Privileged)

Table E—1. Mnemonic List of Instructions (Part 3 of 4)

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
LM	Load Multiple	98	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
LNDR	Load Negative, Long	21	2	r_1,r_2	r_1,r_2
LNER	Load Negative, Short	31	2	r_1,r_2	r_1,r_2
LNR	Load Negative	11	2	r_1,r_2	r_1,r_2
LPDR	Load Positive, Long	20	2	r_1,r_2	r_1,r_2
LPER	Load Positive, Short	30	2	r_1,r_2	r_1,r_2
LPR	Load Positive	10	2	r_1,r_2	r_1,r_2
LPSW	Load Program Status Word	82	4	(Privileged)	(Privileged)
LR	Load	18	2	r_1,r_2	r_1,r_2
LRC	Longitudinal Redundancy Check	830E	4	(Privileged)	(Privileged)
LRR	Load Relocation Register	A3	4	(Privileged)	(Privileged)
LTDR	Load and Test, Long	22	2	r_1,r_2	r_1,r_2
LTER	Load and Test, Short	32	2	r_1,r_2	r_1,r_2
LTR	Load and Test	12	2	r_1,r_2	r_1,r_2
M	Multiply	5C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MD	Multiply, Long	6C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MDR	Multiply, Long	2C	2	r_1,r_2	r_1,r_2
ME	Multiply, Short	7C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MER	Multiply, Short	3C	2	r_1,r_2	r_1,r_2
MH	Multiply Half Word	4C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MIO	Move I/O	81	4	(Privileged)	(Privileged)
MP	Multiple Decimal	FC	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
MR	Multiply	1C	2	r_1,r_2	r_1,r_2
MSS	Modify Storage and Skip	E3	6	$d_1(i_1,b_1),d_2(i_3,b_2)$	$s_1(i_1),s_2(i_3)$
MVC	Move Characters	D2	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
MVCL	Move Character Long	0E	2	r_1,r_2	r_1,r_2
MVI	Move Immediate	92	4	$d_1(b_1),i_2$	s_1,i_2
MVN	Move Numerics	D1	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
MVO	Move With Offset	F1	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
MVZ	Move Zones	D3	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
N	AND Logical	54	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
NC	AND Logical	D4	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
NI	AND Logical Immediate	94	4	$d_1(b_1),i_2$	s_1,i_2
NR	AND Logical	14	2	r_1,r_2	r_1,r_2
O	OR Logical	56	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
OC	OR Logical	D6	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
OI	OR Logical Immediate	96	4	$d_1(b_1),i_2$	s_1,i_2
OR	OR Logical	16	2	r_1,r_2	r_1,r_2
PACK	Pack	F2	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
RESET	Reset	8301	4	(Privileged)	(Privileged)
S	Subtract	5B	4	$r_1,s_2(x_2)$	r_1,s_2
SD	Subtract Normalized, Long	6B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SDR	Subtract Normalized, Long	2B	2	r_1,r_2	r_1,r_2
SDV	Start Device	9C	4	(Privileged)	(Privileged)
SE	Subtract Normalized, Short	7B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SER	Subtract Normalized, Short	3B	2	r_1,r_2	r_1,r_2
SH	Subtract Half Word	4B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SHL	Shift Logical	9B	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2

Table E—1. Mnemonic List of Instructions (Part 4 of 4)

