SPERRY UNIVAC COMPUTER SYSTEMS

PUBLICATIONS REVISION

Operating System/4 (OS/4)

Assembler Programmer Reference

UP-7935 Rev. 1

This SPERRY UNIVAC™ Operating System/4 (OS/4) Library Memo announces the release and availability of "SPERRY UNIVAC Operating System/4 (OS/4) Assembler Programmer Reference," UP-7935 Rev. 1. This is a Standard Library Item (SLI).

This revision reflects the current version of the OS/4 assembler at the time of publication. Various technical corrections and additions have been made throughout this manual.

This revision merges the information contained in the "UNIVAC 9400 System Assembler/Central Processor Unit Programmer Reference," UP-7600 and the "UNIVAC 9700 System OS/4 Assembler Programmer Reference," UP-7935 to provide one OS/4 Assembler manual for the SPERRY UNIVAC 9400, 9480, 90/60, and 90/70 Systems.

Note that the title of the manual has been changed to reflect the software system rather than the hardware system.

Section 6 of this revision describes floating-point instructions. These do not apply to SPERRY UNIVAC 9400/9480 System users. Throughout other sections, those instructions that apply to SPERRY UNIVAC 90/60,70 Systems only are so noted. Also where necessary in the description of an instruction, operational differences between the 9400/9480 OS/4 Assembler and the 90/60,70 OS/4 Assembler are given.

Appendix B describes hardware differences between 9400/9480 and 90/60,70 Systems.

Destruction Notice: This revision supersedes and replaces the following:

- "UNIVAC 9400 System Assembler/Central Processor Unit Programmer Reference," UP-7600, released on UNIVAC 9400 System Library Memo 4 dated September 30, 1968 and associated update packages.

  NOTE: Section 2 of UP-7600 has been superseded by "SPERRY UNIVAC 9400/9480 Systems Processors Programmer Reference," UP-8080, released in July, 1974.

- "UNIVAC 9700 System OS/4 Assembler Programmer Reference," UP-7935, released in August, 1972. Please destroy all copies of UP-7600, UP-7600-A, UP-7600-B, UP-7600-C, UP-7600-D, and their Library Memos and UP-7935 and its Library Memo.

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

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS:
Mailing Lists 217, 630, and 692	Mailing Lists 60, 61, 65, and 66 (Covers and 379 pages)	Library Memo for UP-7935 Rev. 1
		RELEASE DATE: May, 1975

SPERRY UNIVAC
Operating System/4 (OS/4)
Assembler

Programmer Reference

SPERRY UNIVAC COMPUTER SYSTEMS

UP-7935 Rev. 1

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-7935 Rev.

Part/Section	Page Number	Update Level
Cover/Disclaimer		
PSS	1	
Contents	1 thru 13	
1	1 thru 14	
2	1 thru 13	
3	1 thru 8	
4	1 thru 40	
5	1 thru 20	
6	1 thru 53	
7	1 thru 49	
8	1 thru 16	
9	1 thru 15	
10	1 thru 16	
11	1 thru 14	
12	1 thru 28	
13	1 thru 26	
14	1 thru 5	
Appendix A	1 thru 6	
Appendix B	1 thru 4	
Appendix C	1 thru 7	
Appendix D	1 thru 4	
Appendix E	1 thru 6	
Appendix F	1 thru 3	
Index	1 thru 16	
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

PAGE STATUS SUMMARY

CONTENTS

11. DATA AND STORAGE DEFINITION

12. ASSEMBLER DIRECTIVES

TABLES

1. Introduction

1.1. GENERAL

This manual provides the basic information necessary for programming in assembly language for the SPERRY UNIVAC Operating System/4 (OS/4). Information is presented concerning data representation, instruction coding, constant and storage definition, assembler directives, conditional assembly, assembler procedures, and assembler error messages.

1.2. ASSEMBLER CHARACTERISTICS

The assembler is an efficient software aid designed to handle most programming problems encountered by the user. Each machine instruction and data form has a simple, convenient representation in assembly language. The assembler translates this language into a form which can be executed by the computer. The rules governing the use of the language are not complex and are easily applied by the programmer.

A program is written on a coding form in assembly language. The information on the form is then keypunched to produce source code cards (the source deck). The source deck is read by the assembler and a relocatable object module and printer listing are produced. The object module is then linked to other object modules to form a load module suitable for loading and execution.

This manual describes the operational characteristics of the OS/4 assembler and the use of assembly language. These characteristics are:

- Mnemonic Operation Codes

  A fixed mnemonic code, consisting of from one to four letters, is assigned to each machine instruction; each code suggests the nature of the instruction. As a further aid in writing branch-on-condition instructions, separate mnemonic codes are provided for each condition. Restrictions concerning the use of these mnemonic codes are described in 13.3.2.

- Flexible Data Representation

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

- Symbolic Addressing and Storage Assignment

  Symbolic labels can be assigned to instructions or groups of data. An instruction then references the labeled data by label rather than by main 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 main storage locations used for a program, assigns all incoming instructions and data to specific locations, and performs base register and displacement calculations.

- Assembler Directives

  The assembler includes a set of directives which specify instructions regarding the operation of the assembler itself. These directives allow the user to control program sectioning, base register assignment, output listing format, sequence checking, and other auxiliary functions. Restrictions concerning the use of assembler directives are described in 13.3.2.

- Conditional Assembly

  The assembler provides a set of directives which permit the user to specify the order of source statement generation, exclude sections of code, include a set of lines in the output of the assembly, and vary the content of generated statements.

- Relocatable Programs and Program Linking

  The assembler produces object modules in relocatable form. In this form, the actual storage locations to be occupied by a program need not be specified at assembly time, but are determined when the program is loaded. Provisions are made for linking, loading, and executing as one program the results of separate assemblies, thereby making more efficient use of machine time. The input to one assembly can be divided into separate sections, each consisting of a group of instructions or data occupying contiguous locations. The relative positions of the sections can be declared at the time the program is linked.

- Macro Facility

  The assembler macro facility can reduce the effort required to write patterns of code which are repeated in a program or shared by several programs. One instruction to the assembler can result in the inclusion in the object program of many instructions and constants, or can result in establishing one or more values for use elsewhere in the program. A macro may be defined so that the pattern of coding generated can vary widely depending upon the parameters supplied in the calling macro instruction.

- Program Listing

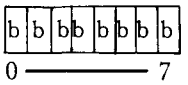
  A printed listing of source and object codes is one output of the assembler. This listing includes error message flags marking any errors detected by the assembler. Source code errors do not halt the assembly. The assembler processes the remainder of the source code and performs its usual error checks,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

- Compatibility

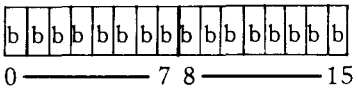
  Source programs and macro definitions which are written for the UNIVAC 9400/9480 Systems but which are to be executed in a UNIVAC 90/60,70 environment may require modification before being acceptable to the assembler.

1.3. DATA FORMATS

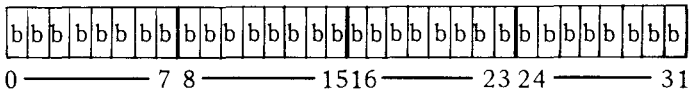
Main storage locations are numbered consecutively. Each address specifies one byte of information. Every time a storage request is made, four adjacent locations are accessed; therefore, data (depending on the length) is accessed in groups of four consecutive bytes. The address of a group of bytes is the address of the leftmost byte of the group. The bits in a byte are also numbered from left to right, starting with zero.



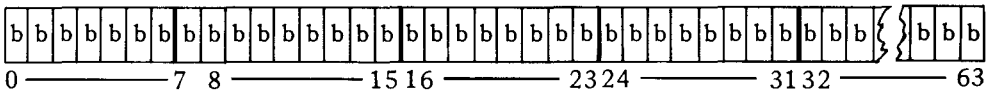
Half-word data formats consist of two consecutive bytes.



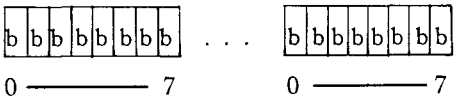
Full-word data formats consist of four consecutive bytes.



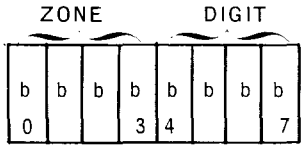
Double-word data formats consist of eight consecutive bytes.



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



It is possible to store 256 different bit combinations in the byte. Thus, 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 as 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.



1.3.1. Fixed-Point Numbers

Each fixed-point number is represented in one of three fixed-length formats composed of a single sign bit followed by an integer field. When the sign bit is 0, the integer represents a positive value; when 1, the integer represents a negative value. Negative integers are represented in twos complement notation. The half-word, full-word, and double-word formats are shown in Figure 1–1.

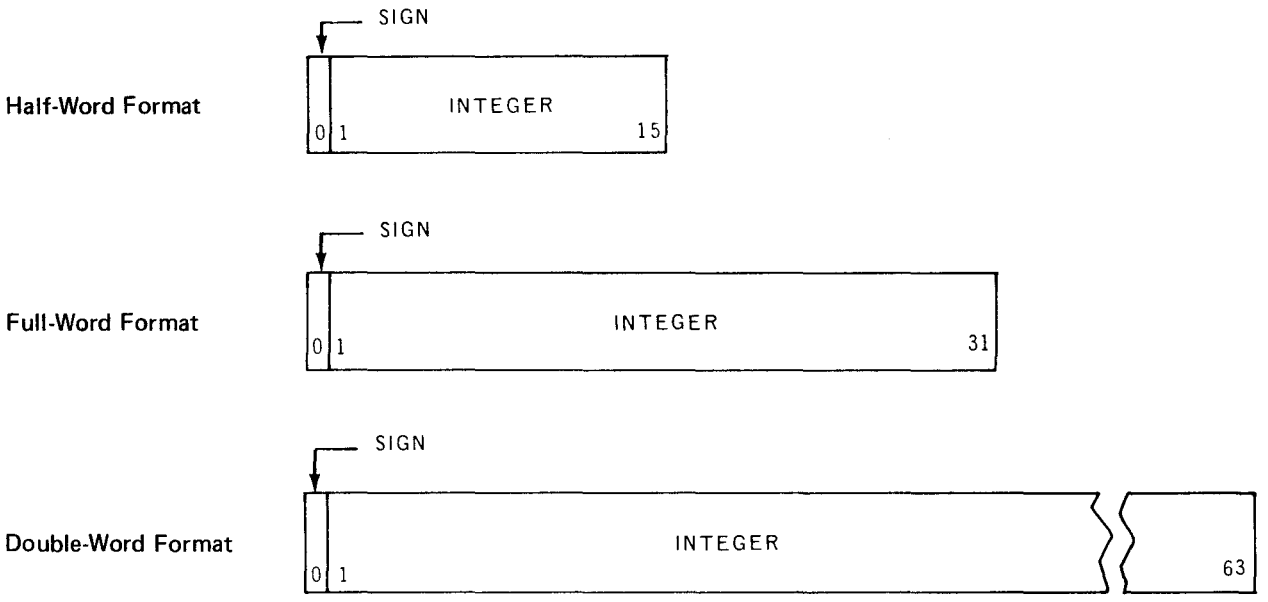


Figure 1–1. Fixed-Point Number Formats

When held in one of the 32 general registers (16 for supervisor functions and 16 for user program functions), a fixed-point number is generally treated as a 32-bit operand. Certain multiply, divide, and shift operations use a 64-bit operand composed of one sign bit and a 63-bit integer field. A 64-bit operand is located in two adjacent general registers and is addressed by referring to the even-numbered register of the even-odd register pair.

When fixed-point data is located in main storage, it may be stored in any of the three formats. This data must be located on the integral main storage boundary of its associated format.

A half word in storage is extended to a full word by propagation of the sign bit through the most significant 16 bits of the full word when it is transferred to the processor. The half word then operates as a full word in fixed-point arithmetic operations.

1.3.2. Floating-Point Numbers

Floating-point numbers are represented in signed absolute value form and have a fixed-length format which is either a full word (short format) or a double word (long format) in length. Both formats may be used in main storage and in the floating-point registers (6.1). Short format numbers provide faster processing and requires less storage space than long format numbers. Long format numbers provide greater precision in computations.

In either format, bit 0 is the sign bit, bits 1 through 7 are the exponent, and the remaining bits are the fraction. The exponent is expressed in excess-64 binary notation. The fraction is expressed as a hexadecimal number having the radix point to the left of the most significant fraction digit. The quantity expressed by the full floating-point number is the product of the fraction and the number 16 raised to the power minus 64 of the exponent.

Separate instructions are provided for operations with long and short format operands (Section 6). The short and long formats are illustrated in Figure 1–2.

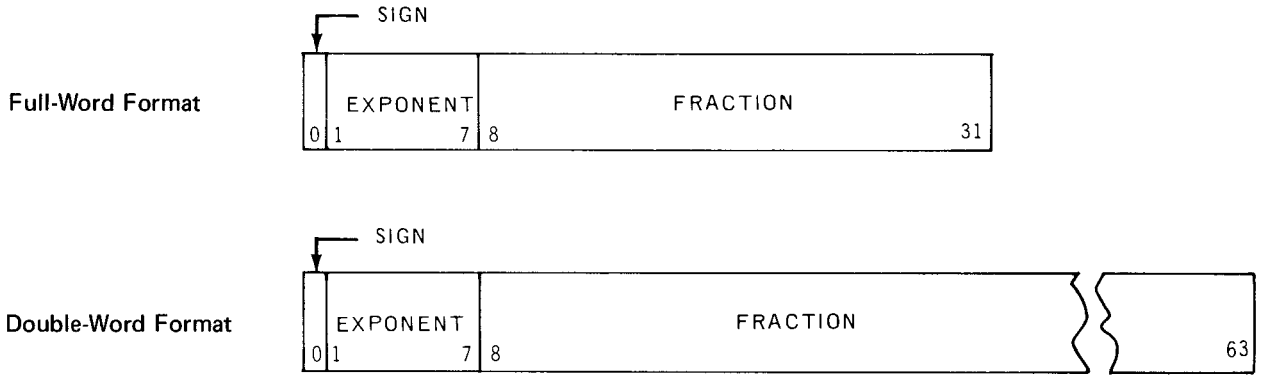


Figure 1–2. Floating-Point Number Formats

1.3.3. Hexadecimal Representation

Hexadecimal digits are considered 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 either a decimal number or alphabetic or symbolic character. (Hexadecimal digits are also used for constant definition as described in Section 11.) 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

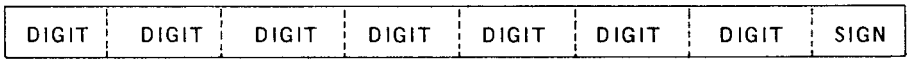
1.3.4. Decimal Number Representation

Decimal numbers are represented in either 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.

ZONE	DIGIT	ZONE	DIGIT	ZONE	DIGIT	SIGN	DIGIT

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 sign of decimal numbers is represented by hexadecimal digits A through F. Any other bit configuration is an invalid sign code which could produce unpredicatable results.

The interpretation of the contents of the sign position is:

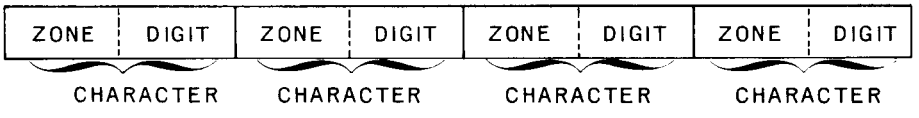
HEXADECIMAL CHARACTER	BINARY VALUE	SIGN VALUE
A	1010	Positive
B	1011	Negative
C	1100	Positive (EBCDIC mode)*
D	1101	Negative (EBCDIC mode)*
E	1110	Positive (EBCDIC mode)
F	1111	Positive (EBCDIC mode)**

*Automatically generated in central processor for decimal operations

**Automatically generated in central processor for zone fill during unpack instruction (Binary value is 0011 in ASCII mode.)

1.3.5. Character Representation

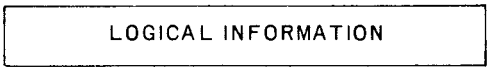
An alphabetic or other symbolic character representation is contained in the full eight bits of a byte. A character field is considered as not containing a sign. This type of field is represented:



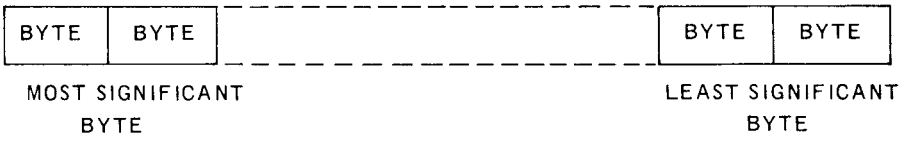
1.3.6. Logical Information

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

Fixed-length logical information consists of one, two, four, or eight bytes.



Variable-length logical information consists of up to 256 bytes.



1.4. STATEMENT CONVENTIONS

The conventions used to delineate the control statements in this manual are:

- Uppercase letters and words and the following characters must be coded exactly as shown:

  , (comma)

  . (period)

  = (equal sign)

  () (parentheses)

  ' (apostrophe)

  * (asterisk)

  # (number sign)

  Examples:

  C

  #

  MCL

  RECSIZE=

- Lowercase letters and words are generic terms representing information supplied by the user. Such lowercase terms may contain hyphens (for readability).

  Examples:

  n

  start-addr

  vol-no-1

- Information within braces represents required entries, one of which must be chosen.

  Examples:

  { file-id }
  { 'file-id' }

  { LAST }
  { OPR }
  { RESET }

- Information within brackets represents optional entries that are included or omitted (depending upon user requirements). Braces within brackets signify that one of the specified entries must be chosen if that parameter is to be included.

  Examples:

  [version-no]

  [(ALT,n)]

  $$\left[\left\{\begin{matrix}\text{JOB}\\\text{STEP}\end{matrix}\right\}\right]$$

  $$\left[\text{VERIFY=}\left\{\begin{matrix}\text{YES}\\\text{NO}\end{matrix}\right\}\right]$$

- An ellipsis (series of three periods) indicates the presence of a variable number of entries.

  Examples:

  Statement with a specific number of parameters

  lun-1 [,lun-2,...[,lun-20]]

  Statement with an unlimited number of parameters

  domid-1,...

1.4.1. Positional Parameters

Positional parameters must be written in the order specified, and each must be separated by a comma. When a positional parameter is omitted, the comma, even though it may be shown inside a bracket, must be retained to indicate the omission, except in the case of omitted trailing positional parameters.

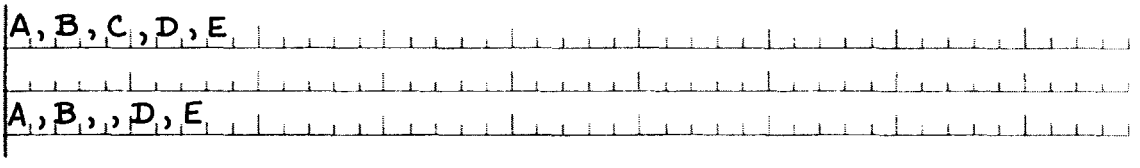
The following examples are provided to aid the user in coding the positional parameter formats used in this manual. The coding possibilities do not necessarily reflect all the coding options that may be used.

- Optional positional parameters within a series of required parameters

  Format:

  A,B,[C],D,E

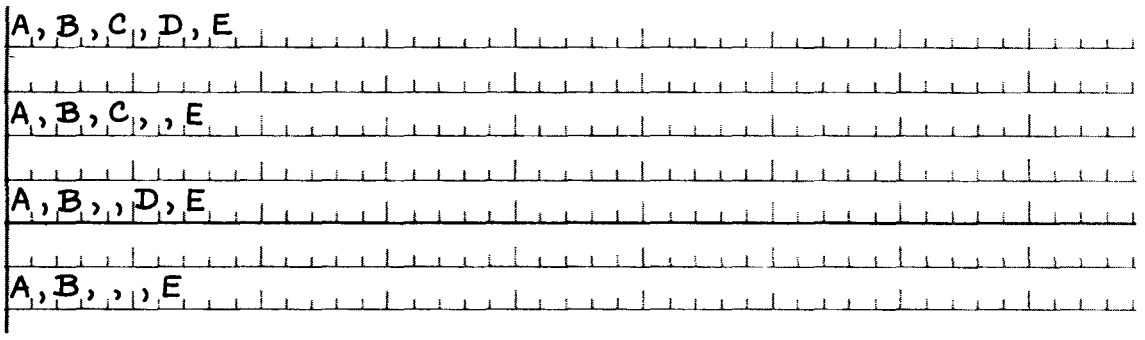
  Coding possibilities:



Format:

A,B,[C],[D],E

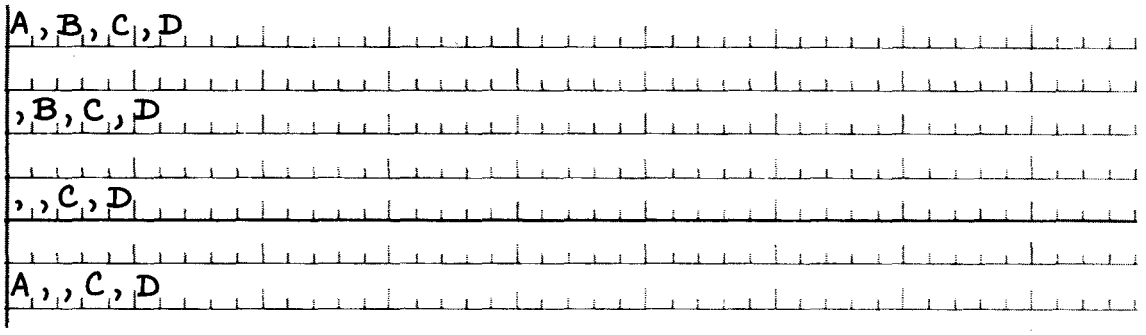
Coding possibilities:



Format:

[A],[B],C,D

Coding possibilities:

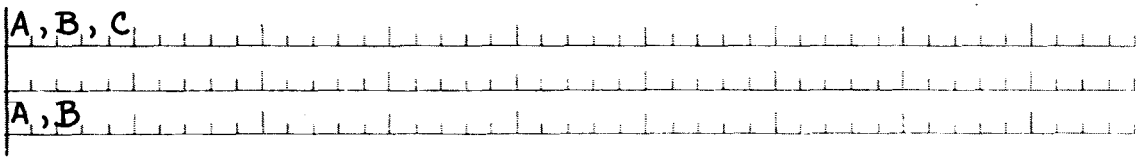


- Optional positional parameters at the end of a series of required parameters

Format:

A,B[,C]

Coding possibilities:



Format:

A,B[,C] [,D]

Coding possibilities:

```
A,B,C,D
A,B,C
A,B,,D
```

- Optional positional parameters within a series and at the end of a series of required parameters

Format:

A,B,[C],D,E[,F]

Coding possibilities:

```
A,B,C,D,E,F
A,B,C,D,E
A,B,,D,E,F
A,B,,D,E
```

Format:

A,B,[C],[D],E[,F]

Coding possibilities:

```
A,B,C,D,E,F
A,B,C,D,E
A,B,C,,E
A,B,,,E
A,B,,D,E,F
A,B,,,E,F
```

1.4.2. Keyword Parameters

A keyword parameter consists of a word or character immediately followed by an equal sign, which is, in turn, followed by a specification. Commas are required only to separate keyword parameters and need not be retained to indicate the omission of a keyword parameter. In the format presentation of this manual, all required keyword parameters are shown first, followed by the optional keyword parameters. In coding, they may be specified in any order desired.

The initial keyword specified in the operand field is always coded without its associated comma. However, each succeeding keyword parameter specified must be preceded by a comma. A comma is never used to begin the coding specified in the operand field when keyword parameters are the only type of parameter specified in a statement. Beginning a line of coding with a comma is applicable only when an optional positional parameter is defined as the initial parameter of the coding. See 1.4.1.

The following examples are provided to aid the user in coding the keyword parameter formats used in this manual. The coding possibilities do not necessarily reflect all the coding options that may be used.

- All keyword parameters required

  Format:

  ABC=nn,XYZ= {1 / 2}

  Coding possibilities:

```
ABC=nn,XYZ=1
XYZ=2,ABC=nn
```

- Combination of required and optional keyword parameters

  Format:

  ARM=HAND[,LEG=FOOT] [,HEAD= {NOSE / EYE}]

  Coding possibilities:

```
ARM=HAND,LEG=FOOT,HEAD=EYE
LEG=FOOT,ARM=HAND,HEAD=NOSE
HEAD=EYE,LEG=FOOT,ARM=HAND
ARM=HAND
HEAD=HAND
HEAD=EYE,ARM=HAND
ARM=HAND,LEG=FOOT
```

- All optional keyword parameters

  Format:

  [PAPER=WOOD] [,GLASS=SAND]

  Coding possibilities:

```
PAPER=WOOD,GLASS=SAND
GLASS=SAND,PAPER=WOOD
PAPER=WOOD
GLASS=SAND
```

1.4.3. Combination of Positional Parameters and Keyword Parameters

When positional parameters and keyword parameters are used in the same statement, the keyword parameter must follow the last specified positional parameter. When a positional parameter within a series of parameters is omitted, the associated comma must be retained to indicate its omission. When a trailing positional parameter is omitted, its associated comma is also omitted, except when followed by a keyword parameter.

- All parameters required

  Format:

  TOY,BROKE= { YES / NO } ,CHILD= { HAPPY / SAD }

  Coding possibilities:

```
TOY,BROKE=YES,CHILD=SAD
TOY,CHILD=HAPPY,BROKE=NO
```

- Required and optional parameters

  Format:

  DOG[,C] [,A] [,T] ,FIGHT= { YES / NO } [,TREE= { YES / NO }][,CATCH= { YES / NO }]

Coding possibilities:

```
DOG,C,A,T,FIGHT=NO,TREE=YES,CATCH=NO
OG,,,AT,FIGHT=YES,CATCH=YES,TREE=NO
DOG,,,T,TREE=YES,FIGHT=NO
DOG,FIGHT=NO
DOG,FIGHT=YES,CATCH=YES
```

- Keyword parameters acting as positional parameters

In some instances, an optional keyword parameter performs the same function as a positional parameter, and is treated as a positional parameter in both presentation and coding (i.e., the comma is shown as being outside the bracket; when the keyword parameter is not selected, the comma must be used to show its omission). The second parameter in the following format is a keyword acting as a positional parameter; [,D=E] and [,Q=T] are keyword parameters.

Format:

$$a,\left[A=\begin{Bmatrix}B\\C\\D\end{Bmatrix}\right],b,c[,D=E]\ [,Q=T]$$

Coding possibilities:

```
a,A=B,b,c,D=E,Q=T
a,,b,c,D=E
a,A=C,b,c,Q=T
```

1.4.4. Subparameters

Parameters enclosed in parentheses and separated by commas are called subparameters. The parentheses must be shown to delimit the series of subparameters, and to prevent the commas from being interpreted as parameter separators. Subparameters follow the same conventions as those for positional and keyword parameters.

1.4.5. Default Options

Underlined parameters are selected automatically when a keyword parameter or subparameter is omitted.

Examples:

$$\left\{\begin{array}{c}\text{time}\\ \underline{500}\end{array}\right\}$$

$$\left\{\begin{array}{c}\text{YES}\\ \text{NO}\\ \underline{93}\end{array}\right\}$$

2. Assembly Language

2.1. CHARACTER SET

The assembly language character sets used in writing statements for the SPERRY UNIVAC Operating System/4 (OS/4) are:

- Letters:

  A through Z

- Special Letters:

  ? $ # @

- Digits:

  0 through 9

- Special Characters:

  + − * / , = blank ( ) . > < & '

The characters ?, $, #, and @ are considered to be special letters because they may be written as one of the characters in a symbol.

In the description of instructions, the following specific definition terms are used:

- Alphabetic – a character of the letter or special letter set;
- Numeric – a character of the digit set; and
- Alphanumeric – a character of the letter, special letter, or digit set.

2.2. STATEMENT FORMAT

Statements in the OS/4 assembly language are written on the assembler coding form (Figure 2–1). Statements and comments are generally written in columns 1 through 71. Column 72 is used to indicate continuation. Statements are continued in column 16 of the following card. Columns 73 through 80 may contain program identification and sequencing information. This format may be altered by the input format control (ICTL) assembler directive (12.7.1).

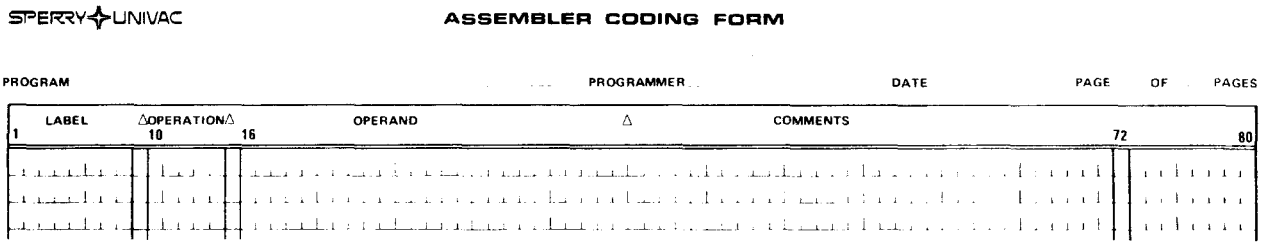


Figure 2–1. Assembler Coding Form

Although assembly language is written in 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 listing which is neater and easier to read. The rules governing the writing of free form assembler statements are discussed in 2.2.6.

2.2.1. Label Field

The optional label field may contain a symbol for which a value is to be defined. The label field must begin in column 1 of the coding form and is terminated by a blank column. A blank in column 1 is interpreted as indicating the absence of this field.

2.2.2. Operation Field

The operation field begins with the first nonblank character after the label field and is terminated by a blank. Embedded blanks are not permitted. The operation field contains the mnemonic operation code for a machine instruction, the name of an assembler directive, or the name of a previously defined macro instruction. The operation code must be written exactly as specified in the assembler instruction or directive format description.

2.2.3. Operand Field

The operand field begins with the first nonblank column after the operation field and is terminated by a blank not contained in a character constant representation, character self-defining term, or character expression. This field may contain data and/or information used by the machine instructions or assembler directives, or parameters completing the specification of the procedure reference. An operand of a source line statement may not terminate in column 71 unless a continuation is called for (in which case column 72 is not blank). Column 72 cannot be used for coding or as the terminating blank of the operand field.

2.2.4. Comments Field

The comments field for a statement begins with the column following the blank column that terminates the operand field and it ends at column 71. It may contain any combination of characters including blanks. It is not processed by the assembler other than to include it on the assembly listing. It may contain remarks to clarify the purpose or operation of the associated coding.

Lines may consist entirely of comments from columns 2 through 71 if column 1 contains an asterisk. Comments written on an instruction line may not be continued and must terminate on or before column 71.

2.2.5. Continuation

If a nonblank character is entered in column 72, the operand field of the current source code line may be continued beginning in column 16 of the following line. Column 72 is the normal continuation column and column 16 is the normal continue column; however, these can be altered by using the ICTL directive described in 12.7.1. Continued lines are regrouped when listed on the line printer.

2.2.6. Statements in Free Format

Statements may be written in free format disregarding the standard form, providing the following rules are observed:

- The label field must start in the begin column, as specified by the ICTL directive.
- If the label field is omitted, the operation field must begin at least one column to the right of the begin column.
- The label and operation fields must appear in the first line of the statement.
- A field must be terminated by at least one blank.
- As in normal form, neither the label nor the operation field may contain embedded blanks. The blank is always used as a delimiter to terminate a field. The operand field can contain blanks within a valid character string.
- The entries must appear in normal sequence: label, operation code, operand, comments.
- An entry may not extend beyond statement boundaries.

2.3. TERMS

Terms are representations of values. The assembler recognizes five classes of terms:

- Self-defining terms
- Literals
- Symbols
- Location counter references
- Attribute references

Self-defining terms are fixed values coded by the programmer. Literals can have their value specified by the programmer or computed by the assembler. Symbols, location counter references, and attribute references are assigned values by the assembler.

2.3.1. Self-Defining Terms

Self-defining terms (SDT) represent fixed values. These representations are not relocatable and are used to specify immediate data, registers, addresses, and masks in machine instructions. The representation can also be used to specify values in directive operands or in expressions. Restrictions on the size of SDT's depend on where they are used. When SDT's are used to designate a register, the value cannot exceed 15. The representation of an address must not have a value greater than the total size of storage. After conversion to a binary format, the value is truncated or filled with nonsignificant 0's to fit the designated field. SDT's can be represented in binary, hexadecimal, decimal, or character form. A description of each type of representation follows.

2.3.1.1. Binary Representation

A binary representation consists of a series of up to 24 zeros and ones enclosed in apostrophes and preceded by the letter B. The bit pattern is stored as specified with high order 0's added when necessary. The following are valid binary representations:

Binary Representation	Binary Value
B'10011'	00010011
B'11'	00000011
B'101101000101'	00001011 01000101

2.3.1.2. Hexadecimal Representation

A hexadecimal representation consists of up to six hexadecimal digits enclosed in apostrophes and preceded by the letter X. This representation is used primarily to convey binary or bit-pattern information to the system. Each hexadecimal digit represents a half byte of information. The hexadecimal digits and their values are:

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

Examples of hexadecimal representations and the binary values they produce are:

Hexadecimal Representation	Binary Value
X'D'	00001101
X'101'	00000001 00000001
X'7FFF'	01111111 11111111
X'ABC'	00001010 10111100
X'F1F2'	11110001 11110010

2.3.1.3. Decimal Representation

A decimal number may be used to specify directly to the assembler a value that will be converted to a binary value or other bit configuration. The decimal number may consist of up to eight digits, 0 through 9, forming a decimal number, 0 through 16,777,215. This number is converted to a binary value occupying one or more bytes depending on the type of field for which it is intended. Decimal numbers and the binary values they produce are:

Decimal Representation	Binary Value
0	00000000
1	00000001
15	00001111
257	00000001 00000001
00013	00000000 00001101
32767	01111111 11111111

2.3.1.4. Character Representation

A character representation consists of up to three characters of the 256 valid characters; however, only 63 of the 256 valid characters are printable. The characters must be enclosed in apostrophes and preceded by the letter C. Each ampersand or apostrophe to be included in a character representation is represented by a double ampersand or double apostrophe, respectively. In this case, there may be more than three characters within the apostrophes which delimit the character representation.

The character representation is used to specify immediate data or binary bit patterns. Character representations and their values are:

Character Representation	Binary Value
C'D'	11000100
C'NOT'	11010101 11010110 11100011
C'9'	11111001
C'''&&'''	01111101 01010000 01111101

2.3.2. Literals

A literal is a representation of data which is replaced by the storage address of the actual data. When the assembler recognizes a literal in the source code, it searches the table of literals that have been previously encountered. If a duplicate is found, then the relocatable address of the literal in the table replaces the original literal in the source code. If a duplicate is not found, then the value of the original literal is entered into the table and its address replaces the source code specification. Literals are similar in form to the operands of DC and DS statements. A more detailed description of literals is given in 11.5.

A literal may be used in any machine instruction that specifes a main storage address and must appear as the complete operand specification. However, the literal may not be specified as the receiving field operand of an instruction that modifies main storage, in address constants, shift instructions, input/output instructions, nor combined with other terms or with an explicit base register specification.

2.3.3. Symbols

A symbol is a group of up to eight alphanumeric characters. The first, or leftmost, character must be alphabetic. Special characters or blanks may not be contained within a symbol. Examples of valid symbols are:

V	CARDAREA
GS279	R$INTRN
BOB	BD#4

Not valid symbols for the reasons stated are:

READ ONE	Embedded blank
SPEC'L	Special character
6AGN	First character not alphabetic

A symbol may be more than eight characters long; however, only the first eight characters are analyzed by the assembler. If the first eight characters of any two symbols are identical, they are considered to be identical symbols regardless of following characters.

The assembler associates three attributes with each symbol it processes: value, length, and relocatability. Symbols defined by the EQU directive adopt the attributes of the expression in the operand field of the statement.

2.3.3.1. Value Attribute

A symbol 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 the adjustment to a half-word boundary, if necessary. 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 that instruction, constant, or storage area.

2.3.3.2. Length Attribute

The length attribute of a symbol is the number of bytes assigned to the instruction, constant, or storage area involved. For example, the label of a 2-byte instruction has a length attribute of 2 and the label of a DS statement reserving 50 words (four bytes per word) would have a length attribute of 4. Symbols equated to location counter references and/or absolute value representations usually have a length attribute of 1.

The maximum length attribute that can be generated by the assembler is 256 bytes; however, a DS may be used to reserve more than 256 bytes of storage.

2.3.3.3. Relocatability Attribute

A symbol may either be absolute or relocatable. Values which are assigned to symbols defined in the label field of a source code line representing an instruction, constant, or storage definition, are relocatable. A relocatable symbol is a symbol whose value would change by a given number of bytes if the program in which it appears is relocated the same number of bytes from its originally assigned address. Relocatable symbols are assigned values relative to the location counter. Decimal, character, binary, and hexadecimal representations are all absolute terms and have a relocation attribute of 0.

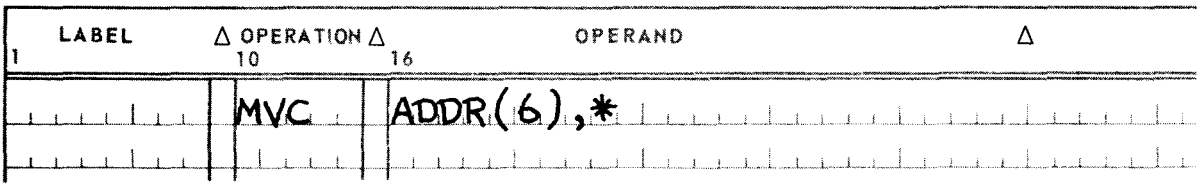
2.3.4. Location Counter References

A location counter is maintained by the assembler for each control section created by the programmer. Each counter contains the next available location for the associated control section. After the assembler processes an instruction or constant, it adds the length of the instruction or constant processed to the current location counter.

Each instruction or address constant must have an address which is a multiple of two bytes. This type of address is said to fall on a half-word boundary. If the value of the location counter is not a multiple of 2 when assembling such a constant or instruction, a 1 is added to the location counter before assigning an address to the current statement. Storage locations bypassed in this way receive binary 0's when the program is loaded.

The current value of the location counter, under which the program is currently being assembled, is available for reference by the programmer. It is represented by the special character * (asterisk). If the asterisk is written in a constant representation or in an instruction operand expression, this character is replaced by the storage address of the leftmost byte allocated to that instruction or constant. Thus, in the following example the instruction generates an object code instruction with the address of the MVC instruction as Operand 2. When executed, the MVC instruction will be transferred to a 6-byte area labeled ADDR.

Example:



An instruction may address data or other instructions in its immediate vicinity in terms of its own storage address. This is one kind of relative addressing and it is 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 data being accessed. Relative addressing is always in terms of bytes and not in terms of words or instructions.

2.3.5. Attribute References

References to symbol attributes assigned by the assembler are treated as terms.

- Length

  The length attribute of a symbol may be referenced (2.3.3.2) by writing L' followed by the symbol. Thus, if the symbol STOREND is the name of a full-word field,

  L'STOREND

  would be considered a term and it would have a value of 4.

The length attribute is not available during conditional assembly processing. Specifically, a paraform (13.3.1.1) may not be a length attribute if it will be on a "DO" line of the procedure definition.

- Number

The number attribute is only valid for paraforms (13.3.1.1) associated with procedure definitions. The attribute refers to the number of items within the list or sublist specified by the paraform. A reference may be made to a number attribute by writing N' followed by the paraform. The reference is replaced by the number of elements.

For example, to obtain the total number of positional parameters associated with the parameter named LST, the following would be coded in the procedure definition:

```
N'LST
```

If positional parameter 1 of LST was a sublist, it would be possible to determine the number of items in the sublist by coding:

```
N'LST(1)
```

Also, if a keyword parameter, KEY, is equated to a sublist on a procedure definition call line, then the number of items in the sublist can be determined by coding:

```
N'KEY
```

2.4. OPERATORS AND EVALUATION

There are 12 operators in the OS/4 assembler which designate the method, and implicitly the sequence, to be employed in combining terms or expressions (Table 2–1). Blanks are not permitted within an expression. Evaluation of an expression begins with the substitution of values for each term. The operations are then performed from left to right in hierachical order as listed in Table 2–1. The operation with the highest hierarchy number is performed first; operations with the same hierarchy number are performed from left to right. Parentheses may be used to alter the order of evaluation. Division by 0 equals 0. The 12 operators are divided into three classes: arithmetic operators, logical operators, and relational operators. More detailed descriptions of these operators are provided in the following paragraphs.

Table 2–1. Summary of Operators

Classification	Operator	Description	Hierarchy
Arithmetic Operators	*/	A*/B is equivalent to $A*2^B$	6
	//	Covered quotient, A//B is equivalent to (A+B–1)/B	5
	/	A/B means arithmetic quotient of A and B	5
	*	A*B means arithmetic product of A and B	5
	–	A–B means arithmetic difference of A and B	4
	+	A+B means arithmetic sum of A and B	4
Logical Operators	**	A**B means Logical Product AND of A and B	3
	++	A++B means Logical Sum OR of A and B	2
	––	A––B means Logical Difference XOR or A and B	2
Relational Operators	=	A=B has value 1 if true; has value 0 if false	1
	>	A>B has value 1 if true; has value 0 if false	1
	<	A<B has value 1 if true; has value 0 if false	1

2.4.1. Arithmetic Operators

The symbols +, −, *, /, //, and */ represent the six arithmetic operators. The intrinsic meanings of +, −, *, and / are the usual ones; that is, + indicates addition, − indicates subtraction, * indicates multiplication, and / indicates binary division.

The operator // denotes a covered quotient where A//B is equivalent to (A+B−1)/B. A covered quotient is equal to regular binary division except that if there is a remainder, a 1 is added to the regular quotient.

The operator */ denotes a binary shift left or right, A*/B indicates a left shift and is equivalent to $A*2^{B}$, A*/(−B) indicates a right shift and is equivalent to $A/2^{-B}$.

2.4.2. Logical Operators

The symbols **, ++, and −− are the three logical operators. The characters ** represent the logical product (AND), the characters ++ represent the logical sum (OR), and the characters −− represent the logical difference (exclusive OR).

Each bit of the first term is compared with its corresponding bit in the second term and the result of the comparison is placed in the corresponding position in the resulting term. The result of the bit comparison for each operator is:

AND	
A**B	Result
1 1	1
1 0	0
0 1	0
0 0	0

OR	
A++B	Result
1 1	1
1 0	1
0 1	1
0 0	0

XOR	
A−−B	Result
1 1	0
1 0	1
0 1	1
0 0	0

2.4.3. Relational Operators

The three relational operators are the equals operator (=), the greater than operator (>), and the less than operator (<).

The equals operator is used to compare the value of two terms or expressions. If the two values are equal, the assembler assigns a value of 1 to the expression; otherwise, a value of 0 is assigned.

The greater than operator makes a comparison between two terms or expressions. If the value of the first (left) term is greater than the value of the second (right) term, then a value of 1 is assigned to the expression; otherwise, a value of 0 is assigned.

The less than operator compares the value of the first (left) expression or term with the second (right) expression. If the value of the first expression is less than the value of the second one, then a value of 1 is assigned to the expression; otherwise, a value of 0 is assigned.

For the expression A+B>C, if the expression A+B has a value greater than the value of C, then the assembler assigns a value of 1 to the expression; otherwise, a value of 0 is assigned.

2.5. EXPRESSIONS

An expression consists of one or more terms connected by operators. A leading minus sign is allowed to produce the negative of the first term.

Two types of expressions, absolute and relocatable, possess various characteristics obtained from the term or terms which compose them. These two types of expressions are discussed in the following paragraphs.

2.5.1. Absolute Expression

An absolute expression is an expression whose value is unchanged by program relocation. The absolute expression can be an absolute term or any combination of absolute terms. Arthmetic operators are permitted between absolute terms.

Relocatable terms alone or relocatable terms in combination with absolute terms can be contained within an absolute expression. This type of absolute expression requires the following conditions:

- The relocatable terms must be paired in even numbers.
- Each pair of relocatable terms must have opposite signs and have the same relocatability attribute (that is, appear in the same control section). However, the paired relocatable terms need not be contiguous.
- If a relocatable term in the absolute expression enters into a multiply or divide operation, an error flag is given and the result is treated as absolute (except multiplication and division by absolute 1). Therefore, R-R*A is flagged and R*A is treated as absolute. Multiplication by absolute 0 is absolute 0.

The effect of relocation is canceled by the pairing of relocatable terms with the same relocatable attribute and opposite signs. The absolute expression is thereby reduced to a single absolute value.

The following are examples of absolute expressions:

```
A
A+A–A
A–A+A+A
R+A–R
R–R+A
(R–R)*A
A*A
```

where:

A is an absolute term.

R is a relocatable term.

2.5.2. Relocatable Expressions

A relocatable expression is an expression whose value changes with program relocation. All relocatable expressions must be a positive value. All arithmetic operators are permitted between the relocatable terms.

Relocatable terms alone or relocatable terms in combination with absolute terms can be contained within a relocatable expression. Either type of relocatable expression requires the conditions:

- An odd number of relocatable terms is necessary.
- All but one relocatable term must be paired.
- A minus sign must not precede the unpaired (remaining) relocatable term.
- Each pair of relocatable terms must have opposite signs and the same relocatability attribute.
- The paired relocatable terms do not have to be contiguous.
- Multiplication and division of a relocatable term by an absolute 1 or multiplication of an absolute 1 by a relocatable term produces a relocatable expression.

Using the above requirements, a relocatable expression is thereby reduced to a single relocatable expression. The following are examples of relocable expressions:

R
R+A or A+R
R–R+R
R–R+A+R
R–A
R*1 or 1*B

where:

A is an absolute term.

R is a relocatable term.

An expression may be negatively relocatable only under certain circumstances (11.8.2). Such an expression consists of either an absolute term minus a relocatable term or an expression that can be reordered to that form as:

A–R
A–R–R+R
R–R+A–R

where:

A is an absolute term.

R is a relocatable term.

Any expression that does not conform to the rules discussed in this section is flagged. Also, the result of invalid pairing of terms is treated as an absolute term.

2.5.3. Length Attribute of Expressions

The length attribute of an expression is determined by the assembler and is a function of the leading term of the expression. If the first term of an expression is an absolute value, a length attribute of one byte is assigned to the expression. If the leading term is a symbol, the number of bytes attributed to the expression is the same as the length attributed to the symbol. Thus, if TAG appears in the label field of an LH (load-half-word) instruction, it would have a length attribute of 4 since LH is a 4-byte instruction. In referencing the same label, the expression TAG+195 also has a length attribute of 4, but the expression 195+TAG has a length attribute of 1 because the leading term is a decimal self-defining term.

2.5.4. Character Expressions

A character expression is either a character string, a character substring, or a concatenation of strings and/or substrings. Character expressions are used mainly to specify set symbol values for replacement in source code statements (12.8.1). They can also appear in arithmetic expressions as terms used by relational operators. All arithmetic values are considered to be greater in value than any character string and any character string is considered to be greater in value than any shorter character string. Where character expressions appear in arithmetic expressions as terms for operations other than the relational operators, they are flagged and treated as 0's.

- Character Strings

  A character string is zero, one, or more of the 256 valid characters enclosed by apostrophes. A character string differs from a character absolute value representation in that the character absolute value representation is converted to and treated as a binary value. A character string is not treated as a value. Character strings can be up to 127 characters in length. Apostrophes within the string must appear as pairs of successive apostrophes. Ampersands must appear as pairs of ampersands.

- Substring Representation

  A character substring is a valid character string followed immediately by two unsigned decimal numbers which are separated by a comma and enclosed in parentheses. The format is:

  character-string(n_1,n_2)

  The first number (n_1) indicates the leftmost character of the original string which is to be included in the substring. The second number (n_2) represents the number of characters to be included in the substring.

  For example:

  'PREDEFINED'(4,6)

  represents the same string of characters as does

  'DEFINE'

- Concatenation

  Concatenation is the joining together of two character strings, two character substrings, or a character string and a character substring. A period is used to designate concatenation that results in the formation of a single string of characters. The characters of the second string or substring are placed immediately following the characters of the first term in the construction of the resultant string. The following example shows concatenation:

  'PRE'.'DEFINE' produces PREDEFINE

  When a substring is to be concatenated with a following character string, the period may be omitted and concatenation is assumed.

2.5.5. Basic Expressions

A basic expression consists of one or more basic terms connected by operators. Basic expressions are used to specify information for certain directives.

The following are basic terms:

- Self-defining terms (SDT)
- Symbols defined by SET directives (set symbols)
- Character expressions
- Number attribute references
- Parameter reference forms (paraforms)

All the operators used by the OS/4 assembler are valid; the rules for expression evaluation are the same as with normal expressions; and parentheses may be used for grouping. Division by 0 equals 0. Blanks are not permitted in the expression except within a defined character string or substring. All terms must be predefined.

3. Introduction to Instructions

3.1. INSTRUCTION TYPES AND FORMATS

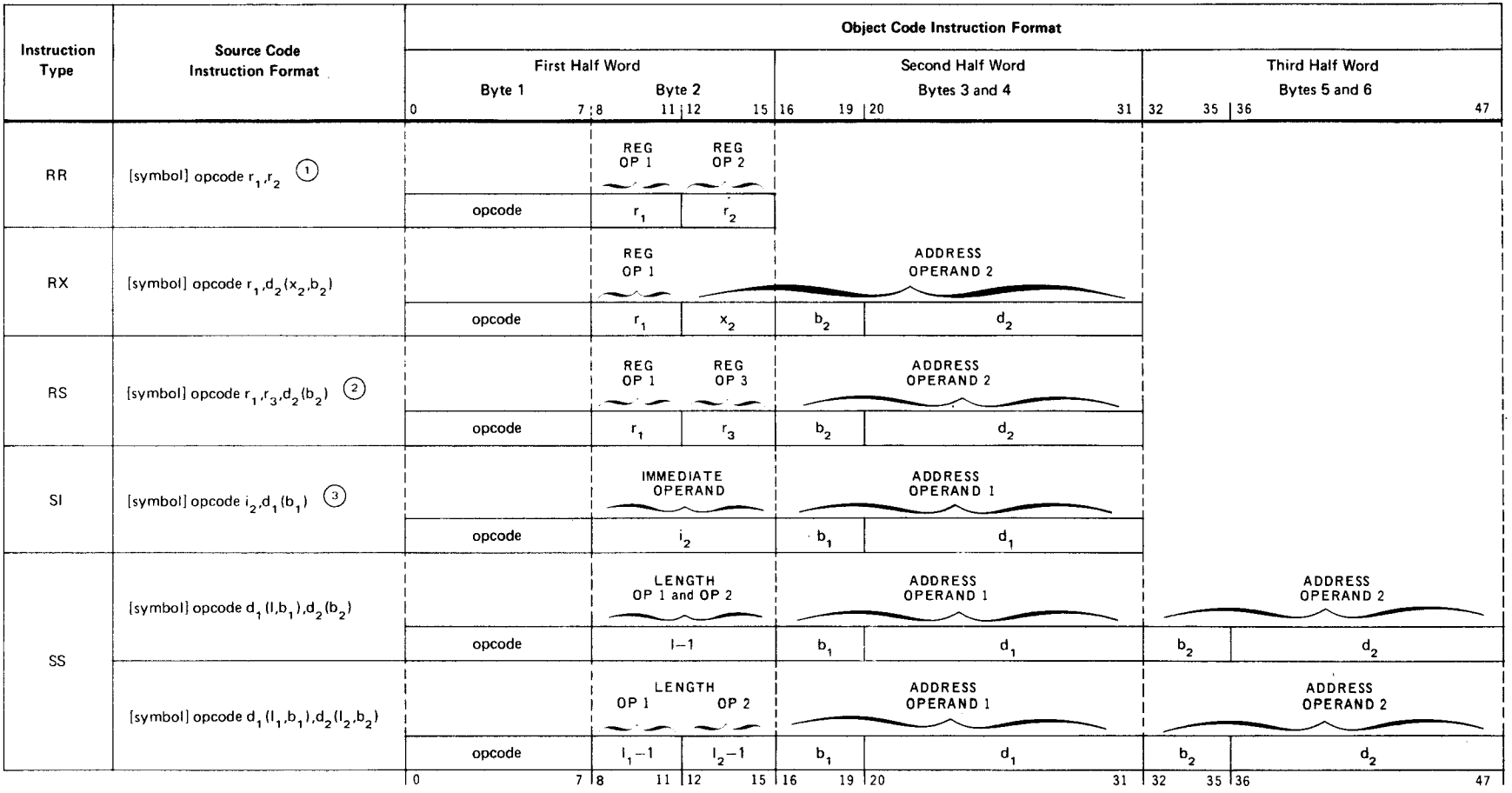
The normal mode of operation for program run under the control the SPERRY UNIVAC Operating System/4 (OS/4) is the 9400/9480 compatible mode; therefore, the descriptions of the instructions in this manual reflect this mode of operation.

An instruction is an executable statement for operations involving data. The instructions for the OS/4 are divided into five types according to the operation specified by the instruction. The instructions are of three lengths: two, four, or six bytes. In a 2-byte, or half-word instruction, the general registers are referenced for both operands. A 4-byte instruction references main storage for one operand, and the general registers or immediate data for the other operands. A 6-byte instruction references main storage for both operands. Each instruction is aligned by the assembler on a half-word boundary; that is, each has an even address. The five instruction types are:

- RR register to register operation, requiring two bytes of main storage.
- RX register and indexed storage operation, requiring four bytes of main storage.
- RS register and storage operation, requiring four bytes of main storage.
- SI storage and immediate operand (one contained in the instruction) operation, requiring four bytes of main storage. Only self-defining terms may be used as immediate operands.
- SS main storage to main storage operation, requiring six bytes of storage. The SS instruction format is used for packed decimal arithmetic (maximum operand length is 16 bytes) and for byte-by-byte processing of data (maximum operand length is 256 bytes).

The basic formats for instruction are shown in Figure 3–1 in source code and object code form.

Table 3–1 defines the abbreviations used in the description of instructions.



NOTES:

(1) The RR instruction has two other forms:

[symbol] opcode i_1 for the SVC and SRF instruction

[symbol] opcode r_1 for the SPM instruction

(2) The RS shift instructions are written without use of the r_3 operand, in the form:

[symbol] opcode $r_1,d_2(b_2)$

(3) Some SI instructions, such as HIO and TIO, do not use an i_2 field. They are written in the form:

[symbol] opcode $d_1(b_1)$

Figure 3–1. Instruction Formats

Table 3–1. Abbreviations Used in Descriptions of Instructions

Abbreviation	Definition
a	Absolute term or expression
Δ	Blank
b_1	Number of the general register which holds the base address of operand 1
b_2	Number of the general register which holds the base address of operand 2
cc	Condition code
d_1	Displacement for the base address of operand 1 (absolute displacement)
d_2	Displacement for the base address of operand 2 (absolute displacement)
e	Expression
i_1	Immediate data used as operand 1
i_2	Immediate data used as operand 2
l	Length of the operands as stated in source code
l_1	Length of operand 1 as stated in source code
l_2	Length of operand 2 as stated in source code
m	Mask
opcode	Instruction operation code
op_1	Operand 1
op_2	Operand 2
op_3	Operand 3
r_1	Number of the general register which holds operand 1
r_2	Number of the general register which holds operand 2
r_3	Number of the general register which holds operand 3
x_2	Number of the general register which holds an index number for operand 2 of an RX instruction

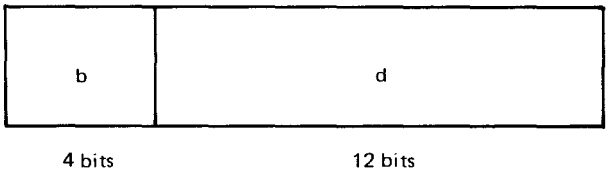
3.2. OPERAND ADDRESSING

Operands may be located in three places:

- contained in the instruction;
- stored in the operating registers; or
- in main storage.

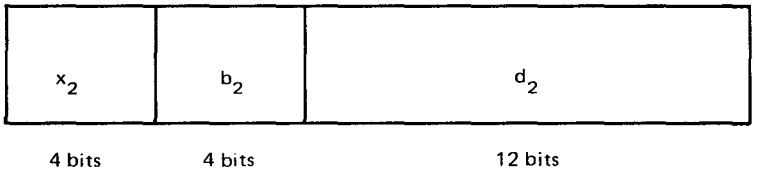
No address is needed for an operand which is part of the instruction, that is, an immediate operand. When operands are located in the register set, the register is addressed by using a 1-digit value. However, operands in main storage must be addressed by a main storage address, the formation of which is explained in the following paragraphs.

Instructions which access operands from main storage have two operand specification fields:

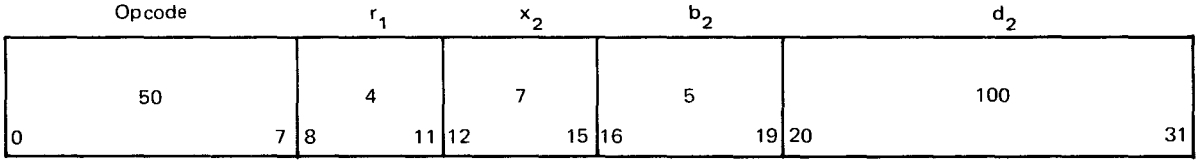


The b (base) field of the operand designates a general purpose register. The value contained in this register represents the base address. The 12 bits of the d (displacement) field contain the displacement. The base address and the displacement are added to obtain the effective address of the operand.

The RX instructions use three fields to specify the operand address :



An example of this addressing follows:



This is the object code format of a store instruction; operand 1 is stored in the effective address of operand 2. The contents of base register 5, the displacement value 100, and the contents of index register 7 are added to obtain the effective address into which the contents of register 4 are stored. Each register requires 32 bits or 4 bytes of main storage.

This example assumes that the flags in the relocation register pertaining to the destination operand are 0.

Before execution:

Contents of register 4	32_{16}
Contents of register 5	48_{16}
Contents of register 7	52_{16}
Contents of main storage locations 19A through 19D, right justified	100_{16}

After execution:

Contents of the registers are unchanged

Effective address is $48 + 100 + 52 = 19A_{16}$

Contents of main storage locations 19A through 19D, right justified 32_{16}

Where an operand is described in terms of a main storage address and a length, the expression used can be simplified from that shown in the instruction format by implying the base register or length. Information supplied in the USING and DROP directives enables the assembler to separate a main storage address into a base register and a displacement (12.4). Table 3–2 lists the complete specification for the operand referencing main storage, applicable instruction types, and the operand format as it can be written utilizing an implied base register or length representation.

Table 3–2. Operand Specification Using Implied Base Register, Implied Length, or No Index Register

Applicable Instruction Type	Complete Specification for One Operand	Operand Specification Using		
		Implied Base Register	Implied Length or No Index Register*	Implied Base Register and Implied Length or No Index Register*
RR	r_1	NA	NA	NA
	r_2	NA	NA	NA
RX	$d_2(x_2,b_2)$	$s_2(x_2)$	$d_2(,b_2)$	s_2
RS	$d_2(b_2)$	s_2	NA	NA
SI	$d_1(b_1)$	s_1	NA	NA
SS	$d_1(l,b_1)$	$s_1(l)$	$d_1(,b_1)$	s_1
	$d_2(b_2)$	s_2	NA	NA
	$d_1(l_1,b_1)$	$s_1(l_1)$	$d_1(,b_1)$	s_1
	$d_2(l_2,b_2)$	$s_2(l_2)$	$d_2(,b_2)$	s_2

* The index register cannot be implied. If used, it must be specified as part of the operand.

LEGEND:

NA = Not applicable
s_1 = Symbolic expression — operand 1
s_2 = Symbolic expression — operand 2

3.2.1. Implied Length

The implied length of an instruction operand is only applicable to the SS instructions. To imply a length, the programmer specifies no length for the operand. The assembler automatically assembles the length attribute of the first operand into the length field of the instruction. The length attribute of an operand is the length attribute of the expression which is used to define the storage location. The length attribute of an expression is equal to the length attribute of its first term; the length of a self-defining term is 1.

Some SS instructions contain two length fields (one for each operand), of which one or both can be implied. In either case, the assembler puts the operand length in the length field.

The following instructions are examples of using the implied length feature of the assembler:

- To move 80 characters from a field labeled OP2 (operand 2) defined as a 90-character field to a field labeled OP1 (operand 1) defined as an 80-character field, the instruction is written:

  ```
  MVC OP1,OP2
  ```

  In the above instruction, the length attribute of OP1 was used. In the following example, 80 characters are still moved from OP2 to OP1 but the length is explicitly stated:

  ```
  MVC OP1(80),OP2
  ```

- If all 90 characters of OP2 are to be moved to OP1, which is defined as 80 characters in length, the instruction would be written with an explicit length so that the length attribute of OP1 is not used.

  The instruction is written:

  ```
  MVC OP1(90),OP2
  ```

3.2.2. Implied Base Registers

Information supplied in the USING and DROP directives (12.4) enables the assembler to separate a main storage address into a base register and a displacement value.

The assembler maintains a USING table of the available registers and the values they are to contain at object time. A USING directive adds a register and a 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 USING table. If the operands of a USING and DROP directive are not valid, that line of the listing is flagged with an error indication.

If an operand address is given as an effective address instead of a base register and displacement specification, the assembler searches the USING table for a value which yields a valid displacement (a value from 0 through 4095) and has the same relocatability attribute. If there is more than one valid displacement value, the value which yields the smallest displacement is used. If more than one register contains a value which yields the smallest displacement, the highest numbered register is selected. If no register can be found which yields a valid displacement, the field is set to 0 and the statement is flagged with an error indication. An absolute address without a base register indication is treated as an effective address and the assembler attempts to convert it into a base register and a displacement value.

The specification of a USING directive indicates that one or more general registers are available for use as base registers. The USING directive operands also state a value which the assembler assumes to be in the base registers at object time.

The value assigned to a register by the USING directive is used by the assembler in the assignment of main storage addresses. The value assigned is assumed to be in the respective base registers at execution time. The effective addresses are then derived by the hardware at execution time by adding the contents of the base register to the displacement value when the instruction is processed.

3.3. PRIVILEGED OPERATION

The assembler instructions may be divided into two categories according to their mode of operation:

- Nonprivileged Instructions

  The nonprivileged, or problem, instructions are available to both user programmer and system software for normal data processing when the processor is in the problem state; that is, the proper bit of the program status word (PSW) is set. If the program attempts to execute a privileged instruction, a program exception occurs.

- Privileged Instructions

  The privileged, or supervisor, instructions are so named because their execution permits special priority processor activity. When the processor is in the supervisor state, that is, when the proper bit of the program status word (PSW) is not set, all instructions are valid. All privileged instructions are designated as such in this manual.

The processor may be switched from one state to the other by providing a new program status word with the proper bit set appropriately. This may be accomplished by executing the load-PSW instruction (9.6). Since the load-PSW instruction is a privileged instruction, the processor must have been in a supervisor state. The switching of status may also occur as a result of an interrupt condition which causes a new program status word to be obtained from main storage.

3.4. PRESENTATION OF INSTRUCTIONS

Sections 4 through 10 describe each instruction in the assembler repertoire. The instructions are grouped in sets according to type:

- Fixed-point instructions
- Decimal instructions
- Floating-point instructions
- Logical instructions
- Branching instructions
- Status switching instructions
- Input/output instructions

The description of each instruction is presented in the following format:

- Instruction name — unless otherwise specified, the instruction applies to both the 9400/9480 and 90/60,70 environments
- Symbolic representation, hexadecimal operation code, format type, and length
- Function — the operation performed by the instruction
- Object instruction format — specified only for those instructions which are exceptions to the formats shown in Figure 3–1

- Operational considerations
  - detailed operation information
  - restrictions on use
  - condition code settings
  - possible program exceptions
  - applicable relocation and indirection flags (90/60,70 systems). Figure 3–2 lists and defines the abbreviations used for relocation and indirection flags as well as illustrating their respective placement in the format for the relocation register.
- Examples

Appendix A contains a list of the assembler instructions in alphabetical order according to mnemonic operation code.

