SPERRY UNIVAC

PUBLICATIONS REVISION

9200/9300 Series

Tape/Disc Assembler

Programmer Reference

UP-7508 Rev. 3

This SPERRY UNIVAC ™ 9200/9300 Series Library Memo announces the release and availability of "SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler Programmer Reference," UP-7508 Rev. 3. This is a Standard Library Item (SLI).

This revision provides additions and corrections throughout the manual.

Destruction Notice: This revision supersedes and replaces "UNIVAC 9200/9200II/9300/9300II Systems Tape/Disc Assembler Programmers Reference," UP-7508 Rev. 2 released on Library Memo dated December 1969 and associated update packages. Please destroy all copies of UP-7508 Rev. 2, UP-7795.2, UP-7508 Rev. 2–B, UP-7508 Rev. 2–C, UP-7508 Rev. 2–D, UP–7508 Rev. 2–E and/or their Library Memos.

Additional copies may be ordered by your local Sperry Univac Univac Representative.

Mailing Lists
217, 630 and
692

Mailing Lists 51D, 52, 53, 53D, 54,
54D, 55, 55D and 56
(Covers and 160 pages)

Library Memo for
UP-7508 Rev. 3

October, 1974

SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler

Programmer Reference

H

SPERRY UNIVAC COMPUTER SYSTEMS

UP-7508 Rev. 3

This document contains the latest information available at the time of publication. However, Sperry Univac reserves the right to modify or revise its contents. To ensure that you have the most recent information, contact your local Sperry Univac representative.

Sperry Univac is a division of Sperry Rand Corporation.

FASTRAND, PAGEWRITER, SPERRY UNIVAC, UNISCOPE, UNISERVO, and UNIVAC are trademarks of the Sperry Rand Corporation.

PRINTED IN U.S.A.

PAGE STATUS SUMMARY

ISSUE: UP-7508 Rev. 3

Part/Section	Page Number	Update Level
Cover/Disclaimer		
PSS	1	
Contents	1 thru 6	
1	1 thru 5	
2	1 thru 23	
3	1 thru 55	
4	1 thru 19	
5	1 thru 7	
6	1 thru 9	
7	1 thru 12	
8	1 thru 14	
Appendix A	1	
Appendix B	1 thru 4	
User Comment Sheet		

Part/Section	Page Number	Update Level

Part/Section	Page Number	Update Level

All the technical changes are denoted by an arrow (→) in the margin. A downward pointing arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward pointing arrow (↑) is found. A horizontal arrow (→) pointing to a line indicates a technical change in only that line. A horizontal arrow located between two consecutive lines indicates technical changes in both lines or deletions.

Contents

7. PREPARATION FOR ASSEMBLY

8. LINKER

APPENDIXES

FIGURES

TABLES

1. Introduction

1.1. GENERAL

The purposes of this manual are to afford the information necessary for programming the SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler and a reference handbook for the programmer. The use of the assembler coding form and the operational characteristics of the assembler are described in succeeding sections. These characteristics are summarized briefly as follows:

- Mnemonic Operation Code

  A fixed name, suggestive of the nature of the instruction and consisting of 2, 3, or 4 letters, is assigned to each instruction. This applies to each variant of the branch instructions as a further aid in writing the program.

- Symbolic Addressing and Automatic Storage Assignment

  Symbolic labels can be assigned to instructions or groups of data. An instruction then references the labeled data by the label rather than the storage address. In many cases, other data required by the instruction (such as operand length) can be supplied automatically by the assembler. The assembler also keeps track of all storage locations used for a particular program and assigns all incoming instructions and data to specific locations. The assembler performs base register and displacement calculations.

- Flexible Data Representation

  Data may be represented in the assembler in binary, hexadecimal, decimal, or character notation, allowing the programmer to choose the most suitable form for each constant.

- Relocatable Programs and Program Linking

  Programs are prepared by the assembler in absolute or relocatable form. In absolute form, the program storage locations are specified within the program. In relocatable form, the actual storage locations to be occupied by a program need not be specified, but if specified, they may be altered easily before loading. Provisions are made for linking together, loading, and running as one program the results of separate assemblies. The machine time needed to make changes to one part of a program is reduced by use of this provision. The input of one assembly can be divided into separate sections, each of which consists of a group of instructions or data, or both, occupying contiguous locations. The relative positions of the various sections can be declared at the time the program is linked. The output of the assembler is not in loadable form; it must be linked before loading.

- Macro Instruction Facility

  The assembler includes a macro instruction facility that reduces the effort required to write patterns of coding repeated in one program or common to several programs. One command to the assembler results in the inclusion in the object program of many instructions and/or constants, or merely results in establishing one or more values for use elsewhere within the program. The flexibility of the facility allows a macro to be written so that the generated pattern of coding can vary widely depending upon the parameters supplied with the call.

- Program Listing

  A printed listing of source and object codes is one output of the assembler. This listing includes error message flags marking any errors detected by the assembler. Source-code errors do not cause the assembly process to halt; the assembler continues to process the remainder of the source code, performing its usual error checks, minimizing the number of assemblies required to produce error-free code.

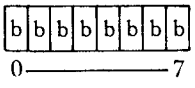
A computer with a tape or disc subsystem can handle data faster than a comparable card-oriented system. The SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler relies heavily on the larger storage and file-handling capabilities of the tape or disc subsystem to perform many processing operations normally difficult to perform with cards only. A built-in macro instruction facility is one of the features of the assembler. Another feature is the linker program that provides a method of combining several separately assembled modules into one executable program.

1.2. DATA FORMATS

The basic unit of data in the SPERRY UNIVAC 9000 Series is an 8-bit byte to which a parity bit is added when the byte is stored in main storage. A byte may represent a character or a number. Main storage locations are numbered consecutively. Each address specifies on byte of information. The address of a group of bytes is the address of the leftmost byte of the group. The bits in a byte also are numbered from left to right, starting with zero. Each of the formats is illustrated in Table 1–1.

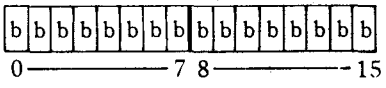
Table 1–1. Data Formats

Byte



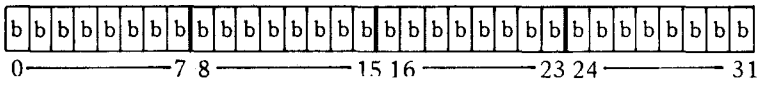
Half-word data formats consist of two consecutive bytes.

Half Word



Full-word data formats consist of four consecutive bytes.

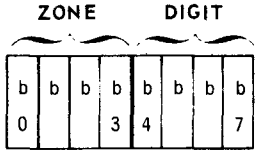
Full Word



Variable data formats consist of a variable number of consecutive bytes.

Variable Data Format

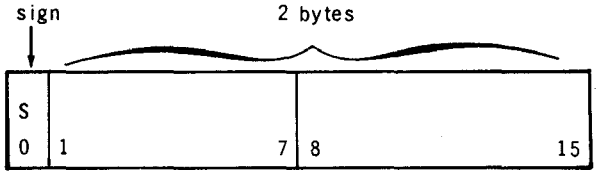
It is possible to store any one of 256 different bit combinations in the byte. Data can be represented in various forms to the programmer; however, certain restrictions are imposed if the data is to be printed or processed arithmetically. The contents of a byte can be considered a binary number, a decimal number, an alphabetic or symbolic character, or logical information. A field used to represent a binary number uses all of the bit positions (except the sign bit) to contain the value; however, each byte in a field representing a decimal number, alphabetic character, or symbol is considered to be divided into zone and digit portions. The zone portion is the most significant four bits; the digit portion is the least significant four bits.



1.2.1. Binary Number Representation

The binary arithmetic operand is a 16-bit binary number. The bits of the operand are stored in adjacent locations. These two consecutive bytes constitute a half word when the address of the more significant byte is an integral multiple of two. Such an address is a half-word boundary.

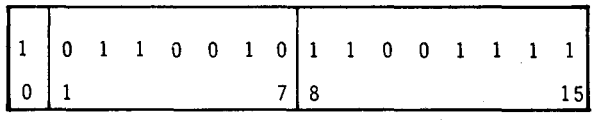
Positive binary numbers are represented in conventional binary notation with a 0 bit in the most significant bit position; this is the sign bit for a binary number.



Negative numbers are represented as the twos complement of the number with a 1 bit in the sign position. For instance, the representation of the negative number –19,761 would be determined as follows:

The binary equivalent of 19,761 is:	0 1 0 0 1 1 0 1 0 0 1 1 0 0 0 1
The ones complement is:	1 0 1 1 0 0 1 0 1 1 0 0 1 1 1 0
The twos complement formed by adding 1 is:	1 0 1 1 0 0 1 0 1 1 0 0 1 1 1 1

With the sign bit set to 1 for a negative number, the two bytes appear as:



The largest possible values that may be represented in the binary format are +32,767 and –32,768.

1.2.2. Hexadecimal Representation

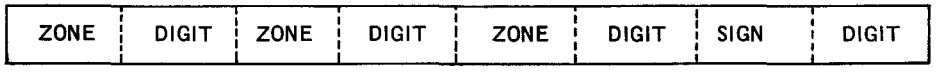
Hexadecimal digits are base 16 numbers with values 0 through F(15). A hexadecimal digit is used to denote a particular bit pattern in the zone or digit portion of a byte representing a decimal number, or alphabetic or symbolic character. (Hexadecimal digits also are used for constant definition as described in Section 2.) The hexadecimal digits and their binary values are:

Hexadecimal Digit	Binary Value	Hexadecimal Digit	Binary Value
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

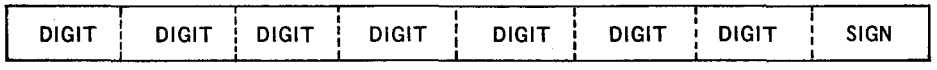
1.2.3. Decimal Number Representation

Decimal numbers are represented in unpacked form (one digit per byte) or packed form (two digits per byte).

In unpacked form, the byte is divided into zone and digit portions. The zone portion usually contains a hexadecimal F bit configuration (1111), which is ignored except in the least significant byte; the zone portion of the least significant byte is interpreted as the sign of the number.



In packed form, digits are contained in both halves of a byte except the least significant half byte of the field, which is interpreted as the sign of the number.



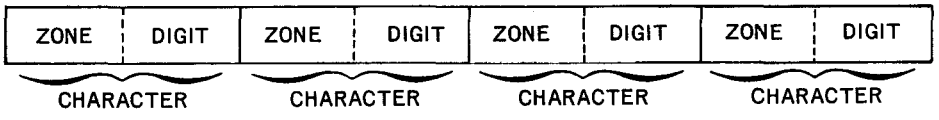
The interpretation of the contents of the sign position is:

Hexadecimal Character	Binary Value	Sign Value
0–8	0000–1000	Positive
9	1001	Negative
A	1010	Positive*
B	1011	Negative*
C	1100	Positive (EBCDIC mode)**
D	1101	Negative (EBCDIC mode)**
E	1110	Positive (EBCDIC mode)
F	1111	Positive (EBCDIC mode)***

* Generated when processor is in the ASCII mode.
** Automatically generated in the central processor for decimal operations.
*** Automatically generated in the central processor for a zone fill during unpack instruction.

1.2.4. Character Representation

An alphabetic or other symbolic character representation is contained in the full eight bits of a byte. A character field is considered as not containing a sign. This type of field is represented:



CHARACTER REPRESENTATION	BINARY VALUE (EBCDIC CODE)
C'D'	11000100
C'GROSS'	1100011111011001110101101110001011100010
C'9'	11111001

1.2.5. Logical Information

Logical information consists of alphabetic or numeric character codes. This information is used in operations such as compare, translate, edit, bit setting, and bit testing. Logical information is processed as fixed- or variable-length data and from left to right, one byte at a time. Variable-length logical information consists of up to 256 bytes.

1.3. STATEMENT CONVENTIONS

The conventions used to illustrate statements in the manual are:

- Capital letters and punctuation marks (except braces, brackets, and ellipses) are information that must be coded exactly as shown.
- Lowercase letters and terms represent information that must be supplied by the programmer.
- Information contained within braces represents necessary entries, one of which must be chosen.
- Information contained within brackets represents optional entries that (depending on program requirements) are included or omitted. Braces within brackets signify that one of the entries must be chosen if that operand is included.
- An ellipsis indicates the presence of a variable number of entries.
- Commas are required after each parameter except after the last parameter specified. When a positional parameter is omitted from a series of parameters, the comma must be retained to indicate the omission.

2. Assembler Language

2.1. CHARACTER SET

The character set used in writing statements in the SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler language consists of:

Letters	A, B, C, . . . ,Z
Digits	0, 1, 2, . . . ,9
Special symbols	/ = & . * + – , ( ) ' blank

The correspondence between graphics and internal code assumed by the assembler is shown in Appendix B.

2.2. STATEMENT FORMAT

Statements in the assembler language are written on a standard coding form, shown in Figure 2–1. Information for the assembler and comments are written in columns 1 through 71. Columns 73 through 80 may contain program identification and sequencing information. The information in columns 1 through 72 consists of the following fields.

2.2.1. Label Field

The label field begins in column 1 and is terminated by a blank column. There may be no embedded blanks. It may be a blank field or contain a symbol of undefined value. More detailed information about symbols is contained in 2.3.4.

2.2.2. Operation Field

The operation field begins with the first nonblank following the label field and is terminated by a blank. It contains the name of an assembler directive, the mnemonic operation code for a machine instruction, or the name of a macro instruction.

2.2.3. Operand Field

The operand field begins with the first nonblank following the operation field and is terminated by a blank not contained in a character representation. This field contains information that defines the operands of a machine instruction or supplies the specifications required with an assembler directive.

SPERRY UNIVAC

ASSEMBLER CODING FORM

UNIVAC 9000 SERIES

PROGRAM _____ PROGRAMMER _____ DATE _____ PAGE ____ OF ____ PAGES

1 LABEL	Δ	OPERATION	Δ 16	OPERAND	Δ	COMMENTS	72	80

Figure 2–1. Assembler Coding Form

2.2.4. Comments Field and Comments Line

The comments field begins with the column following the blank terminating the operand field and ends at column 71. It may contain any combination of characters including blanks. It is not processed by the assembler other than including it on the assembly listing. It may contain remarks to clarify the purpose or operation of the associated coding. A line may consist entirely of comments from columns 2 through 80 if column 1 contains an asterisk. If it is desired to write comments on a line containing a blank operand field, a comma, followed by a space should be placed in the operand field to denote its termination. If this is not done, the first portion of the comments field will be treated as part of the operand field.

Although the assembler language is free form, it is recommended that source code statements be written with the first character of the operation code in column 10 and the first character of the operand field in column 16. Tabulating the statements in this fashion creates a program listing neater in appearance and easier to read. The standard coding form is ruled to conform to this convention. Thus, although the statements on lines 3 and 4 of Figure 2–2 are equivalent to the assembler, the form of line 4 is preferred to that of line 3.

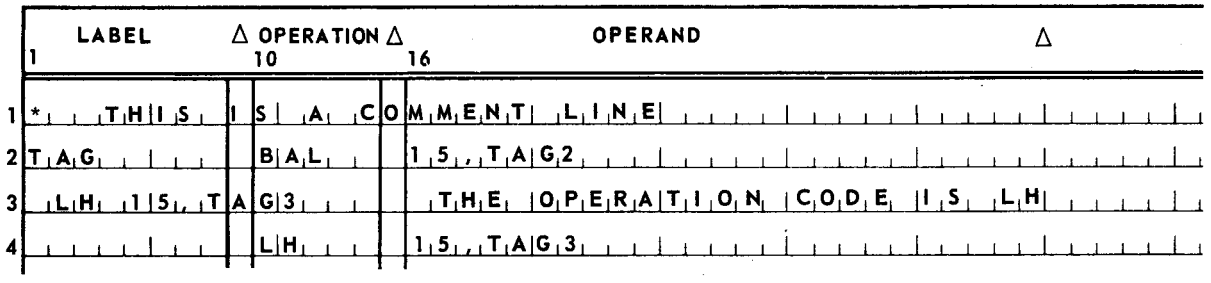


Figure 2–2. Example of Source Code Statements

2.2.5. Continuation Line

If necessary, a statement may occupy more than one line of the coding form. In this case, a nonblank character is placed in column 72 of the first line of the statement, and the statement is continued, beginning with column 16 of the succeeding line of the form. Columns 1 through 15 of the second line must be blank. The statement can be continued on a third line by placing a nonblank character in column 72 of the second line and continuing the statement in column 16 of the following line. Column 72 must be blank in the last (or only) line of each statement.

If the operand field of a line is terminated, prior to column 71, by a comma followed by a space, and a nonblank character appears in column 72 of the line, the operand field is continued in column 16 of the succeeding line. Comments are written after the space terminating the portion of the operand field on the first line.

2.3. EXPRESSIONS, TERMS, AND OPERATORS

The operand field of a statement in the assembler language ordinarily consists of one or more expressions. Expressions are separated by a comma or parenthesis. Examples of the basic operand formats for instructions are shown in Table 3–8. In this table, each letter represents an expression. An expression may be a single term or a number of terms connected by operators. The permissible operators are a plus sign (+) representing addition, a minus sign (–) representing subtraction, an asterisk (*) representing multiplication, and a slash (/) representing division. A leading minus sign also is allowed to produce the negative of the first term. A term may be:

- a decimal, hexadecimal, or character representation of an actual value;
- a location counter reference;
- a symbol;
- a reference to the length attribute of a symbol or of an EXTRN label (with a length attribute of 1); or
- a literal.

2.3.1. Representations of Actual Value

A value may be:

- a value represented directly by a string of up to five digits, forming a binary number from 0 through 65,535, which is converted to a binary value occupying one or two bytes depending on the type of field for which it is intended;
- a value represented by a string of hexadecimal digits identified by a preceding X' and a following ' (apostrophe), each digit occupying half a byte; or
- a value represented by a string of characters identified by a preceding C' and a following ' (apostrophe).

For additional information concerning value representations, see 1.2.

2.3.2. Location Counter

An indication of the next storage location available for assignment is maintained as a counter, called the location counter. After the assembler processes an instruction or constant, it adds the length of the processed instruction or constant to the location counter.

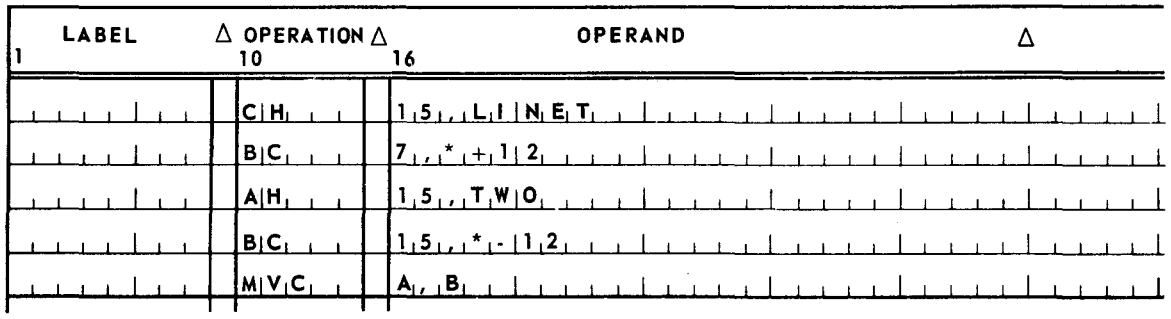
The address of each instruction or address constant must be a multiple of two. Such an address is said to fall on a half-word boundary. If the value of the location counter is not a multiple of two when assembling such a constant or an instruction, a 1 is added to the location counter before assigning an address to the current line. Storage locations reserved by this process receive binary 0's when the program is loaded.

The current value of the location counter is available for reference in the assembler language and is represented by the single special character * (asterisk). If written in a constant representation or in an instruction operand expression, this symbol is replaced by the storage address of the leftmost byte allocated to that instruction or constant. Thus the following instruction represents a one-instruction loop.

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10          16
           BC              1,5,,*
```

2.3.3. Relative Addressing

An instruction may address data in its immediate vicinity in the storage in terms of its own storage address. This is a form of addressing achieved by an expression of the form *+n or *–n where n is the difference in storage addresses of the referring instruction and the instruction or constant being accessed. Relative addressing always is in terms of bytes, not words or instructions. For example, in the coding



the address *+12 in line 2 is the address of the instruction in line 5 and the address *–12 in line 4 is the address of the instruction in line 1 because each of the first four instructions is four bytes long.

Storage addressing is described in 2.5.

2.3.4. Symbols

A symbol is a group of as many as eight alphanumeric characters. The first, or leftmost, must be alphabetic. Special characters or blanks may not be contained within a symbol. The following are examples of valid symbols:

A	LOSS
A72Z	PROFIT
CAT	GRSVALUE

The following are not valid symbols for the reasons stated:

GROSSVALUE	more than eight characters
N PA	embedded blank
SR)N	special character

A symbol may be assigned any value from 0 through 65,535. It is assigned a value, or defined, when it appears in the label field of any source-code statement other than a comment. A symbol appearing in the label field of an EQU or ORG directive is assigned the value of the expression in the operand field. In all other cases, the value assigned is the current value of the location counter after any necessary adjustment to a half-word boundary. The value is assigned to the current label before the location counter is incremented for the next instruction, constant, or storage definition. Thus, if a symbol appears in the label field of a statement defining an instruction, constant, or storage area, the symbol is assigned a value equal to the storage area address of the first byte of that instruction, constant, or storage area.

2.3.5. Relocatable and Absolute Expressions

A single term may be absolute or relocatable. Decimal, character, and hexadecimal representations are absolute terms. In general, a term is absolute unless the expression consists of:

- a location counter reference within a section of relocatable coding;
- a symbol defined by its appearance in the label field of a section of relocatable coding;
- an absolute expression plus a relocatable term; or
- an expression that can be rendered or reduced to an absolute expression plus a relocatable term.

In addition to these relocatable expressions, a negatively relocatable expression is possible. Such an expression consists of an absolute expression minus a relocatable expression, or an expression that may be reordered to that form.

2.3.6. Length Attribute

A length attribute is associated with the operation of most instructions in the repertoire of machine instructions. This length determines the number of bytes to be compared, moved, translated, displayed, or manipulated logically or arithmetically. In the cases of RX and SI format instructions (3.2.1), the operation is performed on fixed-length data and no length specification is needed. For SS1 and SS2 formats, some length specification must be made in the operands to control the amount of data affected by the instruction. If a length is not specified in the operands of SS1 and SS2 format instructions, the assembler supplies a length in accordance with the following principles.

A symbol defined in the label field of a source-code line representing an instruction, constant, or storage allocation is assigned the length associated with the instruction, constant, or storage area involved.

```
LABEL      Δ OPERATION Δ      OPERAND                    Δ
1               10        16
AVR        MVC        TAG,VOR
VOR        DC         C'FOUR'
TAG        DS         CL14
```

In the preceding example, AVR is six bytes long because an MVC is a 6-byte instruction, VOR is four bytes because the constant occupies four bytes, and TAG is 14 bytes because the storage area is defined as such. The instruction labeled AVR moves 14 bytes of data because the implied length of TAG is 14.

The length attribute of an expression is a function of the leading term. If the first term is an absolute value, a length attribute of one byte is assigned. If the first term is a symbol, the length of the symbol is assigned to the expression.

```
RTF        MVC        16+TAG,VOR
           MVC        RTF+16,VOR
```

In the preceding example, the expression 16+TAG is assigned a length of one byte because the leading term is an absolute value, and the expression RTF+16 is assigned a length of six bytes, which is the length of the term RTF.

When a location counter reference appears as the first term of an expression, the expression is assigned a length attribute equal to the length of the instruction or, if the reference to the location counter appears in an EQU statement, the length attribute is one byte.

```
LABEL      Δ OPERATION Δ      OPERAND                    Δ
1               10        16
           MVC     *+12,ABC
DFG        EQU     *
```

In the preceding example, the expression *+12 is assigned a length of six bytes because MVC is a 6-byte instruction, and DFG is assigned a length of one byte because the location counter reference appears in a EQU statement.

The length attribute of an expression may be referenced in assembler coding by the label with the symbol L' preceding it:

L' DFG

is a reference to the length of the expression DFG.

```
SOME       DC      CL10'      FOUR      '
INTR       MVC     STORE(L'SOME),SETS
           UNPK    DEC(L'INTR),PKD(6)
```

In the foregoing example, the move character instruction affects 10 bytes because the length attribute reference was to SOME, which was defined as 10 bytes long. The unpack instruction affects six bytes because the length attribute reference was to INTR, which appeared in the label field of an MVC instruction, a 6-byte instruction.

If no length is specified in an operand that uses the base register and displacement notation, the length attribute assigned to the operand by the assembler is one byte.

```
           CP      6(16,9),78(16,14)
           CP      6(,9),78(,14)
```

In the first line of the example, the length attributed to each operand is 16 bytes, as specified. In the second line, the assembler assigns a length of one byte to each operand.

2.3.7. Evaluation of Expressions and Hierarchy of Operators

The standard mathematical rules of precedence are applied in evaluating expressions; that is, multiplications and divisions are performed before additions and subtractions. Operations of the same precedence are performed from left to right. Parentheses may be used to group terms, overriding the natural precedence of the operators.

The following rules must be observed in writing expressions:

- Two operators may not appear in succession. Write

  A*(–B)

  but not

  A*–B

- A relocatable expression, an expression involving an external reference (4.5.2), or a negative value, may not enter a division. Such a term may be a factor in a multiplication only if the other factor has a value of 0 or 1.

2.4. DATA STORAGE AND CONSTANT FORMATS

The formats for data storage and constant definitions consist of a label field and operation field plus four subfields in the operand field: duplication factor, type length factor, and constant.

Any symbol, as previously defined (2.3.4), may be used in the label field; this use of a symbol is optional.

The operation code is DC for defining constants and DS for defining storage. In either case, the area is assigned the address of the leftmost byte, and the length associated with the data is available to the assembler each time the area is referenced.

2.4.1. Define Constant Statements

The DC statement specifies data that is to be used as stored constants. These constants are produced in object code along with the program instructions. The format of the DC statement is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DC	[d] t [Ln] 'c'

where:

d
Is the optional duplication factor subfield.

t
Is the type subfield.

Ln
Is the optional length factor subfield.

'c'
Is the constant subfield.

2.4.2. Define Storage Statements

The DS statement specifies a storage area to be reserved by the assembler. The format of the DS statement is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DS	[d] t [Ln] ['c']

where:

d
Is the optional duplication factor subfield.

t
Is the type subfield.

Ln
Is the optional length factor subfield.

'c'
Is the optional constant subfield.

The subfields are explained in 2.4.4; however, the following modifications for a DS statement should be noted:

- A constant may be specified in the appropriate subfield of a DS statement, but the constant is not assembled. In H, Y, and S types, the constant is ignored, and in the other types it serves only as a determination of the size of the area defined.
- Storage areas defined by DS statements are not cleared of their contents upon loading.
- Grouping of all DS statements in a program is a good programming practice.

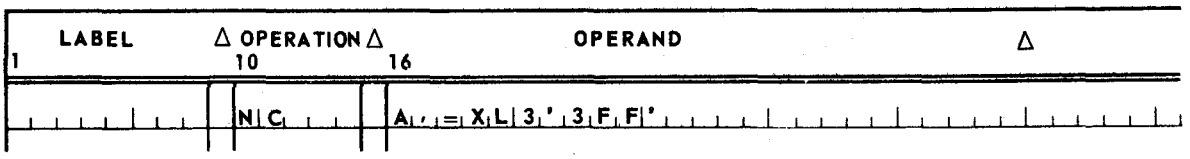
2.4.3. Literals

A constant may be defined explicitly by writing a define constant (DC) assembler directive, or more directly by writing a literal. A literal is written by entering an equal sign (=) in the appropriate portion of the operand field, followed immediately by the description of the constant exactly as it would have appeared in the operand field of a DC directive. The appearance of a literal in a source statement causes the assembler to:

- include the constant in the program the assembler is producing;
- assign an address to the constant; and
- insert the address of the constant in the appropriate portion of the instruction being assembled.

Constants derived from literals are collected into a pool by the assembler. In the absence of other directions to the assembler (4.3.4), this pool is assigned addresses starting at the end of the first control section. Duplicate constants are eliminated from the literal pool on the basis of value, not of form. This is not true for type Y or H constants. Type Y or H constants, which appear to duplicate others, are eliminated only if the forms are identical.

An example of the use of a literal is:



The following rules must be observed in the use of literals:

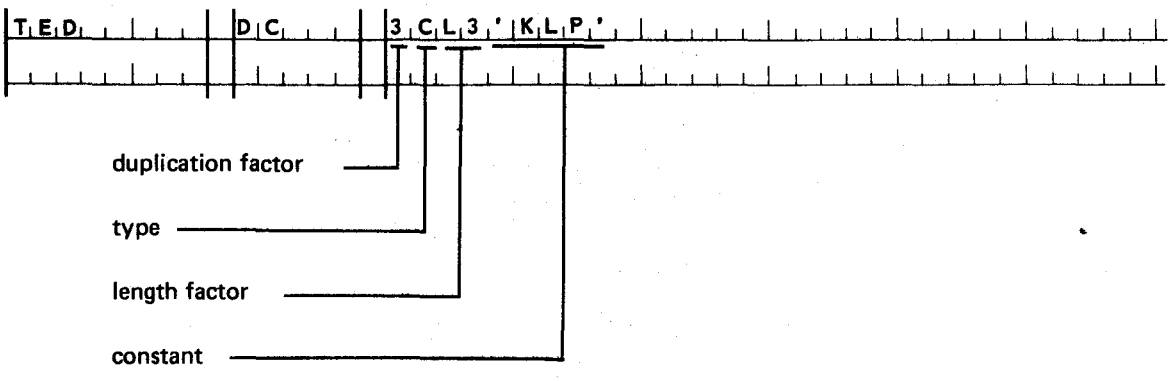
- A literal may appear only as the single term defining a storage address operand of an instruction.
- A literal may not appear in an address field defining the receiving address of an instruction that modifies storage.
- A source-code line can incorporate no more than one literal.
- Reference to the location counter may not be made in a literal.
- Literals may not be used for type S constants.
- A duplication factor of zero is not permitted.

2.4.4. Operand Subfields

The operand field of the DC and DS statements is divided into duplication factor, type, length factor, and constant subfields, which describe or identify the data or storage area to be generated.

The subfields must be specified in the stated order with the duplication factor first and the constant last. Some subfields may be omitted; however, the type subfield always must be present.

The following is a valid example of a typical DC statement with the subfields identified:



2.4.4.1. Duplication Factor Subfield

The duplication factor indicates to the assembler the number of identical constants to be generated. If no duplication factor is specified, the assembler assumes a factor of one. A duplication factor of zero does not generate a constant or storage area, but advances the location counter for proper boundary adjustment if no length is specified and assigns the location counter value to the symbol in the label field.

2.4.4.2. Type Subfield

The type subfield indicates to the assembler the type of constant or storage area to be generated. The types of constants and their characteristics are listed in Table 2–1. For further descriptions of the types, see 2.4.6.

Table 2–1. Constant Characteristics

Constant Type	Explicit Length	Implicit Length	Truncation or Padding	Value Padded	Alignment	Constant Form
C	Variable 1 – 256	Maximum 256	On right side	Blanks	None	Character (EBCDIC)
X	Variable 1 – 256	Maximum 256	On left side	Hexadecimal 0	None	Hexadecimal digits
P	Variable 1 – 16	Maximum 16	On left side	Hexadecimal 0	None	Packed decimal
Z	Variable 1 – 16	Maximum 16	On left side	EBCDIC 0	None	Unpacked decimal
H	Variable 1 – 2	2	On left side	Hexadecimal 0	Half word*	Binary
Y	Variable 1 – 2	2	On left side	Hexadecimal 0	Half word*	Binary address
S	2	2	None	None	Half word*	Base and displacement

*Half-word alignment takes place only if implicit lengths are used.

2.4.4.3. Length Factor Subfield

The length factor subfield designates the number of bytes to be used in generating the data or storage area. The length factor must follow the character 'L' and may be any unsigned decimal within the length limitations of the statement type. The maximum length of any DC statement is 256 bytes, but each type is limited (Table 2–1). The maximum does not apply to DS statements of the C or X type, where the size of main storage associated with the processor is the critical factor.

If no length factor is specified, the size of the generated area is a function of the constant subfield in C, X, P, and Z type statements, and is two bytes for H, Y, and S types (Table 2–1). If the supplied length is less than or more than that needed to express the constant specified in its subfield, truncation or padding occurs as specified in Table 2–1.

Boundary adjustments are performed only on constants with no supplied length factor.

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10        16
EX1       DC              S(32(12))
EX2       DC              CL2'A'
EX3       DC              XL2'FC140'
EX4       DC              CL2'ABC'
```

Each of the foregoing lines of coding produces the following in binary notation:

1100 0001 0100 0000

Each line produces the same constant because of the modification performed by the length specification on the constant originally specified in the coding. In EX1, the address constant is assigned a length of two bytes and translated into binary notation; the base register number occupies the first four bits. In EX2, a character representation two bytes long is specified. To fill the required length, a blank is padded on the right of the constant. The hexadecimal constant, EX3, is too long to fit into the specified length of two bytes. The first four bits are truncated. EX4 specifies a character constant too long for the length specified; three bytes are truncated on the right.