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
SL	Subtract Logical	5F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SLA	Shift Left Single Algebraic	8B	4	$r_1,d_2(b_2)$	r_1,s_2
SLDA	Shift Left Double Algebraic	8F	4	$r_1,d_2(b_2)$	r_1,s_2
SLDL	Shift Left Double Logical	8D	4	$r_1,d_2(b_2)$	r_1,s_2
SLL	Shift Left Single Logical	89	4	$r_1,d_2(b_2)$	r_1,s_2
SLM	Supervisor Load Multiple	B8	4	(Privileged)	(Privileged)
SLR	Subtract Logical	1F	2	r_1,r_2	r_1,r_2
SP	Subtract Decimal	FB	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
SPM	Set Program Mask	04	2	r_1	r_1
SR	Subtract	1B	2	r_1,r_2	r_1,r_2
SRA	Shift Right Single Algebraic	8A	4	$r_1,d_2(b_2)$	r_1,s_2
SRDA	Shift Right Double Algebraic	8E	4	$r_1,d_2(b_2)$	r_1,s_2
SRDL	Shift Right Double Logical	8C	4	$r_1,d_2(b_2)$	r_1,s_2
SRL	Shift Right Single Logical	88	4	$r_1,d_2(b_2)$	r_1,s_2
SRP	Shift and Round Decimal	F0	6	$d_1(l_1,b_1),d_2(b_2),i_3$	$s_1(l_1),s_2,i_3$
SSK	Set System Key	08	2	(Privileged)	(Privileged)
SSM	Set System Mask	80	4	(Privileged)	(Privileged)
SSTM	Supervisor Store Multiple	B0	4	(Privileged)	(Privileged)
ST	Store	50	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STC	Store Character	42	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STCM	Store Characters Under Mask	BE	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2
STCTL	Store Control	B6	4	(Privileged)	(Privileged)
STD	Store Long	60	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STE	Store Short	70	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STEP	Step Queue	B5	4	$d_1(b_1),i_2$	s_1,i_2
STH	Store Half Word	40	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STM	Store Multiple	90	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
STR	Service Timer Register	03	2	(Privileged)	(Privileged)
STRR	Store Relocation Register	A2	4	(Privileged)	(Privileged)
STS	Store Status	8302	4	(Privileged)	(Privileged)
SU	Subtract Unnormalized, Short	7F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SUR	Subtract Unnormalized, Short	3F	2	r_1,r_2	r_1,r_2
SVC	Supervisor Call	0A	2	i	i
SW	Subtract Unnormalized, Long	6F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SWR	Subtract Unnormalized, Long	2F	2	r_1,r_2	r_1,r_2
TM	Test Under Mask	91	4	$d_1(b_1),i_2$	s_1,i_2
TMS	Test Under Mask and Skip	E2	6	$d_1(b_1),i_2,m_3,d_4$	s_1,i_2,m_3,s_4
TR	Translate	DC	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
TRT	Translate and Test	DD	6	$d_1(l,b_1),d_2(b_2)$	
TS	Test and Set	93	4	$d_1(b_1)$	s_1
UNPK	Unpack	F3	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
X	Exclusive OR	57	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
XC	Exclusive OR	D7	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
XI	Exclusive OR, Immediate	97	4	$d_1(b_1),i_2$	s_1,i_2
XR	Exclusive OR	17	2	r_1,r_2	r_1,r_2
ZAP	Zero and Add Decimal	F8	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$

Table E—2. Alphabetic Listing of Instructions (Part 1 of 6)

Instruction Name	Machine Code	Mnemonic
Add	1A	AR
Add	5A	A
Add Decimal	FA	AP
Add Half Word	4A	AH
Add Immediate	9A	AI
Add Immediate	A6	AI
Add Logical	1E	ALR
Add Logical	5E	AL
Add Normalized, Long	2A	ADR
Add Normalized, Long	6A	AD
Add Normalized, Short	3A	AER
Add Normalized, Short	7A	AE
Add Unnormalized, Long	2E	AWR
Add Unnormalized, Long	6E	AW
Add Unnormalized, Short	3E	AUR
Add Unnormalized, Short	7E	AU
AND	14	NR
AND	54	N
AND	94	NI
AND	D4	NC
Branch and Link	05	BALR
Branch and Link	45	BAL
Branch on Condition	07	BCR
Branch on Condition	47	BC
Branch on Count	06	BCTR
Branch on Count	46	BCT
Branch on Index High	86	BXH
Branch on Index Low or Equal	87	BXLE
Clear Channel — Privileged	9F02	CLRCH

Table E—2. Alphabetic Listing of Instructions (Part 2 of 6)