Appendix B describes the hardware differences between the SPERRY UNIVAC 9400/9480 Systems and the SPERRY UNIVAC 90/60,70 Systems.

The basic concepts of address relocation and of indirect addressing are described in 4.6 of the processor manual, UP-7936 (current version).

Flags (bits 3–7)

Bit	Name	
3	R1	branch and relative instruction fetch addresses 0 = absolute addresses 1 = relative addresses
4	ID	indirect destination operand control 0 = direct addresses 1 = IACW addresses
5	RD	relative destination operand control 0 = absolute 1 = relative
6	IO	indirect origin operand control 0 = direct addresses 1 = IACW addresses
7	RO	relative origin operand control 0 = absolute addresses 1 = relative addresses

Offset (bits 8–19)

A 12-bit relocation value.

Figure 3–2. Relocation Register Format

4. Fixed-Point Instructions

4.1. GENERAL

The fixed-point instruction set provides for loading, storing, adding, subtracting, multiplying, dividing, comparing, shifting, and sign control of fixed-point operands on the SPERRY UNIVAC 9400/9480 Systems, and 90/60,70 Systems. See 1.3.1 for information concerning the manner in which fixed-point numbers are represented and their sign code established. Radix conversion of fixed-point operands is provided on the 90/60,70 systems. Unless otherwise noted, both operands are treated as 32-bit signed integers. Negative quantities are always represented in twos complement notation. A 0 result is always represented with a positive sign.

Fixed-point instructions are available in the RR, RX, RS, and SI formats. With the exception of the add- immediate instruction, at least one of the operands is contained in one of the 16 general registers. The other operand may be contained in main storage, in the same general register, or in another register. An operand address in main storage may be specified as relative or absolute on the 9400/9480 systems. On the 90/60,70 systems, an operand address in main storage may be specified as relative or absolute and direct or indirect under the control of the applicable relocation register flags. Unless the first and second operands are contained in the same register, or otherwise noted, the contents of the second operand location remain unchanged by the execution of the instruction.

This section describes the operation of each fixed-point instruction. The instructions are arranged in alphabetical order according to mnemonic operation code. Each description includes a list of the possible program exceptions and condition codes which may result. The relocation and indirection flags that are pertinent to the operand addresses are listed. The object code format of the instruction is shown only for those instructions which differ from the format shown in Figure 3–1. See Table 3–1 for an explanation of the abbreviations used in describing instruction formats and Figure 3–2 for relocation flag abbreviations.

4.2. A (ADD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
A	$r_1,d_2(x_2,b_2)$	5A	RX	Four Bytes

Function:

The contents of the full-word operand 2 in main storage at the address, specified by $d_2(x_2,b_2)$, are algebraically added to the general register specified by r_1.

Operational Considerations:

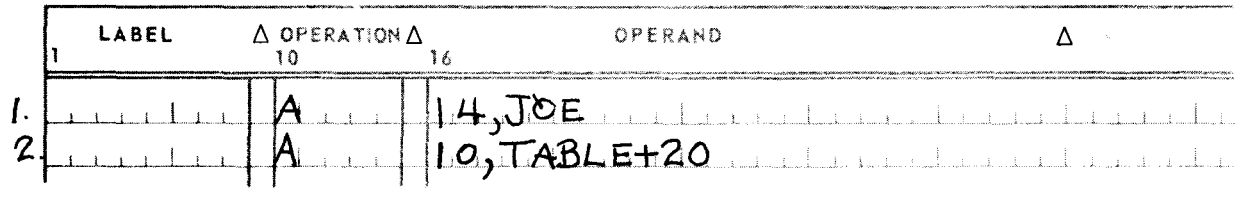
- All 32 bits of both operands are added. An overflow condition exists when the carry out of the sign bit position and the most significant numeric bit position disagree. After overflow, the sign and magnitude of the result are incorrect.

- Operand 2 must be on a full-word boundary.
- The contents of operand 2 remain unchanged.
- An overflow interrupt may be inhibited if bit 36 of the PSW is set to 0.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Fixed-point overflow	Binary overflow
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The contents of the full word in main storage location JOE are added to register 14.
2. The contents of the full word in main storage addressed by TABLE+20 are added to register 10.

4.3. AH (ADD-HALF-WORD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AH	$r_1,d_2(x_2,b_2)$	4A	RX	Four Bytes

Function:

The contents of the half-word operand 2 in main storage at the address, specified by $d_2(x_2,b_2)$, are expanded to a full word by propagating the sign bit value through the 16 most significant bit positions. The operand is then algebraically added to the general register, specified by r_1. The result is stored in operand 1.

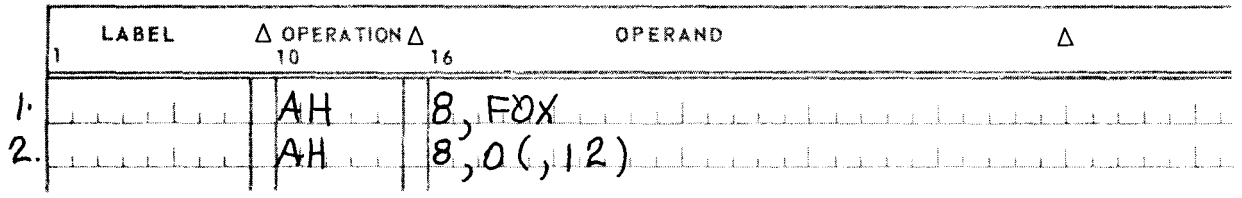
Operational Considerations:

- All 32 bits of both operands are added. An overflow condition exists when the carry out of the sign bit position and the most significant numeric bit position disagree. After overflow, the sign and magnitude of the result are incorrect.
- The contents of operand 2 remain unchanged.
- An overflow interrupt may be inhibited if bit 36 of the PSW is set to 0.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Fixed-point overflow	Binary overflow
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification 1. Operand 2 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The contents of the half word in main storage location FOX are added to register 8.
2. The effective address is obtained by adding the contents of base register 12 to 0. The contents of this address are then added to register 8.

4.4. AI (ADD-IMMEDIATE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AI	$d_1(b_1),i_2$	93 (9400/9480) 9A (90/60,70)	SI	Four Bytes

Function:

The binary value of operand 2, contained in i_2 field, is algebraically added to the contents of the half-word operand 1 in main storage at the address specified by $d_1(b_1)$. The result is stored in operand 1.

Operational Considerations:

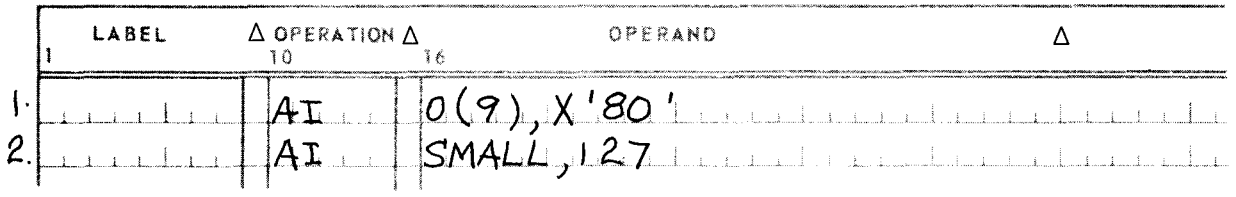
- An overflow condition exists when the carry out of the sign bit position and the most significant numeric bit positions disagree. After overflow, the sign and magnitude of the result are incorrect.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Fixed-point overflow	Binary overflow
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification 1. Operand 1 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: none

Examples:



1. The value −128, specified by 80_{16}, is added to the contents of main storage specified by the displacement 0 modified by the contents of the base register 9.
2. The value 127 is added to the contents of the operand labeled SMALL.

4.5. AR (ADD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AR	r_1,r_2	1A	RR	Two Bytes

Function:

The 32 bits of operand 2, specified by r_2, are added to the 32 bits of operand 1, specified by r_1. The sum is stored in operand 1.

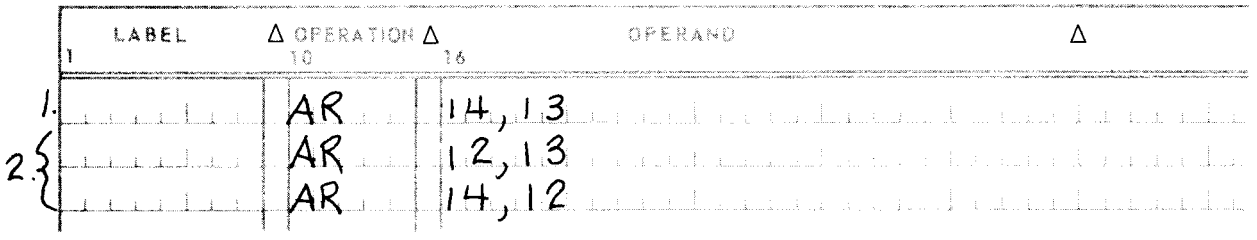
Operational Considerations:

- All 32 bits of both operands are added. An overflow condition exists when the carry out of the sign bit position and the most significant numeric bit position disagree. After overflow, the sign and magnitude of the result are incorrect.
- The contents of r_2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Fixed-point overflow	Binary overflow

- Relocation and indirection flags (90/60,70): none

Examples:



1. The contents of register 13 are added to the contents of register 14.
2. The contents of registers 12, 13, and 14 are added and the sum is placed in register 14. The sum of the contents of registers 12 and 13 is stored in register 12.

4.6. C (COMPARE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
C	$r_1,d_2(x_2,b_2)$	59	RX	Four Bytes

Function:

The full-word operand 1, specified by r_1, is algebraically compared with the full-word operand 2, specified by $d_2(x_2,b_2)$.

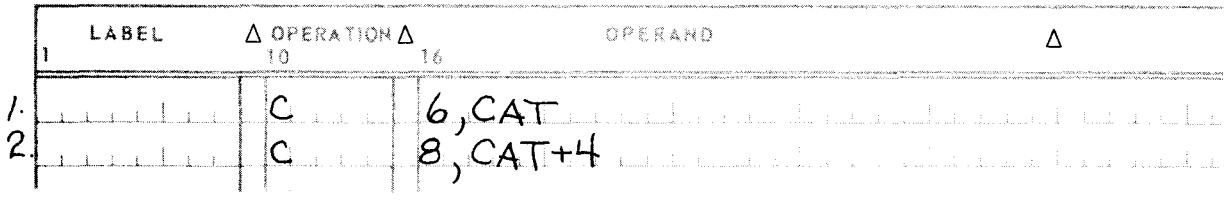
Operational Considerations:

- The contents of both operands remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if the operands are equal;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The contents of register 6 are compared with the full word in main storage location CAT.
2. The contents of register 8 are compared with the full word in main storage labeled CAT+4.

4.7. CH (COMPARE-HALF-WORD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CH	$r_1,d_2(x_2,b_2)$	49	RX	Four Bytes

Function:

The half-word operand 2 specified by $d_2(x_2,b_2)$, is expanded to a full word by propagating the sign bit value through the 16 most significant bits. The full-word operand 1, specified by r_1, is then algebraically compared with operand 2.

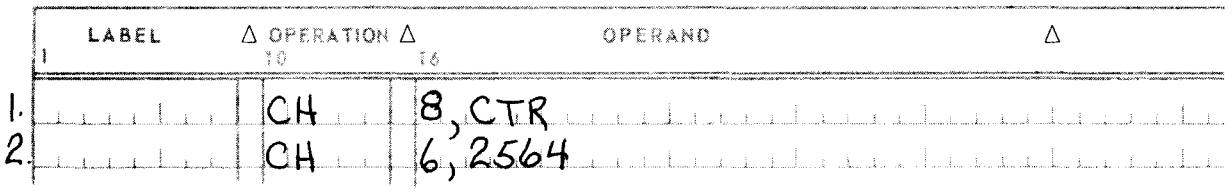
Operational Considerations:

- The contents of both operands remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if the operands are equal;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification 1. Operand 2 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The contents of register 8 are compared with the half word in main storage location CTR.

2. The contents of register 6 are compared with the contents of the half word in main storage location 2564.

4.8. CR (COMPARE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CR	r_1,r_2	19	RR	Two Bytes

Function:

The full-word operand 1, specified by r_1, is algebraically compared with the full-word operand 2, specified by r_2.

Operational Considerations:

- The contents of both operands remain unchanged.
- The condition code is set as follows:

- to 0 (00_2) if the operands are equal;
- to 1 (01_2) if operand 1 is less than operand 2;
- to 2 (10_2) if operand 1 is greater than operand 2; or
- code 3 is not used.

- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:

1. The contents of register 6 are compared with the contents of register 7.

4.9. CVB (CONVERT-TO-BINARY)–90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CVB	$r_1,d_2(x_2,b_2)$	4F	RX	Four Bytes

Function:

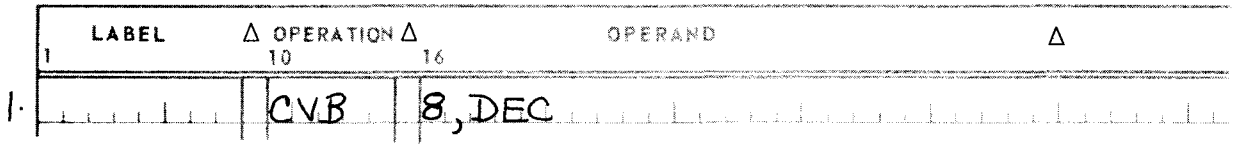
The double-word operand 2, specified by $d_2(x_2,b_2)$, is converted from a packed decimal number to a binary number and placed in the operand 1 location, specified by r_1.

Operational Considerations:

- Operand 2 is a signed 15-digit packed decimal number contained in a double-word main storage location. It must begin on a double-word boundary. The number is checked for valid sign and digit code before conversion to a 32-bit signed integer.
- The maximum number which can be converted and still contained in a 32-bit register is 2,147,483,647. The minimum number is –2,147,483,648. For decimal numbers exceeding this range, the 32 least significant bits are stored in the first operand location and a fixed-point divide exception is generated.
- If operand 2 is negative, the least significant 32 bits of the result are in twos complement notation.
- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.

- Possible program exceptions:
  - addressing exception
  - data exception (invalid sign or digit)
  - fixed-point divide exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (operand 2 not on double-word boundary or IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The packed decimal number located at main storage location DEC is converted to binary and placed in register 8.

4.10. CVD (CONVERT-TO-DECIMAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CVD	$r_1,d_2(x_2,b_2)$	4E	RX	Four Bytes

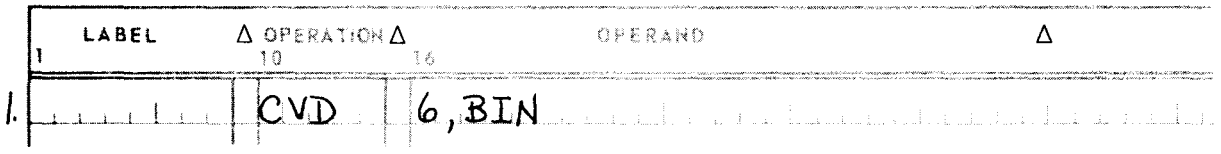
Function:

The full-word operand 1, specified by r_1, is converted from a binary number to a packed decimal number and placed in the double-word operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- Operand 1 is treated as a 32-bit signed integer. It is converted to a signed 15-digit packed decimal number and placed in a double-word main storage location. The location must begin on a double-word boundary.
- The contents of the operand 1 register remain unchanged.
- The low order four bits of the result represent the sign.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (operand 2 not on double-word boundary or IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RD, ID

Example:



1. The contents of register 6 are converted to a packed decimal number and placed in the double-word storage location BIN.

4.11. D (DIVIDE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
D	$r_1,d_2(x_2,b_2)$	5D	RX	Four Bytes

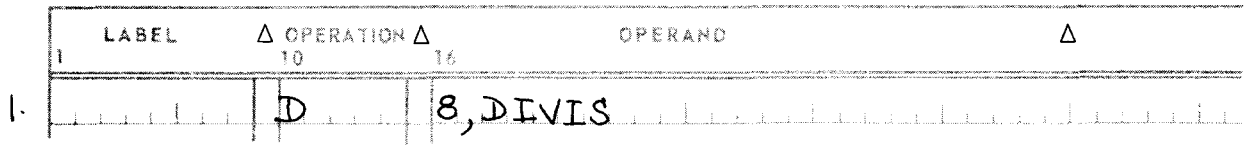
Function:

The double-word operand 1 (the dividend), specified by r_1, is divided by the full-word operand 2 (the divisor), at the address specified by $d_2(x_2,b_2)$. The quotient and remainder are stored in operand 1.

Operational Considerations:

- Operand 1 is treated as a 64-bit signed integer and occupies an even-odd register pair. The operand 1 field must specify an even-numbered register. The 32-bit remainder and 32-bit quotient replace the dividend in the even-numbered and odd-numbered register, respectively.
- Operand 2 is treated as a 32-bit signed integer. The contents of operand 2 remain unchanged after execution.
- The sign of the quotient is determined algebraically and the remainder assumes the sign of the dividend. A 0 quotient or 0 remainder is always positive.
- When the quotient exceeds 32 bits or the divisor is equal to 0, a fixed-point divide exception occurs, no division takes place, and the dividend remains unchanged.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - fixed-point divide exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary; or operand 1 field specifies an odd-numbered register)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The contents of registers 8 and 9 are divided by the contents of the full word specified by DIVIS. The quotient is stored in register 9; the remainder, in register 8.

4.12. DR (DIVIDE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DR	r_1,r_2	1D	RR	Two Bytes

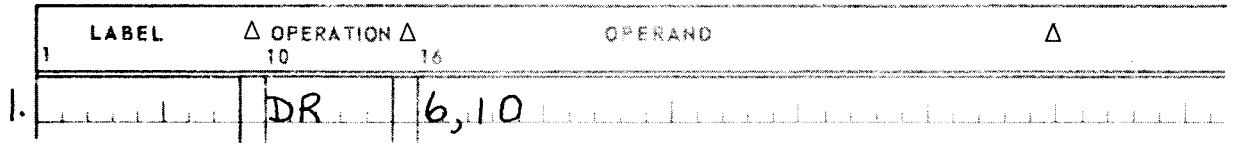
Function:

The double-word operand 1 (the dividend), specified by r_1, is divided by the full-word operand 2 (the divisor), specified by r_2. The quotient and remainder replace operand 1.

Operational Considerations:

- Operand 1 is treated as a 64-bit signed integer and occupies an even-odd register pair. The operand 1 field must specify an even-numbered register. The 32-bit remainder and 32-bit quotient replace the dividend in the even-numbered and odd-numbered register, respectively.
- Operand 2 is treated as a 32-bit signed integer. The contents of operand 2 remain unchanged after execution.
- The sign of the quotient is determined algebraically and the remainder assumes the sign of the dividend. A 0 quotient or 0 remainder is always positive.
- When the quotient exceeds 32 bits or the divisor is equal to 0, a fixed-point divide exception occurs, no division takes place, and the dividend remains unchanged.
- The condition code remains unchanged.
- Possible program exceptions:
  - fixed point divide exception
  - specification exception (operand 1 field specifies an odd-numbered register)
- Relocation and indirection flags: none

Example:



1. The contents of registers 6 and 7 are divided by the contents of register 10. The quotient is stored in register 7; the remainder, in register 6.

4.13. L (LOAD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
L	$r_1,d_2(x_2,b_2)$	58	RX	Four Bytes

Function:

The contents of the full-word operand 2 in main storage at the address specified by $d_2(x_2,b_2)$, are transferred to the operand 1 register, specified by r_1.

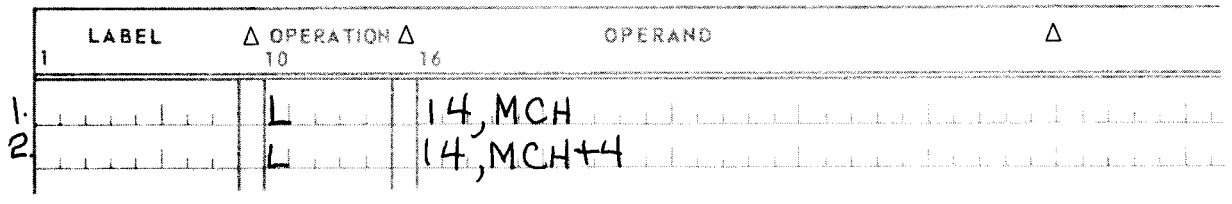
Operational Considerations:

- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The full word in main storage location MCH is loaded into register 14.
2. The four bytes following the four bytes addressed as MCH are loaded into register 14.

4.14. LCR (LOAD-COMPLEMENT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LCR	r_1,r_2	13	RR	Two Bytes

Function:

The twos complement of the full-word operand 2, specified by r_2, is stored in the operand 1 location, specified by r_1.

Operational Considerations:

- The contents of the operand 2 register remain unchanged.
- A fixed-point overflow condition exists when the maximum negative number is complemented; the number remains unchanged.
- Zero remains unchanged under complementation.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:
  - fixed-point overflow exception
- Relocation and indirection flags: none

Example:

```
        LABEL     Δ OPERATION Δ          OPERAND              Δ
        1         10          16
1.                  LCR         7,8
```

1. The twos complement of the contents of register 8 is loaded into register 7.

4.15. LH (LOAD-HALF-WORD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LH	$r_1,d_2(x_2,b_2)$	48	RX	Four Bytes

Function:

The half-word operand 2, specified by d_2 (x_2,b_2), is expanded to a full word by propagation of the sign bit through the 16 most significant bit positions. The resulting full word is stored in operand 1, specified by r_1.

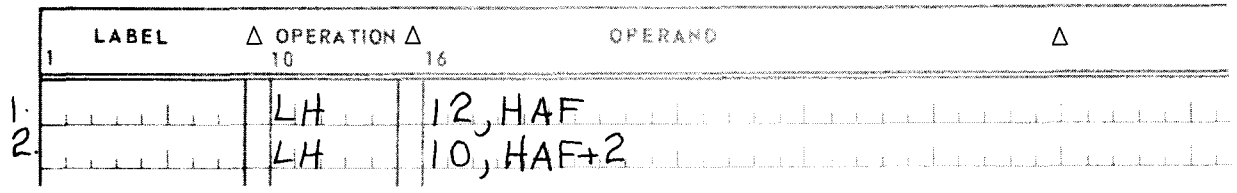
Operational Considerations:

- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification 1. Operand 2 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags 90/60,70:
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The two bytes in main storage location HAF are expanded to a full word and placed in register 12.
2. The two bytes in main storage location HAF+2 are expanded to a full word and placed in register 10.

4.16. LLR (LOAD-LIMITS-REGISTER) – PRIVILEGED INSTRUCTION – 9400/9480

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LLR	$d_2(b_2)$	81	RS	Four Bytes

Function:

The limits registers are loaded with the half word in main storage specified in operand 1, $d_2(b_2)$. Bits 0–7 are loaded into the upper limits register, bits 8–15 into the lower limits register. Treated as no-op when the storage protection feature is not installed.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The instruction loads the half-word operand from main storage specified by $d_2(b_2)$ into the limits register.
- The r_1 and r_3 fields of this instruction are ignored.
- The condition code is set as follows:
  - to 0 (00) if write protection feature is installed;
  - to 1 (01) if write protection feature is not installed;
  - codes 2 (10) and 3 (11) are not used.
- Possible program exceptions:
  - privileged operation exception
  - specification exception

Example:

```
          LABEL     ΔOPERATIONΔ          OPERAND          Δ
1         1         10        16
1.                  LLR       LMT1
```

1. The contents of the half word labeled LMT1 are loaded into the hardware limits registers.

4.17. LM (LOAD-MULTIPLE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LM	$r_1,r_3,d_2(b_2)$	98	RS	Four Bytes

Function:

The general registers starting with the operand 1 register, specified by r_1, and ending with the operand 3 register, specified by r_3, are loaded with full-word main storage operands beginning with operand 2 address, specified by $d_2(b_2)$.

Operational Considerations:

- The registers are loaded in ascending numeric sequence beginning with the operand 1 register and continuing through the operand 3 register.
- One register may be loaded by specifying the same register number for operand 1 and operand 3.
- If the operand 3 specification is lower than the operand 1 specification, all registers with a number greater than or equal to operand 1 and all registers with a number less than or equal to operand 3, are loaded.
- The contents of main storage specified by operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:

	LABEL	△ OPERATION △	OPERAND
1.		LM	6,10,4096
2		LM	8,6,DATA
3.		LM	5,5,COUNT

1. Registers 6, 7, 8, 9, and 10 are loaded with full words beginning with location 4096.

2. Registers 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, and 6 are loaded, in this order, with full words beginning at main storage location DATA.

3. Register 5 is loaded with the full word at main storage location COUNT.

4.18. LNR (LOAD-NEGATIVE) – 90/ 60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LNR	r_1,r_2	11	RR	Two Bytes

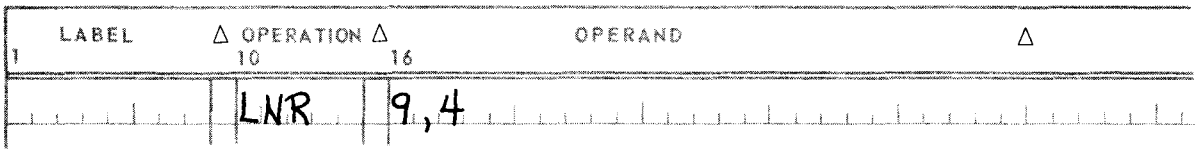
Function:

The twos complement of the absolute value of the full-word operand 2, specified by r_2, is loaded into operand 1, specified by r_1.

Operational Considerations:

- The contents of the operand 2 register remain unchanged.
- Positive numbers are complemented; negative numbers remain unchanged.
- Zero remains unchanged under complementation.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions: none
- Relocation and indirection flags: none

Example:



1. The twos complement of the absolute value of the contents of register 4 is placed in register 9.

4.19. LPR (LOAD-POSITIVE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LPR	r_1,r_2	10	RR	Two Bytes

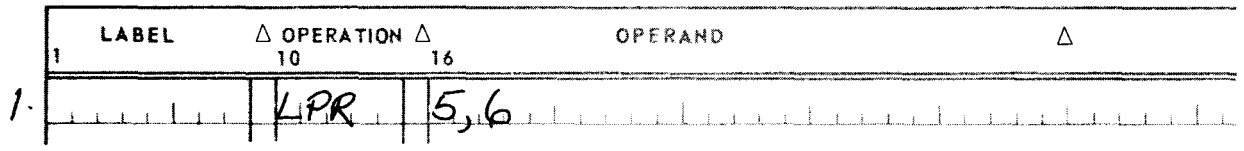
Function:

The absolute value of the full-word operand 2, specified by r_2, is placed into operand 1, specified by r_1.

Operational Considerations:

- The contents of the operand 2 register remain unchanged.
- Positive numbers are unchanged by this operation.
- When operand 2 is negative, the twos complement is placed in the operand 1 location.
- When operand 2 contains the maximum negative number, a fixed-point overflow condition exists and the number remains unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - code 1 is not used;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:
  - fixed-point overflow exception
- Relocation and indirection flags: none

Example:



1. The absolute value of the contents of register 6 is loaded into register 5.

4.20. LR (LOAD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LR	r_1,r_2	18	RR	Two Bytes

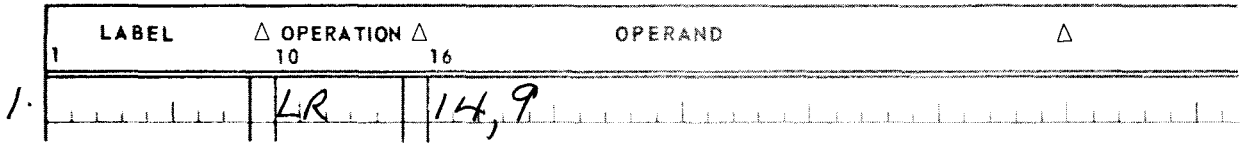
Function:

The contents of the full-word operand 2, specified by r_2, are transferred to the operand 1 register specified by r_1.

Operational Considerations:

- The contents of the operand 2 register remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:



1. The full word in register 9 is loaded into register 14.

4.21. LTR (LOAD-AND-TEST)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LTR	r_1,r_2	12	RR	Two Bytes

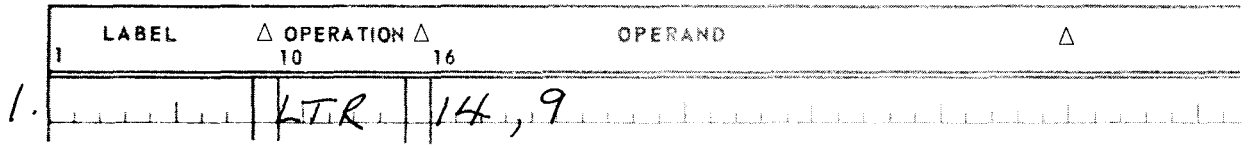
Function:

The contents of the full-word operand 2 register, specified by r_2, are transferred to the operand 1 register, specified by r_1. the condition code is set.

Operational Considerations:

- The contents of the operand 2 register remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none:

Example:



1. The full word in register 9 is loaded into register 14 and the condition code is set.

4.22. M (MULTIPLY) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
M	$r_1,d_2(x_2,b_2)$	5C	RX	Four Bytes

Function:

The full-word operand 1 (multiplicand), specified by r_1, is multiplied by the full-word operand 2 (multiplier), specified by $d_2(x_2,b_2)$, and the product is stored in operand 1.

Operational Considerations:

- Both operands are treated as 32-bit signed integers.
- The contents of operand 2 remain unchanged.

- The product is treated as a 64-bit signed integer and occupies an even-odd register pair; therefore, the operand 1 register must specify an even-numbered register. The first operand is taken from the odd-numbered register. The contents of the even-numbered register are ignored until replaced by the most significant 32 bits of the product.

- The sign of the product is determined algebraically.

- The condition code remains unchanged.

- Possible program exceptions:

  – addressing exception

  – indirect address specification exception

  – indirect addressing exception

  – protection exception

  – specification exception (operand 2 or IACW not on full-word boundary; or operand 1 specifies an odd register address).

- Relocation and indirection flags:

  – operand 1: none

  – operand 2: RO, IO

Example:

	LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
1.		M	6,MULT

1. The contents of register 7 are multiplied by the contents of the full word in main storage location MULT. The product is placed in registers 6 and 7.

4.23. MH (MULTIPLY-HALF-WORD) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MH	$r_1,d_2(x_2,b_2)$	4C	RX	Four Bytes

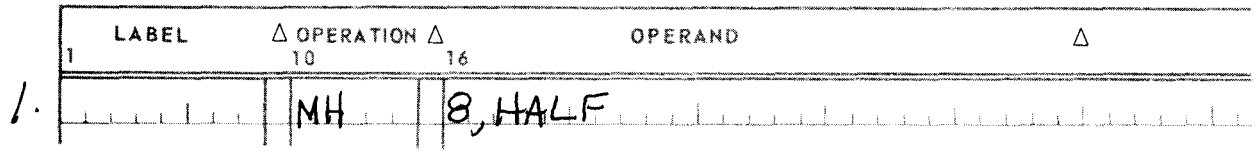
Function:

The half-word operand 2, specified by $d_2(x_2,b_2)$, is expanded to a full word by propagation of the sign bit through the 16 most significant bit positions. Operand 1 (the multiplicand), specified by r_1, is then multiplied by operand 2 (the multiplier) and the product is stored in operand 1.

Operational Considerations:

- Both operands are treated as 32-bit signed integers.
- The product has a length of 47 or fewer bits. After multiplication, the least significant 32 bits of the product are placed in the operand 1 location. If the product exceeds 32 bits, the most significant bits are ignored and an overflow condition is not indicated. The sign of the product may be incorrect when the most significant bits are lost.
- The sign of the product is determined algebraically.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (operand 2 not on half-word boundary or IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The contents of register 8 is multiplied by the half word in main storage location HALF and product is stored in register 8.

4.24. MR (MULTIPLY) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MR	r_1,r_2	1C	RR	Two Bytes

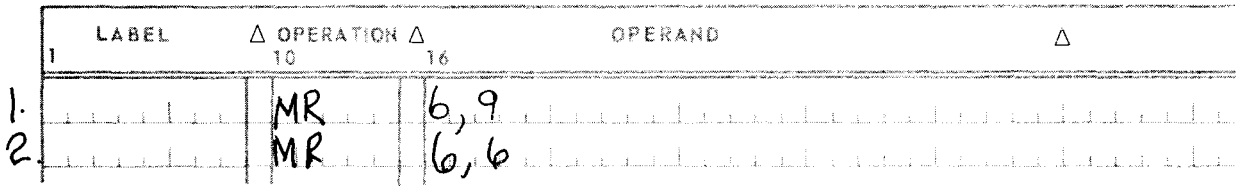
Function:

The full-word operand 1 (the multiplicand), specified by r_1, is multiplied by the full-word operand 2 (the multiplier), specified by r_2, and the product is stored in operand 1.

Operational Considerations:

- Both operands are treated as 32-bit signed integers. The product is treated as a 64-bit signed integer and occupies an even-odd register pair; therefore, the operand 1 specification must specify an even-numbered register. Operand 1 is taken from the odd-numbered register of the pair. The even-numbered register may contain operand 2; if not, the contents of the even-numbered register are ignored until replaced by the most significant 32 bits of the product.
- An overflow cannot occur.
- The sign of the product is determined algebraically.
- The condition code remains unchanged.
- Possible program exceptions:
  - specification exception (operand 1 specifies an odd-numbered register)
- Relocation and indirection flags: none

Examples:



1. The contents of register 7 are multiplied by the contents of register 9. The result is stored in registers 6 and 7.
2. The contents of register 7 are multiplied by the contents of register 6. The result is stored in registers 6 and 7.

4.25. S (SUBTRACT)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
S	$r_1,d_2(x_2,b_2)$	5B	RX	Four Bytes

Function:

The full-word operand 2, specified by $d_2(x_2,b_2)$, is subtracted from the full-word operand 1, specified by r_1, and the result is stored in operand 1.

Operational Considerations:

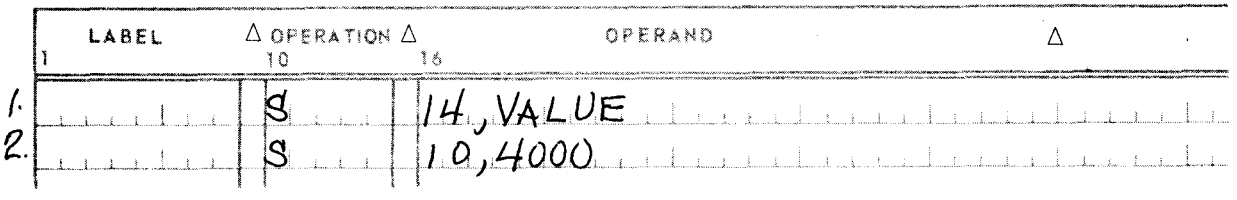
- The subtraction is performed by means of signed algebraic twos complement binary addition.

- All 32 bits of both operands are used. An overflow condition exists when the carry out of the sign bit position and the most significant numeric bit position disagree.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Fixed-point overflow	Binary overflow
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The contents stored in the the full word at main storage location VALUE are converted to a twos complement binary value and added to the contents of register 14.
2. The contents stored in the full word located at address 4000 are converted to a twos complement binary value and added to the contents of register 10.

4.26. SH (SUBTRACT-HALF-WORD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SH	$r_1,d_2(x_2,b_2)$	4B	RX	Four Bytes

Function:

The half-word operand 2, specified by $d_2(x_2,b_2)$, is expanded to a full word by propagation of the sign bit value through the 16 most significant bit positions. Operand 2 is then subtracted from the full-word operand 1, specified by r_1, and the result is placed in operand 1.

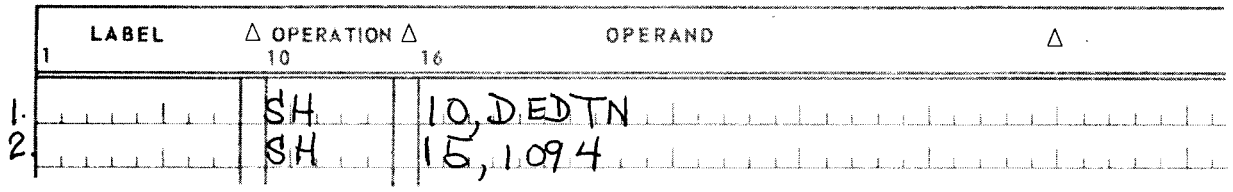
Operational Considerations:

- The subtraction is performed by means of signed algebraic twos complement binary addition.
- All 32 bits of both operands are used. An overflow condition exists when carry out of the sign bit position and the most significant numeric bit position disagree.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Fixed-point overflow	Binary overflow
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification 1. Operand 2 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. Subtract the contents stored in the half word located at DEDTN from the contents of register 10.
2. Subtract the contents stored in the half word located at address 1094 from the contents of register 15.

4.27. SLA (SHIFT-LEFT-SINGLE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SLA	$r_1,d_2(b_2)$	8B	RS	Four Bytes

Function:

The 31-bit integer operand 1, specified by r_1, is shifted left the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

Object Instruction Format:

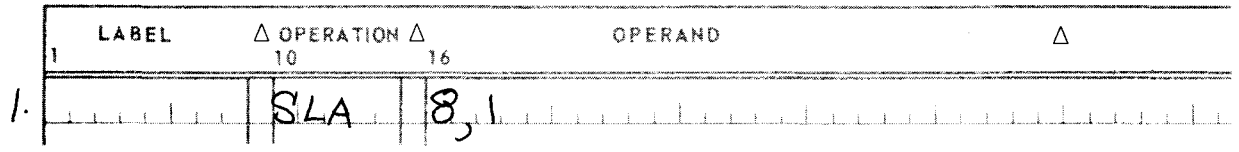
OPERATION CODE 0–7	OPERAND 1 8–11	OPERAND 3 12–15	OPERAND 2 16–19	OPERAND 2 20–31
8B	r_1	unused	b_2	d_2

Operational Considerations:

- The vacated least significant bit positions of the operand 1 register are zero filled.
- The sign bit remains unchanged.
- If a bit not equal to the sign bit is shifted out of the most significant numeric bit position, a fixed-point overflow condition exists.
- For numbers with an absolute value of less than 2^{30}, a left shift of one bit position is equivalent to multiplying the number by 2.
- Shift amounts from 31 to 63 cause the entire integer to be shifted out of the register. When the entire integer field for a positive number has been shifted out, the register contains a value of 0; for a negative number, the register contains -2^{31}.

- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:
  - fixed-point overflow exception
- Relocation and indirection flags: none

Example:



1. The contents of register 8 are shifted to the left one bit position.

4.28. SLDA (SHIFT-LEFT-DOUBLE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SLDA	$r_1,d_2(b_2)$	8F	RS	Four Bytes

Function:

The 63-bit integer operand 1, specified by r_1, is shifted left the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

Object Instruction Format:

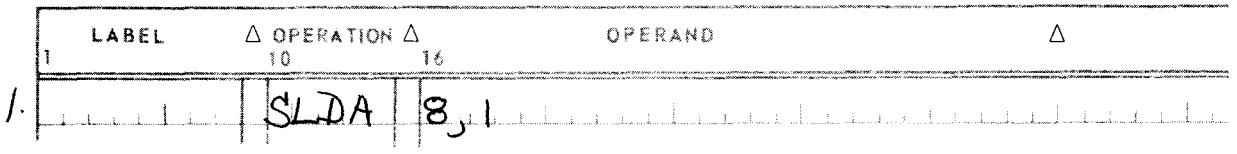
0 OPERATION CODE 7	8 OPERAND 1 11	12 OPERAND 3 15	16 OPERAND 2 19	20 OPERAND 2 31
8F	r_1	unused	b_2	d_2

Operational Considerations:

- The r_1 specification in operand 1 must refer to the even-numbered register of an even-odd register pair. The contents of both registers, except the sign bit of the even-numbered register, are treated as a 63-bit integer.

- The vacated least significant bit positions of the register pair are zero filled.
- The sign bit of the even register remains unchanged.
- If a bit not equal to the sign bit is shifted out of the most significant numeric bit position of the even-numbered register, a fixed-point overflow condition exists.
- A 0 shift value provides a double-length sign and magnitude test.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:
  - Fixed-point overflow exception
  - Specification exception (operand 1 specifies an odd register number)
- Relocation and indirection flags: none

Example:



1. The contents of register 8 and register 9, taken as a 63-bit integer, are shifted to the left one bit position.

4.29. SLM (SUPERVISOR-LOAD-MULTIPLE) – PRIVILEGED INSTRUCTION

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SLM	$r_1,r_3,d_2(b_2)$	B8	RS	Four Bytes

Function:

The problem general registers starting with the operand 1 register specified by r_1, and ending with the operand 3 register, specified by r_3, are loaded with full-word main storage operands beginning with the operand 2 address specified by $d_2(b_2)$.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- This instruction is similar to the load-multiple instruction (4.16) except that in the supervisor-load-multiple instruction the operands always refer to problem general registers even though the processor is in the supervisor state.
- The registers are loaded in ascending numeric sequence beginning with the operand 1 numbered register and continuing through the operand 3 numbered register.
- One register may be loaded by specifying the same register number for operand 1 and operand 3.
- If the operand 3 specification is lower than the operand 1 specification, all registers with a number greater than or equal to operand 1 and all registers with a number less than or equal to operand 3 are loaded.
- The contents of main storage specified by operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Privileged operation	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Examples:

```
      LABEL     Δ OPERATION Δ          OPERAND                    Δ
      1         10          16
1.                SLM         4,9,AREA
2.                SLM         9,2,DATA
```

1. Problem registers 4 through 9 are loaded with full words beginning at main storage location AREA.
2. Problem registers 9, 10, 11, 12, 13, 14, 15, 0, 1, and 2 are loaded with full words beginning at main storage location DATA.

4.30. SR (SUBTRACT)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SR	r_1,r_2	1B	RR	Two Bytes

Function:

The full-word operand 2, specified by r_2, is subtracted from the full-word operand 1, specified by r_1, and the result is stored in operand 1.

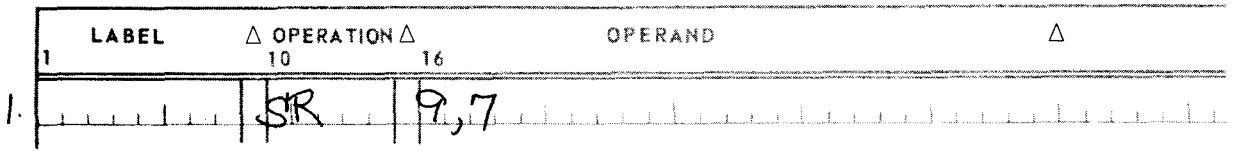
Operational Considerations:

- The subtraction is performed by means of signed algebraic twos complement binary addition.
- All 32 bits of both operands are used. An overflow condition exists when the carry out of the sign bit position and the most significant numeric bit position disagree.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Fixed point overflow	Binary overflow

- Relocation and indirection flags: none

Example:



1. The contents of register 7 are converted to a twos complement binary value and added to the contents of register 9.

4.31. SRA (SHIFT-RIGHT-SINGLE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SRA	$r_1,d_2(b_2)$	8A	RS	Four Bytes

Function:

The 31-bit integer operand 1, specified by r_1, is shifted right the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

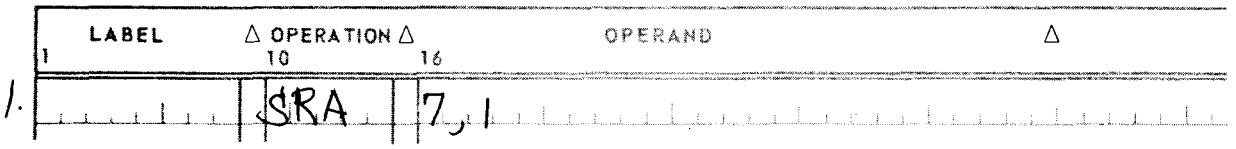
Object Instruction Format:

OPERATION CODE 0–7	OPERAND 1 8–11	OPERAND 3 12–15	OPERAND 2 16–19	OPERAND 2 20–31
8A	r_1	unused	b_2	d_2

Operational Considerations:

- The vacated high order bit positions of the operand 1 register are sign filled.
- The bits shifted out of the least significant bit position of the registers are lost.
- A right shift of one bit position is equivalent to division by 2 with rounding downward. When an even number is shifted right one position, the value of the field is obtained by dividing the value by 2. When an odd number is shifted right one bit position, the value of the field is obtained by subtracting 1, then dividing the value by 2. For example, 5 shifted right one bit position yields 2, whereas –5 yields –3.
- Shift values from 31 through 63 cause the entire integer field to be shifted out of the register. When the entire integer field of a positive number has been shifted out, the register contains a value of 0. For a negative number, the register contains a value of –1.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions: none
- Relocation and indirection flags: none

Example:



1. The contents of register 7 are shifted to the right one bit position.

4.32. SRDA (SHIFT-RIGHT-DOUBLE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SRDA	$r_1,d_2(b_2)$	8E	RS	Four Bytes

Function:

The 63-bit integer operand 1, specified by r_1, is shifted right the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

Object Instruction Format:

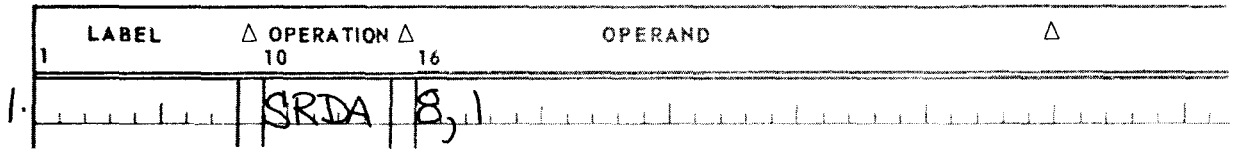
OPERATION CODE 0–7	OPERAND 1 8–11	OPERAND 3 12–15	OPERAND 2 16–19	OPERAND 2 20–31
8E	r_1	unused	b_2	d_2

Operational Considerations:

- The r_1 specification in operand 1 must refer to the even-numbered register of an even-odd register pair. The contents of both registers, except the sign bit of the even-numbered register, are treated as a 63-bit integer.
- The vacated most significant bit positions of the register pair are sign filled.
- A 0 shift value provides a double-length sign and magnitude test.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.

- Possible program exceptions:
  - Specification exception (operand 1 specifies an odd register number)
- Relocation and indirection flags: none

Example:



1. The contents of register 8 and register 9, taken as a 63-bit integer, are shifted to the right one bit position.

4.33. SSTM (SUPERVISOR-STORE-MULTIPLE) – PRIVILEGED INSTRUCTION

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SSTM	$r_1,r_3,d_2(b_2)$	B0	RS	Four Bytes

Function:

The contents of a group of problem general registers, starting with the operand 1 register, specified by r_1, ending with the operand 3 register, specified by r_3, are stored in the main storage location designated by operand 2, specified by $d_2(b_2)$.

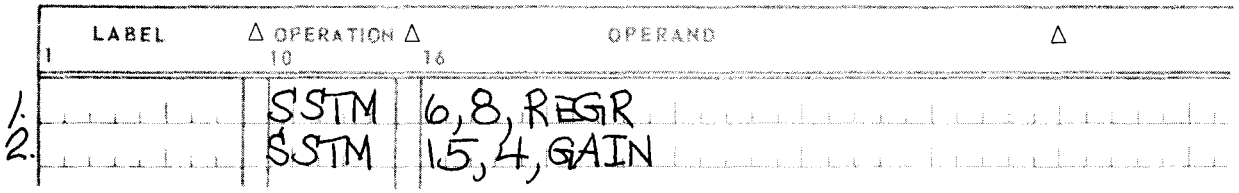
Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- This instruction is similar to the store-multiple instruction (4.35) except that in the supervisor-store-multiple instruction the operands always refer to problem general registers even though the processor is in the supervisor state.
- The contents of operand 1 through operand 3 remain unchanged.
- When the operand 3 specification is lower than the operand 1 specification, the register numbers wrap around from 15 to 0. For this reason, all possible combinations of operand 1 and operand 3 are valid.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	Storage protection
Privileged operation	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RD, ID

Examples:



1. The contents of registers 6, 7, and 8 are placed in the main storage location REGR.
2. The contents of registers 15, 0, 1, 2, 3, and 4 are placed in main storage, beginning at the location labeled GAIN.

4.34. ST (STORE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ST	$r_1,d_2(x_2,b_2)$	50	RX	Four Bytes

Function:

The contents of operand 1, specified by r_1, are stored in the main storage location operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- The contents of operand 1 remain unchanged.
- The condition code remains unchanged.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RD, ID

Example :

```
      LABEL     Δ OPERATION Δ        OPERAND                    Δ
      1         10          16
1.    |         | ST        | 7,RESULT
```

1. The contents of register 7 are placed in main storage location RESULT.

4.35. STH (STORE-HALF-WORD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
STH	$r_1,d_2(x_2,b_2)$	40	RX	Four Bytes

Function:

The least significant 16 bits of the contents of operand 1, specified by r_1, are stored in the half-word main storage location operand 2, specified by $d_2(x_2,b_2)$.

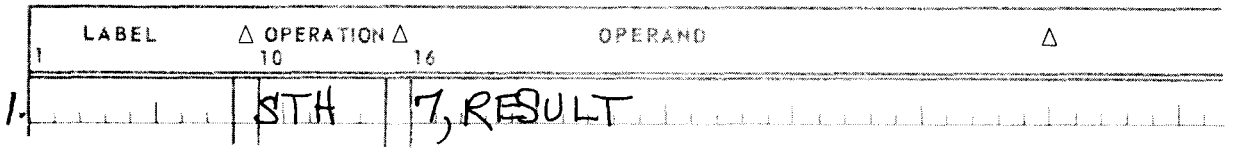
Operational Considerations:

- The contents of the operand 1 register remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification 1. Operand 2 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RD, ID

Example:



1. The least significant 16 bits of register 7 are placed in main storage location RESULT.

4.36. STM (STORE-MULTIPLE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
STM	$r_1,r_3,d_2(b_2)$	90	RS	Four Bytes

Function:

The contents of a group of general registers, starting with the operand 1 register specified by r_1, and ending with the operand 3 register specified by r_3, are stored in the main storage location designated operand 2, specified by $d_2(b_2)$.

Operational Considerations:

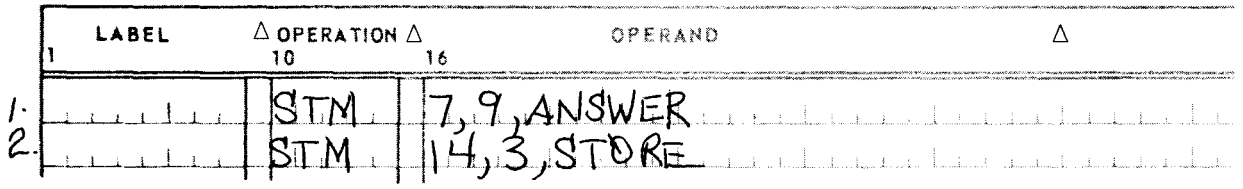
- The contents of operand 1 through operand 3 remain unchanged.
- When the operand 3 specification is lower than the operand 1 specification, the register numbers wrap around from 15 to 0. For this reason, all possible combinations of operand 1 and operand 3 are valid.
- The condition code remains unchanged.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RD, ID

Examples:



1. The contents of registers 7, 8, and 9 are placed in main storage, beginning at the location labeled ANSWER.

2. The contents of registers 14, 15, 0, 1, 2, and 3 are placed in main storage, beginning at the location labeled STORE.

5. Decimal Instructions

5.1. GENERAL

The decimal instruction set provides for adding, subtracting, multiplying, dividing, comparing, and format conversion of variable-length operands. Unless otherwise noted, operands are treated as signed decimal integers in packed format. See 1.3.4 for information concerning the manner in which decimal numbers are represented and their sign codes established.

All decimal instructions are represented in the SS format in which each operand is contained in main storage. On the SPERRY UNIVAC 9400/9480 Systems, each main storage address is absolute; on the SPERRY UNIVAC 90/60,70 Systems, each main storage address may be specified as relative or absolute and direct or indirect under the control of the applicable relocation register flags. The address resulting from the relocation and indirection designates the main storage address of the most significant byte of the operand. Operands are always processed from right to left (that is, least significant byte to most significant byte). If the operands are of unequal length, the shorter is considered to be extended with 0 digits. If most significant digits or carries are lost because the first operand field is too short to accommodate the result of a decimal operation, a decimal overflow exception is detected. Unless the first and second operands overlap, the contents of the second operand location in main storage remain unchanged by the execution of the instruction.

This section describes the operation of each decimal instruction. The instructions are arranged in alphabetical order according to the mnemonic operation code. Each description includes a list of the possible program exceptions and condition codes which may result. The relocation and indirection flags that are pertinent to the operand address are listed. See Table 3–1 for an explanation of the abbreviations used in describing instruction formats.

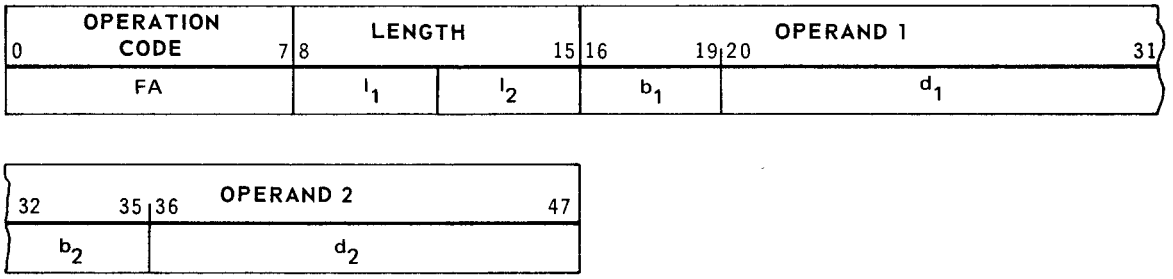
5.2. AP (ADD-DECIMAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AP	$d_1(l_1,b_1),d_2(l_2,b_2)$	FA	SS	Six Bytes

Function:

The contents of operand 2, specified by $d_2(l_2,b_2)$, are added to the contents of operand 1, specified by $d_1(l_1,b_1)$, and the result is stored in the operand 1 location.

Object Instruction Format:



Operational Considerations:

- Addition is performed from right to left.
- If operand 2 is shorter than operand 1, operand 2 is extended with 0 digits.
- An overflow condition results if the capacity of the operand 1 field is exceeded by the result or if the carry out of the most significant digit position of the result field is lost.
- Operand 1 and operand 2 may overlap if their least significant bytes coincide. This makes it possible to add a number to itself.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

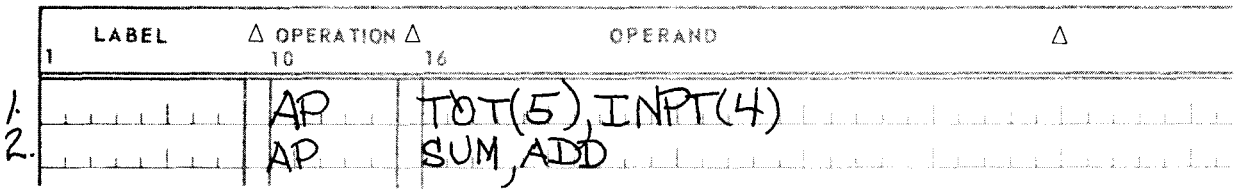
SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Decimal overflow	Decimal overflow
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	
Data exception 1. Invalid overlap 2. Invalid sign or digit code	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Operational Differences:

- 9400/9480 systems
  - If both operand 1 and operand 2 are unsigned, a positive sign is assumed.
  - In the case of overflow, the sign of the answer will be correct even if the answer is zero; a zero answer normally carries a plus sign.
  - If operand 2 is longer than operand 1, the remaining digits of operand 2 are ignored.
- 90/60,70 systems
  - All digits and signs are checked for validity; the sign of the result is determined algebraically.
  - In the case of overflow where the most significant digits are lost, the partial result has the sign which the complete result would have had; a zero result is positive when the operation is completed without overflow.
  - An interrupt may occur as a result of processing the significant digits.

Examples:



1. The 4-byte operand specified by the label INPT is added to the 5-byte operand specified by the label TOT. Assuming that all signs are positive, the contents of the operands may be represented as follows:

TOT before execution	0 9 7 6 9 8 1 3 5 +
INPT before and after execution	9 7 5 3 1 4 2 +
TOT after execution	1 0 7 4 5 1 2 7 7 +

2. The operand specified by the label ADD is added to the contents of the operand specified by the label SUM. The lengths of the operands are implied. The instruction is assigned operand lengths which are determined during the assembly process. The length attribute for each label is placed into the I field of the instruction.

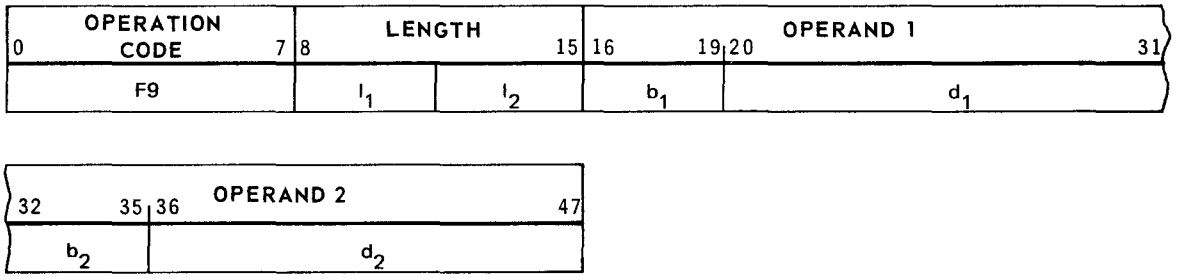
5.3. CP (COMPARE-DECIMAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CP	$d_1(l_1,b_1)d_2,(l_2,b_2)$	F9	SS	Six Bytes

Function:

The contents of operand 1, specified by $d_1(l_1,b_1)$, are algebraically compared with the contents of operand 2, specified by $d_2(l_2,b_2)$.

Object Instruction Format:



Operational Considerations:

- The comparison proceeds from right to left.
- Operands with 0 values and unlike signs compare as equal.
- All valid codes representing the same sign are considered equal.
- Operand 1 and operand 2 may overlap if their least significant bytes coincide.
- The contents of both operands remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if operand 1 equals operand 2;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

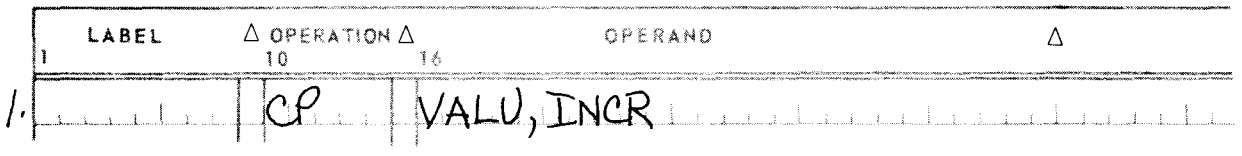
SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	
Data exception 1. Invalid overlap 2. Invalid sign or digit code	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Operational Differences:

- 9400/9480 systems
  - If operand 2 is longer than operand 1, the excess high order digits of operand 2 are ignored.
  - If operand 1 is longer than operand 2, data from operand 2 is zero filled to extend the operand.
- 90/60,70 systems
  - If the operand fields are unequal in length, the shorter field is zero filled to the length of the longer.
  - All signs and digits are checked for validity and the sign of the result is determined algebraically.

Example:



1. The contents of the location labeled VALU are compared with the contents of the location labeled INCR. The operand lengths are implied. The condition code is set.

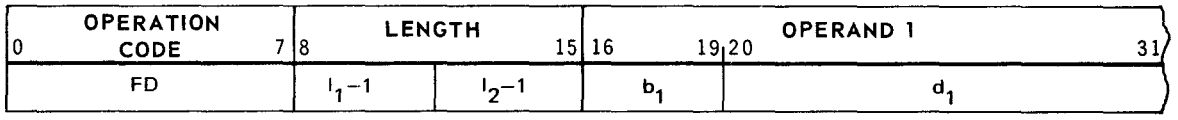
5.4. DP (DIVIDE-DECIMAL)

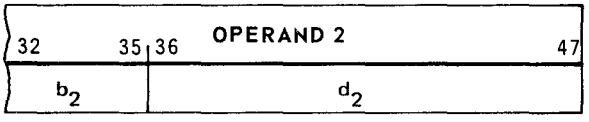
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DP	$d_1(l_1,b_1),d_2(l_2,b_2)$	FD	SS	Six Bytes

Function:

The contents of operand 1 (dividend), specified by $d_1(l_1,b_1)$, are divided by the contents of operand 2 (divisor), specified by $d_2(l_2,b_2)$. The quotient and remainder are stored in the operand 1 location.

Object Instruction Format:





Operational Considerations:

- The length of operand 1 specified by the l_1 field in the instruction is ignored. The length of operand 1 is determined by scanning operand 1 starting with the most significant digit until a sign code is found.
- The dividend (operand 1) must be longer than the divisor (operand 2).
- The quotient and remainder occupy the entire operand 1 field. The remainder is right-justified in the field, carries the sign of operand 1, and is equal in size to operand 2. The quotient, carrying the algebraically determined sign, is right-justified in the rest of the operand 1 field.
- The maximum dividend (operand 1) size is 31 digits and sign. The maximum quotient size is 29 digits and sign. The smallest remainder is one digit and sign.
- If the number of quotient digits exceeds the size of the quotient field or if division by 0 is attempted, a decimal divide exception results; the divisor and dividend remain unchanged in their storage locations.
- A decimal divide exception occurs if the dividend does not have at least one leading 0. The condition for a decimal divide exception can be determined by aligning the leftmost digit of the divisor (operand 2) field with the leftmost-less-one digit of the dividend (operand 1) field and performing a subtraction. If, when so aligned, the divisor is less than or equal to the dividend, a decimal divide exception is indicated.

- The condition code remains unchanged.
- Possible program exceptions:

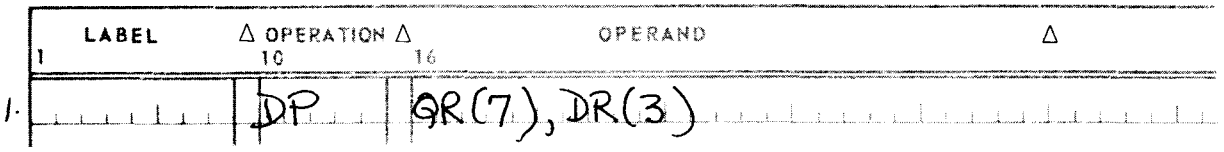
SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Decimal divide	Decimal divide
Indirect address specification	Write protection
Indirect addressing	
Protection	
Specification exception 1. IACW not on full-word boundary 2. Operand 1 is not longer than operand 2.	
Data exception 1. Invalid sign or digit code 2. Incorrect overlap	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Operational Differences:

- 9400/9480 systems
  - Operand 1 and operand 2 fields may not overlap.
  - The maximum divisor (operand 2) length is 31 digits plus sign.
  - Decimal digits greater than 9_{16} are not permitted. The sign portion must contain a sign bit configuration greater than 9_{16}.
- 90/60,70 systems
  - Operand 1 and operand 2 fields may overlap if their least significant bytes coincide.
  - The maximum divisor (operand 2) length is 15 digits plus sign.
  - If a sign is not encountered within the first 16 bytes of data in operand 1, a program exception occurs.
  - All signs and digits are checked for validity.