2.4.4.4. Constant Subfield

The constant subfield specifies the value, subject to modification by the length subfield, of the constant to be generated. The values for the constants are represented in different ways. (See 2.4.6 for the description of the values.) A data value representation is specified by enclosing it in apostrophes, and an address value representation is specified by enclosing it in parentheses.

Data constant	Address constant
'constant'	(constant)

2.4.5. Alignment

All machine instructions must be aligned on half-word boundaries such that the address of the first byte of the instruction must be divisible by 2. Constants, however, can be aligned on a half-word, or no boundary at all. The kind of alignment, when necessary, for data or storage definition statements if no length factor is stated is shown in Table 2–1. When a length factor is specified by the programmer, no alignment is provided. A duplication factor of zero in DC and DS statements does not generate a constant or storage area, but, for some types of constants, it forces a boundary alignment if no length factor is coded. This method affords a convenient means of obtaining a boundary alignment before generating a constant not automatically aligned by the assembler. Any bytes skipped to align constants are zero filled; however, bytes skipped to align storage areas are not zero filled.

2.4.6. Data Constant Types

Data constants are absolute values generated by the assembler and require no modification upon loading. Of the five types of data constants, four do not require boundary alignment; that is, the first byte of these constants can be assembled at any storage address. These constants are discussed in 2.4.6.1 through 2.4.6.4; the one data-constant type requiring boundary adjustment is discussed in 2.4.6.5.

Each type of constant is described and examples of its use are shown in typical DC statements.

2.4.6.1. Character Constants

A character constant is specified by the character C in the type subfield and up to 256 characters enclosed by apostrophes in the constant subfield. Any of the 256 valid card punch combinations can be used. Only 63 characters of the 256 valid characters are printable. Each character is stored in one byte using the 8-bit character code. If no length factor is specified, the length in bytes of the constant equals the number of characters specified. If the length factor is present, the character specification is truncated, or filled with blanks, to the right of the last character and to the length specified.

Two consecutive apostrophes or two consecutive ampersands are necessary to generate the character code for one apostrophe or one ampersand within the constant. A single apostrophe in the character representation terminates the constant.

Examples:

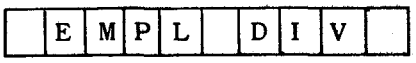
```
     LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10          16
1              DC    CL2'A'
2              DC    C'A'
3              DC    C'A  '
4              DC    CL10'EMPL DIV '
5              DC    CL12'  '
6              DC    C'5063'
7              DC    C'HEADING'
8              DC    3CL4'12345'
9              DC    3CL6'12345'
```

1. Two-byte constant containing:

2. One-byte constant containing:

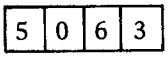
3. Two-byte constant containing:

4. Ten-byte constant containing:



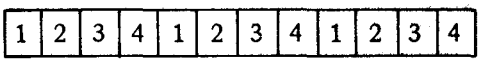
5. Twelve-byte constant containing blanks:

6. Four-byte constant containing:

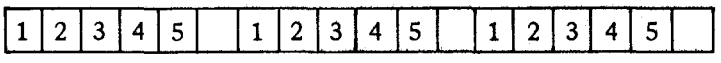


7. Seven-byte constant containing:

8. Twelve-byte constant containing:



9. Eighteen-byte constant containing:



2.4.6.2. Hexadecimal Constants

A hexadecimal constant is specified by the character X in the type subfield and up to 256 hexadecimal digits enclosed by apostrophes in the constant subfield. Two hexadecimal digits are assembled into one byte. If an odd number of digits is specified, the first, or leftmost, byte of the constant contains a hexadecimal 0 in the four leftmost bits and the first digit in the four rightmost bits. If no length factor is specified, the length, in bytes, of the constant is half the sum of the number of digits or 0's specified. If a length factor is present, the decimal specification is truncated, or filled with hexadecimal 0's if necessary, on the leftmost end to the length specified.

Examples:

	LABEL	OPERATION	OPERAND
1		DC	XL3'1'
2		DC	X'12345'
3		DC	X'ABC123D'
4		DC	XL4'FFF'
5		DC	X'FFF000'
6		DC	XL4'A'
7		DC	XL2'12345'

1. Three-byte constant containing:

00000000	00000000	00000001

2. Three-byte constant containing:

00000001	00100011	01000101

3. Four-byte constant containing:

00001010	10111100	00010010	00111101

4. Four-byte constant containing:

00000000	00000000	00001111	11111111

5. Three-byte constant containing:

11111111	11110000	00000000

6. Four-byte constant containing:

00000000	00000000	00000000	00001010

7. Two-byte constant containing:

00100011	01000101

2.4.6.3. Packed Decimal Constants

A packed decimal constant is specified by the character P in the type subfield and decimal digits enclosed by apostrophes in the constant subfield. A leading sign (+ or –) can be coded within the apostrophes. The digits are packed two digits per byte; therefore, each decimal digit requires four bits. In the absence of a sign, a positive sign is assumed. A plus sign is represented by a hexadecimal C and a minus sign is represented by a hexadecimal D. A decimal point can be included in the constant subfield, but is ignored by the assembler.

If no length factor is specified, the length of the constant is the required number of bytes needed to contain the constant, a sign, and a possible addition of zero bits. When an even number of packed decimal digits is specified, the leftmost digit is unpaired because the rightmost digit is paired with the sign. In this case, the most significant four bits of the leftmost byte contain a hexadecimal 0 and the most significant four bits of the least significant (rightmost) byte contain the first (rightmost) digit. The least significant four bits of the rightmost byte always contain the sign of the constant.

If a length factor is present, the decimal specification is truncated, or filled with hexadecimal 0's if necessary, on the leftmost end to the length specified.

Examples:

	LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
1		DC	P'+468'
2		DC	P'-3178'
3		DC	PL2'24.76'
4		DC	PL3'-325'
5		DC	3PL2'381'

1. Two-byte constant containing:

4	6	8	C

2. Three-byte constant, containing:

0	3	1	7	8	D

3. Two-byte constant, with most significant digit truncated, containing:

4	7	6	C

4. Three-byte constant containing:

0	0	3	2	5	D

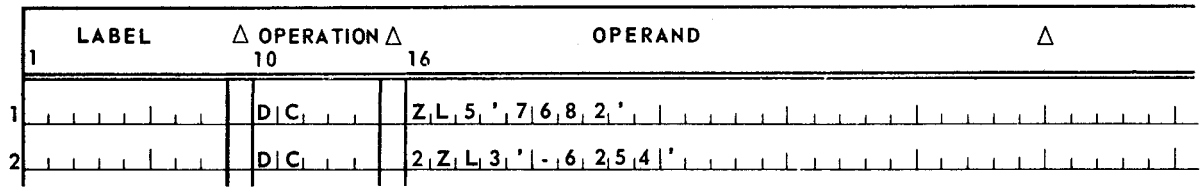
5. Six-byte constant containing:

3	8	1	C	3	8	1	C	3	8	1	C

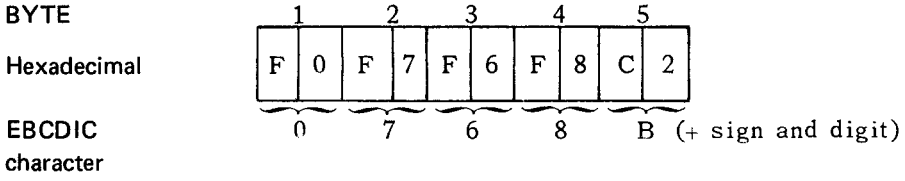
2.4.6.4. Zoned Decimal Constants

A zoned (unpacked) decimal constant is specified by the character Z in the type subfield and decimal digits enclosed by apostrophes in the constant subfield. A plus or minus sign can be coded within the apostrophes; but if none is present, a positive sign is assumed. The digits are assembled, one to a byte, with a hexadecimal F in the most significant four bits of all but the least significant byte. The most significant four bits of the least significant byte contain the sign. If no length factor is specified, the length in bytes of the constant is the number of decimal digits in the constant subfield. If a length factor is present, the decimal specification is truncated, or filled with zoned decimal 0's if necessary, on the leftmost end to the length specified. The rightmost byte always contains the sign and the rightmost digit specified. A plus sign is represented by a hexadecimal C and a minus sign is represented by a hexadecimal D. A decimal point may be included in the constant subfield, but is ignored by the assembler.

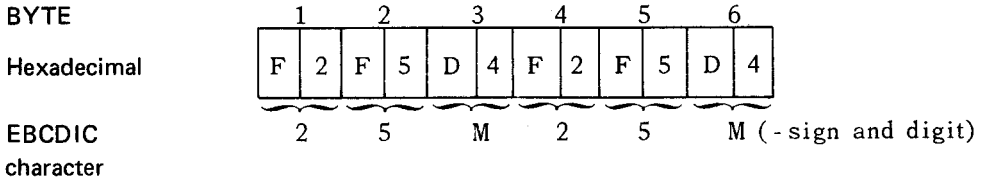
Examples:



1. Five-byte constant containing:



2. Six-byte constant containing:



2.4.6.5. , Half-word Constants

A half-word constant is specified by the character H in the type subfield and up to five significant decimal digits enclosed by apostrophes in the constant subfield. A plus or minus sign can be included within the apostrophes. If no length factor is specified, the constant has an implied length of two bytes and must not contain a value greater than +32,767 or less than −32,768. If the length factor is present, the decimal value specification is truncated, or filled with binary 0's if necessary, on the leftmost end to the length specified. The value specified in the constant subfield may be an integer, a fraction, or a mixed number; however, the fractional portion of the mixed number is lost and the decimal value is converted into a binary format for storage.

Examples:

	LABEL	OPERATION	OPERAND
1		DC	H'27'
2		DC	HL1'27'

1. Two-byte constant half-word aligned containing:

00000000	00011011

2. One-byte constant not half-word aligned containing:

00011011

2.4.7. Address Constant Types

Address constants generate either addresses reflecting the storage locations of the program or values based on these addresses. Address constants often are used to load general registers or reference external addresses. The location counter is adjusted for each constant and any duplication factor used.

2.4.7.1. Half-word Address Constants

A half-word address constant is written by specifying the character Y in the type subfield and an expression enclosed by parentheses in the constant subfield. The expression may be either absolute or relocatable.

If no length factor is specified in the operand, the half-word address constant generates a 2-byte constant containing the value of the expression in the constant subfield. The generated constant is aligned at a half-word boundary. If a length factor of 2 is specified, the generated constant is the same, but no boundary alignment takes place. If a length factor of 1 is specified, a 1-byte value is generated without any boundary alignment.

Examples:

	LABEL	OPERATION	OPERAND
1		DC	Y(BOB)
2		DC	YL2(BOB)
3		DC	2YL1(6)
4		DC	2YL1(BOB)

The preceding statements generate the following constants. (In each example, it is assumed that BOB has a value of 1446. This value may represent an main storage address or may be a binary value used for arithmetic purposes. The hexadecimal value equal to 1446 is stored as shown.)

1. Two-byte constant, aligned on a half-word boundary, containing:

0	5	A	6

2. Two-byte constant, unaligned, containing:

0	5	A	6

3. Two-byte field containing the 1-byte constant duplicated once as:

0	6	0	6

4. Two-byte field containing the truncated value of BOB duplicated once, as:

A	6	A	6

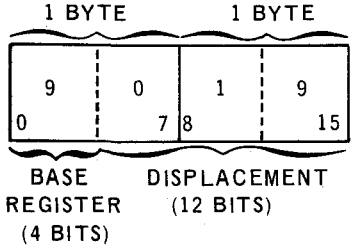
2.4.7.2. Base and Displacement Constants

A base and displacement constant is specified by the character S in the type subfield and one or two expressions enclosed by parentheses in the constant subfield. The expressions may be absolute or relocatable. A length factor, if present, can be only 2. If no length factor is specified, the implied length of the constant is two bytes. Negative relocatable values are not permitted. This type of constant is used to store addresses in the base and displacement form; the leftmost four bits represent the base and the remaining 12 bits represent the displacement. If one expression is present, the assembler converts it to a base plus displacement value. If two expressions are present, the expression representing the base is enclosed in parentheses with the other expression (representing the displacement) preceding it and another set of parentheses enclosing the base and displacement specifications. The S-type constants may not be specified as literals. Adjustment to the half-word boundary is made only if no length specification is used.

Examples:

```
      LABEL      Δ OPERATION Δ      OPERAND                    Δ
      1          10              16
1     START      4096
2     USING      5000,9
3     JOHN
4     SYMBOL     DC      S(JOHN)
5     SYMBOL     DC      S(25(9))
```

Assume that the constant with the label JOHN (line 3) is assigned an address value of 5025_{10} by the location counter, and that the USING directive (line 2) gives the value 5000_{10}, which is assumed to be in register 9 at execution time. The operands in the two statements (lines 4 and 5) produce the same stored base and displacement value. The hexadecimal representation of this stored value is 9019 as follows:



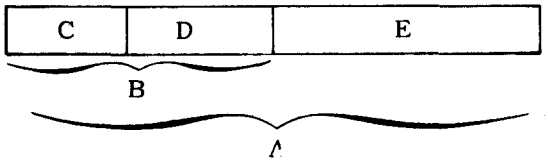
NOTE:

Refer to 4.4 for a description of the USING directive.

2.4.8. Data Storage Definition Examples

The programmer can reference a data storage area in many ways. By defining an area in terms of its smallest divisions or fields, each field can be referenced. The programmer can then use a DS statement with a 0-duplication factor to define a label and give it a length attribute so that more than one field can be referenced simultaneously.

The following example illustrates a typical data storage area and the subdivision of its fields:



where:

A
Is a 40-byte field.

B
Is a 20-byte field.

C
Is a 10-byte field.

D
Is a 10-byte field.

E
Is a 20-byte field.

The following assembler coding example illustrates the ways DS instructions can be used to assign labels and length attributes to the fields of a data storage area:

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10          16
A              DS          0CL40
B              DS          0CL20
C              DS          CL10
D              DS          CL10
E              DS          CL20
```

The first statement of the example does not reserve any storage but does define the label A and its length attribute of 40. Although the label of the second statement has the same value as the label of the first statement, the length attributes of the first and second statements are different. When using the labels in an SS-type instruction with implied lengths, one label (A) would reference 40 bytes, but the other label (B) would reference 20 bytes.

2.5. STORAGE ADDRESSING

Both direct addressing and indirect addressing may be used in the SPERRY UNIVAC 9200/9300 Series. Direct addressing is possible because the operand portion of an instruction is large enough to represent 32,767, the maximum main storage address available in the system. Indexed, or base and displacement, addressing also is possible.

Indexed addressing is achieved by using part of the operand for an indexing indication and the remainder for a displacement value. Either actual index registers or pseudo registers may be used for the indexing indication. Using pseudo registers results in direct addressing that provides a relocatable code.

2.5.1. Base and Displacement Addressing

Inasmuch as the 9200/9300 series instruction format does not include an index register field, the first four bits of the operand itself are set aside for the index or register number. The most significant bit of each operand determines whether indexing is to occur. If that bit is a 1, indexing takes place; otherwise, no indexing is performed.

Because 4 of the 16 bits of the operand are used for indexing, only 12 bits remain for the operand address. This partial operand is the displacement. Since the largest address that can be expressed in 12 bits is 4095, main storage is segmented into 4K blocks.

The registers used for indexing are the processor general registers. They are numbered from 8 through 15. Because each of these numbers begins with a 1 when represented in hexadecimal notation, specifying a register forces indexing. The true main storage address consists of the contents of the register plus the value of the displacement.

The USING assembler directive is used to notify the assembler that a specified register is available for base register assignment. The directive is explained in 4.4. The register must be loaded by the program with the value to be used. From the values supplied in the various USING directives, the assembler builds a base register table which it uses to compute the 12-bit displacements for operands. The computation of displacements continues to be based on the base register table until the displacement limit of 4095 is reached or until the assembler finds another register which was designated in a USING directive with a base address that yields a smaller displacement.

The SPERRY UNIVAC 9200/9300 Series Operating System locator/loader routine places the transfer address of a main program in processor register 15 prior to transferring control to the program. The programmer may take advantage of this fact by coding a USING directive containing the label of the first instruction to be executed and specifying register 15 as the base register, as:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16)	Δ
	START	6000	
	USING	BEGN,15	
BEGN	LH	8,LABL	
	USING	GO,8	
GO	MVC	OUT,TWO	
	.		
	.		
	.		
OUT	DS	H	
TWO	DC	H'2'	
LABL	DC	Y(GO)	
	END	BEGN	

In this example, since register 15 was the base register in force at the time, the machine code generated at BEGN might appear as:

4880FE24

Since the following line declared the availability of register 8 as a base register and since register 8 will now yield a smaller displacement than register 15 for any operands which address locations following GO, the machine code generated at GO might appear as:

D2018E1C8E1E

2.5.2. Pseudo Register Base and Displacement Addressing

True base and displacement addressing requires index register modification of each instruction. Direct addressing is more desirable because it eliminates the extra cycles required to index each instruction, speeding up instruction execution. Direct addressing can be achieved by assembling the program with absolute addresses by using an ORG directive. With this method, however, the program cannot be relocated.

Another type of addressing provides direct addressing and relocatable code, but does not require indexing of each instruction. This form of addressing uses pseudo registers. To designate a pseudo register, the USING directive format is:

LABEL	Δ OPERATION Δ	OPERAND
	USING	*,n

where:

n

Is the pseudo register number, from 0 through 7.

As in true base and displacement addressing, the pseudo register USING directive provides a value for the base register table. But in this case, the values are not arbitrarily provided as a result of the label processing by the assembler; they are fixed values determined by the register number given in the USING directive. The register numbers 0 through 7 produce the table values:

register 0 – 0

register 1 – 4096

register 2 – 8192

register 3 – 12,288

register 4 – 16,384

register 5 – 20,480

register 6 – 24,576

register 7 – 28,672

When using pseudo registers, the operand displacement is determined in the same way as with actual registers. The displacement for an operand is computed by subtracting the table value that produces the smallest 12-bit displacement from the label value. For instance, if TAG were at address 4100, this value would be shown as:

BASE DISPLACEMENT

0001 000000000100

Viewed as a base and displacement address, this operand shows a base register of 1 and displacement of 4. (Remember that pseudo register 1 is the value of 4096.) The 16-bit address also is the binary representation of 4100: 0001000000000100, the actual address of TAG.

Since the pseudo register numbers range from 0 through 7, the most significant bit is zero. The operand, then, does not cause indexing and, when read as a binary number, is a direct address.

True base and displacement addressing is advantageous only when the programmer must be concerned with later conversion to a processor with a main storage capacity exceeding 32K. Even then the use of pseudo registers may be preferable because the conversion requires reassembling the program, and the USING directives can be changed easily at that time.

3. Instructions

3.1. GENERAL

The SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler uses a repertoire of 35 machine instructions, which are described in this section. The instructions are listed in Tables 3–1, 3–2, 3–3, and 3–4.

NOTE:

The 9200 system is not provided with the divide packed decimal, multiply packed decimal, and edit instructions, except as optional features. If the optional hardware instructions are not installed, they may be simulated by UNIVAC-supplied fixed, closed subroutines, which duplicate the functions of the instructions. The use of the optional hardware instructions does not differ from that of the standard 9300 system instructions. The use of the software subroutines is described in the operational conditions of the instructions (3.3.1.5, 3.3.1.6, and 3.3.4.1).

The instructions are grouped by function in this section, as in Table 3–2. The function types are arithmetic, branch, comparison, data manipulation, data transfer, display, input/output, logical, and supervisor. Each type is analyzed and explained separately. The timing for the instructions is presented in Table 3–4. The symbols used in illustrations in this section are listed in Table 3–5.

3.2. INSTRUCTION FORMAT

The machine instruction format consists of an optional label, a mnemonic operation code, and one or two operands.

LABEL	△OPERATION△	OPERAND
[symbol]	code	operand 1, operand 2

The label may be any symbol, as previously defined, but its use is not required. The mnemonic operation code for each instruction is specified in Table 3–1 and in the text explaining each instruction. The operands are of two forms, expressions, as previously defined, or complete specifications. Any symbol used in an expression must be defined within the program. A complete specification includes a base register and a displacement value. The specification is explicit and the assembler uses these numbers to compile the addresses of the operands. Each type of operand is illustrated in the example format:

LABEL	△OPERATION△	OPERAND
[symbol]	code	{ D1(L1,B1), D2(L2,B2) } { expression, expression }

Table 3–1. Instruction Mnemonics

Mnemonic	Function	Hexadecimal Operation Code	Format
AH	Add half word	AA	RX
AI	Add immediate	A6	SI
AP	Add (packed) decimal	FA	SS2
BAL	Branch and link	45	RX
BC	Branch on condition	47	RX
CH	Compare half word	49	RX
CLC	Compare logical character	D5	SS1
CLI	Compare logical immediate	95	SI
CP	Compare (packed) decimal	F9	SS2
DP	Divide (packed) decimal	FD	SS2
ED	Edit	DE	SS1
HPR	Halt and proceed	A9	SI
LH	Load half word	48	RX
LPSC	Load program state control	A8	SI
MP	Multiply (packed) decimal	FC	SS2
MVC	Move characters	D2	SS1
MVI	Move immediate data	92	SI
MVN	Move numerics	D1	SS1
MVO	Move with offset	F1	SS2
NC	*AND* characters	D4	SS1
NI	*AND* immediate data	94	SI
OC	*OR* characters	D6	SS1
OI	OR immediate data	96	SI
PACK	pack	F2	SS2
SH	Subtract half word	AB	RX
SP	Subtract (packed)decimal	FB	SS2
SPSC	Store program state control	A0	SI
SRC	Supervisor request call	A1	SI
STH	Store half word	40	RX
TIO	Test I/O	A5	SI
TM	Test under mask	91	SI
TR	Translate	DC	SS1
UNPK	Unpack	F3	SS2
XIOF	Execute input/output function	A4	SI
ZAP	Zero add (packed) decimal	F8	SS2

Table 3–2. Instruction Types

Type	Mnemonic	Function	Hexadecimal Operation Code	Format
Arithmetic	AH	Add half word	AA	RX
	AI	Add immediate	A6	SI
	AP	Add (packed) decimal	FA	SS2
	DP	Divide (packed) decimal	FD	SS2
	MP	Multiply (packed) decimal	FC	SS2
	SH	Subtract half word	AB	RX
	SP	Subtract (packed) decimal	FB	SS2
	ZAP	Zero add (packed) decimal	F8	SS2
Branch	BAL	Branch and link	45	RX
	BC	Branch on condition	47	RX
Comparison	CH	Compare half word	49	RX
	CLC	Compare logical character	D5	SS1
	CLI	Compare logical immediate	95	SI
	CP	Compare (packed) decimal	F9	SS2
	TM	Test under mask	91	SI
Data manipulation	ED	Edit	DE	SS1
	PACK	Pack	F2	SS2
	TR	Translate	DC	SS1
	UNPK	Unpack	F3	SS2
Data transfer	LH	Load half word	48	RX
	MVC	Move characters	D2	SS1
	MVI	Move immediate data	92	SI
	MVN	Move numerics	D1	SS1
	MVO	Move with offset	F1	SS2
	STH	Store half word	40	RX
Display	HPR	Halt and proceed	A9	SI
I/O	TIO	Test I/O	A5	SI
	XIOF	Execute input/output function	A4	SI
Logical	NC	*AND* characters	D4	SS1
	NI	*AND* immediate data	94	SI
	OC	*OR* characters	D6	SS1
	OI	*OR* immediate data	96	SI
Supervisor	LPSC	Load program state control	A8	SI
	SPSC	Store program state control	A0	SI
	SRC	Supervisor request call	A1	SI

Table 3–3. Instruction Formats

Format	Hexadecimal Operation Code	Mnemonic	Function
RX	40	STH	Store half word
RX	45	BAL	Branch and link
RX	47	BC	Branch on condition
RX	48	LH	Load half word
RX	49	CH	Compare half word
RX	AA	AH	Add half word
RX	AB	SH	Subtract half word
SI	91	TM	Test under mask
SI	92	MVI	Move immediate data
SI	94	NI	*AND* immediate data
SI	95	CLI	Compare logical immediate
SI	96	OI	*OR* immediate data
SI	A0	SPSC	Store program state control
SI	A1	SRC	Supervisor request call
SI	A4	XIOF	Execute input/output function
SI	A5	TIO	Test I/O
SI	A6	AI	Add immediate
SI	A8	LPSC	Load program state control
SI	A9	HPR	Halt and proceed
SS1	D1	MVN	Move numerics
SS1	D2	MVC	Move characters
SS1	D4	NC	*AND* characters
SS1	D5	CLC	Compare logical character
SS1	D6	OC	*OR* characters
SS1	DC	TR	Translate
SS1	DE	ED	Edit
SS2	F1	MVO	Move with offset
SS2	F2	PACK	Pack
SS2	F3	UNPK	Unpack
SS2	F8	ZAP	Zero add (packed) decimal
SS2	F9	CP	Compare (packed) decimal
SS2	FA	AP	Add (packed) decimal
SS2	FB	SP	Subtract (packed) decimal
SS2	FC	MP	Multiply (packed) decimal
SS2	FD	DP	Divide (packed) decimal

Table 3–4. Instruction Execution Times (Part 1 of 2)

Hexadecimal Operation Code	Mnemonic	Function	Format	Times (microseconds)①
40	STH	Store half word	RX	20.4
45	BAL	Branch and link	RX	18.0
47	BC	Branch on condition	RX	15.6 if no branch 18.0 if no branch
48	LH	Load half word	RX	20.4
49	CH	Compare half word	RX	20.4
91	TM	Test under mask	SI	16.8 if no match or match on zero 19.2 if partial or full match
92	MVI	Move immediate	SI	16.8
94	NI	*AND* immediate data	SI	16.8
95	CLI	Compare logical immediate	SI	16.8
96	OI	*OR* immediate data	SI	16.8
A0	SPSC	Store program state control	SI	24.0
A1	SRC	Supervisor request call	SI	12.0
A4	XIOF	Execute I/O function	SI	18.0 for integrated I/O units; variable for multiplexer
A5	TIO	Test I/O status	SI	18.0 for integrated I/O units; variable for multiplexer
A6	AI	Add immediate	SI	19.2
A8	LPSC	Load program state control	SI	24.0 to load entire PSC word; 18.0 otherwise
A9	HPR	Halt and proceed	SI	14.4
AA	AH	Add half word	RX	20.4
AB	SH	Subtract half word	RX	20.4
D1	MVN	Move numerics	SS1	$16.8 + 8.4(N)$②
D2	MVC	Move character	SS1	$16.8 + 8.4(N)$
D4	NC	*AND* characters	SS1	$16.8 + 8.4(N)$
D5	CLC	Compare logical character	SS1	$25.2 + 8.4(N_E)$③
D6	OC	*OR* characters	SS1	$16.8 + 8.4(N)$
DC	TR	Translate	SS1	$16.8 + 14.4(N)$
DE	ED	Edit	SS1	See 4
F1	MVO	Move with offset	SS2	$25.2 + 3.6(N_2) + 6(N_1)$
F2	PACK	Pack	SS2	$25.2 + 3.6(N_2) + 4.8(N_1)$
F3	UNPK	Unpack	SS2	$21.6 + 7.2(N_2) + 4.8(N_1)$
F8	ZAP	Zero and add	SS2	$26.4 + 3.6(N_2) + 4.8(N_1)$
F9	CP	Compare (packed) decimal	SS2	$26.4 + 3.6(N_2) + 4.8(N_1)$
FA	AP	Add (packed) decimal	SS2	$26.4 + 3.6(N_2) + 4.8(N_1)$
FB	SP	Subtract (packed) decimal	SS2	$26.4 + 3.6(N_2) + 4.8(N_1)$
FC	MP	Multiply (packed) decimal	SS2	See④
FD	DP	Divide (packed) decimal	SS2	See④

NOTES:

① Timing for all instructions assumes no indexing. Add 3.6 microseconds for each indexing operation. Timing is given for 9300 systems; for 9200 systems, multiply by two.

② N, N_1, and N_2 equal the number of bytes specified in the length of the operand.

③ N_E equals the number of most significant bytes that compare identically between OP1 and OP2 in the compare logical character instruction.

(4) Detailed formulas for the edit, multiply, and divide instructions are:

- Edit (ED)

$$30 + 6S_2 + 14N + 6(N_{DS} + N_{SS}) + 6I_2 + 6I_1$$

- Multiply (packed) decimal (MP)

$$52 + 13\,(N_1 - N_2) + (16 + 14N_2)\sum MMD + (22 + 14N_2)$$

$$\sum LMD + 24\,(NMMD0) + 6I_2 + 6I_1 + 14$$

$$\left(\sum\left[\left(\frac{|OP2|}{10^{2N_2-1}}\right)\ MMD\right]\ \begin{array}{l}\text{largest}\\ \text{integer}\\ \text{in}\end{array}\ \pm\ \frac{(N_1 + N_2)}{2}\right)$$

This term gives an upper and lower limit (note the ± sign) for a specific MP instruction using the actual values of OP1 and OP2. A series of MP instructions executed with random numbers for operands yielded an average execution time of 28 $(N_1 - N_2)$.

Table 3–6 consists of a hardware timing chart that can be used to approximate the times required for multiplication operations.

- Divide (Packed) decimal (DP)

$$56 + 8N_2 + 6N_1 + (30 + 14N_2)\sum(MQD + 1) + (22 + 14N_2)\sum\overline{LQD} + 6I_2 + 6I_1$$

where:

N_{DS}	=	the number of digit select bytes in OP1 of edit instruction.
N_{SS}	=	the number of significant start bytes in OP1 of edit instruction.
S_2	=	the total number of signs in all the bytes accessed for OP2 of edit instruction.
MMD	=	the value of a nonzero multiplier (OP1) digit positioned in the most significant half of a byte.
NMMD0	=	the number of zero multiplier (OP1) digits positioned in the most significant half of a byte. The N_2 most significant bytes of OP1 are excluded.
LMD	=	the value of a multiplier digit positioned in the least significant half of a byte.
MQD	=	the value of a quotient digit positioned in the most significant half of a byte.
$\overline{LQD}$	=	the tens complement of quotient digits positioned in the least significant half of a byte.
'OP2'	=	multiplicand digits
I_1	=	0 if operand 1 is not indexed; 1 if indexed.
I_2	=	0 if operand 2 is not indexed; 1 if indexed.

Table 3–5. Instruction Symbols

Symbol	Meaning
B1	The number of the general register that holds the base address of operand 1.
B2	The number of the general register that holds the base address of operand 2.
code	The mnemonic operation code of the instruction.
D1	The displacement from the base address of operand 1.
D2	The displacement from the base address of operand 2.
I2	The immediate data used as operand 2 in SI format instructions.
L1	The length of operand 1 as stated in source code.*
L2	The length of operand 2 as stated in source code.*
OP1	Operand 1.
OP2	Operand 2.
R1	The number of the general register that holds operand 1 in RX format instructions.
symbol	The expression or symbolic label used as operand 1.
tag	The expression or symbolic label used as operand 2.

* The length is coded as the true length of the operand, not the length less 1 as required by the object code. The assembler makes the appropriate reduction by 1 when converting source code to object code.

Table 3–6. Hardware Multiply Timing Chart

Multiplier Digits (OP1) \ Multiplicand Digits (OP2)	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	.75	.75	.80	.80	.85	.85	.90	.90	.95	.95	1.00	1.00	1.04	1.04
3	.99	.99	1.04	1.04	1.08	1.08	1.14	1.14	1.19	1.19	1.23	1.23	1.27	1.27
4	1.65	1.65	1.70	1.70	1.75	1.75	1.80	1.80	1.85	1.85	1.90	1.90	1.95	1.95
5	1.97	1.97	2.02	2.02	2.07	2.07	2.12	2.12	2.17	22.17	2.22	2.22	2.27	2.27
6	2.89	2.89	2.94	2.94	3.00	3.00	3.04	3.04	3.09	3.09	3.14	3.14	3.19	3.19
7	3.29	3.29	3.34	3.34	3.39	3.39	3.44	3.44	3.49	3.49	3.54	3.54	3.59	3.59
8	4.46	4.46	4.51	4.51	4.56	4.56	4.61	4.61	4.66	4.66	4.72	4.72	4.76	4.76
9	4.96	4.96	5.00	5.00	5.05	5.05	5.10	5.10	5.15	5.15	5.19	5.19	5.25	5.25
10	6.37	6.37	6.42	6.42	6.47	6.47	6.52	6.52	6.57	6.57	6.62	6.62	6.68	6.68
11	6.94	6.94	6.99	6.99	7.04	7.04	7.09	7.09	7.14	7.14	7.19	7.19	7.25	7.25
12	8.61	8.61	8.66	8.66	8.71	8.71	8.76	8.76	8.81	8.81	8.86	8.86	8.92	8.92
13	9.27	9.27	9.32	9.32	9.37	9.37	9.42	9.42	9.47	9.47	9.51	9.51	9.58	9.58
14	11.19	11.19	11.24	11.24	11.29	11.29	11.34	11.34	11.39	11.39	11.44	11.44	11.50	11.50
15	11.93	11.93	11.98	11.98	12.03	12.03	12.08	12.08	12.13	12.13	12.18	12.18	12.24	12.24

TIME IN MILLISECONDS

NOTE:

The times are averages based on a multiplier consisting of fives. To find 9300 system times, use half the figure in the chart.

3.2.1. Source Code Instruction Format

Four instruction format types, the RX, SI, SS1, and SS2 are available. The type of data field processed and the type of processing determine the format of the instruction. The RX format is for instructions that process data in fixed lengths and usually involves the use of a general register. The SI format is for instructions that process data immediately specified as one of the operands. SS1 and SS2 formats are for instructions that process data in variable lengths; SS1 is used when the operands are of equal length, and SS2 is used when they are determined independently.

The four instruction format types are shown in Table 3–7.

3.2.1.1. Register and Indexed Storage Operation (RX)

The RX format is used by 4-byte instructions with the form:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	code	{ R1, D2(,B2) } { R1, tag }

The RX format instructions are used to process fixed-length data fields with a length of two bytes. One operand usually specifies a general register. Functions such as branching, comparing, adding, storing, and loading are performed by instructions in this format.

3.2.1.2. Storage and Immediate Operand Operation (SI)

The SI format is used by 4-byte instructions with the form:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	code	{ D1(B1),I2 } { symbol,I2 }

The SI format instructions are used to process data one byte in length, using control or additional data contained in the immediate operand. Logical, arithmetic, manipulative, and testing functions are performed by instructions in this format.

3.2.1.3. Storage to Storage Operation (SS1)

The SS1 format is used by 6-byte instructions with the form:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	code	{ D1(L1,B1),D2(B2) } { symbol,tag }

The SS1 format instructions process data of variable length, up to a maximum of 256 bytes, if the operands specify fields of equal length. Functions such as comparing, transferring, and translating are performed by instructions in this format.

3.2.1.4. Storage to Storage Operation (SS2)

The SS2 format is used by 6-byte instructions with the form:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	code	{ D1(L1,B1),D2(L2,B2) } { symbol,tag }

The SS2 format instructions process data of variable-length operands, to a maximum of 16 bytes, when the operands are not of equal length. The SS2 format is used with all packed decimal instructions. Functions such as shift operations, pack, and unpack are performed by instructions in this format.

3.2.2. Object Code Instruction Format

The format of the instruction repertoire in object code differs from source-code format. The hexadecimal object-code format is used in the printed listing that accompanies an assembly; the main storage dump also uses the object-code format. This format is illustrated in Table 3–7.

An example of an assembly printout is shown in Figure 3–1.

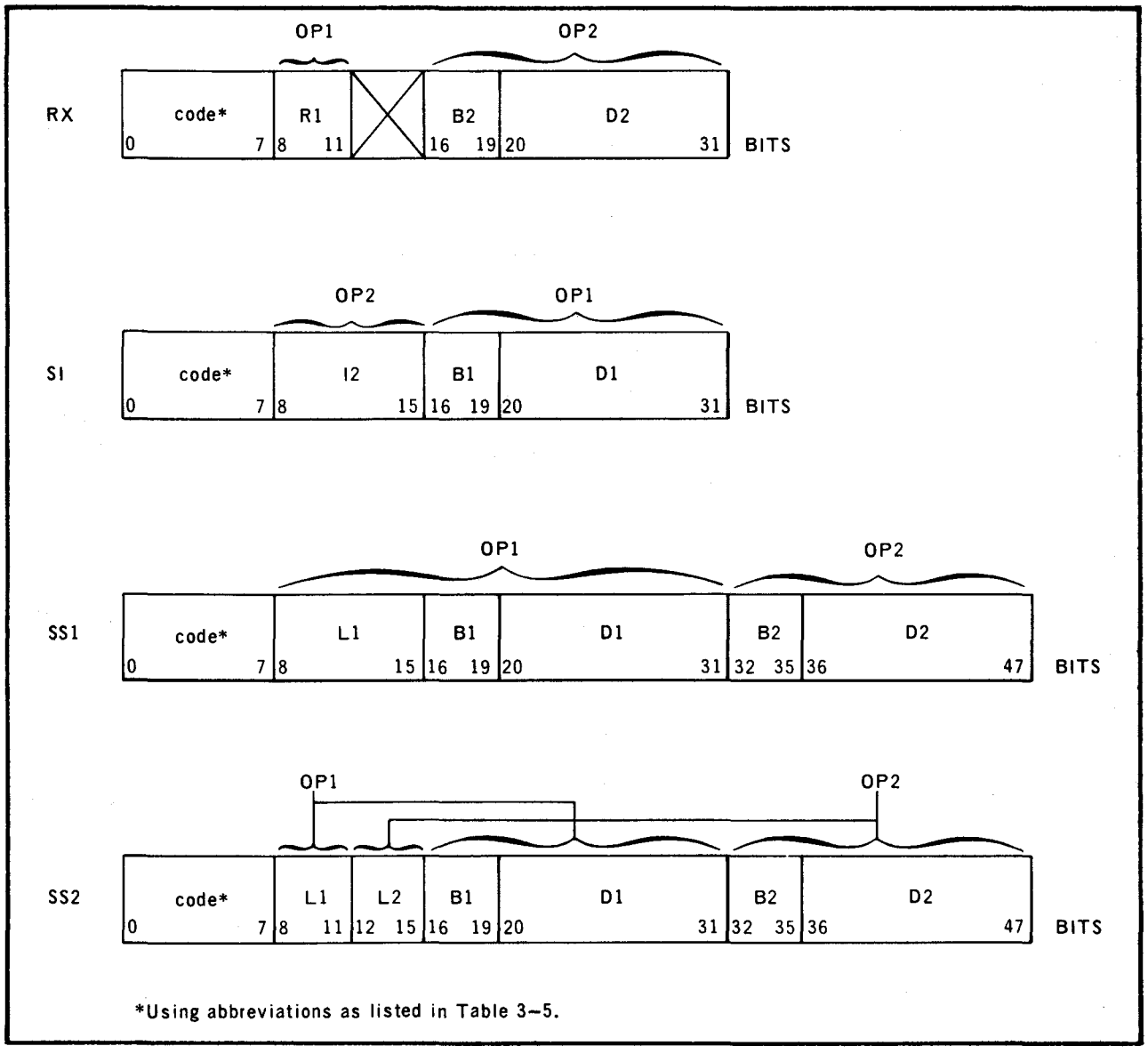
The object code in the third column of Figure 3–1 for the first instruction in the illustration is ABA0B000. The interpretation of the object code is:

AB	subtract half-word operation code
A	OP1 register
0	not used
B	register holding OP2 base address
000	OP2 displacement

The object code of the second instruction (line number 004) in Figure 3–1 is interpreted as:

96	OR immediate operation code
FB	immediate data in OP2
9	register holding OP1 base address
011	OP1 displacement

Table 3–7. Instruction Object-Code Formats



***Using abbreviations listed in Table 3–5.**

LINE NUMBER	ADDRESS	OBJECT OPERATION CODE	LABEL	SOURCE OPERATION CODE	OPERAND	COMMENTS
9300 SYSTEM ASSEMBLY OF STST					DATE 04/15/73	PAGE 001
0001	2000		STST	START	8192	
0002				USING	*,2	
0003	2000	ABA0B000	RX	SH	10,0(,11)	
0004	2004	96FB9011	SI	OI	17(9),X'FB'	
0005	2008	D120C005D00B	SS1	MVN	5(33,12),11(13)	
0006	200E	F8DD8047F056	SS2	ZAP	71(14,8),86(14,15)	
0007	2014	000000002000		END	STST	

Figure 3–1. Assembly Listing

The object code of the third instruction (line number 005) in Figure 3–1 is interpreted as:

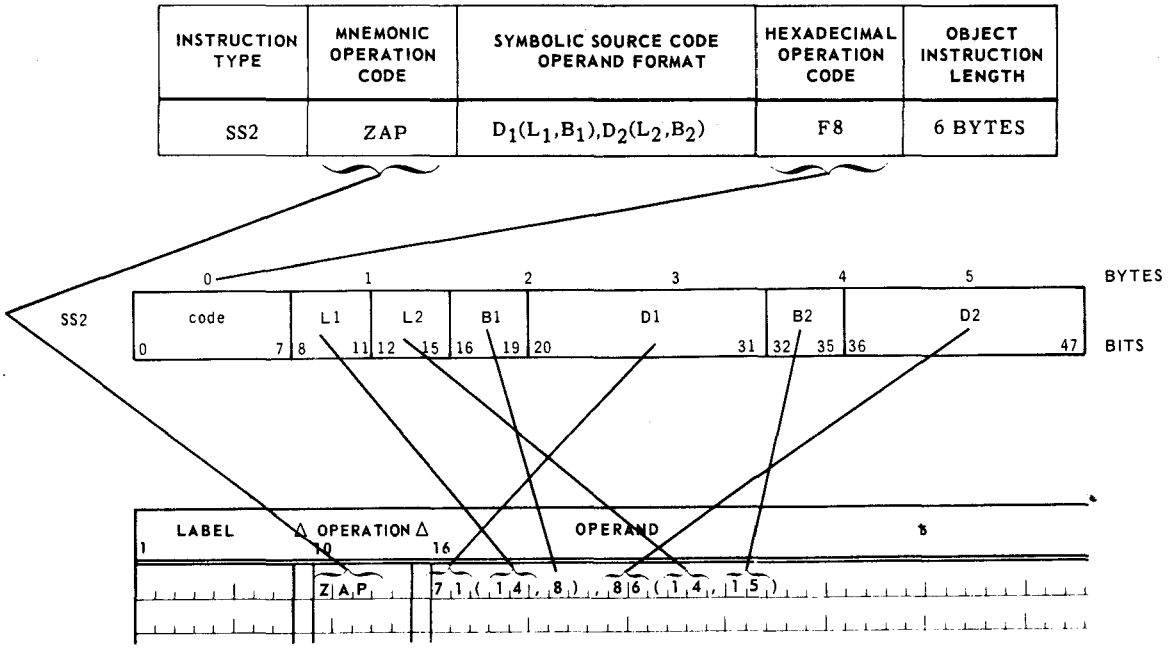
D1	move numeric operation code
20	length minus one of the operands
C	register holding OP1 base address
005	OP1 displacement
D	register holding OP2 base address
00B	OP2 displacement

The object code for the fourth instruction (line number 006) in Figure 3–1 is interpreted as:

F8	zero and add operation code
D	length minus one of OP1
D	length minus one of OP2
8	register holding base address of OP1
047	OP1 displacement
F	register holding base address of OP2
056	OP2 displacement

To facilitate understanding of the object-code form of the instructions, each instruction is accompanied by a small box containing information pertinent to the interpretation of the object-code instruction format. Each instruction type, RX, SI, SS1, and SS2 is in its own object code format, as illustrated in Table 3–7.

The relationship between source- and object-code formats is illustrated for the zero and add packed decimal instruction (3.3.1.9):



The box that accompanies each instruction provides the programmer: the hexadecimal operation code, as F8 for ZAP; the format type, SS2 in the example; the number of bytes the instruction will occupy in main storage, six bytes, in the example; and the mnemonic operation code, ZAP.

3.2.3. Implied Base Register and Length

The complete specification of an operand consists of a displacement and base register to form the address, and a length to determine the size of the operands. If the address is in implied form. The assembler then provides the addresses of the operands. Information supplied in the USING and DROP assembler directives enables the assembler to do this. The length attribute associated with the expression is determined by the assembler according to the rules explained in 2.3.6. The complete and implied form for each instruction format type is shown in Table 3–8.

Table 3–8. Complete and Implied Specifications for Operands

Instruction Type	Specification Type	Operand	
		OP1	OP2
RX	Complete	R1	,D2(B2)
	Relative address	R1	,tag
SI	Complete	D1(B1)	,I2
	Relative address	symbol	,I2
SS1	Complete	D1(L1,B1)	,D2(B2)
	Relative address	symbol(L1)	,tag
	Implied length	D1(,B1)	,D2(B2)
	Relative address and length	symbol	,tag
SS2	Complete	D1(L1,B1)	,D2(L2,B2)
	Relative address	symbol(L1)	,tag(L2)
	Implied length	D1(,B1)	,D2(,B2)
	Relative address and length	symbol	,tag

NOTE:

Symbol and tag can be any symbol,as defined in 2.3.4. Explanation of the notation used in this chart is in Table 3–5.

If the length is not specified in an implied operand, it is determined by the assembler, as explained in 2.3.6.

3.3. INSTRUCTION REPERTOIRE

The entire range of machine instructions for the tape/disc systems is presented in the following paragraphs. Each instruction is shown under its own heading consisting of the full instruction name. Under each heading is a block containing the instruction type, mnemonic operation code, symbolic source-code operand format, hexadecimal operation code, and object instruction length for the instruction. (Refer to 3.2.1 and Tables 3–1 through 3–4.)

The block is followed by a function abbreviation using OP1 and OP2 to refer to the operands, parentheses to mean "the contents of," and an arrow to indicate the storage of the result. The function

$$(OP1) \cdot (OP2) \rightarrow OP1$$

means "the contents of the bytes addressed by the first operand are multiplied by the contents of the bytes addressed by the second operand and the result is stored in the bytes specified by the first operand."

A description of the operation of the instruction is given next, followed by operational conditions, and, in cases where it is necessary for clarity of explanation, an example of the working of the instruction.

The section for each instruction closes with examples of the instruction used in a line of source code as it would be written by a programmer.

3.3.1. Arithmetic Instructions

The arithmetic instructions add, subtract, multiply, or divide values stored in main storage, in registers, or in immediate operands.

The arithmetic instructions include:

Mnemonic	Function	Format
AH	Add half word	RX
AI	Add immediate	SI
AP	Add packed decimal	SS2
DP	Divide packed decimal	SS2
MP	Multiply packed decimal	SS2
SH	Subtract half word	RX
SP	Subtract packed decimal	SS2
ZAP	Zero and add packed decimal	SS2

The formats of the arithmetic instructions vary with the function of the instruction and the type of data operated upon. The result of the execution of the instruction is stored in the first operand of each instruction. The condition code is changed by most of the arithmetic instructions; the exceptions to this are the divide packed decimal and multiply packed decimal. The operands are processed from left to right in most instructions; the exceptions are the packed decimal instructions. The instructions are explained separately in the text and are accompanied by notes on their limitations and operations.

3.3.1.1. Overflow

Binary arithmetic instructions set the condition code after execution. A condition code setting of 3 denotes that overflow occurred during execution of the instruction. Overflow is a condition occurring when the sign of a numeric field, expressed in binary format, is changed erroneously. The sign change can occur when an attempt is made to develop a larger number than can be expressed in 15 bits. Because all binary data must fit into two bytes, one bit of which must be the sign, the maximum expressible numbers are +32,767 and −32,768.

Example:

```
          OP1       0010101101101000  (+ 11,112)
                          |
                          Sign

plus      OP2       0101010010011000  (+ 21,656)
                          |
                          Sign

          Result    1000000000000000  (- 32,768)
                          |
                          Sign
```

An add half-word instruction using the foregoing operands, by adding OP2 to OP1, results in a number, stored in OP1, of −32,768 instead of the true answer +32,768. The true answer does not fit in 15 bits and the sign bit is changed by execution of the instruction.

3.3.1.2. Add Half Word

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	AH	R1,D2(,B2)	AA	4 BYTES

Function:

(R1) + (OP2) → R1

Description:

Execution of the add half-word instruction adds the contents of the half word specified by OP2 to the contents of the register specified by R1; the result is stored in R1.

Operational conditions:

1. The OP2 address should be defined at a half-word boundary.
2. The condition code is set as follows:

   0 Result is zero.
   1 Result is negative.
   2 Result is positive.
   3 Overflow occurred.

Examples:

```
1    LABEL    Δ OPERATION Δ      OPERAND                    Δ    COMMENTS
                10           16
         AH     8,4(,1,4)                  COMPLETE SPECIFICATION
         AH     8,TOTA                     RELATIVE ADDRESS
```

3.3.1.3. Add Immediate

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	AI	$D_1(B_1),I_2$	A6	4 BYTES

Function:

(OP1) + I2 → OP1

Description:

Execution of the add immediate instruction adds the binary value specified by the immediate operand, I2, to the contents of the half word specified by OP1; the result is stored in OP1. The immediate operand may be written in any of several forms. (See examples.)

Operational conditions:

1. The most significant bit of the immediate operand is treated as a sign in a binary field. The I2 field is sign-extended by the hardware before execution of the instruction. Specifying a negative value for I2 results in OP1 being decremented by that value.

2. The OP2 address should be defined at a half-word boundary.

3. The condition code is set as follows:

   0 Result is zero.
   1 Result is negative.
   2 Result is positive.
   3 Overflow occurred.

Examples:

LABEL (1)	△OPERATION△ (10)	OPERAND (16)	△ COMMENTS
	AI	20(14),25	COMPLETE SPECIFICATION
	AI	BASF,X'33'	RELATIVE ADDRESS
	AI	BASF,-2	RELATIVE ADDRESS

3.3.1.4. Add Packed Decimal

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	AP	$D_1(L_1,B_1),D_2(L_2,B_2)$	FA	6 BYTES

Function:

(OP1) + (OP2) → OP1

Description:

Execution of the add packed decimal instruction causes the algebraic addition of the bytes specified by OP1 to the bytes specified by OP2. The result is stored in OP1. Both OP1 and OP2 should contain packed decimal fields.

Operational conditions:

1. Operands are processed from right to left.

2. If OP1 is shorter than OP2, the excess digits of OP2 are ignored; if OP2 is shorter than OP1, zeros are assumed to extend OP2.

3. OP1 and OP2 may occupy the same, or some of the same, bytes in main storage without affecting the operation of the instruction only if the low-order bytes of the two operands coincide in main storage.

4. If overflow occurs, the sign stored in OP1 always reflects the sign of the true answer. The sign of a zero is positive in all cases other than overflow.

5. The maximum size of operands is 16 bytes.

6. The condition code is set as follows:

   0 Result is zero.
   1 Result is negative.
   2 Result is positive.
   3 Overflow occurred.

Examples:

```
LABEL    Δ OPERATION Δ    OPERAND                    Δ    COMMENTS
1             10          16
         AP      9,6(,6,,9),,3(4,,8)                     COMPLETE SPECIFICATION
         AP      9,6(,,9),,3(,,8)                        IMPLIED LENGTH
         AP      PREPA(6),,RATE(4)                       RELATIVE ADDRESS
         AP      PREPA,,RATE                             IMPLICIT LENGTH AND ADDRESS
```

3.3.1.5. Divide Packed Decimal

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	DP	$D_1(L_1,B_1),D_2(L_2,B_2)$	FD	6 BYTES

Function:

(OP1) ÷ (OP2) → OP1

Description:

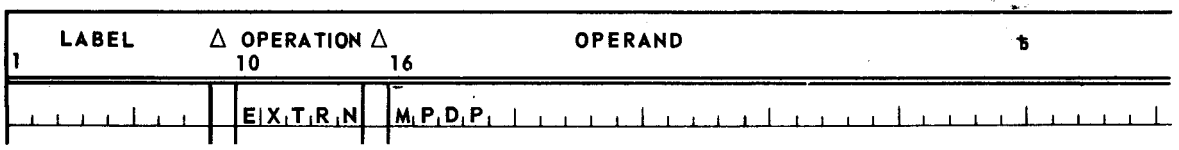
Execution of the divide packed decimal instruction divides the contents of the bytes specified by OP1 by the contents of the bytes specified by OP2 as the divisor. The quotient and remainder are stored in OP1. Both OP1 and OP2 should contain packed decimal fields.

Operational Conditions:

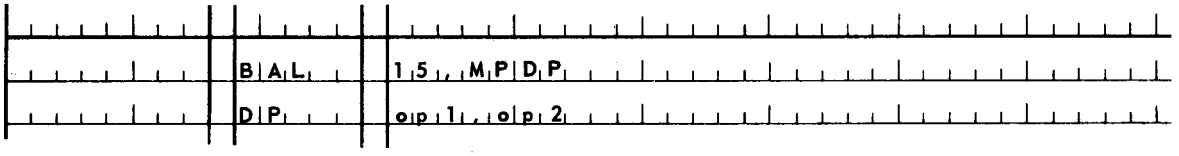
1. The following pertains to the 9200 system optional software versions of this instruction; also see card utility programs programmer reference manual, UP-4120.

   - The software divide and multiply functions are provided by one fixed, closed subroutine which is in relocatable object code and requires approximately 450 bytes of main storage. The multiply/divide subroutine must be linked to the problem program. To make the linking possible, the following source code statement is necessary:

- The source code necessary to accomplish the instruction is:



- The software version of the instruction differs in that:



OP1 length must be defined in the instruction; the length is not determined by the recognition of a sign in the field.

Maximum length of OP2 is eight bytes.

A divide check error in the software version causes a display of 29EE.

2. Operands are processed from right to left.

3. The length provided for OP1 must be large enough to accommodate both the quotient and a remainder. If the length provided is not sufficient, the answer is unreliable and bytes beyond OP1 may be affected by the execution of the instruction.

   A formula for determining a sufficient length for OP1 is:

   $$L = d + v$$

   where:

   L = length of OP1 in bytes.
   d = length of the dividend and sign in bytes (minimum of 2).
   v = length of the divisor and sign in bytes.
   All fractions rounded to the next higher integer.

   This formula yields a length that meets or exceeds the requirements of the instruction.

4. Both dividend and divisor must be right-justified in their respective fields; leading zeros must fill the fields.

5. At least one leading zero must precede the dividend. The absolute value of the divisor must be larger than the absolute value of the L2 most significant bytes of OP1. If these requirements are not met, a decimal-divide exception occurs and the processor halts.

6. OP2 and OP1 cannot overlap in main storage.

7. OP2 cannot exceed 16 bytes.

8. After execution, OP1 contains the quotient right-justified in the most significant bytes. Its length equals L1 minus L2 bytes. The remainder, if any, is right-justified in the least significant bytes of OP1; the length of the remainder equals L2. Leading zeros pad the quotient and the remainder.

9. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is the same as the sign of the dividend.

10. If a quotient digit greater than 9 is formed, a divide-check error occurs and the processor halts.

11. Division by zero produces a decimal-divide exception.

12. Execution of the instruction does not change the condition code setting.

13. An example of the use of the instruction is:

```
     LABEL     Δ OPERATION Δ          OPERAND                    Δ
     1              10        16
1    DIVD      DS              CL10
2    DSOR      DS              CL4
3                :
4                ZAP           DIVD,NUM1
5                ZAP           DSOR,NUM2
6                DP            DIVD,DSOR
7                MVC           OUT1(6),DIVD
8                MVC           OUT2(4),DIVD+6
```

In the sample source code, two storage areas labeled DIVD and DSOR are defined in lines 1 and 2. After processing of the problem program, data is moved to the storage area in lines 4 and 5. For the example, DIVD now contains the number:

00 00 00 00 99 03 05 07 00 1_+

DSOR contains the divisor:

03 00 04 0_+

Execution of the divide instruction, as coded in line 6, stores the quotient and the remainder in DIVD:

00 00 03 30 05 7_+ 02 02 72 1_+

The quotient is right-justified in the most significant bytes of DIVD, and the remainder is right-justified in the least significant bytes. The remainder always occupies the same number of bytes as OP2.

In the rest of the example coding, the result of the division is moved to separate data fields.

Examples:

```
LABEL   Δ OPERATION Δ   OPERAND                    Δ   COMMENTS
1           10          16
        DP              2,8(,10),,15),,8(,4,,14)       COMPLETE SPECIFICATION
        DP              DIVD(10),,DISR(4)              RELATIVE ADDRESS
        DP              DIVD,DSOR                      IMPLIED LENGTH AND ADDRESS
```

3.3.1.6. Multiply Packed Decimal

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	MP	$D_1(L_1,B_1),D_2(L_2,B_2)$	FC	6 BYTES

Function:

(OP2) · (OP1) → OP1

Description:

Execution of the multiply packed decimal instruction multiplies the contents of the bytes specified by OP2, the multiplicand, by the contents of the bytes specified by OP1, the multiplier. The result is stored in OP1. Both OP1 and OP2 should contain packed decimal fields.

Operational Conditions:

1. The following pertains to the 9200 system optional software versions of this instruction; see also card utility programs programmer reference manual, UP-4120.

   - The software multiply and divide functions are provided by one fixed, closed subroutine in relocatable object code requiring approximately 450 bytes of main storage. The multiply/divide subroutine must be linked to the problem program. To make linking possible, the following source code statements are necessary:

```
        EXTRN   MPDP
```

The source code necessary to perform the instruction is:

```
        BAL     15,MPDP
        MP      op1,op2
```

- The software version of the instruction differs in that:
  - OP1 length must be defined in the instruction; the length is not determined by the recognition of a sign in the field.
  - Maximum length of OP2 is eight bytes.
  - The multiplier is OP2, and the multiplicand is OP1 in the software version of the instruction. The result is still stored in OP1.

2. Operands are processed from right to left.
3. The length provided for OP1 must be equal to the number of bytes specified for OP2, plus the number of bytes needed to express the multiplier. The multiplier cannot be longer than L1 minus L2.
4. Zeros must fill the leftmost bytes of OP1.
5. Any excess length of the multiplier in OP1 will be ignored during execution.
6. The sign positions of both operands must contain values greater than nine.
7. The sign of the result is determined in accordance with the normal algebraic rules.
8. The maximum length of OP2 is 16 bytes.
9. OP1 and OP2 cannot overlap in main storage.
10. Failure to meet the requirements results in an unreliable product, but does not set an error indication. The bytes beyond OP1 can be affected by execution of the instruction unless the requirements are met.
11. Execution of the instruction does not change the condition code setting.
12. An example of the use of the multiply packed decimal instruction follows:

    To multiply 800 by –80, the multiplier would be

    08 0D

    OP2, the multiplicand, would be

    80 0F

    To fulfill the requirements of the instruction, OP1 contains the multiplier and a number of bytes equal to OP2. The total four bytes look like

    00 00 08 0D

    After execution OP1 contains

    00 64 00 0D

Examples:

```
LABEL   Δ OPERATION Δ   OPERAND                    Δ   COMMENTS
1           10          16
            MP          4,1(,4,,9),,4,1(,2,,1,0)       COMPLETE SPECIFICATION
            MP          WEX(,4),,LER(,2)               RELATIVE ADDRESS
            MP          WEX,,LER                       IMPLICIT LENGTH AND ADDRESS
```

3.3.1.7. Subtract Half Word

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	SH	$R_1,D_2(,B_2)$	AB	4 BYTES

Function:

(R1) – (OP2) → R1

Description:

Execution of the subtract half-word instruction subtracts the contents of OP2 from the contents of R1. The result is stored in R1.

Operational conditions:

1. The OP2 address should be defined at a half-word boundary.

2. The condition code is set as follows:

   0 Result is zero.
   1 Result is negative.
   2 Result is positive.
   3 Overflow occurred.

Examples:

```
            SH          14,,80(,,11)       COMPLETE SPECIFICATION
            SH          14,,FISCA          RELATIVE ADDRESS
```

3.3.1.8. Subtract Packed Decimal

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	SP	$D_1(L_1,B_1),D_2(L_2,B_2)$	FB	6 BYTES

Function:

(OP1) – (OP2) → OP1

Description:

Execution of the subtract packed decimal instruction subtracts algebraically the bytes specified by OP2 from the bytes specified by OP1. The result is stored in OP1. Both OP1 and OP2 should contain packed decimal fields.

Operational conditions:

1. The operands are processed from right to left.
2. If OP1 is larger than OP2, zeros are assumed to extend OP2. If OP2 is larger than OP1, the remaining digits of OP2 are ignored.
3. OP2 and OP1 may occupy some, or all, of the same bytes in main storage without affecting execution of the instruction only if the low-order bytes of the operands coincide in main storage.
4. If overflow occurs, the sign stored in OP1 will reflect the sign of the true result. The sign of a zero is always positive, except in overflow conditions.
5. The maximum size of operands is 16 bytes.
6. The condition code is set as follows:

   0 Result is zero.
   1 Result is negative.
   2 Result is positive.
   3 Overflow occurred.

Examples:

```
LABEL     Δ OPERATION Δ     OPERAND                          Δ  COMMENTS
1              10           16
          SP                73(6,10),76(3,10)                COMPLETE SPECIFICATION
          SP                73(,10),76(,10)                  IMPLIED LENGTH
          SP                HRSWK(6),HRSOU(3)                RELATIVE ADDRESS
          SP                HRSWK,HRSOU                      IMPLICIT LENGTH AND ADDRESS
```

3.3.1.9. Zero and Add Packed Decimal

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	ZAP	$D_1(L_1,B_1),D_2(L_2,B_2)$	F8	6 BYTES

Function:

$0 \rightarrow OP1; (OP2) \rightarrow OP1$

Description:

Execution of the zero and add packed decimal instruction clears the bytes specified by OP1 to zero and stores the contents of the bytes specified by OP2 in OP1. To avoid unpredictable results, OP2 should contain a packed decimal field; no restriction is placed on OP1 because its contents are overlaid by execution of the instruction.

Operational conditions:

1. Operands are processed from right to left.
2. If OP1 is larger than OP2, the excess positions of OP1 will be zero filled. If OP2 is larger than OP1, the most significant bytes of OP2 are ignored.
3. OP1 and OP2 can overlap in main storage only if the rightmost byte of OP1 coincides with, or lies to the right of, the rightmost byte of OP2.
4. If the same field is designated as both OP1 and OP2, the field remains unchanged.
5. The maximum size of operands is 16 bytes.
6. The condition code is set as follows:

   0 OP2 is zero.
   1 OP2 is negative.
   2 OP2 is positive.

Examples:

```
LABEL      OPERATION      OPERAND                  COMMENTS
1          10             16
           ZAP            4(14,8),8(6,14),15)      COMPLETE SPECIFICATION
           ZAP            4(8),8(6),15)            IMPLIED LENGTH
           ZAP            HEYB(14),ERGER(14)       RELATIVE ADDRESS
           ZAP            HEYB,ERGER               IMPLICIT LENGTH AND ADDRESS
```

3.3.2. Branch Instructions

Flexibility of program coding is provided with the branch instructions. Used after an instruction that changes the condition code setting, the branch instructions test the condition code and change the coding path according to the programmer's intentions. The branch and link instruction affords the additional device of allowing the program to return to the starting point of the varying path. The branch instructions are:

Mnemonic	Function	Format
BAL	Branch and link	RX
BC	Branch on condition	RX

The assembler affords extended mnemonic codes as shorthand symbols to facilitate the writing of branch instructions. The extended mnemonic codes are listed in Table 3–9. All of these codes represent the branch on condition instruction with different code settings in the R1 field of the instruction format.

The branch on condition instruction tests the condition code setting but does not change it. The branch on condition and branch and link instructions are explained separately in the text and are accompanied by notes on their limitations and operations.

3.3.2.1. Branch and Link

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	BAL	$R_1,D_2(,B_2)$	45	4 BYTES

Function:

Branch to OP2; store address at R1

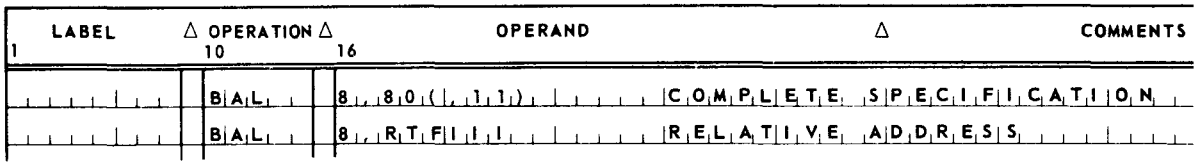
Description:

The branch and link instruction affords an unconditional branch to the address specified by OP2

Operational conditions:

1. Because the OP2 address is accessed first during execution of the instruction, no conflict is inherent in specifying the same register for R1 that is used as a base register in the indexed address of OP2.
2. Execution of the instruction does not change the condition code setting.

Examples:



3.3.2.2. Branch on Condition

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	BC	$R_1,D_2(,B_2)$	47	4 BYTES

Function:

If match branch to OP2

Description:

Execution of the branch on condition instruction tests the condition code setting with the mask specified by the R1 bits, and, if the test is met, a branch to the address specified by OP2 is executed. If the test is not met, the next instruction in sequence after the branch instruction is executed.

Operational conditions:

1. Each 1 bit in the R1 mask tests one of the four possible condition code settings. More than one bit can be used in the mask to test more than one condition code; the branch to the OP2 address occurs if one or more of the conditions is present.

Condition Code	R1
0	8
1	4
2	2
3	1

2. If R1 is zero, the result is a NO-OP; if R1 is 15, the result is an unconditional branch to the OP2 address.

3. Execution of the instruction does not change the condition code setting.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	BC	8,,1,1,0(,,1,4)	COMPLETE SPECIFICATION
	BC	8,,PROBA	RELATIVE ADDRESS
	BC	15,,NEXT	UNCONDITIONAL BRANCH
	BC	0,,NOT	SKIP

3.3.2.3. Extended Mnemonic Codes

The extended mnemonic codes use the following format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	code	$D_2(,B_2)$

where:

code
Is any mnemonic as listed in Table 3–9.

$D_2(,B_2)$

Is an address, either complete specification or relative form.

Table 3–9. Extended Mnemonic Codes

Mnemonic	Function	Hexadecimal Operation Code	R1	Format
B	Branch	47	F	RX
NOP	No operation	47	0	RX
	Used After Comparison Instructions			
BH	Branch if high	47	2	RX
BL	Branch if low	47	4	RX
BE	Branch if equal	47	8	RX
BNH	Branch if not high	47	D	RX
BNL	Branch if not low	47	B	RX
BNE	Branch if not equal	47	7	RX
	Used After Test Under Mask Instructions			
BO	Branch if all ones	47	1	RX
BZ	Branch if all zeros	47	8	RX
BM	Branch if mixed	47	4	RX
BNO	Branch if not all ones	47	E	RX
BNZ	Branch if not all zeros	47	7	RX
BNM	Branch if not mixed	47	B	RX
	Used After Arithmetic Instructions			
BO	Branch if overflow	47	1	RX
BZ	Branch if zero	47	8	RX
BM	Branch if minus	47	4	RX
BP	Branch if positive	47	2	RX
BNO	Branch if no overflow	47	E	RX
BNZ	Branch if not zero	47	7	RX
BNM	Branch if not minus	47	B	RX
BNP	Branch if not positive	47	D	RX

The format is a variation of the RX instruction format. The assembler uses the extended mnemonic to form both the operation code and the R1 field of the instruction.

Examples:

LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16)	Δ COMMENTS
	BH	24(,16)	COMPLETE SPECIFICATION
	BNZ	OUTPUTA	RELATIVE ADDRESS
	BNP	PRPB	RELATIVE ADDRESS

3.3.3. Comparison Instructions

The comparison instructions compare, either logically or algebraically, the contents of a location in storage to the contents of a register, of another storage location, or of an immediate operand. The comparison instructions include:

Mnemonic	Function	Format
CH	Compare half word	RX
CLC	Compare logical character	SS1
CLI	Compare logical immediate	SI
CP	Compare packed decimal	SS2
TM	Test under mask	SI

The formats of the comparison instructions vary with the function of the instruction. The data compared is not altered in memory. The condition code is set by all the comparison instructions. The operands are processed from left to right in most cases; in the compare packed decimal instruction the operands are processed from right to left.

The instructions are explained separately in the text and are accompanied by notes on their limitations and operations.

3.3.3.1. Compare Half Word

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	CH	$R_1,D_2(,B_2)$	49	4 BYTES

Function:

(R1) : (OP2)

Description:

The compare half-word instruction compares algebraically the contents of the register specified by R1 to the contents of the half word specified by OP2.

Operational Conditions:

1. The OP2 address should be defined at a half-word boundary.

2. The condition code is set as follows:

   0 (R1) = (OP2)
   1 (R1) < (OP2)
   2 (R1) > (OP2)

Examples:

```
LABEL    △ OPERATION △          OPERAND                     △         COMMENTS
1           10         16
            CH         13,26(,14)         COMPLETE SPECIFICATION
            CH         13,RTNRT           RELATIVE ADDRESS
```

3.3.3.2. Compare Logical Character

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	CLI	$D_1(B_1),I_2$	95	4 BYTES

Function:

(OP1) : I2

Description:

The compare logical immediate instruction compares logically the contents of the byte specified by OP1 to the immediate operand, I2.

Operational Conditions:

1. The operands are compared without regard to sign.

2. The condition code is set as:

   0 (OP1) = I2
   1 (OP1) $<$ I2
   2 (OP1) $>$ I2

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	CLI	23(9),X'09'	COMPLETE SPECIFICATION
	CLI	CHERABLE,X'09'	RELATIVE ADDRESS

3.3.3.3. Compare Logical Immediate

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	CLC	$D_1(L,B_1),D_2(B_2)$	D5	6 BYTES

Function:

(OP1) : (OP2)

Description:

The compare logical character instruction performs an absolute binary comparison of the contents of the address specified by OP1 of the contents of the address specified by OP1 to the contents of the address specified by OP2. The number of bytes to be compared is determined by the length specification in the first operand; the maximum is 256 bytes.

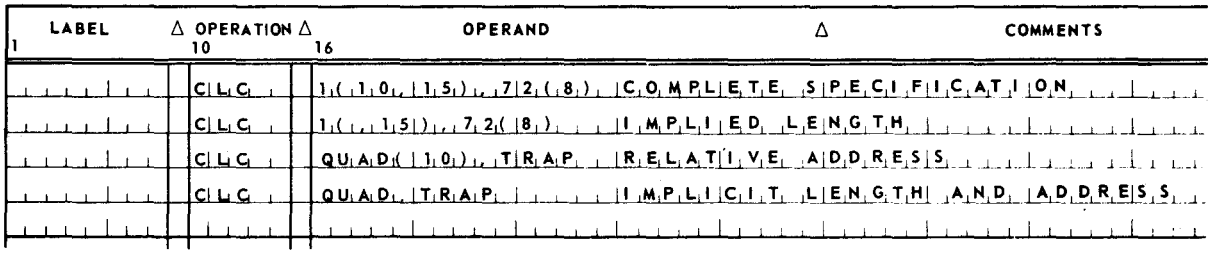
Operational Conditions:

1. Execution of the instruction is terminated and the condition code is changed after the first inequality is found.

2. The condition code is set as:

   0 (OP1) = (OP2)
   1 (OP1) < (OP2)
   2 (OP1) > (OP2)

Examples:



LABEL	OPERATION	OPERAND	COMMENTS
	CLC	1(10,15),72(8)	COMPLETE SPECIFICATION
	CLC	1(,15),72(8)	IMPLIED LENGTH
	CLC	QUAD(10),TRAP	RELATIVE ADDRESS
	CLC	QUAD,TRAP	IMPLICIT LENGTH AND ADDRESS

3.3.3.4. Compare Packed Decimal

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	CP	$D_1(L_1,B_1),D_2(L_2,B_2)$	F9	6 BYTES

Function:

(OP1) : (OP2)

Description:

The compare packed decimal instruction algebraically compares the contents of the bytes specified by OP1 to the bytes specified by OP2; the maximum number of bytes is 16. Both OP1 and OP2 should contain packed decimal fields.

Operational Conditions:

1. The operands are processed from right to left.

2. If OP1 is larger than OP2, the remaining digits of OP1 are compared to zero. If OP2 is larger than OP1, the remaining digits of OP2 are ignored.

3. A sign is assumed in the four rightmost bits of the least significant byte of both operands and is considered in the comparison.

4. The condition code is set as:

   0 (OP1) = (OP2)
   1 (OP1) < (OP2)
   2 (OP1) > (OP2)

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	CP	6(13,9),78(13,14)	COMPLETE SPECIFICATION
	CP	6(,9),78(,14)	IMPLIED LENGTH
	CP	PROVO(13),SDS(13)	RELATIVE ADDRESS
	CP	PROVO,SDS	IMPLICIT ADDRESS AND LENGTH

3.3.3.5. Test Under Mask

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	TM	$D_1(B_1),I_2$	91	4 BYTES

Function:

test (OP1) : I2

Description:

The test under mask instruction tests the bits in the byte specified by OP1, using I2 as a mask for the test. Bit positions not to be tested are indicated by 0 bits in the mask; 1 bits indicate bit positions to be tested.

Operational Conditions:

1. The condition code is set as:

   0 No match
   1 Partial match
   2 Not used
   3 Full match

Examples:

```
LABEL    Δ OPERATION Δ    OPERAND                    Δ    COMMENTS
1             10          16
         TM               8,0,(,1,4),,X,',F,1,'           COMPLETE SPECIFICATION
         TM               PHEAA,,X,',F,1,'                RELATIVE ADDRESS
```

3.3.4. Data Manipulation Instructions

The data manipulation instructions edit, translate, or change formats of data stored in main storage. The instructions include:

Mnemonic	Function	Format
ED	Edit	SS1
PACK	Pack	SS2
TR	Translate	SS1
UNPK	Unpack	SS2

Of the data manipulation instructions, only the edit instruction sets the condition code. Operands are processed from left to right in the translate and edit instructions and from right to left in the pack and unpack instructions.

The instructions are explained separately in the text and are accompanied by notes on their limitations and operands.

3.3.4.1. Edit

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	ED	$D_1(L,B_1),D_2(B_2)$	DE	6 BYTES

Function:

(OP2) → OP1

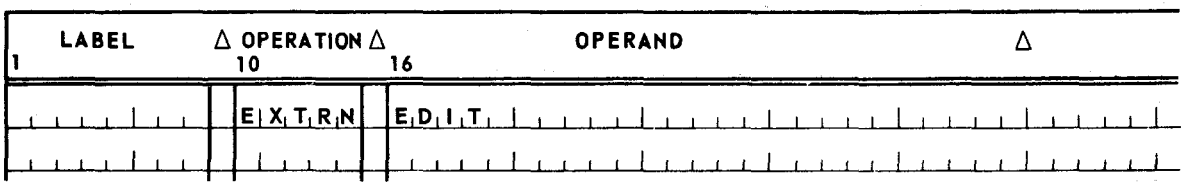
Description:

Execution of the edit instruction transfers data from the bytes specified by OP2 to the bytes specified by OP1, changing the data format from packed decimal to unpacked decimal, inserting zone bits and editing symbols, and suppressing leading zeros. This editing process is controlled by the editing mask in OP1, which is overlaid by execution of the instruction. OP2 should be a packed decimal field and OP1 should contain the editing mask.

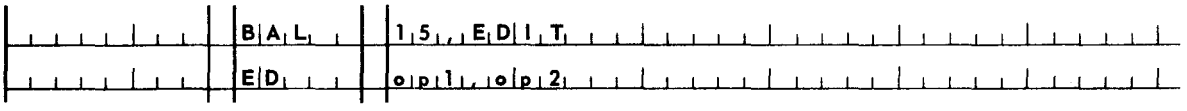
Operational Conditions:

1. The following pertains to the 9200 system optional software version of this instruction; see also card utility programs programmer reference manual, UP-4120.

   - The software edit function is provided by a fixed, closed subroutine in relocatable object code. It requires approximately 375 bytes of main storage and must be linked to the problem program. To make this linking possible, the following source code must be inserted in the program:



   - The source code necessary to accomplish the instruction is:



   - Other than the above, the software edit instruction is identical in operation to the hardware version.

2. The operands are processed from left to right.

3. The contents of OP1, after execution of the instruction, are dependent upon:

   Fill character
   Digit select byte (DSB)
   Significant start byte (SSB)
   Field separator byte (FSB)
   Editing symbols
   Sign of OP2

   a. Fill character

      The leftmost byte of OP1 contains the fill character, which can be any alphanumeric or special character. It is used to replace bytes in the OP1 editing mask. The first byte always remains as the fill character.

      The fill character is substituted for:

      - Leading zeros – any zero in OP2 not preceded by a significant nonzero digit.
      - Leading editing symbols – any byte containing one of the available characters specified for editing in the OP1 mask not preceded by a significant digit.
      - Field separator byte – a special byte that acts as a separator in multiple field editing.
      - Rightmost bytes of OP1 when the sign of OP2 is positive.

      The most commonly used fill characters are the asterisk and the blank.

b. Digit select byte (DSB)

The digit select byte, a hexadecimal 20, is replaced by digits from OP2 or by the fill character. The DSB is replaced by a digit from OP2 if that digit is significant or was preceded by a significant digit. The fill character replaces the DSB in all other situations.

c. Significant start byte (SSB)

By using a significant start byte, hexadecimal 21, in OP1, the presence of a significant digit in OP2 is simulated. The suppression of leading zeros and fill characters ceases with the byte immediately following the SSB.

d. Field separator byte (FSB)

The field separator byte, a hexadecimal 22, is used in the OP1 mask to separate fields in multiple-field editing. The FSB always is replaced by the fill character, and all leading zeros or editing symbols following the FSB also are replaced by the fill character. This procedure continues until a significant digit is processed or an SSB is specified in OP1 just as if this were the beginning of a new editing instruction.

e. Editing symbols

Any alphanumeric or special character may be specified as an editing symbol in OP1. These symbols never are replaced by a digit from OP2, but are replaced by the fill character before a significant digit has been processed.

f. Sign of OP2

If any byte of OP2 contains a 4-bit digit with a value greater than eleven (X'0B'), it is recognized as a sign. If the sign of OP2 is positive, any editing symbols in the mask at the end of OP1, or between the last DSB and an FSB, are replaced by the fill character. If the sign of OP2 is negative, these editing symbols remain in OP1.

4. Zone bits are inserted in OP1 when digit bits from OP2 overlay the editing mask. The bits inserted are 1111 in the EBCDIC mode, and 0101 in the ASCII mode.

5. An example of the use of the edit instruction is:

```
     LABEL     ↹ OPERATION ↹          OPERAND                    ↹
1              10          16
1    OPON      DC          XL6'0'
2    OPTU      DC          XL4'0'
3    EDMK      DC          X'402020206B2021204B2020C3D9'
4              :
5    ALPA      MVC         OPON,EDMK
6              ED          OPON,OPTU
7              PUT         PRINT,OPON
8              :
9              BC          15,ALPA
```

Line 1. A work area, OPON, is defined and binary 0's are used to fill it. It is the same length as the editing mask.

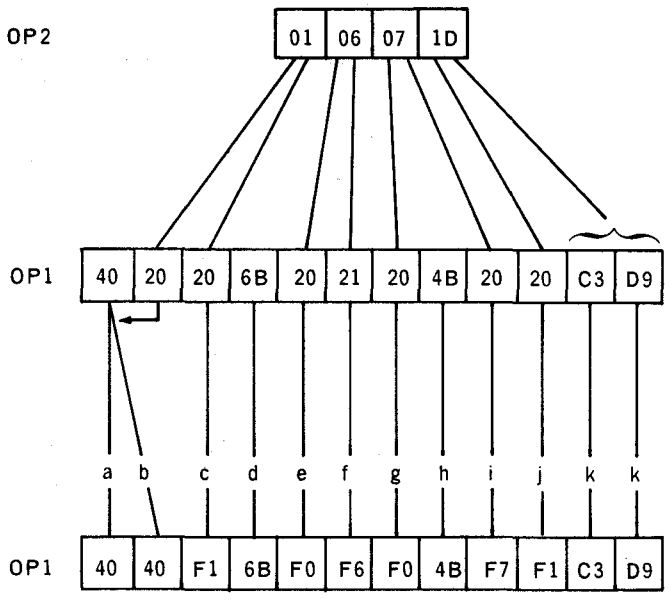
Line 2. A work area, OPTU, is defined and filled with binary 0's. The packed number to be edited is moved here.

Line 3. The editing mask, EDMK, is defined.

Line 4. Processing takes place; the number is generated.

Line 5. The editing mask is moved to preserve it.

Line 6. If OPTU contains a value representing –106071 and the desired output format is xx,xxx.xx if positive and xx,xxx.xxCR if negative, the operation of the edit instruction is:



a. The fill character remains; this first byte never is replaced.

b. The corresponding digit in OP2 is a leading zero; the fill character replaces this DSB.

c. The corresponding digit in OP2 is significant; it replaces the DSB.

d. This editing symbol, a comma, remains in OP1 because it was preceded by a significant digit.

e. This DSB is replaced by the digit from OP2 because it was preceded by a significant digit.

f. This SSB is replaced by the significant digit from OP2. It is placed in the mask to ensure that the next digits are placed in OP1 even if they are leading zeros.

g. This DSB is replaced because it is preceded by a significant digit and because it is preceded by an SSB; either is sufficient.

h. This editing symbol, a period or decimal point, remains in OP1 because it was preceded by a significant digit or an SSB.

i. This DSB is replaced by the significant digit from OP2.

j. This DSB is replaced by the significant digit from OP2.

k. These editing symbols, C and R (any number of the available characters could have been specified), remain in OP1 because the sign of OP2, a hexadecimal D, was negative.

The result of the operation stored in OP1 is 1,060.71CR.

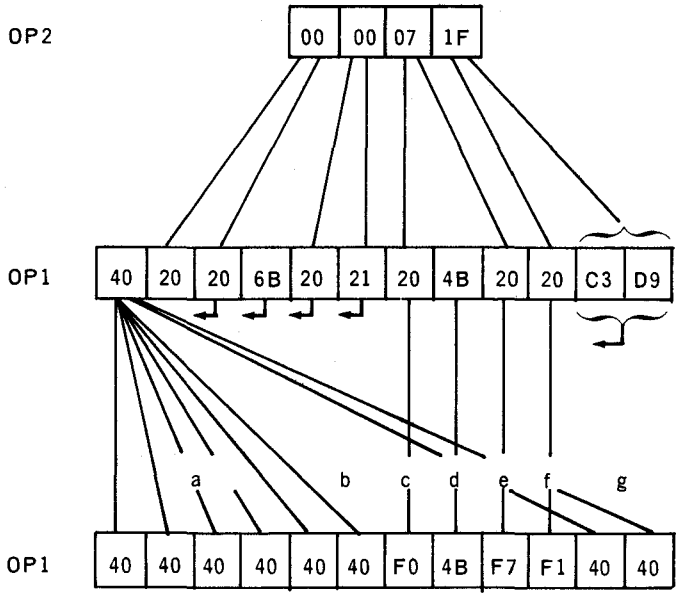
Line 7. The result of the editing process is then moved before printing.

Line 8. More processing occurs and another number is derived for editing.

Line 9. An unconditional branch back to the editing path is specified.

Line 5. The mask is moved again.

Line 6. If OPTU contains a value representing .71, and the output format is the same, the operation of the instruction is:



a. The fill character replaces the first six bytes of OP1 because no significant digit was processed in OP2.

b. The SSB is replaced by the fill character, but the next byte is replaced by the corresponding digit from OP2.

c. This DSB is replaced from the OP2 digits because it was preceded by an SSB.

d. This editing symbol remains because of the previous SSB.

e. This DSB is replaced by the significant digit from OP2.

f. This DSB is replaced by the significant digit from OP2.

g. These editing symbols are replaced by the fill character because the sign of OP2, a hexadecimal F, is positive.

The result of the operation sorted in OP1 is 0.71.

6. The edit instruction sets the condition code as:

0 The portion of OP2 following the last FSB is zero, or the last byte of OP1 is an FSB.

1 The portion of OP2 following the last FSB is not zero and is placed in OP1.

2 The portion of OP2 following the last FSB is not zero and is replaced by the fill character in OP1.

The condition code settings are useful in multiple-field editing, but do not reflect the contents of any part of OP2 that precedes the last FSB in the editing mask.

Examples:

LABEL (1)	△ OPERATION △ (10)	OPERAND (16)	△ COMMENTS
	ED	16(21,14),83(9)	COMPLETE SPECIFICATION
	ED	16(,14),83(9)	IMPLIED LENGTH
	ED	OPON(21),FLD2	RELATIVE ADDRESS
	ED	OPON,FLD2	IMPLICIT LENGTH AND ADDRESS

3.3.4.2. Pack

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	PACK	$D_1(L_1,B_1),D_2(L_2,B_2)$	F2	6 BYTES

Function:

(OP2) → OP1

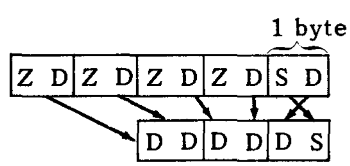
Description:

Execution of the pack instruction transfers the contents of the bytes specified by OP2 to the bytes specified by OP1, altering the data format from unpacked decimal to packed decimal.

Operational conditions:

1. Operands are processed from right to left, as:

OP2 (unpacked decimal)
OP1 (packed decimal)



where:

Z = zone bits
D = digit bits
S = sign bits

Note that the first byte sign and digit positions are reversed by execution of the instruction.

2. If OP1 is larger than OP2, the remaining digits of OP1 are zero filled; if OP2 is larger than OP1, the excess digits of OP2 are ignored.

3. The maximum length of operands is 16 bytes.

4. Execution of the instruction does not change the condition code setting.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	PACK	0(3,8),19(15,9)	COMPLETE SPECIFICATION
	PACK	0(,8),19(,9)	IMPLIED LENGTH
	PACK	EMI(3),JBL(5)	RELATIVE ADDRESS
	PACK	EMI,JBL	IMPLICIT LENGTH AND ADDRESS

3.3.4.3. Translate

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	TR	$D_1(L,B_1),D_2(B_2)$	DC	6 BYTES

Function:

Translate (OP1) using (OP2) → OP1

Description:

Execution of the translate instruction replaces the bytes of OP1 with the contents of the translation table at OP2. The binary value of each OP1 byte is used as the relative address of its replacement in the translation table.

Operational conditions:

1. Translation of OP1 occurs one byte at a time in a left-to-right sequence.

2. The capacity of the translate table is 256 characters.

3. The length specified in OP1 determines the number of bytes translated.

4. OP2 is not changed by execution of the instruction.

5. Execution of the instruction does not change the condition code setting.

6. The instruction operates as:

   The user has a message in his program that is in his own code, which, for this example, consists of a simple substitution of a number for a letter, 1 for A, 2 for B, ..., 26 for Z, and zero for blank. If the message is to be printed, the characters must be in EBCDIC code.

   The programmer sets up a table and stores it in main storage; in this case, it is stored at location 6000 and labeled TBLE. The table is 27 bytes long, but could be a maximum of 256 bytes. TBLE contains the hexadecimal values:

TBLE	(location 6000)	:	40
TBLE	+1	:	C1
TBLE	+2	:	C2
	.		.
	.		.
	.		.
TBLE	+26	:	E9

   In other words, TBLE contains the equivalent of the EBCDIC code for the alphabet.

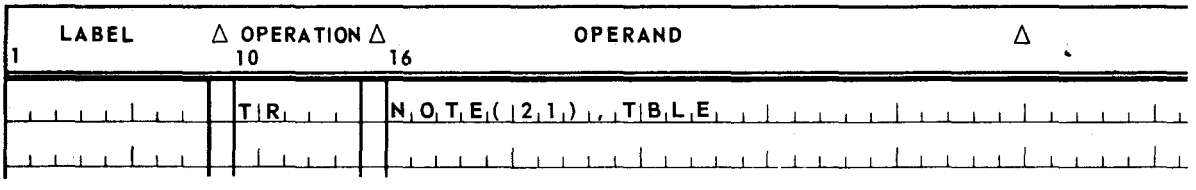
   The message in the program that must be translated is, in hexadecimal:

   0D, 05, 13, 13, 01, 07, 05, 00, 09, 13, 00, 14, 12, 01,

   0E, 13, 0C, 01, 14, 05, 04

   The message is labeled NOTE and is 21 bytes long.

   The instruction used to translate NOTE is:



   In the operation, the binary value of each byte is used to address relatively its replacement in the translation table.

   The value of the first byte of NOTE is 13 (because 0D in hexadecimal equals 13 in decimal); it is replaced by the thirteenth byte of TBLE (location 6000 + 13). TBLE + 13 contains the value 1101 0100, an EBCDIC M. That value replaces the first byte of NOTE.

The operation is the same for the second byte. Its value is 05, or 5; it is replaced by the value at location 6005, an EBCDIC E. Each byte of NOTE is replaced in that manner for the number of bytes specified by the length of the first operand in the instruction.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	TR	35(20,9),16(11)	COMPLETE SPECIFICATION
	TR	35(,9),16(11)	IMPLIED LENGTH
	TR	NOTE(20),TBLE	RELATIVE ADDRESS
	TR	NOTE,TBLE	IMPLICIT LENGTH AND ADDRESS

3.3.4.4. Unpack

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	UNPK	$D_1(L_1,B_1),D_2(L_2,B_2)$	F3	6 BYTES

Function:

(OP2) → OP1

Description:

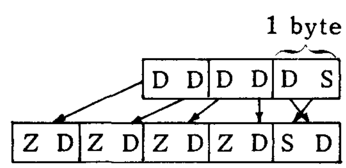
Execution of the unpack instruction transfers the contents of the bytes specified by OP2 to the bytes specified by OP1, altering the data format from packed decimal to unpacked decimal.

Operational conditions:

1. Operands are processed from right to left, as:

   OP2 (packed decimal)

   OP1 (unpacked decimal)



   where:

   Z = zone bits
   D = digit bits
   S = sign bits

   A sign is assumed to be in the rightmost byte of OP2 and the byte sign and digit positions are reversed by execution of the instruction.

2. The zone bits of OP1 are added to the digit bits from OP2. Either 1111, an EBCDIC zone pattern, or 0101, an ASCII zone pattern, is added.

3. If OP1 is larger than OP2, the remaining bits of OP1 are filled with the unpacked zero character (X'F0' EBCDIC). If OP2 is larger than OP1, the excess bits of OP2 are ignored.

4. Maximum length of operands is 16 bytes.

5. Execution of the instruction does not change the condition code setting.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	UNPK	38(10,15),128(6,15)	COMPLETE SPECIFICATION
	UNPK	38(,15),128(,15)	IMPLIED LENGTH
	UNPK	DFLD(10),NUMFLD(6)	RELATIVE ADDRESS
	UNPK	DFLD,NUMFLD	IMPLICIT LENGTH AND ADDRESS

3.3.5. Data Transfer Instructions

The data transfer instructions move data between storage locations and registers, other storage locations, or immediate operands. The data transfer instructions include:

Mnemonic	Function	Format
LH	Load half word	RX
MVC	Move characters	SS1
MVI	Move immediate data	SI
MVN	Move numerics	SS1
MVO	Move with offset	SS2
STH	Store half word	RX

The formats of the data transfer instructions vary with the function of the instruction. The instructions do not change the condition code setting, and operands are processed from left to right on all instructions except the move with offset.

The instructions are explained separately in the text and are accompanied by notes on their limitations and operations.

3.3.5.1. Load Half Word

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	LH	$R_1,D_2(,B_2)$	48	4 BYTES

Function:

(OP2) → R1

Description:

The load half-word instruction transfers the contents of the half word specified by OP2 to the register specified by R1.

Operational conditions:

1. The OP2 address should be defined at a half-word boundary.
2. Execution of the instruction does not change the condition code setting.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	LH	8,192(,15)	COMPLETE SPECIFICATION
	LH	8,POSTI	RELATIVE ADDRESS

3.3.5.2. Move Characters

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	MVC	$D_1(L,B_1),D_2(B_2)$	D2	6 BYTES

Function:

(OP2) → OP1

Description:

Execution of the move characters instruction transfers data from the bytes of main storage specified by OP2 to the address specified by OP1. The number of bytes transferred is determined by the length specification in OP1. The maximum number of bytes that can be moved is 256.

Operational conditions:

1. Characters are moved unaltered and OP2 is unaltered by the instruction.
2. Execution of the instruction does not change the condition code setting.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	MVC	25(80,9),114(,15)	COMPLETE SPECIFICATION
	MVC	TAG,114(,15)	IMPLIED LENGTH
	MVC	FISYR(80),CARD	RELATIVE ADDRESS
	MVC	FISYR,CARD1	IMPLICIT LENGTH AND ADDRESS

3.3.5.3. Move Immediate Data

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	MVI	$D_1(B_1),I_2$	92	4 BYTES

Function:

I2 → OP1

Description:

The move immediate data instruction transfers the immediate data, I2, to the byte specified by the OP1 address.

Operational conditions:

Execution of the instruction does not change the condition code setting.

Examples:

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ     COMMENTS
1              10        16
          MVI   142(12),X'8A'                        COMPLETE SPECIFICATION
          MVI   SYSALP,X'8A'                         RELATIVE ADDRESS
```

3.3.5.4. Move Numerics

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	MVN	$D_1(L,B_1),D_2(B_2)$	D1	6 BYTES

Function:

(OP2) → OP1

Description:

The move numeric instruction transfers the numeric bits from the bytes of data specified by OP2 to the numeric bits of the bytes specified by OP1. The number of bytes affected by the instruction is dependent on the length specification of OP1; the maximum is 256.

Operational conditions:

1. The zone bits of the bytes in OP1 are unaffected by the instruction.

2. Execution of the instruction does not change the condition code setting.

3. The instruction functions as:

	1 byte					
OP1 (before execution)	F 1	F 2	F 3	F 4	F 7	F A
OP2	F 4	C 4	C 4	E 4	C 4	D 1
OP1 (after execution)	F 4	F 4	F 4	F 4	F 4	F 1

After execution of the instruction, the zone bits of OP1 are unaltered and the digit bits are overlaid by the digit bits from OP2.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	MVN	8(9,10),11(12)	COMPLETE SPECIFICATION
	MVN	8(,10),11(12)	IMPLIED LENGTH
	MVN	OCT(9),ZEN	RELATIVE ADDRESS
	MVN	OCT,ZEN	IMPLICIT LENGTH AND ADDRESS

3.3.5.5. Move With Offset

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS2	MVO	$D_1(L_1,B_1),D_2(L_2,B_2)$	F1	6 BYTES

Function:

(OP2) → OP1

Description:

Execution of the move with offset instruction transfers data from the bytes specified by OP2 to the bytes specified by OP1, moving the data four bits to the left as it is processed.