Instruction Name	Machine Code	Mnemonic
Clear Device — Privileged	9D	CLRDV
Compare	19	CR
Compare	59	C
Compare and Swap Under Mask	B9	CSM
Compare Decimal	F9	CP
Compare Half Word	49	CH
Compare Logical	15	CLR
Compare Logical	55	CL
Compare Logical	95	CLI
Compare Logical	D5	CLC
Compare Logical Characters Under Mask	BD	CLM
Compare Logical Immediate and Skip	E1	CLIS
Compare Logical Characters Long	OF	CLCL
Compare, Long	29	CDR
Compare, Long	69	CD
Compare, Short	39	CER
Compare, Short	79	CE
Convert to Binary	4F	CVB
Convert to Decimal	4E	CVD
Dequeue	B4	DEQ
Divide	1D	DR
Divide	5D	D
Divide Decimal	FD	DP
Divide, Long	2D	DDR
Divide, Long	6D	DD
Divide, Short	3D	DER
Divide, Short	7D	DE
Edit	DE	ED
Edit and Mark	DF	EDMK
Enqueue	B3	ENQ

Table E—2. Alphabetic Listing of Instructions (Part 3 of 6)

Instruction Name	Machine Code	Mnemonic
Enqueue I/O — Privileged	EO	EIO
Exclusive OR	17	XR
Exclusive OR	57	X
Exclusive OR	97	XI
Exclusive OR	D7	XC
Execute	44	EX
Execute Diagnose — Privileged	8300	EXD
Halt and Proceed — Privileged	99	HPR
Halt Device — Privileged	9E01	HDV
Halve, Long	24	HDR
Halve, Short	34	HER
Insert Character	43	IC
Insert Characters Under Mask	BF	ICM
Insert Storage Key — Privileged	09	ISK*
Load	18	LR
Load	58	L
Load Address	41	LA
Load and Test	12	LTR
Load and Test, Long	22	LTDR
Load and Test, Short	32	LTER
Load Channel Register — Privileged	9F03	LCHR
Load Complement	13	LCR
Load Complement, Long	23	LCDR
Load Complement, Short	33	LCER
Load Control — Privileged	B7	LCTL
Load Directive Address — Privileged	51	LDA
Load Half Word	48	LH
Load I/O Address — Privileged	61	LIA
Load, Long	28	LDR
Load, Long	68	LD

Table E—2. Alphabetic Listing of Instructions (Part 4 of 6)

Instruction Name	Machine Code	Mnemonic
Load Multiple	98	LM
Load Negative	11	LNR
Load Negative, Long	21	LNDR
Load Negative, Short	31	LNER
Load Positive	10	LPR
Load Positive, Long	20	LPDR
Load Positive, Short	30	LPER
Load PSW — Privileged	82	LPSW
Load Relocation Register	A3	LRR
Load, Short	38	LER
Load, Short	78	LE
Longitudinal Redundancy Check — Privileged	830E	LRC
Modify Storage and Skip	E3	MSS
Move	92	MVI
Move	D2	MVC
Move I/O — Privileged	81	MIO
Move Characters Long	OE	MVCL
Move Numerics	D1	MVN
Move With Offset	F1	MVO
Move Zones	D3	MVZ
Multiply	1C	MR
Multiply	5C	M
Multiply Decimal	FC	MP
Multiply Half Word	4C	MH
Multiply, Long	2C	MDR
Multiply, Long	6C	MD
Multiply, Short	3C	MER
Multiply, Short	7C	ME
OR	16	OR
OR	56	O

Table E—2. Alphabetic Listing of Instructions (Part 5 of 6)