Example:



1. The contents of the 7-byte area QR are divided by the contents of the 3-byte area DR. The quotient and remainder are placed in QR.

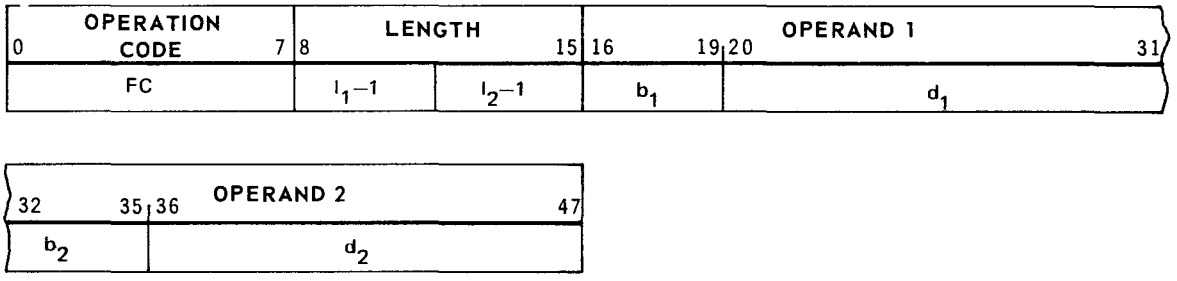
5.5. MP (MULTIPLY-DECIMAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MP	$d_1(l_1,b_1),d_2(l_2,b_2)$	FC	SS	Six Bytes

Function:

The contents of operand 1 (the multiplicand), specified by $d_1(l_1,b_1)$, are multiplied by the contents of operand 2 (the multiplier), specified by $d_2(l_2,b_2)$, and the product is stored in the operand 1 location.

Object Instruction Format:



Operational Considerations:

- The sign of the product is determined algebraically.
- The size of the multiplier (operand 2) cannot be more than 15 digits and sign.
- The length of operand 1 specified by the l_1 field in the instruction is ignored. The length of operand is determined by scanning operand 1 starting with the most significant digit until a sign code is found.
- The number of digits in the product is equal to the number of digits in the operands; therefore, the multiplicand (operand 1) must have a field of most significant 0 digits equal to the number of digits in operand 2. The maximum product size is 31 digits plus sign. At least one most significant digit of the product field is 0.

- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Protection	Write protection
Addressing	Addressing
Indirect address specification	
Indirect addressing	
Specification 1. Multiplier exceeds 15 digits 2. Operand 1 is not longer than operand 2. 3. IACW is not on full-word boundary.	
Data exception 1. Invalid sign or digit code 2. Incorrect overlap 3. Operand 1 does not have sufficient high-order zero digits.	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Operational Differences:

- 9400/9480 systems
  - Operand 1 and operand 2 may not overlap.
- 90/60,70 systems
  - Operand 1 and operand 2 may overlap if their least significant bytes coincide.
  - All signs and digits are checked for validity.
  - If the sign if not encountered within the first 16 bytes of data in operand 1, a program exception occurs.

Example:

```
      LABEL     Δ OPERATION Δ          OPERAND                    Δ
1               10          16
/.              MP          HOUR(7),RATE(3)
```

1. The contents of the operand HOUR are multipled by the contents of the operand RATE. The contents of the operands may be represented as follows:

HOUR before execution	0 0 0 0 0 0 9 9 9 9 9 9 +
RATE before and after execution	9 9 9 9 9 +
HOUR after execution	0 9 9 9 9 8 9 9 0 0 0 0 1 +

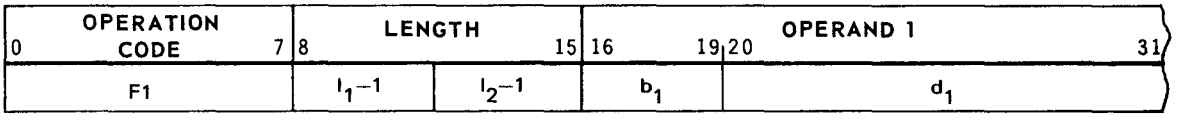
5.6. MVO (MOVE-WITH-OFFSET)

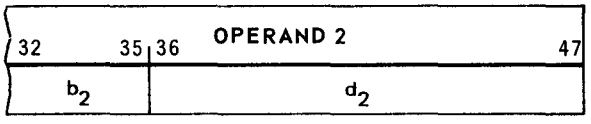
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MVO	$d_1(l_1,b_1),d_2(l_2,b_2)$	F1	SS	Six Bytes

Function:

The contents of operand 2, specified by $d_2(l_2,b_2)$, are shifted to the left four bits and stored in the operand 1 location, specified by $d_1(l_1,b_1)$.

Object Instruction Format:





Operational Considerations:

- This instruction proceeds from right to left.
- The operands are not checked for valid codes.
- Overlapping fields may occur. Unless the operands overlap, operand 2 and the least significant four bits of operand 1 remain unchanged.
- If operand 2 is exhausted before operand 1, the remaining operand 1 field is zero filled. If the result exceeds the capacity of the operand 1 field, the remaining digits of operand 2 are ignored. This operation, in effect, prefixes the least significant digit or sign of operand 1 with the digits of operand 2.
- The condition code remains unchanged.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Write protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):

  – operand 1: RD, ID

  – operand 2: RO, IO

Examples:

	LABEL (1)	△ OPERATION (10) △	OPERAND (16) △
1.		MVO	DEST(5),ORIG(3)
2.		MVO	VAL(4),VAL(4)
3.		MVO	MAR+1(4),MAR(4)

1. The contents of the 3-byte area in main storage specified by ORIG are moved with offset to the main storage location specified by DEST. The contents of the operand may be represented as follows:

DEST before execution	C B A F E D C B A +
ORIG before and after execution	2 4 6 8 9 1
DEST after execution	0 0 0 2 4 6 8 9 1 +

2. The 4-byte area in main storage specified by VAL is moved with offset to the main storage specified by VAL. The contents of the operands may be represented as follows:

VAL before execution	2 3 5 6 8 9 0 +
VAL after execution	3 5 6 8 9 0 + +

   The digit 2, which is in VAL before execution, is lost.

3. The 4-byte area in main storage specified by MAR is moved with offset to the main storage specified by MAR 1; this effectively results in a shift to the right of one byte and an offset to the left of four bits. The contents of the operands may be represented as follows:

MAR before execution	9 8 7 6 5 4 3 2 1 +
MAR after execution	0 9 8 7 6 5 4 3 2 +

Operand 1 encompasses four bytes on the right end of a 5-byte MAR. Operand 2 encompasses four bytes on the left end of a 5-byte MAR. The move-with-offset instruction moves operand 2 to operand 1, offsetting the data four bits. The move is in effect a 4-bit shift to the right.

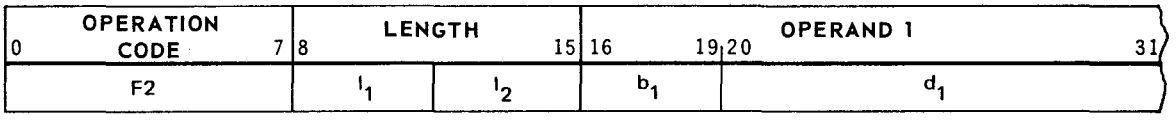
5.7. PACK (PACK)

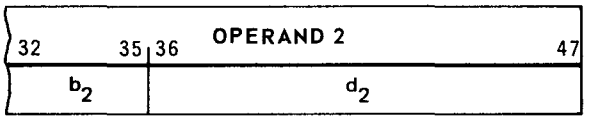
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
PACK	$d_1(l_1,b_1),d_2(l_2,b_2)$	F2	SS	Six Bytes

Function:

The contents of operand 2, specified by $d_2(l_2,b_2)$, are converted from unpacked (zoned) format to packed format and placed in the operand 1 location, specified by $d_1(l_1,b_1)$. This instruction prepares the operand for decimal arithmetic operations.

Object Instruction Format:





Operational Considerations:

- This instruction transfers the decimal portion of an unpacked byte in operand 2 to a byte in operand 1, packing two decimal digits (four bits each) into a single byte. The sign of the operand 2 field (four most significant bits of the least significant byte) is transferred into the four least significant bits of the least significant byte of operand 1. The result is automatically padded with a leading 0, if necessary, to cause the number to begin on a byte boundary. The operation is performed in the manner illustrated:

  Unpacked zone-digit format operand 2 Z 4 Z 3 Z 2 sign 7

  Packed digit-digit format operand 1 0 4 3 2 7 sign

  where:

  Z represents the zone portion.

- If operand 2 does not fill operand 1, the remaining operand 1 field is zero filled.
- If the result exceeds the capacity of the operand 1 field, the remaining operand 2 digits are ignored.
- The operands are not checked for valid codes.

- Overlapping fields may occur; each resultant byte is processed after each operand byte. The instruction proceeds from right to left.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Write protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Examples:

```
      LABEL    Δ OPERATION Δ        OPERAND                    Δ
      1        10          16
1.            PACK        ARIT(6),NMBR(10)
2.            PACK        ARIT(6),NMBR(10)
```

1. The contents of operand 2, specified by NMBR, are 10 decimal digits and sign. The sign and digits are transferred to the operand 1 location, ARIT, according to the following representation:

   NMBR Z 2 Z 4 Z 6 Z 8 Z 1 Z 3 Z 5 Z 7 Z 9 sign 6

   ARIT 0 2 4 6 8 1 3 5 7 9 6 sign

2. Same as example 1, except operand 2 contains 11 decimal digits and sign. The transfer is made according to the following representation:

   NMBR Z 9 Z 8 Z 7 Z 6 Z 5 Z 4 Z 3 Z 2 Z 1 Z 0 sign 2

   ARIT 9 8 7 6 5 4 3 2 1 0 2 sign

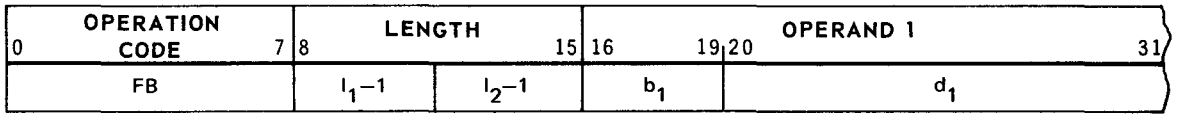
5.8. SP (SUBTRACT-DECIMAL)

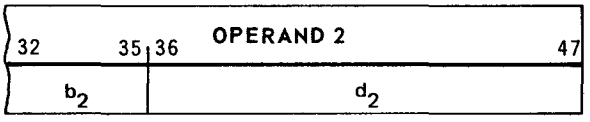
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SP	$d_1(l_1,b_1),d_2(l_2,b_2)$	FB	SS	Six Bytes

Function:

The contents of operand 2, specified by $d_2(l_2,b_2)$, are subtracted from the contents of operand 1, specified by $d_1(l_1,b_1)$, and the result is placed in the operand 1 location.

Object Instruction Format:





Operational Considerations:

- Subtraction is accomplished by reversing the sign of operand 2 and performing a decimal add. The contents and sizes of operand 2 are not affected by this operation.
- The sign of the result is determined algebraically.
- A 0 result has a positive sign when the operation is completed without overflow.
- When most significant digits are lost because of overflow, the partial result has the sign which the correct result would have had.
- If operand 2 is shorter than operand 1, operand 2 is extended with 0 digits.
- An overflow condition results if the capacity of the operand 1 field is exceeded by the result or if the carry out of the most significant digit position of the result field is lost.
- Operand 1 and operand 2 may overlap if their least significant bytes coincide.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - to 3 (11_2) if overflow occurs.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Decimal overflow	Decimal overflow
Indirect address specification	Write protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	
Data exception 1. Invalid sign or digit code 2. Incorrect overlap	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Operational Differences:

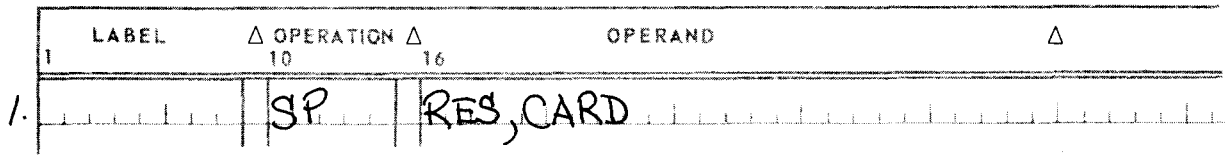
- 9400/9480 systems

  If operand 2 is longer than operand 1, the excess high order digits of operand 2 are ignored.

- 90/60,70 systems

  All signs and digits are checked for validity.

Example:



1. The sign of operand 2, specified by CARD, is reversed and the result is added to the contents of operand 1, specified by RES.

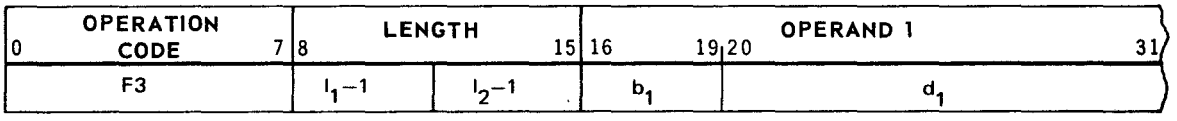
5.9. UNPK (UNPACK)

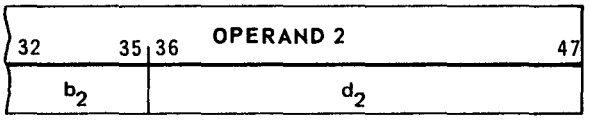
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
UNPK	$d_1(l_1,b_1),d_2(l_2,b_2)$	F3	SS	Six Bytes

Function:

The contents of operand 2, specified by $d_2(l_2,b_2)$, are converted from packed format to unpacked (zoned) format and placed in the operand 1 location, specified by $d_1(l_1,b_1)$.

Object Instruction Format:





Operational Considerations:

- The decimal data of operand 2 is transferred sequentially, right to left, to the numeric portion of each operand 1 byte. The zone supplied depends on the state of the A mode of the current program status word (PSW) (1111_2 for EBCDIC, 0011_2 for ASCII).
- The sign in the packed operand, located in the four least significant bits of the least significant byte of operand 2, is transferred to the four most significant bits of the least significant byte of operand 1, as shown in the illustration:

  Packed digit-digit format operand 2 0 4 3 2 7 sign

Unpacked zone-digit format operand 1 Z 0 Z 4 Z 3 Z 2 sign 7

where:

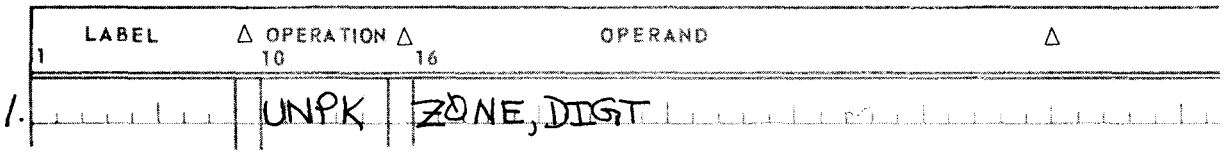
Z represents the zone portion

- If operand 2 does not completely fill operand 1, the remaining operand 1 bytes are set to 0 with the appropriate zone.
- If the result exceeds the capacity of the operand 1 field, the remaining operand 2 digits are ignored.
- The operands are not checked for valid codes.
- Overlapping fields may occur; each result byte is processed after processing each operand 2 byte. Except for the least significant operand 2 byte, containing the sign, each operand 2 byte produces two result bytes. If the operand fields are to be overlapped, the least significant position of operand 1 must be to the right of the least significant position of operand 2 by the number of bytes in operand 2 minus 2. If one or two bytes are to be unpacked, the least significant positions of the operands may coincide.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Write protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. Assume the contents of operand 2, specified by DIGT, to be 2468901. To unpack the 4-byte operand 2 field, it is necessary to have a 7-byte operand 1 field. The length of operand 1 equals the length of operand 2 in bytes times 2 minus 1.

   The contents of the operands appear as follows:

DIGIT	2 4 6 8 9 0 1 sign
ZONE	Z 2 Z 4 Z 6 Z 8 Z 9 Z 0 sign 1

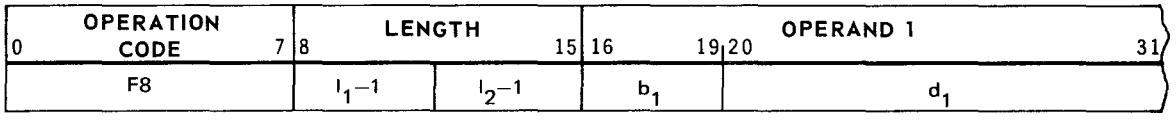
5.10. ZAP (ZERO-AND-ADD)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ZAP	$d_1(l_1,b_1),d_2(l_2,b_2)$	F8	SS	Six Bytes

Function:

Operand 1, specified by $d_1(l_1,b_1)$; is cleared to 0, the contents of operand 2, specified by $d_2(l_2,b_2)$, are added to the contents of operand 1.

Object Instruction Format:



Operational Considerations:

- The zero-and-add instruction is equivalent to the add-decimal instruction with 0 as the contents of operand 1.

- A 0 result has a positive sign, except when digits are lost due to overflow. In this case, a 0 result has the sign of operand 2.
- If operand 2 does not fill the operand 1 field, operand 2 is extended with 0's.
- Operand 1 and operand 2 may overlap if their least significant bytes coincide, or if the least significant byte of operand 1 is to the right of the rightmost byte of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Write protection
Indirect addressing	
Protection	
Decimal overflow	
Specification (IACW not on full-word boundary)	
Data exception 1. Invalid sign or digit code 2. Incorrect overlap	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Operational Differences:

- 9400/9480 systems

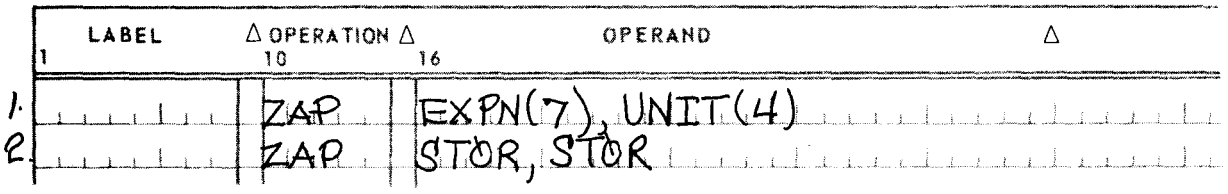
  If operand 2 is longer than operand 1, the most significant digits of operand 2 are ignored.

- 90/60,70 systems

  If operand 2 is longer than operand 1 and significant digits are lost, a decimal overflow condition will occur.

  All signs and digits of operand 2 are checked for validity.

Examples:



1. The contents of the 4-byte operand labeled UNIT are added to the area in main storage specified by the label EXPN after EXPN has been forced to 0. The contents of the operands may be represented as follows:

EXPN before execution	1 0 C C F F 9 F F 2 F 3 A 7
UNIT before and after execution	3 9 1 7 2 3 8 A
EXPN after execution	0 0 0 0 0 0 3 9 1 7 2 3 8 C

2. The contents of main storage labeled STOR are not changed unless the sign is modified in the addition. The condition code is set.

6. Floating–Point Instructions–90/60,70

6.1. GENERAL

The floating-point instruction set is provided on the SPERRY UNIVAC 90/60,70 Systems only. This instruction set is added to the 90/60,70 systems instruction repertoire as part of the Floating-Point Control Feature, F1334–00. An operation exception results if a floating-point instruction is issued to a processor in which the floating-point control feature has not been installed.

The floating-point instruction set provides for loading, adding, subtracting, comparing, multiplying, dividing, storing, and sign control of short or long format floating-point operands. See 1.3.2 for information concerning the manner in which floating-point numbers are represented and their sign codes established. Four double-word floating-point registers are provided to accommodate storing and loading of results and operands. These registers are numbered 0, 2, 4, and 6. The specification of any other register number results in a specification exception. For long format operands, the entire double-word register is involved in the operation. For short format operands, excluding the product in the short format multiply instruction, only the most significant word of the double-word register is involved in the operation. The least significant word remains unchanged.

The floating-point instructions are available in RR and RX formats. Therefore, at least one of the operands is contained in one of the floating-point registers. The other operand is located in the same or another register or in main storage. Each main storage address may be specified as relative or absolute and direct or indirect under control of the applicable relocation register flags.

To increase the precision of certain computations, an additional least significant digit, the guard digit, is carried within the hardware in the intermediate result of the following operations: add-normalized, subtract-normalized, add-unnormalized, subtract-unnormalized, compare, halve, and multiply. In the execution of add-normalized, subtract-normalized, add-unnormalized, subtract-unnormalized, and compare instructions, when a right shift of the fraction is required to equalize two exponents, the last hexadecimal digit to be shifted out of the least significant digit position of the fraction is saved by the processor hardware as the guard digit. The shifted fraction, including the guard digit, is used in computing the intermediate result. In the halve instruction, the least significant bit position of the fraction is saved as the most significant bit position of the guard digit. In the long format multiply instruction, the guard digit is carried as the fifteenth digit of the fraction of the intermediate product. If the intermediate result is subsequently normalized, the guard digit is shifted left to become part of the normalized fraction.

This section describes the operation of each floating-point instruction. The instructions are arranged in alphabetical order according to mnemonic operation code. Each description includes a list of the possible program exceptions and condition codes which may result. The relocation and indirection flags pertinent to each operand are also listed. See Table 3–1 for an explanation of the abbreviations used in describing instruction formats.

6.2. AD (ADD-NORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AD	$r_1,d_2(x_2,b_2)$	6A	RX	Four Bytes

Function:

The double-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically added to the double-word contents of operand 1, specified by r_1. The normalized sum is placed in operand 1.

Operational Considerations:

- Floating-point addition

  Floating-point addition consists of exponent equalization and fraction addition. If the exponents are equal, the fractions are added to form an intermediate sum. If the exponents are unequal, the smaller exponent is subtracted from the larger. The difference indicates the number of hexadecimal digit shifts to the right to be performed on the fraction having the smaller exponent. Each hexadecimal digit shift to the right causes the exponent to be increased by 1. After equalization, the fractions are added to form an intermediate sum.

  A carry out of the most significant hexadecimal digit position of the intermediate sum causes the intermediate sum to be shifted right one digit position and the exponent to be increased by 1. If an exponent overflow condition occurs, the resultant floating-point number consists of a normalized and correct fraction, a correct sign, and an exponent which is 128 less than the correct value.

- Normalization

  The intermediate sum is composed of 14 hexadecimal digits, a guard digit (6.1), and a possible carry. If any most significant digits of the intermediate sum are 0, the fraction including the guard digit is shifted left to form a normalized fraction. Vacated least significant digit positions are zero filled and the exponent is reduced by the number of shifts. If normalization is unnecessary, the guard digit is lost.

- Exponent underflow

  If normalization causes the exponent to become less than 0, an exponent underflow condition results. If the exponent underflow mask bit of the current program status word (PSW) is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 more than the correct value. If the exponent underflow mask of the current PSW is 0, the result is a true 0. The exponent underflow condition causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- Zero result

  If the intermediate sum, including the guard digit, is 0, a significance exception exists. If the significance mask bit of the current PSW is 1, the result is not normalized and the exponent remains unchanged. If the significance mask bit of the current PSW is 0 and the intermediate sum is 0, the result is made a true 0. Exponent underflow cannot occur for a 0 fraction. The significance exception causes a program interrupt if the significance mask bit and the program exception mask bit of the current PSW are 1.

- The sign of an arithmetic result is determined algebraically. The sign of a result with a 0 fraction is always positive.
- The condition code is set as follows:
  - to 0 (00_2) if the result fraction is 0;
  - to 1 (01_2) if the result fraction is less than 0;
  - to 2 (10_2) if the result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 not on double-word boundary or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:

1. The double-word contents of floating-point register 4 and the main storage location labeled DEND are added. The result is placed in register 4.

6.3. ADR (ADD-NORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ADR	r_1,r_2	2A	RR	Two Bytes

Function:

The double-word contents of operand 2, specified by r_2, are algebraically added to the double-word contents of operand 1, specified by r_1. The normalized sum is placed in operand 1.

Operational Considerations:

- Floating-point addition

  Floating-point addition consists of exponent equalization and fraction addition. If the exponents are equal, the fractions are added to form an intermediate sum. If the exponents are unequal, the smaller exponent is subtracted from the larger. The difference indicates the number of hexadecimal digit shifts to the right to be performed on the fraction having the smaller exponent. Each hexadecimal digit shift to the right causes the exponent to be increased by 1. After equalization, the fractions are added to form an intermediate sum.

  A carry out of the most significant hexadecimal digit position of the intermediate sum causes the intermediate sum to be shifted right one digit position and the exponent to be increased by 1. If an exponent overflow condition occurs, the resultant floating-point number consists of a normalized and correct fraction, a correct sign, and an exponent which is 128 less than the correct value.

- Normalization

  The intermediate sum is composed of 14 hexadecimal digits, a guard digit (6.1), and a possible carry. If any most significant digits of the intermediate sum are 0, the fraction including the guard digit is shifted left to form a normalized fraction. Vacated least significant digit positions are zero filled and the exponent is reduced by the number of shifts. If normalization is unnecessary, the guard digit is lost.

- Exponent underflow

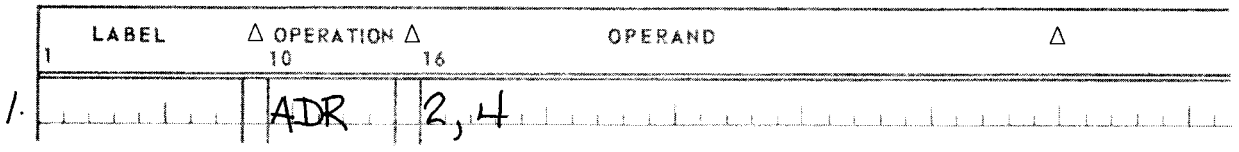
  If normalization causes the exponent to become less than 0, an exponent underflow condition results. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 more than the correct value. If the exponent underflow mask of the current PSW is 0, the result is a true 0. The exponent underflow condition causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- Zero result

  If the intermediate sum, including the guard digit, is 0, a significance exception exists. If the significance mask bit of the current PSW is 1, the result is not normalized and the exponent remains unchanged. If the significance mask bit of the current PSW is 0 and the intermediate sum is 0, the result is made a true 0. Exponent underflow cannot occur for a 0 fraction. The significance exception causes a program interrupt if the significance mask bit and the program exception mask bit of the current PSW are 1.

- The sign of an arithmetic result is determined algebraically. The sign of a result with a 0 fraction is always positive.
- The condition code is set as follows:
  - to 0 (00_2) if the result fraction is 0;
  - to 1 (01_2) if the result fraction is less than 0;
  - to 2 (10_2) if the result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point registers 2 and 4 are added and the result is placed in register 2.

6.4. AE (ADD-NORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AE	$r_1,d_2(x_2,b_2)$	7A	RX	Four Bytes

Function:

The full-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically added to the full-word contents of operand 1, specified by r_1. The normalized sum is placed in operand 1.

Operational Considerations:

- Floating-point addition

  Floating-point addition consists of exponent equalization and fraction addition. If the exponents are equal, the fractions are added to form an intermediate sum. If the exponents are unequal, the smaller exponent is subtracted from the larger. The difference indicates the number of hexadecimal digit shifts to the right to be performed on the fraction having the smaller exponent. Each hexadecimal digit shift to the right causes the exponent to be increased by 1. After equalization, the fractions are added to form an intermediate sum.

  A carry out of the most significant hexadecimal digit position of the intermediate sum causes the intermediate sum to be shifted right one digit position and the exponent to be increased by 1. If an exponent overflow condition occurs, the resultant floating-point number consists of a normalized and correct fraction, a correct sign, and an exponent which is 128 less than the correct value.

- Normalization

  The intermediate sum is composed of six hexadecimal digits, a guard digit, and a possible carry. If any most significant digits of the intermediate sum are 0, the fraction including the guard digit is shifted left to form a normalized fraction. Vacated least significant digit positions are zero filled and the exponent is reduced by the number of shifts. If normalization is unnecessary, the guard digit is lost.

- Exponent underflow

  If normalization causes the exponent to become less than 0, an exponent underflow condition results. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 more than the correct value. If the exponent underflow mask of the current PSW is 0, the result is a true 0. The exponent underflow condition causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- Zero result

  If the intermediate sum, including the guard digit, is 0, a significance exception exists. If the significance mask bit of the current PSW is 1, the result is not normalized and the exponent remains unchanged. If the significance mask bit of the current PSW is 0, and the intermediate sum is 0, the result is made a true 0. Exponent underflow cannot occur for a 0 fraction. The significance exception causes a program interrupt if the significance mask bit and the program exception mask bit of the current PSW are 1.

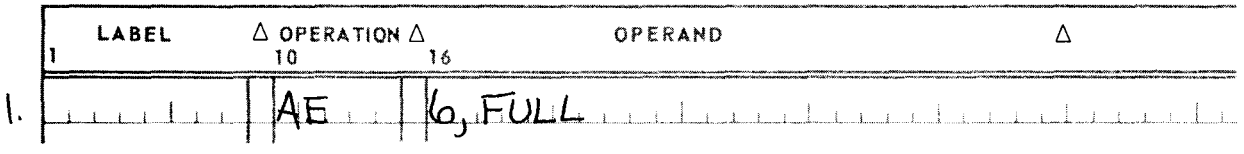
- The sign of an arithmetic result is determined algebraically. The sign of a result with a 0 fraction is always positive.

- The condition code is set as follows:

  - to 0 (00_2) if the result fraction is 0;
  - to 1 (01_2) if the result fraction is less than 0;
  - to 2 (10_2) if the result fraction is greater than 0; or
  - code 3 is not used.

- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 not on full-word boundary or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of floating-point register 6 and the main storage location FULL are added. The result is placed in register 6.

6.5. AER (ADD-NORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AER	r_1,r_2	3A	RR	Two Bytes

Function:

The full-word contents of operand 2, specified by r_2, are algebraically added to the full-word contents of operand 1, specified by r_1. The normalized sum is placed in operand 1.

Operational Considerations:

- Floating-point addition

  Floating-point addition consists of exponent equalization and fraction addition. If the exponents are equal, the fractions are added to form an intermediate sum. If the exponents are unequal, the smaller exponent is subtracted from the larger. The difference indicates the number of hexadecimal digit shifts to the right to be performed on the fraction having the smaller exponent. Each hexadecimal digit shift to the right causes the exponent to be increased by 1. After equalization, the fractions are added to form an intermediate sum.

  A carry out of the most significant hexadecimal digit position of the intermediate sum causes the intermediate sum to be shifted right one digit position and the exponent to be increased by 1. If an exponent overflow condition occurs, the resultant floating-point number consists of a normalized and correct fraction, a correct sign, and an exponent which is 128 less than the correct value.

- Normalization

  The intermediate sum is composed of six hexadecimal digits, a guard digit, and a possible carry. If any most significant digits of the intermediate sum are 0, the fraction including the guard digit is shifted left to form a normalized fraction. Vacated least significant digit positions are zero filled and the exponent is reduced by the number of shifts. If normalization is unnecessary, the guard digit is lost.

- Exponent underflow

  If normalization causes the exponent to become less than 0, an exponent underflow condition results. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 more than the correct value. If the exponent underflow mask of the current PSW is 0, the result is a true 0. The exponent underflow condition causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- Zero result

  If the intermediate sum, including the guard digit, is 0, a significance exception exists. If the significance mask bit of the current PSW is 1, the result is not normalized and the exponent remains unchanged. If the significance mask bit of the current PSW is 0 and the intermediate sum is 0, the result is made a true 0. Exponent underflow cannot occur for a 0 fraction. The significance exception causes a program interrupt if the significance mask bit and the program exception mask bit of the current PSW are 1.

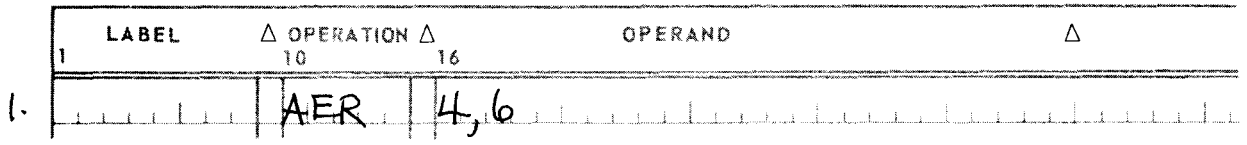
- The sign of an arithmetic result is determined algebraically. The sign of a result with a 0 fraction is always positive.

- The condition code is set as follows:

  - to 0 (00_2) if the result fraction is 0;
  - to 1 (01_2) if the result fraction is less than 0;
  - to 2 (10_2) if the result fraction is greater than 0; or
  - code 3 is not used.

- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point registers 4 and 6 are added and the result is placed in register 4.

6.6. AU (ADD-UNNORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AU	$r_1,d_2(x_2,b_2)$	7E	RX	Four Bytes

Function:

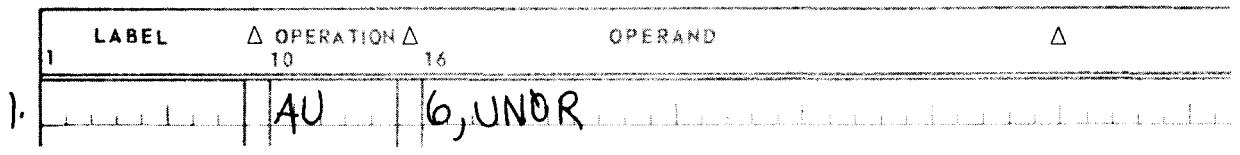
The full-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically added to the full-word contents of operand 1, specified as r_1. The sum is placed in operand 1.

Operational Considerations:

- The execution of the AU instruction is identical to the AE instruction, except that the sum is not normalized before being placed in operand 1.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.

- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of floating-point register 6 and main storage location UNOR are added and the result is placed in register 6.

6.7. AUR (ADD-UNNORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AUR	r_1,r_2	3E	RR	Two Bytes

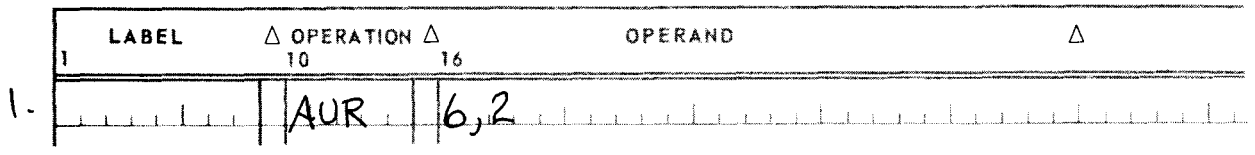
Function:

The full-word contents of operand 2, specified by r_2, are algebraically added to the full-word contents of operand 1, specified as r_1. The sum is placed in operand 1.

Operational Considerations:

- The execution of the AUR instruction is identical to the AER instruction, except that the sum is not normalized before being placed in operand 1.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - exponent overflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point registers 6 and 2 are added and the result is placed in register 6.

6.8. AW (ADD-UNNORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AW	$r_1,d_2(x_2,b_2)$	6E	RX	Four Bytes

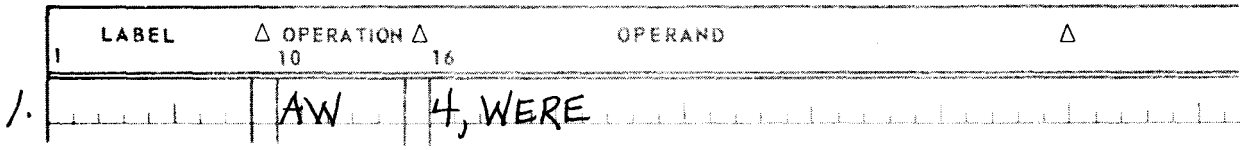
Function:

The double-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically added to the double-word contents of operand 1, specified by r_1. The sum is placed in operand 1.

Operational Considerations:

- The execution of the AW instruction is identical to the AD instruction, except that the sum is not normalized before being placed in operand 1.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 not on double-word boundary or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The double-word contents of floating-point register 4 and the main storage location WERE are added and the result is placed in register 4.

6.9. AWR (ADD-UNNORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AWR	r_1,r_2	2E	RR	Two Bytes

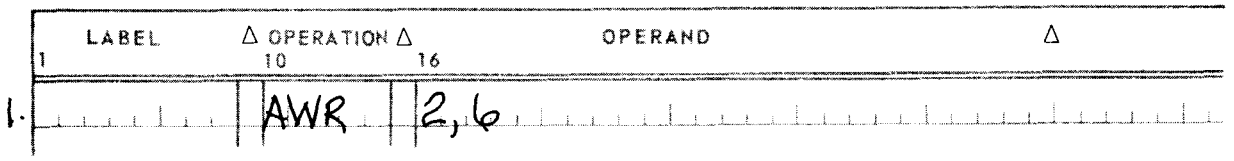
Function:

The double-word contents of operand 2, specified by r_2, are algebraically added to the double-word contents of operand 1, specified as r_1. The sum is placed in operand 1.

Operational Considerations:

- The execution of the AWR instruction is identical to the ADR instruction, except that the sum is not normalized before being placed in operand 1.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - exponent overflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point registers 2 and 6 are added and the result is placed in register 2.

6.10. CD (COMPARE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CD	$r_1,d_2(x_2,b_2)$	69	RX	Four Bytes

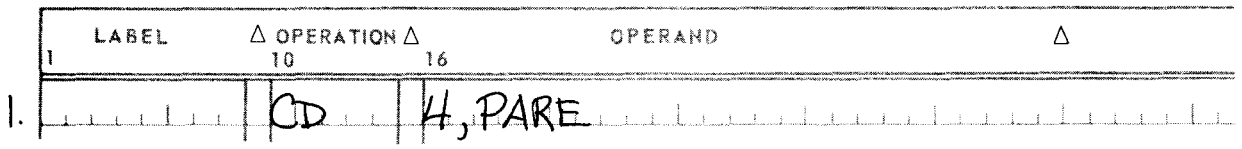
Function:

The double-word contents of operand 1, specified by r_1, are algebraically compared with the double-word contents of operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- Comparison is accomplished by the rules for normalized floating-point subtraction. The operands are equal when the intermediate sum, including the guard digit, is 0.
- Operands with 0 fractions compare as equal even when their signs or exponents are different.
- The condition code is set as follows:
  - to 0 (00_2) when operand 1 equals operand 2;
  - to 1 (01_2) when operand 1 is less than operand 2;
  - to 2 (10_2) when operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 not on double-word boundary or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The double-word contents of floating-point register 4 and main storage location PARE are compared and the condition code is set.

6.11. CDR (COMPARE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CDR	r_1,r_2	29	RR	Two Bytes

Function:

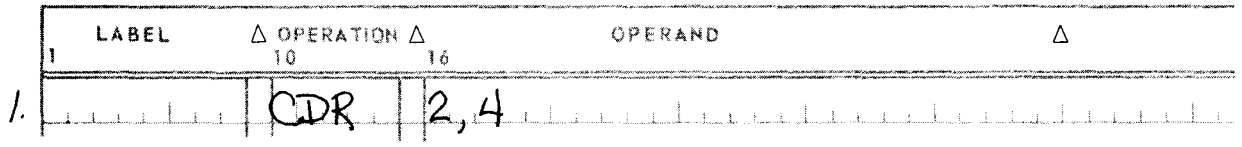
The double-word contents of operand 1, specified by r_1, are algebraically compared with the double-word contents of operand 2, specified by r_2.

Operational Considerations:

- Comparison is accomplished by the rules for normalized floating-point subtraction. The operands are equal when the intermediate sum, including the guard digit, is 0.
- Operands with 0 fractions compare as equal even when their signs or exponents are different.
- The condition code is set as follows:
  - to 0 (00_2) when operand 1 equals operand 2;
  - to 1 (01_2) when operand 1 is less than operand 2;
  - to 2 (10_2) when operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

  operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point registers 2 and 4 are compared and the condition code is set.

6.12. CE (COMPARE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CE	$r_1,d_2(x_2,b_2)$	79	RX	Four Bytes

Function:

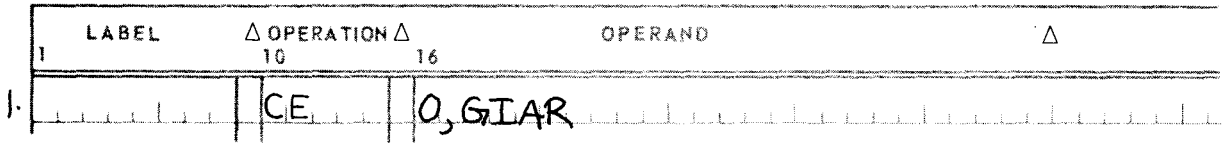
The full-word contents of operand 1, specified by r_1, are algebraically compared with the full-word contents of operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- Comparison is accomplished by the rules for normalized fixed-point subtraction. The operands are equal when the intermediate sum, including the guard digit, is 0.
- Operands with 0 fractions compare as equal even when their signs or exponents are different.
- The condition code is set as follows:
  - to 0 (00_2) when operand 1 equals operand 2;
  - to 1 (01_2) when operand 1 is less than operand 2;
  - to 2 (10_2) when operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of floating-point register 0 and main storage location GIAR are compared and the condition code is set.

6.13. CER (COMPARE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CER	r_1,r_2	39	RR	Two Bytes

Function:

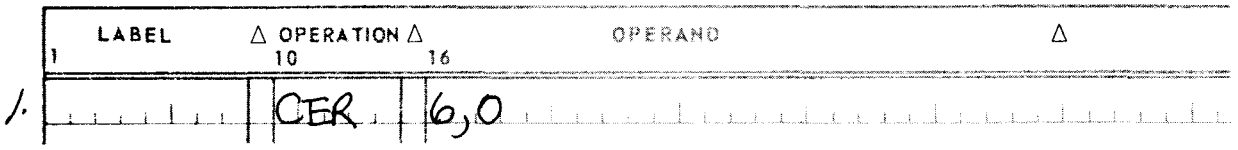
The full-word contents of operand 1, specified by r_1, are algebraically compared with the full-word contents of operand 2, specified by r_2.

Operational Considerations:

- Comparison is accomplished by the rules for normalized floating-point subtraction. The operands are equal when the intermediate sum, including the guard digit, is 0.
- Operands with 0 fractions compare as equal even when their signs or exponents are different.
- The condition code is set as follows:
  - to 0 (00_2) when operand 1 equals operand 2;
  - to 1 (01_2) when operand 1 is less than operand 2;
  - to 2 (10_2) when operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)

- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point registers 0 and 6 are compared and the condition code is set.

6.14. DD (DIVIDE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DD	$r_1,d_2(x_2,b_2)$	6D	RX	Four Bytes

Function:

The double-word contents of operand 1 (dividend), specified by r_1, are divided by the double-word contents of operand 2 (divisor), specified by $d_2(x_2,b_2)$. The normalized quotient is placed in operand 1. The remainder is not preserved.

Operational Considerations:

- Floating-point division consists of exponent subtraction and fraction division. The intermediate quotient exponent is obtained by subtracting the exponents of the two operands and increasing the difference by 64.

- Both operands are normalized before division (6.2). Consequently, the intermediate quotient is correctly normalized or a right shift of one digit position may be required. The exponent of the intermediate result is increased by 1 if the shift is necessary. All operand 1 fraction digits are used in forming the quotient even if the normalized operand 1 fraction is larger than the normalized operand 2 fraction.

- If the final quotient exponent exceeds 127, an exponent overflow exception results. The quotient consists of the correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value.

- If the final quotient exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the quotient has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. Underflow does not apply to the intermediate result or the operands during normalization. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- Attempted division by a divisor with a 0 fraction leaves the dividend unchanged and a program exception for floating-point divide occurs. When division of a 0 dividend is attempted, the quotient fraction is 0. The quotient sign and exponent are made 0, giving a true 0 result. No program exceptions occur.

- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - floating-point divide exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 not on double-word boundary, or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:

1. The double-word contents of floating-point register 4 are divided by the double-word contents at main storage location STAR. The result is placed in register 4.

6.15. DDR (DIVIDE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DDR	r_1,r_2	2D	RR	Two Bytes

Function:

The double-word contents of operand 1 (dividend), specified by r_1, are divided by the double-word contents of operand 2 (divisor), specified by r_2. The normalized quotient is placed in operand 1. The remainder is not preserved.

Operational Considerations:

- Floating-point division consists of exponent subtraction and fraction division. The intermediate quotient exponent is obtained by subtracting the exponents of the two operands and increasing the difference by 64.

- Both operands are normalized (6.2) before division. Consequently, the intermediate quotient is correctly normalized or a right shift of one digit position may be required. The exponent of the intermediate result is increased by 1 if the shift is necessary. All operand 1 fraction digits are used in forming the quotient even if the normalized operand 1 fraction is larger than the normalized operand 2 fraction.

- If the final quotient exponent exceeds 127, an exponent overflow exception results. The quotient consists of the correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value.

- If the final quotient exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the quotient has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. Underflow does not apply to the intermediate result or the operands during normalization. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

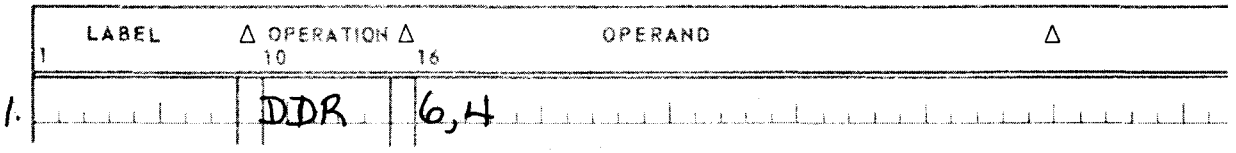
- Attempted division by a divisor with a 0 fraction leaves the dividend unchanged and a program exception for floating-point divide occurs. When division of a 0 dividend is attempted, the quotient fraction is 0. The quotient sign and exponent are made 0, giving a true 0 result. No program exceptions occur.

- The condition code remains unchanged.

- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - floating-point divide exception
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)

- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point register 6 are divided by the contents of floating-point register 4. The result is placed in register 6.

6.16. DE (DIVIDE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DE	$r_1,d_2(x_2,b_2)$	7D	RX	Four Bytes

Function:

The full-word contents of operand 1 (dividend), specified by r_1, are divided by the full-word contents of operand 2 (divisor), specified by $d_2(x_2,b_2)$. The normalized quotient is placed in operand 1. The remainder is not preserved.

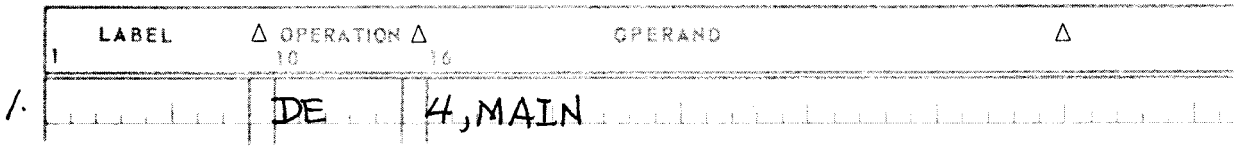
Operational Considerations:

- Floating-point division consists of exponent subtraction and fraction division. The intermediate quotient exponent is obtained by subtracting the exponents of the two operands and increasing the difference by 64.
- Both operands are normalized (6.2) before division. Consequently, the intermediate quotient is correctly normalized or a right shift of one digit position may be required. The exponent of the intermediate result is increased by 1 if the shift is necessary. All operand 1 fraction digits are used in forming the quotient even if the normalized operand 1 fraction is larger than the normalized operand 2 fraction.
- If the final quotient exponent exceeds 127, an exponent overflow exception results. The quotient consists of the correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value.
- If the final quotient exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the quotient has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. Underflow does not apply to the intermediate result or the operands during normalization. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.
- Attempted division by a divisor with a 0 fraction leaves the dividend unchanged and a program exception for floating-point divide occurs. When division of a 0 dividend is attempted, the quotient fraction is 0. The quotient sign and exponent are made 0, giving a true 0 result. No program exceptions occur.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - floating-point divide exception
  - indirect address specification exception

- indirect addressing exception
- operation exception
- protection exception
- specification exception (operand 2 or IACW not on full-word boundary; or operand 1 is not 0, 2, 4, or 6)

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of floating-point register 4 are divided by the full-word contents in main storage location MAIN. The result is placed in register 4.

6.17. DER (DIVIDE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DER	r_1,r_2	3D	RR	Two Bytes

Function:

The full-word contents of operand 1 (dividend), specified by r_1, are divided by the full-word contents of operand 2 (divisor), specified by r_2. The normalized quotient is placed in operand 1. The remainder is not preserved.

Operational Considerations:

- Floating-point division consists of exponent subtraction and fraction division. The intermediate quotient exponent is obtained by subtracting the exponents of the two operands and increasing the difference by 64.
- Both operands are normalized before division. Consequently, the intermediate quotient is correctly normalized or a right shift of one digit position may be required. The exponent of the intermediate result is increased by 1 if the shift is necessary. All operand 1 fraction digits are used in forming the quotient even if the normalized operand 1 fraction is larger than the normalized operand 2 fraction.

- If the final quotient exponent exceeds 127, an exponent overflow exception results. The quotient consists of the correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value.

  An exponent overflow exception causes a program interrupt if the program exception mask bit of the current PSW is 1.

- If the final quotient exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the quotient has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. Underflow does not apply to the intermediate result or the operands during normalization. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- Attempted division by a divisor with a 0 fraction leaves the dividend unchanged and a program exception for floating-point divide occurs. When division of a 0 dividend is attempted, the quotient fraction is 0. The quotient sign and exponent are made 0, giving a true 0 result. No program exceptions occur.

- The condition code remains unchanged.

- Possible program exceptions:

  — exponent overflow exception

  — exponent underflow exception

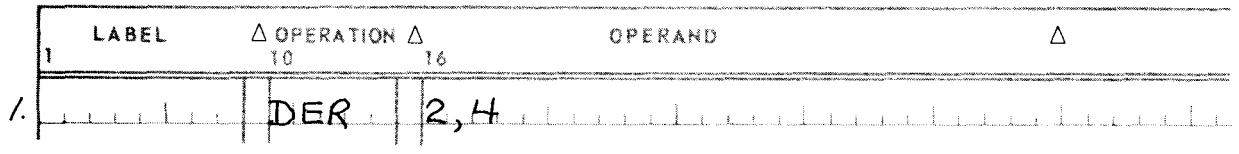
  — floating-point divide exception

  — operation exception

  — specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)

- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point register 2 are divided by the full-word contents of floating-point register 4. The result is placed in register 2.

6.18. HDR (HALVE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
HDR	r_1,r_2	24	RR	Two Bytes

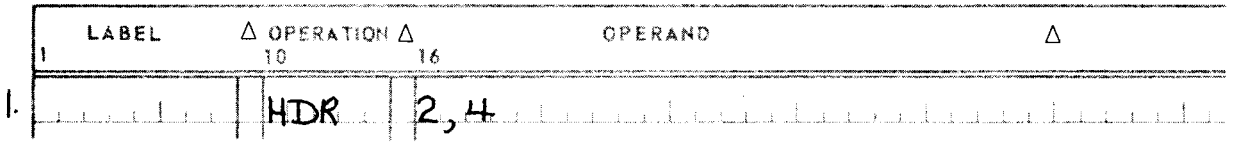
Function:

The double-word contents of operand 2, specified by r_2, are divided by 2. The normalized quotient is placed in operand 1, specified by r_1.

Operational Considerations:

- The fraction of operand 2 is shifted right one bit position, placing the least significant bit of the fraction into the most significant bit position of the guard digit and filling the vacated fraction bit position with 0. The intermediate result is normalized and placed in the operand 1 location.
- When normalization causes the exponent to become less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the exponent of the result is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made true 0. An exponent underflow exception causes an interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.
- When the fraction of operand 2 is 0, the result is made a true 0, normalization is not attempted, and a significance exception does not occur.
- The condition code remains unchanged.
- Possible program exceptions:
  - exponent underflow exception
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point register 4 are divided by 2. The result is placed in floating-point register 2.

6.19. HER (HALVE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
HER	r_1,r_2	34	RR	Two Bytes

Function:

The full-word contents of operand 2, specified by r_2, are divided by 2. The normalized quotient is placed in operand 1, specified by r_1.

Operational Considerations:

- The fraction of operand 2 is shifted right one bit position, placing the least significant bit of the fraction into the most significant bit position of the guard digit and filling the vacated fraction bit position with 0. The intermediate result is normalized and placed in the operand 1 location.

- When normalization causes the exponent to become less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the exponent of the result is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made true 0. An exponent underflow exception causes an interrupt if the exponent underflow mask bit and the program exception mask bit of the current PSW are 1.

- When the fraction of operand 2 is 0, the result is made a true 0, normalization is not attempted, and a significance exception does not occur.

- The condition code remains unchanged.

- Possible program exceptions:

  - exponent underflow exception
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)

- Relocation and indirection flags: none

Example:

	LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
1.		HER	0,6

1. The full-word contents of floating-point register 6 are divided by 2. The result is placed in register 0.

6.20. LCDR (LOAD-COMPLEMENT, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LCDR	r_1,r_2	23	RR	Two Bytes

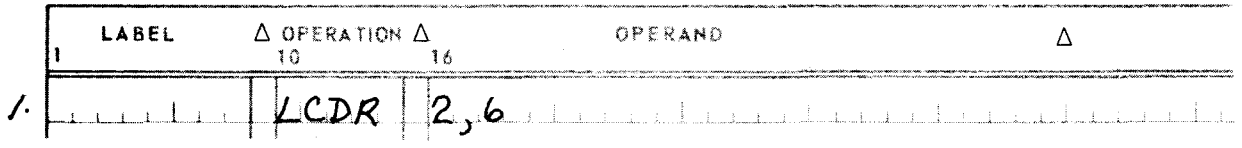
Function:

The sign of double-word operand 2, specified by r_2, is reversed and the result is placed in operand 1, specified by r_1.

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The sign of the double-word contents of floating-point register 6 is reversed and the result is placed in register 2.

6.21. LCER (LOAD-COMPLEMENT, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LCER	r_1,r_2	33	RR	Two Bytes

Function:

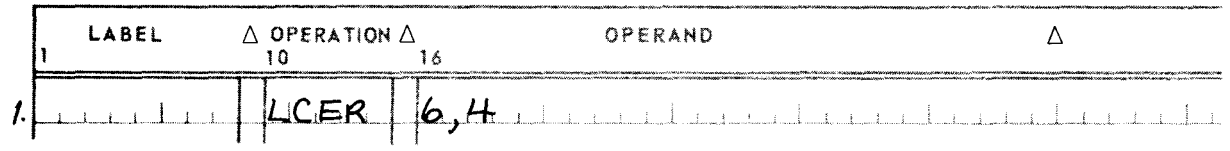
The sign of full-word operand 2, specified by r_2, is reversed and the result is placed in operand 1,specified by r_1.

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 remain unchanged.

- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_3) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The sign of the full-word contents of floating-point register 4 is reversed and the result is placed in register 6.

6.22. LD (LOAD, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LD	$r_1,d_2(x_2,b_2)$	68	RX	Four Bytes

Function:

The contents of double-word operand 2, specified by $d_2(x_2,b_2)$, are placed in operand 1, specified by r_1.

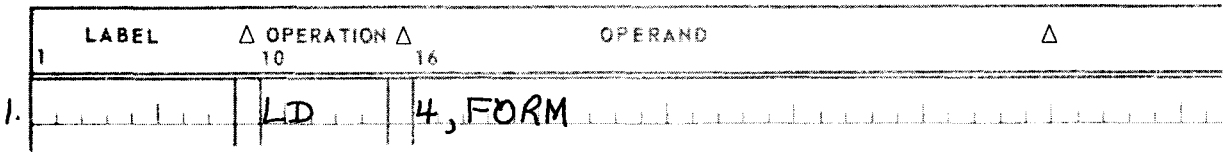
Operational Considerations:

- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception

- indirect addressing exception
- operation exception
- protection exception
- specification exception (operand 2 not on double-word boundary or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The double-word contents of main storage location FORM are placed in floating-point register 4.

6.23. LDR (LOAD,LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LDR	r_1,r_2	28	RR	Two Bytes

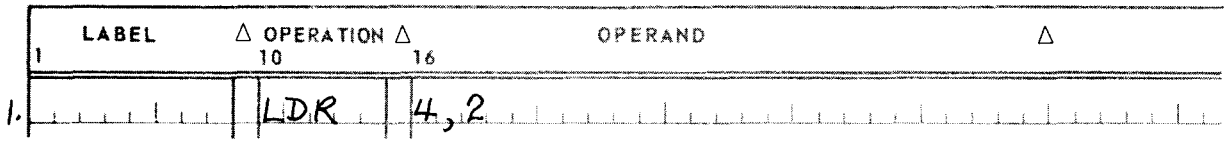
Function:

The contents of double-word operand 2, specified by r_2, are placed in operand 1, specified by r_1.

Operational Considerations:

- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point register 2 are placed in floating-point register 4.

6.24. LE (LOAD, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LE	$r_1,d_2(x_2,b_2)$	78	RX	Four Bytes

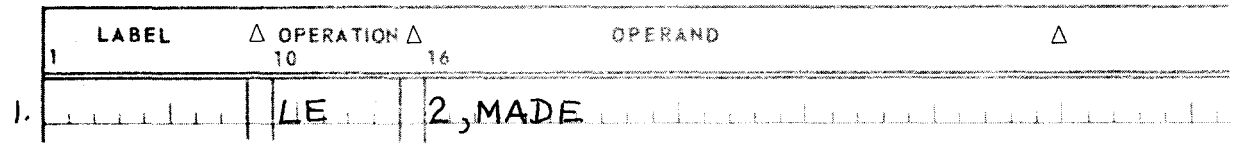
Function:

The contents of full-word operand 2, specified by $d_2(x_2,b_2)$, are placed in operand 1, specified by r_1.

Operational Considerations:

- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of main storage location MADE are placed in floating-point register 2.

6.25. LER (LOAD, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LER	r_1,r_2	38	RR	Two Bytes

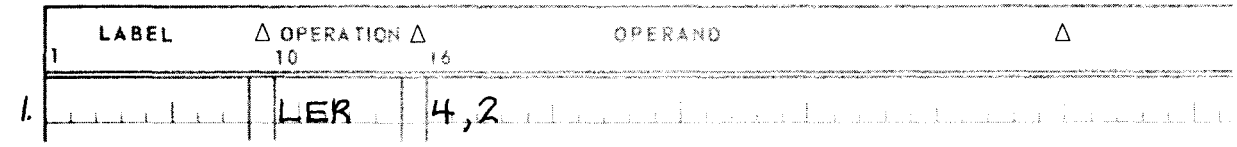
Function:

The contents of full-word operand 2, specified by r_2, are placed in operand 1, specified by r_1.

Operational Considerations:

- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point register 2 are placed in floating-point register 4.

6.26. LNDR (LOAD-NEGATIVE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LNDR	r_1,r_2	21	RR	Two Bytes

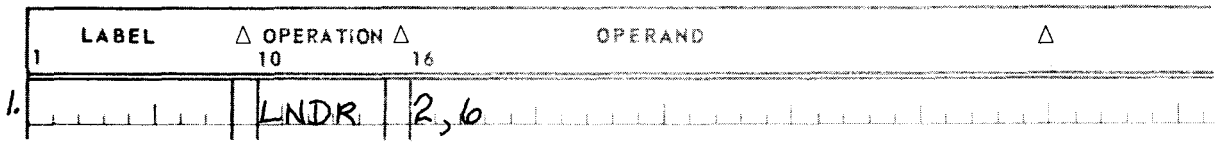
Function:

The sign of double-word operand 2, specified by r_2, is made negative and the result is placed in operand 1, specified by r_1.

Operational Considerations:

- Operand 2 is made negative even if the fraction is 0.
- The exponent and fraction are not changed.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The sign of the double-word contents of floating-point register 6 is made negative and the result is placed in floating-point register 2.

6.27. LNER (LOAD-NEGATIVE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LNER	r_1,r_2	31	RR	Two Bytes

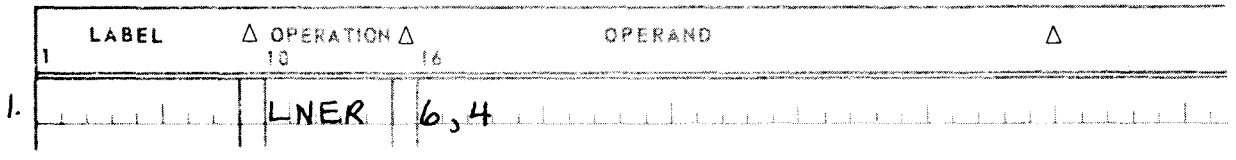
Function:

The sign of full-word operand 2, specified by r_2, is made negative and the result is placed in operand 1, specified by r_1.

Operational Considerations:

- Operand 2 is made negative even if the fraction is 0.
- The exponent and fraction are not changed.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The sign of the full-word contents of floating-point register 4 is made negative and the result is placed in floating-point register 6.

6.28. LPDR (LOAD-POSITIVE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LPDR	r_1,r_2	20	RR	Two Bytes

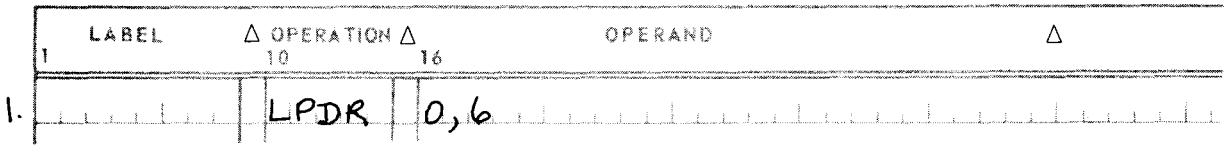
Function:

The sign of double-word operand 2, specified by r_2, is made positive and the result is placed in operand 1, specified by r_1.

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 2 (10_2) if result is greater than 0; or
  - codes 1 and 3 are not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The sign of the double-word contents of floating-point register 6 is made positive and the result is placed in register 0.

6.29. LPER (LOAD-POSITIVE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LPER	r_1,r_2	30	RR	Two Bytes

Function:

The sign of full-word operand 2, specified by r_2, is made positive and the result is placed in operand 1, specified by r_1.

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 2 (10_2) if result is greater than 0; or
  - codes 1 and 3 are not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:

```
      LABEL     Δ OPERATION Δ          OPERAND                Δ
      1         10          16
1.                LPER        4,0
```

1. The sign of the full-word contents of floating-point register 0 is made positive, and the result is placed in floating-point register 4.

6.30. LTDR (LOAD-AND-TEST, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LTDR	r_1,r_2	22	RR	Two Bytes

Function:

The contents of double-word operand 2, specified by r_2, are placed in operand 1, specified by r_1.

Operational Considerations:

- The contents of operand 2 remain unchanged.
- When the same register is specified for operand 1 and operand 2, the operation is equivalent to a test without data movement.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:

	LABEL (1)	△ OPERATION (10) △	OPERAND (16)	△
1.		LTDR	2,4	

1. The double-word contents of floating-point register 4 are placed in floating-point register 2 and the condition code is set.

6.31. LTER (LOAD-AND-TEST, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LTER	r_1,r_2	32	RR	Two Bytes

Function.

The contents of full-word operand 2, specified by r_2, are placed in operand 1, specified by r_1.