Operational conditions:

1. The operands are processed from right to left.

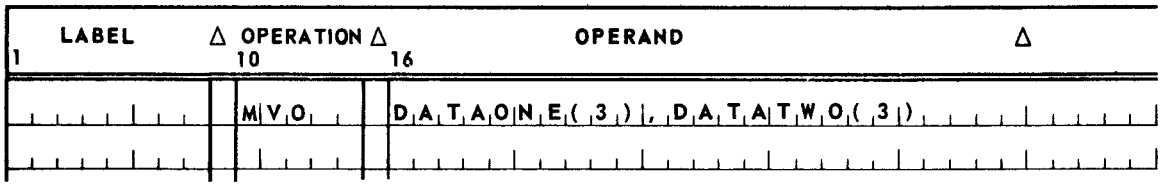
2. The first four bits of OP1, the least significant or rightmost bits, always remain unchanged.

3. If OP1 is larger than OP2, the remaining leftmost positions of OP1 are zero filled. OP1 must be at least one byte larger than OP2 to avoid truncation of the data on the left. If OP2 is larger than OP1, however, the excess leftmost positions of OP2 are ignored.

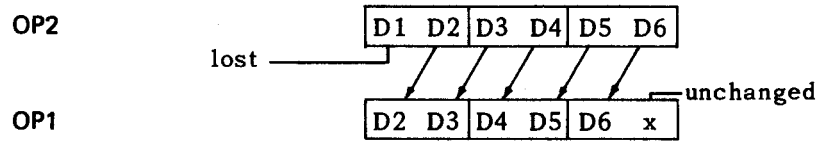
4. The maximum length of operands is 16 bytes.

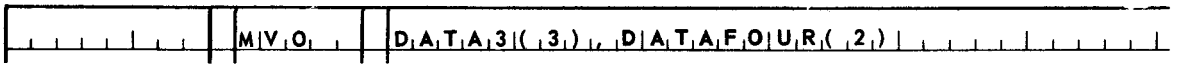
5. Execution of the instruction does not change the condition code setting.

6. The instruction functions as:

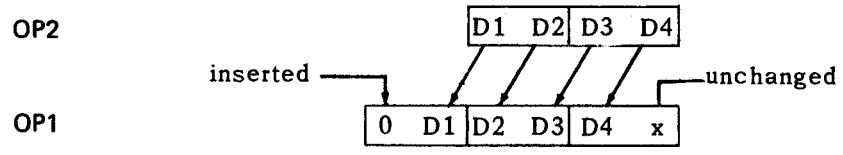


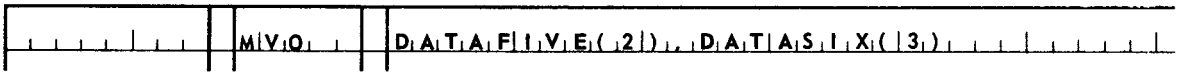
results in:



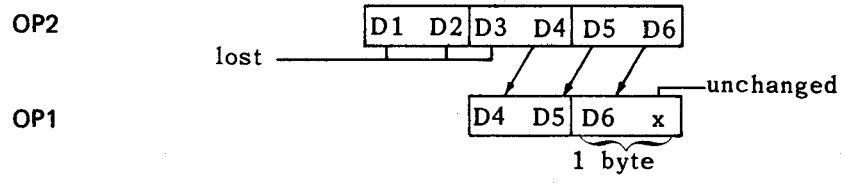


results in:





results in:



where:

D1,...,D6 = digits occupying four bits of the byte.
x = the original contents of these positions.
0 = zero fill of the MVO instruction.

Examples:

```
LABEL   Δ OPERATION Δ   OPERAND                    Δ   COMMENTS
1           10          16
        MVO             1,1(16),,11),,20(15,,11)       COMPLETE SPECIFICATION
        MVO             1,1(,,1(1)),,20(,,11)          IMPLIED LENGTH
        MVO             MOOG(16),,SYNTH                RELATIVE ADDRESS
        MVO             MOOG,,SYNTH                    IMPLICIT LENGTH AND ADDRESS
```

3.3.5.6. Store Half Word

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
RX	STH	$R_1,D_2(,B_2)$	40	4 BYTES

Function:

(R1) → OP2

Description:

The store half-word instruction transfers the contents of the register specified by R1 to the half word in main storage specified by the OP2 address.

Operational conditions:

1. The OP2 address should be defined at a half-word boundary.

2. Execution of the instruction does not change the condition code setting.

Examples:

```
        STH             15,,0(,,9)          COMPLETE SPECIFICATION
        STH             15,,USEFUL          RELATIVE ADDRESS
```

3.3.6. Display Instruction

The halt and proceed instruction is a display instruction; its execution stops the processor and displays a selected series of digits in the halt/display indicators of the console.

Mnemonic	Function	Format
HPR	Halt and proceed	SI

3.3.6.1. Halt and Proceed

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	HPR	$D_1(B_1),I_2$	A9	4 BYTES

Function:

display OP1

Description:

Execution of the halt and proceed instruction stops the processor and causes a display in the halt/display indicator on the console. If the most significant bit of OP1 is 0, the display is the 16 bits of OP1, as specified in the instruction. If the most significant bit of OP1 is 1, the display consists of the contents of the register specified by the leftmost 4 bits of OP1, plus the rightmost twelve bits of OP1.

Operational conditions:

1. I2 of the SI format need not be specified with this instruction.
2. The next instruction in sequence is executed by pressing START.
3. Execution of the instruction does not change the condition code setting.

Examples:

```
LABEL     Δ OPERATION Δ          OPERAND                    Δ
1              10        16
          HPR       X'3FFF'     DISPLAY 3FFF
          HPR       24(8)       DISPLAY ADDRESS
```

3.3.7. Input/Output Instructions

The input/output instructions are a pair of instructions that test the status of an input/output device or initiate an input/output function. The instructions are:

Mnemonic	Function	Format
TIO	Test input/output status	SI
XIOF	Execute input/output function	SI

The instructions are explained separately in the text and are accompanied by notes on their limitations and operations.

3.3.7.1. Test Input/Output Status

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	TIO	$D_1(B_1),I_2$	A5	4 BYTES

Function:

status → OP1

Description:

Execution of the test input/output status instruction tests the status of the device specified by I2 and sets the condition code; the status of the device tested is stored in the byte specified by the OP1 address.

Operational conditions:

1. The instruction clears the device status for the device tested if the device is not busy. If the device is busy, the status is not reset.

2. The interrupt request is part of the device status and is cleared by the instruction.

3. The condition code is set as:

   0 Device is available.
   1 Valid status.
   2 Busy.
   3 Rejected.

Examples:

LABEL (1)	OPERATION (10)	OPERAND (16)	COMMENT
	TIO	20(,15),X'05'	COMPLETE SPECIFICATION
	TIO	EVEN,X'05'	RELATIVE ADDRESS

3.3.7.2. Execute Input/Output Function

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	XIOF	$D_1(B_1),I_2$	A4	4 BYTES

Function:

execute OP1

Description:

The execute input/output function instruction initiates the function defined by the specification in OP1 on the device specified by OP2.

Operational conditions:

1. The XIOF instruction uses information characteristic to each peripheral input/output device. The necessary information can be found in the central processor unit programmers reference manual, UP-7546.

2. The buffer control word associated with the specified device must be loaded with the proper control information for this device before the XIOF instruction is executed.

3. If the instruction is executed with interrupt inhibited, the status or device address is never stored automatically. The procedure to be followed is the same as if I/O operations are performed in the I/O mode. In particular, the interrupt pending bit is set when the I/O operation is completed.

4. The function specification in OP1 defines the type of operation to be initiated.

5. For the dedicated I/O devices, bit 27 (the H bit) is reserved to inhibit generation of all interrupt requests when the operation ends. In this case, the interrupt pending bit is set at the completion of the I/O operation initiated by the instruction.

6. The condition code is set as:

   0 Function accepted.
   1 Status pending.
   2 Busy.
   3 Function rejected, invalid device number.

Example:

LABEL (1)	△ OPERATION △ (10)	OPERAND (16)	△	COMME
	XIOF	X'61',X'05'		COMPLETE SPECIFICATION

3.3.8. Logical Instructions

The logical instructions form the logical products or sums of data in storage, using other data in storage or immediate operands. The instructions include:

Mnemonic	Function	Format
NC	AND characters	SS1
NI	AND immediate data	SI
OC	OR characters	SS1
OI	OR immediate data	SI

The formats of the logical instructions vary with the function of the instruction. The instructions set the condition code and the result is stored in the first operand with each logical instruction.

The instructions are explained separately in the text and are accompanied by notes on operations and limitations.

3.3.8.1. AND Characters

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	NC	$D_1(L,B_1),D_2(B_2)$	D4	6 BYTES

Function:

(OP1) AND (OP2) → OP1

Description:

Execution of the AND characters instruction forms the logical product of the bytes specified by OP1 and OP2, and stores the product in OP1. The number of bytes affected by the instruction is determined by the length specification in OP1; the maximum is 256.

Operational conditions:

1. Each 1 bit in OP1 matched by a 1 bit in the same relative position in OP2 produces a 1 bit in OP1; in any other case, a 0 bit results in OP1.

2. The condition code is set as:

   0 Result is zero; no 1 bits in the product.
   1 Result is not zero; one or more 1 bits in the product.

Examples:

LABEL (1)	OPERATION (10)	OPERAND (16)	COMMENTS
	NC	25(,194,8),20(,14)	COMPLETE SPECIFICATION
	NC	25(,8),20(14)	IMPLIED LENGTH
	NC	NUEVA(,194),SINIS	RELATIVE ADDRESS
	NC	NUEVA,SINIS	IMPLICIT LENGTH AND ADDRESS

3.3.8.2. AND Immediate Data

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	NI	$D_1(B_1),I_2$	94	4 BYTES

Function:

(OP1) AND I2 → OP1

Description:

Execution of the AND immediate data instruction forms the logical product of I2 and the byte specified by OP1. The product is stored in OP1.

Operational conditions:

1. Each 1 bit in OP1 matched by a 1 bit in the same relative position in I2 produces a 1 bit in OP1; in any other case, a 0 bit results in OP1.

2. The condition code is set as:

   0 Result is zero; no 1 bits in the product.
   1 Result is not zero; one or more 1 bits in the product.

Examples:

```
1   LABEL   Δ OPERATION Δ   OPERAND                    Δ   COMMENTS
                10      16
        NI      28(15),X'80'    COMPLETE SPECIFICATION
        NI      BRTH,X'80'      RELATIVE ADDRESS
```

3.3.8.3. OR Characters

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SS1	OC	$D_1(L,B_1),D_2(B_2)$	D6	6 BYTES

Function:

(OP1) OR (OP2) → OP1

Description:

Execution of the OR characters instruction forms the logical sum of the bytes specified by OP1 and OP2 and stores the sum in OP1.

Operational conditions:

All 1 bits of OP2 are superimposed on OP1; 0 bits in OP2 do not affect OP1. The maximum number of bytes allowed is 256.

The condition code is set as:

0 Result is zero; no 1 bits in the sum.
1 Result is not zero; all or some 1 bits in the sum.

Examples:

LABEL	OPERATION	OPERAND	COMMENTS
	OC	115(120,8),4(9)	COMPLETE SPECIFICATION
	OC	115(,8),4(9)	IMPLIED LENGTH
	OC	LSMF(120),TEE	RELATIVE ADDRESS
	OC	LSMF,TEE	IMPLICIT LENGTH AND ADDRESS

3.3.8.4. OR Immediate Data

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	OI	$D_1(B_1),I_2$	96	4 BYTES

Function:

(OP1) OR I2 → OP1

Description:

Execution of the OR immediate data instruction forms the logical sum of I2 and the byte specified by OP1 and stores the sum in OP1.

Operational conditions:

All 1 bits of I2 are superimposed on OP1; 0 bits in I2 do not affect OP1.

The condition code is set as follows:

0 Result is zero; no 1 bits in the sum.
1 Result is not zero; all or some 1 bits in the sum.

Examples:

LABEL	△ OPERATION △	OPERAND	△ COMME
	OC	115(120,8),4(9)	COMPLETE SPECIFICATION
	OC	115(,8),4(9)	IMPLIED LENGTH
	OC	LSMF(120),TEE	RELATIVE ADDRESS
	OC	LSMF,TEE	IMPLICIT LENGTH AND ADDRESS

3.3.9. Supervisor Instructions

The supervisor instructions execution affects the contents of the lower bytes of main storage. The instructions store or modify one of the program state control words or communicate between operator and problem program. The supervisor instructions are:

Mnemonic	Function	Format
LPSC	Load program state control	SI
SPSC	Store program state control	SI
SRC	Supervisor request call	SI

While there are technically no "privileged" instructions on the 9200/9300 series systems, the use of the LPSC/SPSC instructions is usually reserved by software convention to the operating system.

The instructions are explained separately in the text and are accompanied by notes on their limitations and operations.

3.3.9.1. Load Program State Control

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	LPSC	$D_1(B_1),I_2$	A8	4 BYTES

Function:

(OP1) → PSC

Description:

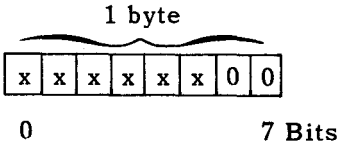
Execution of the load program state control instruction may replace or modify the specified program state control word in low-order main storage. The word specified by the OP1 address, under the control of the bits of the immediate operand I2, is stored at the PSC location.

Operational conditions:

1. The storage area specified by OP1 must be defined on a half-word boundary.

2. The functions of the OP2 bits are:

OP2



where:

x
May be either a 1 bit or a 0 bit, depending on the programmer's requirements.

Bits 0,1 control the loading of the PSC word and the ASCII bit.

0,1

00 PSC word remains unchanged.
01 Load full PSC word stored at the OP1 address.
10 Set ASCII bit in PSC word to zero.
11 Set ASCII bit in PSC word to one.

Bit 2 specifies which PSC is to be affected by the instruction.

2

0 Load processor PSC.
1 Load input/output PSC.

Bit 3 specifies which PSC word controls the next instruction executed.

3

0 Processor PSC has control.
1 Input/output PSC has control.

Bits 4,5 control the alter function.

4,5

00 Alter state remains unchanged.
01 Restriction to location 4 is removed.
10 Alter is restricted to location 4.
11 Result is unpredictable.

Bits 6, 7 are always 0.

Examples:

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ  COMMENTS
1              10          16
          LPSC              28(14),X'40'               COMPLETE SPECIFICATION
          LPSC              CHNG,X'08'                 RELATIVE ADDRESS
```

3.3.9.2. Store Program State Control

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	SPSC	$D_1(B_1),I_2$	A0	4 BYTES

Function:

PSC → OP1

Description:

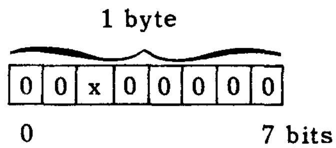
Execution of the store program state control instruction moves the specified program state control word to the address specified by OP1.

Operational conditions:

1. The storage area specified by OP1 must be defined on a half-word boundary.

2. I2, the immediate operand, controls the PSC that is stored; the third bit of the byte selects the PSC word:

OP2



where:

0 = bits not used.
x = bit used to select the PSC word:

0 Stores the processor PSC;
1 Stores the input/output PSC.

Examples:

LABEL (1)	OPERATION (10)	OPERAND (16)	COMMENTS
	SPSC	22(11),X'20'	COMPLETE SPECIFICATION
	SPSC	CHNG,X'00'	RELATIVE ADDRESS

3.3.9.3. Supervisor Request Call

INSTRUCTION TYPE	MNEMONIC OPERATION CODE	SYMBOLIC SOURCE CODE OPERAND FORMAT	HEXADECIMAL OPERATION CODE	OBJECT INSTRUCTION LENGTH
SI	SRC	$D_1(B_1),I_2$	A1	4 BYTES

Function:

I2 → SRC

Description:

Execution of the supervisor request call instruction stores the immediate operand, I2, in the SRC portion of the I/O PSC word and sets the interrupt request.

Operational conditions:

1. OP1 is ignored.
2. Interrupt occurs immediately when the processor is under control of the processor PSC word, but is stored when input/output PSC is in control.
3. When the interrupt request is accepted, no device address or status is stored.
4. An I/O operation can be completed when an SRC instruction is executed in the processor mode. The device address and status are stored, making them available when the interrupt occurs.

Example:

```
LABEL     Δ OPERATION Δ          OPERAND                    Δ
1              10        16
                SRC       X'1F'    COMPLETE SPECIFICATION
```

4. Assembler Directives

4.1. DIRECTIVES

In addition to the representation of machine instructions, constants, and storage, the SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler language includes several assembler directives. These are instructions to the assembler to perform certain functions; they afford the user of the assembler language control of the operation of the assembler.

The assembler directives, grouped by function, are:

Symbol definition

- EQU

Assembler control

- START
- END
- ORG
- LTORG

Base register assignment

- USING
- DROP

Program linking and sectioning

- CSECT
- DSECT
- COM
- ENTRY
- EXTRN

Listing control

PRINT

SPACE

EJECT

TITLE

Input control

ISEQ

Assembler directives EQU, END, ORG, USING, DROP, ENTRY, and EXTRN may use one or more symbols in the operand field, and, with the exception of ENTRY, EXTRN, USING, and DROP, the symbols must have appeared in the label field of a previous statement.

4.2. SYMBOL DEFINITION

EQU – Equate

The value and length attribute of a symbol may be defined explicitly. The statement format is:

LABEL	Δ OPERATION Δ	OPERAND
symbol	EQU	$e_1[,e_2]$

where:

e_1 and e_2 are expressions.

The symbol in the label field is defined as the value of the first expression in the operand field. If the value of the first expression in the operand field is not between 0 and 65,535, the statement is flagged with an error indication and the symbol remains undefined.

The symbol is defined to have a length attribute equal to the value of the second expression in the operand. The second expression in the operand may be omitted, in which case, the symbol is defined to have the length attribute of the first expression.

If the value of the location counter is 2000 when the following lines are detected,

LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16)	Δ
TAG	DS	25CL10	
HIDE	EQU	100+TAG,150	
SEEK	EQU	TAG+270,*	

TAG has an absolute value of 2000 and a length attribute of 10. HIDE has an absolute value of 2100 and a length attribute of 150. SEEK has an absolute value of 20 and a length attribute of 10.

4.3. ASSEMBLY CONTROL

Assembler directives are available to control the program name and initial location, to alter the location counter in a specified manner, to indicate the end of the program and the instruction with which execution of the object program is to begin, and to control the placement of pooled literals.

4.3.1. Program Start Directive (START)

The START directive defines the program name and tentative starting location. It must precede all other program statements except comments and macro definitions. The format of the START directive is:

LABEL	Δ OPERATION Δ	OPERAND
symbol	START	decimal or hexadecimal representation

The expression in the operand field is evaluated and, if necessary, incremented to make it a multiple of two. The result becomes the initial setting of the location counter and is the value of the symbol in the label field. This symbol becomes the name of the first control section and the program name and is available as an entry point without being separately defined as such. Although the operand of the START directive is an absolute value, it is treated as if it were relocatable.

Both the value of the location counter and the coding that follows a START directive are relocatable. Any one of the following statements will cause the program to be assigned to locations starting at 1068, and to have the symbol SORT defined with the relocatable value 1068.

```
LABEL      Δ OPERATION Δ      OPERAND                Δ
1                10                16
SORT       START              1067
SORT       START              1068
SORT       START              X'42C'
```

A START directive preceded by one or more statements, other than comments or macro definitions, is ignored and flagged as an error. A START directive whose operand field does not have a value from 0 to 65,532 is ignored and flagged as an error. If there is no valid START directive, the location counter is set to 0.

The value stated on the START directive is affected by the e_2 specification of the ORG directive; see 4.3.3.

4.3.2. Program End Directive (END)

The END directive indicates to the assembler the end of the program being assembled. The format of the END directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	END	[expression]

When the assembler detects an END directive during its last pass, it writes any remaining text data which it has accumulated, then writes a transfer record. If the operand field of the END directive contains an expression, its value is entered in the transfer record to signify the address at which the program execution may begin. If the END directive does not contain an expression in the operand field, the corresponding field of the transfer record is blank.

If a symbol appears in the label field of the END directive, it is assigned the current value of the location counter. This is normally one greater than the highest address assigned to the program being assembled.

4.3.3. Assign Location Counter Origin Directive (ORG)

The ORG directive is used to set the location counter to a specified value. The format of the ORG directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	ORG	$e_1[,e_2]$

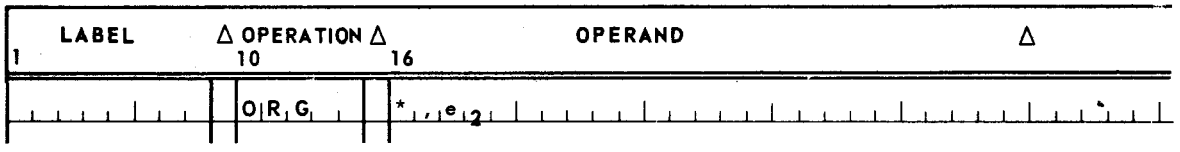
The value to which the location counter is set is determined by the values of the expressions in the operand field. If e_2 is not expressed, then the location counter is set to the value of e_1. If e_2 is expressed, the location counter is set to the next value greater than or equal to the value of e_1 which is a multiple of e_2. Some examples follow.

Operand	Resulting Location Counter Value
1000	1000
1000,2	1000
1000,16	1008

The value of e_2 must be a number that is the result of 2 raised to a power.

If e_2 is used, the value stated on the START directive must be a multiple of the highest e_2 value expressed in the assembly. The assembler assumes a start value of zero until the final pass, when base adjustments are computed. The ORG value of e_2 will be lost if the adjustment factor is not a proper multiple.

Because the beginning address of each CSECT is based on the ending address of the previous CSECT, and ORG*,e_2 statement must precede each CSECT directive when such ORG statements are used in the following section, as follows:

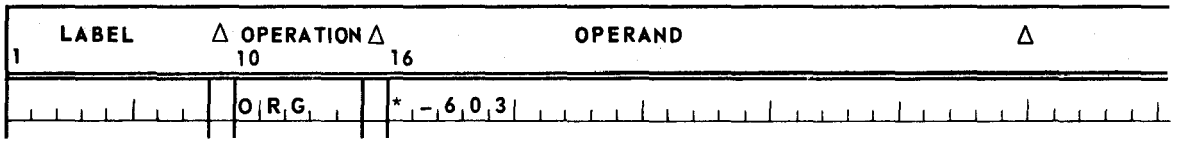


where:

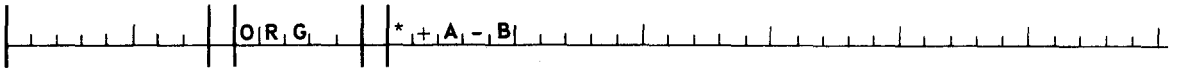
e_2 is the highest e_2 value used.

If a symbol appears in the label field, its value also is the value to which the location counter is set and the symbol is assigned a length attribute of one. The value to which the location counter is to be set must be either an absolute value between 0 and 65,535 or a relocatable value between the initial location counter setting and 65,535. If the value is not within this range, the ORG directive is ignored and the line is flagged with an error indication. With the ORG directive, it is possible to set the location counter to a value that is not a half-word boundary.

The ORG directive to set the location counter to a value 603 less than its current setting would be:



The ORG directive may be used to reserve a number of locations that are not expressed as a single decimal integer. For example, to reserve A minus B bytes of storage, where A and B are previously defined symbols, the statement is written as:



Bytes of storage reserved with a DS or ORG directive are not set to zero when the program is loaded.

If e_1 is a relocatable expression, the value to which the location counter is set and the coding that follows the ORG directive are relocatable. If absolute, the value to which the location counter is set and the coding that follows the ORG directive are absolute.

4.3.4. Assign Literal Pool Origin Directive (LTORG)

Literals are gathered into one or more pools with elimination of duplicates within one pool at the end of the first control section (4.5.4). An LTORG statement directs the assembler to create a pool from all literals detected since the last LTORG directive or beginning of the program. It also assigns to the pool the present value of the location counter, after adjusting the location counter to a half-word boundary.

The form of the LTORG statement is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	LTORG	unused

The label, if present, is assigned the value of the address of the first byte of the literal pool and a length attribute of one.

Because literals are gathered into pools before they are generated and the pools are processed as standard specifications, errors are not flagged when the literal is written. Invalid specifications are not discovered until the literal pool is dumped.

4.4. BASE REGISTER ASSIGNMENT

The assembler assumes the responsibility for converting storage addresses to base register and displacement values for insertion in instructions being assembled. To do this the assembler must be informed of the available registers and the values assumed to be in those registers. The assembly directives USING and DROP are available for this purpose.

4.4.1. Assign Base Register Directive (USING)

The USING directive informs the assembler that a specified register is available for base register assignment and that it contains a specified value. The format of the USING directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	USING	$r,a_1,\ldots,a_n$

where:

r
Is a relocatable or absolute expression and $a_1,\ldots,a_n$ are register numbers.

The second and succeeding expressions in the operand field must be numbers from 8 through 15, denoting general registers. The first expression represents the value that the assembler assumes is in the first register at object time. That number, plus 4096, will be assumed in the second register, and so on. Thus, if the label 'ABC' has a value of 400, the statement

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10          16
          USING   ABC,12,9,13
```

informs the assembler that, at object time, general register 12 contains the value 400, general register 9 contains the value 4496, and general register 13 contains the value 8592.

The registers must be loaded with the correct values by the user program. For instance, the instructions for the situation described previously would be:

```
          :
          USING   ABC,12,9,13
ABC       LH      12,TAG1
          LH      9,TAG2
          LH      13,TAG3
          :
TAG1      DC      Y(ABC)
TAG2      DC      Y(ABC+4096)
TAG3      DC      Y(ABC+8192)
          :
```

4.4.2. Unassign Base Register Directive (DROP)

The format of the DROP directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DROP	$a_1,\ldots,a_n$

where:

$a_1,\ldots,a_n$
Are absolute expressions, each with a value from 8 through 15.

Each expression denotes a general register that no longer contains a value available to the assembler for computing base register and displacement values.

4.4.3. Function of USING and DROP Directives

The assembler maintains a table of the available registers and the values they contain at object time, the USING table. A USING directive adds a register and value to the USING table, or revises the value for a register already in the table. A DROP directive removes a register and its associated value from the table. If the operands of a USING or DROP directive are not valid, the directive is ignored and the line is flagged with an error indication.

If an operand address is given as a relative address instead of a base register and displacement specification, the assembler searches the USING table for a value yielding a valid displacement; that is, a displacement of 4095 or less. If it finds more than one such value, that value yielding the smallest displacement is chosen. If no value yields a valid displacement, the operand address is set to zero and the line is flagged with an error indication. If more than one register contains the value yielding the smallest displacement, the highest numbered register is selected.

An absolute address with no indicated base register is treated as an absolute, direct address unless an appropriate value is found in the USING table.

The placement of a USING directive determines the generation of operand addresses within instructions, based on that USING statement. The first operand of the USING statement determines the portion of the program that may be addressed using the specified register. Thus, if a program contains the coding

```
LABEL        Δ OPERATION Δ        OPERAND                    Δ
1                 10              16
             USING   B,,10
A            LH      10,,B
             .
             .
             .
B            DC      Y(,C)
C            DS      CL10
             .
             .
             .
```

the B2 and D2 fields of the instruction labeled A contain 10 and 0, respectively. Moreover, if the program contains no USING directives for register 10, other than those shown, the second line, labeled A, is the only line in the program for which the assembler would consider 10 as a register available for addressing the line labeled B.

The SPERRY UNIVAC 9200/9300 Series operating system locator/loader routine places the transfer address of a main program in processor register 15 prior to transferring control to it. All other registers must be loaded by the program itself in a manner consistent with the information given to the assembler in the USING directives. The following example shows how this is done.

LABEL (1)	OPERATION (10)	OPERAND (16)
	USING	A,15
A	LH	12,B
	USING	C,12
	.	
	.	
	.	
B	DC	Y(C)
	.	
	.	
	.	
C	DS	CL10
	.	
	.	
	.	
	END	A

Lines 2 and 3 of the example exemplify the following general rule:

> An LH instruction to load a value into a general register must precede the USING directive, which informs the assembler that the value is available.

Also, it is possible to specify an absolute value for the first expression in the operand of a USING directive. The entry in the USING table made in response to such a USING directive is not used to break down relative addresses. It is used, instead, to break down absolute addresses. For example, given the following coding,

LABEL	OPERATION	OPERAND
	USING	4000,15
A	LH	14,4096

the B2 and D2 fields of the instruction labeled A will contain 15 and 96, respectively.

4.4.4. Direct Addressing

The machine instruction format provides base register and displacement addressing (indexed addressing) or direct addressing. Instructions using direct addressing have a faster execution time. To facilitate error checking by the assembler, direct addressing is described to the assembler in terms of the pseudo base registers 0, 1, 2, 3, 4, 5, 6, and 7, which contain the values 0, 4096, 8192, 12,288, 16,384, 20,480, 24,576, and 28,672, respectively. Thus, the direct address 512 would be treated, by the assembler, as an address consisting of a reference to the pseudo base register 0 and a displacement of 512. The address 4098 would yield a base of 1 and a displacement of 512. The address 4098 would yield a base of 1 and a displacement of 2. The additional forms of the USING directive available for direct addressing are, specifically:

LABEL	Δ OPERATION Δ	OPERAND
	USING	*,0
	USING	*,1
	.	.
	.	.
	.	.
	USING	*,7

The first line makes direct addressing available for addresses in the range 0 to 4095. The second makes direct addressing available for addresses in the range 4096 to 8191, and so on. The DROP directive also may refer to the pseudo registers 0 through 7 to terminate direct addressing.

A program involving direct addressing still may be relocatable.

When used in the operand of the USING directive the asterisk (*), does not indicate the current value of the location counter.

Both general registers and pseudo registers may not be specified in one USING directive.

4.5. PROGRAM LINKING AND SECTIONING

The assembler provides, as part of its output, information that allows the results of separate assemblies to be linked, loaded, and executed as a single program. Proper partitioning reduces the machine time required to make changes to an existing program. If a change is required, only that part which is changed need be reassembled. The output is then linked with the remaining parts to produce the altered program. Proper partitioning of a program also reduces the number of symbols required in each of the separate assemblies.

A symbol defined in the label field of module A and addressed in module B is said to be defined in module A and referenced in module B. Thus, by using the ENTRY and EXTRN directives, proper linkage is supplied when the separate modules are assembled. This information is given to the linker program by the external definition records and the external reference records which are outputs of the assembler and inputs to the linker.

4.5.1. Identify Entry-Point Directive (ENTRY)

That portion of a program submitted as input to a single assembly is called a module. Each module must declare the symbols defined within that module to which reference is made by other modules. These symbols are declared by the ENTRY directive. The directive format is:

LABEL	Δ OPERATION Δ	OPERAND
unused	ENTRY	symbol,symbol,...,symbol

Each symbol in the operand field is defined by its use as a label in a line of code within the module and referenced by one or more separately assembled modules.

4.5.2. Identify Externally Defined Symbols Directive (EXTRN)

The assembler must be informed of all symbols defined in other modules and referred to in the module being assembled. These symbols are declared by the EXTRN directive. The directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	EXTRN	symbol,symbol,...,symbol

Each symbol in the operand field is defined in some other module.

4.5.3. Sectioning

The assembler also permits sectioning of input to one assembly. Dividing a program into sections is optional; therefore, when writing unsectioned programs, the programmer does not need to know the points stressed in the following discussion and the discussion of CSECT, DSECT, and COM.

In a program without CSECT or DSECT sectioning directives, the entire program is one section that begins with the START directive and ends with the END directive. If, however, sectioning directives are used, the first section of the program begins with START and ends with a CSECT or DSECT sectioning directive, which also indicates the beginning of the next section. Each subsequent sectioning directive informs the assembler of the end of one section and the beginning of the next. The last section of the program ends with the END directive. The sectioning directive of each section is named by a label.

LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16)	Δ
PROG	START	1068	
ABLE	USING	PROG,12,,9,,13	
	:		
FROG	CSECT		
BAKE	DS	25CL10	
	:		

(cont.)

```
LABEL      Δ OPERATION Δ      OPERAND                    Δ
1                10          16
TROG           CSECT
CHAR           ORG          4000,4
               :
DROG           DSECT
               :
NMRE           END
```

ABLE is within a section named PROG; BAKE is within FROG; and so forth. The last section, a dummy control section, ends with the END directive.

A control section is a group of instructions, constants, and storage areas where positions relative to each other are fixed, and must be preserved for proper coding performance. An instruction in one section must not depend on its positional relation to instructions or data in another section for its proper execution. Sections appear in the input to the assembler in any order along with statements belonging to one or more other sections. The first control section begins with the first source statement. If the first statement is a START directive, its label becomes the name of the first control section.

4.5.4. Control Section Identification (CSECT)

The CSECT directive indicates to the assembler that the source statements that follow belong to a control section different from other preceding source statements. Use of the CSECT directive allows the programmer to code parts of the logical sections of a program in the order in which he determines the need for them. The format of the CSECT directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	CSECT	unused

The label field of the CSECT statement contains the name of the control section. This symbol must not appear in the label field of any other source statement except another CSECT statement. It is the name of the control section and is defined as an entry point of the program being assembled. The value of the symbol is the address of the first byte of the control section. If the label of the CSECT statement has appeared as a label of previous CSECT statement, the succeeding statements are a continuation of the control section of that name.

A programmer may find the use of CSECT convenient if, in the past, it was necessary to discontinue coding a section of a program to code another section upon which the original section is dependent. Shuffling of coding sheets is eliminated by setting up the new section with a CSECT directive and continuing. After the second logical section is coded or even partly coded, the programmer can revert to the original section by writing a CSECT directive with the same label given to the original section. The assembler reorganizes parts of each section and assembles it as one continuous control section. It is important to note, however, that neither the listing nor the sequence of object coding is reorganized. The reorganization that takes place is with respect to the final structure of the coding within main storage after loading; the addresses of the coding within main storage are indicated on the listing.

Example:

LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
BILL	START	1065 * AUTOMATICALLY SETS UP CSECT
	.	
	.	CODING: PART A OF FIRST
	.	SECTION (1A)
MIKI	CSECT	
	.	
	.	CODING: PART A OF SECOND
	.	SECTION (2A)
BILL	CSECT	
	.	
	.	CODING: PART B OF FIRST
	.	SECTION (1B)
MIKI	CSECT	
	.	
	.	CODING: PART B OF SECOND
	.	SECTION (2B)
BILL	CSECT	
	.	
	.	CODING: PART C OF FIRST
	.	SECTION (1C)
	END	

Assembled output:

First section labeled BILL – all of coding 1A, 1B, and 1C;
Second section labeled MIKI – all of coding 2A and 2B.

Operational conditions:

- Direct addressing between control sections must not be attempted.

- The first CSECT with a unique label also sets up an automatic entry point.

- An ORG directive with $*,e_2$ in the operand must be placed at the end of any CSECT when such ORG directives appear in succeeding sections; the e_2 specification must be the highest e_2 specification used (4.3.3).

  NOTE:

  Modules which contain sections defined by CSECT and COM directives with duplicate names must not be linked to one another. A blank label field is permitted as a legitimate label of spaces only once. When both CSECT and COM directives have no labels, the label used for both is spaces; however, only one label of spaces is permitted.

- USING directives in different sections that define the same register must be redefined when each section is reentered.

4.5.5. Dummy Control Section Identification (DSECT)

The DSECT directive indicates to the assembler that the statements which follow are used to redefine a data storage area reserved either in the module being programmed or on another separately assembled module. If the data storage area is reserved in a separately assembled module, that module is later linked to the module containing the dummy control section. No storage is reserved for a dummy control section. Data and instructions appearing in a dummy control section do not become part of the assembled program. The format of the DSECT statement is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DSECT	unused

A DSECT statement may not contain a blank label field in a program that either contains no START statement or contains a START or CSECT statement with a blank label field.

An LTORG directive may not appear in a dummy section. Labels of statements in a dummy section are called dummy labels.

The following rules must be observed in the use of dummy labels:

- An unpaired dummy label may appear only in an expression defining a storage address for a machine instruction or a constant of type S.
- A base register may not be designated for this address field, but the resulting value must be covered by a USING statement.
- The programmer must ensure that the appropriate value is loaded into the register specified in the USING statement.

The last source code input to an assembly must not be part of a dummy control section.

More than one dummy control section can be used within a module.

Example:

PROGRAM MODULE A

LABEL	Δ OPERATION Δ	
1	10	16
BELL	START	1065
	ENTRY	AREA
	.	
	.	
AREA	DS	CL260
	.	
	END	

PROGRAM MODULE B

LABEL	Δ OPERATION Δ	
1	10	16
EASE	CSECT	2095
	EXTRN	AREA
LAKE	DC	Y(AREA)
	LH	9,LAKE
SAIL	DSECT	
FLDA	DS	CL2
FLDB	DS	CL4
MIKI	CSECT	
	END	

In module A, the symbol AREA, defined as an ENTRY point, is specified as 260 bytes.

In module B, the base address of AREA is externally defined. Portions of AREA are redefined by DSECT as FLDA, containing 2 bytes, and FLDB, containing 4 bytes. FLDA and FLDB are relatively addressed as location 0 and location 2, respectively. Before FLDA and FLDB are addressed, register 9 must contain the base address of LAKE, which receives its true value at linker time.

4.5.6. Common Storage Definition (COM)

The ENTRY and EXTRN directives provide one mechanism for communication between separately assembled routines. The COM directive allows the programmer to define a control section that is a storage area common to two or more separately assembled routines. One assembly can define only one common section. Each COM statement, after the first, defines a continuation of the common section previously defined. The assembler reorganizes all common sections in a given assembly into one common section. Like the CSECT directive (4.5.4), neither the listing nor the sequence of object coding is reorganized. The last source code input to an assembly must not be a part of the common section. The format of the COM statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	COM	unused

Statements following the COM directive may be DS and DC directives used to define labels appearing within the section. Like the dummy control section (4.5.5), no data or instructions are assembled into a common section, nor may an LTORG directive appear in a common section. Data is entered into a common section only by execution of a program that refers to it. Unlike a dummy control section, a common storage is directly addressed by a program that defines it.

When several routines defining common storage are linked, the resulting module contains only one common section, corresponding to the longest common section in any of the input modules. The linker places the one common section above the highest address of any module or above job control's highest address, whichever is larger (Figure 4–1). In addition to being common to more than one module making up a loadable program, the common section can be made common to more than one link in a chain of programs by use of the CHAIN command card (Section 8). When programs are chained, the linker relocates the common section above the highest location in any chained program.

NOTE:

The linker always relocates the common section to an address higher than the highest located phase and higher than job control.

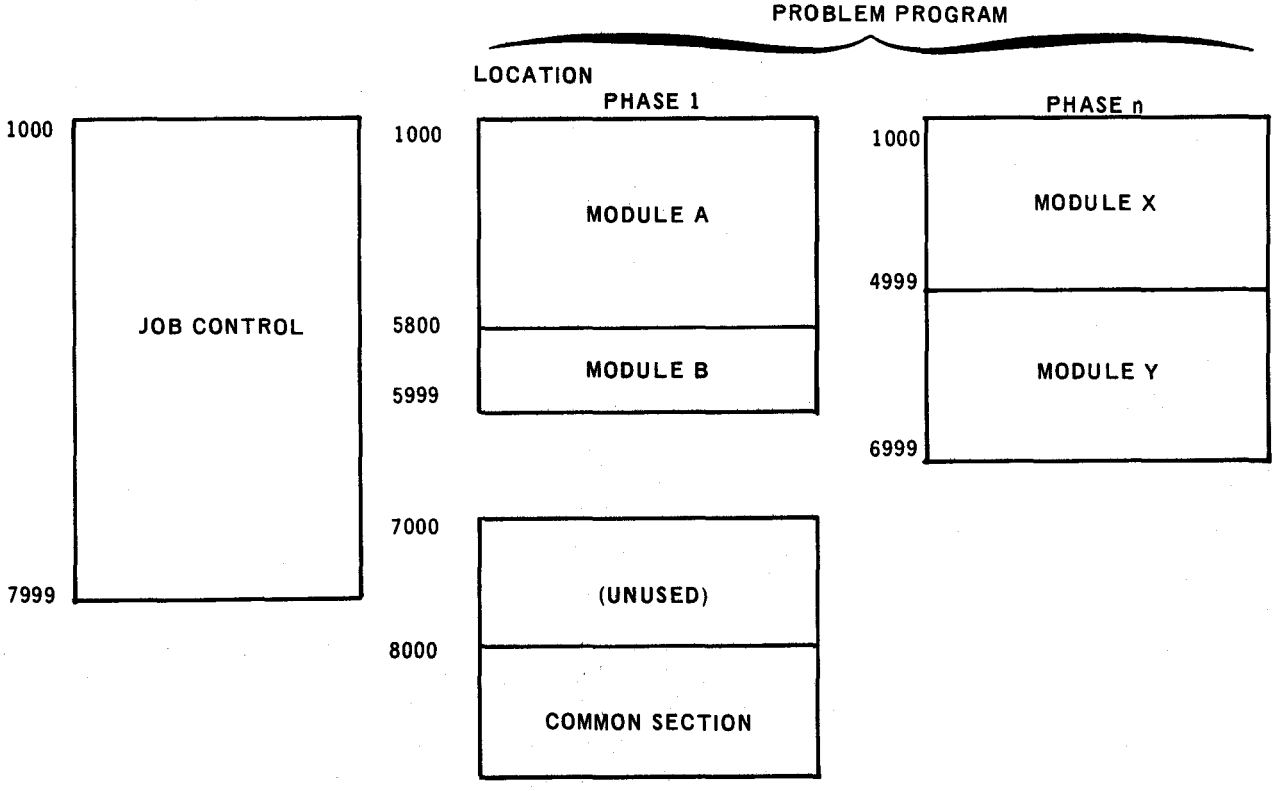


Figure 4–1. Relocation of the Common Section

Example:

PROGRAM MODULE A

	LABEL	OPERATION	
	MODA	START	
		.	
		.	
1.		COM	
	RECA	DC	Y(PROB)
	RECB	DC	C'DATE'
	AREA	DS	100CL800
	INFL	DS	CL80
	OUFL	DS	CL80
		.	
	G?FT	CSECT	
		.	
2.		COM	
	TJK	DC	CL5'SEVEN'
	JOBA	DS	CL80
	G?FT	CSECT	
		.	
		.	
		END	

PROGRAM MODULE B

	LABEL	OPERATION	
	MODB	START	
		.	
3.		COM	
	RECA	DS	Y(PROB)
	RECB	DC	C'DATE'
	AREA	DS	100CL800
	INFL	DS	CL80
	LIST	CSECT	
		.	
		.	
		END	

In module A, two common storage areas are defined by lines 1 and 2. The COM specified by line 2 is a continuation of the COM specified by line 1.

In module B, the COM specified by line 3 uses the common storage area initially defined by line 2.

NOTE:

Modules which contain sections defined by CSECT and COM directives with duplicate names must not be linked to one another. A blank label field is permitted as a legitimate label of spaces only once. When both CSECT and COM directives contain no labels, the label used for both is spaces; however, only one label of spaces is permitted.

4.6. LISTING CONTROL

One output of the assembler is a source and object code listing. An assembler directive is available to control the listing content.

4.6.1. Listing Content Control (PRINT)

The PRINT directive provides control over the listing content.

The following are available options:

- No listing
- Listing not including lines generated by macro instruction calls
- Listing including all lines

The format of the PRINT statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	PRINT	a

where:

a

Is ON,GEN to print the lines generated by a macro instruction.

ON, NOGEN to print the macro call, but not to print the lines generated by a macro instruction.

OFF to produce no listing, except for lines on which errors were detected.

One PRINT statement controls the listing of source and object statements until the next PRINT statement is detected among the source statements. The settings before the first PRINT statement are ON and GEN. The settings for operands omitted in a PRINT statement are not altered by the PRINT statement.

4.6.2. Listing Format Control (SPACE)

The SPACE directive provides control over the format of the listing.

The format of the SPACE statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	SPACE	a

where:

a

Is 0 or blank, which is equivalent to 1 line space.

1–999, the number of lines to be spaced over; form overflow terminates spacing.

Spacing continues until a form overflow condition occurs, at which point a home-paper command is issued and spacing is discontinued.

4.6.3. Listing Format Control (EJECT)

The EJECT directive provides control over the format of the listing.

The format of the EJECT statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	EJECT	unused

The EJECT statement causes a home-paper command to be issued, which results in the next line being a title line on a new page.

4.6.4. Listing Format Control (TITLE)

The TITLE directive provides control over the format of the listing.

The format of the TITLE statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	TITLE	'a'

where:

'a'

Is a string of up to 54 characters.

The title line is contained within apostrophes to permit centering of the title in the case of leading blanks. Two consecutive apostrophes are necessary to express an apostrophe within the body of the title. If the apostrophes are omitted, the operand field is still used as a title; however, the data is left-justified.

The TITLE statement permits identification of many pages of listing under a common title that appears at the top of each page. The operand of the TITLE statement is printed on all succeeding pages until a new TITLE statement is detected. When a TITLE statement is read, the assembler advances the printer form to the top of the next page.

4.7. INPUT CONTROL

The assembler provides optional sequence checking of source statements.

4.7.1. Input Sequence Control (ISEQ)

The ISEQ statement directs the assembler to perform a sequence check of the source statements on columns 76–80. The format of the ISEQ statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	ISEQ	unused

Sequence checking begins with the source statements following the ISEQ statement. It is terminated by an END statement.

Only one ISEQ directive can be in a source code module, and it must be the first statement in the source code following any macro definitions. The ISEQ directive has effect for source code only. In the absence of an ISEQ directive no sequence check is performed.

4.8. ERROR CONTROL

The assembler provides two indications of errors during an assembly.

4.8.1. User Program Sense Indicator (UPSI) Byte Setting

If an error is detected during assembly, the single bit of the UPSI byte (location X'117'), shown as 1 (00000100), is set to 1. This permits detection of errors by a subsequent program in the job stream or by a / SKIP job control card, providing a / JOB card does not intervene. The assembler never resets this bit to zero; therefore, any error in a series of assemblies may be tested for after the last assembly.

4.8.2. Total Error Count

A total error count is available on the last line of the assembly listing. This line always prints no matter which print options have been chosen. (See 7.4.1 for the format of this final line.)

5. Macro Instructions

5.1. MACRO INSTRUCTION FORMAT

Each source statement causes the SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler to take one specific action or represents a single constant or machine instruction. The assembler contains a library facility, where one source statement may result in the inclusion of many lines of code determined by a predefined pattern of code called a macro definition. The macro definition may be contained either in a library of definitions or included with the source statements of the program being assembled. The statement requesting the inclusion of the code is called a macro instruction. If a macro definition is included with the source statements, it must precede any macro instructions referring to it, precede the START card, and may not appear within the source code proper.

A macro instruction is similar in form to a source code instruction. It contains a label (optional), an operation code, and an operand consisting of none or one or more expressions separated by commas. The prime difference is that the macro instruction causes the generation of a series of source code instructions representing a number of assembler operations; whereas a source code instruction causes the assembler to do one specific operation.

The format for a macro instruction is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	code	$P_1,P_2,P_3,\ldots,P_n,N_1=P_{n+1},N_2=P_{n+2},N_3=P_{n+3},\ldots,N_m=P_{n+m}$

The label may be any symbol, but is not necessarily assigned the current value of the location counter. The operation is the name of the macro definition describing the pattern of code to be included. The operand, P_1 through P_{n+m}, is a sequence of expressions specifying parameters.

P_1 through P_n are positional parameters. P_{n+1} through P_{n+m} are keyword parameters. A macro instruction may have parameters of either or both types, or none at all.

5.1.1. Parameters

All positional parameters must be specified before any keyword parameters are specified. The order of the expressions in the operand determines the order of the parameters specified. Parameter specifications are separated by commas. When a positional parameter specification is omitted, the comma must be retained to indicate the omission. Thus, if a macro instruction contains positional parameters and the second is not specified, the operand appears as:

$P_1,,P_3$

If the third parameter is not specified, instead of the second, the operand is written

P_1,P_2

No trailing commas need be included.

The specification of a keyword parameter is:

N=P

where N is the name of the parameter (any symbol of seven or fewer characters is a legitimate keyword name) and P is the parameter specification (a value or a character string). Keyword parameter specifications are separated by commas; however, the comma need not be retained if the specification is omitted. A comma must be between the last positional parameter and the first keyword parameter. The order of the keyword parameter specifications is not significant. For example, if a macro definition contains three keyword parameters, the operand of the macro instruction might be:

$N_1=P_1,N_2=P_2,N_3=P_3$

or

$N_2=P_2,N_1=P_1,N_3=P_3$

and so on.

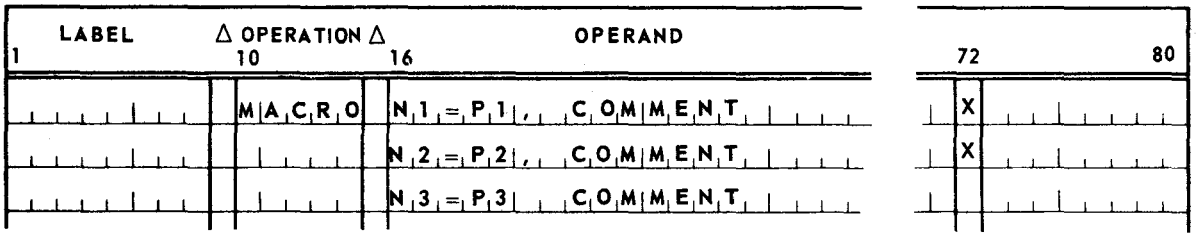
A macro instruction may contain positional and keyword parameters with commas separating the specifications. For example, the operand of a macro instruction with three positional and two keyword parameters might be:

$P_1,P_2,P_3,N_1=P_4,N_2=P5$

The number of parameters that may be specified with one macro instruction depends on how much space is required to store the specifications. One macro instruction may ordinarily specify as many as 50 parameters in its operand. When the operand overflows the space provided in one record, provision is made to continue the operand in the following record by putting a nonblank in column 72. The continuation of the operand begins with column 16. The macro pass of the assembler searches for a continuation record as soon as one of the two following events occurs:

- Information is taken from column 71 of the current record.
- A comma, followed by a space, is detected in the current record.

Columns 1 through 15 of a continuation record must be blank. If the information in a record is terminated prior to column 71 by a comma followed by a space, comments may be written after the space. For example, a macro instruction with three keyword parameters might be written as:



The specification of a parameter may not contain an equal sign or a comma, and may have a maximum of 127 characters.

5.2. WRITING MACRO DEFINITIONS

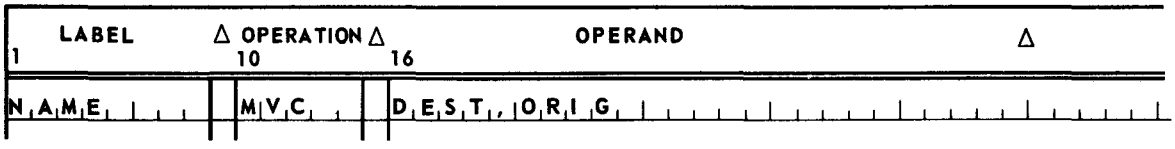
The routines for the macro library are written in standard assembler source code and entered in the library by the librarian. To distinguish one macro from another in the library, three directives are used: PROC, NAME, END.

- PROC directive

The first source code statement of a macro definition is a PROC directive in the format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	PROC	[operands]

The label may be any symbol, but is optional and, when used, the label in the macro instruction is substituted for the PROC label wherever the PROC label appears. For example, if the symbol MOVE were specified for the label of a macro instruction; and the label of the PROC directive of the associated macro definition were NAME, and an instruction within the macro definition were to contain the following line of source code,



then the source code generated by the definition would appear as:

If the PROC directive does not contain a label, but the macro instruction does, that label will remain undefined.

- NAME directive

The second line of a macro definition must be a NAME directive in the format:

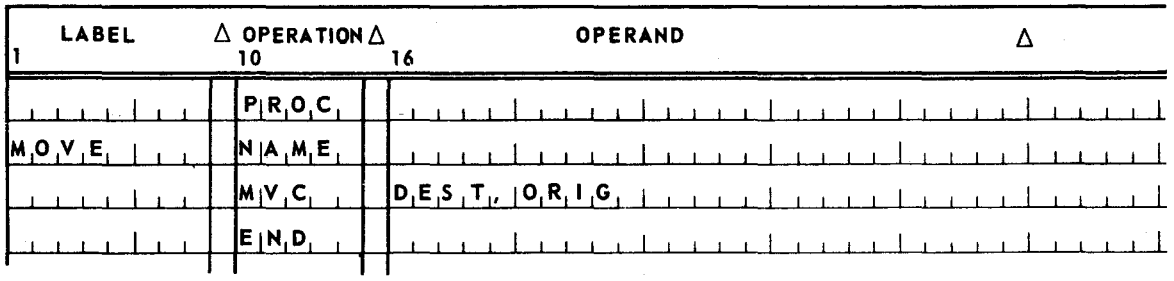
LABEL	Δ OPERATION Δ	OPERAND
symbol	NAME	[operand]

Symbol is the name by which the macro definition may be called, and is the name specified in the operation field of the macro instruction calling it. As many as eight characters may be in a label; the first must be alphabetic, the others alphanumeric.

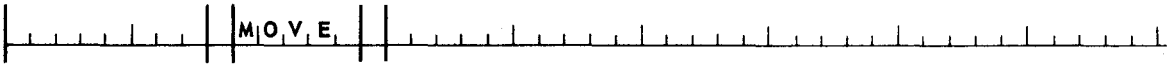
- END directive

The end of a macro definition is indicated by an END directive. It has no operand and requires no label.

If the following macro definition is in the library,



the macro instruction



is equivalent to the source code instruction:

Note that none of the directives (PROC, NAME, END) is output of the macro pass.

5.3. INCORPORATING PARAMETERS INTO MACRO DEFINITION CODING

The operand of a PROC directive, when used, is:

$p,n,N_1,N_2,N_3,\ldots,N_m$

The first expression, p, in the operand is a symbol used to address the parameters within the macro definition. This expression and its use are explained in this section. The second expression, n, is the number of associated positional parameters. $N_1,\ldots,N_m$ are the names of the keyword parameters. Any symbol of seven or fewer characters is a legitimate keyword name. Listing the keyword parameters in this way makes them positional parameters to the macro definition.

For example, if the PROC directive is the following format:

LABEL	Δ OPERATION Δ	OPERAND
unused	PROC	$p,3,N_1,N_2,N_3$

The macro definition contains three positional parameters, P_1, P_2, and P_3, and three keyword parameters, N_1, N_2, and N_3. Thus, the keyword parameters become, in effect, positional parameters P_4, P_5, and P_6.

The value specified for a parameter is substituted in the coding for the expression:

$p_{(n)}$

where p is the first expression in the PROC directive operand and n is the number of the positional parameter. The first is numbered 1, the second, 2, and so forth. As an example, if the following macro definition is in the library,

LABEL 1	△ OPERATION △ 10	OPERAND 16 △
	PROC	P,0,DEST,LGTH,ORIG
MOVE	NAME	
	MVC	P(1)(P(2)),P(3)
	END	

the macro instruction

	MOVE	DEST=OUT,LGTH=16,ORIG=IN
	or	
	MOVE	OUT,16,IN

is equivalent to the source code instruction.

	MVC	OUT(16),IN

A keyword parameter also is addressed by preceding its name with an ampersand. Thus, the MVC instruction within the macro definition of the previous example could have been written:

	MVC	&DEST(&LGTH),&ORIG

If a parameter whose value is not specified in the macro instruction is a complete term in an expression containing more than one term, it is replaced by the value zero; otherwise, it is replaced by the null character string (the character string that contains no characters).

5.4. NAME DIRECTIVE

More than one NAME statement may follow the PROC statement of a macro definition. (All the NAME statements must immediately follow the PROC statement.) Each such NAME statement specifies a different name by which the same macro may be called.

The object of giving more than one name is to permit reference to different versions of the procedure embodied in the macro definition. The versions are distinguished by the operands of the NAME statements.

Only one expression may appear in the operand of a NAME statement and may be assigned a value ranging from zero through 65,535. This expression is essentially a parameter of the macro and may be addressed within the macro definition as:

p(0)

where p is the first expression in the PROC statement operand; consequently, it may be used to distinguish between versions of a macro definition.

For example, if the following macro definition is in the library,

```
1     LABEL     Δ OPERATION Δ     OPERAND                    Δ
                10              16
                  PROC            P
MV4               NAME            4
MV8               NAME            8
                  MVC             DEST(P(0)),ORIG
                  END
```

the macro instruction

```
                  MV4
```

would produce the source code

```
                  MVC             DEST(4),ORIG
```

and the macro instruction

```
                  MV8
```

would produce:

```
                  MVC             DEST(8),ORIG
```

If a NAME statement contains no operand, the parameter p(0) is assigned a value of zero.

If a macro definition contains no parameters and it makes no reference to the operand of any of its NAME statements, its PROC statement has no operand.

If a macro instruction requests the inclusion of a macro definition of a given name, and, if one with such a name is included in the source statements of the program being assembled, it is included, even if one of the same name is in the macro definition library. The user may override a library macro definition with a macro definition of his own.

5.5. BUILT-IN MACRO DEFINITIONS

The following names must not be assigned to macro definitions because they are already assigned to macro definitions permanently built into the assembly structure:

CLOSE	READ	LBRET
CNTRL	SETL	ESETL
GET	TRUNC	EOJ
OPEN	WRITE	MSG
PUT	WAITF	CANCL
		FETCH

6. Conditional Assembly Instructions

6.1. GENERAL

The SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler recognizes certain directives that can exclude lines of coding from the output of the assembly, include a set of lines in the output of the assembly more than once, and can establish and alter values that may be used to determine whether a set of lines shall be included or excluded.

These directives are for use within a macro definition to control the pattern of generated coding, based on the parameters supplied by the macro instruction.

6.2. DO AND ENDO DIRECTIVES

A DO directive controls the inclusion or exclusion of the lines following it up to its associated ENDO directive. For example, in the following sequence of coding,

```
DO      1
        2
DO      3
        4
        5
ENDO    6
        7
        8
ENDO    9
```

the first ENDO directive is associated with the second DO directive and the second ENDO directive with the first DO directive. In other words, DO and ENDO directives are paired to produce nests. Thus, the first DO directive controls lines 2 through 8 and the second DO directive controls lines 4 and 5. DOs may be nested to a depth of 10.

The operand field of a DO statement contains a single expression. If the value of this expression is greater than zero, it represents the number of times the lines controlled by the DO statement are included in the output; otherwise these lines do not appear. For example, if the following macro definition is in the library,

LABEL	Δ OPERATION Δ	OPERAND
	PROC	P,0,ACT
MOVE	NAME	
	DO	P(1)
	MVC	DEST,ORIG
	ENDO	
	END	

the macro instruction

	MOVE	ACT=1

would produce the instruction

	MVC	DEST,ORIG

in the output of the macro pass; whereas, the macro instruction

	MOVE	ACT=0

would not produce the instruction.

Note that the macro instruction shown below also would suppress the instruction.

	MOVE	

A DO statement may contain a symbol in the label field, which may be used only in the statements controlled by the DO. Its value is 1 the first time these statements are generated, 2 the second time they are generated, and so on. If the DO is under the control of another DO and is reactivated, the count begins at 1 again.

6.3. GOTO AND LABEL DIRECTIVES

The format of the LABEL statement is:

LABEL	Δ OPERATION Δ	OPERAND
symbol	LABEL	unused

The symbol in the label field of the LABEL statement is not defined in the usual sense. It may be used only in the operand field of a GOTO statement.

The GOTO directive directs the assembler to another point in the macro definition in its production of source code.

The format of the GOTO statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	GOTO	symbol

The symbol in the operand field must be the label of a LABEL statement.

If the following macro definition is available,

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10           16
          PROC             P,,0,,FOUR
MOVE      NAME
          DO               P(1)
          MVC              DEST(4),,ORIG
          GOTO             END
          ENDO
          MVC              DEST(8),,ORIG
END       LABEL
          END
```

the macro instruction

```
          MOVE             FOUR=1
```

would produce the instruction

```
          MVC              DEST(4),,ORIG
```

and the macro instruction

```
          MOVE
```

would produce the instruction.

```
          MVC              DEST(8),,ORIG
```

If the GOTO statement is in a macro definition, the corresponding LABEL statement must appear in the same macro definition.

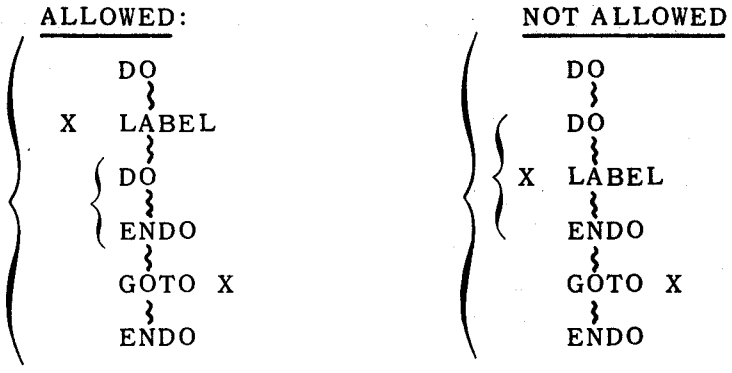
If a GOTO is within the range of statements controlled by a DO statement, but the addressed label is not, execution of the GOTO terminates the DO. Termination takes place whether the count, as expressed in the operand field of the DO statement, is exhausted or not.

NOTES:

1. *If a backward GOTO is itself within the range of a DO statement, the GOTO may not pass control to a LABEL that is outside the range of that same DO statement.*

2. *A LABEL that is the object of a backward GOTO may not reside within the range of a DO statement, unless the respective GOTO also resides in that same range.*

Both rules must be strictly observed to avoid unpredictable results.



6.4. CHARACTER EXPRESSIONS

A character expression is either a character value or a concatenation of character values.

A character value is a string of up to 127 characters enclosed in single apostrophes. Apostrophes within the character string must appear as two successive apostrophes. Ampersands within the character string must appear as two successive ampersands except for an ampersand that is the first character of a parameter or set variable reference.

6.5. SET VARIABLES

A set variable is a symbol to which a value is assigned during the generation of the code of a macro definition. Unlike an ordinary symbol, the value assigned to a set variable may be altered during the course of an assembly. A set variable may be either a local or a global variable. A global variable, once declared and given a value by a SET statement, remains defined throughout the assembly and retains the same value until that value is changed by another SET statement. A local variable is defined only within the macro definition in which it is declared. The value of a local variable within one macro definition is not affected by the declaration of a local variable with the same name in another macro definition.

Before a set variable can be set, it must first be declared by a GBL or an LCL directive. The symbol naming a set variable must consist of seven or fewer characters.

6.5.1. GBL Directive

The GBL statement format is:

LABEL	Δ OPERATION Δ	OPERAND
unused	GBL	symbol,...,symbol

Each symbol in the operand field of the GBL statement is declared to be the name of a global set variable. Each symbol must consist of seven or fewer characters.

6.5.2. LCL Directive

The LCL statement format is:

LABEL	Δ OPERATION Δ	OPERAND
unused	LCL	symbol,...,symbol

Each symbol in the operand field of the LCL statement is declared to be the name of a local set variable. Each symbol must consist of seven or fewer characters.

6.5.3. SET Directive

The SET directive assigns a value to a set variable. The format of the SET statement is:

LABEL	Δ OPERATION Δ	OPERAND
symbol	SET	expression

The symbol in the label field is the name of the global or local set variable to which a value is being assigned; the expression in the operand is the value to which the set variable is to be set. The value of the expression may range from zero through 65,535. Until a GBL or LCL variable is set by a SET directive, it has the following value:

- If referenced as a complete term in an expression containing more than one term, zero;
- otherwise, the null character string.

Once it has been set to a specific value by a SET directive, the set variable retains that value until it loses its declaration or is set to another value by another SET directive.

Declaring a set variable does not affect its value.

A set variable is addressed by preceding its name with an ampersand, except that if a local set variable has a name in the form L%xx, or if a global set variable has a name in the form G%xx, the set variable may be addressed by its name alone.

6.5.4. Relational and Logical Operators

Expressions in the operand field of all machine instruction and most assembler directives contain only the arithmetic operators add, subtract, multiply, and divide. The expression in the operand field of a SET or DO statement may contain the arithmetic operators, but it may, in addition, contain the following:

Relational Operators

greater than (>)

equal (=)

less than (<)

Logical Operators

logical AND (**)

logical OR (++)

The relational operators compare two (unsigned) binary numbers. The value of a relational expression is 1 if the relation is satisfied; otherwise, it is zero. Thus, if CHANNEL is the name of a keyword parameter, the expression

&CHANNEL=5

would have a value of 1 if 5 had been specified as a value of the parameter CHANNEL; if 5 had not been specified, &CHANNEL=5 would have a value of zero.

The logical operators treat a value as a bit string of 16 bits and produce a 16-bit value. The logical AND operator corresponds to the NC instruction and the logical OR operator corresponds to the OC instruction.

The precedence relation of the various operators in decreasing order is:

1. * /
2. + –
3. **
4. ++
5. >=<

Parentheses may be used to override the precedence relation. Parenthetical expressions may be nested to 14 levels.

6.5.5. Use of Character Expressions

Although set variables may have 2-byte binary values, they also may have character values of up to eight bytes. A set variable must not be assigned both types of values in one assembly.

A character expression also may appear as an operand of a relational operator. The following rules apply:

- A numeric value is greater than a character value.

- A character value is greater than any shorter character value.

A series of characters enclosed by separators is considered a character string. The following are separators:

```
& ( * > . ) / = Δ + ** < , - ++ '
```

A character string that matches a positional parameter designation or set variable name of the form L%xx or G%xx and is not enclosed in apostrophes will be accepted as the name of that set variable. Such a character string must be enclosed in apostrophes to be treated as a character value. If the operand field of a SET or DO statement contains an expression that could not otherwise be evaluated, that expression is treated as a character value even though it is not enclosed in apostrophes.

The following is an example of the use of a local set variable. If the following macro definition is in the library,

```
LABEL      Δ OPERATION Δ      OPERAND                    Δ
1          10              16
           PROC               P,,O,,ACT
MOVE       NAME
           LCL                &L%OO
&L%OO      SET                P(1)='YES'
           DO                 &L%OO
           MVC                DEST,ORIG
           ENDO
           END
```

the macro instruction

```
           MOVE               ACT=YES
```

would produce the source-code instruction.