Instruction Name	Machine Code	Mnemonic
OR	96	OI
OR	D6	OC
Pack	F2	PACK
Reset — Privileged	8301	RESET
Service Timer Register — Privileged	03	STR
Set Program Mask	04	SPM
Set Storage Key — Privileged	08	SSK*
Set System Mask — Privileged	80	SSM
Shift and Round Decimal	F0	SRP
Shift Left Double	8F	SLDA
Shift Left Double Logical	8D	SLDL
Shift Left Single	8B	SLA
Shift Left Single Logical	89	SLL
Shift Logical	9B	SHL
Shift Right Double	8E	SRDA
Shift Right Double Logical	8C	SRDL
Shift Right Single	8A	SRA
Shift Right Single Logical	88	SRL
Start Device — Privileged	9C02	SDV
Step Queue	B5	STEP
Store	50	ST
Store Character	42	STC
Store Characters Under Mask	BE	STCM
Store Control — Privileged	B6	STCTL
Store Half Word	40	STH
Store, Long	60	STD
Store Multiple	90	STM
Store Relocation Register — Privileged	A2	STRR
Store, Short	70	STE
Store Status — Privileged	8302	STS

Table E—2. Alphabetic Listing of Instructions (Part 6 of 6)

Instruction Name	Machine Code	Mnemonic
Subtract	1B	SR
Subtract	5B	S
Subtract Decimal	FB	SP
Subtract Half Word	4B	SH
Subtract Logical	1F	SLR
Subtract Logical	5F	SL
Subtract Normalized, Long	2B	SDR
Subtract Normalized, Long	6B	SD
Subtract Normalized, Short	3B	SER
Subtract Normalized, Short	7B	SE
Subtract Unnormalized, Long	2F	SWR
Subtract Unnormalized, Long	6F	SW
Subtract Unnormalized, Short	3F	SUR
Subtract Unnormalized, Short	7F	SU
Supervisor Call	0A	SVC
Supervisor Load Multiple — Privileged	B8	SLM
Supervisor Store Multiple — Privileged	B0	SSTM
Test and Set	93	TS
Test Under Mask	91	TM
Test Under Mask and Skip	E2	TMS
Translate	DC	TR
Translate and Test	DD	TRT
Unpack	F3	UNPK
Zero and Add	F8	ZAP

*Added as a feature.

Table E—3. List of Instructions by Machine Code (Part 5 of 6)

Machine Code	Mnemonic	Instruction Name
93	TS	Test and Set
94	NI	AND
95	CLI	Compare Logical
96	OI	OR
97	XI	Exclusive OR
98	LM	Load Multiple
99	HPR	Halt and Proceed — Privileged
9A	AI	Add Immediate
9B	SHL	Shift Logical
9C	SDV	Start Device — Privileged
9D	CLRDV	Clear Device — Privileged
9E01	HDV	Halt Device — Privileged
9F02	CLRCH	Clear Channel — Privileged
9F03	LCHR	Load Channel Register — Privileged
A2	STRR	Store Relocation Register — Privileged
A3	LRR	Load Relocation Register — Privileged
B0	SSTM	Supervisor Store Multiple — Privileged
B3	ENQ	Enqueue
B4	DEQ	Dequeue
B5	STEP	Step Queue
B6	STCTL	Store Control — Privileged
B7	LCTL	Load Control — Privileged
B8	SLM	Supervisor Load Multiple — Privileged
B9	CSM	Compare and Swap Under Mask
BD	CLM	Compare Logical Characters Under Mask
BE	STCM	Store Characters Under Mask
BF	ICM	Insert Characters Under Mask
D1	MVN	Move Numerics
D2	MVC	Move
D3	MVZ	Move Zones

Table E—3. List of Instructions by Machine Code (Part 6 of 6)

Machine Code	Mnemonic	Instruction Name
D4	NC	AND
D5	CLC	Compare Logical
D6	OC	OR
D7	XC	Exclusive OR
DC	TR	Translate
DD	TRT	Translate and Test
DE	ED	Edit
DF	EDMK	Edit and Mark
E0	EIO	Enqueue I/O — Privileged
E1	CLIS	Compare Logical Immediate and Skip
E2	TMS	Test Under Mask and Skip
E3	MSS	Modify Storage and Skip
F0	SRP	Shift and Round Decimal
F1	MVO	Move With Offset
F2	PACK	Pack
F3	UNPK	Unpack
F8	ZAP	Zero and Add
F9	CP	Compare Decimal
FA	AP	Add Decimal
FB	SP	Subtract Decimal
FC	MP	Multiply Decimal
FD	DP	Divide Decimal

*Added as a feature.