Operational Considerations:

- The contents of operand 2 remain unchanged.
- When the same register is specified for operand 1 and operand 2, the operation is equivalent to a test without data movement.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is less than 0;
  - to 2 (10_2) if result is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:

1. The full-word contents of floating-point register 2 are placed in floating-point register 0 and the condition code is set.

6.32. MD (MULTIPLY, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MD	$r_1,d_2(x_2,b_2)$	6C	RX	Four Bytes

Function:

The contents of double-word operand 1 (multiplicand), specified by r_1, are multiplied by the contents of double-word operand 2 (multiplier), specified by $d_2(x_2,b_2)$. The normalized product is placed in operand 1.

Operational Considerations:

- Floating-point multiplication consists of exponent addition and fraction multiplication. The exponent of the intermediate product is obtained by adding the exponents of the two operands and reducing the sum by 64.
- Both operands are normalized before multiplication and the intermediate product is normalized after multiplication. The intermediate product fraction is truncated to 14 digits and a guard digit (6.1) before normalization.
- If the exponent of the final product exceeds 127, an exponent overflow condition exists. The resultant floating-point number consists of a correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value. The overflow condition does not occur for an intermediate product exponent exceeding 127 if the final exponent is brought within range during normalization. An exponent overflow condition causes a program interrupt if the program exception mask bit of the current PSW is 1.
- If the final product exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. When an underflow characteristic becomes less than 0 during normalization before multiplication, an underflow exception is not recognized. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit in the current PSW are 1.
- When all digits of the intermediate product are 0, the result is made a true 0.
- When the result fraction is 0, a program exception for exponent underflow or overflow does not occur.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 not on double-word boundary, or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:

1. The double-word contents of floating-point register 2 are multiplied by the double-word contents of main storage location MULTP. The result is placed in register 2.

6.33. MDR (MULTIPLY, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MDR	r_1,r_2	2C	RR	Two Bytes

Function:

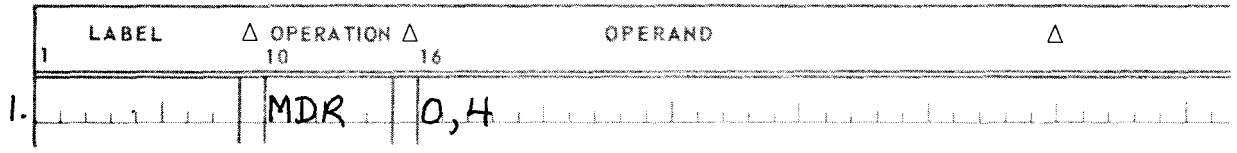
The contents of double-word operand 1 (multiplicand), specified by r_1, are multiplied by the contents of double-word operand 2 (multiplier), specified by r_2. The normalized product is placed in operand 1.

Operational Considerations:

- Floating-point multiplication consists of exponent addition and fraction multiplication. The exponent of the intermediate product is obtained by adding the exponents of the two operands and reducing the sum by 64.
- Both operands are normalized before multiplication and the intermediate product is normalized after multiplication. The intermediate product fraction is truncated to 14 digits and a guard digit before normalization.
- If the exponent of the final product exceeds 127, an exponent overflow condition exists. The resultant floating-point number consists of a correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value. The overflow condition does not occur for an intermediate product exponent exceeding 127 if the final exponent is brought within range during normalization. An exponent overflow condition causes a program interrupt if the program exception mask bit of the current PSW is 1.
- If the final product exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. When an underflow characteristic becomes less than 0 during normalization before multiplication, an underflow exception is not recognized. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit in the current PSW are 1.
- When all digits of the intermediate product are 0, the result is made a true 0.

- When the result fraction is 0, a program exception for exponent underflow or overflow does not occur.
- The condition code remains unchanged.
- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point register 0 are multiplied by the double-word contents of floating-point register 4. The result is placed in register 0.

6.34. ME (MULTIPLY, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ME	$r_1,d_2(x_2,b_2)$	7C	RX	Four Bytes

Function:

The contents of full-word operand 1 (multiplicand), specified by r_1, are multiplied by the contents of full-word operand 2 (multiplier), specified by $d_2(x_2,b_2)$. The normalized product is placed in operand 1.

Operational Considerations:

- Floating-point multiplication consists of exponent addition and fraction multiplication. The exponent of the intermediate product is obtained by adding the exponents of the two operands and reducing the sum by 64.
- Both operands are normalized before multiplication and the intermediate product is normalized after multiplication. The intermediate product fraction is truncated to 14 digits, the two least significant digits of which are 0, before normalization.

- If the exponent of the final product exceeds 127, an exponent overflow condition exists. The resultant floating-point number consists of a correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value. The overflow condition does not occur for an intermediate product exponent exceeding 127 if the final exponent is brought within range during normalization. An exponent overflow condition causes a program interrupt if the program exception mask bit of the current PSW is 1.

- If the final product exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. When an underflow characteristic becomes less than 0 during normalization before multiplication, an underflow exception is not recognized. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit in the current PSW are 1.

- When all digits of the intermediate product are 0, the result is made a true 0.

- When the result fraction is 0, a program exception for exponent underflow or overflow does not occur.

- The condition code remains unchanged.

- Possible program exceptions:

  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, or 6)

- Relocation and indirection flags:

  - operand 1: none
  - operand 2: RO, IO

Example:

```
     LABEL    Δ OPERATION Δ        OPERAND              Δ
     1        10          16
1.           | ME        | 6,NORM
```

1. The full-word contents of floating-point register 6 are multiplied by the full-word contents of main storage location NORM. The result is placed in register 6.

6.35. MER (MULTIPLY, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MER	r_1,r_2	3C	RR	Two Bytes

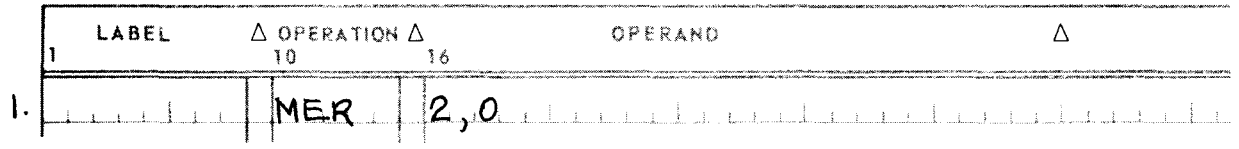
Function:

The contents of full-word operand 1 (multiplicand), specified by r_1, are multiplied by the contents of full-word operand 2 (multiplier), specified by r_2. The normalized product is placed in operand 1.

Operational Considerations:

- Floating-point multiplication consists of exponent addition and fraction multiplication. The exponent of the intermediate product is obtained by adding the exponents of the two operands and reducing the sum by 64.
- Both operands are normalized before multiplication and the intermediate product is normalized after multiplication. The intermediate product fraction is truncated to 14 digits, the two least significant digits of which are 0, before normalization.
- If the exponent of the final product exceeds 127, an exponent overflow condition exists. The resultant floating-point number consists of a correct and normalized fraction, a correct sign, and an exponent which is 128 less than the correct value. The overflow condition does not occur for an intermediate product exponent exceeding 127 if the final exponent is brought within range during normalization. An exponent overflow condition causes a program interrupt if the program exception mask bit of the current PSW is 1.
- If the final product exponent is less than 0, an exponent underflow condition exists. If the exponent underflow mask bit of the current PSW is 1, the resultant floating-point number has a correct and normalized fraction, a correct sign, and an exponent which is 128 greater than the correct value. If the exponent underflow mask bit of the current PSW is 0, the result is made a true 0. When an underflow characteristic becomes less than 0 during normalization before multiplication, an underflow exception is not recognized. An exponent underflow exception causes a program interrupt if the exponent underflow mask bit and the program exception mask bit in the current PSW are 1.
- When all digits of the intermediate product are 0, the result is made a true 0.
- When the result fraction is 0, a program exception for exponent underflow or overflow does not occur.
- The condition code remains unchanged.
- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - operation exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point register 2 are multiplied by the full-word contents of floating-point register 0. The result is placed in register 2.

6.36. SD (SUBTRACT-NORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SD	$r_1,d_2(x_2,b_2)$	6B	RX	Four Bytes

Function:

The double-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically subtracted from the double-word contents of operand 1, specified by r_1. The normalized difference is placed in operand 1.

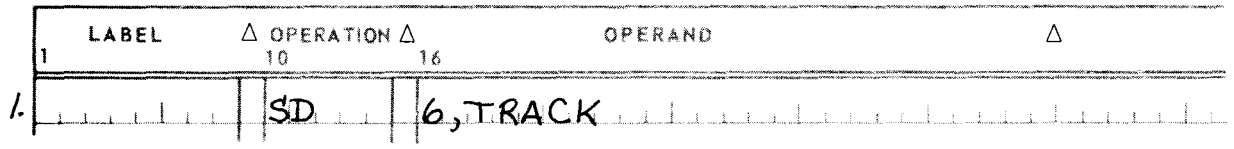
Operational Considerations:

- The execution of the SD instruction is identical to that of the AD instruction, except that the sign of operand 2 is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception

- protection exception
- significance exception
- specification exception (operand 2 not on double-word boundary, or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)

- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The double-word contents of main storage location TRACK are subtracted from the double-word contents of floating-point register 6. The result is placed in register 6.

6.37. SDR (SUBTRACT-NORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SDR	r_1,r_2	2B	RR	Two Bytes

Function:

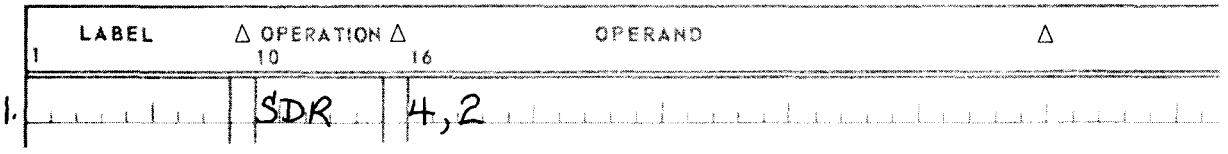
The double-word contents of operand 2, specified by r_2, are algebraically subtracted from the double-word contents of operand 1, specified by r_1. The normalized difference is placed in operand 1.

Operational Considerations:

- The execution of the SDR instruction is identical to that of the ADR instruction (6.3), except that the sign of operand 2 is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.

- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The double-word contents of floating-point register 2 are subtracted from the double-word contents of floating-point register 4. The result is placed in register 4.

6.38. SE (SUBTRACT-NORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SE	$r_1,d_2(x_2,b_2)$	7B	RX	Four Bytes

Function:

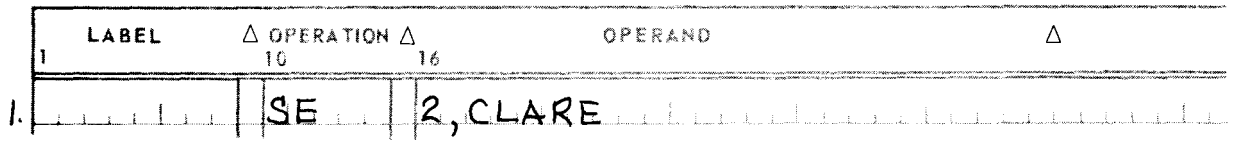
The full-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically subtracted from the full-word contents of operand 1, specified by r_1. The normalized difference is placed in operand 1.

Operational Considerations:

- The execution of the SE instruction is identical to that of the AE instruction (6.4), except that the sign of operand 2 is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.

- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - exponent underflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 or IACW not on full-word boundary; or operand 1 register is not 0, 2, 4, 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of main storage location CLARE are subtracted from the contents of floating-point register 2. The result is placed in register 2.

6.39. SER (SUBTRACT-NORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SER	r_1,r_2	3B	RR	Two Bytes

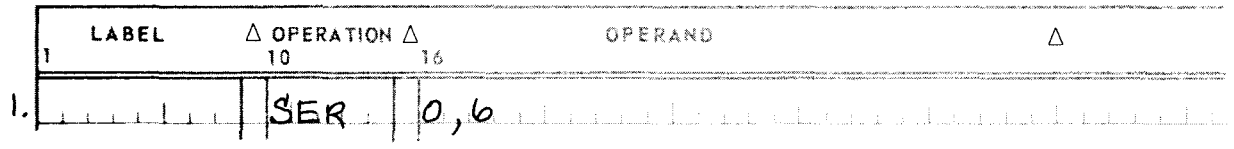
Function:

The full-word contents of operand 2, specified by r_2, are algebraically subtracted from the full-word contents of operand 1, specified by r_1. The normalized difference is placed in operand 1

Operational Considerations:

- The execution of the SER instruction is identical to that of the AER instruction (6.5), except that the sign of operand 2 is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - exponent overflow exception
  - exponent underflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:



1. The full-word contents of floating-point register 6 are subtracted from the full-word contents of floating-point register 0. The result is placed in register 0.

6.40. STD (STORE, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
STD	$r_1,d_2(x_2,b_2)$	60	RX	Four Bytes

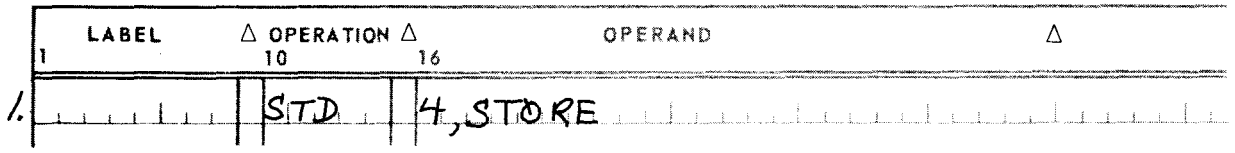
Function:

The double-word contents of operand 1, specified by r_1, are placed in main storage at the location designated by operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 not on double-word boundary, or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RD, ID

Example:



1. The double-word contents of floating-point register 4 are placed in main storage location STORE.

6.41. STE (STORE, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
STE	$r_1,d_2(x_2,b_2)$	70	RX	Four Bytes

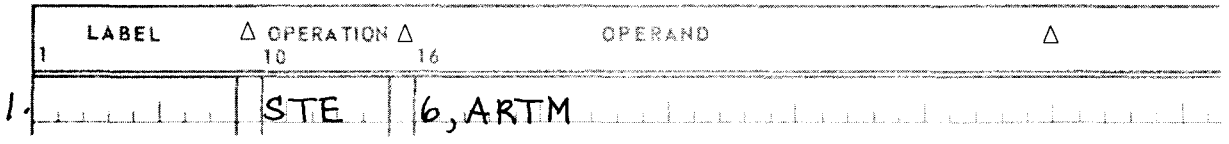
Function:

The full-word contents of operand 1, specified by r_1, are placed in main storage at the location designated by operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RD, ID

Example:



1. The full-word contents of floating-point register 6 are placed in main storage location ARTM.

6.42. SU (SUBTRACT-UNNORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SU	$r_1,d_2(x_2,b_2)$	7F	RX	Four Bytes

Function:

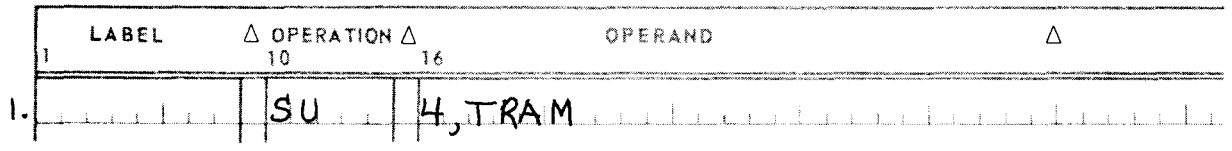
The full-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically subtracted from the full-word contents of operand 1, specified by r_1. The difference is placed in operand 1.

Operational Considerations:

- The execution of the SU instruction is identical to that of the AU instruction (6.6), except that the sign is reversed before addition.

- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The full-word contents of main storage location TRAM are subtracted from the full-word contents of floating-point register 4. The result is placed in register 4.

6.43. SUR (SUBTRACT-UNNORMALIZED, SHORT FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SUR	r_1,r_2	3F	RR	Two Bytes

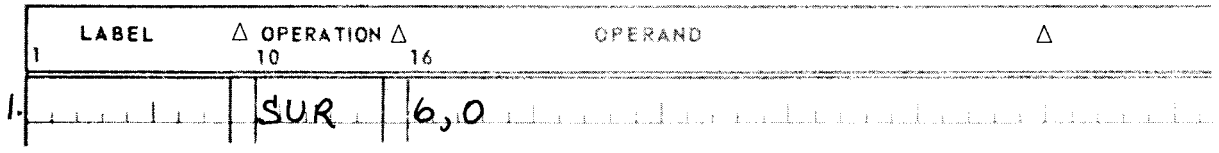
Function:

The full-word contents of operand 2, specified by r_2, are algebraically subtracted from the full-word contents of operand 1, specified by r_1. The difference is placed in operand 1.

Operational Considerations:

- The execution of the SUR instruction is identical to that of the AUR instruction (6.7), except that the sign is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - exponent overflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)

Example:



1. The full-word contents of floating-point register 0 are subtracted from the full-word contents of floating-point register 6. The result is placed in register 6.

6.44. SW (SUBTRACT-UNNORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SW	$r_1,d_2(x_2,b_2)$	6F	RX	Four Bytes

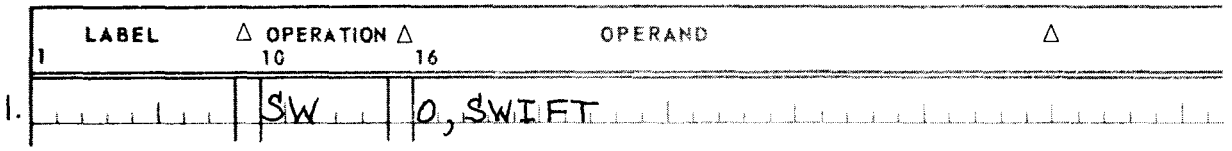
Function:

The double-word contents of operand 2, specified by $d_2(x_2,b_2)$, are algebraically subtracted from the double-word contents of operand 1, specified by r_1. The difference is placed in operand 1.

Operational Considerations:

- The execution of the SW instruction is identical to that of the AW instruction (6.8), except that the sign is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - exponent overflow exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception
  - protection exception
  - significance exception
  - specification exception (operand 2 not on double-word boundary, or IACW not on full-word boundary, or operand 1 register is not 0, 2, 4, or 6)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The double-word contents of main storage location SWIFT are subtracted from the double-word contents of floating-point register 0. The result is placed in register 0.

6.45. SWR (SUBTRACT-UNNORMALIZED, LONG FORMAT) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SWR	r_1,r_2	2F	RR	Two Bytes

Function:

The double-word contents of operand 2, specified by r_2, are algebraically subtracted from the double-word contents of operand 1, specified by r_1. The difference is stored in operand 1.

Operational Considerations:

- The execution of the SWR instruction is identical to that of the AWR instruction (6.9), except that the sign is reversed before addition.
- The condition code is set as follows:
  - to 0 (00_2) if result fraction is 0;
  - to 1 (01_2) if result fraction is less than 0;
  - to 2 (10_2) if result fraction is greater than 0; or
  - code 3 is not used.
- Possible program exceptions:
  - exponent overflow exception
  - operation exception
  - significance exception
  - specification exception (operand 1 or operand 2 register is not 0, 2, 4, or 6)
- Relocation and indirection flags: none

Example:

	LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16)	Δ
1.		SWR	6,2	

1. The double-word contents of floating-point register 2 are subtracted from the double-word contents of floating-point register 6. The result is placed in register 6.

7. Logical Instructions

7.1. GENERAL

The logical instruction set provides for the adding, subtracting, moving, comparing, bit manipulating, bit testing, translating, editing, and shifting of logical operands. A logical operand may be a full word or double word, a single character, or a variable-length field. Depending on the instruction, logical operands may be treated as unsigned integers or as unsigned and signed, packed and unpacked fields.

Logical operands are available in the RR, RX, RS, SS, and SI formats. The operands may reside in the general registers, in main storage, or within a field in the instruction itself. On the SPERRY UNIVAC 9400/9480 Systems, the address of an operand in main storage is specified as absolute. On the SPERRY UNIVAC 90/60,70 Systems, the address of an operand in main storage may be specified as relative or absolute and direct or indirect under the control of the applicable relocation register flags. The first and second operand fields in main storage may overlap in any way; however, unpredictable results may occur during translation and editing operations. The effect of overlapping may be understood by considering the operands to be processed one byte at a time from left to right.

This section describes the operation of each logical instruction. The instructions are arranged in alphabetical order according to mnemonic operation code. Each description includes a list of the possible program exceptions and condition codes which may result. The relocation and indirection flags that are pertinent to the operand addresses are listed. The object code format of the instruction is shown only for those instructions which differ from the format shown in Figure 3–1. See Table 3–1 for an explanation of the abbreviations used in describing instruction formats.

7.2. AL (ADD-LOGICAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
AL	$r_1,d_2(x_2,b_2)$	5E	RX	Four Bytes

Function:

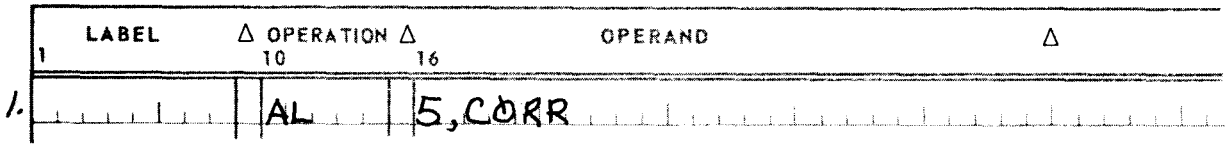
The contents of operand 2, specified by $d_2(x_2,b_2)$, are logically added to the contents of operand 1, specified by r_1, and the sum is placed in operand 1.

Operational Considerations:

- Logical addition is performed by adding all 32 bits of each operand.
- The contents of operand 2 remain unchanged.

- The condition code is set as follows:
  - to 0 (00_2) if result is 0; no carry out of most significant bit;
  - to 1 (01_2) if result is not 0; no carry out of most significant bit;
  - to 2 (10_2) if result is 0; carry out of most significant bit; or
  - to 3 (11_2) if result is not 0; carry out of most significant bit.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The contents of the main storage location CORR are logically added to the contents of register 5. The result is placed in register 5.

7.3. ALR (ADD-LOGICAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ALR	r_1,r_2	1E	RR	Two Bytes

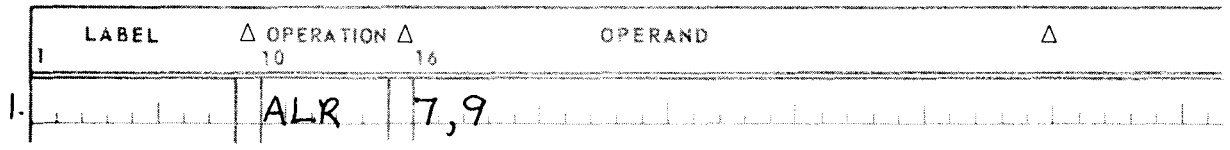
Function:

The contents of operand 2, specified by r_2, are logically added to the contents of operand 1, specified by r_1, and the sum is placed in operand 1.

Operational Considerations:

- Logical addition is performed by adding all 32 bits of each operand.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0; no carry out of most significiant bit;
  - to 1 (01_2) if result is not 0; no carry out of most significant bit;
  - to 2 (10_2) if result is 0; carry out of most significant bit; or
  - to 3 (11_2) if result is not 0; carry out of most significant bit.
- Possible program exceptions: none
- Relocation and indirection flags: none

Example:



1. The contents of register 9 are logically added to the contents of register 7. The result is placed in register 7.

7.4. CL (COMPARE-LOGICAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CL	$r_1,d_2(x_2,b_2)$	55	RX	Four Bytes

Function:

The contents of operand 1, specified by r_1, and operand 2, specified by $d_2(x_2,b_2)$, are compared and the condition code is set according to the comparison.

Operational Considerations:

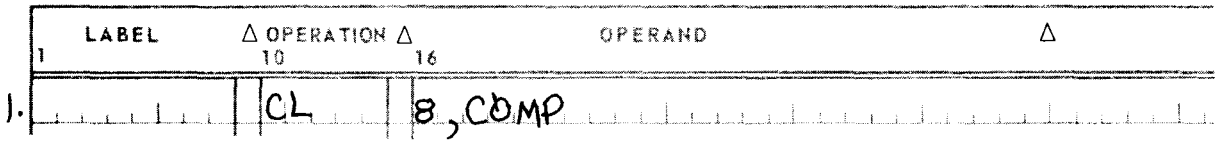
- Operands are considered as unsigned binary numbers and all bit combinations are valid.
- The contents of both operands remain unchanged.

- The condition code is set as follows:
  - to 0 (00_2) if the operands are equal;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The contents of register 8 are compared with the contents of main storage location COMP.

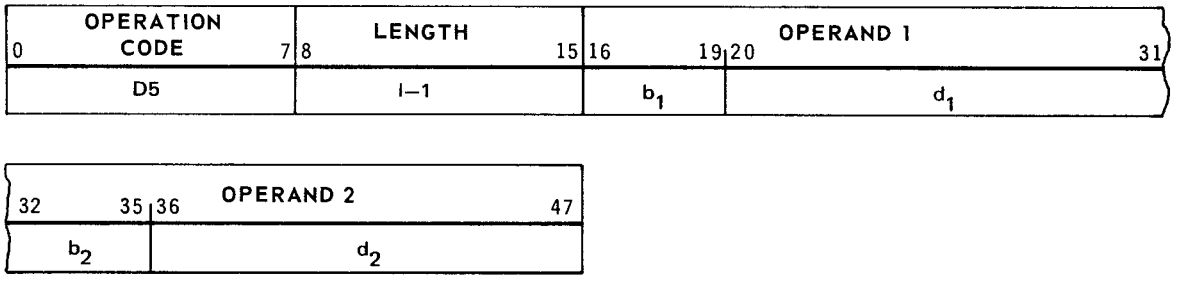
7.5. CLC (COMPARE-LOGICAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CLC	$d_1(l,b_1),d_2(b_2)$	D5	SS	Six Bytes

Function:

The contents of operand 1, specified by $d_1(l,b_1)$, and operand 2, specified by $d_2(b_2)$, are compared and the condition code is set according to the comparison.

Object Instruction Format:



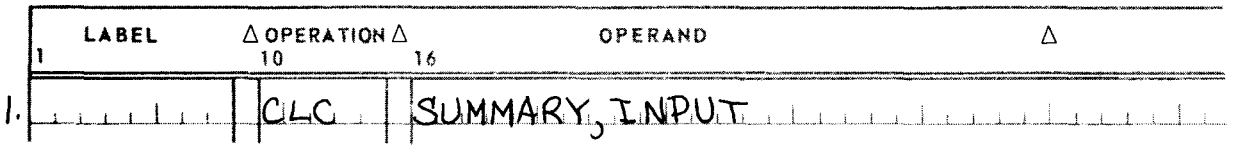
Operational Considerations:

- The length specification of operand 1 specifies the length of both operands.
- Operands are considered unsigned binary numbers and all bit combinations are valid.
- The contents of both operands remain unchanged.
- The instruction is processed from left to right, byte by byte.
- The condition code is set as follows:
  - to 0 (00_2) if the operands are equal;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. The contents of main storage location SUMMARY are compared with the contents of main storage location INPUT. SUMMARY and INPUT are labels with defined locations and lengths. The length attribute of the symbol SUMMARY determines the length of the operands.

7.6. CLI (COMPARE-LOGICAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CLI	$d_1(b_1),i_2$	95	SI	Four Bytes

Function:

The one-byte contents of operand 1, specified by $d_1(b_1)$, and operand 2, contained in the i_2 field, are compared and the condition code is set according to the comparison.

Operational Considerations:

- Operands are considered unsigned binary numbers and all bit combinations are valid.
- The contents of operand 1 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if the operands are equal;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RO, IO
  - operand 2: none

Examples:

```
   LABEL      △ OPERATION △          OPERAND                        △
1              10          16
1.             CLI         VALUE,100
2.             CLI         TEST,X'03'
```

1. The contents of main storage location VALUE are compared with 100.
2. The contents of main storage location TEST are compared with the hexadecimal value 3.

7.7. CLR (COMPARE-LOGICAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
CLR	r_1,r_2	15	RR	Two Bytes

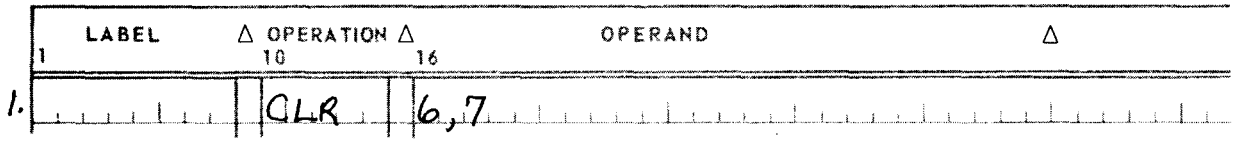
Function:

The contents of operand 1, specified by r_1, and the contents of operand 2, specified by r_2, are compared and the condition code is set according to the comparison.

Operational Considerations:

- Operands are considered unsigned binary numbers and all bit combinations are valid.
- The contents of both operands remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if the operands are equal;
  - to 1 (01_2) if operand 1 is less than operand 2;
  - to 2 (10_2) if operand 1 is greater than operand 2; or
  - code 3 is not used.
- Possible program exceptions: none.
- Relocation and indirection flags (90/60,70): none

Example:



1. The contents of register 6 and the contents of register 7 are logically compared and the condition code is set.

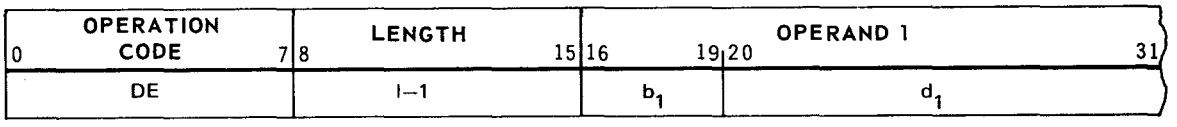
7.8. ED (EDIT)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ED	$d_1(l,b_1),d_2(b_2)$	DE	SS	Six Bytes

Function:

The data of operand 2, specified by $d_2(b_2)$, is changed from packed to unpacked format, is edited under the control of operand 1 (the mask), and is placed in the operand 1 location, specified by $d_1(l,b_1)$.

Object Instruction Format:



Operational Considerations:

- Editing includes sign and punctuation control and the suppression and protection of leading 0's. It also facilitates programmed blanking for all 0 fields. Several fields may be edited in one operation, and numeric information may be combined with text.
- The instruction proceeds from left to right.
- Operand 2 data must be in packed format and must contain valid numerics and sign codes.
- The original contents of operand 1 is the mask, the pattern which controls the edit process. Depending on the edit requirements, some of the bytes originally in operand 1 are replaced by data from operand 2. The mask is expressed in unpacked format and may consist of any combination of 8-bit characters.

- As the mask is scanned from left to right, one of three things happens to each mask character:

  – An operand 2 digit is expanded to a zoned character. The zoned character replaces the mask character. When the operand 2 digit is stored as the result, its code is expanded from packed to unpacked format by attaching a generated zone code. When the A mode bit of the current program status word is 0, the EBCDIC zone code (1111_2) is generated; when the A mode bit is 1, the ASCII zone code (0011_2) is generated.

  – The mask character is left unchanged.

  – A fill character is stored in the result. The fill character is taken from the first byte position of the mask. The choice of this character is not dependent upon the editing function initiated by this code. The editing function occurs after the code has been assigned as a fill character.

  Each mask character is replaced by a result character that depends on three conditions:

  – the digit obtained from operand 2;

  – the mask character; and

  – the S switch status.

  When a digit select or significance start byte is found in the mask, the S switch and an operand 2 digit are examined. This results in either the unpacked operand 2 digit or the fill character replacing the mask character. A valid decimal digit (if mask byte is significance start) or nonzero decimal digit (if mask byte is digit select) sets the S switch to on if the operand 2 byte does not contain a plus code in the four least significant bit positions (Table 7–1).

- Significance Indicator (S Switch)

  The significance indicator, referred to as the S switch, indicates by its on or off state the significance or nonsignificance, respectively, of subsequent operand 2 digits or message characters. Significant operand 2 digits replace their corresponding digit select or significance start characters in the result. Significant message characters remain unchanged in the result.

  When the S switch is off, 0's to be transferred from operand 2 are suppressed and the fill character is inserted in the corresponding operand 1 position. When the S switch is on, any 0 to be transferred from operand 2 is unpacked into the corresponding operand 1 position. At the beginning of execution the S switch is off.

- Fill Character

  The fill character is the leftmost character of the edit mask (operand 1). Any hexadecimal value (Appendix C) may be used as a valid fill character. This character is retained for the editing which follows. This position does not receive a digit from the operand 2 data.

- Digit Select Byte

  The digit select byte is a character in the operand 1 mask represented by the EBCDIC code 20 or the ASCII code 80. If the digit select byte is encountered and the S switch is on, any digit, 0 through 9, is unpacked to replace the digit select byte. If the S switch is off, the operand 2 digit is examined and only nonzero digits are unpacked into operand 1. The fill character replaces the digit select byte if the examined digit is 0. The S switch is turned on when the first nonzero operand 2 digit is encountered; this allows succeeding 0's from operand 2 to be included in the result.

- Significance Start Byte

  The significance start byte is represented in the edit mask by the EBCDIC code 21 or the ASCII code 81. The significance start byte performs the same function as the digit select byte except the significance start byte turns the S switch on regardless of the value of the current operand 2 digit. Once the S switch is on, it remains on for all succeeding digits; however, the current digit is not affected. The S switch may be turned off by a field separator byte or by a positive sign code within operand 2.

- Message Character

  Any other symbol or data in the operand 1 edit mask, as represented by hexadecimal codes, is retained unchanged if the S switch is on. If the S switch is off, the other data is replaced by the fill character. During this operation, the digit of operand 2 is neither accessed nor address advanced.

- Field Separator Byte

  Multiple-field editing operations are indicated by the presence of one or more field separator bytes (EBCDIC code 22, ASCII code 82). The field separator byte identifies the individual fields in this operation and is always replaced in the mask with a fill character. The S switch is always off after the field separator byte is encountered. If field separators are not indicated by the mask, the entire operand 2 is considered one field.

- Sign Consideration for Operand 2

  The sign of operand 2, positive or negative, must be a value greater than binary 9 (1001_2). Any hexadecimal value A through F is acceptable. The sign itself is not moved to operand 1; instead, a sign indicator, such as a minus sign or letters CR, is either deleted from or retained in operand 1, depending on the sign of operand 2.

  The sign of operand 2 also affects the S switch. A positive sign turns the S switch off, thus causing the following characters in operand 1 to be replaced by the fill character. A negative sign leaves the S switch unchanged.

- If the fill character is a blank, if no significance start byte appears in the mask, and if operand 2 is all 0's, the editing operation blanks the result field.

- Overlapping operand 1 and operand 2 fields produce unpredictable results.

- Operand Length

  The length specification in the object instruction specifies the length of the mask (operand 1). The length of the mask can be determined as follows:

  – one byte for the fill character;

  – one byte for each digit select byte, significance start byte, and field separator byte; and

  – one byte for each message character.

  Usually operand 2 is shorter than operand 1, since, for each operand 2 digit, a zone and a numeric are inserted in the result. The total number of digit select and significance start bytes in the mask must equal the number of operand 2 digits to be edited.

- If operand 2 containing unpacked data is to be edited, it must first be packed by the PACK instruction. In packing an odd number of bytes, an odd number of digit positions and the sign are produced. In packing an even number of bytes, an odd number of digit positions and the sign are produced. The extra digit position in the latter case is 0, and is the most significant position in operand 2. The extra position must be provided for in the mask by specifying an extra DSB or SSB. Space, asterisk, or other character fill occurs and may be dropped when transferring the edited operand to output.

- The condition code, reflecting the status of the last source field edited, is set as follows:

  – to 0 when all of the operand 2 digits in the last field are 0. If the mask of the last field has no significance start or digit select bytes, the operand 2 digits are not examined and the condition code is set to 0;

  – to 1 (01_2) when a nonzero operand 2 digit without an associated plus sign is detected;

  – to 2(10_2) when an operand 2 digit greater than zero is detected; or

  – code 3 is not used.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	
Data exception (invalid sign or digit code)	

- Relocation and indirection flags (90/60,70):

  – operand 1: RD, ID

  – operand 2: RO, IO

- The operation of the edit instruction is summarized in Table 7–1.

Table 7–1. Edit Instruction Operation

Mask (Operand 1) Character	EBCDIC/ASCII Code	S Switch Status	Data (Operand 2) Character	Resulting (Operand 1) Character	Resulting S Switch Status
Fill character	Any	Off	Not examined	Fill character	Off
Digit select byte	20/80	On	Digit	Digit	On*
		Off	Nonzero	Digit	On*
		Off	Zero	Fill character	Off
Significance start byte	21/81	On	Digit	Digit	On*
		Off	Nonzero	Digit	On*
		Off	Zero	Fill character	On*
Message character	Any except: 20/80, 21/81, 22/82	On	Not examined	Message character	On*
		Off	Not examined	Fill character	Off
Field separator byte	22/82	On	Not examined	Fill character	Off
		Off	Not examined	Fill character	Off

* Sign detection (examined simultaneously with operand 2 digit) affects the S switch as follows:
A plus sign detected as a least significant digit causes the S switch to be turned off.
A minus sign has no effect on the S switch.

Examples:

```
     LABEL      OPERATION      OPERAND                                          COMMENTS
     1          10             16
1.   MASK       DC             XL13'40206B20202206B20202214B2020'
           .    EDT            MASK(13),AMT
2.   MASK       DC             XL15'40206B20202020B20202214B2020C3D9'
           .    EDT            MASK(15),VALU
3.   MASK       DC             XL22'5C20206B20202204B20202221220202220 6B20202048'
           .    EDT            MASK(22),FIELD
```

1. The packed operand AMT is edited according to the mask MASK and the result is placed in the operand 1 location. Assume AMT to contain the value 000012698. This value appears in five bytes as follows:

   0 0 0 0 1 2 6 9 8 +

   The edit mask appears in 13 bytes as follows:

   4 0 2 0 6 B 2 0 2 0 2 0 6 B 2 0 2 1 4 B 2 0 2 0

   The result is:

   ΔΔΔΔΔΔ126.98.

2. The packed operand VALU is edited according to the mask MASK and the result is placed in the operand 1 location. If operand 2 is negative, the letters CR are printed to the right of the result amount. Assume VALU to contain the value 000456789. This value appears in five bytes as follows:

   0 0 0 4 5 6 7 8 9–

   The edit mask appears in 15 bytes as follows:

   4 0 2 0 6 B 2 0 2 0 2 0 6 B 2 0 2 0 2 1 4 B 2 0 2 0 C 3 D 9

   The result appears as:

   ΔΔΔΔ4,567.89CR.

   This example employs the field separator byte to allow editing of three fields with one edit instruction.

   Packed value of operand 2:

   0 0 1 2 3 4 5 C 1 2 3 C 1 2 3 4 5 C

   Edit mask in hexadecimal:

   5 C 2 0 2 0 6 B 2 0 2 0 2 0 4 B 2 0 2 0 2 2 2 1 2 0 2 0 2 2 2 0 2 0 6 B 2 0 2 0 2 0 4 B

   The edited result is:

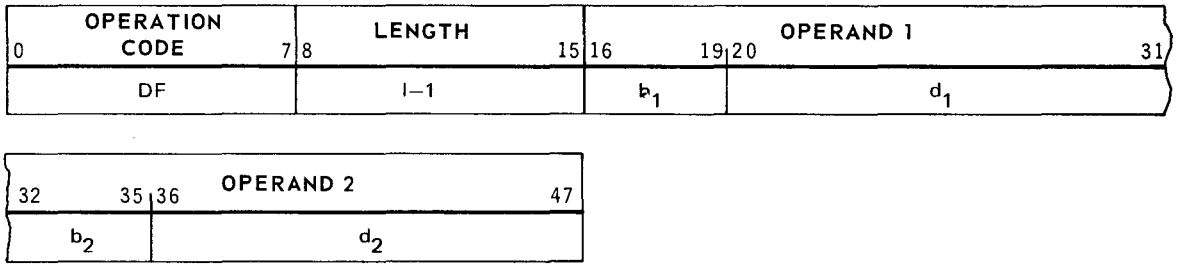
   * * * * 1 2 3 . 4 5 * 1 2 3 * 1 2 , 3 4 5

7.9. EDMK (EDIT-AND-MARK) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
EDMK	$d_1(l,b_1),d_2(b_2)$	DF	SS	Six Bytes

The data of operand 2, specified by $d_2(b_2)$, is changed from packed to unpacked format, is edited under the control of operand 1 (the mask), and is placed in the operand 1 location, specified by $d_1(l,b_1)$. The address of the first significant result character is placed in general register 1.

Object Instruction Format:



Operational Considerations:

- The EDMK instruction is identical to the ED instruction with the addition of storing the address of the first significant result character in the least significant 24 bits of general register 1. This occurs when the result character is a digit from 1 to 9 and the S switch was off before examination of the digit.

- When an EDMK instruction is used to edit more than one field, the address of each succeeding field replaces the contents of the least significant 24 bits of general register 1; therefore, only the first significant character of the last field edited is available.

- This instruction is used to faciliate currency symbol insertion. The address stored in register 1 is one more than the address where a currency symbol must be inserted. The branch-on-count instruction with 0 in the operand 2 field is used to reduce the inserted address by 1 (BCTR 1,0).

- The condition code, reflecting the status of the last source field edited, is set as follows:

  – to 0 when all of the operand 2 digits in the last field are 0. If the mask of the last field has no significance start or digit select bytes, the operand 2 digits are not examined and the condition code is set to 0.

  – to 1 (01_2) when a nonzero operand 2 digit is detected and the S switch is set on after the last mask digit is examined;

  – to 2 (10_2) when a nonzero operand 2 digit is detected and the S switch is set off after the last mask digit is examined; or

  – code 3 is not used.

- Possible program exceptions:

  – addressing exception

  – data exception (invalid sign or digit code)

  – indirect address specification exception

  – indirect addressing exception

  – protection exception

  – specification exception (IACW not on full-word boundary)

- Relocation and indirection flags:

  – operand 1: RD, ID

  – operand 2: RO, IO

Example:

```
    LABEL      Δ OPERATION Δ          OPERAND                Δ
1              10          16
1.             EDMK        MASK(14),TOTAL
```

1. The contents of main storage location TOTAL are edited according to the contents of main storage location MASK. The result is placed in the MASK location, and the address of the first significant character is placed in general register 1.

7.10. IC (INSERT CHARACTER)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
IC	$r_1,d_2(x_2,b_2)$	43	RX	Four Bytes

Function:

The byte of main storage in the operand 2 location, specified by $d_2(x_2,b_2)$, is placed in the least significant eight bits of the operand 1 register, specified by r_1.

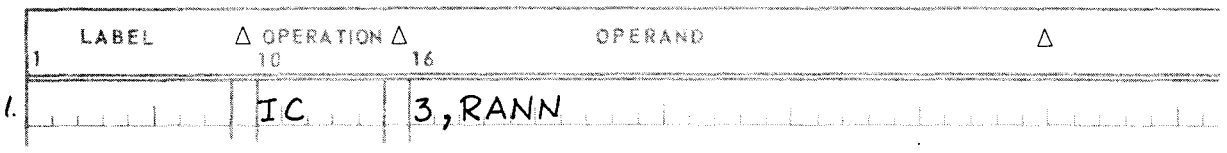
Operational Considerations:

- The contents of operand 2 remain unchanged.
- The contents of the most significant 24 bits of the operand 1 register remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The byte of main storage labeled RANN is placed into the least significant byte of general register 3.

7.11. LA (LOAD-ADDRESS)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LA	$r_1,d_2(x_2,b_2)$	41	RX	Four Bytes

Function:

The main storage operand 2 address, specified by $d_2(x_2,b_2)$, is loaded into the least significant bits of operand 1, specified by r_1. The most significant bits of r_1 are set to 0.

Operational Considerations

- The generated address is not checked for validity.
- The contents of operand 2 remain unchanged.
- If x_2 or b_2 specifies the same register as r_1, the contents of the register are incremented by the value specified as d_2.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Operational Differences:

- 9400/9480 systems

  The storage address of operand 2, specified by $d_2(x_2,b_2)$, is loaded into the least significant 17 bits of operand 1, specified by r_1. The most significant 15 bits of r_1 are set to 0.

- 90/60,70 systems

  The storage address of operand 2, specified by $d_2(x_2,b_2)$, is loaded into the least significant 24 bits of operand 1, specified by r_1. The most significant 8 bits of r_1 are set to 0.

Examples:

	LABEL	△ OPERATION △	OPERAND △
1.		LA	10,ADDR
2.		LA	8,25(8,0)
3.		LA	10,16(0,10)

1. The main storage address labeled ADDR is loaded into register 10.

2. The contents of register 8 are incremented by 25, and the most significant bits are set to 0.

3. The contents of register 10 are incremented by 16, and the most significant bits are set to 0.

7.12. MVC (MOVE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MVC	$d_1(l,b_1),d_2(b_2)$	D2	SS	Six Bytes

Function:

The contents of operand 2, specified by $d_2(b_2)$, are placed in the operand 1 location, specified by $d_1(l,b_1)$.

Object Instruction Format:

OPERATION CODE (0–7)	LENGTH (8–15)	OPERAND 1 (16–19)	OPERAND 1 (20–31)
D2	l–1	b_1	d_1

OPERAND 2 (32–35)	OPERAND 2 (36–47)
b_2	d_2

Operational Considerations:

- The transfer proceeds from left to right.
- The number of bytes transferred is specified by the length field in operand 1.
- The contents of operand 2 remain unchanged.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Examples:

```
      LABEL     Δ OPERATION Δ        OPERAND                    Δ
      1         10          16
1.              MVC         DESTIN,ORIGIN
2.              MVC         DESTIN(20),ORIGIN
```

1. The contents of main storage location ORIGIN are transferred to main storage location DESTIN. The length is implied by DESTIN.
2. A length of 20 overrides the implied length of DESTIN.

7.13. MVI (MOVE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MVI	$d_1(b_1),i_2$	92	SI	Four Bytes

Function:

The data contained in the i_2 field is moved to the main storage byte of operand 1, specified by $d_1(b_1)$.

Operational Considerations:

- The condition code remains unchanged.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):

  – operand 1: RD, ID

  – operand 2: none

Examples:

	LABEL (1)	△ OPERATION △ (10)	OPERAND (16) △
1.		MVI	STORE,X'9E'
2.		MVI	0(10),12

1. The hexadecimal value 9E (10011110_2) is placed in main storage location STORE.

2. The binary value of the decimal number 12 is stored in the main storage location specified by the address value 0 modified by the contents of base register 10.

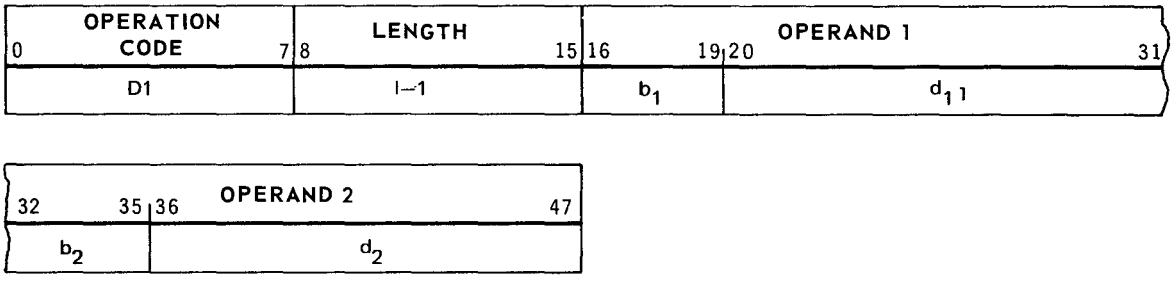
7.14. MVN (MOVE-NUMERICS)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MVN	$d_1(l,b_1),d_2(b_2)$	D1	SS	Six Bytes

Function:

The least significant four bits (the numeric portion) of each byte of operand 2, specified by $d_2(b_2)$, are transferred to the least significant four bits of each byte of operand 1, specified by $d_1(l,b_1)$.

Object Instruction Format:



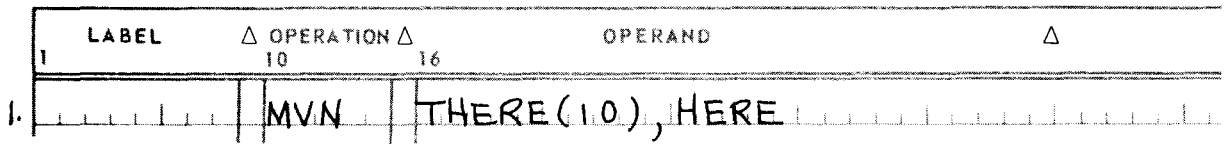
Operational Considerations:

- The four most significant bits of each byte (zone portion) of operand 1 remain unchanged.
- The contents of operand 2 remain unchanged.
- Overlapping of operands is permitted.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. The numeric portions of 10 consecutive bytes are transferred from main storage location HERE to main storage location THERE.

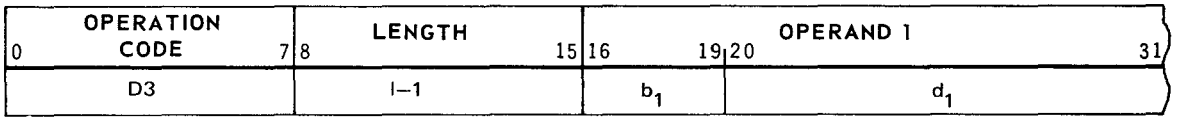
7.15. MVZ (MOVE-ZONES)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
MVZ	$d_1(l,b_1),d_2(b_2)$	D3	SS	Six Bytes

Function:

The most significant four bits (zone portion) of each byte of operand 2, specified by $d_2(b_2)$, are transferred to the most significant four bits of each byte of operand 1, specified by $d_1(l,b_1)$.

Object Instruction Format:



Operational Considerations:

- The four least significant bits of each byte (numeric portion) of operand 1 remain unchanged.
- The contents of operand 2 remain unchanged.
- Overlapping of operands is permitted.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:

```
      LABEL     Δ OPERATION Δ          OPERAND                Δ
      1         10          16
1.    |         | MVZ       | TRAY(25),SARD
```

1. The zone portions of 25 bytes are transferred from main storage location SARD to main storage location TRAY.

7.16. N (AND)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
N	$r_1,d_2(x_2,b_2)$	54	RX	Four Bytes

Function:

A logical product (AND) operation is performed on the contents of operand 1, specified by r_1, and operand 2, specified by $d_2(x_2,b_2)$. The result is stored in operand 1.

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in both operations contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical product are illustrated by the following truth table:

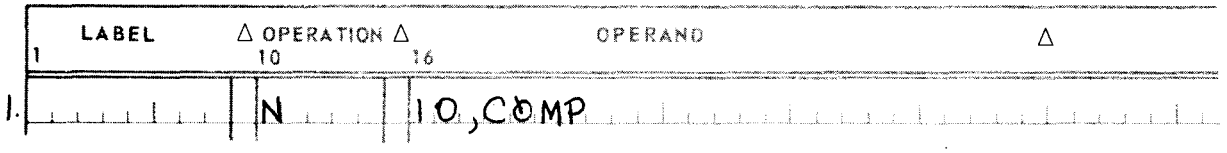
Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

- It is possible to clear selected bits in operand 1 by specifying 0's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The logical product of the contents of register 10 and main storage location COMP produces a result which is stored in register 10.

7.17. NC (AND)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
NC	$d_1(l,b_1),d_2(b_2)$	D4	SS	Six Bytes

Function:

A logical product (AND) operation is performed on the contents of operand 1, specified by $d_1(l,b_1)$, and operand 2, specified by $d_2(b_2)$. The result is stored in operand 1.

Object Instruction Format:

0 OPERATION CODE 7	8 LENGTH 15	16 OPERAND 1 19	20 OPERAND 1 31
D4	I–1	b_1	d_1

32 OPERAND 2 35	36 OPERAND 2 47
b_2	d_2

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in both operations contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical product are illustrated by the following truth table:

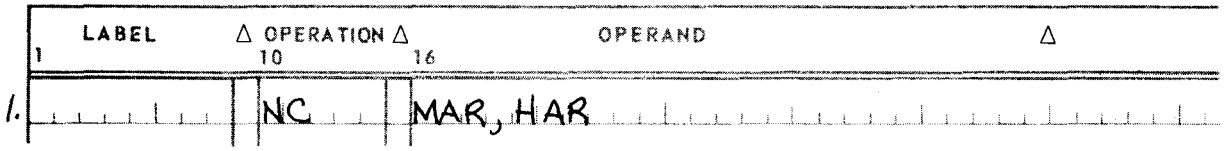
Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

- It is possible to clear selected bits in operand 1 by specifying 0's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. The logical product of the contents of main storage locations MAR and HAR produces a result which is stored in MAR. The length is implied by MAR.

7.18. NI (AND)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
NI	$d_1(b_1),i_2$	94	SI	Four Bytes

Function:

A logical product (AND) operation is performed on the contents of the operand 1 byte, specified by $d_1(b_1)$, and operand 2, contained in the i_2 field. The result is stored in operand 1.

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in both operations contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical product are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

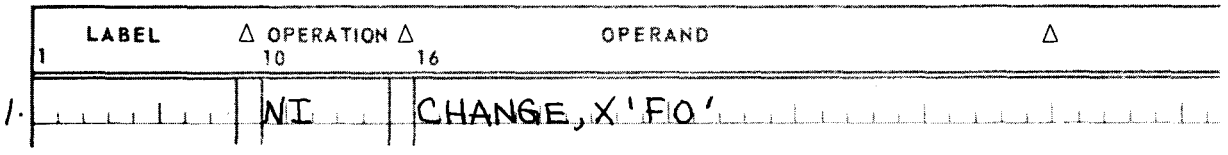
- It is possible to clear selected bits in operand 1 by specifying 0's in the corresponding bit positions of operand 2.

- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: none

Example:



1. Assume that CHANGE addresses a byte in main storage containing the following bit configuration:

CHANGE before execution	10101010
F0	11110000
CHANGE after execution	10100000

7.19. NR (AND)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
NR	r_1,r_2	14	RR	Two Bytes

Function:

A logical product (AND) operation is performed on the contents of operand 1, specified by r_1, and operand 2, specified by r_2. The result is stored in operand 1.

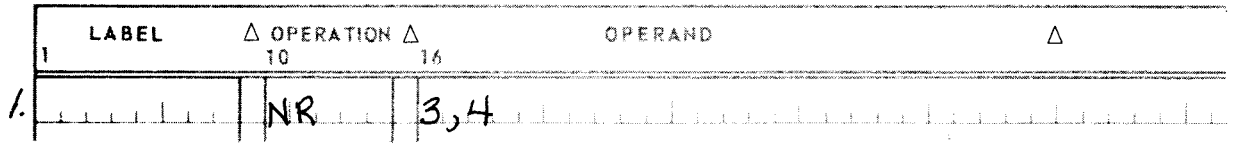
Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in both operations contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical product are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

- It is possible to clear selected bits in operand 1 by specifying 0's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:



1. The logical product of the contents of registers 3 and 4 is placed in register 3.

7.20. O (OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
O	$r_1,d_2(x_2,b_2)$	56	RX	Four Bytes

Function:

A logical addition (inclusive OR) operation is performed on the contents of operand 1, specified by r_1, and operand 2, specified by $d_2(x_2,b_2)$. The result is stored in operand 1.

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in either or both operands contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical addition are illustrated by the following truth table:

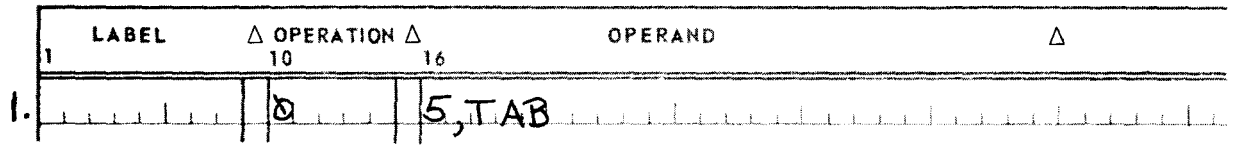
Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	1
1	0	1
1	1	1

- Selected bits of operand 1 can be set by specifying 1's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Example:



1. The logical sum of the contents of register 5 and main storage location TAB is placed in register 5.

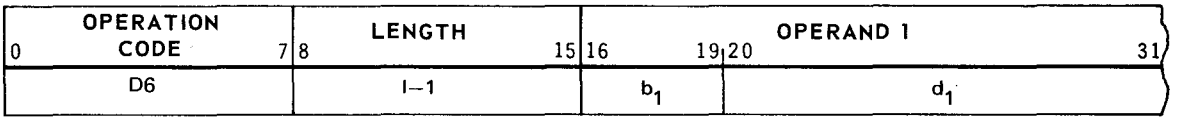
7.21. OC (OR)

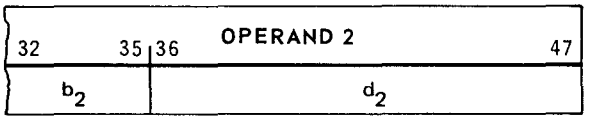
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
OC	$d_1(l,b_1),d_2(b_2)$	D6	SS	Six Bytes

Function:

A logical addition (inclusive OR) operation is performed on the contents of operand 1, specified by $d_1(l,b_1)$, and operand 2, specified by $d_2(b_2)$. The result is stored in operand 1.

Object Instruction Format:





Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in either or both operands contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical addition are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	1
1	0	1
1	1	1

- Selected bits of operand 1 can be set by specifying 1's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):

  – operand 1: RD, ID

  – operand 2: RO, IO

Example:

	LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
1.		OC	PATR,INCD

1. The contents of main storage locations PATR and INCD are logically added. The result is stored in PATR.

7.22. OI (OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
OI	$d_1(b_1),i_2$	96	SI	Four Bytes

Function:

A logical addition (inclusive OR) operation is performed on the contents of the operand 1 byte, specified by $d_1(b_1)$, and operand 2, contained in the i_2 field. The result is stored in operand 1.

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in either or both operands contain 1; otherwise, the result bit position is set to 0.

- The rules of operation for logical addition are illustrated by the following truth table:

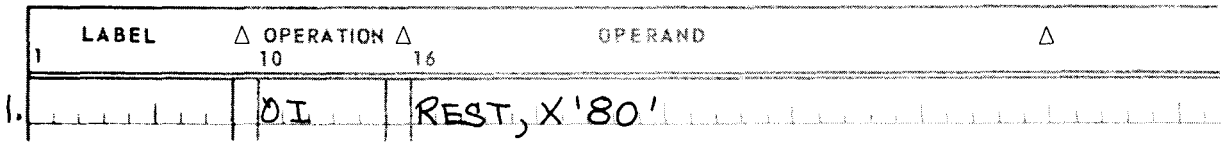
Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	1
1	0	1
1	1	1

- Selected bits of operand 1 can be set by specifying 1's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags:
  - operand 1: RD, ID
  - operand 2: none

Example:



1. Assume that REST addresses a byte in main storage containing the following bit configuration:

REST before execution	01111111
80	10000000
REST after execution	11111111

7.23. OR (OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
OR	r_1,r_2	16	RR	Two Bytes

Function:

A logical addition (inclusive OR) operation is performed on the contents of operand 1, specified by r_1, and operand 2, specified by r_2. The result is stored in operand 1.

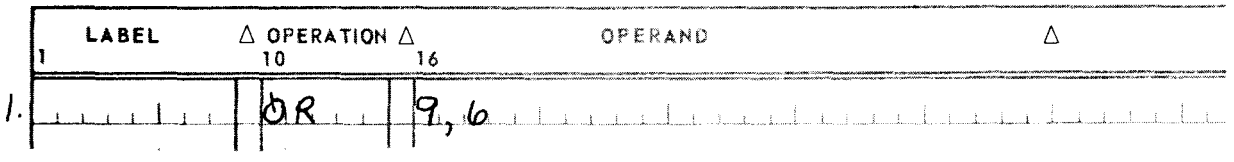
Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in either or both operands contain 1; otherwise, the result bit position is set to 0.
- The rules of operation for logical addition are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	1
1	0	1
1	1	1

- Selected bits of operand 1 can be set by specifying 1's in the corresponding bit positions of operand 2.
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:



1. The contents of registers 9 and 6 are logically added and the sum is stored in register 9.

7.24. SL (SUBTRACT-LOGICAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SL	$r_1,d_2(x_2,b_2)$	5F	RX	Four Bytes

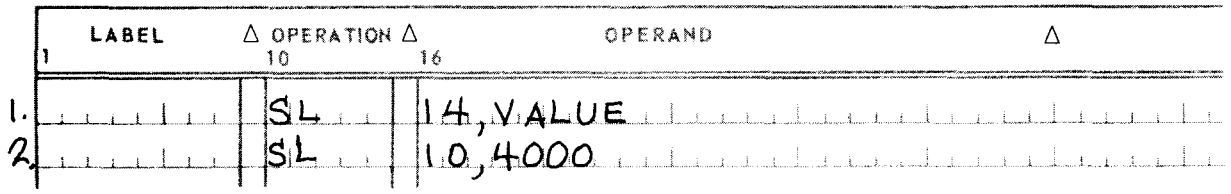
Function:

The full-word operand 2, specified by $d_2(x_2,b_2)$, is logically subtracted from the full-word operand 1, specified by r_1, and the result is placed in operand 1.

Operational Considerations:

- The subtraction is performed by adding the twos complement of operand 2 to operand 1.
- All 32 bits of both operands are used.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 1 (01_2) if result is not 0 (no carry out of most significant bit position);
  - to 2 (10_2) if result is 0 (carry out of most significant bit position);
  - to 3 (11_2) if result is not 0 (carry out of most significant bit position); or
  - code 0 is not used. A 0 difference cannot be obtained without a carry out of the most significant bit position.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (operand 2 or IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RO, IO

Examples:



1. The contents of the full word addressed by main storage location VALUE are converted to a twos complement binary value and added to the contents of register 14.

2. The contents of the full word located at main storage address 4000 are converted to a twos complement binary value and added to the contents of register 14.