```
           MVC                DEST,ORIG
```

Any other form of the MOVE macro instruction would suppress production of the source-code instruction.

The following is an example of the use of a global set variable. Assume the following two macro definitions are in the library.

```
LABEL      Δ OPERATION Δ          OPERAND                    Δ
1               10          16
           PROC          P,O,ACT
GIVE       NAME
           GBL           &G%OO
&G%OO      SET           P(1)='YES'
           DO            &G%OO
           MVC           DEST,ORIG
           ENDO
           END
           PROC
TAKE       NAME
           DO            &G%OO
           MVC           ORIG,DEST
           ENDO
           END
```

If the only macro instructions in the source statements for a particular assembly are the following,

```
           GIVE          ACT=YES
           TAKE
```

in the order shown, the following source code would be produced.

```
           MVC           DEST,ORIG
           MVC           ORIG,DEST
```

If the only macro instructions in the source statements for a particular assembly are the following,

```
           GIVE
           TAKE
```

no source code would be produced.

If the only macro instructions in the source statements are the following,

```
           TAKE
           GIVE          ACT=YES
```

the following source code would be produced.

```
     LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10              16
                 MVC           DEST,ORIG
```

Thus, the value of a global set variable is determined by the order of the macro instructions in the source statements.

6.6. CONCATENATION

A symbolic parameter or set variable with its leading ampersand may be concatenated with other characters, other symbolic parameters, and other set variables. If the name of a parameter or set variable is to be followed by another alphanumeric character, a left parenthesis, or a period, then a period must be inserted between the parameter or set variable name and the character that is to follow it. The result of this concatenation is the one obtained by replacing each parameter or set variable name and the period that follows it, if any, by the characters corresponding to that parameter or set variable. The following is an example of the use of concatenation. Assume the following macro definition is in the library:

```
                 PROC          P,0,NN,MM,LL
ZILCH            NAME
                 MVC           Z&NN(4),Z&MM
                 MVC           &LL.1(3),&LL.2
&LL.1            DS            CL3
&LL.2            DC            C'ABC'
```

The macro instruction

```
                 ZILCH         X,Y,Z
```

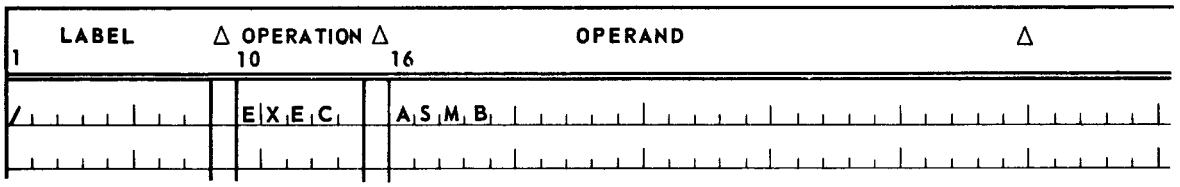
would produce the same code:

```
                 MVC           ZX(4),ZY
                 MVC           Z1(3),Z2
Z1               DS            CL3
Z2               DC            C'ABC'
```

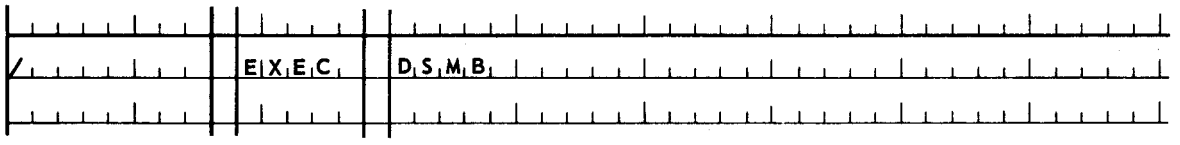
7. Preparation For Assembly

7.1. GENERAL

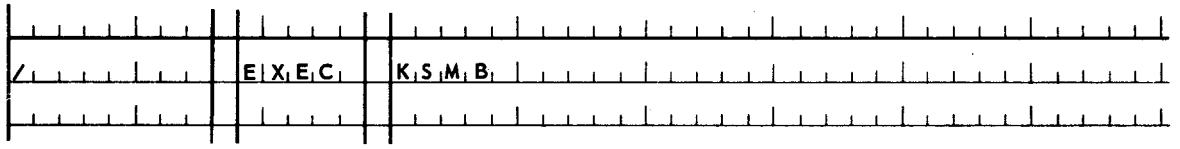
For an assembly to be performed by the SPERRY UNIVAC 9200/9300 Series Tape/Disc Assembler, certain cards are required to appear in the control stream. One card that is always required is an EXEC card in one of the following formats:



for the tape assembler;

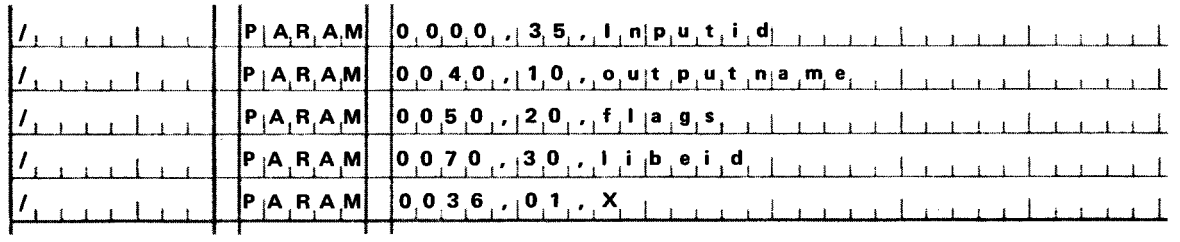


for the 8410 disc assembler;



for the 8411/8414 disc assembler.

This may be followed by any of the following PARAM cards in any order:



where:

inputid
Is groupname1/groupname2/ .../modulename if the source program is the module named "module name" located within a specific nest of groups on the input file; /modulename if the source program is the module named "module name" and selectivity based on the group structure is not required.

x
If source input is from tape, this parameter will cause the source input tape to stay open and allow stacked assemblies without rewinding the input tape; x is any nonblanks.

outputname
Is the name of the output module as it will be entered in the ELT record.

flags
Is a string of letters identifying the flags to be associated with the output module in its ELT record. If an asterisk (*) is used as the first character in the flags parameter, the option of rewinding logical unit 1 (rather than backspacing while using it as a scratch tape) is used. Stacked assemblies are still possible because the rewind option does not affect assembler use of the flags parameter in any way.

libeid
Is groupname1/ .../groupnameN identifying the nest of groups in the macro library within which the library search is to be performed. The macro library is located on logical unit zero for the tape assembler. For the disc assembler, it is assumed to be located in SYSFILE on logical unit zero, unless a PARAM 0131 card is used (7.2).

If the source program is in the control stream, PARAM cards must be followed immediately by a DATA card, which precedes the source program. This DATA control card must contain a T in column 16. In this case, the PARAM card defining inputid is omitted. If the PARAM card defining the name of the output module is omitted, the label of the START directive is taken as the output module name. The PARAM card defining the flags is omitted if no flags are to be associated with the output module. The PARAM card defining the limits of the library search is omitted if the entire macro library is searched. The name of the entire library is MACROLIB. If libeid is specified as the number 9, the assembler assumes that no calls requiring a library search are made in the program to be assembled.

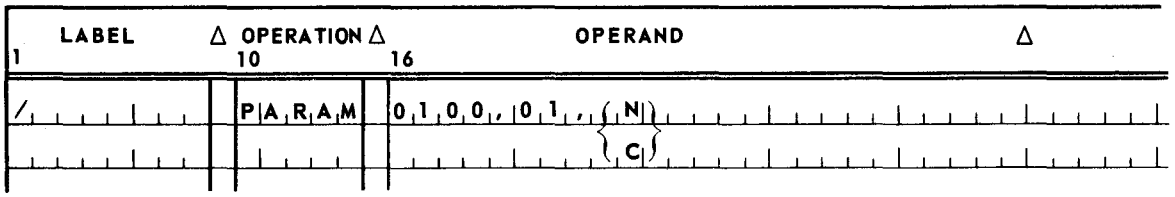
Macro definitions may be submitted with source code and must precede the source code in the input deck. When the assembler detects a macro instruction, it searches the macro definitions submitted with the source code before it searches the library. Each macro definition in the library should be preceded immediately by an ELT demarcation record. Also, the very first line of each macro definition must be the PROC directive.

7.2. GENERAL PROCEDURES FOR DISC ASSEMBLERS

In addition to the preceding PARAM cards, other PARAM cards apply specifically to the disc assembler. General notes pertaining to these cards are:

- Logical unit numbers may be 1- or 2-digit hexadecimal numbers between 0 and 3F.
- Filenames must contain a maximum of eight characters.
- The output file and the scratch file must be unique; that is, neither file may contain the same filename and logical unit as any other file. Neither file may be SYSFILE on logical unit zero.

This PARAM card is optional for 8410 disc units, and it is not used for 8411/8414 disc units. It is used to specify that all disc writes are to be verified (check read); if this PARAM card is omitted, no verification is made:

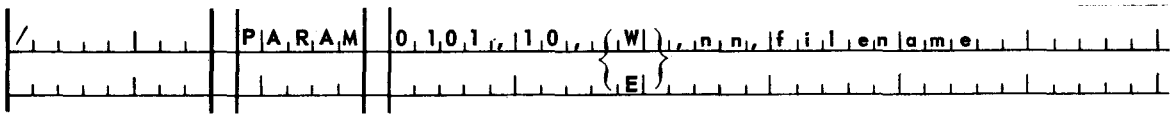


where:

N = Do not verify all disc writes.

C = Verify all disc writes.

This PARAM card is required because it defines the final output file:



where:

W

Specifies that the file is not to be extended; i.e., the object module is to be written at the beginning of the file.

E

Specifies that the file contains information to be preserved and extended; i.e., the object module is to be written immediately after all other modules contained in this file.

nn

Specifies the disc logical unit number in hexadecimal (0 through 3F).

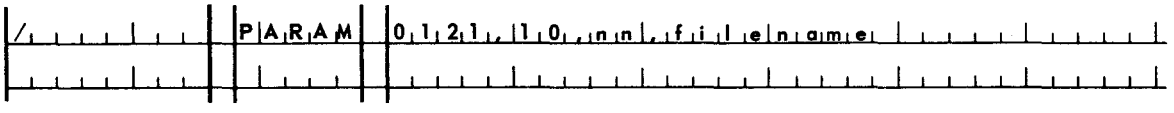
NOTE:

Logical unit numbers may be a 1- or a 2-digit number (n or nn).

filename

Specifies the name (maximum of eight characters) of this file.

This PARAM card is required and is used to define a file that is to be used as a scratch file if the source code is on cards, or is to be used as the input file if the source code is on disc. If this file is to be used as a scratch file, it may not be called SYSFILE if it is mounted on logical unit zero.



where:

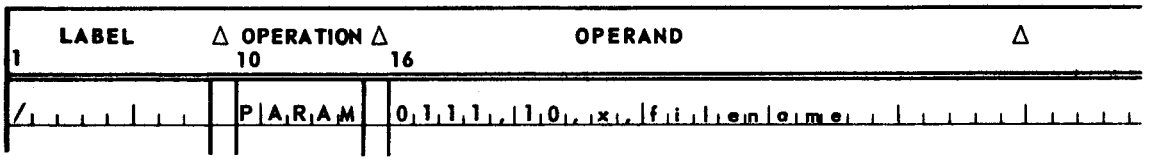
nn
Specifies the disc logical unit number in hexadecimal (0 through 3F).

NOTE:

Logical unit numbers may be a 1- or a 2-digit number (n or nn).

filename
Specifies the name of this file.

This PARAM card is required and defines a scratch file:



where:

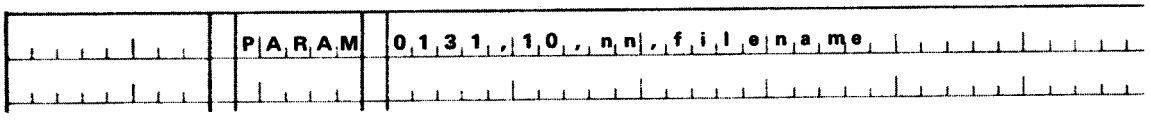
nn
Specifies the disc logical unit number in hexadecimal (0 through 3F).

NOTE:

Logical unit numbers may be a 1- or a 2-digit number (n or nn).

filename
Specifies the name of file.

This PARAM card is optional and is used to define the file in which the macro instruction library is stored:



where:

nn
Specifies the disc logical unit number in hexadecimal (0 through 3F).

NOTE:

Logical unit numbers may be a 1- or a 2-digit number (n or nn).

filename
Specifies the name of the file in which the macro library is stored.

If no PARAM card is specified, there is a default of 00,SYSFILE.

7.2.1. Disc File Organization

The disc assembler works with five files, represented as:

FILE A — The system file (normally on logical unit 0 and called SYSFILE) is used for locating macro definitions and fetching assembler overlays.

FILE B — The final output file; also used as an intermediate scratch (PARAM 0101).

FILE C — This is a scratch file (PARAM 0111).

FILE D — One file used, either as the input source code file, or, if the input is on cards, as a scratch file (PARAM 0121).

FILE E — If used, specifies the file in which the macro library resides (PARAM 0131).

It is recommended for efficiency of time that the following setup be used:

- For greatest efficiency, five separate discs can be used for the foregoing files.
- If less than five discs are used, the files may be set up as follows:

SYSFILE – FILE A	Normally on logical unit 0
Final output – FILE B	Set up on a different LU from file C
Scratch – FILE C	Set up on a different LU from file B
Input or scratch – FILE D	Set up on a different LU from file C and a different LU from the SYSFILE (FILE A)
MACRO library – FILE E	Set up on a different LU from file C

Some examples of disc drive assignments are shown in Figure 7–1.

2 drives	FILES	DRIVE 0 | 1 A | B C | D E
3 drives	FILES	DRIVE 0 | 1 | 2 A | B | C E | D
4 drives		DRIVE 0 | 1 | 2 | 3 A | B | C | D E

Figure 7–1. Disc Drive Assignment Examples

7.2.2. Allocation of File Space for Disc

A disc assembler user allocates space to his assembler files based on his needs and the type of assembly being run at any particular time. The sizes vary, depending on:

- the amount of space a user can afford at assembly time;
- the number of disc units available;
- whether the bulk of input is cards or disc; and
- whether macro are being submitted with source code.

The following discussion concerns the use of scratch files by the assembler. The available space in the final output file also is used as scratch space.

The assembler generates one internal item, and possibly more, for every source item processed. If the source item is a macro instruction, the item generation and required space will be expanded dynamically during generation of the instruction. Thus, since each case is different, the user can choose to allocate space based on each assembly or set up an all-inclusive allocation procedure to handle the largest assembly that he expects to turn out. In view of these possibilities the following guideline is proposed. It is intended as a guideline only and may or may not satisfy the requirements of a particular installation.

It is assumed that the reader has a knowledge of the disc types and is aware of the requirements and procedures necessary for the operation of the VTOC for the 8410 disc subsystem (see the 9200/9300 systems 8410 disc subsystem utility programs manual, UP-7668) or the DASM for the 8411 or 8414 disc subsystems (see the 9200/9300 systems 8411/8414 disc subsystems utility programs manual, UP-7835).

A typical, large single-phase program is the first pass of the assembler. This module comprises approximately 3000 source items. The disc I/O handler generates two items per disc record. Inasmuch as each source item causes the assembler to turn out an internal item, 6000 items are to be processed for this assembly. In addition to the SYSFILE, the assembler requires three work files to be used in the course of the assembly. Each file must contain two extents. The first extent consists of the file directory and the second consists of the file itself. Space allocation for the directory may be minimal and file allocation space is the most important consideration. The following space allocations were assigned in assembling the first pass of the assembler (with 3000 source records and no macro instructions) for each of the three files required:

- 8410 disc — 2 items per record, 100 sectors per track

  3000 records = 3000 sectors = 30 tracks

  For file directories, one track per file is allocated. Thirty tracks per file are allocated for file contents.

- 8411 disc — 2 items per record, 16 records per head

  3000 records = 200 heads = 20 cylinders

  One head per file is allocated for file directories, and 20 cylinders per file are allocated for file contents.

- 8414 disc — 2 items per record, 25 records per head

  3000 records = 120 heads = 12 cylinders

  One head per file is allocated for file directories. Twelve cylinders per file are allocated for file contents.

NOTE:

Although tthe allocation of directory space during the assembly may be minimal, should the same file be used for storage of program libraries at other times, the size of the file directory should be adequate for file usage.

7.3. GENERAL PROCEDURES FOR TAPE ASSEMBLER

The tape assembler requires that a scratch tape be available on logical unit 3, that its input, if on tape, be on logical unit 2, and that a tape be available for writing on logical unit 1. This tape must be either a rewound scratch tape or must be in library tape format positioned immediately past the tape mark. This tape is used for scratch as well as to receive the final output. If the tape units are not assigned properly, SWAP cards to effect this assignment must precede the EXEC card.

7.3.1. Restart Procedure for Final Phase of Assembly

After the assembler has begun producing its printed output, two restart options are available: stop printing and repeat printing.

7.3.1.1. Stop Printing Option

If it is necessary to abort the production of printed output because of forms misalignment, torn ribbon, or other printer malfunctions, the stop printing option may be invoked by the operator by setting the DATA ENTRY switches to any value from X'01' to X'OF' and pressing the OP REQ switch. The assembler ceases printing and, upon completion of writing the object module, restarts the printer listing from the beginning. It is suggested that, when the stop printing option is used, the printer be taken offline while the printer malfunction or forms alignment problem is being corrected.

7.3.1.2. Repeat Printing Option

If a second copy of the assembler printer listing is desired, the repeat printing option may be used by setting the DATA ENTRY switches to zero (X'0') and pressing the OP REQ switch. Upon completion of the first listing, a second listing will be produced.

NOTES:

1. *The foregoing operator requests may be made only after the assembler has begun printing. If made earlier, a 413F display occurs and the keyin is ignored.*

2. *An operator request with a value greater than X'OF' causes a 413F display, or is interpreted as a keyin to the operating system or to a symbiont.*

3. *The use of either option does not affect assembler production of a relocatable output module.*

7.4. PRINTER OUTPUT

When a PRINT directive requests a listing of all source code lines, the output listing is:

Print Column	Data
1–4	Sequence number of original input line; blank for macro-generated code
5	Blank
6–13	Error codes (eight allowable on any one line)
14	Blank
15–18	Address
19, 20	Blank
21–36	Object code (maximum of eight bytes or 16 digits printed)
37–40	Blank
41–120	Original source code line

The printer uses columns 6–13 to list error codes in the assembly listing. The codes used, and their meanings, are listed in Table 7–1.

Table 7–1. Printer Error Codes

Code	Meaning
C	Covering error
D	Duplicate label or reference to a duplicate label
E	Expression too large, improper syntax, or keyword parameter specified in a macro instruction not defined in the proc line of the macro definition
F	Upper main storage table full in pass 1 of assembler; too many DO statements to process in available main storage
H	Half-word boundary error
I	Instruction error, invalid operation code
L	Location counter value too large
M	More DOs than ENDOs
O	ORG error, second definition of a label
P	PROC error
R	Relocation error, terms too many or improper
S	Sequence error
T	Truncation of oversized term
U	Undefined label reference
X	Continuation error or ESID table overflow
Y	Assembly input ends with common section or dummy control section
Z	END card missing

7.4.1. Assembler Listing Print Format

The following are examples of the printer format for the page header and the last line of the assembler listing containing the total errors detected.

- Page Header Format

Print Column	Data
1–29	UNIVAC 9300 {TAPE / DISC} ASSEMBLY OF
30–37	progname-name in label field of START card
38–39	blank
40–93	title – inserted by TITLE directive 54 characters long
94	blank
95–98	DATE
99	blank
100–101	characters 1 and 2 of supervisor date
102	/
103–104	characters 3 and 4 of supervisor date
105	/
106–107	characters 5 and 6 of supervisor date
108–109	blank
110–113	PAGE
114	blank
115–117	page number
118–120	blank

- Final Line of Assembler Listing

Print Column	Data
1–39	same as PAGE HEADER format
40–49	blank
50–55	ERRORS
56	blank
57–61	error count – grand total of all errors in listing
62–70	blank
71–78	REVISION
79	blank
80–81	revision number
82–94	blank
95–109	same as for PAGE HEADER format
110–120	blank

NOTE:

The last line of every assembly listing contains total errors (which can be greater in count than the total of all the flags because it is only possible to print eight flags per line) and the revision number of the SPERRY UNIVAC software. This revision number should be referred to when reporting problems with the software.

7.5. DISPLAYS FOR DISC ASSEMBLER AND INPUT/OUTPUT

Displays on the control console associated with the disc assembler are listed in Tables 7–2 and 7–3.

Table 7–2. Disc Assembler Displays (Part 1 of 2)

Hexadecimal Display	Reason	Action
30x0	Logical unit x is down.	Press START to cancel.
30u1	The source code element on physical unit u does not exist as specified in PARAM 0000 card, or physical unit u does not contain any part of the macro library, or source code is to be introduced from the control stream, but no source is present.	Press START to cancel.

Table 7–2. Disc Assembler Displays (Part 2 of 2)

Hexadecimal Display	Reason	Action
	The following stops are due to bad parameter cards.	
3002	A W or an E was not specified on PARAM 0101.	Press START to cancel.
3003	A file name was not supplied.	Press START to cancel.
3004	A logical unit was not supplied or each character of the logical unit number was not in the range 0–9 or A–F.	Press START to cancel.
3005	The filename (other than disc source input file or macro library) was SYSFILE and was said to be on logical unit 0.	Press START to cancel.
3006	More than one file was specified by the same name and logical unit number.	Press START to cancel.

NOTE:

u = physical unit number.

Table 7–3. Disc Assembler Input/Output Displays

Hexadecimal Display	Reason	Action
14u1	8410 – unrecoverable abnormal line indication received from dispatcher 8411/8414 – software error	Key nonzero into location 4 to retry, or press START to cancel.
14u2	8410 – unrecoverable output bus check indication received from dispatcher 8411/8414 – hardware error	Same as 14u1
14u3	File cannot be opened because the VTOC header cannot be located or the file's format-1 label does not reside in area specified by the VTOC.	Press START to cancel.
14u4	No find on search	Same as 14u1
14u5	8410 – catastrophic failure indication received from dispatcher	Same as 14u1
14u6	Expiration date does not meet requirements	Press START to cancel.
14u7	8410 – nonoperational channel indication received from dispatcher	Same as 14u1
14u8	8410 – Invalid function indication received from dispatcher 8411/8414 – wrong record length	Same as 14u1
14u9	File extent limit reached; READ or WRITE attempted that references a point beyond physical extent for this file	Press START to cancel.
14uB	File directory filled to capacity	Press START to cancel.
14uD	Final output file described as to be extended (E) but next record pointers in format-1 label contain zeros	Key in nonzero to location 4 to treat the file as a new (W instead of E) file, or press START to cancel.
14uE	File on unit u not a library file; i.e., it does not have two extents	Press START to cancel.
14uF	8410 – an unload buffer command unsuccessfully issued	Same as 14u1

7.6. DISPLAYS FOR TAPE INPUT/OUTPUT ROUTINE

Displays on the control console associated with the tape assembler are listed in Table 7–4.

Table 7–4. Tape Assembler Input/Output Displays

Hexadecimal Display	Reason	Action
20u0	The end block was not found on physical unit u. This display occurs with respect to logical unit 1 when the assembler cannot find the END record on logical unit 1 preparatory to writing out its final output (Appendix C).	Press START to try again. Restart if unsuccessful.
20u1	The expiration date was not accepted on physical unit u.	Replace with correct tape and press START to try again, or key a nonzero into location 4 and press START to cancel.
20u5	The end-of-physical-tape (while writing) or a tape mark (while reading) was detected on physical unit u.	Press START to cancel.
20u6	A wrong-length block was read on physical unit u.	Press START to cancel.
20uD	Physical unit u is in a nonready condition.	Correct the problem at physical unit u and press START to continue.
20uE	The block searched for was not found on physical unit u.	Ensure that physical unit contains the correct tape. Key one of the following into location 4 and press START: 0 to try again, 1 to read forward, 2 to read backward. Restart if unsuccessful.
30u1	The source code element on physical unit u does not exist as specified in PARAM 0000 card, or physical unit u does not contain any part of the macro library, or source code is to be introduced from the control stream, but no source is present.	Press START to cancel.

NOTE:

u = physical unit number

8. Linker

8.1. GENERAL

Whether a program consists of one or more modules, the assembled module or modules must be combined into one executable program. This combining process is called linking and is performed by a utility program referred to as the linker.

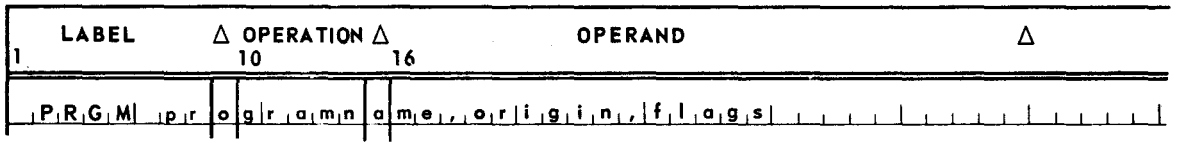
8.2. LINKER COMMANDS

Linker command cards may contain information in columns 1 through 77. Logical unit numbers are expressed in hexadecimal. Spaces are required in all linker command formats before and after the command's opcode and following the last parameter specified in the operand field.

8.2.1. PRGM

The PRGM command instructs the linker to begin construction of a normal, executable program. It provides the name to be associated with the program and the address at which the program is to begin in storage.

The format of this command is:

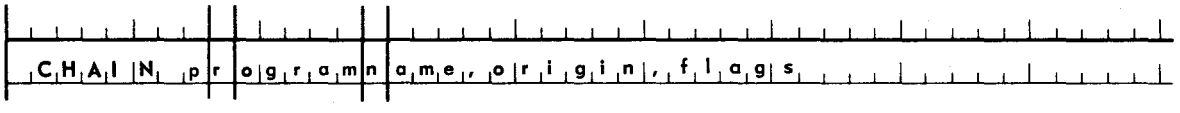


The absolute origin address may be expressed as a number or as an asterisk. In the latter case, the program is assigned the same origin as was assigned to the linker doing the linking.

8.2.2. CHAIN

The CHAIN command instructs the linker to begin the construction of a program which is one of the links in a chained job, but is otherwise a normal executable program.

The format of this command is:



8.2.2.1. Fetching

To get the next chained program, the program's name must be placed in locations X'136' – X'13D' by the previous program and a FETCH be performed or another / EXEC statement may be given to job control; e.g.,

```
LABEL      ΔOPERATIONΔ          OPERAND                    Δ
1          10                   16
/ EXEC     PRGM1
/ EXEC     CHAINED1
/ EXEC     CHAINED2
       or
MVC        X'136'(8),=CL8'CHAINED1'
FETCN      CHAINED1,,FORWARD
```

Each link of a chain must begin with a CHAIN command, and the links must be described to the linker in succession. The linker allocates to each link the maximum storage required by any one link and ensures that COMMON is assigned the same address in each link. A maximum of 10 consecutive chained programs is permitted. All programs consisting of a common section are given the same address when the chained programs are linked.

8.2.3. SYMB

The SYMB command instructs the linker to begin the construction of a symbiont. No overlays are allowed within a symbiont.

The format of the command is:

```
SYMB  symbiontname,symbiontnumber,flags
```

In the preceding commands (PRGM, CHAIN, and SYMB), the flags field is a character string. Permissible characters in the string are the 26 alphabetics and the multipunch, which, when translated to EBCDIC, yields the EBCDIC byte 11100001. (In Hollerith card code, this would be an 11-0-9-1 punch.)

The PRGM, CHAIN, and SYMB commands also produce on the output file an ELT record preceding the module to be constructed by the linker. The program or symbiont name becomes the module name (preceded by the names of the groups in which the module is contained). The flags determine the bits of the flag field (columns 6–9) of the ELT record in the following format:

*DHLPSWzAEIMQTXzBFJNRUYzCGKOxVZz

where:

*
: Is always 1.

x
: Represents the EBCDIC byte 11100001.

z
: Is always zero.

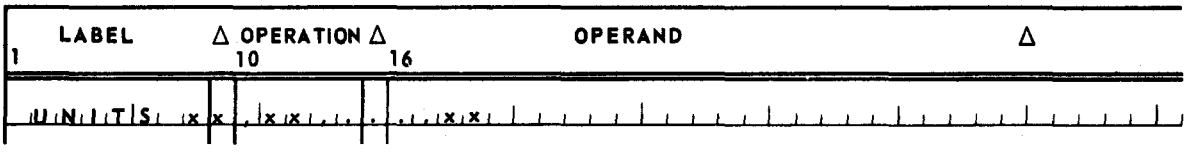
Comments, beginning with the first nonblank column after the flags field, from the PRGM, CHAIN, or SYMB command are placed in bytes 60–80 of the output ELT record.

The first command given to the linker must be the PRGM, the CHAIN, or the SYMB command; otherwise the linker does not allow further processing and cancels.

8.2.4. UNITS

The UNITS command, if present, must follow immediately a PRGM, CHAIN, SYMB, or another UNITS command. It instructs the linker to construct a list of logical unit numbers for insertion in the phase record of the program to be created.

The format of this command is:



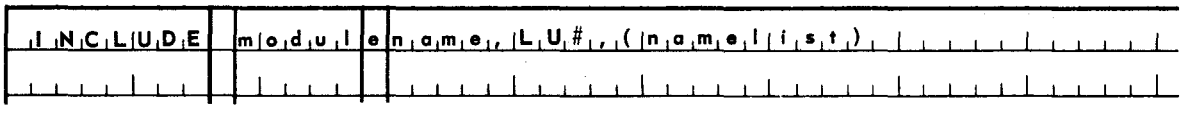
where:

Each xx is a 2-digit hexadecimal number, the logical unit number of a unit required by the program.

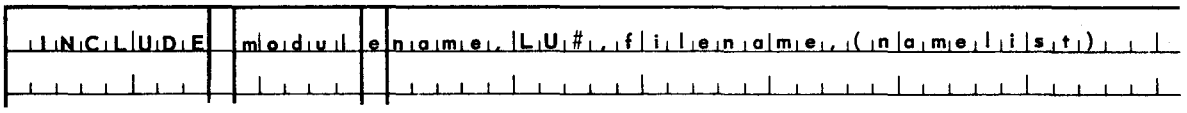
8.2.5. INCLUDE

The INCLUDE command instructs the linker to include a specified module or specified portions of a module in the current phase. The format of this command is:

for tape linker:



for disc linker:



If modulename is omitted (marked by a leading comma), the module is in the control stream immediately following the INCLUDE command; otherwise, the module is found on the indicated unit or filename. The logical unit number is expressed as one or two hexadecimal digits. If the logical unit number is omitted, the same unit is used as for the last INCLUDE with the indicated modulename. The initial value is 2 for the tape linker; for the disc linker, it is the logical unit number specified for the input file on the CTL card. For the disc linker, if the filename is omitted, the same filename is used as for the last INCLUDE with a module name indicated. The initial filename is the filename specified on the CTL card for the input file. If the tape linker is used and the logical unit number is specified, the tape is rewound before the search for the module begins. If the logical unit number is not specified, the tape is not rewound.

Modulename may consist of a single name preceded by a slash, in which case the indicated logical unit and/or file is read until a module with the specified name is found. This module then will be included; otherwise, the modulename must include the name of all the groups within which the element is contained, from outermost to innermost. Thus, if module MODLA is group GRP2, which is in group GRP1, the modulename is written as:

```
LABEL      Δ OPERATION Δ          OPERAND                    Δ
1                10        16
 INCLUDE  GRP1/GRP2/MODLA,LU#,(namelist)
```

If an INCLUDE command specifies an LU#, the tape linker rewinds the specified tape before beginning the search for the module.

The last parameter, if included, is a list of one or more control section names with the list enclosed in parentheses. Only the named control sections are included as a result of the INCLUDE command. If however, a module contains a common section, the common section is included when any control section of the module is included. A control section may be included in more than one phase. In such a case, all references to that control section are handled as follows:

- If the reference is in a phase in which the control section is included, the reference is linked to the copy of the control section in the phase.
- If the reference is in a phase in which the control section is not included, the reference is linked to the first copy of the control section included in the program.
- The common section is located immediately above the highest location of any other section coding or job control coding.

8.2.5.1. Sectioning (CSECT, DSECT)

A label in a CSECT section should be referenced only within its own section; otherwise, an ESID UND error occurs when linking that particular section. If necessary, to reference a label in a section other than the section needed, the programmer must include both sections when linking a particular phase to avoid ESID UND errors.

All ESIDs will be resolved if the following procedure is used:

Example (Assembly):

```
LABEL      ΔOPERATIONΔ      OPERAND                    Δ
1          10               16
TST        START    0
           USING    *,0
BGN        MSG      1
           BC       15,TAG
CON        DC       X'11'
CS2        CSECT
           MVC      BGN+7,0
TAG        CLI      CON,X'12'
           BC       3,BGN
           BC       15,TAG
TST        CSECT
           DC       X'00'
GO         DC       X'00'
           END      BGN
```

Example (Linker):

```
PRGM       TEST     ,*
INCLUDE    /TST,00,(TST,CSZ)
```

8.2.6. PHASE

The PHASE command instructs the linker to begin construction of a new phase and indicates how its origin is to be determined.

The format of this command is:

```
PHASE  phasename,origin
```

where:

phase name
Represents the name of the phase to be constructed and origin may be one of the following:

blank
Indicates the new phase will immediately follow the phase just completed.

nnnn

Is a decimal number or a hexadecimal number of the form X'nnnn' and specifies the origin of the phase.

symbol

Is a previously defined symbol (an entry point of one of the modules already included). The value assigned to this symbol is assigned as the origin of the phase.

symbol±n

Is symbol and n as defined previously. The value assigned to the symbol is modified as indicated and the result assigned as the origin of the phase.

(phasename list)

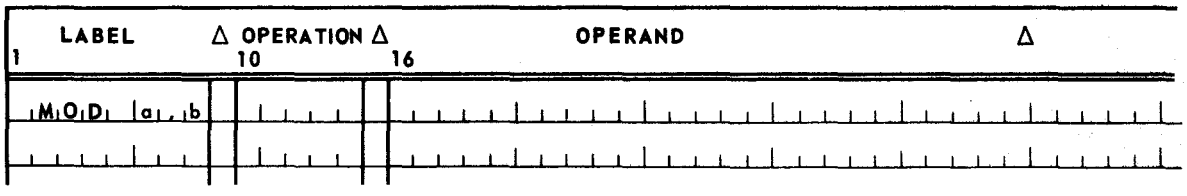
Is a list of one or more phase names separated by commas if more than one name is listed. The list is enclosed in parentheses. The phase follows the phase named in the list with whose highest allocated address is the greatest. The address allocated is a multiple of two.

A phase is terminated by the next PRGM, CHAIN, SYMB, PHASE, or END command. The entry point for a phase is the address from the first transfer card for a module in the phase. In a phase containing selected control sections of a module, the transfer card address is accepted only if its address is external to that module or contained in a control section included in the phase. In the absence of a transfer card with an acceptable address, the first address of the phase is used as an entry point. The maximum allowable number of phases is 20 to 30, depending upon the number of characters in the phase names.

8.2.7. MOD

The MOD (modular set) command instructs the linker to set the location counter to the value calculated from the operand specifications.

The format for this command is:



where:

a

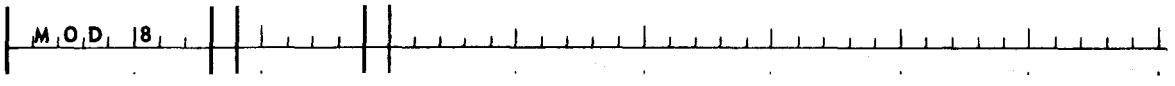
Must be a decimal expression with a value that is the result of 2 raised to a power.

b

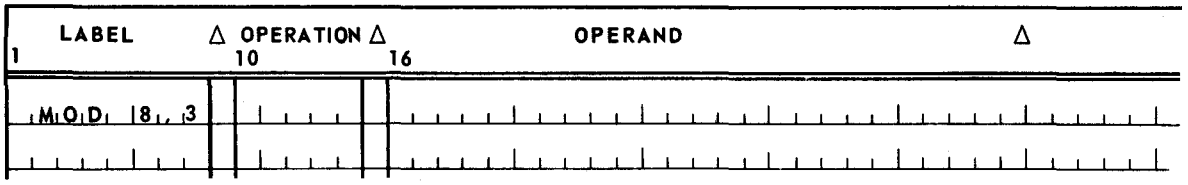
May be omitted, or it may be a decimal expression.

The location counter is set to the next number with the value of b, plus a multiple of a which is greater than, or equal to, the present value of the location counter.

Examples:



If the present value of the location counter 4024, the value is not modified. If the current value is 4025, it is modified to 4032.

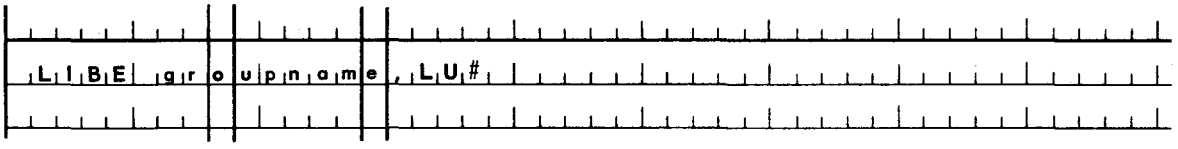


If the current value of the location counter is 4028, its value is modified to 4035.

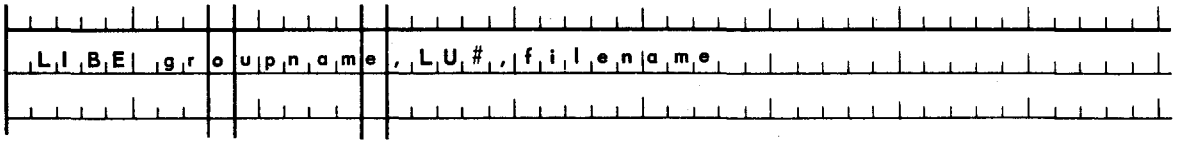
8.2.8. LIBE

The LIBE command instructs the linker to search for modules externally referenced within the modules being linked and not included via INCLUDE statements. The format of this command is:

for the tape linker:



for the disc linker:



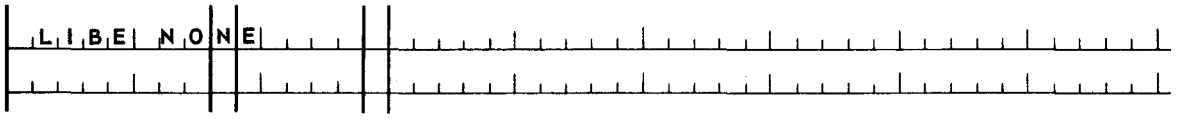
The search is limited to modules within the group named on the indicated logical unit or file. If the group to be searched is in another group, groupname must so indicate. Thus, if all modules in GRP3 are to be searched and GRP3 is in GRP2, which is contained in GRP1, groupname would be written:

GRP1/GRP2/GRP3

If LU# is omitted, the search is performed on the logical unit most recently named in a LIBE statement. Initially, the search is set to operate on logical unit 0 in group RLOCATBL.

For the disc linker, if filename is omitted, the filename most recently named in a LIBE statement is used. Initially, filename SYSFILE is used.

If no search is needed, the format is:



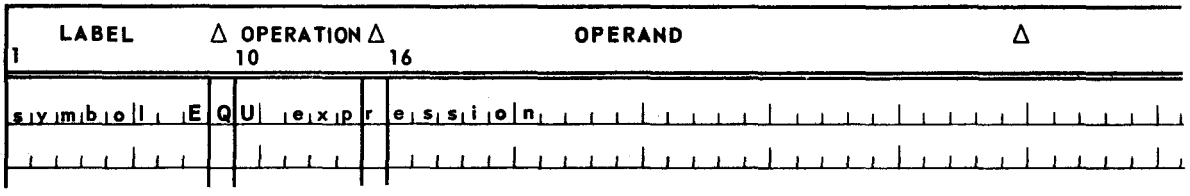
A LIBE command is effective for the program in which it appears and all programs until the next LIBE or END card. If several LIBE cards appear between two PRGM cards, only the last is effective. Routines included by the automatic search feature are included in the first phase of that program. They are not overlaid by any phase of the program because they are assigned load addresses beginning immediately after the highest address occupied by any phase of the program.

A module is included automatically only if no flags are associated with it or if all its flags also appear in the PRGM, CHAIN, or SYMB command for the program being linked.

8.2.9. EQU

The EQU command to the linker defines values not otherwise supplied in the normal linking process.

The format of this command is:



where:

symbol
Represents the name of the symbol, the value by which it is to be defined.

expression
Is any sum or difference of decimal values, hexadecimal values of the form X'nnnn', and previously defined symbols. A previously defined symbol is one that is an entry point of a module previously named on an INCLUDE card, or a symbol already defined by an EQU card.

8.2.10. END

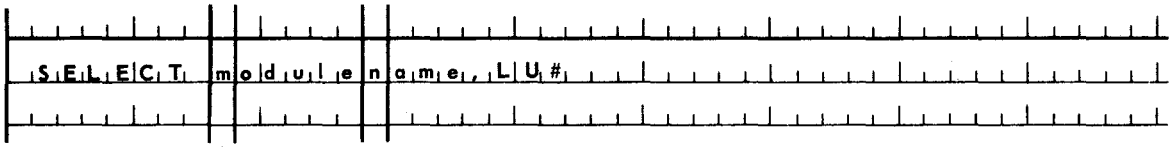
The END card (/* in columns 1 and 2) terminates the linker operation. To rewind logical unit 1 at the end of the tape linking, place a nonblank in column 11 of the card.

8.2.11. SELECT

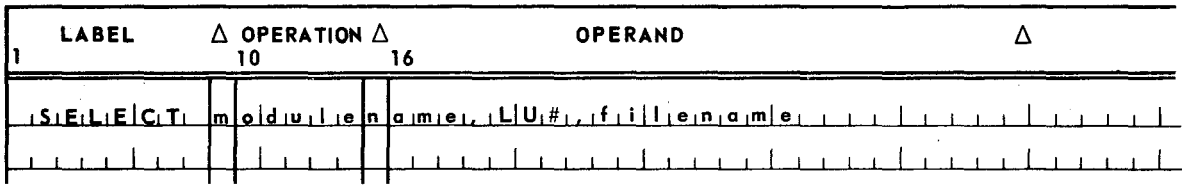
The SELECT command causes the linker to copy a module from an input file (or tape) to the final output file (or tape).

The format of this command is:

for the tape linker:



for the disc linker:



where:

modulename
Is the modulename or the groupname and modulename in the input ELT record of the module to be selected. The module may be source, relocatable, or object code. It must begin with an ELT record and be followed by a BOG, EOG, ELT, or END record.

LU#
Is the logical unit number of the tape or disc containing the input module to be selected.

filename
Is the name of the file containing the input module to be selected.

If no logical unit number is specified, the last logical unit number specified in a SELECT or INCLUDE command is used; if no previous logical unit number (and filename, for disc) was specified previously in a SELECT or INCLUDE command, the search is made on the initial logical unit number/filename. The initial logical unit number for the tape linker is 2; for the disc linker the initial logical unit number and filename are specified in the CTL card. The search for the named module begins in a forward direction from the current position, if tape.

If a logical unit number or filename is specified in a SELECT command, that designation remains selected until changed by a subsequent SELECT or INCLUDE command. The tape linker rewinds the tape on the specified logical unit before beginning the search for the named module.

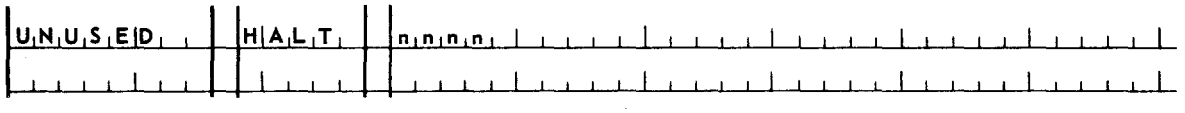
The SELECT command should be placed after the last INCLUDE command of a program or symbiont or after the last INCLUDE command of the last program of a chain of programs. SELECT terminates a chain, but another chain may follow.

When the linker detects a SELECT command, it treats the command as an indication that all commands for a program, a symbiont, or a chain of programs under construction have been submitted. After completion of the program construction, the linker locates the named module on the specified logical unit and copies the module on the output file. The copied module is preceded by an ELT record on the output file.

8.2.12. HALT

The HALT command permits the operator to take appropriate action during a link; e.g., changing input disc packs.

The format of this command is:



where:

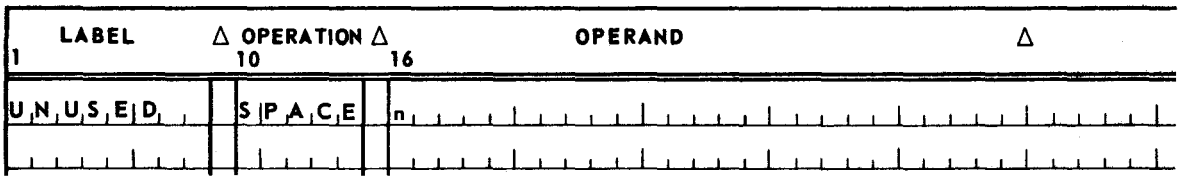
nnnn
Is any allowable hexadecimal character, 0 through F. The value is right-justified in the light display; that is, HALT 1 appears as 0001.

8.2.13. SPACE

The SPACE command controls the spacing of the listing by specifying SPACE 1 or SPACE 2. If an error is made in the number of spaces designated, the default of 1 space occurs.

This option is not available in the tape linker, which always produces a single-spaced listing.

The format of this command is:



where:

n
Is the number 1 or the number 2.

8.3. LINKER OPERATING INFORMATION

The procedures and displays used to operate the linker, the error messages produced on the printer, and an illustration of a sample deck are described in the following paragraphs.

8.3.1. Disc Linker Procedures

To execute the disc linker, the following control stream must be submitted:

```
/ EXEC     DL11  (DL00 for 8410 linker)
/ DATA     C
  CTL      (See 8.3.1.1 for parameters)
  .
  .
  .
  .        linker commands
/*
```

Disc packs containing files required by the linker must be mounted before executing the linker.

NOTES:

1. *If using 8410 disc drives, remove the file protect from disc on logical unit number 0 as DL00 uses the Fastband for scratch space.*

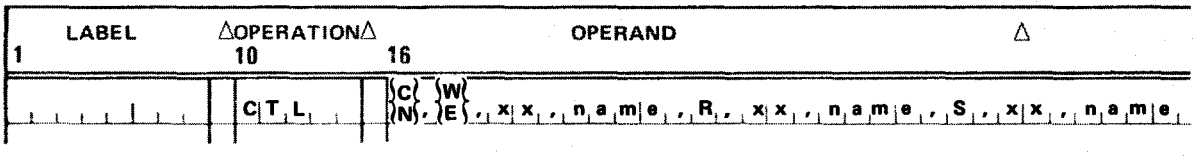
2. *File protect on logical unit number 0 is allowed on the 8411/8414 disc subsystem unless the scratch or final output file is located on unit 0.*

3. *Do not change output discs during linking on the 8411/8414 disc subsystems. With a careful choice of areas it is possible to change input disc packs as many times as required by use of the HALT command and pack substitution.*

8.3.1.1. CTL Card

Only one CTL card may be specified for the disc linker. The CTL card is mandatory and must follow the / DATA C card in the control stream.

The format of the card is:



where:

C
Output is to use check write.

N
Output is to use normal write, or unit is 8411/8414 disc drive.

W
Indicates final output file; output to start with beginning of file.

E
Indicates final output file; output to be written as an extension of the information already in the file.

R
Indicates first input file.

S
Indicates scratch file; this must be a library type file though the directory need only hold one record.

xx
Is a logical unit number; one or two hexadecimal digits.

name
Indicates filename.

No default option is permitted. Absolute control over the location of the linker output module is imperative. The control card is printed. Each file description must state all three designators; i.e., type of file, unit, and filename. All files used by the linker must be in library format; i.e., one directory extent and one data extent.

8.3.2. Tape Linker Procedures

To execute the tape linker, the following control stream must be sumitted:

```
/ EXEC TL00
/ DATA C
   .
   . (linker commands)
   .
/*
```

Appropriate tapes, as required by the linker, including the output tape on logical unit 1 and the scratch tape on logical unit 3, must be mounted before executing the linker.

8.4. PRINTER ERROR MESSAGES

The linker routine notifies the programmer of problem program errors. The messages that can be made by printer listing during a linker run are listed in Table 8–1.

The revision number of the SPERRY UNIVAC software is printed on the last line of output. The revision number must be referenced when reporting software problems.

8.4.1. User Program Sense Indicator (UPSI) Byte Setting

If an error is detected, during a link, the single bit of the UPSI byte (location X'117'), shown as a 1 (00001000), is set to 1. This permits detection of errors by a subsequent program in the job stream or by a / SKIP job control card, providing a / JOB control stream command does not intervene. The linker never resets this bit to zero; therefore, any error in a series of links may be tested for after the last link.

8.4.2. Total Error Count

A total error count is printed in the last line of the linker listing.

8.4.3. Suppression of Second Pass Printing

To cause the suppression of all printer output during the second pass of the linker (except page heading, error messages, and the total error count), place the following parameter card after the / EXEC card and in front of the / DATA card.

LABEL	△OPERATION△	OPERAND	△
1	10	16	
/	PARAM	0,0,0,1,,0,1,,1	

Table 8–1. Linker Printed Error Messages

Message	Meaning
NAME UND	Name is undefined.
PH NM UN	Phase name is undefined.
HEX NG	Hexadecimal value is not good.
ESID UND	Undefined ESID appears in relocatability data (RLD) on transfer card or text card.
NAME BLK	Name is blank.
OVERFLOW	Overflow of table area (any table).
SYMB NA	Symbiont is not allowed.
COMMA NA	Comma is not allowed.
EXPRESHN	Expression is in error.
OPRND NG	Operand is not good.
)MISSING	Right parenthesis is missing.
BAD UNIT	Physical unit is not allocated; logical unit is 1, 3, or greater than 7 for the tape linker, or is not in the system configuration.
HOLE CNT	The sum of the bytes from columns 8–72 of an input record did not agree with the checksum in column 7.
ABS MODL	Absolute module is not allowed.
ALR INCL	Module already is included.
CARDCNT	Card count error is in module.
UNEQU	Unequal value is on defined name.
COL 1?	Column 1 error is on EQU.
NO GROUP	Group specified on LIBE card is not in this file.
NO ELT	ELT record specified on INCLUDE card is not in the file.

8.5. DISPLAYS FOR DISC LINKER

The displays for the disc linker routine are listed in Table 8–2.

Displays 14u1 through 14uF for the disc linker are the same as displays 14u1 through 14uF, as shown in Table 7–3.

Table 8–2. Disc Linker Displays

Hexadecimal Display	Reason	Action
1F03	Wrong input image was read	Press START to ignore the image.
1F04	Card out of sequence	Press START to ignore the card.
1F05	Card hole count error	If the display occurs while reading an object module from the control stream, the card on which the error occurred is the second card from the top in the output stacker. To reread the error card, place it and all following cards (including the card in the wait station) in the input hopper, feed a card, and press START. If the display occurs while reading a tape or disc object module, pressing START may result in the generation of an invalid object module by the linker.
1FFF	Unrecoverable condition of missing or invalid phase name, invalid device, overflow message, or a PRGM, CHAIN, or PHASE command card is preceded by a command card which is not in proper sequence.	Press START to cancel.
4102	The first card read was a CTL card containing invalid specifications, or was not a CTL card.	Reinitialize the linker input deck with a valid CTL card as the first card in the reader, feed a card, and press START.

8.6. DISPLAYS FOR TAPE LINKER

The displays for the tape linker are listed in Table 8–3. Displays 20u0 through 20uE for the tape linker are the same as displays 20u0 through 20uE as shown in Table 7–4.

Table 8–3. Tape Linker Displays

Hexadecimal Display	Reason	Action
1554	Cannot read from logical unit 3 to load pass 1.	Press START to cancel.
1555	Cannot read from logical unit 1 to load pass 2.	Press START to cancel.
1F03	Wrong input image was read.	Press START to ignore the image.
1F04	Card out of sequence	Press START to ignore the card.
1F05	Hole count error	See action for disc linker display 1F05.
1FFF	Unrecoverable condition of missing or invalid phase name, invalid device, overflow message, or a preceding 1555 display.	Press START to cancel.

Appendix A. Tape Language Processor Conventions

Certain processor conventions are used consistently by the language processors such as the assembler, the linker, the FORTRAN compiler, and the COBOL compiler. The following describe these conventions.

Each processor produces its output on logical unit 1. The system is assumed to be on logical unit 0. Each processor, at the beginning of its operation, must test to see if the tape on logical unit 1 is at load point. If the tape is not at load point, no action is taken. If it is, the processor must read past any volume headers. The processor must then check the HDR1 record expiration date against the date stored in the supervisor. If the HDR1 date is less than or equal to the date stored in the supervisor, or if there is no HDR1 label, the processor rewinds the tape.

A processor using logical unit 1 as a scratch must record an END record indicating block number 1 and zero group levels if that tape is at load point. Following this, a tape mark is recorded on tape. If the tape is not at load point, it is assumed that it is positioned immediately past the tape mark that follows an END record. This tape then may be used by the processor for scratch. When the processor is ready to write its output, it must back space past the tape mark. It must then read the END record backward to pick up the block count and group names it will use in producing its output. It then overwrites the END record in producing its first block of output with the block number that had been in the end record.

If a processor does not use logical unit 1 for scratch, it may postpone its load point test until it is ready to write its output. In this case, it would not need to write the END record and tape mark, and immediately backspace and overwrite them.

When a processor that has found logical unit 1 at load point is ready to write on logical unit 1, it must first read past any volume headers then write an HDR1 label with an expiration date of zeros before producing any output of its own.

When the processor finishes producing its output, it writes an appropriate END record and tape mark on logical unit 1 following the output. The tape is not rewound.

If a scratch tape is at load point, a processor reads past any volume headers. It checks the HDR1 record expiration date against the date stored in the supervisor. If the HDR1 date is larger, it rewinds the tape with interlock and gives a standard error display. It tries again if START is pressed. If the HDR1 date is less than or equal to the date stored in the supervisor, it writes a tape mark and the remainder of tape is available for scratch. If there is no HDR1 label, it writes one with an expiration date of zeros, followed by a tape mark. The rest of the tape is then available.

Appendix B. Standard Card, EBCDIC, and Printer Graphic Codes

SPERRY UNIVAC 9200/9300 Series software is designed to use the standard card, internal, and printer graphic codes as shown in Table B–1. These codes are based on a byte divided into a zone and digit portion, each containing four bits, as:

zone digit

0123	4567	bits

In Table B–1, the zone and digit bits of the byte are the matrices; for each position the Hollerith code and the printer graphic, if any, are given.

Table B–1. Standard Codes (Part 1 of 4)

(Two Most Significant Bits of Zone 00)

Digit	Two Least Significant Bits of Zone			
	00	01	10	11
0000	12-0-9-8-1	12-11-9-8-1	11-0-9-8-1	12-11-0-9-8-1
0001	12-9-1	11-9-1	0-9-1	9-1
0010	12-9-2	11-9-2	0-9-2	9-2
0011	12-9-3	11-9-3	0-9-3	9-3
0100	12-9-4	11-9-4	0-9-4	9-4
0101	12-9-5	11-9-5	0-9-5	9-5
0110	12-9-6	11-9-6	0-9-6	9-6
0111	12-9-7	11-9-7	0-9-7	9-7
1000	12-9-8	11-9-8	0-9-8	9-8
1001	12-9-8-1	11-9-8-1	0-9-8-1	9-8-1
1010	12-9-8-2	11-9-8-2	0-9-8-2	9-8-2
1011	12-9-8-3	11-9-8-3	0-9-8-3	9-8-3
1100	12-9-8-4	11-9-8-4	0-9-8-4	9-8-4
1101	12-9-8-5	11-9-8-5	0-9-8-5	9-8-5
1110	12-9-8-6	11-9-8-6	0-9-8-6	9-8-6
1111	12-9-8-7	11-9-8-7	0-9-8-7	9-8-7

Table B–1. Standard Codes (Part 2 of 4)

(Two Most Significant Bits of Zone 01)

Digit	Two Least Significant Bits of Zone			
	00	01	10	11
0000	 Δ	12 &	11 –	12-11-0
0001	12-0-9-1	12-11-9-1	0-1	12-11-0-9-1
0010	12-0-9-2	12-11-9-2	11-0-9-2	12-11-0-9-2
0011	12-0-9-3	12-11-9-3	11-0-9-3	12-11-0-9-3
0100	12-0-9-4	12-11-9-4	11-0-9-4	12-11-0-9-4
0101	12-0-9-5	12-11-9-5	11-0-9-5	12-11-0-9-5
0110	12-0-9-6	12-11-9-6	11-0-9-6	12-11-0-9-6
0111	12-0-9-7	12-11-9-7	11-0-9-7	12-11-0-9-7
1000	12-0-9-8	12-11-9-8	11-0-9-8	12-11-0-9-8
1001	12-8-1	11-8-1	0-8-1	8-1
1010	12-8-2 ¢	11-8-2 !	12-11	8-2 :
1011	12-8-3 .	11-8-3 $	0-8-3 ,	8-3 #
1100	12-8-4 <	11-8-4 *	0-8-4 %	8-4 @
1101	12-8-5 (	11-8-5)	0-8-5 –	8-5 '
1110	12-8-6 +	11-8-6 ;	0-8-6 >	8-6 =
1111	12-8-7 |	11-8-7 ¬	0-8-7 ?	8-7 "

Table B–1. Standard Codes (Part 3 of 4)

(Two Most Significant Bits of Zone 10)

Digit	Two Least Significant Bits of Zone			
	00	01	10	11
0000	12-0-8-1	12-11-8-1	11-0-8-1	12-11-0-8-1
0001	12-0-1	12-11-1	11-0-1	12-11-0-1
0010	12-0-2	12-11-2	11-0-2	12-11-0-2
0011	12-0-3	12-11-3	11-0-3	12-11-0-3
0100	12-0-4	12-11-4	11-0-4	12-11-0-4
0101	12-0-5	12-11-5	11-0-5	12-11-0-5
0110	12-0-6	12-11-6	11-0-6	12-11-0-6
0111	12-0-7	12-11-7	11-0-7	12-11-0-7
1000	12-0-8	12-11-8	11-0-8	12-11-0-8
1001	12-0-9	12-11-9	11-0-9	12-11-0-9
1010	12-0-8-2	12-11-8-2	11-0-8-2	12-11-0-8-2
1011	12-0-8-3	12-11-8-3	11-0-8-3	12-11-0-8-3
1100	12-0-8-4	12-11-8-4	11-0-8-4	12-11-0-8-4
1101	12-0-8-5	12-11-8-5	11-0-8-5	12-11-0-8-5
1110	12-0-8-6	12-11-8-6	11-0-8-6	12-11-0-8-6
1111	12-0-8-7	12-11-8-7	11-0-8-7	12-11-0-8-7

Table B–1. Standard Codes (Part 4 of 4)

(Two Most Significant Bits of Zone 11)

Digit	Two Least Significant Bits of Zone			
	00	01	10	11
0000	12-0	11-0	0-8-2	0 0
0001	12-1 A	11-1 J	11-0-9-1	1 1
0010	12-2 B	11-2 K	0-2 S	2 2
0011	12-3 C	11-3 L	0-3 T	3 3
0100	12-4 D	11-4 M	0-4 U	4 4
0101	12-5 E	11-5 N	0-5 V	5 5
0110	12-6 F	11-6 O	0-6 W	6 6
0111	12-7 G	11-7 P	0-7 X	7 7
1000	12-8 H	11-8 Q	0-8 Y	8 8
1001	12-9 I	11-9 R	0-9 Z	9 9
1010	12-0-9-8-2	12-11-9-8-2	11-0-9-8-2	12-11-0-9-8-2
1011	12-0-9-8-3	12-11-9-8-3	11-0-9-8-3	12-11-0-9-8-3
1100	12-0-9-8-4	12-11-9-8-4	11-0-9-8-4	12-11-0-9-8-4
1101	12-0-9-8-5	12-11-9-8-5	11-0-9-8-5	12-11-0-9-8-5
1110	12-0-9-8-6	12-11-9-8-6	11-0-9-8-6	12-11-0-9-8-6
1111	12-0-9-8-7	12-11-9-8-7	11-0-9-8-7	12-11-0-9-8-7

Comments concerning this manual may be made in the space provided below. Please fill in the requested information.

System: _____

Manual Title: _____

UP No: _____ Revision No: _____ Update: _____

Name of User: _____

Address of User: _____

Comments:

CUT

FOLD

FIRST CLASS
PERMIT NO. 21
BLUE BELL, PA.

BUSINESS REPLY MAIL NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

SPERRY UNIVAC

P.O. BOX 500
BLUE BELL, PA.
19422

ATTN: SYSTEMS PUBLICATIONS DEPT.

FOLD

CUT