7.25. SLDL (SHIFT-LEFT-DOUBLE-LOGICAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SLDL	$r_1,d_2(b_2)$	8D	RS	Four Bytes

Function:

The double-word operand 1, specified by r_1, is shifted left the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

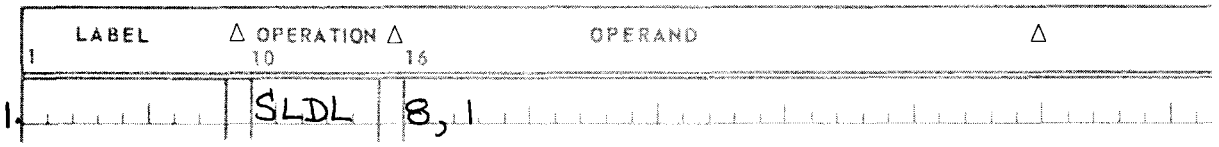
Object Instruction Format:

0 OPERATION CODE 7	8 OPERAND 1 11	12 OPERAND 3 15	16 19 20 OPERAND 2 31	
8D	r_1	unused	b_2	d_2

Operational Considerations:

- The r_1 specification in operand 1 must refer to the even-numbered register of an even-odd register pair.
- The vacated least significant bit positions of the register pair are zero filled.
- Bits shifted out of the even register are lost.
- The condition code remains unchanged.
- Possible program exceptions:
  - specification exception (operand 1 specifies an odd-numbered register)
- Relocation and indirection flags: none

Example:



1. The contents of registers 8 and 9, taken as a double word, are shifted to the left one bit position.

7.26. SLL (SHIFT-LEFT-SINGLE-LOGICAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SLL	$r_1,d_2(b_2)$	89	RS	Four Bytes

Function:

The full-word operand 1, specified by r_1, is shifted left the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

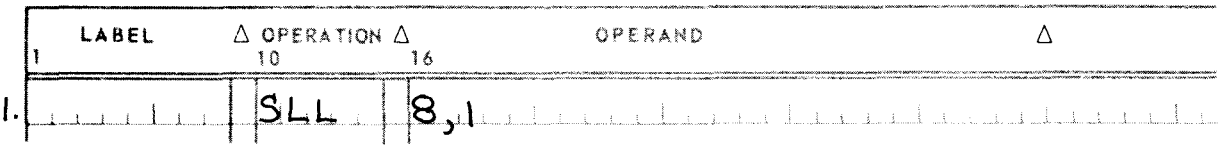
Object Instruction Format:

OPERATION CODE 0–7	OPERAND 1 8–11	OPERAND 3 12–15	OPERAND 2 16–19	OPERAND 2 20–31
89	r_1	unused	b_2	d_2

Operational Considerations:

- The vacated least significant bit positions of the register are zero filled.
- Bits shifted out of the register are lost.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:



1. The contents of register 8 are shifted to the left one bit position.

7.27. SLR (SUBTRACT-LOGICAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SLR	r_1,r_2	1F	RR	Two Bytes

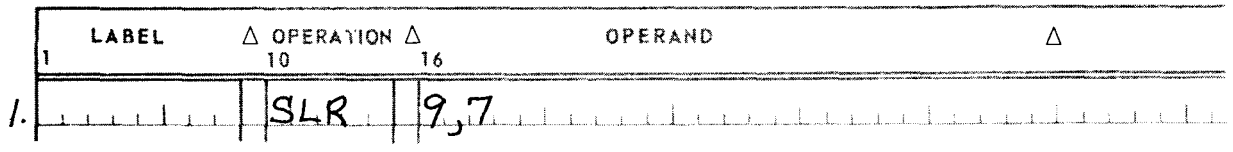
Function:

The full-word operand 2, specified by r_2, is logically subtracted from the full-word operand 1, specified by r_1, and the result is placed in operand 1.

Operational Considerations:

- The subtraction is performed by adding the twos complement of operand 2 to operand 1.
- All 32 bits of both operands are used.
- The contents of operand 2 remain unchanged.
- The condition code is set as follows:
  - to 1 (01_2) if result is not 0 (no carry out of most significant bit position);
  - to 2 (10_2) if result is 0 (carry out of most significant bit position);
  - to 3 (11_2) if result is not 0 (carry out of most significant bit position); or
  - code 0 is not used. A 0 difference cannot be obtained without a carry out of the most significant bit position.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:



1. The contents of register 7 are converted to a twos complement binary value and added to the contents of register 9.

7.28. SRDL (SHIFT-RIGHT-DOUBLE-LOGICAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SRDL	$r_1,d_2(b_2)$	8C	RS	Four Bytes

Function:

The double-word operand 1, specified by r_1, is shifted right the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

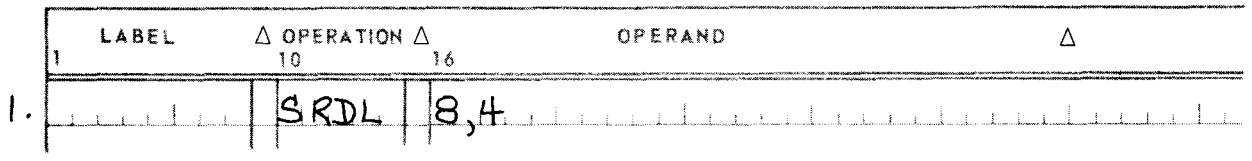
Object Instruction Format:

OPERATION CODE 0–7	OPERAND 1 8–11	OPERAND 3 12–15	OPERAND 2 16–19	OPERAND 2 20–31
8C	r_1	unused	b_2	d_2

Operational Considerations:

- The r_1 specification in operand 1 must refer to the even-numbered register of an even-odd register pair.
- The vacated most significant bit positions of the register pair are zero filled.
- Bits shifted out of the odd register are lost.
- The condition code remains unchanged.
- Possible program exceptions:
  - specification exception (operand 1 specifies an odd-numbered register)
- Relocation and indirection flags: none

Example:



1. The contents of register 8 and register 9, taken as a double word, are shifted to the right four bit positions.

7.29. SRL (SHIFT-RIGHT-SINGLE-LOGICAL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SRL	$r_1,d_2(b_2)$	88	RS	Four Bytes

Function:

The full-word operand 1, specified by r_1, is shifted right the number of bit positions specified by the least significant six bits of the operand 2 address, specified by $d_2(b_2)$.

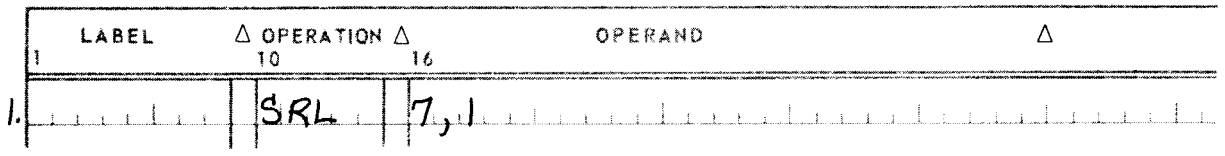
Object Instruction Format:

OPERATION CODE (0–7)	OPERAND 1 (8–11)	OPERAND 3 (12–15)	OPERAND 2 (16–19)	OPERAND 2 (20–31)
88	r_1	unused	b_2	d_2

Operational Considerations:

- The vacated most significant bit positions of the register are zero filled.
- Bits shifted out of the register are lost.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags 90/60,70: none

Example:



1. The contents of register 7 are shifted to the right one bit position.

7.30. STC (STORE-CHARACTER)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
STC	$r_1,d_2(x_2,b_2)$	42	RX	Four Bytes

Function:

The least significant eight bits of operand 1, specified by r_1, are stored in the storage location operand 2, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- The contents of operand 1 remain unchanged.
- The condition code remains unchanged.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RD, ID

Example:

LABEL	OPERATION	OPERAND
1.	STC	9,STAR

1. The rightmost eight bits of register 9 are stored in main storage location STAR.

7.31. TM (TEST-UNDER-MASK)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
TM	$d_1(b_1),i_2$	91	SI	Four Bytes

Function:

The main storage byte operand 1, specified by $d_1(b_1)$, is tested for the presence of 1 bits according to the 8-bit mask operand 2, specified by i_2.

Operational Considerations:

- The 1 bits of the operand 2 mask are used to test the bits of operand 1.
- The contents of operand 1 remain unchanged.
- The condition code is set as follows:
  - to 0 (00_2) if all the 1 bits in the mask match 0 bits in the byte tested or if all the bits in the mask are 0;

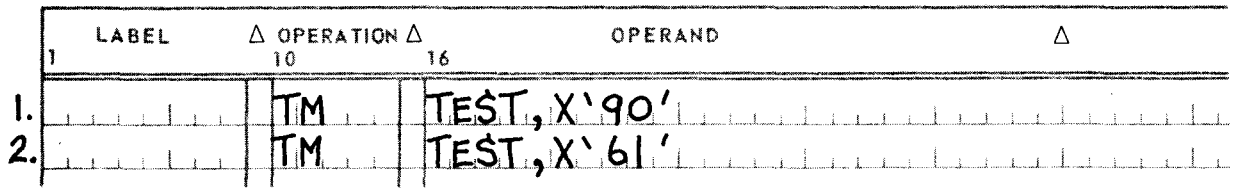
- to 1 (01_2) if some of the 1 bits in the mask match 0 bits in the byte tested;
- to 3 (11_2) if all the 1 bits in the mask correspond with 1 bits in the byte tested; or
- code 2 is not used.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RO, IO
  - operand 2: none

Examples:



1. Assume that main storage location TEST contains the following value:

TEST	10010000
90	10010000

   Condition code 3 is set.

2. Assume that main storage location TEST contains the following value:

TEST	10010000
61	01100001

   Condition code 0 is set.

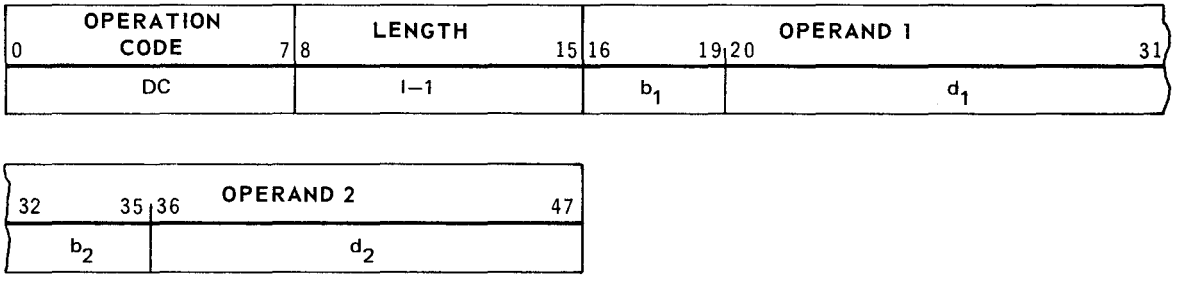
7.32. TR (TRANSLATE)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
TR	$d_1(l,b_1),d_2(b_2)$	DC	SS	Six Bytes

Function:

Data stored in operand 1, specified by $d_1(l,b_1)$, is translated according to a table stored in the operand 2 location, specified by $d_2(b_2)$.

Object Instruction Format:



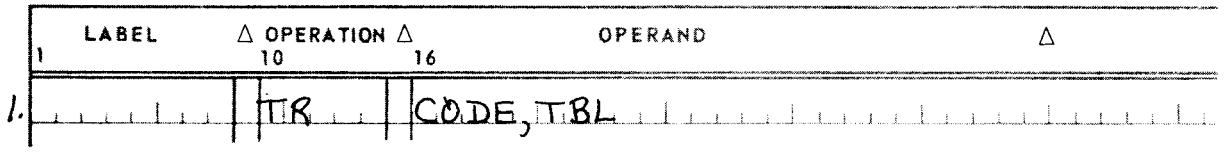
Operational Considerations:

- The 8-bit code of each character of operand 1 is added to the base table address specified by operand 2 to obtain the address of the character which is to replace the original character of operand 1.
- Translation continues until all characters specified by the length (l) have been translated.
- The contents of the table are not changed unless overlap occurs.
- The condition code remains unchanged.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. Assume TBL specifies the leftmost translate table address, 5000_{16}. The table length is 256 bytes. All values given are hexadecimal.

Location	5000	5001	....	5008	....	5080	....	50FF
Table contents	77	30		1F		01		A9

Assume CODE is a 3-byte area whose contents are:

Operand 1	CODE	CODE+1	CODE+2
Contents before execution	01	08	80
Contents after execution	30	1F	01

Translate Operation:

- The contents of main storage location CODE (01_{16}) are added to the base address assigned to TBL, which is 5000_{16}. The resulting address is 5001_{16}. The contents (30_{16}) at table location 5001_{16} are transferred to location CODE, replacing the original value 01_{16} with 30_{16}.
- CODE+1 originally contains 08_{16}. This is added to the base table address, which is 5000_{16}, to derive the address 5008_{16}. Location 5008_{16} in the table contains $1F_{16}$. This value is transferred to location CODE+1 to replace the original contents.
- CODE+2 originally contains the value 80_{16}. This is added to the base table address, which is 5000_{16}, to derive the address 5080_{16}. After the translation cycle, CODE+2 contains the value 01_{16}, obtained from the translate table.
- The programmer may place whatever values are required into the 256-byte translate table. When it is known what kind of bit configurations are expected as input (each unique configuration produces an address pointing to a unique table address), the desired value may be placed in the table to produce a translation.

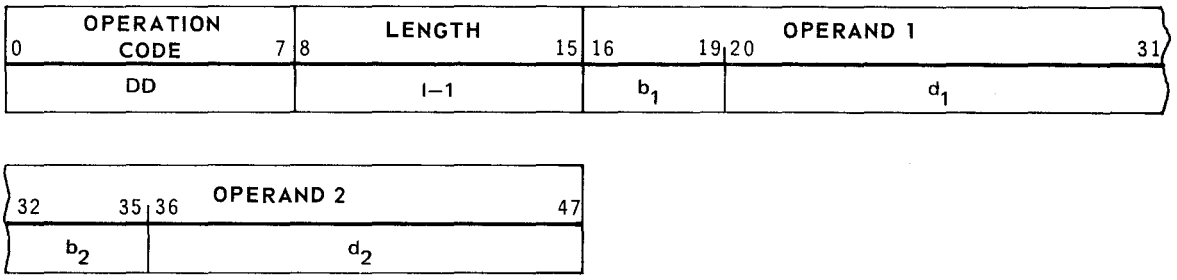
7.33. TRT (TRANSLATE-AND-TEST) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
TRT	$d_1(l,b_1),d_2(,b_2)$	DD	SS	Six Bytes

Function:

The data stored in operand 1, specified by d_1 (l,b_1), is translated according to a table stored in the location designated by operand 2, specified by $d_2(b_2)$, and the result is tested.

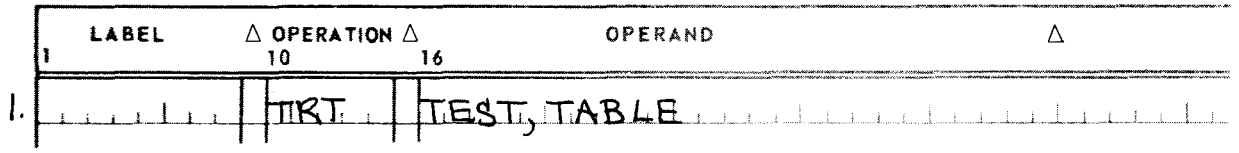
Object Instruction Format:



Operational Considerations:

- The translate function of this instruction proceeds in the same manner as the TR instruction.
- The selected byte (result byte) in the translate table is examined and tested for a pattern of all 0's. If the result byte is all 0's, it is ignored and the translate operation is continued. If the result byte is nonzero, the address of the corresponding operand 1 byte is stored in the least significant 24 bit positions of general register 1; the result byte is stored in the least significant eight bit positions of general register 2 and the operation is terminated.
- The contents of both operands remain unchanged.
- The address stored in general register 1 is a program relative address if the RD flag of the current relocation register is 1 or if the ID flag of the current relocation register and the R flag of the last associated IACW are 1. Otherwise, the address stored is an absolute address.
- The condition code is set as follows:
  - to 0 (00_2) if all result bytes are 0. In this case, registers 1 and 2 remain unchanged.
  - to 1 (01_2) if the result byte corresponding to any except the last operand 1 byte is nonzero;
  - to 2 (10_2) if the result byte corresponding to the last operand 1 byte is nonzero; or
  - code 3 is not used.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - protection exception
  - specification exception (IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. The contents of main storage location TEST are translated according to the table designated as TABLE. The result is tested for 0's and the appropriate data is stored in general registers 1 and 2. The contents of TEST are not altered by this operation.

7.34. X (EXCLUSIVE-OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
X	$r_1,d_2(X_2,b_2)$	57	RX	Four Bytes

Function:

A logical difference (exclusive OR) operation is performed on the contents of operand 1, specified by r_1, and the contents of operand 2, specified by d_2 (x_2b_2). The result is stored in operand 1.

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in the operands are unlike; otherwise, the bit position in the result is set to 0.
- The rules of operation for the exclusive OR operation are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

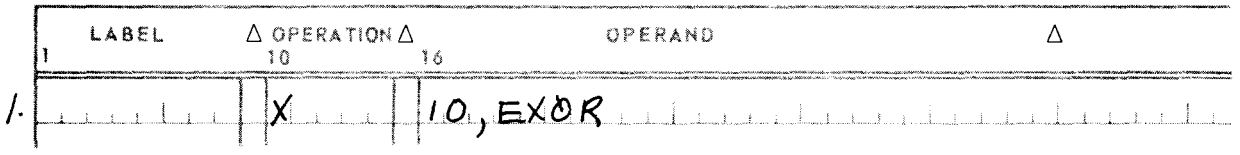
- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Specification
Indirect addressing	
Protection	
Specification (operand 2 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RO, IO

Example:



1. An exclusive OR operation is performed on the bits of register 10 and the bits in main storage location EXOR. The result replaces the contents of register 10.

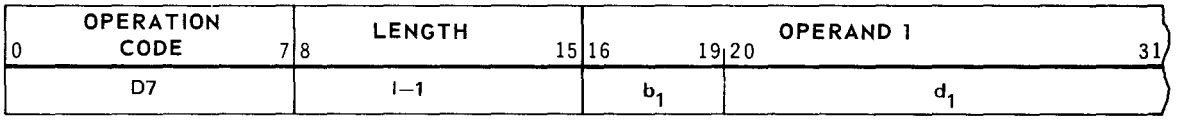
7.35. XC (EXLUSIVE-OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
XC	$d_1(l,b_1),d_2(b_2)$	D7	SS	Six Bytes

Function:

A logical difference (exclusive OR) operation is performed on the contents of operand 1, specified by d_1 (l,b_1), and the contents of operand 2, specified by d_2 (b_2). The result is stored in operand 1.

Object Instruction Format:



Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in the operands are unlike; otherwise, the bit position in the result is set to 0.
- The rules of operation for the exclusive OR operation are illustrated by the following truth table:

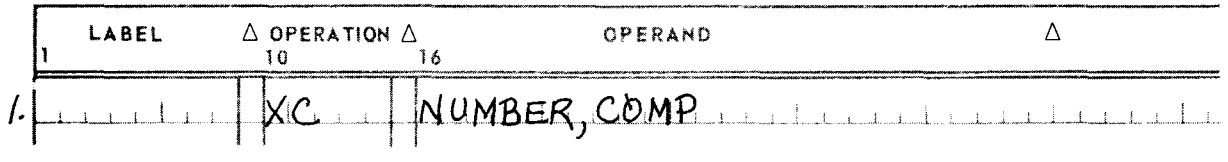
Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: RO, IO

Example:



1. An exclusive OR operation is performed on the contents of main storage locations NUMBER and COMP. The result is stored in main storage location NUMBER.

7.36. XI (EXCLUSIVE-OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
XI	$d_1(b_1),i_2$	97	SI	Four Bytes

Function:

A logical difference (exclusive OR) operation is performed on the operand 1 byte, specified by $d_1(b_1)$, and the operand 2 byte, contained in the i_2 field. The result replaces operand 1.

Operational Considerations:

- A bit position in the result is set to 1 if the corresponding bit positions in the operands are unlike; otherwise, the bit position in the result is set to 0.
- The rules of operation for the exclusive OR operation are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Addressing
Indirect address specification	Storage protection
Indirect addressing	
Protection	
Specification (IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RD, ID
  - operand 2: none

Example:

LABEL (1)	△ OPERATION △ (10)	(16) OPERAND	△
1.	XI	CNTRLS,X'0F'	

1. Assume that CNTRLS contains the following value:

CNTRLS before execution	01001100
0F	00001111
CNTRLS after execution	01000011

7.37. XR (EXCLUSIVE-OR)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
XR	r_1,r_2	17	RR	Two Bytes

Function:

A logical difference (exclusive OR) operation is performed on the contents of operand 1, specified by r_1, and the contents of operand 2, specified by r_2. The result is stored in operand 1.

Operational Considerations:

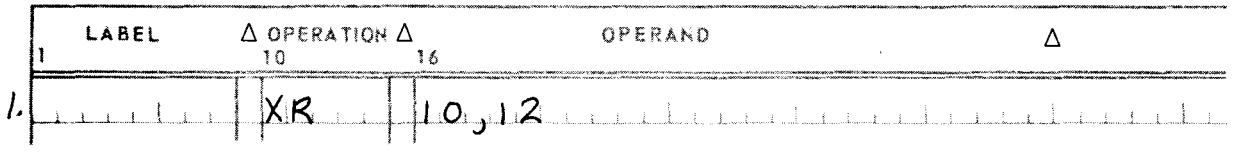
- A bit position in the result is set to 1 if the corresponding bit positions in the operands are unlike; otherwise, the bit position in the result is set to 0.

- The rules of operation for the exclusive OR operation are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

- The condition code is set as follows:
  - to 0 (00_2) if result is 0;
  - to 1 (01_2) if result is not 0; or
  - codes 2 and 3 are not used.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:



1. An exclusive OR operation is performed on the contents of registers 10 and 12. The result replaces the contents of register 10.

8. Branching Instructions

8.1. GENERAL

The branching instruction set provides the program-controlled capability of altering the normally sequential execution of instructions. Branching instructions provide a means for making a choice, for jumping to or from a subroutine, or for repeating a segment of coding.

Each branching instruction specifies the address of the instruction to be executed if the conditions specified by the instruction are met. In all the branching instructions, the second operand address is used as the branch address. A branch in the sequence of instructions is performed by loading the branch address into the instruction address field (bits 40 through 63) of the current program status word (PSW). However, since the contents of the instruction address field of the current PSW must always specify an absolute main storage address, the branch address may be converted from relative to absolute before the instruction address field is loaded.

On the SPERRY UNIVAC 90/60,70 Systems, whether a specified branch address is to be relocated, that is, whether it is relative or absolute, depends on the state of the RI flag (bit 3) in the current relocation register. Except for the branch-and-link-external (BALE) and execute (EX) 90/60,70 instructions, branch addresses are never indirect addresses.

Arithmetic, logical, and input/output instructions set a 2-bit condition code to one of four states: 0, 1, 2, and 3. The condition code reflects conditions such as 0, low, high, or overflow results, and equal, low, or high comparisons of two operands. The condition code remains unchanged until modified by a subsequent instruction. The branch-on-condition instruction inspects this code and uses the setting as the criterion for branching.

Extended mnemonic codes facilitate the use of branch-on-condition instructions. See 8.2 for a description of extended mnemonics.

This section describes the operation of each branching instruction. The instructions are in alphabetical order according to mnemonic operation code. Each description includes a list of the possible program exceptions and condition codes which may result. The execution of a branch instruction is considered to extend to the point where the absolute address is loaded into the instruction address field of the current PSW. Only those program exceptions that may occur before that point are listed. Thus, even though a specified branch address may be invalid (for example, reference to an out-of-bounds location) the resulting program exception is detected only when the branch address is actually used to access main storage. However, since the branch instruction is considered to have been completed by that point, the program exception (address exception) is not listed in the description of the branch instruction. Furthermore, if an interrupt occurs due to the program exception, the instruction address field of the program exception old PSW reflects the instruction at the branch address and not the branch instruction itself.

The pertinent relocation and indirection flags are listed for each instruction on 90/60,70 systems. The object instruction format is shown only for those instructions which differ from the format shown in Figure 3–1. See Table 3–1 for an explanation of the abbreviations used in describing instruction formats.

8.2. EXTENDED MNEMONIC CODES

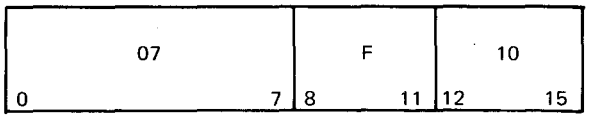
Extended mnemonic codes are provided in assembly language as abbreviated nototations for writing branch-on-condition instructions. Table 8–1 lists the extended mnemonic codes and their meanings. These codes represent the branch-on-condition instruction with different condition code settings in the m_1 field (8.6 and 8.7).

Table 8–1. Extended Mnemonic Codes

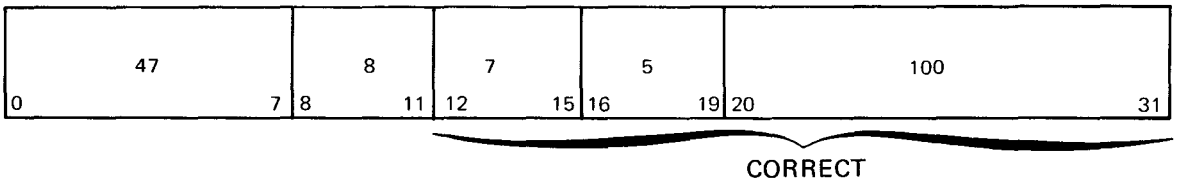
RR Type Instructions		RX Type Instructions		Function
Mnemonic	Hexadecimal Operation Code/m_1	Mnemonic Code	Hexadecimal Operation Code/m_1	
BR	07 F	B	47 F	Branch
NOPR	07 0	NOP	47 0	No operation
Used After Comparison Instructions				
BHR	07 2	BH	47 2	Branch if high
BLR	07 4	BL	47 4	Branch if low
BER	07 8	BE	47 8	Branch if equal
BNHR	07 D	BNH	47 D	Branch if not high
BNLR	07 B	BNL	47 B	Branch if not low
BNER	07 7	BNE	47 7	Branch if not equal
Used After Test Under Mask Instructions				
BOR	07 1	BO	47 1	Branch if all ones
BZR	07 8	BZ	47 8	Branch if all zeros
BMR	07 4	BM	47 4	Branch if mixed
BNOR	07 E	BNO	47 E	Branch if not all ones
BNZR	07 7	BNZ	47 7	Branch if not all zeros
BNMR	07 B	BNM	47 B	Branch if not mixed
Used After Arithmetic Instructions				
BOR	07 1	BO	47 1	Branch if overflow
BZR	07 8	BZ	47 8	Branch if zero
BMR	07 4	BM	47 4	Branch if minus
BPR	07 2	BP	47 2	Branch if positive
BNOR	07 E	BNO	47 E	Branch if no overflow
BNZR	07 7	BNZ	47 7	Branch if not zero
BNMR	07 B	BNM	47 B	Branch if not minus
BNPR	07 D	BNP	47 D	Branch if not positive

Examples:

Source code BR 10 produces an unconditional branch to the address contained in register 10. The object instruction format is:



Source code BZ CORRECT produces a conditional branch if the condition code is 0 to the address specified by CORRECT. The object instruction format is:



NOTE:

The value of the index regiater is implied to be 0 if not specifically stated in source code. If it is not equal to 0, it must be coded.

8.3. BAL (BRANCH-AND-LINK)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BAL	$r_1,d_2(x_2,b_2)$	45	RX	Four Bytes

Function:

The current PSW instruction length code, program mask, and instruction address field (bits 32 to 63) are stored in operand 1, specified by r_1, and the address of operand 2, specified by $d_2(x_2,b_2)$, is stored in the current PSW instruction address field.

Operational Considerations:

- Operand 2 is the address branched to by the program.
- The return address is preserved in operand 1.
- The branch address is determined before the return address is stored. This allows correct operation if the r_1 and x_2 or b_2 registers are the same.
- The condition code remains unchanged.
- Possible program exceptions: none

- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RI
- 90/60,70 systems – If RI (bit 3 of the current relocation register) is 1, the offset value contained in the current relocation register is subtracted from the updated instruction address to form the return address. If RI is 0, the updated instruction address is the return address and is not modified.

Example:

```
   LABEL      Δ OPERATION Δ        OPERAND                          Δ
   1            10      16
1.            BAL      10,SUBRTN
   RETURN     AR       6,4
              .
              .
              .

   SUBRTN     AH       6,UNOR
              .
              .
              BCR      15,0
```

1. The address of main storage location RETURN is stored in register 10 and branch is made to the address specified by SUBRTN. After SUBRTN execution, branch is made to the address stored in register 10 (RETURN).

8.4. BALE (BRANCH-AND-LINK-EXTERNAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BALE	$r_1,d_2(x_2,b_2)$	4D	RX	Four Bytes

Function:

The current relocation and indirection flags and the updated instruction address are stored in operand 1, specified by r_1 (link register). The contents of operand 2, specified by $d_2(x_2,b_2)$, are used to compute the branch address.

Operational Considerations:

- Bits 0 through 7 (relocation and indirection flags) of the current relocation register replace bits 0 through 7 of operand 1. If RI (bit 3 of the current relocation register) is 1, the updated instruction address is converted to relative by subtracting the offset value contained in the current relocation register. If RI is 0, the updated instruction address remains unmodified. The updated instruction address is then placed in bit positions 8 through 31 of the operand 1 location (link register).

- The d_2 field and the contents of the base register specified by b_2 are added. The sum specifies the address (relative if RI, bit 3 of the current relocation register, equals 1) of an IACW. The address of the final IACW is computed according to the rules of indirect and relative addressing. Bits 8 through 31 of the final IACW plus the contents of the register specified by x_2 specify the branch address (realtive if R, bit 6 of the final IACW, is 1). If R is 1, the branch address is converted to absolute by adding the offset value contained in the current relocation register. If R is 0, the branch address is considered to be an absolute address and is not further modified. The absolute branch address then replaces the instruction address field of the current PSW. The RI flags of both the current relocation register and the applicable relocation register in main storage are replaced by R.

- The branch address is computed before the link register is loaded. This allows for correct execution of the branch if the register specified by r_1 is the same as that specified by x_2 or b_2.

- If the b_2 designator in the instruction contains the value 0, the branch is not accomplished but the link information is stored in the operand 1 location.

- The condition code remains unchanged.

- Possible program exceptions:

  – indirect address specification exception

  – indirect addressing exception

  – operation exception (if current PSW specifies IBM native mode)

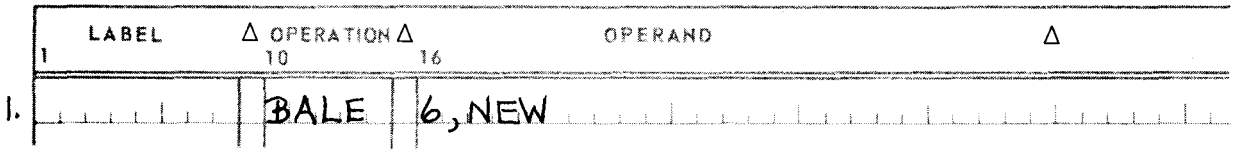
  – specification exception (IACW not on full-word boundary)

- Relocation and indirection flags:

  – operand 1: none

  – operand 2: RI

Example:



1. The current instruction address is stored in register 6 and a branch is made to the main storage address contained in the location specified by NEW.

8.5. BALR (BRANCH-AND-LINK)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BALR	r_1,r_2	05	RR	Two Bytes

Function:

The current PSW instruction length code, program mask, and instruction address fields (bits 32 to 63) are stored in operand 1, specified by r_1, and the address of operand 2, specified by r_2, is stored in the current PSW instruction address field.

Operational Considerations:

- Operand 2 is the address branched to by the program.
- The return address is preserved in operand 1.
- If RI (bit 3 of the current relocation register) is 1, the offset value contained in the current relocation register is subtracted from the updated instruction address to form the link address. If RI is 0, the updated instruction address is the link address and is not modified.
- The branch address is determined before the operand 2 address is stored. This allows correct operation if the r_1 and r_2 registers are the same.
- If the operand 2 register is 0, the link information is stored in the operand 1 location, but no branch is accomplished. Instruction sequencing continues with the updated instruction address.
- The condition code remains unchanged.
- 90/60,70 systems – If RI (bit 3 of the current relocation register) is 1, the offset register is subtracted from the updated instruction address to form the link address. If RI is 0, the updated instruction address is the link address and is not modified.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RI

Example:

```
   LABEL      ΔOPERATIONΔ              OPERAND                 Δ
1             10        16
1.            LA        7,SUBRTN
              BALR      6,7
   CLAR       LR        2,3
                :
   SUBRTN     XR        2,2
                :
              BCR       15,6
```

1. The return address is stored in register 6 and a branch is made to the address in register 7. After SUBRTN execution, an unconditional branch is made to the address in register 6 (CLAR).

8.6. BC (BRANCH-ON-CONDITION)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BC	$m_1,d_2(x_2,b_2)$	47	RX	Four Bytes

Function:

The operand 1 mask, specified by m_1, is compared with the current condition code. If equal, the instruction at the address specified by operand 2, specified by $d_2(x_2,b_2)$, is executed; otherwise, the next instruction in sequence is executed.

Object Instruction Format:

OPERATION CODE 0–7	OPERAND 1 8–11	OPERAND 2 12–15	OPERAND 2 16–19	OPERAND 2 20–31
47	m_1	x_2	b_2	d_2

Operational Considerations:

- The mask, considered operand 1, occupies bits 8, 9, 10, and 11 of the object instruction. The mask specification determines the condition code setting to be tested, as follows:
  - An 8 produces the mask 1000_2 which tests bit 8 for a 0 condition code.
  - A 4 produces the mask 0100_2 which tests bit 9 for a 1 condition code.
  - A 2 produces the mask 0010_2 which tests bit 10 for a 2 condition code.
  - A 1 produces the mask 0001_2 which tests bit 11 for a 3 condition code.
  - A 0 produces the mask 0000_2 which is equivalent to no operation.
  - Any combination of 1's and 0's in the mask tests for more than one condition code.
  - Any 1 bit on and tested produces the branch.
- A mask specification of 15 (1111_2) produces an unconditional branch.
- See 8.2 for a list of extended mnemonic codes which may be used for branch-on-condition instructions.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RI

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		BC	8,ENDPROG
2.		BC	12,JAX
3.		BC	15,BEGIN

1. If the condition code is set to 0, a branch is made to ENDPROG.

2. If the condition code is set to 0 or 1, a branch is made to JAX.

3. An unconditional branch is made to BEGIN.

8.7. BCR (BRANCH-ON-CONDITION)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BCR	m_1,r_2	07	RR	Two Bytes

Function:

The operand 1 mask, specified by m_1, is compared with the current condition code. If equal, the instruction at the address stored in operand 2, specified by r_2, is executed; otherwise, the next instruction in sequence is executed.

Object Instruction Format:

OPERATION CODE (0–7)	OPERAND 1 (8–11)	OPERAND 2 (12–15)
07	m_1	r_2

Operational Considerations:

- The mask, considered operand 1, occupies bits 8, 9, 10, and 11 of the object instruction. The mask specification determines the condition code setting to be tested, as follows:

  – An 8 produces the mask 1000_2 which tests bit 8 for a 0 condition code.

  – A 4 produces the mask 0100_2 which tests bit 9 for a 1 condition code.

  – A 2 produces the mask 0010_2 which tests bit 10 for a 2 condition code.

  – A 1 produces the mask 0001_2 which tests bit 11 for a 3 condition code.

  – A 0 produces the mask 0000_2 which is equivalent to no operation.

- Any combination of 1's and 0's in the mask tests for more than one condition code.
- Any 1 bit on the tested produces the branch.

- A mask specification of 15 (1111_2) produces an unconditional branch.
- If operand 2 is register 0, the instruction is equivalent to no operation.
- See 8.2 for a list of extended mnemonic codes which may be used for branch-on-condition instructions.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RI

Examples:

```
      LABEL   Δ OPERATION Δ   OPERAND   Δ
      1       10          16
1.            BCR         8,10
2.            BCR         15,8
```

1. If the condition code is set to 0, branch is made to the address stored in register 10.
2. An unconditional branch is made to the address stored in register 8.

8.8. BCRE (BRANCH-ON-CONDITION-TO-RETURN-EXTERNAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BCRE	m_1,r_2	OC	RR	Two Bytes

Function:

The operand 1 mask, specified by m_1, is compared with the current condition code. If equal, and depending on the relocation and indirection flags, the instruction at the address stored in operand 2, specified by r_2, is executed; otherwise, the next instruction in sequence is executed.

Object Instruction Format:

OPERATION CODE (0–7)	OPERAND 1 (8–11)	OPERAND 2 (12–15)
OC	m_1	r_2

Operational Considerations:

- If the operand 1 mask equals 0, bits 0 through 7 of operand 2 are placed in bit positions 0 through 7 of the current relocation register in main storage. If the new RI flag (bit 3 of the current relocation register) is 1 and the old RI flag had been 0, the offset value contained in the current relocation register is added to the updated instruction address to form the next instruction address. If the new RI flag is 0, the updated instruction address remains unchanged.

- If a bit in the operand 1 mask corresponds to the current condition code setting, bits 0 through 7 of operand 2 are placed in bit positions 0 through 7 of the current relocation register and the applicable relocation register in main storage. If the new RI flag is 1, the offset value contained in the current relocation register is added to the address contained in the register specified by r_2 to form the branch address. If RI is 0, the branch address is the address contained in the operand 2 register specified by r_2. The branch address replaces the instruction address field of the current program status word.

- If the operand 1 mask does not equal 0 and a bit in the mask does not correspond to the current condition code setting, no operation takes place.

- Except for the no-operation case, the following constraint applies: If the current relocation register flags indicate relative instruction (RI=1), the bits in the operand 2 register corresponding to relative instruction, origin, and destination (RI, RO, RD) must all be set to 1. If this requirement is not met, a specification exception is generated and no operation takes place.

- The mask, considered operand 1, occupies bits 8, 9, 10, and 11 of the object instruction. The mask specification determines the condition code setting to be tested, as follows:

  - An 8 produces the mask 1000_2 which tests bit 8 for a 0 condition code.
  - A 4 produces the mask 0100_2 which tests bit 9 for a 1 condition code.
  - A 2 produces the mask 0010_2 which tests bit 10 for a 2 condition code.
  - A 1 produces the mask 0001_2 which tests bit 11 for a 3 condition code.
  - A 0 produces the mask 0000_2 which is equivalent to no operation.
  - Any combination of 1's and 0's in the mask tests for more than one condition code.
  - Any 1 bit on and tested produces the branch.

- A mask specification of 15 (1111_2) produces an unconditional branch.

- The condition code remains unchanged.

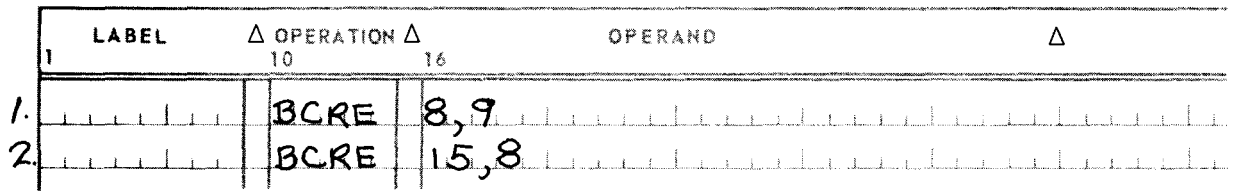
- Possible program exceptions:

  - operation exception (if current PSW specifies IBM native mode)
  - specification exception (current relocation flags indicate relative instruction and RI, RO, and RD of operand 2 are not all equal to 1)

- Relocation and indirection flags:

  - operand 1: none
  - operand 2: RI

Examples:



1. If the condition code is 0, the contents of register 9 are used to form the branch address.

2. An unconditional branch is made to the address determined by the contents of register 8.

8.9. BCT (BRANCH-ON-COUNT)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BCT	$r_1,d_2(x_2,b_2)$	46	RX	Four Bytes

Function:

The contents of operand 1, specified by r_1, are reduced by 1. If the result is 0, instruction sequencing continues. If the result is nonzero, a branch is made to the operand 2 address, specified by $d_2(x_2,b_2)$.

Operational Considerations:

- The BCT instruction proceeds as follows:

  – The count value is loaded into the operand 1 register, r_1 by a prior instruction.

  – The operand 1 register is decremented by 1 each time the BCT instruction is executed.

  – The test for 0 is made after each count.

  – If 0, the next instruction is executed.

  – If not 0, the branch is made to the address specified as operand 2.

- If the operand 1 register is initially 0, the count is decremented through 0 and is treated as an unsigned positive number with a magnitude of 2^{32}.

- The branch address is determined prior to the counting operation. This allows correct operation if the r_1 and b_2 or x_2 registers are the same.

- The condition code remains unchanged.

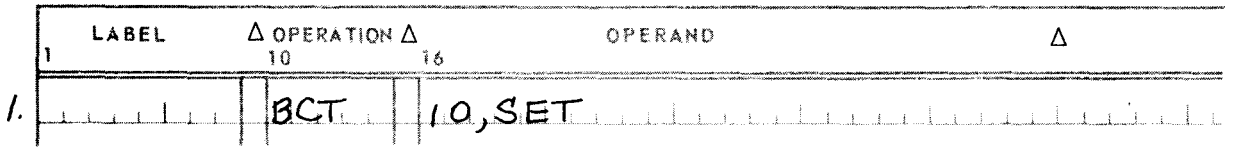
- Possible program exceptions: none

- Relocation and indirection flags:

  – operand 1: none

  – operand 2: RI

Example:



1. The count in register 10 is decremented and tested. If it is not equal to 0, branch is made to the address specified by SET. If it equals 0, the next sequential instruction is executed.

8.10. BCTR (BRANCH-ON-COUNT)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BCTR	r_1,r_2	06	RR	Two Bytes

Function:

The contents of operand 1, specified by r_1, are reduced by 1. If the result is 0, instruction sequencing continues. If the result is not equal to 0, a branch is made to the address stored in operand 2, specified by r_2.

Operational Considerations:

- The BCTR instruction proceeds as follows:
  - The count value is loaded into the operand 1 register, r_1 by a prior instruction.
  - r_1 is decremented by 1 each time the BCTR instruction is executed.
  - The test for 0 is made after each count.
  - If 0, the next instruction is executed.
  - If not 0, the branch is made to the address stored in the operand 2 register.
- If the operand 1 register is initially 0, the count is decremented through 0 and is treated as an unsigned positive number with a magnitude of 2^{32}.
- If the operand 2 register field is 0, counting is performed without branching.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70):
  - operand 1: none
  - operand 2: RI

Example:

	LABEL	Δ OPERATION Δ	OPERAND
1.		BCTR	10,12

1. Decrement and test the count in register 10. If it is not equal to 0, branch is made to the address stored in register 12. If it equals 0, the next sequential instruction is executed.

8.11. BXH (BRANCH-ON-INDEX-HIGH) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BXH	$r_1,r_3,d_2(b_2)$	86	RS	Four Bytes

Function:

The sum of the contents of operand 1 and operand 3, specified by r_1 and r_3, respectively, is algebraically compared to a comparison value. If the sum is greater than the comparison value, a branch is made to the operand 2 address, specified by $d_2(b_2)$; otherwise, sequential instruction execution proceeds.

Operational Considerations:

- The comparison value is contained in an odd-numbered register. The register is r_3, if r_3 is odd, or r_3+1, if r_3 is even.
- Following the comparison, the sum is placed in the operand 1 register.
- All quantities are treated as signed integers.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags:

  – operand 1: none

  – operand 2: RI

Examples:

	LABEL	Δ OPERATION Δ	OPERAND
1.		BXH	4,5,HAR
2.		BXH	6,8,HIGH

1. The contents of registers 4 and 5 are added and the sum is placed in register 4. If the new contents of register 4 are greater than the contents of register 5, a branch is made to main storage location HAR. Otherwise, instruction execution proceeds sequentially.

2. The contents of registers 6 and 8 are added and the sum is placed in register 6. If the new contents of register 6 are greater than the contents of register 9, a branch is made to main storage location HIGH. Otherwise, instruction execution proceeds sequentially.

8.12. BXLE (BRANCH-ON-INDEX-LOW-OR-EQUAL) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
BXLE	$r_1,r_3,d_2(b_2)$	87	RS	Four Bytes

Function:

The sum of the contents of operand 1 and operand 3, specified by r_1 and r_3, respectively, is algebraically compared to a comparison value. If the sum is less than or equal to the comparison value, a branch is made to the operand 2 address specified by $d_2(b_2)$; otherwise, sequential instruction execution proceeds.

Operational Considerations:

- The comparison value is contained in an odd-numbered register. The register is r_3, if r_3 is odd, or r_3+1, if r_3 is even.
- Following the comparison, the sum is placed in the operand 1 register.
- All quantities are treated as signed integers.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags:
  - operand 1: none
  - operand 2: RI

Examples:

```
      LABEL   Δ OPERATION Δ        OPERAND          Δ
      1       10          16
1.            BXLE        6,7,LOW
2.            BXLE        5,6,BRANCH
```

1. The contents of registers 6 and 7 are added and the sum is placed in register 6. If the new contents of register 6 are less than or equal to the contents of register 7, a branch is made to the address specified by LOW. Otherwise, instruction execution proceeds sequentially.

2. The contents of registers 5 and 6 are added and the sum is placed in register 5. If the new contents of register 5 are less than or equal to the contents of register 7, a branch is made to the address specified by BRANCH. Otherwise, instruction execution proceeds sequentially.

8.13. EX (EXECUTE) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
EX	$r_1,d_2(x_2,b_2)$	44	RX	Four Bytes

Function:

If the contents of operand 1, specified by r_1, are 0, the instruction at the operand 2 address, specified by $d_2(x_2,b_2)$, is executed without modification. If operand 1 is not 0, the contents are used to modify the operand 2 instruction before that instruction is executed.

Operational Considerations:

- The address specified by operand 2 must be on a half-word boundary.
- When operand 1 is not 0, modification of the operand 2 instruction proceeds as follows: A logical addition (OR) is performed on the contents of bits 24 through 31 of operand 1 and bits 8 through 15 of the instruction at the operand 2 address. The result replaces bits 8 through 15 of the operand 2 instruction. The rules of operation for logical addition are illustrated by the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	1
1	0	1
1	1	1

- Modification of the operand 2 instruction affects only the execution of the instruction and does not alter the contents stored at the operand 2 location.
- The modified instruction is executed as if it were in the normal instruction sequence except that the instruction length code and updated instruction address fields of the current PSW reflect the EX instruction.
- Normally, instruction sequencing continues with the instruction following the EX instruction. However, if the instruction at the operand 2 address is a successful branch instruction, the instruction address field of the current PSW is replaced by the branch address and instruction sequencing continues with the instruction located at the branch address. If the operand 2 instruction is branch-and-link or branch-and-link-external, the instruction address stored in the link register is that of the instruction following the EX instruction.

- If an interrupt occurs after the completion of the subject instruction, the old PSW contains the address of the instruction following the EX instruction or the branch address.
- The condition code may be set by the instruction at the operand 2 address.
- Possible program exceptions:

  NOTE:

  A program exception condition can be caused by the EX instruction or the instruction specified in the EX instruction.

  – addressing exception

  – execute exception

  – indirect address specification exception

  – indirect addressing exception

  – protection exception

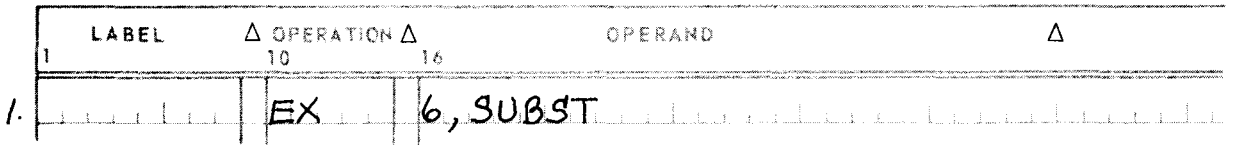
  – specification exception (IACW not on full-word boundary; or the address specified by operand 2 is odd)

- Relocation and indirection flags:

  – operand 1: none

  – operand 2: RO, IO

Example:



1. The instruction located at the address specified by SUBST is executed after modification according to the contents of register 6. After the execution of the instruction at main storage location SUBST, instruction sequence continues with the instruction following the EX instruction.

9. Status Switching Instructions

9.1. GENERAL

The status switching instruction set provides the capacility of altering the operating characteristics of the SPERRY UNIVAC Processor. Status switching instructions may be used to replace part or all of the current program status word (PSW) or to alter the contents of the SPERRY UNIVAC 90/60,70 protect key main storage. Certain of these instructions provide maintenance functions.

Status switching instructions are available in the RR, RS, and SI formats. As such, the operands may be contained in the general registers, main storage, or within the instruction itself. The address of an operand in main storage may be specified as relative or absolute and direct or indirect under the control of the applicable relocation register flags.

This section describes the operation of each status switching instruction. The instructions are arranged in alphabetical order according to mnemonic operation code. Each description includes a list of the possible program exceptions and condition codes which may result. In addition, the 90/60,70 relocation and indirection flags pertinent to the operand addresses and, in the case of the LPSW instruction, to the operand itself are listed.

The object instruction format is shown only for those instructions which differ from the format shown in Figure 3–1. For an explanation of the abbreviations used in describing instruction formats, see Table 3–1.

9.2. DIAG (DIAGNOSE) – PRIVILEGED INSTRUCTION – 90/60,70

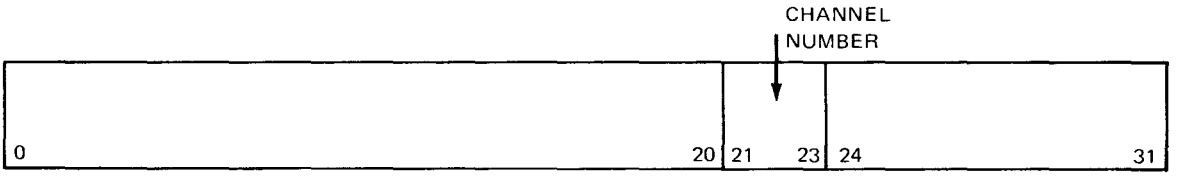
Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
DIAG	$d_1(b_1),i_2$	83	SI	Four Bytes

Function:

The DIAG instruction is used to control diagnostic operation:

- Channel Tester

  When the i_2 field specifies a value of 80_{16} or 81_{16}, the DIAG instruction relates to the channel tester. The contents of the 32-bit base register specified by b_1 are added to the contents of the d_1 field to form a 32-bit field with the following format:



If the i_2 field specifies 80_{16}, the channel specified by bits 21 through 23 is disconnected from the I/O interface and connected to the channel tester.

If the i_2 field specifies 81_{16}, the channel specified by bits 21 through 23 is disconnected from the channel tester and reconnected to the I/O interface.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The condition code remains unchanged.
- Possible program exceptions:
  - privileged operation exception
- Relocation and indirection flags: none

9.3. HPR (HALT-AND-PROCEED) – PRIVILEGED INSTRUCTION

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
HPR	$d_1(b_1),i_2$	99	SI	Four Bytes

Function:

The processor is halted without loss of data and the contents of the operands may be displayed.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- 9400/9480 systems – The processor halts and the storage address formed by $d_1 + (b_1)$ is placed into the internal data address register, A. The contents of this register can be displayed on the maintenance panel. The next instruction in the program is executed when the operator presses the RUN switch. The i_2 field of this instruction is ignored. A diagnostic error is issued whenever i_2 is specified.
- 90/60,70 systems – The operand 1 address is placed in bit positions 8 to 31 of the storage address register. Operand 2, contained in the i_2 field, if present, is placed in bit positions 8 to 15 of the operation code register. The processor halts and the HPR indicator is activated on the system maintenance panel. The registers may then be selected and displayed on the system maintenance panel. The processor remains halted until the RUN switch is pressed.
- The condition code remains unchanged.

- Possible program exceptions:
  - privileged operation exception
- Relocation and indirection flags (90/60,70): none

Examples:

```
      LABEL   Δ OPERATION Δ   OPERAND                Δ
      1           10          16
1.                HPR       X'F1'(0)
2.    TAGA        HPR       0(5),X'81'
3.                HPR       250(0),TAGX
```

1. The processor is halted and the value 0000000011110001_2 is displayed. (Applicable only to 9400/9480 format.)

2. The processor is halted and the value of the address specified by 0 indexed by the contents of register 5 is displayed. On the 90/60,70, the hexadecimal value 81 is placed in the operation code register. (Format correct for 90/60,70.)

3. The processor is halted and the value 0000000011111010_2 is displayed. On the 90/60,70, the value of TAGX, which must have been previously equated to a value in the range 0 to 255, is placed in the operation code register. (Format correct for 90/60,70; an error code is generated in 9400/9480 system.)

9.4. ISK (INSERT-STORAGE-KEY) — PRIVILEGED INSTRUCTION — 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
ISK	r_1,r_2	09	RR	Two Bytes

Function:

The 5-bit storage key contained in the key memory location in operand 2, specified by r_2, is inserted into the operand 1 register, specified by r_1.

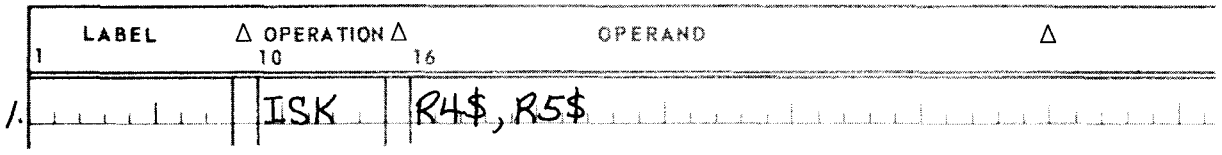
Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- Bits 8 through 20 of operand 2 designate the location from which the storage key is to be taken.
- The storage key is inserted into bits 24 through 28 of operand 1.
- Operand specifications are the same as for the SSK instruction (9.10).
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception (operand 2 specifies a nonexistent block of storage)

- privileged operation exception
- specification exception (bits 28 through 31 of operand 2 are not 0)

- Relocation and indirection flags: none

Example:



1. Presuming that R4$ and R5$ have been equated to 4 and 5, respectively, the storage key in the key memory location specified by bits 8 through 20 of register 5 is inserted in bit positions 24 through 28 of register 4.

9.5. LBR (LOAD-BASE-REGISTER) – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LBR	r	OB	RR	Two bytes

Function:

The offset value contained in the current relocation register is subtracted from the updated instruction address. The relative instruction address thus formed is placed in bit positions 8 through 31 of the register specified by r. Binary zeros will be placed in bit positions 0 through 7 of the register specified by r. The constant 00010101 is placed in bit positions 0 through 7 of the current relocation register and the applicable relocation register in main storage.

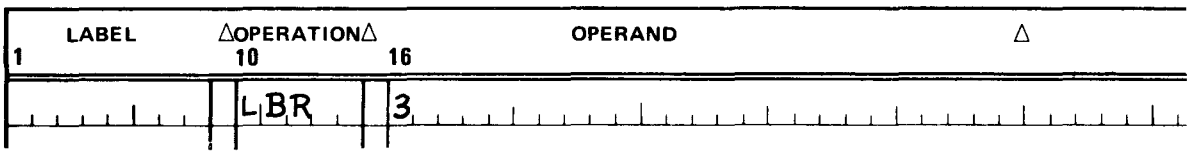
Object Instruction Format:

0 OPERATION CODE 7	8 OPERAND 1 11	12 OPERAND 2 15
OB	unused	r

Operational Considerations:

- If the r designator is zero, no operation takes place.
- The condition code remains unchanged.
- Possible program exceptions: none
- Relocation and indirection flags: none

Example:



9.6. LCS (LOAD-CONTROL-STORAGE) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LCS	$r_1,r_3,d_2(b_2)$	B1	RS	Four Bytes

Function:

The number of microinstructions specified by the r_1 field plus 1 are transferred from main storage, beginning with the microinstruction located at the operand 2 address, specified by $d_2(b_2)$, to the proper section of the control storage, beginning at the address specified by the contents of operand 3, specified by r_3.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The number of microinstructions to be transferred is computed as the operand 1 register specification plus 1.
- The register specified by operand 3 contains the address of the section of control storage to be loaded as follows:
  - If bit 16 of the operand 3 register is 0, bits 23 through 31 specify the location in the address calculator (AC) section of the control storage into which the first microinstruction is to be loaded. Bits 0 through 15 and 17 through 22 are ignored.
  - If bit 16 of the operand 3 register is 1, bits 21 through 31 specify the location in the operand manipulation (OM) section of the control storage into which the first microinstruction is to be loaded. Bits 0 through 15 and 17 through 20 are ignored.
- The condition code remains unchanged.
- Possible program exceptions:
  - privileged operation exception
  - addressing exception (specified main storage or control storage location is nonexistent)
  - indirect address specification exception

- indirect addressing exception
- protection exception
- specification exception (operand 2 or IACW not on full-word boundary)

Examples:

```
      LABEL    Δ OPERATION Δ      OPERAND                Δ
      1             10        16
1.                  LCS       5,2,INST
2.                  LCS       7,3,0(4)
3.                  LCS       9,1,8(3)
4.                  LCS       6,7,TAG4(2)
```

1. Six instructions, beginning at location INST, are loaded into control storage. Register 2 contains the address in control storage into which the instructions are loaded.
2. Eight instructions, beginning at the location specified in register 4, are loaded into control storage. Register 3 contains the address in control storage into which the instructions are loaded.
3. Ten instructions, beginning at location 8 indexed by the value in register 3, are loaded into control storage. Register 1 contains the address in control storage into which the instructions are loaded.
4. Seven instructions, beginning at location TAG4 indexed by the value in register 2, are loaded into control storage. Register 7 contains the address in control storage into which the instructions are loaded. TAG4 must have been previously equated to a value not greater than 4095.

9.7 LPSW (LOAD-PROGRAM-STATUS-WORD) – PRIVILEGED INSTRUCTION

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LPSW	$d_1(b_1),i_2$	82	SI	Four Bytes

Function:

The current program status word (PSW) is altered according to the contents of operand 1, specified by $d_1(b_1)$. Operand 2, contained in the i_2 field, contains a secondary operation code.

Operational Considerations

- This is a privileged instruction which is executed and controlled by the supervisor.
- 9400/9480 systems – The double-word operand is in storage at address $d_1 + (b_1)$. This operand is placed into the program status register unaltered. The interrupt code and instruction length code of the current PSW, bits 16 through 33, remain unchanged. The i_2 field of this instruction is ignored. A diagnostic error is issued whenever i_2 is specified.

- Bits 0 through 23 and 34 through 39 of double-word operand 1 replace the corresponding bits of the current PSW. If the RI flag of the current relocation register is 0, bits 40 through 63 of operand 1 replace the corresponding bits (the instruction address field) of the current PSW. If the RI flag of the current relocation register is 1, bits 40 through 63 of operand 1 are added to the offset value from the current relocation register. The sum replaces bits 40 through 63 of the current PSW.

- 90/60,70 systems – The i_2 field contains the secondary operation code. If the secondary operation code is 0, the hardware priority circuit examines the state of all interrupt request lines immediately after the current PSW is replaced by operand 1. If an interrupt request is pending and the corresponding mask bit of the current PSW is 1, an interrupt initiation sequence (IIS) takes place. If the secondary operation code is 1, the hardware priority circuit is inhibited and no IIS can occur after the current PSW is replaced by operand 1. This inhibition is removed when the processor resumes instruction execution under control of the current PSW. If the secondary operation code is not 0 or 1, the result of the LPSW instruction is unpredictable. If i_2 is not specified, 0 is assumed.

- The condition code is set equal to bits 34 and 35 of operand 1.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Privileged operation
Indirect address specification	Specification
Indirect addressing	
Privileged operation	
Protection	
Specification (operand 1 or IACW not on full-word boundary)	

- Relocation and indirection flags (90/60,70):
  - operand 1: RO, IO
  - operand 2: none
  - bits 40 through 63 of operand 1: RI

Examples:

```
      LABEL    Δ OPERATION Δ      OPERAND              Δ
      1        10            16
1.            LPSW           NEWPSW
2.            LPSW           NEWPSW,1
```

1. Bits 0 through 23 and 34 through 63 of the double word specified by NEWPSW replace the contents of the current PSW. The secondary operation code is presumed to be 0 (9400/9480 format).

2. Bits 0 through 23 and 34 through 63 of the double word specified by NEWPSW replace the contents of the current PSW. Since the secondary operation code is 1, on 90/60,70 systems, no interrupt can take place after the current PSW is replaced until the processor resumes execution of instructions (90/60,70 format).

9.8. RDD (READ-DIRECT) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
RDD	$d_1(b_1),i_2$	85	SI	Four bytes

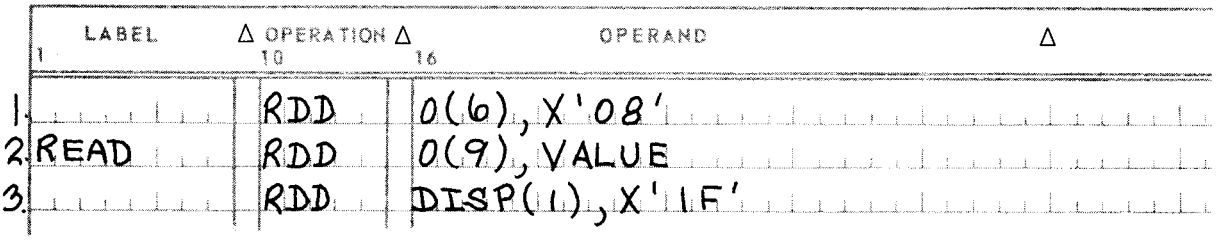
Function:

The operand 1 address specified by $d_1(b_1)$ is used as the storage location for the data received on the direct control bus-in lines. Acceptance of this data prepares the processor for receiving data from an external device or processor from which the signal originated. The operand 2 byte, contained in the i_2 field, is sent as eight timing signals on the timing signal bus-out lines.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The RDD instruction is used as part of the Direct Control and External Interrupt Feature (F1335–00) which provides direct communications of controlling and synchronizing information between two processors or between a processor and an external device.
- The signal received on the eight direct control bus-in lines is placed in the operand 1 storage location, provided no HOLD signal is present. The HOLD signal prevents the direct control bus-in lines from being read until the signal data is present. A parity bit is generated as the data is stored.
- The byte specified as i_2 is sent as eight timing signals on the timing signal bus-out lines. No parity is associated with these signals. At the same time, an identical timing signal is sent on the read-out line to inform the other device that the processor is receiving data on the direct control bus-in lines.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception (direct control and external interrupt feature not installed)
  - privileged operation exception
  - protection exception
  - specification exception (IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: RD, ID
  - operand 2: none

Examples:



1. One byte is read into main storage location 0 indexed by the value in register 6. The value 08_{16} is transmitted to the sending processor.

2. One byte is read into main storage location 0 indexed by the value in register 9. The hexadecimal value in main storage location VALUE is transmitted to the sending processor. VALUE must have been previously equated to a value in the range 0 to 255.

3. One byte is read into main storage location DISP indexed by the value in register 1. The value $1F_{16}$ is transmitted to the sending processor. DISP must have been previously equated to a value not greater than 4095.

9.9. SPM (SET-PROGRAM-MASK)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SPM	r	04	RR	Two bytes

Function:

The program mask field of the current PSW is changed according to the contents of the register specified by r.

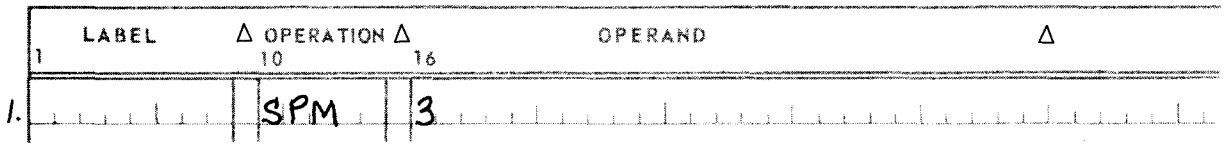
Object Instruction Format:

0 OPERATION CODE 7	8 OPERAND 1 11	12 OPERAND 2 15
04	r	unused

Operational Considerations:

- 90/60,70 systems — Bits 2 through 7 of the full-word contents of the specified register replace the program mask field (bits 34 through 39) of the current PSW.
- Bits 0, 1, and 8 through 31 of the register are ignored.
- 9400/9480 systems — Bits 2 through 5 of the register, defined by r, replace the condition code and program mask portion of the current program status word (PSW), bits 34 through 37. All other bits of this register are ignored.
- The condition code is set equal to bits 2 and 3 of the specified register.
- Possible program exceptions: none
- Relocation and indirection flags: none

Example:



1. The program mask field of the current PSW is changed to the value specified by the contents of bits 2 through 7 of register 3.

9.10, SSK (SET-STORAGE-KEY) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SSK	r_1,r_2	08	RR	Two Bytes

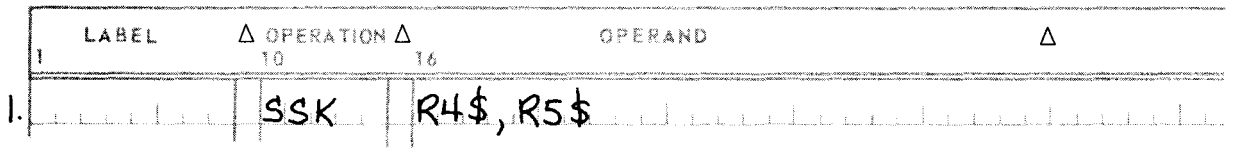
Function:

A main storage key is defined by the contents of operand 1, specified by r_1 and is placed in the processor key memory location designated by the contents of operand 2, specified by r_2.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The 5-bit value contained in bit positions 24 through 28 of operand 1 is placed in the processor key memory location specified by bits 8 through 20 of operand 2.
- The processor key memory location designated by operand 2 is associated with a block of main storage located on an integral boundary which is a multiple of 2048 bytes. Bits 28 through 31 of operand 2 must be 0. Bits 0 through 7 and 21 through 27 of operand 2 are ignored.
- The main storage key, bits 24 through 28 of operand 1, provides for write only or write and read protection within a 2048-byte block of main storage. Bits 24 through 27 provide write protection identity for the specified block. If bit 28 is 1, read protection is provided. Bits 0 through 23 and 29 through 31 of operand 1 are ignored.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception (operand 2 specified a nonexistent block of main storage)
  - privileged operation exception
  - specification exception (bits 28 through 31 of operand 2 are not 0)
- Relocation and indirection flags: none

Example:



1. Presuming that R4$ and R5$ have been equated to 4 and 5, respectively, bit positions 24 through 28 of register 4 are placed in the key memory location specified by bits 8 through 20 of register 5.

9.11. SSM (SET-SYSTEM-MASK) – PRIVILEGED INSTRUCTION

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SSM	d(b)	80	SI	Four Bytes

Function:

The system mask bits of the current PSW are changed according to the contents of the operand, specified by d(b).

Object Instruction Format:

0 OPERATION CODE 7	8 IMMEDIATE OPERAND 15	16 OPERAND 1 19	20 31
80	unused	b	d

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- 9400/9480 systems – The byte operand in storage at address $d_1 + (b_1)$ replaces the system mask field of the current PSW, bits 0 through 6. The low order bit of the byte operand is ignored.
- 90/60,70 systems – Bits 0 through 12 of the half-word operand replace the system mask (bits 0 through 12) of the current PSW.
- The condition code remains unchanged.
- If the location specified by the operand contains the hexadecimal value FFD8, the following interrupts are allowed:
  - program check
  - machine check
  - external
  - timer
  - read direct

- selector 1
- selector 2
- selector 3
- selector 4
- communications intelligence channel
- multiplexer channel - standard
- multiplexer channel - status table

If the location specified by the operand contains the value 0000_{16}, all interrupts are inhibited except the supervisor call interrupt.

- Possible program exceptions:

SPERRY UNIVAC 90/60,70 Systems	SPERRY UNIVAC 9400/9480 Systems
Addressing	Privileged operation
Indirect address specification	
Indirect addressing	
Privileged operation	
Protection	
Specifications 1. Operand 1 not on half-word boundary 2. IACW not on full-word boundary	

- Relocation and indirection flags (90/60,70):
  - operand 1: RO, IO

Example:

```
      LABEL     Δ OPERATION Δ        OPERAND                    Δ
1               10          16
/.              SSM         SYSMASK
```

1. 9400/9480 systems – Bits 0 through 6 of the byte specified by SYSMASK replace the system mask (bits 0 through 6) of the current PSW.

   90/60,70 systems – Bits 0 through 12 of the half word specified by SYSMASK replace the system mask (bits 0 through 12) of the current PSW.

9.12. SVC (SUPERVISOR-CALL)

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SVC	i	0A	RR	Two Bytes

Function:

A supervisor call interrupt request is generated.

Object Instruction Format:

OPERATION CODE (0–7)	OPERAND 1 (8–15)
0A	i

Operational Considerations:

- When the interrupt is granted, the contents of the i field are stored as the interrupt code in the current PSW. The current PSW is stored in the supervisor call old PSW location, and the contents of the supervisor call new PSW location replace the current PSW.
- The condition code is set equal to bits 34 and 35 of the supervisor call new PSW. It remains unchanged in the old PSW.
- Possible program exceptions: none
- Relocation and indirection flags (90/60,70): none

Example:

	LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
1.		SVC	X'0F'

1. A supervisor call interrupt is generated and the value 00001111_2 is stored in the old program status word.

9.13. WRD (WRITE-DIRECT) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
WRD	$d_1(b_1),i_2$	84	SI	Four Bytes

Function:

The contents of the byte at the operand 1 main storage location specified by $d_1(b_1)$ are made static signals on the direct control bus-out lines. The operand 2 byte, contained in the i_2 field, is sent as eight timing signals on the timing signal bus-out lines. These signals are used to alert an external device or another processor for communications.

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The WRD instruction is used as part of the Direct Control and External Interrupt Feature (F1335–00) which provides direct communications of controlling and synchronizing information between two processors or between a processor and an external device.
- The byte contents of operand 1 are made a static signal on the eight direct control bus-out lines. No parity is associated with the signal. The signal remains until the execution of another WRD instruction.
- The byte contained in the i_2 field is sent as eight timing signals on the timing signal bus-out lines. No parity is associated with these signals. At the same time, an identical timing signal is sent on the write-out line to inform the external device that the processor is sending data on the direct control bus-out lines.
- The condition code remains unchanged.
- Possible program exceptions:
  - addressing exception
  - indirect address specification exception
  - indirect addressing exception
  - operation exception (direct control and external interrupt feature not installed)
  - privileged operation exception
  - protection exception
  - specification exception (IACW not on full-word boundary)
- Relocation and indirection flags:
  - operand 1: RO, IO
  - operand 2: none

Examples:

```
      LABEL     Δ OPERATION Δ     OPERAND                    Δ
      1              10        16
1.                  WRD       0(5),X'01'
2.                  WRD       0(5),VALUE
3.                  WRD       DISP(1),X'1F'
```

1. One byte is written from main storage location 0 indexed by the value in register 5. The hexadecimal value 01 is transmitted to the receiving processor.

2. One byte is written from main storage location 0 indexed by the value in register 5. The hexadecimal value in main storage location VALUE is transmitted to the receiving processor. VALUE must have been previously equated to a value in the range 0 to 255.

3. One byte is written from main storage location DISP indexed by the value in register 1. The value $1F_{16}$ is transmitted to the receiving processor. DISP must have been previously equated to a value not greater than 4095.

10. Input/Output Instructions

10.1. GENERAL

The input/output instruction set of the SPERRY UNIVAC Operating System/4 (OS/4) Assembler provides for the initiation, testing, and termination of operations executed by the multiplexer channel selector channel, communications intelligence channel (CIC), and operating system storage facility (OSSF) control channel.

The execution of an input/output instruction begins with the activation of a signal from the processor to the appropriate channel requesting initiation of the operation. The processor then waits for an acknowledge signal from the channel. Depending on the state of the channel, the operation is initiated, or the processor is informed of the reason why the operation was not initiated.

This section describes the operation of each input/output instruction. The instructions are arranged in alphabetical order according to mnemonic operation code. A list of the possible program exceptions and condition codes which may result is included. There are no pertinent relocation and indirection flags for input/output instructions. See Table 3–1 for an explanation of the abbreviations used in describing instruction formats.

Table 10–1 describes the letters used to form codes describing the channel states as applicable to the input/output instruction. The state of the individual unit is indicated by the position of the letter in the channel state code. From left to right within the code is an indication of the state of the channel, subchannel, and subsystem.

Table 10–1. Channel State Codes

Code	Meaning	Unit	Description
A	Available	Channel	Ready to accept I/O instruction and execute the operation specified. Multiplexer channel is always available.
		Subchannel	Selector channel: same as available channel. Multiplexer channel: mode in the hard channel control word is idle.
		Subsystem	If subsystem control unit contains information on the state of the addressed device or queries the addressed device, the subsystem is available if neither the control unit nor the addressed device is executing a previously initiated operation or is holding pending status. If the subsystem contains no information on the state of the addressed device or does not query the addressed device, it is available if the control unit is neither executing a previously initiated operation nor holding pending status.
I	Interrupt Pending	Channel	Selector channel: interrupt-causing device or subchannel status has developed, an interrupt request has been activated, and all other operations have been halted. Multiplexer channel: device status has been accepted from a standard device and stored in the hard channel control word, or an interrupt has been detected in a channel command word related to a standard subchannel, an interrupt request has been activated, and all other status is in a waiting state.
		Subchannel	Selector channel: same as channel in interrupt pending state. Multiplexer channel: mode in the hard channel control word is terminated or reset.
		Subsystem	The subsystem is holding pending device status.
N	Nonoperational	Channel	Channel is offline or not present.
		Subchannel	Multiplexer channel: specified subchannel is not installed. Selector channel: same as nonoperational channel.
		Subsystem	Offline, powered down, not installed, or does not recognize its address during the initial selection sequence.
W	Working	Channel	Selector channel: operating in burst mode or a chaining sequence is in progress. Multiplexer channel: not possible.
		Subchannel	Selector channel: same as working channel. Multiplexer channel: mode in the hard channel control word is active or chain.
		Subsystem	Control unit or device is executing a previously initiated operation.
X	Any operational state		

10.2. HIO (HALT-I/O) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
HIO	d(b)	9E	SI	Four Bytes

Function:

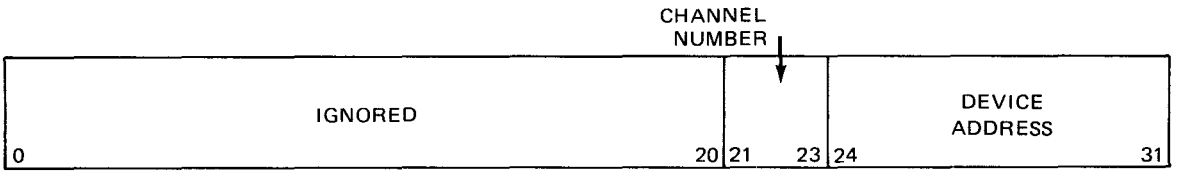
The halt-I/O instruction causes the addressed selector channel, subchannel, and device to terminate the current operation. Any pending device or subchannel status is stored in the initial status word (ISW), and the appropriate condition code is set.

Object Instruction Format:

0 OPERATION CODE 7	8 IMMEDIATE OPERAND 15	16 OPERAND 1 19	20 OPERAND 1 31
9E	unused	b	d

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The HIO instruction applies to the operation of the selector channel only.
- The contents of the 32-bit register specified by the b field are added to the contents of the d field to form a 32-bit field with the following format:



- The channel specified by bits 21 through 23 and the device and subchannel, implied by the device number, specified by bits 24 through 41 are addressed, and the operation proceeds as follows:

State	Procedure
AAX	If the addressed channel and subchannel are in the available state, the condition code is set to 0 and no ISW is written.
WWX	If the addressed channel is transferring data in burst mode, the device address in the HIO instruction is ignored, and the operation with the device is terminated by an interface disconnect sequence. Command or data chaining is suppressed. An ISW with the device address of the terminated device and the incorrect length bit set to 1 is written. The condition code is set to 2. The mode is set to idle.

State	Procedure
IIX	If the addressed channel and subchannel are in the interrupt pending state with pending device or subchannel status, the pending status is written into the ISW, the interrupt pending condition is cleared, and the condition code is set to 1. The mode is set to idle.
NXX	If the channel is not available or nonoperational, the condition code is set to 3 and no ISW is written. If the channel is available or in the interrupt pending state, the device is not addressed. In these cases, the fact that a device was nonoperational would not be indicated until the next reference of the device for a test-I/O or start-I/O instruction.

Table 10–2 summarizes the resulting condition codes and ISW contents for the HIO instruction.

Table 10–2. HIO Instruction Condition Codes and Initial Status Words

Channel	Channel State	Condition Code	ISW Contents		
			Device Address	Device Status	Subchannel Status
Selector	AAX	0	No ISW written		
	WWX	2	Active device address *	Unpredictable	Incorrect length and any other present
	IIX	1	Address of device associated with pending status *	Any pending status	Any pending status
	NXX	3	No ISW written		
Multiplexer	XXX	1	Addressed device	0	Program check

* The device address specified in the operand field of HIO is ignored.

- An HIO instruction issued to the multiplexer channel results in an ISW with the program check bit set and a condition code of 1.
- Possible program exceptions:
  - – privileged operation exception
- Relocation and indirection flags: none

Example:

```
     LABEL      △ OPERATION △          OPERAND                  △
     1            10        16
1.               HIO        10(8)
```

1. The channel and device address is represented by the sum of 10 and the contents of register 8.

10.3. LCHR (LOAD-CHANNEL-REGISTER) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
LCHR	d(b)	AD	SI	Four Bytes

Function:

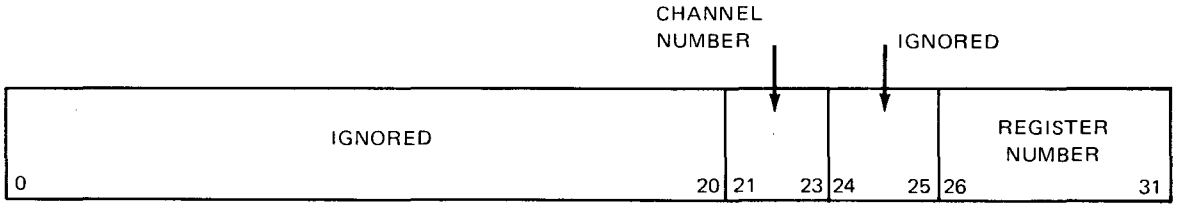
The load-channel-register instruction is used to transfer 32 bits of information from location 180_{16} in main storage to a location in the channel register stack (CRS), specified by d(b).

Object Instruction Format:

0 OPERATION CODE 7	8 OPERAND 1 15	16 19	20 OPERAND 2 31
AD	unused	b	d

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The LCHR instruction applies to the operation of the multiplexer channel.
- The contents of the 32-bit register specified by the b field are added to the contents of the d field to form a 32-bit field with the following format:



- The register number field (bits 26 through 31) specifies in binary the location of one of 32 full words in the CRS (64 if the Subchannel Expansion Feature, F1518, is installed). The contents of location 180 are transferred to the specified register, and the condition code is set as follows:
  - If the transfer is completed successfully, the condition code is set to 0.
  - If the channel detects a storage error on the access of location 180, an ISW with the appropriate channel control check code is written and the condition code is set to 1.
  - If the channel is nonoperational or the specified register is not installed, the condition code is set to 3.
  - If the LCHR instruction is issued to a selector channel in the available state, an ISW with the program check bit set is written; the condition code is set to 1. An LCHR instruction issued to a selector channel in any state other than available causes the condition code to be set to 2.

Table 10–3 summarizes the condition codes and ISW's for the LCHR instruction. The codes for the channel states are given in Table 10–1.

Table 10–3. LCHR Instruction Condition Codes and Initial Status Words

Channel	Channel State	Condition Code	ISW Contents		
			Device Address	Device Status	Subchannel Status
Multiplexer	AXX	0	No ISW written		
	AXX	1	0	0	Channel control check *
	NXX ANX	3	No ISW written		
Selector	AXX	1	0	0	Program check
	IXX WXX	2	No ISW written		

* Storage error in access of 180_{16}.

- Possible program exceptions:
  - privileged operation exception
- Relocation and indirection flags: none

Example:

LABEL (1)	△ OPERATION △ (10)	OPERAND (16) △
1.	LCHR	LOADC(6)

1. The contents of location LOADC and the contents of register 6 are added to form the binary value of the channel and device address. Assume that LOADC has been previously defined as equal to a value not greater than 4095.

10.4. SCHR (STORE-CHANNEL-REGISTER) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SCHR	d(b)	AC	SI	Four Bytes

Function:

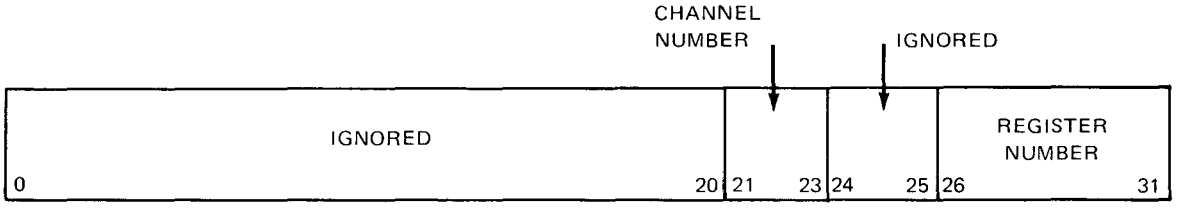
The store-channel-register instruction is used to transfer 32 bits of information from a specified location in the CRS, specified by d(b), to location 180_{16} in main storage.

Object Instruction Format:

0 OPERATION CODE 7	8 IMMEDIATE OPERAND 15	16 OPERAND 1 19	20 OPERAND 1 31
AC	unused	b	d

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The SCHR instruction applies to the operation of the multiplexer channel.
- The contents of the 32-bit register specified by the b field are added to the contents of the d field to form a 32-bit field with the following format:



The register number field (bits 26 through 31) specifies in binary the location of one of 32 full words in the CRS (64 if the Subchannel Expansion Feature, F1518, is installed). The contents of the specified register are transferred to location 180_{16}, and the condition code is set as follows:

– If the transfer is completed successfully, the condition code is set to 0.

– If the channel detects a storage error on the access of location 180, an ISW with the appropriate channel control check code is written and the condition code is set to 1.

– If the channel is nonoperational or the specified register is not installed, the condition code is set to 3.

– If the SCHR instruction is issued to a selector channel in the available state, an ISW with the program check bit set is written; the condition code is set to 1. An SCHR instruction issued to a selector channel in any state other than available causes the condition code to be set to 2.

Table 10–4 summarizes the condition codes and initial status words for the SCHR instruction. The codes for the channel states are given in Table 10–1.

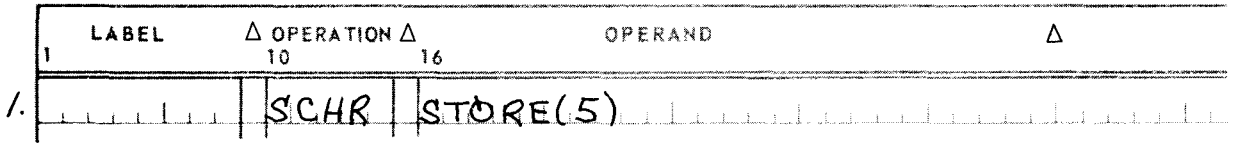
Table 10–4. SCHR Instruction Condition Codes and Initial Status Words

Channel	Channel State	Condition Code	ISW Contents		
			Device Address	Device Status	Subchannel Status
Multiplexer	AXX	0	No ISW written		
	AXX	1	0	0	Channel control check *
	NXX ANX	3	No ISW written		
Selector	AXX	1	0	0	Program check
	IXX WXX	2	No ISW written		

* Storage error in accesss of 180_{16}.

- Possible program exceptions:
  - privileged operation exception
- Relocation and indirection flags: none

Example:



1. The contents of main storage location STORE and the contents of register 5 are added to form the binary value of the channel and device address. Assume that STORE has been previously equated to a value not greater than 4095.

10.5. SIO (START-I/O) — PRIVILEGED INSTRUCTION

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
SIO	d(b)	9C	SI	Four Bytes

Function:

The start-I/O instruction is used to initiate all read, read backwards, write, control, and sense operations. If in the proper state, the specified channel reads the channel address word (CAW) and the first CCW and initiates the operation with the device. On the 90/60,70 systems, if any device or subchannel status develops during the initiation of the operation, an ISW is written. The completion of the SIO instruction sets the appropriate condition code in the current PSW.

Object Instruction Format:

0 OPERATION CODE 7	8 IMMEDIATE OPERAND 15	16 OPERAND 1 19	20 OPERAND 1 31
9C	unused	b	d

9400/9480 Operational Considerations:

- This is a privileged instruction and is executed only in the supervisor mode.
- Execution of the start I/O instruction performs the following:
  - Accesses the channel address word (CAW) for the address of the first channel command word (CCW) for selector channels. CAW is stored in supervisor general register 0.
  - The selector channel puts the CCW into hardware.
  - The channels access the device and initiate the operation.

- The CCW for the selector channel or the SCW and BCW for the multiplexer channel specify the type of operation, data address, controls, and data byte count.

- The 32 binary value produced by adding the contents of b_1 to the d_1 field specify the channel and device address. The channel is specified by bits 21 through 23 and the device specified by bits 24 through 31.

  – The specified device is initiated by and operates under the control of the CCW, SCW, or BCW.

- Condition code is set as follows:

  – to 0 (0) if the I/O operation is initiated and is being executed;

  – to 1 (01) if an immediate status word has been stored;

  – to 2 (10) if the selector channel is busy;

  – to 3 (11) if device or channel is not operational.

- Basic procedures for using the start I/O instruction are as follows:

  – establish one or more CCWs, (SEL), or SCWs and BCWs (MPX) in main storage;

  – load the channel address word (CAW) with the address of the first CCW (SEL) or the command in the CAW (MPX);

  – specify the channel and device number in the operand 1 portion of the start I/O instruction:

  – issue the start I/O instruction;

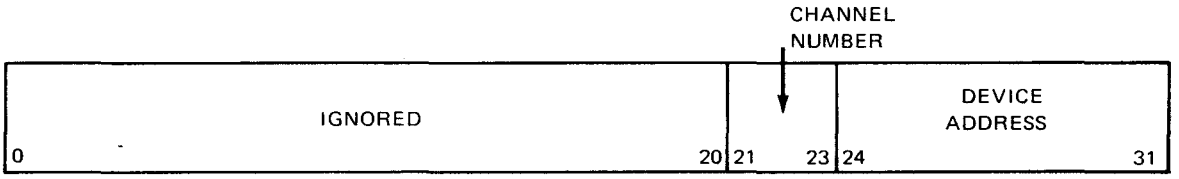
  – test the conditon code for determination of result of I/O operation.

90/60,70 Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.

- The SIO instruction applies to the operation of the multiplexer and selector channels.

- The contents of the 32-bit register specified by the b field are added to the contents of the d field to form a 32-bit field with the following format:



- The channel specified by bits 21 through 23 and the device specified by bits 24 through 31 are addressed and the operation proceeds as follows:

State	Procedure
AAA	If the addressed channel, subchannel, and device are in the available state, the condition code is set to 0 and the prescribed I/O operation proceeds under the control of the channel and subchannel.
IIX, WWX (Selector) XIX,XWX (Multiplexer)	If the addressed channel is in any state other than available, the device is not addressed; the operation is halted, and the condition code is set to 2.
AAW, AAI	If the addressed channel and subchannel are available but the addressed device is in the working or interrupt pending state, an ISW with the appropriate device status is written and the condition code is set to 1. For devices which present channel end and device end separately, an available subchannel may have a working or interrupt pending device. In this case, the selector channel or multiplexer subchannel becomes available after the interrupt with the channel end indication. The SIO instruction is issued before the device end has been accepted at the channel. Similarly, in subsystems with the dual channel access or dual access facility, the control unit may be operating under the control of an active subchannel in another channel while receiving the SIO instruction from the addressed channel. If the control unit is in the interrupt pending state with device status (for example, device end or attention) for the addressed device, the pending device status including the busy bit is written into the ISW and cleared in the control unit. If the control unit is working or in the interrupt pending state with pending status for a device other than the addressed device, the device status written into the ISW will contain the busy and status modifier bits; however, the pending status is not cleared in the control unit.
NXX, XNX, XNN	If the addressed channel, subchannel, or device is nonoperational, the condition code is set to 3.

The resulting condition codes and ISW contents are summarized in Table 10–5. The codes for the channel states are given in Table 10–1.

Table 10–5. SIO Instruction Condition Codes and Initial Status Words

Channel	Channel State	Condition Code	ISW Contents		
			Device Address	Device Status	Subchannel Status
Multiplexer and Selector	AAA	0	No ISW written		
	AAI	1	Addressed device	①	0
	AAW	1	Addressed device	②	0
	NXX XNX XXN	3	No ISW written		
Multiplexer	XIX XWX	2	No ISW written		
Selector	IIX WWX	2	No ISW written		

① If the control unit contains pending status for the addressed device, busy and pending status bits are set in the device status field; otherwise, busy and status modifier bits are set.

② If the control unit is working, busy and status modifier bits are set in the device status field. If the control unit is available and the device is working, only the busy bit is set.

- Detection of any of the following errors causes the operation to be aborted by a selective reset to the device:
  - storage errors in the access of the CAW, the relocation register, or the first CCW;
  - format errors in the contents of the CAW or CCW;
  - parity errors in the address or status from the device.

  If any of these errors are detected, an ISW is written with the appropriate subchannel status bits set and the condition code is set to 1.
- Possible program exceptions:
  - privileged operation exception
- Relocation and indirection flags: none
  - none

Example:

1. Ten is added to the contents of register 8 to form the binary value of the channel and device address.

10.6. TCH (TEST-CHANNEL) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
TCH	d(b)	9F	SI	Four Bytes

Function:

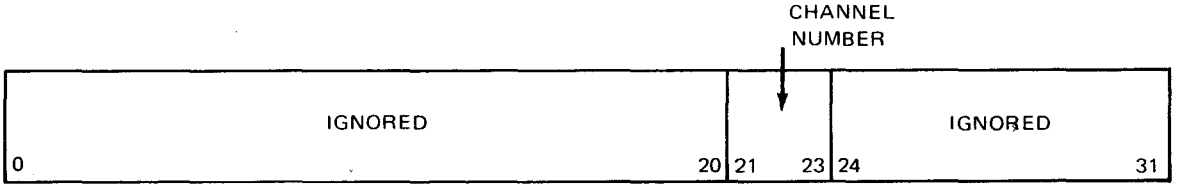
The test-channel instruction is used to determine the current state of the addressed channel. The appropriate condition code is set in the current PSW and any pending status is written into the ISW.

Object Instruction Format:

0 OPERATION CODE 7	8 IMMEDIATE OPERAND 15	16 19	20 OPERAND 1 31
9F	unused	b	d

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The contents of the 32-bit register specified by the b field are added to the d field to form a 32-bit field with the following format:



- The channel specified by bits 21 through 23 is addressed and the condition code is set. The operation proceeds as follows:

State	Procedure
AXX	If the addressed channel is available, the condition code is set to 0 and no ISW is written.
IXX	If the addressed channel is in the interrupt pending state, the pending device and subchannel status and the device address associated with that status are written into the ISW and the condition code is set to 1. The interrupt request associated with the pending status is cleared.
WXX	If the addressed channel is in the working state, the condition code is set to 2 and no ISW is written. A condition code of 2 from the selector channel implies that the channel is operating in burst mode. A condition code of 2 from the multiplexer channel is not possible.
NXX	If the addressed channel is not present or not operational, the condition code is set to 3 and no ISW is written.

Table 10–6 summarizes the resulting condition codes and ISW contents for the TCH instruction. The codes for the channel states are given in Table 10–1.

Table 10–6. TCH Instruction Condition Codes and Initial Status Words

Channel	Channel State	Condition Code	ISW Contents		
			Device Address	Device Status	Subchannel Status
Multiplexer and Selector	AXX	0	No ISW written		
	IXX	1	Address of device associated with pending status	Any pending status	Any pending status
	NXX	3	No ISW written		
Selector	WXX	2	No ISW written		

- Possible program exceptions:
  - privileged operation exception
- Relocation and indirection flags: none

Example:

```
      LABEL     Δ OPERATION Δ          OPERAND                    Δ
1               10          16
/.              TCH         1(9)
```

1. One is added to the contents of register 9 to form the binary value of the channel and device address.

10.7. TIO (TEST-I/O) – PRIVILEGED INSTRUCTION – 90/60,70

Mnemonic Operation Code	Source Code Operand Format	Hexadecimal Operation Code	Format Type	Object Instruction Length
TIO	d(b)	9D	SI	Four Bytes

Function:

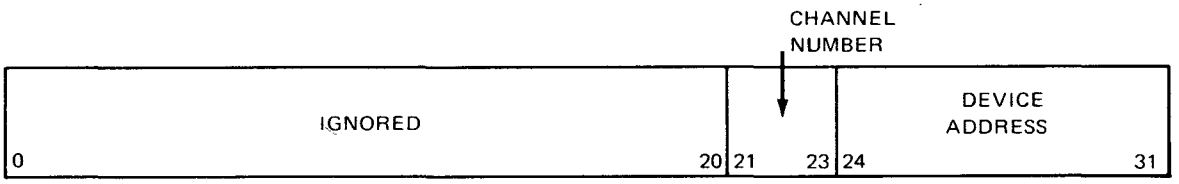
The test-I/O instruction is used to determine the current state of the addressed channel, subchannel, and device. The appropriate condition code is set in the current PSW. Any status pending in the subchannel or device is written into the ISW and the pending interrupt condition is cleared.

Object Instruction Format:

0 OPERATION CODE 7	8 IMMEDIATE OPERAND 15	16 OPERAND 1 19	20 OPERAND 1 31
9D	unused	b	d

Operational Considerations:

- This is a privileged instruction which is executed and controlled by the supervisor.
- The contents of the 32-bit register specified by the b field are added to the contents of the d field to form a field with the following format.



- The contents of the 32-bit register specified by bits 21 through 23, and the device and subchannel implied by the device address specified by bits 24 through 31, are addressed and the condition code is set. The operation proceeds as follows:

State	Procedure
AAA	If the addressed channel, subchannel, and device are available, the condition code is set to 0 and no ISW is written.
AAI, AAW	If the addressed channel and subchannel are available and the control unit contains pending status for the addressed device, the device status is cleared in the control unit and written into the ISW. The condition code is set to 1. If the control unit is working or contains pending status for a device other than the one addressed, the device status written in the ISW contains the busy and status modifier bits, and any pending status is not cleared in the control unit. If the control unit is available but the addressed device is working, the device status contains the busy bit only.
AIX (Multiplexer)	If the addressed channel is available (or interrupt pending for other than the addressed device) but the addressed subchannel is in the interrupt pending state for the addressed device, the device is addressed and any device status returned is written along with any pending subchannel status into the ISW. The condition code is set to 1. Depending on the setting of the mode in the appropriate hard channel control word (HCCW) and the device status returned, the subchannel is left in the interrupt pending state or cleared to the available state. If the mode is reset, then any device status returned and the pending subchannel status are written in the ISW. The mode is then set to idle and the subchannel is available. If the mode is terminate, the device status must be examined. If the device status contains at least the busy bit, the device status and any pending subchannel status are written into the ISW. However, the mode is left as terminate and the subchannel remains in the interrupt pending state. If the device status contains at least the channel end bit, the device status and any subchannel status are written into the ISW. The mode is then set to idle and the subchannel is available.

State	Procedure
	If the addressed subchannel contains pending subchannel status for a device other than the addressed device, the addressed device is not interrogated, the condition code is set to 2 and the subchannel status is left pending.
IIX (Selector)	If the addressed channel (or subchannel) contains pending device or subchannel status for the addressed device, the device is addressed and any status returned is merged with the pending status in the ISW; the condition code is set to 1. If the addressed channel contains pending status for a device other than the addressed device, the condition code is set to 2 and the status is left pending. (If the addressed device became nonoperational between the time that device status was accepted or subchannel status developed and the time that the device is addressed, the pending status would be written into the ISW and the condition would be cleared in the channel. The fact that the device was nonoperational would be indicated the next time the device is interrogated.)
IXX (Multiplexer)	If the addressed channel is in the interrupt pending state with pending device status for the addressed device, the device is interrogated and any device status is merged with the pending device status. The merged device status and any associated subchannel status is written into the ISW and the condition code is set to 1. If the addressed channel is in the interrupt pending state for other than the addressed device, the state of the subchannel determines further operation. That is, if the subchannel is in the interrupt pending state, the operation continues as for the AIX state; if in the working state, as for the XWX state.
WXX (Selector)	If the addressed channel or subchannel is in the working state, the condition code is set to 2 and the operation with the device continues.
XWX (Multiplexer)	If the addressed subchannel is in the working state, the condition code is set to 2 and the operation with the device continues.
NXX, XNX, XXN	If the addressed channel, subchannel, or device is not present or not operational, the condition code is set to 3.

The resulting condition codes and ISW contents are summarized in Table 10–7. The codes for the channel states are given in Table 10–1.

Table 10–7. TIO Instruction Condition Codes and Initial Status Words

Channel	Channel State	Condition Code	ISW Contents		
			Device Address	Device Status	Subchannel Status
Multiplexer and Selector	AAA	0	No ISW written		
	AAI	1	Addressed device	Pending status (1)	Any present
				Busy and status modifier (2)	
	AAW	1	Addressed device	Busy and status modifier (3)	Any present
				Busy (4)	
	NXX XNN XXN	3	No ISW written		
Multiplexer	AIX	1	Addressed device	Any returned by device (5)	Pending status
		2	No ISW written (6)		
	IXX	1	Addressed device	Merge of returned and pending (1)	Any present
		Same as AXX, where X = A, I, or W (2)			
	XWX	2	No ISW written		
Selector	IIX	1	Addressed device	Merge of returned and pending (5)	Any pending
		2	No ISW written (6)		
	WXX	2	No ISW written		

(1) Pending status for addressed device.

(2) Pending status for other than addressed device.

(3) Control unit working.

(4) Control unit available; device working.

(5) Interrupt pending for addressed device.

(6) Interrupt pending for other than addressed device.

11. Data and Storage Definition

11.1. GENERAL

Two statements are available to the SPERRY UNIVAC Operating System/4 (OS/4) for specifying either data to be used as stored constants (DC) or storage areas to be reserved (DS). The formats for these statements are similar to machine instruction format. A symbol may appear in the label field, is assigned the address of the leftmost character of the constant or storage area specified in the operand field, and has a length attribute equal to that of the constant or storage area. The maximum length attribute of a symbol is 256. The programmer must branch around areas reserved in line with the program.

The operation code is DC (define constant) or DS (define storage). The operand field is divided into subfields which specify the information needed to create the data or storage area. The general format of the operand field is described in 11.2 and 11.3. Table 11–1 lists the characteristics of constant and storage types. Detailed descriptions are provided in 11.7 and 11.8.

Table 11–1. Characteristics of Constant and Storage Types

Type Code	Constant or Storage Type	Alignment	Storage Code Specification	Storage Format	Truncation or Padding	Length in Bytes		
						Implied	Minimum Explicit	Maximum Explicit
C	Character	None	Characters	Character	Right	Variable	1	256 (DC) 65,535 (DS)
X	Hexadecimal	None	Hexadecimal digits	Hexadecimal	Left	Variable	1	256 (DC) 65,535 (DS)
B	Binary	None	Binary digits	Binary	Left	Variable	1	256
P	Packed decimal	None	Decimal digits	Packed decimal	Left	Variable	1	16
Z	Zoned decimal	None	Decimal digits	Character	Left	Variable	1	16
H	Half-word fixed point	Half word	Decimal digits	Fixed-point binary	Left	2	1	8
F	Full-word fixed point	Full word	Decimal digits	Fixed-point binary	Left	4	1	8
Y	Half-word address	Half word	Expression	Binary	Left	2	1	2
A	Full-word address	Full word	Expression	Binary	Left	4	1	4
S	Base and displacement	Half word	One or two expressions	Base and displacement	None	2	2	2
V	External address	Full word	Relocatable	Binary	Left	4	4	4

11.2. DC (DEFINE CONSTANT) STATEMENT

The DC statement specifies data to the assembler that is to be used as stored constants. These constants are generated and produced in object output format ready to be loaded along with the program instructions. The format of the DC statement is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DC	[d] t [Ln] {'c' / (c)}

where:

d is the duplication factor.

t is the type of constant.

Ln is the explicit length (modifier).

C is the constant specification.

Multiple operands in a DC statement are not permitted.

The various constants and the specifications necessary for each type of DC statement are described in 11.4.

11.3. DS (DEFINE STORAGE) STATEMENT

The DS statement is used to specify a storage area to be reserved by the assembler.

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DS	[d] t [Ln] [{'c' / (c)}]

where:

d is the duplication factor.

t is the type of constant.

Ln is the explicit length (modifier).

c is the constant specification.

The various constants and specifications necessary for each type of DS statement are described in 11.4. The following modifications should be noted:

- Data can be specified in the constant subfield; however, the constant is not assembled. It allows the assembler to determine the size of the storage area needed when implied length is variable. The type field must be specified.
- All types of constants are legal.

- C and X type constants have a maximum length of 65,535 bytes instead of 256 bytes as in DC statements and literals. However, the maximum length attribute associated with the constant subfield is 256.
- Storage areas reserved by a DS statement are not set to 0.
- Storage locations reserved by boundary alignment for DS statements are not set to 0.

The grouping together of all DS statements produces a more efficient object code.

11.4. DC AND DS STATEMENT OPERAND SUBFIELDS

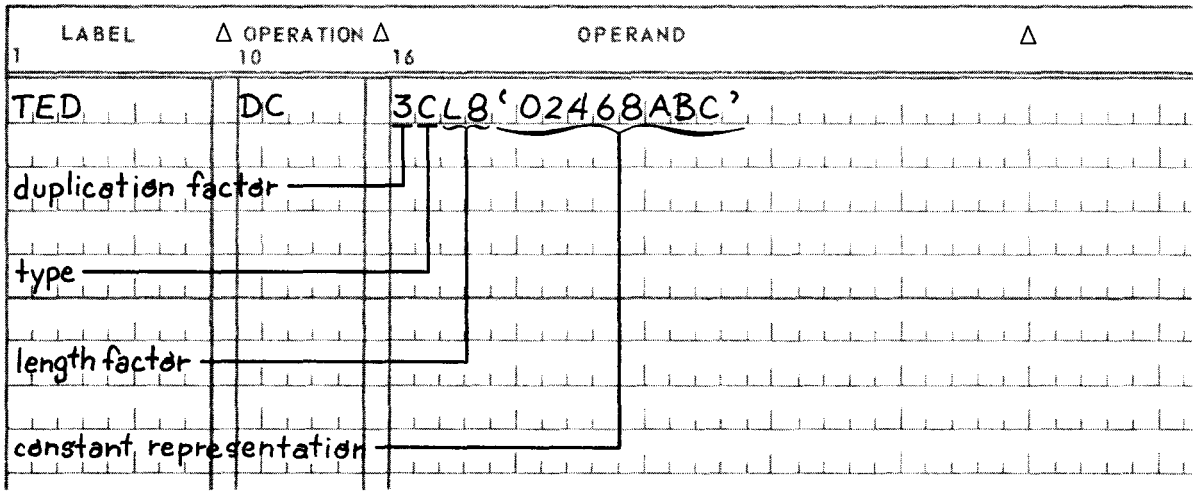
The operand field (dtlc) is divided into four subfields that describe and identify the data or storage space to be generated. The four subfields are:

- Duplication (d)
- Type (t)
- Length modifier (l)
- Constant (c)

The subfields must be specified in the stated order with the duplication factor first and the constant last. The type (t) subfield is always present, the constant (c) subfield is required for DC statements only.

The following is a valid example for a typical DC statement with the subfields identified.

Example:



Blanks or punctuation marks cannot separate the subfields. Blanks can appear only in the constant subfield as part of a character constant.

11.4.1. Duplication Subfield

The duplication subfield designates the number of identical constants to be generated. Either an unsigned decimal self-defining value or a positive absolute expression enclosed in parentheses (with all terms predefined) may be used to specify the duplication factor. If the subfield is omitted, the duplication factor is assumed to be 1. A duplication factor of 0 does not generate a constant or storage area, but advances the location counter to proper boundary alignment if no length is specified and assigns the location counter value to the symbol in the label field (if a label is present). In generating literals, a duplication factor of 0 is illegal.

11.4.2. Type Subfield

The type subfield designates the type of constant to be generated (11.7). One of 11 characters is used to specify this type. Paraforms and set symbols cannot be used to specify replacement of the type character. Valid characters and what they represent are shown in Table 11–1. The type subfield is to be present as it determines the alignment, truncation, storage form, and implicit length of the constant.

11.4.3. Length Modifier Subfield

The length modifier subfield designates the number of bytes to be used in generating the constant. The length factor follows the character L and can be either an unsigned decimal value or a positive absolute expression enclosed in parentheses (with all terms predefined). This factor can be stated for all types of constants and is used to establish the number of bytes the constant or storage definition occupies. Constants that do not exactly fit the area specified are padded or truncated to the length specified. The character L may not be generated by the replacement of paraforms or set symbols.

If the length factor is specified, boundary alignment is not provided; however, when the length is omitted, the implied length is used and boundary alignment is provided for most types of constants (Table 11–1).

11.4.4. 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 various types of constants are represented in different ways. 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)

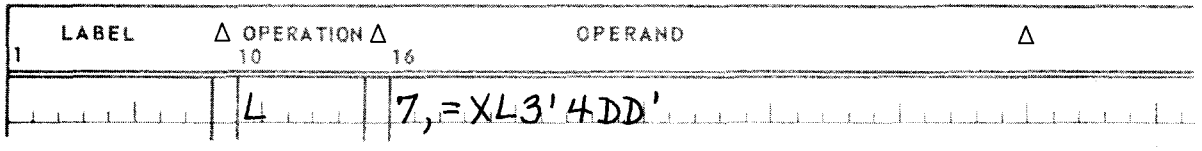
11.5. LITERALS

Literals can be used in machine instructions wherever a storage address is permitted, because they are replaced with the storage address of the constant generated from the literal specifications.

A literal is identified by an equal sign (=) preceding a constant specification in the format described in 11.4.

The handling of literals by the assembler is described in 2.3.2.

Example:



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

- Only one literal can appear in an instruction.
- Assembler directives cannot contain literals.
- Literals cannot have a duplication factor of 0.
- S type literals are not permitted.

11.6. ALIGNMENT

All machine instructions must be aligned on half-word boundaries. The first byte of the instruction must have an address that is divisible by 2. Constants, however, can be aligned on a half word, full word, or no boundary at all. Table 11–1 indicates the kind of alignment, when necessary, for data or storage definition statements if no length factor is stated. When a length factor is specified by the programmer, no alignment is provided. A duplication factor of 0 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 provides a convenient means of obtaining a boundary alignment before generating a constant that is 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.

11.7. DATA CONSTANT TYPES

Data constants are absolute values generated by the assembler which require no modification by the relative loader. The seven types of data constants are discussed in the following paragraphs.

11.7.1. Character Constants

A character constant is specified by the character C in the type subfield and up to 255 characters enclosed by apostrophes in the constant subfield. Any of the 256 valid card punch combinations can be used. 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 (if necessary) to the right of the last character and to the length specified. Boundary alignment is not required.

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. Multiple constants are not permitted for this constant type.

Examples:

	LABEL	Δ OPERATION Δ	OPERAND	Δ
1.		DC	CL2'A'	
2.		DC	C'A'	
3.		DC	C'A '	
4.		DC	CL10' EMPL DIV '	
5.		DC	CL12' '	
6.		DC	3CL4'12345'	
7.		DC	3CL6'12345'	

1. A 2-byte constant containing the following:

   AΔ

2. A 1-byte constant containing the following:

   A

3. A 2-byte constant containing the following:

   AΔ

4. A 10-byte constant containing the following:

   ΔEMPLΔDIVΔ

5. A 12-byte constant containing blanks.

6. A 12-byte constant containing the following:

   123412341234

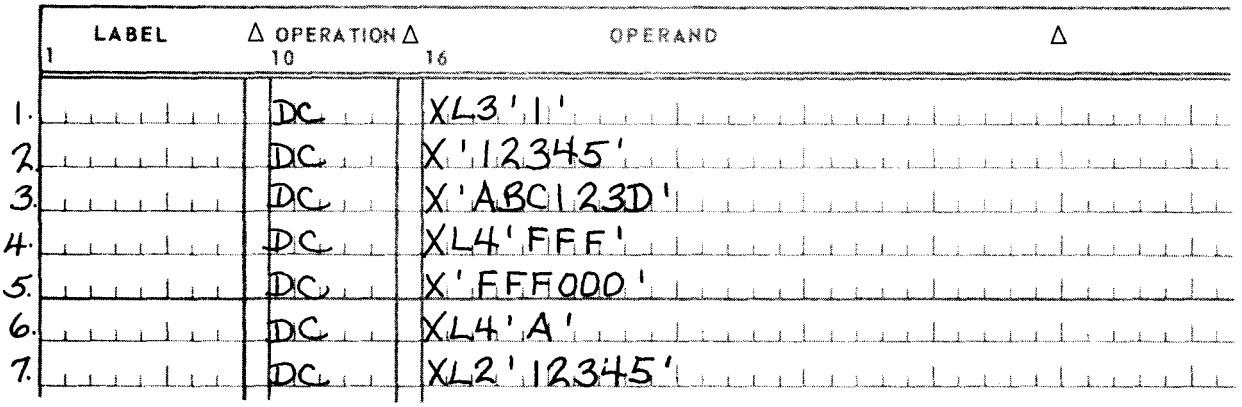
7. An 18-byte constant containing the following:

   12345Δ12345Δ12345

11.7.2. Hexadecimal Constants

A hexadecimal constant is specified by the character X in the type subfield and up to 255 hexadecimal digits enclosed by apostrophes in the constant subfield. Two hexadecimal digits are assembled into one byte. The maximum length that can be specified in the length modifier for a hexadecimal constant is 256 bytes (512 hexadecimal digits). 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 the length factor is present, the hexadecimal specification is truncated or filled with hexadecimal 0's (on the leftmost end) if necessary, to the length specified.

Examples:



1. A 3-byte constant containing the following:

   00000000 00000000 00000001

2. A 3-byte constant containing the following:

   00000001 00100011 01000101

3. A 4-byte constant containing the following:

   00001010 10111100 00010010 00111101

4. A 4-byte constant containing the following:

   00000000 00000000 00001111 11111111

5. A 3-byte constant containing the following:

   11111111 11110000 00000000

6. A 4-byte constant containing the following:

   00000000 00000000 00000000 00001010

7. A 2-byte constant containing the following:

   00100011 01000101

11.7.3. Binary Constants

A binary constant is specified by the character B in the type subfield and up to 255 binary digits (bits) enclosed by apostrophes in the constant subfield. Eight bits are assembled into one byte. The maximum length that can be specified in the length modifier for a binary constant is 256 bytes. Because only 255 bits can be specified, only the least significant 32 bytes of the constant may have their value specified. The high order bytes are zero filled. Binary 0's are added to the leftmost end as necessary to ensure byte boundary alignment. If no length factor is specified, the length in bytes of the constant is one-eighth the number of the digits specified (the binary 0's added are counted as digits). If the length factor is present, the binary specification is truncated or filled with binary 0's (on the leftmost end) as necessary, to the length specified.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		DC	B'101'
2.		DC	BL2'11100101101000011010'
3.		DC	BL2'10110111'

1. A 1-byte constant containing the following:

   00000101

2. A 2-byte constant (with most significant bits truncated) containing the following:

   01011010 00011010

3. A 2-byte constant containing the following:

   00000000 10110111

11.7.4. Packed Decimal Constants

A packed decimal constant is specified by the character P in the type subfield and up to 31 decimal digits enclosed by apostrophes in the constant subfield. The digits are packed two digits to a byte; therefore, each decimal digit requires four bits. A leading sign (+ or –) can be coded within the apostrophes. A plus sign is represented by a hexadecimal C and a minus sign is represented by a hexadecimal D. If a sign is not specified, a plus sign is assumed. Multiple constants are permitted.

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 the possible addition of 0 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)	OPERATION (10)	OPERAND (16)
1.		DC	P'+468'
2.		DC	PL2'24.76'
3.		DC	PL3'-325'
4.		DC	3PL2'381'

1. A 2-byte constant containing the following:

   468C

2. A 2-byte constant (with most significant digit truncated) containing the following:

   476C

3. A 3-byte constant containing the following:

   00325D

4. A 6-byte constant containing the following:

   381C381C381C

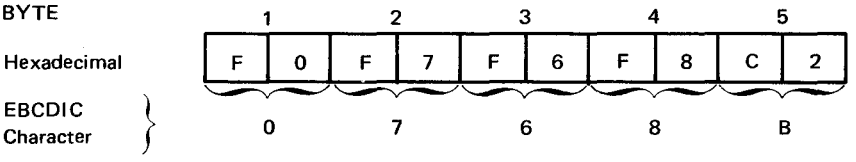
11.7.5. Zoned Decimal Constants

A zoned (unpacked) decimal constant is specified by the character Z in the type subfield and by up to 16 decimal digits enclosed by apostrophes in the constant subfield. A plus or minus sign can be coded within the apostrophes; if none is present, a positive sign is assumed. The digits are assembled one to a byte with a hexadecimal F (EBCDIC) or hexadecimal 3 (ASCII) inserted into 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 the length factor is present, the decimal specification is truncated or filled with 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.

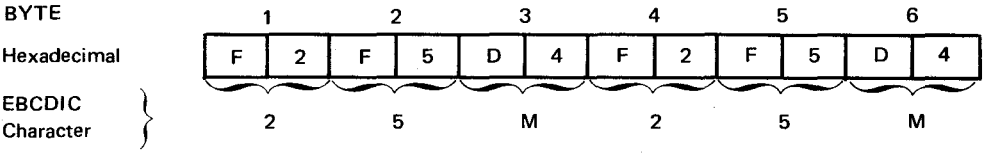
Examples:

```
        LABEL    Δ OPERATION Δ    OPERAND                    Δ
        1        10              16
1.      .....    DC              ZL5'7682'
2.      .....    DC              2ZL3'-6254'
```

1. A 5-byte constant containing the following:



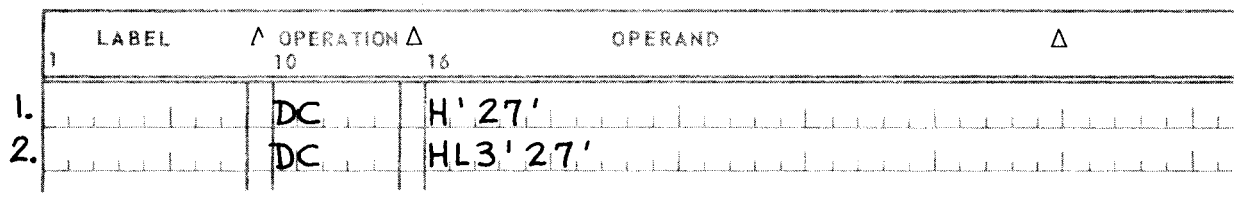
2. A 6-byte constant containing the following:



11.7.6. Half-Word Constants

A half-word constant is specified by the character H in the type subfield and up to 10 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 the implied length of two bytes and must not contain a value greater than +32767 or −32768. 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 of the entire number is converted into a binary format for storage.

Examples:



1. A 2-byte constant containing the following:

   00000000 00011011

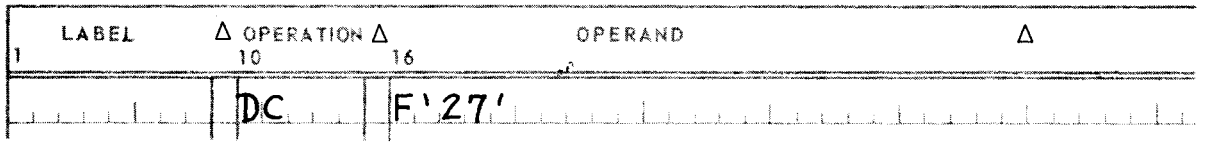
2. A 3-byte constant containing the following:

   00000000 00000000 00011011

11.7.7. Full-Word Constants

A full-word constant is specified by the character F in the type subfield and up to 10 significant decimal digits enclosed by apostrophes in the constant subfield. A plus or minus sign can be included within the apostrophes. If a length factor is not specified, the constant has the implied length of four bytes. 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 a mixed number is lost and the decimal value is converted into a binary format for storage.

Example:



A 4-byte constant containing the following:

00000000 00000000 00000000 00011011

11.8. ADDRESS CONSTANT TYPES

Address constants are relocatable values generated by the assembler that are usually altered by the relative loader to reflect the storage address that the program occupies when it is executed. Address constants are often used to load base registers or to provide a means of referencing external addresses. If there is a location counter reference in the constant subfield of an address constant and the duplication factor is greater than 1, then the value of the location counter is adjusted for each duplication of the constant.

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

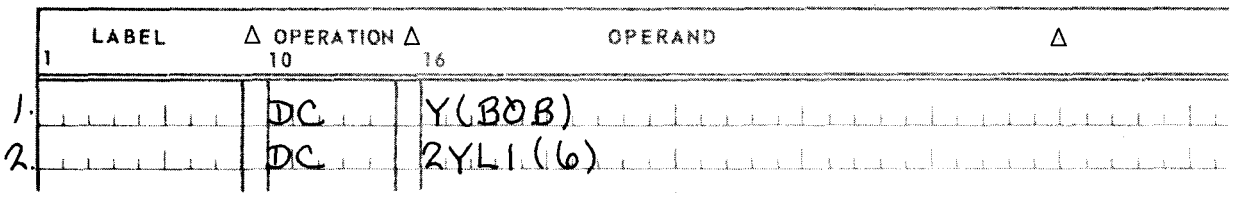
11.8.1. Half-Word Address Constants

A half-word address constant is specified by the character Y in the type subfield, and an expression enclosed by parentheses in the constant subfield. The expression may be absolute or relocatable. A length factor of 1 can be specified only for absolute expressions. Negative relocatable values are permitted. If no length factor is specified, the constant has an implied length of two bytes. If the length factor is present, the binary value is truncated or filled with binary 0's (on the leftmost end) to the length specified.

NOTE:

Y-type constants allow addressing of only the first 32K bytes of main storage. Due to the possibility of multiprogramming in an environment of greater than 32K, the use of A-type constants instead of Y-type constants is recommended.

Examples:



1. Assuming that BOB equals 1446, the hexadecimal value is stored as a 2-byte constant containing the following:

   05A6

2. A 2-byte constant is generated containing the following:

   0606

11.8.2. Full-Word Address Constants

A full-word address constant is specified by the character A in the type subfield and an expression enclosed by parentheses in the constant subfield. The expression may be absolute, relocatable, or complexly relocatable. A length factor of less than three bytes can be specified only for absolute expressions. Negative relocatable values are permitted. If no length factor is specified, the constant has an implied length of four bytes and is aligned to a full-word boundary. If the length factor is present, the binary value is truncated or filled with binary 0's (on the leftmost end), to the length specified, The maximum length that may be specified is four bytes. The maximum value that may be specified for a full-word constant is $2^{24}-1$.

Examples:

```
   LABEL      △ OPERATION △              OPERAND                        △
 1               10        16
1.               DC        A(SAM)
2.               DC        2AL1(X'416'+1)
```

1. Assuming that SAM equals 3863, the hexadecimal value is stored as a 4-byte constant containing the following:

   00000F17

2. A 2-byte constant is generated containing the following:

   1717

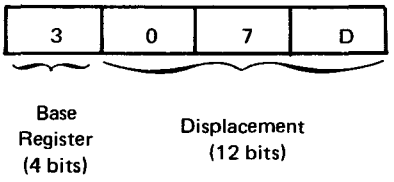
11.8.3. Base and Displacement Constants

A base and displacement constant is specified by the character S in the type subfield and one or two expressions enclosed by parentheses for each constant in the constant subfield. Only a length factor of 2 can be specified. If no length factor is specified, the constant has an implied length of two bytes and is aligned on a half-word boundary. 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 a constant is defined by a single expression, it may be either an absolute or a relocatable expression and the assembler converts it to a base plus displacement value. If two expressions are used to define a constant, 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. In this case, both expressions must be absolute. The S-type constants may not be specified as literals.

Examples:

```
   LABEL      △ OPERATION △              OPERAND                        △
 1               10        16
1. DON           START     512
                 .
                 .
2.               USING     DON+488,3
                 .
                 .
3. JOHN
                 .
                 .
4.               DC        S(JOHN)
5.               DC        S(125(3))
```

Assume that constant with the label JOHN (line 3) has been assigned an address value of 1125 by the location counter, and that the USING directive (line 2) gives the effective value 1000, which is assumed to be in register 3 at execution time (12.4.2). 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 307D:

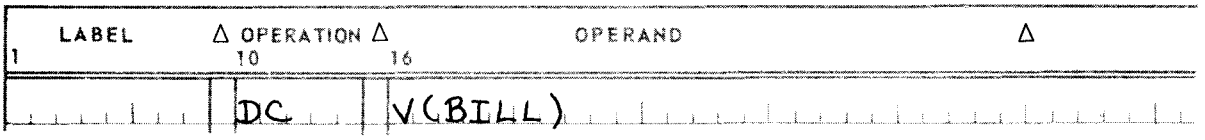


11.8.4. External Address Constants

An external address constant is specified by the character V in the type subfield and an external symbol enclosed by parentheses in the constant subfield. The constant cannot be used to reference external data. The symbol need not be identified by an EXTRN statement. A length factor of 4 is permitted. If no length factor is specified, the constant has an implied length of four bytes and is aligned to a full-word boundary. The specification of a symbol in the operand field of a V-type constant does not constitute a definition of that symbol. A V-type constant within a CSECT of reference is converted to an A-type constant.

Until the linkage editor replaces the hexadecimal representation in each byte with the correct value of the external labels, the value of each assembled constant is 0.

Example:



This DC statement generates the following constant:

00000000

11.9. CCW (DEFINE-CHANNEL-COMMAND-WORD) DIRECTIVE

The CCW directive defines and generates an 8-byte channel command word aligned on a double-word boundary. The channel command word is used to direct the operation of the multiplexer and selector channels. The format of the CCW directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	CCW	code,address, flags, count

where:

code	is an absolute expression specifying the command code.
address	is an expression specifying the address of the data. This value is assembled as a 3-byte constant.
flags	is an absolute expression specifying the flag bits.
count	is an absolute expression specifying the number of bytes to be transferred.

If a symbol appears in the label field, it is defined as equal to the address of the leftmost byte of the CCW and has a length attribute of 8.

All four operands must be specified and separated by commas.

Example:

LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16) Δ
	CCW	1,TED,X'40',50

where:

1	is the command code for a write operation.
TED	is the label of the data address.
X'40'	indicates a command chaining operation.
50	is the number of bytes to be transferred.

12. Assembler Directives

12.1. GENERAL

The SPERRY UNIVAC Operating System/4 (OS/4) Assembler directives are statements which enable the user to control assembler operation. These directives control the assembler at assembly time just as operation codes control the operation of the central processor at execution time. Assembler directives are represented by mnemonic codes entered in the operation field of a line of code. The directives are used to define symbols, adjust location counter values, control assembly I/O formats, section programs, provide boundary alignment, and assign base registers.

The following paragraphs describe the directives for the OS/4 assembler. The directives are arranged alphabetically within functional groups.

12.2. EQU (SYMBOL-DEFINITION) DIRECTIVE

The EQU directive is provided for symbol definition. It is used primarily for defining the length and value of a symbol. The format of the EQU directive is:

LABEL	Δ OPERATION Δ	OPERAND
symbol	EQU	e[,a]

where:

- e represents an absolute or relocatable expression.
- a represents an absolute expression.

All terms must be predefined.

The symbol in the label field is defined as having the value of the first expression in the operand field. This value must be of the range -2^{23} to $2^{24}-1$. If overflow occurs during evaluation of this expression, the directive is flagged. The symbol has a length attribute equal to the value of the second expression in the operand. The maximum value allowed for this expression is 256. This expression may be omitted, in which case the symbol is defined as having the length attribute of the first term in the first expression. If that term is * or a self-defining term, the length attribute is 1.

Examples:

LABEL	△ OPERATION △	OPERAND
1. TAG	DS	25CL10
2. HIDE	EQU	100+TAG,150
3. SEEK	EQU	TAG+1270-*
4. GO	EQU	TAG+1270-*,200

Assuming that the value of the location counter is 2000 when these lines of code are encountered, the symbols are assigned the following values:

1. TAG has a relocatable value of 2000 and a length attribute of 10. The location counter is advanced to 2250.

2. HIDE has a relocatable value of 2100 (100 + 2000) and a length attribute of 150. The location counter remains at 2250.

3. SEEK has an absolute value of 1020 (2000 + 1270 – 2250) and a length attribute of 10 (same as length of first term).

4. GO has an absolute value of 1020 (2000 + 1270 – 2250) and a length attribute of 200. (The 200 overrides the length of TAG.)

12.3. ASSEMBLY CONTROL DIRECTIVES

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

12.3.1. ASCII Directive

The ASCII directive is used to define ASCII constant generation and literals immediately following the directive, up to the recognition of the next mode directive. The format of the ASCII directive is:

LABEL	△ OPERATION △	OPERAND
unused	ASCII	unused

If no mode directive is used, EBCDIC constants are generated. For further information, see 3.6.1 in DOS interchange standards, UP-7902 (current version). Literal constants are generated according to the mode under which they are referenced rather than the mode for the region in which they are generated.

Example:

LABEL	△OPERATION△	OPERAND
	ASCII	

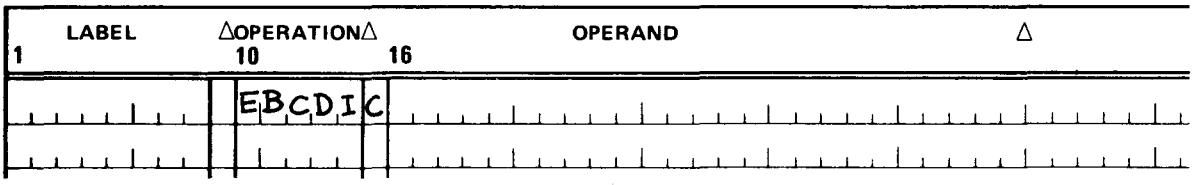
12.3.2. EBCDIC Directive

The EBCDIC directive is used to define EBCDIC constant generation immediately following the directive, up to the recognition of the next mode directive. The format of the EBCDIC directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	EBCDIC	unused

If no mode directive is specified, EBCDIC constants are generated.

Example:



12.3.3. CNOP (Conditional-No-Operation) Directive

The CNOP directive is used to adjust the location counter to a half-word, full-word, or double-word storage boundary. The format of the CNOP directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	CNOP	a,a

where:

a is an absolute expression.

The first expression in the operand field indicates a byte to which the location counter must be set. Legal values for the first expression are 0 and 2 for alignment relative to a full-word boundary and 0, 2, 4, and 6 for alignment relative to a double-word boundary. Zero indicates that the full-word or double-word boundary is desired; 2, the second byte (first half word) past the boundary; 4, the fourth byte (second half word) past a double-word boundary; and 6, the sixth byte (third half word) past a double-word boundary.

Permissible values for the second expression are 4 and 8, indicating that the adjustment is relative to a full-word or double-word boundary, respectively.

If the location counter is already set to the indicated byte, CNOP has no effect. When alignment is needed, one, two, or three no-operation instructions are generated to increment the location counter to the proper half-word boundary and to ensure correct instruction processing. All terms must be predefined.

Examples:

```
      LABEL     Δ OPERATION Δ      OPERAND              Δ
      1         10         16
1.                CNOP       0,8
2.                CNOP       2,4
```

1. The current location counter is advanced, if necessary, to the first byte of the next double-word boundary. A legal double-word boundary is any address value divisible by 8.

2. The current location counter is advanced, if necessary, to the second byte (first half word) past the next full-word boundary. A legal full-word boundary is any address value divisible by 4.

12.3.4. END (Program-End) Directive

The END directive indicates to the assembler the end of a source module or a procedure definition being assembled. The format of the END directive for program end is as follows; the format for Proc-Definition-End directive is described in 13.1.3.

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

where:

e is a relocatable expression.

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. The END directive must always be the last statement in the source module or procedure definition. If the operand field contains an expression, it designates the point in the program or in a separately assembled program where control may be transferred after the program is loaded. If the END directive is missing, an END directive with a blank operand field is supplied by the assembler.

Examples:

```
      LABEL     Δ OPERATION Δ      OPERAND              Δ
      1         10         16
1.    FOX         END        BEGN
2.                END        GO+324
3.                END
```

1. When the statement FOX is encountered, the assembly process is brought to an orderly halt. A transfer record is produced to identify the transfer address as the address of the instruction labeled BEGN. The label FOX receives an address value equal to the value of the location counter when the END statement is assembled.

2. If GO has a value of 1000, a transfer record with a transfer address of 1324 is generated.

3. A transfer record is generated with a transfer address equal to the first address loaded.

12.3.5. LTORG (Generate-Literals) Directive

The LTORG directive is used to generate all literals previously defined, but not generated, in the source module. The format of the LTORG directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	LTORG	unused

The literals are generated following the occurrence of the LTORG directive. A symbol coded in the label field represents the first byte of the generated literal pool. LTORG directives may not appear within a dummy control section or in a blank common storage area. If there are no LTORG statements in a program and literals are specified or if any literals are specified after the last LTORG directive in a program, these literals are generated at the end of the first control section. The programmer must then ensure that there is a valid base register available at all times to address the locations at the end of the first control section.

12.3.6. ORG (Specify-Location-Counter) Directive

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

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	ORG	[e]

where:

e is a relocatable expression.

The location counter is set to the value of the expression in the operand field. When the expression is not present, the location counter is set equal to a value one greater than the highest location previously assigned in the current section. If a symbol appears in the label field, its value is also the value of the expression in the operand field and it is assigned a length attribute of 1. The expression in the operand field must be a relocatable expression whose unpaired relocatable term must represent an address in the same control section in which the ORG occurs. The value must be equal to or greater than the initial setting of the current location counter. If the expression is in error, the ORG directive is ignored and the line is flagged. The ORG directive makes it possible to set the location counter to a value which is not a half-word boundary.

All terms in the expression must be predefined.

Bytes of storage reserved with a DS statement or an ORG directive are not set to 0 or cleared when the program is loaded.

Example:

```
LABEL      Δ OPERATION Δ      OPERAND                    Δ
1                10                16
AREA       ORG              *+A+B
```

This statement reserves A plus B bytes of storage, where A and B are previously defined symbols with absolute values. If A = 80, B = 160, and the value of the location counter is 1048, then 240 bytes are reserved beginning at the location 1048. The program may reference this 240-byte area by specifying AREA minus 240 as an operand.

12.3.7. START (Program-Start) Directive

The START directive defines the program name and tentative starting location. The format of the START directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	START	[a]

where:

a is an absolute expression.

All terms must be self-defining terms.

The expression in the operand field is evaluated and incremented, if necessary, to make it a multiple of 8. The result becomes the initial setting of the location counter for listing purposes and is the value of the symbol in the label field. This symbol is available as an entry point without being separately defined as such, and is the name assigned to the object module. The length attribute of this symbol is 1. If no name is assigned, the object module name is ASMOBJ00. The operand of the START directive is an absolute value but is treated as relocatable. Thus, the value of the location counter and the coding following a START directive are relocatable. The actual storage location at which the program is to be located is determined by the supervisor.

A START directive may be preceded by statements which do not alter or reference the location counter. If no START directive appears within the program, an invalid one is encountered, or the operand field is left blank, and the program is assembled relative to 0. If the label field is blank, an unnamed control section is defined.

Example:

```
LABEL      Δ OPERATION Δ      OPERAND                    Δ
1                10                16
TEST       START            1064
TEST       START            X'422'
```

Either one of these statements results in the program being assigned to locations starting at 1064 and having the symbol TEST defined with the relocatable value 1064.

12.4. BASE REGISTER ASSIGNMENT DIRECTIVES

The assembler assumes the responsibility of converting storage addresses to base register and displacement values for insertion into 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.

12.4.1. DROP (Unassign-Base-Register) Directive

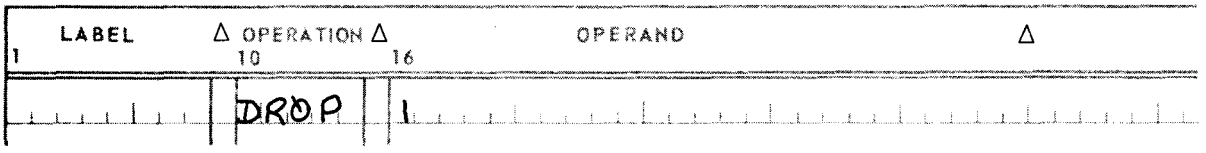
The DROP directive informs the assembler that the specified registers are not available for base register assignment. The format of the DROP directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	DROP	$r_1[,...,r_n]$

where each operand is an absolute expression which specifies by number (0 through 15) a register which is no longer available.

Registers previously made available for base register assignment can be dropped and registers can be made available again (in a USING directive) after they have been dropped. The value which is assumed to be in a base register can be changed by coding another USING directive without an intervening drop of that register.

Example:



This statement specifies that register 1 is no longer available to the assembler.

12.4.2. USING (Assign-Base-Register) Directive

The USING directive informs the assembler that a specified register is available for base register assignment in operand addresses and that it will contain a specific value at execution time. The value must be loaded by the problem program into the registers specified by the USING directive. The format of the USING directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	USING	$v,r_1[,...,r_n]$

where:

v — is an expression which gives the value that is assumed to be in the specified registers at execution time. This value may be relocatable or absolute. Literals are not permitted.

$r_1[,...,r_n]$ — are absolute expressions specifying the numbers (0 through 15) of the registers into which the value or modified values are placed. These register numbers do not necessarily have to be assigned in ascending sequence.

The first register specified after the expression (v) is assigned the value of v, the next register is assigned the value of the first register plus 4096, the next register is assigned the value of the second register plus 4096, and so on through all the registers specified. A USING directive may specify a single register or a group of registers, or the registers may be specified by individual USING directives.

The only addresses that may be covered by the registers indicated in a USING statement are those in the same control, dummy, or common section as the address represented by the first expression of the operand field of the USING statement.

Register 0 may be specified as a valid base register. However, the assembler assumes that is always contains the value 0. Any program using register 0 as a base register is not relocatable. Register 0 must be operand r_1, and any register specified in the operand field following register 0 is assumed to contain increments of 4096 from 0. If cover by Register 0 is desired for load address instructions and register 0 is not specified in a USING statement, cover error messages can be avoided through the use of the RO$ PARAM statement (E.2.7.).

When the expression v is absolute, the indicated registers can be used to process only absolute effective addresses.

When the expression v is relocatable, the indicated registers can be used to process only relocatable effective addresses. The registers r_1,... are used to process only those addresses in the same control section as the address represented by the expression v.

The value specification in a USING directive sets the lower limit of an address range. The upper limit of the range is automatically set 4095 bytes above the lower limit. The upper limit of a USING directive may be set less than 4095 bytes by being overlapped by the lower limit of another USING directive.

The range specified by a USING directive is used by the assembler to assign base register and displacement values to those effective operand addresses that fall within that range.

If an operand address is specified as an effective address instead of a base register and displacement specification, the assembler searches the USING table for a value yielding a displacement of 4095 or less. If there is more than one such value, the value that yields the smallest displacement is chosen. If no value yields a valid displacement, the operand address is set to 0 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.

Examples:

```
      LABEL     Δ OPERATION Δ        OPERAND                    Δ
      1         10          16
1.              USING  4000,8
2.              USING  8000,1,2,3,6,7,8,12
3.              USING  *,10
```

1. A range of 4096 bytes is specified beginning at location 4000 and ending at 8095. A value of 4000 is assumed to be stored in register 8.

2. The value 8000 is assumed to be in register 1, 12096 in register 2, 16192 in register 3, 20288 in register 6, 24384 in register 7, 28480 in register 8, and 32576 in register 12. These register numbers and their assumed values are entered into the USING table in the order specified.

3. This statement indicates to the assembler that the current value of the location counter is to be in register 10 at object time. (This is a relocatable value. A label can be used in place of a location counter.)

12.5. PROGRAM LINKING AND SECTIONING DIRECTIVES

A program or a portion of a program that is assembled as a single unit is called a module. A complex program can be made up of many modules, some of which are standard subroutines that can be used in any program.

The assembler provides, as part of its output, information which allows these modules to be linked together, loaded, and then executed as a single program. Proper partitioning or sectioning reduces the execution time required to make changes to an existing program. If a change is required, only that module which is changed must 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 externally defined (by an ENTRY directive) in module A and referenced (by an EXTRN directive) in module B. Thus, by using the ENTRY and EXTRN directives, proper linkage is supplied when the separate modules are assembled. This information is passed to the linkage editor by the external definition records and the external reference records which are outputs of the assembler.

The assembler also provides the capability of dividing one module into different sections. A control section is a group of instructions, constants, and storage areas. The proper execution of an instruction in one section must not depend on its position relative to instructions or data in another section. Sections can appear in the input in any order, and statements belonging to one section may be separated by statements belonging to one or more other sections. If the first statement is a START directive, its label becomes the name of the first control section.

Each module can have a maximum of 255 external symbol identification items (ESID items). An ESID item contains special information that is used by the linkage editor in relocating modules and module sections, and in resolving references between modules. The following items cause the assembler to generate an ESID item:

- each symbol in the operand field of an EXTRN directive;
- each symbol used in an external address (V-type) constant;
- each control section;
- each dummy control section; and
- a common storage definition section.

12.5.1. COM (Common-Storage-Definition) Directive

The COM directive enables the programmer to define a control section which is a storage area common to two or more separately assembled routines. The format of the common section can be described by DS and DC directives. Labels which appear within the sections are defined. No data or instructions are assembled in a common section which has a separate location counter with an initial value of 0. Data may be entered into a common section only by execution of a program which refers to it. Labels defined in a common section are not subject to the restrictions imposed on dummy section labels.

One assembly can define only one common section. However, several COM directives may appear among the source statements. Each COM directive after the first is taken to define a continuation of the common section previously described. When several routines defining common storage are linked, the resulting module contains only one section corresponding to the common section previously described. When several routines defining common storage are linked, the resulting module contains only one section corresponding to the common sections in the input modules. The length of this section is the length of the largest common section in the input modules. The format of the COM directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	COM	unused

Examples:

PROGRAM MODULE 1 PROGRAMMER

	LABEL	Δ OPERATION Δ	OPERAND	Δ
	1	10	16	
1.	MOD1	CSECT		
		.		
		.		
		.		
2.		COM		
		DS		
		DS		
		END		

PROGRAM MODULE 2 PROGRAMMER

	LABEL	Δ OPERATION Δ	OPERAND	Δ
	1	10	16	
3.	MOD2	CSECT		
		.		
		.		
		.		
4.		COM		
		DS	CL260	
		END		

1. When module 1 is assembled, it uses the common storage area defined by line 2.
2. The common storage area used by module 1 and module 2.
3. When module 2 is assembled, it also uses the common storage area defined by line 2.
4. The common storage area used by module 1 and module 2.

12.5.2. CSECT (Control-Section-Identification) Directive

The CSECT directive indicates to the assembler that the source statements which follow belong to a control section diffent from other preceding source statements. Use of the CSECT directive allows the programmer to code parts of logical sections of a program in the order in which he encounters 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 a previous CSECT statement, the succeeding statements are a continuation of the control section of that name.

The use of CSECT eliminates the need to discontinue coding a section of a program in order to code another section upon which the original section is dependent by setting up the new section with CSECT and continuing. After the second logical section is coded or even partly coded, the programmer can revert to the original section by setting up CSECT 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	COMMENTS
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)		

- 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.

  NOTE:

  Care must be taken to prevent linking CSECT and COM directives with duplicate names. A blank label field is permitted as a legitimate label only once. When both CSECT and COM directives have no labels, the label used for both is a blank; however, only one such label is permitted.

  – USING directive values between sections which define the same register must be redefined when each section is reentered.

12.5.3. DSECT (Dummy-Control-Section-Identification) Directive

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 in 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 have a blank label field in a program which either has no START statement or has 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.

- To guarantee alignment between the actual storage area and the dummy control section, the programmer should align the storage areas on doubl-word boundaries. All dummy control sections are adjusted to begin at location 0.

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

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

Example:

PROGRAM MODULE A PROGRAMMER

LABEL	OPERATION	OPERAND
BELL	START	1065
	ENTRY	AREA
	.	
	.	
AREA	DS	CL260
	.	
	END	

PROGRAM MODULE B PROGRAMMER

LABEL	OPERATION	OPERAND
EASE	START	2095
	EXTRN	AREA
	.	
LAKE	DC	Y(AREA)
	L	9,LAKE
	USING	SAIL,9
	.	
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 two bytes, and FLDB, containing four 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.

12.5.4. ENTRY (Externally-Defined-Symbol-Declaration) Directive

Each module must declare to the assembler the symbols defined within the module to which reference is made by other modules. Each symbol is referred to as being externally defined and is declared by the ENTRY directive. The format of the ENTRY directive is:

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

Each symbol in the operand field is declared to be defined in this module. Their name and assigned values are included in the output of the assembler as external definition records. The maximum number of operands in an ENTRY Directive statement is nine. Continuation is not allowed.

12.5.5. EXTRN (Externally-Referenced-Symbol-Declaration) Directive

The assembler must be informed of all symbols referred to in the module being assembled but defined in some other module. A reference to such a symbol is called an external reference, and such symbols are declared in the EXTRN directive. The format of the EXTRN directive is:

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

Each symbol in the operand field is declared to be a symbol defined in some other module. The symbolic name and the external symbol identification assigned by the assembler are included as input to the linkage editor as an external reference record. Each symbol is assigned a unique ESID and, therefore, cannot be paired with another symbol in an expression.

Examples:

PROGRAM MODULE A PROGRAMMER

LABEL (1)	Δ OPERATION Δ (10)	OPERAND (16)	Δ
FOX	MVO	DEST(5),ORIG(3)	
	DC	A(CAT)	
	DC	A(DOG)	
JOE	BC	8,1048	
	•		
	•		
	•		
MAT	BCT	10,SET	
	DC	A(PIG)	
	•		
	•		
	•		
	ENTRY	FOX,JOE,MAT	
	EXTRN	CAT,DOG,PIG	

Coding continued:

PROGRAM MODULE B PROGRAMMER

LABEL	OPERATION	OPERAND
	DC	A(FOX)
	.	
	.	
	.	
CAT	PRINT	DATA
	DC	A(JOE)
DOG	.	
	DC	A(MAT)
	.	
	.	
	.	
PIG	AU	6,UNOR
	ENTRY	CAT,DOG,PIG
	EXTRN	FOX,JOE,MAT

In module A, the symbols FOX, JOE, and MAT are specified with the ENTRY directive so that they may be used in module B as specified by EXTRN.

In module B, the symbols CAT, DOG, and PIG are specified with the ENTRY directive so that they may be used in module A as specified by EXTRN.

An externally defined symbol may be an absolute value which forms the explicit base/displacement specification of an instruction.

Examples:

PROGRAM MODULE A PROGRAMMER

LABEL	OPERATION	OPERAND
	EXTRN	OUT,C
	MVC	OUT(30,C),=CL30'ETC'

PROGRAM MODULE B PROGRAMMER

1 LABEL	△OPERATION△ 10	16 OPERAND △
OUT	EQU	LOC-START
C	EQU	12
	ENTRY	OUT,B

In module B, OUT is defined as the absolute displacement of the desired location under the explicit cover register C which is also defined in module B.

The only valid EXTRN symbol references, other than the base/displacement specification, are those used in address constants.

12.6. LISTING CONTROL DIRECTIVES

One of the outputs of the assembler process is a listing of source and object codes. Assembler directives are available to control the format of the listing. Their functions are:

- to provide headings for each page;
- to eject or skip to a new page;
- to space for extra blank lines; and
- to provide for printing or nonprinting of the output.

12.6.1. EJECT (Advance-Listing) Directive

The EJECT directive causes the assembler to advance to the next page for continued listing. The format of the EJECT directive is:

LABEL	△OPERATION△	OPERAND
unused	EJECT	unused

If the next line of the listing were to cause a page change, the EJECT directive has no effect.

When the EJECT directive is encountered, the form is skipped to the next page. If a title has been previously specified, the title is printed on the new page.

12.6.2. PRINT (Listing-Content-Control) Directive

The PRINT directive enables the programmer to control the contents of the assembly listing. The format of the PRINT directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	PRINT	[{ON / OFF}][,{GEN / NOGEN}][,{DATA / NODATA}]

where:

ON specifies the printed listing.

OFF specifies that no listing is printed.

GEN specifies that lines generated by a macro instruction are printed.

NOGEN specifies that lines generated by a proc call are not printed, except that the proc call, any PNOTE messages generated, and generated lines that contain error flags are printed.

DATA specifies that all characters of each constant representation are printed.

NODATA specifies that only the first eight characters of each constant representation are printed.

If a PRINT directive specifies OFF and also other parameters, the other specifications are not effective until a PRINT directive is encountered which specifies that the listing facility is to be turned ON.

In this directive, the comma is not required if a parameter is omitted. The initial print condition of assembly printing is ON, GEN, NODATA. This condition remains until the first PRINT directive changes it. PRINT directives may change only one or two of the parameters; any unspecified parameters remain in their previous condition. A PRINT directive may not appear in a procedure definition.

Any program statement or instruction that produces an assembly error condition is listed regardless of specified PRINT options.

Examples:

```
      LABEL     Δ OPERATION Δ     OPERAND                    Δ
      1         10                16
1.                PRINT  DATA
2.                PRINT  OFF
3.                PRINT  ON,GEN,DATA
```

1. Data is printed in full.
2. All printing is suppressed except lines of coding which produce error conditions.
3. Full printing is restored with complete printing of data constants.

12.6.3. SPACE (Space-Listing) Directive

The SPACE directive causes the assembler to advance the paper in the printer a specified number of lines. The operand field contains an unsigned decimal integer which specifies the number of lines the paper is to be advanced. If no operand is coded, one line is spaced. If the number specified is greater than the number of lines remaining on the page, the SPACE directive has the same effect as an EJECT directive. The SPACE directive does not appear on the listing.

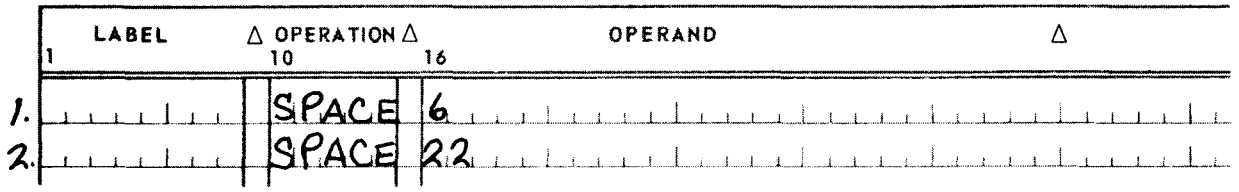
The format of the SPACE directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	SPACE	[i]

where:

i is an unsigned decimal integer.

Examples:



1. The printer advances the form six lines before printing the next line.
2. The printer advances the form 22 lines before printing the next line.

12.6.4. TITLE (Listing-Title-Declaration) Directive

The TITLE directive provides data for the heading appearing at the top of each page of the assembler listing. A TITLE directive also causes the printer form to be advanced to a new page. The format of the TITLE directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	TITLE	'c'

where:

c is up to 100 characters of heading.

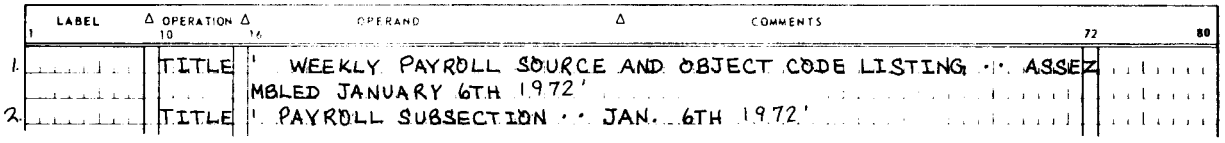
The following conditions apply to characters in the operand field:

- Any character may be specified, including spaces, within the defining apostrophes.
- An apostrophe within the operand must be specified as a pair of apostrophes.
- An ampersand within the operand must be specified as a pair of ampersands.
- Spaces may be specified freely to separate heading words.

More than one TITLE directive is permitted in a program. A TITLE directive provides the heading for all pages in the listing which succeed it.

The first TITLE card in the program may have a special symbol in the label field (one to four alphanumeric characters in any order) which is used as a program identification on the listing.

Examples:



1. The Z in column 72 specifies that the title is continued on the next line.
2. The second title line indicates a change in the page heading, and the page headings are specified in the second title line.

12.7. INPUT AND OUTPUT CONTROL DIRECTIVES

The assembler input and output control directives provide the necessary control for sequence checking, formatting, punching data, and reproducing data.

12.7.1. ICTL (Input-Format-Control) Directive

The ICTL directive specifies new values for the beginning, ending, and continuation coding columns. The format of the ICTL directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	ICTL	[b] [,e] [,c]

where:

b is an unsigned decimal integer specifying the beginning column. It must be less than 80.

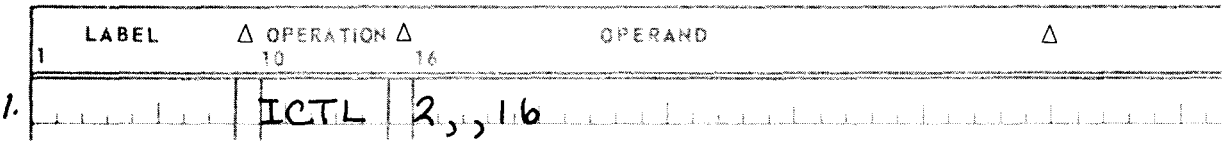
e is an unsigned decimal integer specifying the ending column. It must be greater than b and less than or equal to 80.

c is an unsigned decimal integer specifying the continuation column. It must be greater than b and less then e.

If b is omitted, it is assumed to be 1. If e is omitted, it is assumed to be 71. If c is omitted or if e equals 80, continuation records are not allowed.

There can be only one ICTL directive in a source code module and it must immediately precede or follow any program-defined procedure definitions. The ICTL directive applies only to those source statements that follow it. All procedure definitions are assumed to have normal output format.

Example:



1. Coding is to follow standard format except that it is to start in column 2.

12.7.2. ISEQ (Input-Sequence-Control) Directive

The ISEQ directive specifies to the assembler which columns of the source statement contain the field used for checking the sequence of statements. It also controls the initiation and termination of sequence checking. The format of the ISEQ directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	ISEQ	[l,r]

where:

l is a decimal integer specifying the leftmost column of the field to be used for the sequence check.

r is a decimal integer specifying the rightmost column of the field to be used for the sequence check. r must be greater than or equal to the specification for l.

Columns to be checked should not fall between the beginning and ending input columns specified for the program.

The sequence check begins with the first source statement after the first ISEQ directive and is terminated by an ISEQ directive with a blank or invalid operand field.

Sequence checking is not performed on statements generated from procedure definitions.

Example:

LABEL	△ OPERATION △	OPERAND △
1.	ISEQ	56,60

1. Input record sequence is to be checked using the sequence numbers found in columns 56 through 60.

12.7.3. PUNCH (Produce-a-Record) Directive

The PUNCH directive is used to produce specified data in the object code output of the assembled program. The format of the PUNCH directive is:

LABEL	△ OPERATION △	OPERAND
unused	PUNCH	$'c_1,\ldots,c_{80}'$

where:

$c_1 \ldots, c_{80}$ represents a string of up to 80 characters produced as a record in the object code output.

The following conditions apply to the characters specified in the operand field:

- Up to 80 characters including spaces, may be specified within the enclosing apostrophes.
- An apostrophe within the operand must be specified as a pair of apostrophes.
- An ampersand within the operand must be specified as a pair of ampersands.
- Spaces may be used to separate fields.
- In counting characters for the limit of 80, a pair of ampersands or apostrophes written to express a single ampersand or apostrophe counts as one character.

Although this directive may be included within a procedure definition, it may not occur before or between the procedure definitions. It may be written after the procedure definitions, but prior to the first control section of the program. PUNCH directives thus written produce records prior to the object module.

Example:

LABEL	△ OPERATION △	OPERAND △	COMME
	PUNCH	'THIS RECORD APPEARS IN THE OUTPUT'	

12.7.4. REPRO (Reproduce-Following-Record) Directive

The REPRO directive is used to reproduce a record in its entirety (columns 1 through 80) during assembly time. The format of the REPRO directive is:

LABEL	△ OPERATION △	OPERAND
unused	REPRO	unused

This directive causes the contents of the following record on the coding form to be produced as a record in the assembler output. Each REPRO directive produces one record. A maximum of 80 bytes are reproduced.

A REPRO directive prior to the first control section of the program produces records prior to the first control section.

No substitution for variable symbols occurs in the record thus produced. This directive cannot appear in a macro definition.

12.8. CONDITIONAL ASSEMBLY

The assembler recognizes certain directives which 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, or establish and alter values which may be used to determine whether a set of lines shall be included or excluded.

These directives are known as conditional assembly directives. While they are frequently used within procedure definitions, they can be effectively used at the basic assembly level.

12.8.1. SET Directive

The SET directive is used to define or redefine the value represented by set symbols. A set symbol is a symbol to which a value is assigned during the generation of code corresponding to procedure references and DO directives. It can be used as a counter or as a switch to control the generation of code. Unlike an ordinary symbol, the value assigned to a set symbol can be altered during the course of an assembly.

A set symbol can be either local or global. A global set symbol, 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 set symbol is defined only within the procedure definition in which it is declared. The value of a local set symbol within one procedure definition is not affected by the declaration of either a local or global set symbol with the same name in another procedure definition or at the source code level.

Before a set symbol may be set or referenced, it must first be declared by a GBL or an LCL directive.

The format of the SET directive is:

LABEL	△ OPERATION △	OPERAND
symbol	SET	b

where:

b is a basic expression (2.5.5).

The symbol in the label field is assigned the value represented by the basic expression in the operand field. Only a basic expression may be used.

Operand expressions cannot be greater in value than $+2^{24}-1$. Set symbols may represent character strings up to eight characters in length; also, symbols originally set to an arithmetic value can be redefined to represent a character string of up to eight characters.

12.8.2. LCL (Local-Symbol-Declaration) Directive

The LCL directive is used to declare and initialize local set symbols before they are defined or referenced. The format of the LCL directive is:

LABEL	△ OPERATION △	OPERAND
unused	LCL	symbol[,symbol,...,symbol]

Each symbol which appears in the operand field is declared to be a local set symbol and is set equal to a null character string.

Although the LCL directive is primarily for use within procedure definitions, it may be used at the basic source code level to declare set symbols which may be referenced only at the source code level.

12.8.3. GBL (Global-Symbol-Declaration) Directive

The GBL directive is used to declare that a symbol is a global set symbol.

The format of the GBL directive is:

LABEL	△ OPERATION △	OPERAND
unused	GBL	symbol[,symbol,...,symbol]

Each symbol which appears in the operand field is declared to be a global set symbol, and is set equal to the null character string when declared for the first time. Declaring, as a global set symbol, a symbol which has already been defined as a global set symbol does not affect its value.

Examples:

```
                LABEL     Δ OPERATION Δ          OPERAND                    Δ
                1              10              16
                          START
                          .
                          .
                          .
1.                        GBL       WALT
2.                        LCL       TED
                          .
                          .
3.              WALT      SET       1
                          .
                          .
4.              TED       SET       WALT*432+X'7E'
                          .
                          .
5               WALT      SET       2
                          .
                          .
6.              TED       SET       'CX1&&144'
                          .
                          .
                          .
                          END
```

1. Declares the label WALT to be a global set symbol.

2. Declares the label TED to be a local set symbol.

3. Defines the value labeled WALT.

4. Defines the value labeled TED.

5. Redefines the value labeled WALT.

6. Redefines the value labeled TED.

12.8.4. DO (Start-of-Range) Directive

The DO directive defines the start of the range of code to be generated repetitively and specifies the number of times it is to be generated. The format of the directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DO	b

where:

b is a basic expression (2.5.5).

The expression in the operand field indicates the number of times the source code statements following the DO directive are to be produced in the object code. All lines of coding following the DO directive, until its associated ENDO directive is encountered, are generated. The value of the expression in the operand field may be any positive value or 0.

Any valid source code statement may be within the range of a DO directive including other DO directives with their corresponding ENDO statements. DO directives may be nested up to 10 levels.

The symbol in the label field, when specified, is used as a counter for the number of times a set of lines within the range of a DO statement have been generated. Its value is 1 the first time through the statements, 2 the second time through the statements, and so forth. It may be referenced in the same manner as a set symbol.

If a DO directive is within the range of another DO directive and the nested DO directive is reentered, its count begins at 1 again. The value of the label of the DO directives is available to the statements following the ENDO directive even if the operation of the DO directive cycle is interrupted.

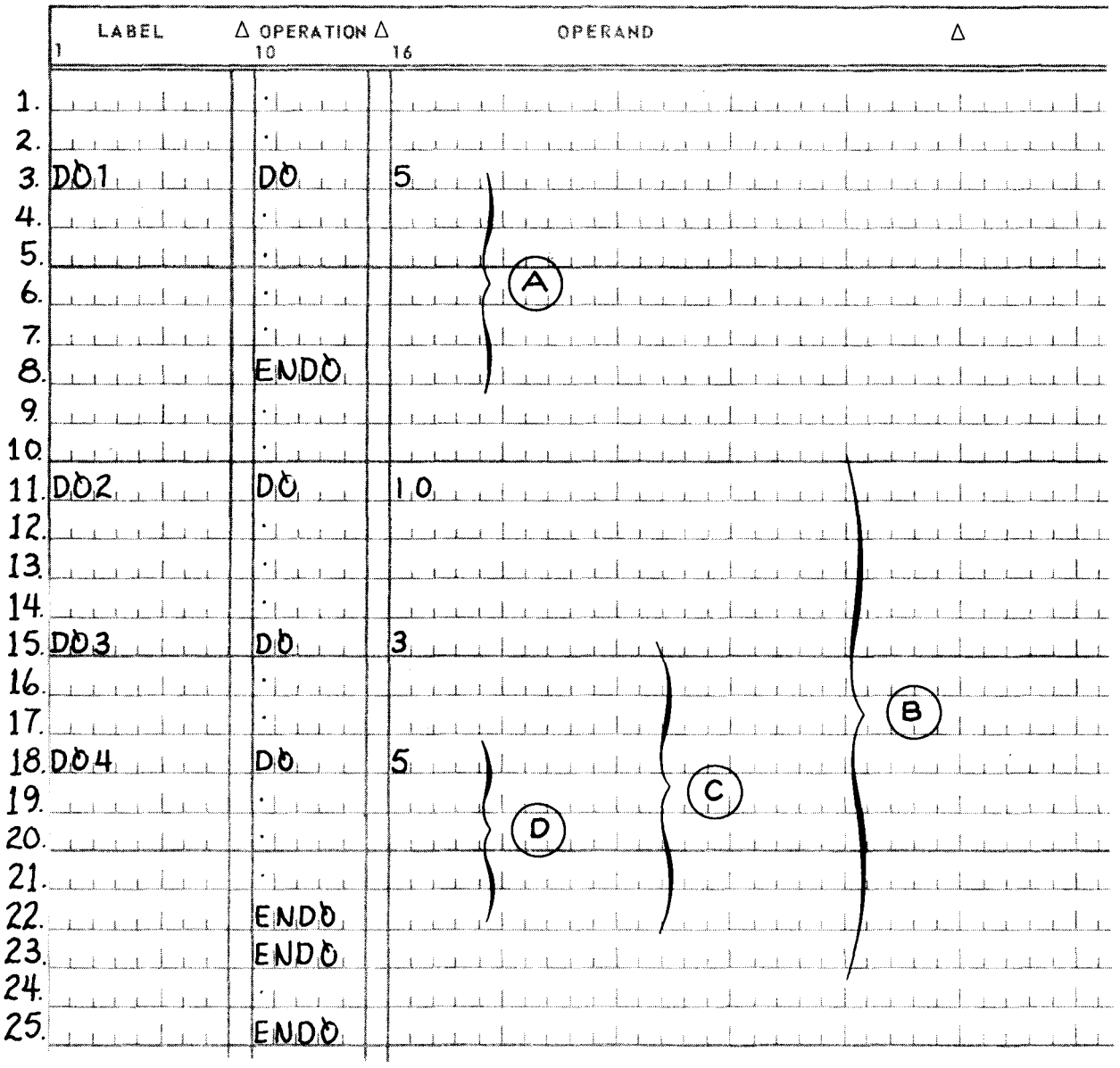
12.8.5. ENDO (End-Range-of-DO) Directive

The ENDO directive is used to signal the end of the range of a DO statement. The ENDO directive has the following form:

LABEL	Δ OPERATION Δ	OPERAND
unused	ENDO	unused

For every DO directive there must be an ENDO directive to define the range.

Examples:



(A) Lines 4, 5, 6, and 7 following the first DO directive (on line 3) are produced in the output five different times. The ENDO directive on line 8 signals the end of the lines of coding to be generated.

(B) Lines 12, 13, 14, 24, and the lines produced by the operation of the two DO directives (lines 15 and 18) are generated in the output 10 different times.

(C) Within each of the 10 sets produced by the DO directives on line 11, lines 16 and 17 and the lines generated by the operation of the DO directive on line 18 are produced in the output three different times.

(D) Within each of the 30 sets produced by the two DO directives (lines 11 and 15), lines 19, 20, and 21 are generated in the output five different times.

NOTE:

The first DO directive produces 20 lines of output coding. The second, third, and fourth DO directives combined produce 550 lines of coding.

12.8.6. GOTO (Assembly-Branch) Directive

The GOTO statement is used to direct the assembler to another point in the source code. The form of the GOTO directive is as follows:

LABEL	Δ OPERATION Δ	OPERAND
unused	GOTO	symbol

The symbol in the operand field specifies the LABEL directive at which the assembler should resume processing the source code. The following rules and conditions must be observed when using the GOTO directive:

- The symbol used must be identical to the symbol in the label field of the LABEL directive.
- A GOTO directive within a section of basic source code may not indicate a destination within a procedure definition.
- A GOTO directive within a procedure definition may not specify a destination within another procedure definition and it may not specify a destination within a section of basic source code.
- A GOTO statement, within the range of a DO directive that specifies a LABEL directive that is not within the same range, interrupts the operation of the DO directive and continues source code processing at the LABEL statement.
- A GOTO directive may specify a destination point either forward or backward in the source code.
- A period is allowed as the first character of the operand. The period will not appear on the assembly listing.
- If the symbol in a GOTO statement is not satisfied by a LABEL statement, the assembler falls through to the next line of source code.

12.8.7. LABEL (Assembly-Destination) Directive

The LABEL directive is used to identify a destination point for the GOTO directive only.

The format of the LABEL directive is as follows:

LABEL	Δ OPERATION Δ	OPERAND
symbol	LABEL	unused

The symbol specified in the label field of the LABEL directive is not defined by the assembler in the usual way; instead, the assembler maintains a special list of LABEL directive symbols which are the only valid destination points for the GOTO directive.

The GOTO directive may not specify a label that has not been defined by a LABEL directive.

A period is allowed as the first character of the label field of a LABEL statement. The period will not appear on the assembly listing.

Example:

	LABEL	OPERATION	OPERAND
	1	10	16
1.			
2.			
3.		GOTO	BREAK
4.	RETURN	LABEL	
5.			
6.			
7.		GOTO	CONTINUE
8.	BREAK	LABEL	
9.			
10.			
11.		GOTO	RETURN
12.			
13.			
14.	CONTINUE	LABEL	
15.			
16.			

The GOTO directive on line 3 provides an unconditional branch to line 8. Lines 9 and 10 and processed by the assembler; then line 11 specifies a GOTO to line 4, where lines 5 and 6 are processed by the assembler. Line 7 specifies a GOTO to line 14 where the assembler continues processing at line 15.

13. Assembler Procedures

13.1. SPECIAL DIRECTIVES

The SPERRY UNIVAC Operating System/4 (OS/4) Assembler, by the use of special directives and conditional assembly directives (12.8), allows the programmer to specify and generate repetitive sequences of coding. To save time and effort required to write a series of instructions repeatedly and to eliminate possible errors in transcription, the series can be written once in a procedure definition. A procedure definition (proc) is a series of one or more assembler statements beginning with a PROC directive, followed by one or more NAME directives, and ending with an END directive. The PROC directive signals the beginning of a procedure definition. The NAME directive declares a label by which the procedure definition can be referenced. The END directive signals the end of the procedure definition. Each time the instructions are needed, a procedure call line is written. The assembler inserts 0 or more lines of coding at the point of reference.

The procedure definition specifies to the assembler the coding and instructions for a particular operation, and the procedure call line specifies the variable parameters. The assembler then combines the coding of the procedure with the parameters to produce a specific section of coding.

13.1.1. PROC (Procedure-Definition) Directive

The procedure definition is introduced into the source program by the PROC directive. This directive is used to signal the beginning of a procedure definition. The format of the procedure definition is:

LABEL	△ OPERATION △	OPERAND
[symbol]	PROC	[{ s / s,n / s,n,k / s,n,k,... }]

where:

- s represents the name or symbol in the operand field to be used in referencing parameters.
- n is a decimal, self-defining term that represents the maximum number of positional parameters that are found in the proc call line.
- k represents keyword parameters.

If the label field contains a valid symbol, it represents the label of the proc call line (13.4).

The operand field specifies the names to be used when referring to parameters in the source code statement of the proc call line. The first subfield (s) must contain a valid symbol if any parameters are to be referenced, and the second subfield (n) must contain a number that indicates the maximum number of positional parameters that can be specified in the proc call line if any keyword parameters are to be referenced. The remaining subfields specify names used in referencing keyword parameters in the body of the procedure coding.

A method is provided whereby the programmer can preset the value of a keyword. This preset value is automatically used if the particular keyword is not specified; however, the preset value can be changed or overridden by specifying a new value for that keyword. A predefined keyword appears in the operand field of the PROC directive as follows:

k=v

where:

k represents a symbol or name which is used to identify the parameter.

v represents the preset value.

13.1.2. NAME (Call-Label) Directive

The NAME directive specifies a name by which the procedure is referenced. The format of the NAME directive is:

LABEL	△OPERATION△	OPERAND
symbol	NAME	[p]

where:

p is a parameter or a parameter sublist.

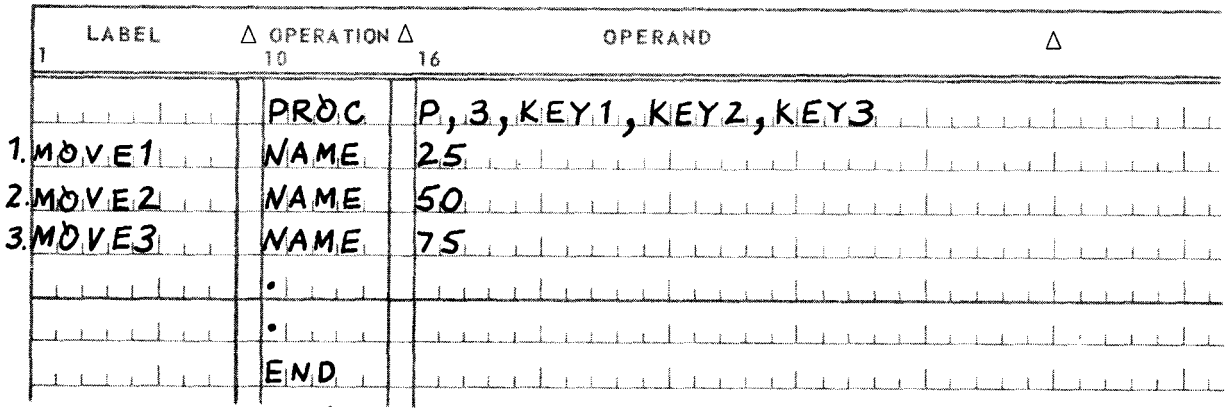
The first NAME directive must immediately follow the PROC directive. More than one NAME directive may be coded but all must be at the beginning of the definition. Each such NAME directive specifies a different name for the proc. The symbol in the label field is available for reference within procs as well as at the source code level.

The operand field is used to provide a parameter to the proc. When more than one NAME directive follows the PROC directive, only the operand of the NAME directive whose symbol is used to reference the proc is available to the body of the definition.

Reference is made to the parameter or parameter sublist in the operand field by means of paraforms which are discussed in 13.3.1.1.

Multiple NAME directives allow the programmer to specify a different parameter for each NAME directive and to select the parameter by calling on that particular NAME directive. The following example lists three NAME directives; the proc can be called by any one of them.

Examples:



1. MOVE1 calls in the procedure and provides parameter 25.
2. MOVE2 calls in the procedure and provides parameter 50.
3. MOVE 3 calls in the procedure and provides parameter 75.

13.1.3. END (Proc-Definition-End) Directive

The END directive is used to signal the end of a proc, as well as the end of the source module. An END directive format is:

LABEL	Δ OPERATION Δ	OPERAND
unused	END	unused

The operand field should be blank. The assembler pairs each END directive with the most recently encountered PROC directive which is unpaired.

The statements between paired PROC and END directives are defined as the body of a proc. An END directive that cannot be paired signals the end of the module to be assembled.

13.1.4. PNOTE (Message) Directive

The PNOTE directive may be used within a proc or at the source code level to generate comments and/or error messages on the printer listing. The format of this directive is:

LABEL	Δ OPERATION Δ	OPERAND
unused	PNOTE	e,c

where:

e is a message character string.

c is a comments character string.

The PNOTE directive indicates to the assembler that a comment line is to be generated in the output listing. The first subfield of the operand must contain either an asterisk or a character expression of not more than six characters. The second subfield may contain a character expression of not more than 79 resultant characters. When the first subfield contains a character expression, the resultant characters appear on the listing as error flags and the resultant characters of the second expression are printed as a comment line. If the first subfield contains an asterisk, an asterisk is printed in the error flag field and the resultant characters of the second expression are printed as a comment line.

Any characters generated in the error flag field are treated as diagnostic errors. Message and error flags appear in the listing even when the NOGEN option of the PRINT directive is in effect.

13.2. CODING PARAMETERS

In order to activate a proc, certain information must be given to it at the time it is referenced. Each item of information is called a parameter and coded in a subfield on the line which calls a proc. A string of subfields separated by commas is called a field and is the operand field for a proc reference.

13.2.1. Types of Parameters

There are two types of parameters: positional and keyword. The distinction between the two is in the way they are identified. Positional parameters are identified by their position within the operand field of the call line. Keyword parameters are identified by the symbols which are assigned to them in the call line.

13.2.1.1. Positional Parameters

Positional parameters must be specified before the keyword parameters in any call line. The order of the expressions in the operand determines the order of the parameters specified. Positional parameter specifications are separated by commas. When a nontrailing positional parameter specification is omitted, the comma must be retained to indicate the omission. Thus if a proc call line has four positional parameters

v_1,v_2,v_3,v_v

and the second one is not specified, the operand would appear:

v_1,v_3,v_4

If the third and fourth parameters are not specified, the operand is written:

v_1,v_2

If only the last parameter is specified, the operand is written:

$,,,v_4$

Thus, preceding or intervening missing parameters must be indicated by a comma. Trailing parameters which are missing need not be indicated.

13.2.1.2. Keyword Parameters

Keyword parameters must follow the positional parameters, when both are used, on the call line. Keyword parameters need not appear in any specific order. Each keyword is equated to a symbol, value, or character string. Keywords are coded on the call line as follows:

$k_1=v_1,k_2=v_2,\ldots,k_n=v_n$

where:

k represents a symbol or name which is used to identify the parameter and

v represents the parameter value.

Keyword parameter specifications must be separated by commas and can appear in any order. Because a keyword parameter is identified by name and not by position, a comma must not be used to indicate a missing keyword parameter. A comma must separate the last positional parameter from the first keyword parameter when a combination of both is used.

If a PROC directive specifies three keyword parameters in the operand field

k_1,k_2,k_3

and if the call line specifies only two of the three with the second keyword parameter missing, the format is:

$k_1=v_1,k_3=v_3$ or $k_3=v_3,k_1=v_1$

or if the call line specifies only one of the parameters, the format is:

$k_3=v_3$

If the value of the missing keyword parameters has been preset in the PROC directive, then the preset values will be used in the called procedure. If values have not been preset, then the missing keyword parameters are set to a null character string.

13.2.1.3. Combined Positional and Keyword Parameters

Both positional and keyword parameters can be specified in the proc call line. The following rules apply to the proc call line:

- In all cases all parameters are separated by commas.
- Positional parameters can be specified without keyword parameters, or keyword parameters can be specified without positional parameters, or a combination of both can be used.
- Preceding or intervening positional parameters which are missing must be indicated by a comma; trailing positional parameters which are missing need not be indicated.
- Omitted keyword parameters do not need a comma to indicate the omission.
- Keyword parameters can be specified in any order.
- An omitted keyword parameter that has been assigned a preset value receives the preset value in the procedure coding.
- An omitted keyword parameter without a preset value receives a value of a null character string.

13.2.2. Parameter Sublists

Each subfield in the proc call line can contain more than one parameter. Multiple parameters in either positional or keyword subfields are called parameter sublists. Each of the parameters in a parameter sublist is separated by commas, and the whole parameter sublist is enclosed by parentheses. A proc call line operand having a positional parameter sublist in the second subfield appears as follows:

LABEL	Δ OPERATION Δ	OPERAND
	proc-name	$v_1,(v_{2,1},v_{2,2}),v_3$

where each v represents a parameter. A keyword parameter sublist has the format:

LABEL	Δ OPERATION Δ	OPERAND
	proc-name	$k_n=(v_{n,1},v_{n,2},\ldots,v_{n,m})$

where:

k_n represents the symbol or name used to identify the keyword parameter sublist and

$v_{n,1},v_{n,2},\ldots,v_{n,m}$ represent the parameters within the parameter sublist.

13.3. REFERENCING AND REPLACING PARAMETERS AND SET SYMBOLS

A coordinate system is provided by the assembler for references to parameters coded on a proc call line. In addition, keyword parameters may be referenced by the keyword symbol alone. Parameter references may be coded in any statement within the proc. When the proc is referenced (called), the assembler replaces all parameter references with the information coded in the designated subfield of the proc call line.

In this section, the referencing and replacing of set symbols is compared to the referencing and replacing of parameters in order to show the similarities and differences. The basic term paraform is a parameter reference form. Thus, a paraform is a reference to the parameter or parameter sublist in the operand field of the NAME directive, or a reference to the parameters on the PROC call line.

13.3.1. Reference Formats

References to parameters and set symbol values are usually made by coding the paraform or the set symbol at the place where replacement is desired. However, delimiters must be coded in certain cases to separate the reference from the remainder of the statement.

Where replacement is desired in proc call line operand fields, in NAME directive operand fields, or within paired apostrophes, the reference must be preceded by a single ampersand.

- Any data constant specification
- Character self-defining terms
- Hexadecimal self-defining terms
- Binary self-defining terms
- Character-strings or substrings

A paraform or set symbol may be concatenated with other characters within the apostrophes. If the paraform or set symbol is followed by an alphanumeric character, a period, or a left parenthesis, a period (designating concatenation) must be coded after the reference.

When a period follows a set symbol or paraform, the period is discarded when replacement occurs. For example, if the set symbol ABC has the character value '5' the term

C'&ABC..7'

after replacement would yield

C'5.7'

A set symbol or paraform may also be concatenated with other characters, set symbols, or paraforms to form a single term outside of paired apostrophes. The reference may be coded exactly as it would have been to concatenate with the same preceding and following characters within paired apostrophes. However, the leading ampersand is required only if the character preceding the reference is alphanumeric. In this case a period could be used instead of the ampersand with the same results. However, it is strongly recommended that the leading ampersand always be written.

13.3.1.1. Paraforms

Two types of paraforms are used within the body of a procedure to reference the parameters on the proc call line: positional and keyword.

Positional paraforms may be used to reference any parameter on the call line. The format of a paraform is:

s(n)

where:

- s represents the name or symbol in the first subfield of the PROC directive operand field. This is the name used to reference parameters.
- n represents the numeric order of the positional parameters in the call line and/or the position of the keyword parameters in the operand of the PROC directive.

The parameters are identified by the numbers assigned to the subfield they occupy. Subfield s(0) is defined as the operand of the line on which the procedure name is specified. Subfields s(1) through s(n) are defined as the n positional parameters coded on the proc call line. The maximum number (n) of positional parameters to be expected for any procedure is coded in the second subfield of the PROC directive operand field. The next subfields, s(n+1) through s(n+m), correspond to the keywords shown in the operand field of the PROC directive.

Keyword paraforms may be used to reference keyword parameters only. A keyword parameter reference consists of the keyword itself coded in the statements where replacement is desired. No subfield is needed because the parameter value is identified by the keyword on the proc call line.

Example:

LABEL 1	Δ OPERATION Δ 10	OPERAND 16 Δ
	PROC	&BOB,3,&KEY1,&KEY2,&KEY3
TRANSFER	NAME	24
	•	&BOB(1),• • •
	•	• • • ,&BOB(2)
	•	• • • &(BOB(3)),• • •
	•	&KEY1(&KEY2),&KEY3
	END	

The preceding example specifies six variable parameters and one fixed parameter. The fixed parameter is defined when the procedure is written (in this case 24), and the six variables are defined &BOB(1), &BOB(2), &BOB(3), &KEY1, &KEY2, and &KEY3. The parameters represented by &KEY1, &KEY2, and &KEY3 could also be referenced by paraforms &BOB(4), &BOB(5), and &BOB(6), respectively.

These seven parameters can be referred to as paraforms and are referenced by:

&BOB(0) represents the fixed value on the NAME line.
&BOB(1) represents the first positional parameter.
&BOB(2) represents the second positional parameter.
&BOB(3) represents the third positional parameter.
&BOB(4) represents the first keyword parameter.
&BOB(5) represents the second keyword parameter.
&BOB(6) represents the third keyword parameter.

LABEL	Δ OPERATION Δ	OPERAND
	TRANSFER	4,,2,KEY3=7,KEY2=10

The preceding procedure call line allows replacement of the paraforms with these values:

&BOB(0) is 24.
&BOB(1) is 4, the first positional parameter.
&BOB(2) is a null character string.
&BOB(3) is 2, the third positional parameter.
&BOB(4) is a null character string.
&BOB(5) is 10, for the second keyword parameter.
&BOB(6) is 7, for the third keyword parameter.

Missing parameters are equated to a null character string. A procedure can have more than one name assigned by the NAME directives and each name can have a different value in the operand field. The different values can be selected by specifying the appropriate name in the call line.

If a subfield contains a parameter sublist, a different numbering system is necessary to reference its contents. A matrix notation system is used where the first number indicates the subfield and the second indicates the parameter within the subfield. A paraform referencing a parameter within a parameter sublist would have the form:

s(n,i)

where:

s represents the name or symbol used for referencing the parameters.

n represents the subfield number.

i represents the parameter number within the subfield.

In the previous example, if the operand field of the NAME directive had been

(24,17)

then:

&BOB(0,1) would be replaced by 24.
&BOB(0,2) would be replaced by 17.

When a parameter sublist is coded in a subfield and the subfield only is addressed, the paraform is replaced by all the characters in the sublist including the surrounding parentheses. Thus in the previous example where the NAME directive of the operand contained a sublist, the reference &BOB(0) would be replaced by:

(24,17)

If the operand had contained a sublist of three factors

(32,,12)

then:

&BOB(0) would be replaced by (32,,12)

Keyword paraforms may be used to reference parameters in a keyword parameter sublist in the format:

k(i)

where:

k represents the keyword symbol.

i represents the parameter number within the subfield.

Thus, if the following proc call line were coded

LABEL	Δ OPERATION Δ	OPERAND
	PCL	KEY1=(14,56)

then:

&KEY1	would be replaced by (14,56).
&KEY1(1)	would be replaced by 14.
&KEY1(2)	would be replaced by 56.

13.3.1.2. Set Symbols

The expressions and values which set symbols represent may be referenced in the assembly module anywhere they are defined. The set symbol itself is coded where the value is desired. The assembler replaces the symbol as explained in 13.3.2.2.

Although it is seldom necessary to precede set symbol references by an ampersand, it should be done to differentiate the set symbol from the other characters.

13.3.2. Replacement

References to parameters and set symbol values cause information to replace those references in the source code line. The format of the references and the rules for where references may occur are similar to those for keywords, set symbols, and paraforms. However, parameters are not replaced by the same method as set symbols.

Replacement is subject to the following limitations:

- Parameter and set symbol replacements may not be used to construct other parameter or set symbol references.
- Character substrings and the concatenation of character strings may not be used to construct parameter or set symbol references.
- Parameter and set symbol replacements and character manipulation may not be used to construct the following directive mnemonics:

  SET, DO, ENDO, PROC, NAME, END, ISEQ, ICTL

- Assembler directive and operation code mnemonics should not be used as set symbols, keyword names, PROC directive labels, or DD statement labels.
- Parameter and set symbol replacement may not be used to construct the type subfield or the character L in DC, DS, or literal operand fields.

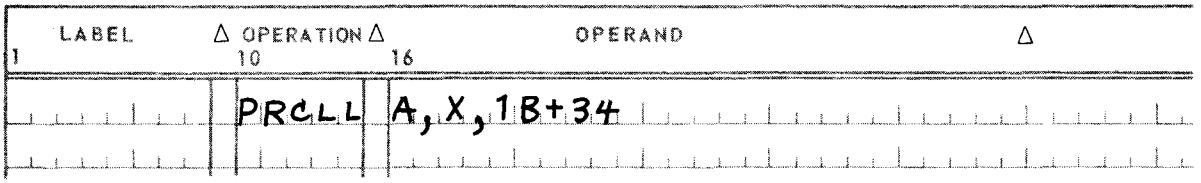
13.3.2.1. Parameter Replacement

Parameters which appear in the operand field of a proc call line or a NAME directive are not evaluated. Parameter references and set symbol references that appear on a proc call line are replaced but are not analyzed. Information coded as a parameter is treated as a collection of one or more characters delimited by commas.

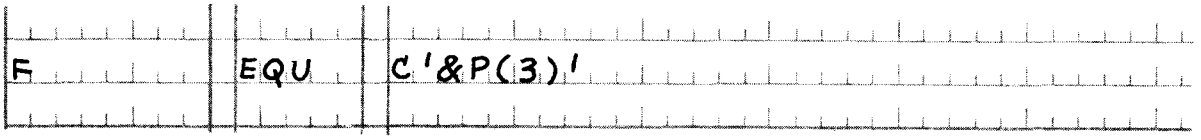
In any statement other than proc call lines, after the set symbol values have been referenced and all the paraforms and keywords have been replaced by the characters from the appropriate parameter subfields, the newly constructed source code line is scanned for conversion and evaluation. This allows the programmer to submit parameter combinations of characters which could not otherwise be used.

Example:

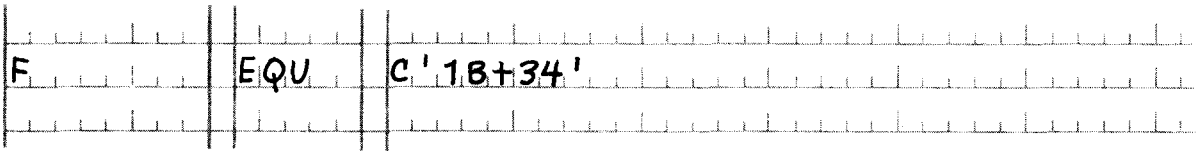
Given the following proc call line



the following line coded within the proc



would effectively produce the following line



and no error flags would occur. If the characters 1B had occurred anywhere else in the module as a term (e.g., 2+1B+34), they would have caused an expression error to be generated.

13.3.2.2. Set Symbol Replacement

Set symbol replacement differs from parameter replacement. The expression or character string which the set symbol represents has already been evaluated. The expression in the operand field of a set directive has been converted into a binary value. When the set symbol value is referenced, it is converted to a string of decimal digits which represents that value. These decimal characters then replace the set symbol reference as part of the newly constructed source code line. When the set symbol represents a character string, no conversion is necessary; thus characters of the character string replace the set symbol reference.

After all set symbol value and parameter references have been replaced, the new source code line is converted and fully evaluated.

Example:

Given the following SET directive statements:

LABEL	OPERATION	OPERAND
&ALFA	SET	X'FF'
&X	SET	'BOB'

and a reference to those symbols

BETA	EQU	&ALFA+1
Y	EQU	&X+27

then the effective results after substitution would be:

BETA	EQU	255+1
Y	EQU	BOB+27

The value that symbol BETA represents is 256, and if BOB represents the value 17, then Y would represent the value 44.

13.3.2.3. Null Character-String Replacement

When a parameter is referenced but is not present on the proc call line or when a set symbol is referenced after it has been declared but before it has been defined, the reference is treated as a null character string. A null character string is a convenient representation of a void. Null character strings generated from references to undefined set symbols or parameters delete the original reference.

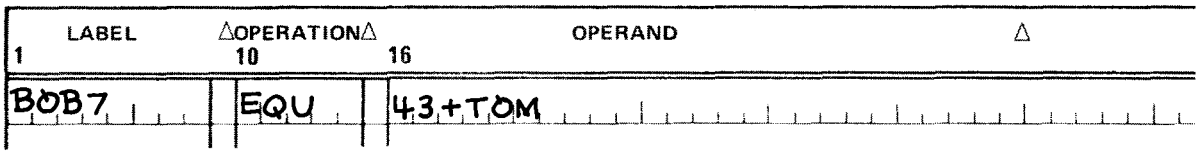
Given the following proc call

LABEL	OPERATION	OPERAND
	PRCL	1,,0,4

this statement within the proc

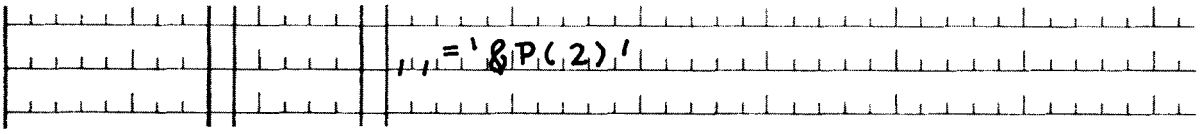
BOB,&P(2)	.7	EQU 43+TOM

would appear and be processed as follows:

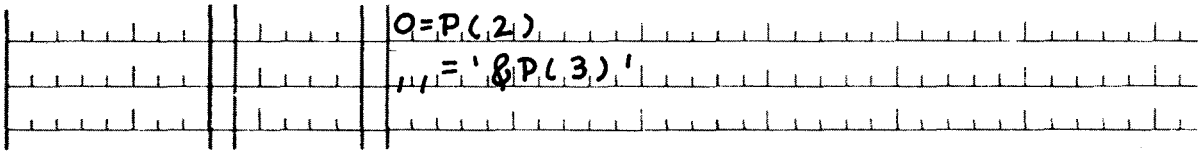


The null character string may also be coded in certain places in the source statements. As coded, a null character string consists of two successive apostrophes which are not inside any other paired apostrophes. It should only be coded as a separate term in arithmetic expressions. Unless it is an argument of a relational operation, a null character string is treated as 0.

In a relational operation, the null character string can be used effectively to test for the presence or the absence of parameters. Thus, given the preceding proc call line, the following expression would have a value of 1;



and these expressions would have a value of 0:



In a relational operation in which one operand is a paraform or set symbol that may be replaced by the null character string, it should be coded as the second operand.

13.4. CALL LINE LABELS

The label of a proc call line is represented by a dummy label which is specified in the PROC directive statement label field. The dummy label is coded within the proc wherever the call line label is to be generated. When the proc is called, the call line label replaces the dummy label wherever it is coded. If there is no label on the call line, the dummy label is replaced by a null character string. The dummy label may be coded in the label operation code, and/or operand field of any instruction, directive, or proc call statement (except special directives) within the proc. It is used like a set symbol (13.3).

13.5. NAME LEVELS AND PROC NESTING

A proc reference may appear within a proc definition; this is considered a second level reference. A maximum of three levels of proc references may be generated within an assembly module. At the source code level and at each of the three proc levels, set symbol definitions may be declared to be available for reference anywhere in the module. Set symbols declared this way (by means of the GBL directive) are globally defined. When a set symbol is declared to be locally defined (by means of the LCL directive), its definition may only be referenced at the level where it is declared. A set symbol locally declared at the source code level is not available for reference within procs and one which is locally declared at some proc level can only be referenced within that proc. Locally declared set symbols may be used within procs or at the source code level without fear of conflicting with set symbols defined locally at some other level or globally throughout the module.

Keyword symbols, the dummy label, and the symbol used to reference positional parameters are treated in the same way as locally declared set symbols. They cannot be referenced outside of the proc in which they are declared. When the proc is completely processed, all the locally declared symbols and their values are discarded.

13.6. METHOD OF WRITING AND REFERENCING PROCS

Although the following examples are limited to procedures within a given program, the system library also contains procs. A call on a library procedure causes that proc to be brought into memory. The assembler then substitutes the input information given in the operand field of the call line and produces the required object lines.

The generation of code from a given procedure is done only at assembly time when a proc call is encountered. The coding thus generated is an integral part of the object program.

Program-defined procs must precede all other statements in the source code module.

Examples:

The following examples illustrate all of the pertinent points and rules necessary in writing and generating procedures.

In a given program it is found necessary to compare two numbers frequently and to load the smaller of the two numbers into a register. The symbolic labels assigned the storage positions to be compared are BOB and JOE, respectively. Because the same load and comparison must be done several times, a proc to generate the proper code can be written. Such a proc might look like this:

	LABEL	OPERATION	OPERAND
	1	10	16
1.		PROC	
2.	SMALL	NAME	0
3.		L	6,BOB
4.		C	6,JOE
5.		BL	*+8
6.		L	6,JOE
7.		END	

1. Initiates a proc.
2. Names the proc.
3. Loads the contents of BOB into register 6.
4. Compares contents of JOE to register 6.
5. Branches out if BOB is smaller.
6. If JOE is smaller, loads JOE into register 6.
7. Ends the proc.

Register 6 has been designated to contain the smaller number. When the following proc call line is encountered in the source code

```
   LABEL    Δ OPERATION Δ          OPERAND          Δ
1           10          16
            $MALL
```

lines 3 through 6 within the proc are generated and inserted at the point where the proc call, SMALL, is encountered.

In order to compare two numbers not located in BOB and JOE, and to store the smaller number in a register other than register 6, the following proc could be coded to allow parameters.

```
   LABEL    Δ OPERATION Δ          OPERAND          Δ
1           10          16
1. &DL      PROC        &P,3
2. $MALL    NAME        0
3. &DL      L           &P(1),&P(2)
4.          C           &P(1),&P(3)
5.          BL          *+8
6.          L           &P(1),&P(3)
7.          END
```

1. Initiates a proc with parameters.

2. Names the proc.

3. Uses paraforms to reference parameters. The &DL in the label field causes the symbol in the label field of the call to be defined as the label of this instruction.

4. Uses paraforms to reference parameters.

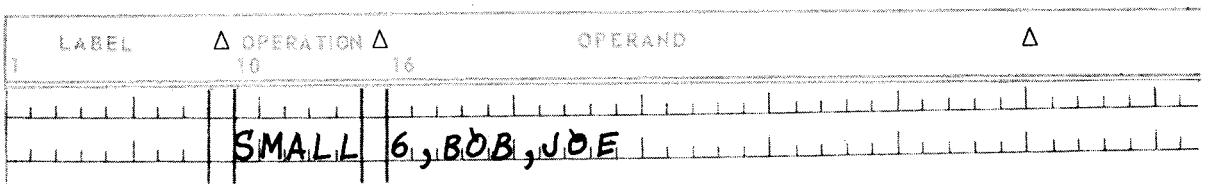
5. Branches out if &P(2) is smaller than &P(3).

6. If &P(3) is smaller than or equal to &P(2), load &P(3) into &P(1).

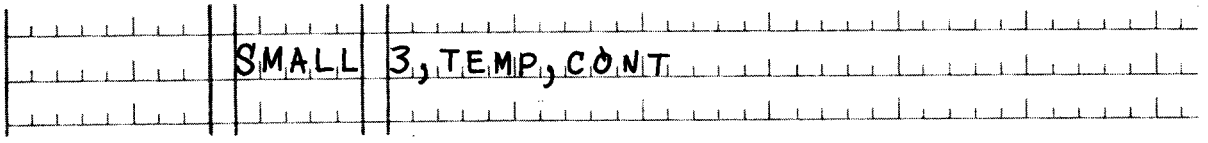
7. Ends the proc.

The &P,3 on line 1 specifies that there are three positional parameters in the body of the procedure; these parameters are referenced on lines 3, 4, and 6. Note that a parameter can be used many times.

A calling statement to generate the same object code as in the previous example would contain these expressions in the call line:



Using the same procedure, other values can be compared and another register can be used by coding:



where TEMP and CONT represent the addresses of two other numbers, and the 3 represents register 3.

To provide a more general procedure to handle two numbers anywhere in storage, using any register and also comparing for either a smaller or larger factor and storing that factor in the register, a proc could be written as follows:

```
1 &DL      PROC    &P,3
2 SMALL    NAME    0
3 LARGE    NAME    1
4 &DL      L       &P(1),&P(2)
5          C       &P(1),&P(3)
6          DO      &P(0)=1
7          BH      *+8
8          GOTO    TND
9          ENDO
10         BL      *+8
11         LABEL
12         L       &P(1),&P(3)
13         END
```

1. Initiates a proc with parameters.
2. Names the proc for smaller comparison.
3. Names the proc for larger comparison.
4. Uses paraforms to reference parameters.
5. Uses paraforms to reference parameters.

6. Does the following two lines of coding if &P(0) equals 1 (this makes comparison for the larger of two factors).

7. Branches out if &P(2) is larger.

8. Branches to label defined as TND.

9. Terminates DO statements.

10. Branches out if &P(2) is smaller.

11. Defines TND as a GOTO label.

12. Uses paraforms to reference parameters.

13. Ends the proc.

A call line using the proc name SMALL generates the same object coding as in the previous examples, because the expression in the operand field &(P(0)) of the DO line is 0; therefore lines 7 and 8 are not evaluated. If the following call line is used

LABEL	OPERATION	OPERAND
AB1	LARGE	6,BOB,JOE

then the DO line expression is equal to 1, lines 7 and 8 are evaluated, line 10 is skipped, and the following object statements are generated.

AB1	L	6,BOB
	C	6,JOE
	BH	*+8
	L	6,JOE

A simpler procedure to accomplish the same result can be coded:

```
1.&DL     PROC   &P,3
2.SMALL   NAME   L
3.LARGE   NAME   H
4.&DL     L      &P(1),&P(2)
5.        C      &P(1),&P(3)
6.        B&P(0)    *+8
7.        L      &P(1),&P(3)
8.        END
```

1. Initiates proc with three parameters.
2. Names the proc for smaller (low) comparison.
3. Names the proc for larger (high) comparison.
4. Uses parameter reference forms for the register (&P(1), displacement (&P(2,1)), index register (&P(2,2)), and base register (&P(2,3)).
5. Same as line 4 except uses parameter 3 instead of 2.
6. Branches out. &P(0) varies either L or H depending on the call line.
7. Same as line 5.
8. Ends the proc.

The character L or H on lines 2 or 3 replaces the parameter &P(0) in the source code line on line 6. Line 6 is then evaluated as either:

```
LABEL     Δ OPERATION Δ     OPERAND                    Δ
1              10        16
          BL              *+8
```

or

```
          BH              *+8
```

The explicit form of base displacement addressing with the ability to specify index registers for the generated instruction can be provided by the following:

```
1 &DL      PROC    &P,3
2 SMALL    NAME    L
3 LARGE    NAME    H
4 &DL      L       &P(1),&P(2,1),&P(2,2),&P(2,3)
5          C       &P(1),&P(3,1),&P(3,2),&P(3,3)
6          B&P(0)      *+8
7          L       &P(1),&P(3,1)(&P(3,2),&P(3,3))
8          END
```

If the proc call line is specified as follows:

```
LABEL     Δ OPERATION Δ                OPERAND                     Δ
1           10          16
            LARGE       6,(50,4,15),(150,5,15)
```

the following object coding would be generated:

```
1.          L           6,50(4,15)
2.          C           6,150(5,15)
3.          BH          *+8
4.          L           6,150(5,15)
```

The generated coding on line 1 specifies loading into register 6 the contents of a location whose address is determined by adding the displacement 50 to the contents of index register 4 and then adding the contents of base register 15. Lines 2 and 4 are handled similarly by using the displacement of 150 plus the contents of index register 5 plus the contents of base register 15.

If one of the numbers is already in the correct register, then the first load instruction need not be generated. By omitting the second parameter on the call line when the number is in the register, the proc can test that subfield to determine when to generate the first load instruction.

```
&DL         PROC        &P,3
SMALL       NAME        L
LARGE       NAME        H
            LCL         &LCLI
&LCLI       SET         '&DL'
            DO          '&P(2)'=''=0
&DL         L           &P(1),&P(2,1)(&P(2,2))
LCLI        SET         ''
            ENDO
&LCLI       C           &P(1),&P(3,1)(&P(3,2))
            B&P(0)      *+8
            L           &P(1),&P(3,1)(&P(3,2))
            END
```

4. Declares a local set symbol for use in inserting the label of the proc call in the first generated line.

5. Sets the local set symbol to the character string in the label of the proc call line.

6. Generates the load instruction only if the call to this proc has a non-null second operand.

7. The load instruction, if generated, always has the label of the proc call.

8. Sets the local set symbol to a null character string so that line 8 will be unlabeled if line 7 is generated.

10. Defines the label on the proc call line only if line 7 is not generated.

If the call line for the preceding is:

```
1    LABEL     ΔOPERATIONΔ     OPERAND                    Δ
                10          16
WXYZ           SMALL       6,,(JOE,5)
```

then the object code of the following statements would be generated:

```
WXYZ           C           6,JOE(5)
               BL          *+8
               L           6,JOE(5)
```

A proc call such as:

```
UVW            SMALL       6,(BOB),(JOE,5)
```

would generate the following statements in object coding:

```
UVW            L           6,BOB()
               C           6,JOE(5)
               BL          *+8
               L           6,JOE(5)
```

13.7. VARIABLE SYMBOLS

A variable symbol may be used as any of the following:

- a symbolic parameter
- a set symbol
- the label of a DO directive
- the label of a PROC directive
- a system variable symbol

A variable symbol consists of from two through nine characters, the first of which is an ampersand (&), the second a letter or special letter, and each of the remaining characters a letter, special letter, or digit. A symbolic parameter represents the label or the macro instruction or one of the operands of the macro instruction by which the macro definition is called. SET symbols and the DO directive are described in 12.8 and 12.8.4, respectively. System variable symbols are described in 13.7.2.

13.7.1. Use of Variable Symbols

SET symbols are replaced whenever found, including operation fields and label fields, except for SET statements. Therefore, assembler directives and mnemonic operation codes are not permitted to be used as:

- SET symbols (LCL or GBL)
- keyword parameters
- labels of PROC directives
- labels of DO statements

13.7.1.1. Concatenation of Variable Symbols

A variable symbol may appear in a statement concatenated with other variable symbols or other characters. If a variable symbol is to be immediately followed by a letter, digit, left parenthesis, or period, a period must be written after the variable symbol to distinguish the variable symbol from the characters following it. The variable symbol and the period following it are replaced by the characters representing the value of the variable symbol. The period does not appear in the resultant statement.

13.7.2. System Variable Symbols

System variable symbols are assigned values automatically by the assembler. Two of these symbols, &SYSNDX and &SYSECT, can appear only in the label, operation, or operand fields of statements in proc definitions. Two other system variable symbols, &SYSDATE and &SYSTIME, can appear in proc definitions or in source modules.

13.7.2.1. &SYSNDX

The system variable symbol &SYSNDX is used to prevent the occurrence of doubly-defined labels; that is, &SYSNDX can be combined with other characters to create unique names for statements generated within the same proc definition.

Initially, &SYSNDX is assigned 4-digit number 0001 to correspond to the first proc definition processed by the assembler. For each subsequent proc definition, symbol &SYSNDX is incremented by 1 so that it effectively keeps a running count of all procs being processed. Thus, if &SYSNDX is used in a proc definition, the value substituted for it will correspond to the current proc definition being processed.

Throughout one use of a proc definition, the value of &SYSNDX remains constant, independent of any nested procs within that definition.

The following coding example illustrates the use of the &SYSNDX symbol. The assumption is that the program is calling the same proc twice, and that the proc itself (called MAIN) contains a nested proc (called NEST).

```
                LABEL     OPERATION     OPERAND
                1         10            16
1.                        PROC          &P,0
                NEST      NAME
                          GBL           &NDXNUM
                A&SYSNDX  SR            3,6
                          CR            3,6
                          BE            A&NDXNUM
                          B             A&SYSNDX
                          END
2.                        PROC          &P,0
                MAIN      NAME
                          GBL           &NDXNUM
3.              &NDXNUM   SET           &SYSNDX
                A&NDXNUM  SR            4,7
                          AR            4,9
4.                        NEST
                B&SYSNDX  SR            2,5
                          END
                          .
                          .
```

Coding continued:

```
    LABEL     ΔOPERATIONΔ        OPERAND                              Δ
1             10     16
              .
 5.           BALR   1,0
              USING  *,1
 6. PROGA     MAIN                                         } Generated Code
              GBL    &NDXNUM                               } from MAIN proc
 7. &NDXNUM   SET    0001                                  } (first call)
    A1        SR     4,7
              AR     4,9
 8.           NEST
              GBL    &NDXNUM   } Generated Code
 9. A0002     SR     3,6       } from NEST proc
              CR     3,6       } (firstcall)
10.           BE     A1
11.           B      A0002
12. B0001     SR     2,5
13. PROGB     MAIN                                         } Generated Code
              GBL    &NDXNUM                               } from MAIN proc
14. &NDXNUM   SET    0003                                  } (second call)
    A3        SR     4,7
              AR     4,9
15.           NEST
              GBL    &NDXNUM   } Generated Code
16. A0004     SR     3,6       } from NEST proc
              CR     3,6       } (second call)
17.           BE     A3
18.           B      A0004
19. B0003     SR     2,5
              END
```

1. Start of nested proc definition.
2. Start of main proc definition.
3. &SYSNDX is initially assigned the value 0001.
4. Call line for nested proc.
5. Start of main program (showing all generated coding).
6. Call line for main proc (first time).
7. &SYSNDX = 0001, &NDXNUM set to 1 (leading 0's dropped).

8. Call line for nested proc (first time).

9. &SYSNDX = 0002.

10. &NDXNUM = 1 (from main proc).

11. &SYSNDX = 0002.

12. &SYSNDX = 0001, within first call to MAIN.

13. Call line for main proc (second time).

14. &SYSNDX = 0003, &NDXNUM set to 3 (leading 0's dropped).

15. Call line for nested proc (second time).

16. &SYSNDX = 0004.

17. &NDXNUM = 3 (from main proc).

18. &SYSNDX = 0004.

19. &SYSNDX = 0003.

13.7.2.2. &SYSECT

The system variable symbol &SYSECT is used to represent the name of a control section in which a proc call appears; that is, for each proc call processed by the assembler, &SYSECT is assigned a value that corresponds to the name of the control section in which the proc call appears.

Control section statements (CSECT or DSECT) processed in a proc definition affect the value for &SYSECT for any subsequent nested procs in that definition. However, throughout the use of a proc definition, the value of &SYSECT may be considered a constant, independent of any CSECT or DSECT statements or nested procs. The name assigned to &SYSECT will always be that of the last CSECT, DSECT, or START statement, regardless of whether that statement is correct.

The following coding example illustrates the use of the &SYSECT symbol. Here, the program is calling two procs (called MAINA and MAINB) in succession, the first of which (MAINA) involves calling the same nested proc twice in succession.

LABEL	Δ OPERATION Δ	OPERAND	Δ	COMMENTS

```
1.  &NESECT    PROC     &P,0
    NEST       NAME     0
    &NESECT    CSECT
               DC       A(&SYSECT)
               END
                 .
                 .
                 .
2.             PROC     &P,0
    MAINA      NAME
    C$MAINA    CSECT
               DS       CL100
3.  NESTA      NEST
4.  NESTB      NEST
               DC       A(&SYSECT)
5.  &SYSECT    CSECT
               END
6.             PROC     &P,0
    MAINB      NAME
               DC       A(&SYSECT)
               END
                 .
                 .
                 .
                 .
                 .
7.  MAINPROG   CSECT
               DS       CL200
8.             MAINA
    C$MAINA    CSECT                                                  }
               DS       CL100                                         }
9.  NESTA      NEST                                                   }
10. NESTA      CSECT                } Generated Code from             }
               DC       A(C$MAINA)  } NEST proc (first call)          } Generated Code
11. NESTB      NEST                                                   } from MAINA proc
    NESTB      CSECT                } Generated Code from             }
12.            DC       A(NESTA)    } NEST proc (second call)         }
13.            DC       A(MAINPROG)                                   }
14. MAINPROG   CSECT
15.            MAINB
16.            DC       A(MAINPROG) } Generated Code from MAINB proc
               END
```

1. Start of nested proc definition.
2. Start of main proc A definition.
3. Call line for nested proc NESTA.
4. Call line for nested proc NESTB.
5. Returns to the CSECT under which code was being generated when MAINA was called.
6. Start of main proc B definition.
7. Start of main program (showing all generated coding).
8. Call line for main proc A.
9. Call line for nested proc NESTA.
10. &SYSECT = CSMAINA.
11. Call line for nested proc NESTB.
12. &SYSECT = NESTA.
13. Return to main proc A (&SYSECT = MAINPROG).
14. Returns to the CSECT under which code was being generated when MAINA was called.
15. Call line for main proc B.
16. &SYSECT = NESTB.

13.7.2.3. &SYSDATE

The system variable symbol &SYSDATE provides the date of the assembly. The format of &SYSDATE is the character string mm/dd/yy representing month, day, and year. The value of &SYSDATE is constant during the assembly.

13.7.2.4. &SYSTIME

The system variable symbol &SYSTIME provides the time of the assembly. The format of &SYSTIME is the character string hh:mm representing hours and minutes. The value of &SYSTIME is constant during the assembly.

14. Error Messages

14.1. MESSAGE TYPES AND FORMAT

Fatal, diagnostic, and academic are the three levels of error messages provided by the assembler. Each error message (flag) is a single alphabetic character. The assembler analyzes each source code statement (except after a fatal error) as it processes the statement. When an error is found, the appropriate flag is printed on the same line as the source code statement containing the error on the program listing to the left of the relative storage address.

14.2. FATAL ERRORS

Fatal error flags signify that the processing of any remaining source code statements would produce meaningless results. The assembler produces a partial listing of the program but all statements are not completely analyzed.

The fatal errors are:

- ESID Overflow – B
- Storage Overflow – F

14.3. DIAGNOSTIC ERRORS

Diagnostic error flags signify incorrectly specified source code statements. These errors are not serious enough to prevent the normal processing and the generation of binary output; however, they prohibit execution of the program. The appropriate flags are printed on the program listing on the same line as the statements containing errors.

The diagnostic errors are:

- Expression Not Relocatable – A

- Covering Error – C
- Duplication Error – D
- Expression Error – E
- Statement Too Large – G
- Boundary Alignment – H
- Operation Code Error – I
- Syspool Overflow – K
- Location Counter – L
- Undefined Symbol – U
- Internal Assembler Failure – V
- Continuation Error – X
- Too Many Nested DO or PROC Directives – Z

14.4. ACADEMIC MESSAGES

Academic message flags signify that certain actions have been taken by the assembler. These actions can be caused by erroneous coding or by the programmer who wishes to obtain a specific result. Academic messages do not prohibit or have any effect on the output of an assembly, and these occurrences are not considered serious in nature.

- Conditional Assembly Error – M
- Name Field Error – N
- Relocation Information Dropped – R
- Statement Out of Sequence – S
- Truncation – T
- Symbol to a Character String – W

14.5. ERROR MESSAGE SUMMARY

Table 14–1 is a summary of all error messages, the type of error, and the meaning of each message.

Table 14–1. Error Message Summary (Part 1 of 3)

Flag	Level	Explanation	Response Action
A	Diagnostic	Expression not relocatable – An absolute expression is provided where a relocatable expression is required.	Replace absolute expression with relocatable expression.
B	Fatal	External symbol identification (ESID) overflow – More than 254 external symbol identifications exclusive of COM are specified.	Reduce number of ESIDs to conform to limit of 254.
C	Diagnostic	Covering error – No valid base register can be found to cover or reach the effective storage address and still have a displacement value from 0 to 4095.	Specify appropriate base register.
D	Diagnostic	Duplication error – A label is defined more than once. Set symbols can be redefined without producing a flag.	Eliminate duplicate labels.
E	Diagnostic	Expression error – The operand field for an instruction or a directive has an incorrect format.	Correct incorrect formats.
F	Fatal	There is no more space available to the assembler for expanding tables during macro generation. This does not apply to literals, location counter derived symbols, and symbols defined by the EQU directive.	
G	Diagnostic	Statement too large – The number of characters included in the statement exceeds the size of the buffer from which the statement is processed.	Either decrease size of statement, or increase the size of buffer.
H	Diagnostic	Boundary alignment error – Refer to the // PARAM LST=(4) statement in E.2.3. The operand addresses of RX, SI, and RS instructions are checked for boundary alignment. The flag is generated if the address violates the boundary alignment rules.	Correct alignment.
I	Diagnostic	Operation code error – An illegal operation or an undefined operation is specified.	Correct specification.
K	Diagnostic	Syspool overflow – The disc assembler has overflowed the syspace area allotted to processing EXTRN and ENTRY records.	Remap the syspool using DACMAP (utility and service routines, UP-7713) and resubmit job. If problem persists, additional syspool area must be allotted to the job.

Table 14–1. Error Message Summary (Part 2 of 3)

Flag	Level	Explanation	Response Action
L	Diagnostic	Location counter overflow – Either one of the location counters has overflowed or the assembled program has exceeded the specified maximum storage space available. The assembler assumes that the computer in which the object module is to be executed has the same configuration as the computer in which the program is assembled, unless otherwise stated.	Reassemble specifying correct object computer size.
M	Academic	Conditional assembly error – DO and ENDO statements have not been paired.	Ensure that DO and ENDO statements are paired.
N	Academic	Name field error – The label field contains an illegal symbol, no symbol when one is necessary, or a symbol when one is not allowed.	Correct label error.
R	Academic	Relocation information dropped – A relocatable term is used in such a manner that its relocation information is no longer valid in this instance. This condition occurs frequently when a relocatable term is used in an expression.	Correct the usage of relocatable terms.
S	Academic	Statement out of sequence – An S flag signifies that a START, PROC, NAME, or ICTL statement is out of sequence. This flag also signifies that sequence numbers in the source records are not in ascending order.	Place statements and records in proper sequence.
T	Academic	Truncation – A specified or completed value is too large for the field in which it is stored; therefore, it is truncated and inserted into the field. Truncation can be intentional; it does not prevent the execution of the program.	Reduce length of value or increase length of receiving field, if truncation is not intentional.
U	Diagnostic	Undefined symbol – One or more symbols in a source code statement are undefined during the assembly. Any symbols that remain undefined are equated to 0. However, undefined paraforms do not produce an error flag. All symbols that are defined in another object module must be identified in the EXTRN statement so that they can be properly processed and not cause U flags.	Define all undefined symbols.

Table 14–1. Error Message Summary (Part 3 of 3)

Flag	Level	Explanation	Response Action
V	Diagnostic	Internal assembler failure – An internal assembler failure is detected. This error is caused primarily by catastrophic hardware failure.	Notify Sperry Univac customer representative.
W	Academic	Symbol to character string – An undefined symbol is coded in a basic expression. The symbol is treated by the assembler as a character string.	Correct coding errors.
X	Diagnostic	Continuation error – A continuation card has not been provided for a statement whose operand field is incomplete.	Correct coding errors.
Z	Diagnostic	Too many nested DO or PROC directives – More than 10 levels of DO directives or three levels of PROC directives are coded for assembly.	Correct coding errors.

Appendix A. Instruction Repertoire

Mnemonic Code	Description	Operation Code	Type	90/60, 70	9400/ 9480
A	Add	5A	RX	X	X
AD	Add-normalized (long format)	6A	RX	X	
ADR	Add-normalized (long format)	2A	RR	X	
AE	Add-normalized (short format)	7A	RX	X	
AER	Add-normalized (short format)	3A	RR	X	
AH	Add-half-word	4A	RX	X	X
AI	Add-immediate	9A	SI	X	
		93	SI		X
AL	Add-logical	5E	RX	X	
ALR	Add-logical	1E	RR	X	
AP	Add-decimal	FA	SS	X	X
AR	Add	1A	RR	X	X
AU	Add-unnormalized (short format)	7E	RX	X	
AUR	Add-unnormalized (short format)	3E	RR	X	
AW	Add-unnormalized (long format)	6E	RX	X	
AWR	Add-unnormalized (long format)	2E	RR	X	
BAL	Branch-and-link	45	RX	X	X
BALE	Branch-and-link-external	4D	RX	X	
BALR	Branch-and-link	05	RR	X	X
BC	Branch-on-condition	47	RX	X	X
BCR	Branch-on-condition	07	RR	X	
BCRE	Branch-on-condition-to-return-external	0C	RR	X	
BCT	Branch-on-count	46	RX	X	X

Mnemonic Code	Description	Operation Code	Type	90/60, 70	9400/ 9480
BCTR	Branch-on-count	06	RR	X	X
BXH	Branch-on-index-high	86	RS	X	
BXLE	Branch-on-index-low-or-equal	87	RS	X	
C	Compare	59	RX	X	X
CD	Compare (long format)	69	RX	X	
CDR	Compare (long format)	29	RR	X	
CE	Compare (short format)	79	RX	X	
CER	Compare (short format)	39	RR	X	
CH	Compare-half-word	49	RX	X	X
CL	Compare-logical	55	RX	X	X
CLC	Compare-logical	D5	SS	X	X
CLI	Compare-logical	95	SI	X	X
CLR	Compare-logical	15	RR	X	X
CP	Compare-decimal	F9	SS	X	X
CR	Compare	19	RR	X	X
CVB	Convert-to-binary	4F	RX	X	
CVD	Convert-to-decimal	4E	RX	X	
D	Divide	5D	RX	X	
DD	Divide (long format)	6D	RX	X	
DDR	Divide (long format)	2D	RR	X	
DE	Divide (short format)	7D	RX	X	
DER	Divide (short format)	3D	RR	X	
DIAG	Diagnose (privileged instruction)	83	SI	X	
DP	Divide-decimal	FD	SS	X	X
DR	Divide	1D	RR	X	
EA	Emulation-aid	EZ	*	X	
ED	Edit	DE	SS	X	X
EDMK	Edit-and-mark	DF	SS	X	
EX	Execute	44	RX	X	
HDR	Halve (long format)	24	RR	X	
HER	Halve (short format)	34	RR	X	
HIO	Halt-I/O (privileged instruction)	9E	SI	X	

Mnemonic Code	Description	Operation Code	Type	90/60, 70	9400/ 9480
HPR	Halt-and-proceed (privileged instruction)	99	SI	X	X
IC	Insert-character	43	RX	X	X
ISK	Insert-storage-key	09	RR	X	
L	Load	58	RX	X	X
LA	Load-address	41	RX	X	X
LBR	Load-base-register	0B	RR	X	
LCDR	Load-complement (long format)	23	RR	X	
LCER	Load-complement (short format)	33	RR	X	
LCHR	Load-channel-register (privileged instruction)	AD	SI	X	
LCR	Load-complement	13	RR	X	
LCS	Load-control-storage (privileged instruction)	B1	RS	X	
LD	Load (long format)	68	RX	X	
LDR	Load (long format	28	RR	X	
LE	Load (short format)	78	RX	X	
LER	Load (short format)	38	RR	X	
LH	Load-half-word	48	RX	X	X
LLR	Load-limits-register	81	RS		X
LM	Load-multiple	98	RS	X	X
LNDR	Load-negative (long format)	21	RR	X	
LNER	Load-negative (short format)	31	RR	X	
LNR	Load-negative	11	RR	X	
LPDR	Load-positive (long format)	20	RR	X	
LPER	Load-positive (short format)	30	RR	X	
LPR	Load-positive	30	RR	X	
LNR	Load-negative	10	RR	X	
LPSW	Load-program-status-word (privileged instruction)	82	SI	X	X
LR	Load	18	RR	X	X
LTDR	Load-and-test (long format)	22	RR	X	
LTER	Load-and-test (short format)	32	RR	X	
LTR	Load-and-test	12	RR	X	X

Mnemonic Code	Description	Operation Code	Type	90/60, 70	9400/ 9480
M	Multiply	5C	RX	X	
MD	Multiply (long format)	6C	RX	X	
MDR	Multiply (long format)	2C	RR	X	
ME	Multiply (short format)	7C	RX	X	
MER	Multiply (short format)	3C	RR	X	
MH	Multiply-half-word	4C	RX	X	
MP	Multiply-decimal	FC	SS	X	X
MR	Multiply	1C	RR	X	
MVC	Move	D2	SS	X	X
MVI	Move	92	SI	X	X
MVN	Move-numerics	D1	SS	X	X
MVO	Move-with-offset	F1	SS	X	X
MVZ	Move-zones	D3	SS	X	X
N	AND	54	RX	X	X
NC	AND	D4	SS	X	X
NI	AND	94	SI	X	X
NR	AND	14	RR	X	X
O	OR	56	RX	X	X
OC	OR	D6	SS	X	X
OI	OR	96	SI	X	X
OR	OR	16	RR	X	X
PACK	Pack	F2	SS	X	X
RDD	Read-direct (privileged instruction)	85	SI	X	
S	Subtract	5B	RX	X	X
SCHR	Store-channel-register (privileged instruction)	AC	SI	X	
SD	Subtract-normalized (long format)	6B	RX	X	
SDR	Subtract-normalized (long format)	2B	RR	X	
SE	Subtract-normalized (short format)	7B	RX	X	
SER	Subtract-normalized (short format)	3B	RR	X	
SH	Subtract-half-word	4B	RX	X	X
SIO	Start-I/O (privileged instruction)	9C	SI	X	X

Mnemonic Code	Description	Operation Code	Type	90/60, 70	9400/ 9480
SL	Subtract-logical	5F	RX	X	
SLA	Shift-left-single	8B	RS	X	
SLDA	Shift-left-double	8F	RS	X	
SLDL	Shift-left-double-logical	8D	RS	X	
SLL	Shift-left-single-logical	89	RS	X	X
SLM	Supervisor-load-multiple (privileged instruction)	B8	RS	X	X
SLR	Subtract-logical	1F	RR	X	
SP	Subtract-decimal	FB	SS	X	X
SPM	Set-program-mask	04	RR	X	X
SR	Subtract	1B	RR	X	X
SRA	Shift-right-single	8A	RS	X	
SRDA	Shift-right-double	8E	RS	X	
SRDL	Shift-right-double-logical	8C	RS	X	
SRL	Shift-right-single-logical	88	RS	X	X
SSK	Set-storage-key (privileged instruction)	08	RR	X	
SSM	Set-system-mask (privileged instruction)	80	SI	X	X
SSTM	Supervisor-store-multiple (privileged instruction)	B0	RS	X	X
ST	Store	50	RX	X	X
STC	Store-character	42	RX	X	X
STD	Store (long format)	60	RX	X	
STE	Store (short format)	70	RX	X	
STH	Store-half-word	40	RX	X	X
STM	Store-multiple	90	RS	X	X
SU	Subtract-unnormalized (short format)	7F	RX	X	
SUR	Subtract-unnormalized (short format)	3F	RR	X	
SVC	Supervisor-call	0A	RR	X	X
SW	Subtract-unnormalized	6F	RX	X	
SWR	Subtract-unnormalized (long format)	2F	RR	X	
TCH	Test-channel (privileged instruction)	9F	SI	X	
TIO	Test-I/O (privileged instruction)	9D	SI	X	

Mnemonic Code	Description	Operation Code	Type	90/60, 70	9400/ 9480
TM	Test-under-mask	91	SI	X	X
TR	Translate	DC	SS	X	X
TRT	Translate-and-test	DD	SS	X	
UNPK	Unpack	F3	SS	X	X
WRD	Write-direct (privileged instruction)	84	SI	X	
X	Exclusive-OR	57	RX	X	X
XC	Exclusive-OR	D7	SS	X	X
XI	Exclusive-OR	97	SI	X	X
XR	Exclusive-OR	17	RR	X	X
ZAP	Zero-and-add	F8	SS	X	X

Appendix B. 9400/9480 and 90/60,70 Hardware Differences

B.1. GENERAL

The SPERRY UNIVAC Operating System/4 (OS/4) provides SPERRY UNIVAC 9400/9480 compatibility mode on SPERRY UNIVAC 90/60,70 hardware. This compatibility mode does not duplicate all the characteristics of the 9400/9480 systems hardware. Therefore, there are minor hardware differences between the 9400/9480 systems and the 90/60,70 systems. These differences may require some coding modifications to 9400/9480 programs.

For additional hardware information, see the processor programmer references, UP-7936 (current version) for 90/60,70 and UP-8080 (current version) for 9400/9480 systems.

B.2. INSTRUCTION DIFFERENCES

B.2.1. Add Immediate (AI)

- 9400/9480 systems

  AI has an op code of 93_{16}

- 90/60,70 systems

  In 9400/9480 compatibility mode, the AI op code is 93_{16} or $9A_{16}$.

B.2.2. Add Decimal (AP) and Subtract Decimal (SP)

- 9400/9480 systems

  If operand 2 is longer than operand 1, the high-order digits of operand 2 are ignored.

- 90/60,70 systems

  A program exception interrupt may occur as a result of processing the significant digits.

B.2.3. Compare Decimal (CP)

- 9400/9480 systems

  If operand 2 is longer than operand 1, the high-order digits of operand 2 are ignored.

- 90/60,70 systems

  The shorter operand is extended with 0's.

B.2.4. Divide Decimal (DP)

- 9400/9480 systems

  Operand 1 length is ignored and execution depends upon first occurrence of a sign. The divisor (operand 2) may be a maximum of 31 digits plus a sign.

- 90/60,70 systems

  If a sign is not encountered with the first 16 bytes of data, a program exception interrupt occurs. The divisor (operand 2) may be a maximum of 15 digits plus a sign.

B.2.5. Load Address (LA)

- 9400/9480 systems

  The 9400/9480 systems use 18-bit main storage addresses.

- 90/60,70 systems

  The 90/60,70 systems use 24-bit main storage addresses.

B.2.6. Multiply Decimal (MP)

- 9400/9480 systems

  Operand 1 length is ignored and execution depends upon first occurrence of a sign.

- 90/60,70 systems

  If sign is not encountered with the first 16 bytes of data, a program exception interrupt occurs. The multiplier (operand 2) may be a maximum of 15 digits plus a sign.

B.2.7. Set Program Mask (SPM) and Program Status Word (PSW)

- 9400/9480 systems

  Bit position 12 is for ASCII mode.

  Bit positions 38 and 39 of the PSW are not used.

- 90/60,70 systems

  Bit positions 2 to 7 of the specified register are transferred to bit positions 34 to 39 of the current PSW.

  Bit position 16 is for ASCII mode.

  Bit positions 38 and 39 are used due to the additional hardware capabilities. Other differences will be noted in the interrupt code portion of the PSW.

B.2.8. Set System Mask (SSM)

- 9400/9480 systems

  The operand (mask) is a 1-byte data field.

- 90/60,70 systems

  The operand (mask) is a 2-byte data field which must be aligned on a half-word boundary.

B.3. BUFFER CONTROL WORD (BCW) DIFFERENCES

The 90/60,70 I/O interfaces do not use BCWs. Any programs which may have communicated with the I/O at the physical I/O level require BCW translation to channel command words (CCW).

B.4. CHANNEL COMMAND WORD (CCW) DIFFERENCES

The 90/60,70 I/O channels contain minor differences from those used in the 9400/9480 systems. Therefore, the following considerations must be given to the CCWs used in I/O processes:

- Command code field of 0 (acceptable in the 9400/9480 systems) is not acceptable in the 90/60,70 environment.
- A TIC-to-TIC operation (acceptable in the 9400/9480 systems) causes a program check in subchannel status in the 90/60,70 systems.
- The 9400/9480 command codes TIO, SIS, and RIS require modification for the OS/4 environment.
- A 0-byte count field for commands other than TIC (acceptable in 9400/9480 systems) results in an I/O interrupt (subchannel status) in the 90/60,70 systems.
- The 90/60,70 hardware has the ability to create an interrupt in an incorrect length condition when the length of data transferred does not equal the byte count.
- Although the CCW proc facilities of the 9400 system software provided double-word alignment, the 9400/9480 systems hardware did not actually enforce this requirement. In the 90/60,70 systems, CCWs must be aligned on double-word boundaries or execution is not allowed.

B.5. STANDARD EQUATE PROC

- 9400/9480 systems

  This proc (STDEQU) contains appropriate system labels, including those which represent 9400/9480 systems low order storage areas.

- 90/60,70 systems

  This proc (STDEQUSH) contains additional system labels, unique to the OS/4 90/60,70 environment.

B.6. REFERENCE TO NONEXISTENT STORAGE

- 9400/9480 systems

  In systems with 262K storage, reference to nonexistent storage results in wraparound addressing. In smaller systems, the result varies according to the operation as follows:

  – Read (load), zeros are picked up.

  – Write (store) results in no-op unless write-protection is included thus causing interrupts.

  – Branch results in illegal instruction interrupt at the nonexistent location.

- 90/60,70 systems

  References to nonexistent storage result in an addressing exception interrupt.

B.7. MCP TELETYPEWRITER LINE TERMINALS

- 9400/9480 systems

  MCP supported use of multiplexer channel adapter which checks start of message (SOM) and end of message (EOM) functions.

- 90/60,70 systems

  Multiplexer channel adapter SOM and EOM check features are not available.

B.8. STORAGE REQUIREMENTS OF PREAMBLE AND EXTENT/PROTECTED DTF AREAS

- 9400/9480 systems

  Storage allocated to the preamble and the extent/protected DTF areas is:

Storage Size	Preamble Area (Bytes)	Extent/Protected DTF Area (Bytes)
131K or less	512	512 or a multiple thereof
262K	1024	1024 or a multiple thereof

- 90/60,70 systems

  Storage allocation for both the preamble and the extent/protected DTF areas is 2048 bytes or a multiple thereof.

Appendix C. ASCII,EBCDIC, and Punched Card Codes

Table C–1. ASCII (American Standard Code for Information Interchange) Character Codes

		Bit Positions 7, 6, 5							
		000	001	010	011	100	101	110	111
Bit Positions 4, 3, 2, 1	0000	NUL	DLE	SP	0	@	P	`	p
	0001	SOH	DC1	! ①	1	A	Q	a	q
	0010	STX	DC2	"	2	B	R	b	r
	0011	ETX	DC3	#	3	C	S	c	s
	0100	EOT	DC4	$	4	D	T	d	t
	0101	ENQ	NAK	%	5	E	U	e	u
	0110	ACK	SYN	&	6	F	V	f	v
	0111	BEL	ETB	'	7	G	W	g	w
	1000	BS	CAN	(	8	H	X	h	x
	1001	HT	EM	)	9	I	Y	i	y
	1010	LF	SUB	*	:	J	Z	j	z
	1011	VT	ESC	+	;	K	[	k	{
	1100	FF	FS	,	<	L	\	l	| ②
	1101	CR	GS	–	=	M	]	m	}
	1110	SO	RS	.	>	N	^ ①	n	~
	1111	SI	US	/	?	O	_	o	DEL

③ ④ ⑤

NOTES:

ASCII bits are numbered from the left in descending numerical sequence: 7 6 5 4 3 2 1

① The following optional graphics can be substituted in the character set:

¬ for ^

¦ for !

② For 63-character printers, the following substitution is made:

\ for ¦

③ Sixty-three printable character set.

④ Graphics available by use of the type 0768-02 printer which prints a 94-character set (DEL is not a graphic)

⑤ Ninety-four printable character set.

Table C–2. EBCDIC (Extended Binary Coded Decimal Interchange Code) Character Codes

Bit Positions 4, 5, 6, 7	Bit Positions 0, 1, 2, 3															
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00000	NUL	DLE	DS (1)		SP	&	–						{ (4)	} (4)	\ (4)	0
0001	SOH	DC1	SOS (1)				/		a (4)	j	~ (4)		A	J		1
0010	STX	DC2	FS (1)	SYN					b	k	s		B	K	S	2
0011	ETX	DC3							c	l	t		C	L	T	3
0100									d	m	u		D	M	U	4
0101	HT		LF						e	n	v		E	N	V	5
0110		BS	ETB						f	o	w		F	O	W	6
0111	DEL		ESC	EOT					g	p	x		G	P	X	7
1000		CAN							h	q	y		H	Q	Y	8
1001		EM						` (4)	i	r	z		I	R	Z	9
1010					[(2)	] (2)	¦ (3)	:								
1011	VT				.	$	,	#								
1100	FF	FS		DC4	<	*	%	@								
1101	CR	GS	ENQ	NAK	(	)	_	'								
1110	SO	RS	ACK		+	;	>	=								
1111	SI	US	BEL	SUB	!	^	?	"								

NOTES:

EBCDIC bits are numbered from the left in ascending numerical order:

0 1 2 3 4 5 6 7

(1) DC, SOS, FS are the control characters for the EDIT instruction and have been assigned for ASCII mode processing so as not to conflict with the corresponding character positions previously assigned in the EBCDIC chart. As these characters are not outside the range as defined in *American National Standard, X3.4 – 1968,* they must not appear in external storage media, such as ANSI standard tapes. This presents no difficulty due to the nature of the EDIT instruction.

(2) The following optional graphics can be substituted in the character set:

¢ for [

! for]

(3) For 63-character printers, the following substitution is made:

\ for ¦

(4) The lowercase alphabet and indicated graphics are introduced by use of the type 0768–02 printer, which prints a 94-character set.

Table C–3. Punched Card, ASCII, and EBCDIC Codes (Part 1 of 5)

Character	Printed Symbol	Card Punches	ASCII		EBCDIC	
			Hexadecimal	Decimal	Hexadecimal	Decimal
Letters						
Uppercase A	A	12–1	41	65	C1	193
Uppercase B	B	12–2	42	66	C2	194
Uppercase C	C	12–3	43	67	C3	195
Uppercase D	D	12–4	44	68	C4	196
Uppercase E	E	12–5	45	69	C5	197
Uppercase F	F	12–6	46	70	C6	198
Uppercase G	G	12–7	47	71	C7	199
Uppercase H	H	12–8	48	72	C8	200
Uppercase I	I	12–9	49	73	C9	201
Uppercase J	J	11–1	4A	74	D1	209
Uppercase K	K	11–2	4B	75	D2	210
Uppercase L	L	11–3	4C	76	D3	211
Uppercase M	M	11–4	4D	77	D4	212
Uppercase N	N	11–5	4E	78	D5	213
Uppercase O	O	11–6	4F	79	D6	214
Uppercase P	P	11–7	50	80	D7	215
Uppercase Q	Q	11–8	51	81	D8	216
Uppercase R	R	11–9	52	82	D9	217
Uppercase S	S	0–2	53	83	E2	226
Uppercase T	T	0–3	54	84	E3	227
Uppercase U	U	0–4	55	85	E4	228
Uppercase V	V	0–5	56	86	E5	229
Uppercase W	W	0–6	57	87	E6	230
Uppercase X	X	0–7	58	88	E7	231
Uppercase Y	Y	0–8	59	89	E8	232
Uppercase Z	Z	0–9	5A	90	E9	233
Lowercase a	a	12–0–1	61	97	81	129
Lowercase b	b	12–0–2	62	98	82	130
Lowercase c	c	12–0–3	63	99	83	131

Table C–3. Punched Card, ASCII, and EBCDIC Codes (Part 2 of 5)

Character	Printed Symbol	Card Punches	ASCII		EBCDIC	
			Hexadecimal	Decimal	Hexadecimal	Decimal
Lowercase d	d	12–0–4	64	100	84	132
Lowercase e	e	12–0–5	65	101	85	133
Lowercase f	f	12–0–6	66	102	86	134
Lowercase g	g	12–0–7	67	103	87	135
Lowercase h	h	12–0–8	68	104	88	136
Lowercase i	i	12–0–9	69	105	89	137
Lowercase j	j	12–11–1	6A	106	91	145
Lowercase k	k	12–11–2	6B	107	92	146
Lowercase l	l	12–11–3	6C	108	93	147
Lowercase m	m	12–11–4	6D	109	94	148
Lowercase n	n	12–11–5	6E	110	95	149
Lowercase o	o	12–11–6	6F	111	96	150
Lowercase p	p	12–11–7	70	112	97	151
Lowercase q	q	12–11–8	71	113	98	152
Lowercase r	r	12–11–9	72	114	99	153
Lowercase s	s	11–0–2	73	115	A2	162
Lowercase t	t	11–0–3	74	116	A3	163
Lowercase u	u	11–0–4	75	117	A4	164
Lowercase v	v	11–0–5	76	118	A5	165
Lowercase w	w	11–0–6	77	119	A6	166
Lowercase x	x	11–0–7	78	120	A7	167
Lowercase y	y	11–0–8	79	121	A8	168
Lowercase z	z	11–0–9	7A	122	A9	169
Numerals						
0	0	0	30	48	F0	240
1	1	1	31	49	F1	241
2	2	2	32	50	F2	242
3	3	3	33	51	F3	243
4	4	4	34	52	F4	244
5	5	5	35	53	F5	245
6	6	6	36	54	F6	246

Table C–3. Punched Card, ASCII, and EBCDIC Codes (Part 3 of 5)

Character	Printed Symbol	Card Punches	ASCII		EBCDIC	
			Hexadecimal	Decimal	Hexadecimal	Decimal
7	7	7	37	55	F7	247
8	8	8	38	56	F8	248
9	9	9	39	57	F9	249
Symbols						
Exclamation point	!	12–8–7	21	33	4F	79
Quotation mark, dieresis	"	8–7	22	34	7F	127
Number sign, pound sign	#	8–3	23	35	7B	123
Dollar sign	$	11–8–3	24	36	5B	91
Percent sign	%	0–8–4	25	37	6C	108
Ampersand	&	12	26	38	50	80
Apostrophe, acute accent	'	8–5	27	39	7D	125
Opening parenthesis	(	12–8–5	28	40	4D	77
Closing parenthesis	)	11–8–5	29	41	5D	93
Asterisk	*	11–8–4	2A	42	5C	92
Plus sign	+	12–8–6	2B	43	4E	78
Comma, cedilla	,	0–8–3	2C	44	6B	107
Minus sign, hyphen	–	11	2D	45	60	96
Period, decimal point	.	12–8–3	2E	46	4B	75
Slash, virgule, solidus	/	0–1	2F	47	61	97
Colon	:	8–2	3A	58	7A	122
Semicolon	;	11–8–6	3B	59	5E	94
Less than	<	12–8–4	3C	60	4C	76
Equal sign	=	8–6	3D	61	7E	126
Greater than	>	0–8–6	3E	62	6E	110
Question mark	?	0–8–7	3F	63	6F	111
Commercial at symbol	@	8–4	40	64	7C	124
Opening bracket	[	12–8–2	5B	91	4A	74
Closing bracket	]	11–8–2	5D	93	5A	90
Reverse slash	\	0–8–2	5C	92	E0	224
Circumflex	^	11–8–7	5E	94	5F	95

Table C–3. Punched Card, ASCII, and EBCDIC Codes (Part 4 of 5)

Character	Printed Symbol	Card Punches	ASCII		EBCDIC	
			Hexadecimal	Decimal	Hexadecimal	Decimal
Underline	—	0–8–5	5F	95	6D	109
Grave accent	`	8–1	60	96	79	121
Opening brace	{	12–0	7B	123	C0	192
Closing brace	}	11–0	7D	125	D0	208
Vertical line	|	12–11	7C	124	6A	106
Overline, tilde	~	11–0–1	7E	126	A1	161

Character	Card Punches	ASCII		EBCDIC	
		Hexadecimal	Decimal	Hexadecimal	Decimal
Nonprintable Characters					
ACK (Acknowledge)	0–9–8–6	06	6	2E	46
BEL (Bell)	0–9–8–7	07	7	2F	47
BS (Backspace)	11–9–6	08	8	16	22
CAN (Cancel)	11–9–8	18	24	18	24
CR (Carriage return)	12–9–8–5	0D	13	0D	13
DC1 (Device control 1)	11–9–1	11	17	11	17
DC2 (Device control 2)	11–9–2	12	18	12	18
DC3 (Device control 3)	11–9–3	13	19	13	19
DC4 (Device control 4)	9–8–4	14	20	3C	60
DEL (Delete)	12–9–7	7F	127	07	7
DLE (Data link escape)	12–11–9–8–1	10	16	10	16
DS (Digit select)	11–0–9–8–1	80	128	20	32
EM (End of medium)	11–9–8–1	19	25	19	25
ENQ (Enquiry)	0–9–8–5	05	5	2D	45
EOT (End of transmission)	9–7	04	4	37	55
ESC (Escape)	0–9–7	1B	27	27	39
ETB (End of transmission block)	0–9–6	17	23	26	38
ETX (End of text)	12–9–3	03	3	03	3
FF (Form feed)	12–9–8–4	0C	12	0C	12
FS (File separator)	11–9–8–4	1C	28	1C	28

Table C–3. Punched Card, ASCII, and EBCDIC Codes (Part 5 of 5)

Character	Card Punches	ASCII		EBCDIC	
		Hexadecimal	Decimal	Hexadecimal	Decimal
FS (Field separator)	0–9–2	82	130	22	34
GS (Group separator)	11–9–8–5	1D	29	1D	29
HT (Horizontal tabulation)	12–9–5	09	9	05	5
LF (Line feed)	0–9–5	0A	10	25	37
NAK (Negative acknowledge)	9–8–5	15	21	3D	61
NUL (Null)	12–0–9–8–1	00	0	00	0
RS (Record separator)	11–9–8–6	1E	30	1E	30
SI (Shift in)	12–9–8–7	0F	15	0F	15
SO (Shift out)	12–9–8–6	0E	14	0E	14
SOH (Start of heading)	12–9–1	01	1	01	1
SOS (Significance start)	0–9–1	81	129	21	33
SP (Space)		20	32	40	64
STX (Start of text)	12–9–2	02	2	02	2
SUB (Substitute)	9–8–7	1A	26	3F	63
SYN (Synchronous idle)	9–2	16	22	32	50
US (Unit separator)	11–9–8–7	1F	31	1F	31
VT (Vertical tabulation)	12–9–8–3	0B	11	0B	11

Appendix D. Conventions for the Use of FORTRAN Library Routines

D.1. GENERAL

References to FORTRAN library routines may be included in the assembly language program. Both arithmetic and elementary analytic functions are available. For a list of the routines and their corresponding argument requirements, refer to the FORTRAN supplementary reference, UP-7693 (current version).

D.2. ROUTINE CALLING CONVENTIONS

The following paragraphs describe the conventions required for using the FORTRAN library routines.

D.2.1. Parameter List

The address of the parameter list must be placed in register 1. The parameter list contains the address or addresses of arguments, set up according to FORTRAN specifications, to be used by the library routine. Each entry in the parameter list consists of four bytes aligned on a full-word boundary. The last three bytes of each entry contain the 24-bit address of an argument. The first byte of each entry contains zeros unless it is the last parameter in the list. If it is the last parameter, the sign bit of the entry is set (contains hexadecimal 80).

D.2.2. Save Area

The address of an 18-word save area, aligned on a full-word boundary, must be placed in register 13 by the assembly language program. This storage is used by the library routine to save information, such as the entry point to this program, the address to which this program returns, register contents, and save area addresses used by programs other than the FORTRAN subroutine. This may be the same save area used by data management except when a FORTRAN routine is used within an exit from a data management routine.

D.2.3. Calling Sequence

A calling sequence must be coded to transfer control to the library routine which includes:

1. establishing an EXTRN for the particular FORTRAN library routine;
2. placing the address of the save area in register 13;
3. loading register 1 with the address of the parameter list;
4. placing the entry address of a called FORTRAN library routine in register 15; and
5. branching to register 15 and saving the address in register 14.

Upon return from the called library routine, register 15 contains the address of the result.

Example:

```
LABEL      OPERATION      OPERAND          COMMENTS
1          10             16
           EXTRN   ATAN2                  .FORTRAN ARCTANGENT LIBRARY ROUTINE
           LA      R13,SAVE               .LOAD ADDR OF 13 WORD SAVE AREA
           LA      R01,SADD               .LOAD ADDR OF PARAMETER LIST
           L       R15,=A(ATAN2)          .LOAD ADDR OF ARCTANGENT ROUTINE
           BALR    R14,R15
                                          {EXECUTE ARCTANGENT ROUTINE
                                          {RETURN TO USER ROUTINE AT
                                          {ADDRESS IN REGISTER 14
RETURN POINT
SARG       DC      X'81100000000C'
           DC      X'805000000000'
SAVE       DS      18F                    .CALCULATIONS FROM ARCTANGENT ROUTINE
SADD       DC      AL4(SARG)
           DC      XL1'80'                .INDICATES LAST PARAMETER
           DC      AL3(SAR615)
           END
```

D.3. INTERNAL VALUE REPRESENTATION

The types of variables and their internal representation used by the FORTRAN functions are as follows:

INTEGER – 5 bytes

range of integer = ±999999999

00	00	00	51	2	C	+512
00	00	97	64	3	D	–97643

REAL – 4-byte mantissa

1-byte exponent

mantissa range = ±9999999

exponent range = +127 to –128 biased by 128

81	31	41	00	0C
7E	31	41	00	0D

assumed decimal point

$3.141 = .3141 \times 10^{1}$

$-.003141 = -.3141 \times 10^{-2}$

DOUBLE PRECISION – 9-byte mantissa

1-byte exponent

mantissa range = ±99999999999999999

exponent range = +127 to –128 biased by 128

exponent

mantissa

88	97	65	43	21	12	30	00	00	0C

assumed decimal point

$97654321.123 = .97654321123 \times 10^{8}$

exponent

mantissa

78	12	47	99	00	00	00	00	00	0C

assumed decimal point

$.00000000124799 = .124799 \times 10^{-8}$

COMPLEX – 10 bytes

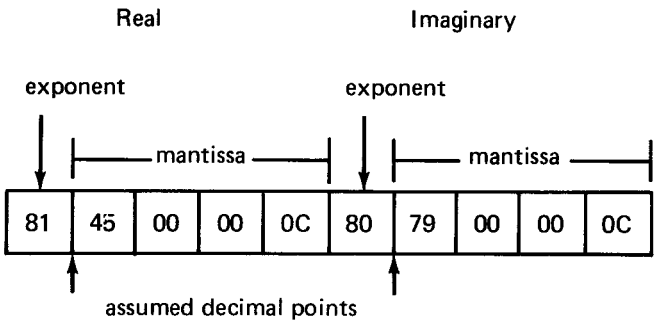
Real part: 4-byte mantissa

1-byte exponent

Imaginary part: 4-byte mantissa

1-byte exponent

The range of real and imaginary parts is the same as specified for the real number.



4.5 + .79i

Appendix E. Use of Param Statement

E.1. GENERAL

The user may include PARAM statements in a control stream to supply additional information to the assembler. The information is specified by means of operands in the PARAM statement. This appendix describes the operands and their use. It is an assembler function to process the PARAM statements; however, because the PARAM statement is a part of the control stream, it is a job control function to obtain each statement and make it available to the assembler.

An error detected in the operand field of a PARAM statement results in the automatic abort of the job step after any remaining operands in the statement and any additional PARAM statements are evaluated.

Error messages are written on the line printer and a message indicating a program abort is sent to the console typewriter.

Termination of PARAM statement processing occurs when any of the following conditions are detected:

1. an end-of-data response is given;
2. a start-of-data image is detected (/$); or
3. a slash is not detected in column 1.

The remaining statements in that job are bypassed until the first statement of the next job step is encountered; statement processing then continues.

E.2. PARAM STATEMENT OPERANDS

The following sections describe the options that may be specified in the operand field of the PARAM statements for an assembly operation.

E.2.1. IN — Source Library Input

This option is used to indicate the file on which the source program resides.

Format:

1	10
//ΔPARAM	IN = program-name/file-name

where:

program-name — is the source program name which can consist of up to eight characters.

file-name — is the file name of the source program and consists of up to eight characters. This name must be specified in an LFD control statement.

The file name may be any user-specified name. SCR2 may be submitted as file name for the assembler. If no file name is submitted, then SYSRES is assumed.

This option directs the assembler to search the file indicated for the specified program name in the source library. The tape being searched must be in standard library format. If the option is omitted, the source code will be in the control stream (preceded by a /$ statement and followed by a /* statement).

E.2.2. LIN – Referencing the Proc Library

This option is used to identify a library containing proc groups.

Format:

1	10
//ΔPARAM	LIN = { file-name(group$_1$,group$_2$,...) / (N) }

where:

file-name — identifies the file on which the input library resides. The name must appear on an LFD control statement and consists of up to eight characters.

group$_i$ — is a proc group number. A maximum of 10 groups may be specified.

N — indicates that a proc library is not to be searched (only procs defined within the source module being assembled will be considered).

An LIN statement other than LIN=(N) must specify at least one group number. Only those groups specified in the LIN statement are searched for proc definitions on the specified file. If a file name is not specified, the file searched will be that named PROC$.

A maximum of one file name may be specified. The file containing the procs must be in standard library format. The names OBJFIL, SCR1, SCR2, or SCR3 are not permitted.

If an LIN statement is not specified, processing proceeds as though LIN=(1) had been specified. The procs accessed are those in the first group PROC$.

E.2.3. LST – Selecting List Options

This option allows the programmer to indicate the types of listings desired.

Format:

1	10
// PARAM	LST = (spec$_1$,spec$_2$,...)

where:

$spec_i$ may be any or all of the following:

B – produce debug mode proc generation

C – produce cross-reference listing

H – provide a boundary alignment check for the operand address of RX, SI, and RX instructions. If the addresses violate the boundary alignment rules, an H flag will be generated by the assembler. The check is made based on the effective addresses; i.e., the assembler checks the value which results from adding the covering register value and the displacement value.

N – inhibit all output listings

O – produce object program listing with corresponding source code (code-edit) and external symbol dictionary

P – list source code defined procs

W – inhibit the listing of warning diagnostics (errors identified by a severity code of P)

If a single specification is written, parentheses are not required.

If an LST option is not submitted, the assembler produces an object program listing, diagnostic errors, and a nondebug proc mode listing.

E.2.4. OUT – Output Module Type

This option allows the programmer to determine whether the output of the assembler is to be an object module or a load module. It also allows the object module to be copied to a tape.

Format:

1	10
//ΔPARAM	OUT = {(L) (N) (T) (P)}

where:

(L) indicates a load module is to be produced.

(N) indicates that writing of a primary output for the assembler is to be inhibited.

(T) indicates that the object module generated by the assembler is to be copied from MCL to a tape specified as OBJFIL.

(P) indicates a loadable load module is to be produced with a prefix loader which will not be processed by the linkage editor. This need not be used for the disc assembler; if L and T are specified, the module is unconditionally produced with a prefix loader.

If L is specified, the load module name generated is eight characters in length, left-justified, and zero filled if less than eight characters.

If the OUT option is not specified, an object module is produced in the MCL only by the disc assembler. The resultant object module must be processed by the linkage editor before being executed. The object module name generated is eight characters in length, left-justified, and blank filled, if less than eight characters.

If the L parameter is used, the source program being assembled is subject to the following restrictions:

- Unresolved EXTRN references are not permitted.
- I/O protect is not provided.
- V-type constants are not permitted.
- Common directives are not permitted.
- Only single phase load modules are permitted.
- The resultant load module may not be processed by the linkage editor.

E.2.5. VER – Version Number

This option allows a change to be made to the version number of an object module.

Format:

1	10
//ΔPARAM	VER = level-number[/update-number]

where:

level-number is a 1- or 2-digit decimal number (0–99) representing a level number.

update-number is a 1- or 2-digit decimal number (0–99) representing an update number.

The assembler converts the version number to one or two packed decimal digits (as required) and stores it in the version number field in the object module header and sets the least significant bytes (the update number portion) to 0 if the update number is not specified with the VER option.

If the VER option is not specified, the assembler assigns a version number as follows:

- The version number field is set to packed decimal zeros in the object module header if the source module was submitted in the job stream.
- The version number inserted into the object module header is identical to that of the source module if the source module was submitted from tape.

E.2.6. CDE – Produce Compatible Code

The OS/4 assembler produces code intended to be run on the SPERRY UNIVAC 90/60,70. If it is desired to assemble a program which can run on either the 90/60,70 or the SPERRY UNIVAC 9400/9480, the following PARAM statement should be included in the control stream.

Format:

1	10
//ΔPARAM	CDE = { 94 / (94) }

This statement, in either format, causes the assembler to:

- flag as an illegal operation code any instruction not in the repertoire of the UNIVAC 9400, and
- assemble the AI instruction with the hexadecimal operation code (X'93') that is appropriate for execution on the 9400/9480.

In this mode, the only legal instruction mnemonic operation codes are:

A	BCTR	ED	LTR	NR	SLM	STM
AH	C	HPR	MP	O	SP	SVC
AI	CH	IC	MVC	OC	SPM	TM
AP	CL	L	MVI	OI	SR	TR
AR	CLC	LA	MVN	OR	SRL	UNPK
BAL	CLI	LH	MVO	PACK	SSM	X
BALR	CLR	LLR	MVZ	S	SSTM	XC
*BC	CP	LM	N	SH	ST	XI
*BCR	CR	LPSW	NC	SIO	STC	XR
BCT	DP	LR	NI	SLL	STH	ZAP

*The extended mnemonic operation codes associated with these instructions are also legal.

E.2.7. RO$ – Suppressing Covering Error Flag

When the RO$ PARAM statement is specified, the disc assembler will not generate a C flag on the instruction line for either of the following conditions:

- Displacement field has an absolute value that is less than 4096.
- Displacement field has a relocatable value that is less than 4096.

If the RO$ PARAM statement is omitted, the disc assembler generates C flags for both of the aforementioned conditions.

Format:

1	10
//ΔPARAM	RO$=YES

The following example illustrates typical situations or conditions in which source statements are flagged as covering errors.

Example:

```
                LABEL      ΔOPERATIONΔ          OPERAND                    Δ
                1          10           16
1.              TEST       START        2048
2.                         LA           1,35
3.                         LA           1,*
4.                         USING        0,0
5.                         LA           1,35
6.                         LA           1,*
7.                         DROP         0
8.                         USING        *,0
9.                         LA           1,35
10.                        LA           1,*
```

If R0$=YES is not specified, the source statements coded on lines 2, 3, 6, and 10 are flagged as covering errors. If R0$=YES is specified, however, only coding lines 3 and 6 are flagged as covering errors. Flags are not generated for coding lines 2 and 10, respectively, because R0$=YES implies that absolute displacements less than 4096 are allowed (line 2); and the USING *,0 and R0$=YES are both in effect (line 10). Coding lines 3 and 6 are still in error because the appropriate USING directives are not in effect.

Appendix F. Executing the Assembler

F.1. GENERAL

The assembler software elements and the hardware systems to which each is related are listed in Table F–1.

Table F–1. Assembler Software Element Names

Software Element	Name	
	9400/9480 Systems	90/60,70 Systems
Tape assembler	ASM	
IBM 360 compatible tape assembler	ASMC	
Basic tape assembler	BASM	
Basic disc assembler	BDASM	
Disc assembler	DASM	
Disc assembler		DASM4
IBM 360 compatible disc assembler	DASMC	
Disc/tape assembler	DTASM	

F.2. JOB CONTROL STREAMS

Sample job control streams for executing assembler software elements are provided in the current versions of the 9400/9480 operations handbook, UP-7871 and the 90/60,70 operations handbook, UP-7937.

F.3. MAIN STORAGE REQUIREMENTS

The assembler utilizes additional main storage when it is assigned. The following effects are realized.

- Additional space is assigned for the variable symbols table which permits a greater number of variable symbols to be defined. This includes PROC labels, DO labels, SET symbols, PROC parameters, and system SET symbols.

- Reduces the recycling of cross-reference listing.
- Increases performance by reducing assembly time through the use of additional tables and I/O buffers.

F.4. SPECIAL CONSIDERATIONS AND RESTRICTIONS

- Blank Cards

  Blank cards cannot be placed between PROC definitions appearing in a source code deck or between the last source defined PROC and the first statement of the source deck. PROC definitions appearing in a source deck must immediately precede the START directive.

- Restrictions for BASM and BDASM Software Elements

  The BASM and BDASM software elements do not support the following PARAM statement options:

  ```
  // PARAM LST=(N)

  // PARAM OUT=(N)
  ```

- Restrictions for ASM, BASM, BDASM, and DTASM Elements

  The preservation of the UPSI byte applies only to job steps and not to the entire job.

- Restrictions for ASM, BASM, and BDASM Elements

  A restriction in the use and processing of local and global set symbols exists in ASM, BASM, and BDASM elements. This restriction is due primarily to the fact that the DTF macro instruction for index sequential access method (DTFIS) makes extensive use of these symbols and therefore approaches the current limit of the assembler to process such symbols. Consequently, if a user is also using global or local set symbols in his source module, he may exceed the limit of the assembler. If the user source module does not include any global or local set symbols, but does include a DTFIS call, then the assembler limit is not exceeded unless the source module includes calls to macros which define other global set symbols. If, however, global and/or local set symbols are declared in the source module, it is then possible that the global set symbols previously defined may become undefined. In the event that this occurs, the DTFIS macro call must be assembled separately and the appropriate linkage established with the processing modules.

  - Five levels only of DO nesting are permitted.
  - Replacement of parameter or set symbols in the operations field is not permitted.
  - When specifying a TITLE directive that is shorter in length than that previously defined by a TITLE directive, make certain that the new directive contains enough trailing blanks to make the new length equal to the previously defined TITLE. This requirement is necessary to insure that the buffer for the TITLE directive is properly reinitialized.
  - The PRINT directive is not implemented.
  - The number of EXTRN and ENTRY directives processed by this version of the assembler is limited to 50 each.
  - The ICTL directive is not supported.

- Restrictions for BDASM Element

  No single SYSPOOL support is provided and use of the 8424/8425 disc unit is restricted.

- Restriction of Specifying a Literal in an Address Constant

  The assembler terminates abnormally whenever it encounters a literal specified as an operand of a supervisor imperative macro instruction. Assembler design does not allow a literal to be specified as an address constant.

- File Overflow Error Condition

  Whenever it is unable to recycle EXTRN and ENTRY records due to an overflow of allotted space on SYSPOOL, the assembler causes a FILE OVERFLOW error condition to be displayed on the console, produces a main storage dump, and terminates processing. The corrective action to be taken for such a situation consists of remapping SYSPOOL by use of DACMAP and resubmitting the job. If the problem persists, additional SYSPOOL space must be allotted to the job.

- Incorrect Proc Processing

  The disc tape assembler (DTASM) may incorrectly process the last proc of a tape proc file due to the failure of a WAIT instruction to be issued. This condition may be prevented by including a dummy proc as the last proc of the file.

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