SPERRY UNIVAC
COMPUTER SYSTEMS

ATTN: CHARLIE GIBBS

00014
CAV208M45541 UP 8913-B

UAS

SPERRY UNIVAC
1 - 1818 CORNWALL STREET
VANCOUVER B C

V6J 1C7

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Assembler

User Guide

UP-8913-B

This Library Memo announces the release and availability of Updating Package B to "SPERRY UNIVAC Operating System/3 (OS/3) Assembler User Guide", UP-8913.

This update provides the 8.0 release enhancements to OS/3 Assembler. The enhancements include:

- The addition of two privileged instructions (Put IORB, Get IORB)
- An additional UPSI byte setting for diagnostic errors
- The addition of STXIT island code (providing the capability to continue job streams when program checks occur)
- The display of error messages on the console or workstation
- An additional warning message when using continuation characters with macroinstructions (a comma after the last operand is checked)

Appendix A was expanded to include job control information.

Copies of Updating Package B are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-8913-B. To receive the complete manual, order UP-8913.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists B00, B18, 28U and 29U (Package B to UP-8913, 192 pages plus Memo)	Library Memo for UP-8913-B
		RELEASE DATE: September, 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

SPERRY UNIVAC
SUITE 906
1177 WEST HASTINGS ST
VANCOUVER BC V6E 2K3
ATTN: CHARLIE GIBBS
00158
CAV208M45541 UP 8913-A

UAS

CAV

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Assembler

User Guide

UP-8913-A

This Library Memo announces the release and availability of Updating Package A to "SPERRY UNIVAC Operating System/3 (OS/3) Assembler User Guide", UP-8913.

This update discusses assembler features that are new to release 7.1. These features include:

- Added instructions: IPL and SWLS
- A new option for the ASM listing parameter: LST=NR
- ASM, ASML, and ASMLG jproc enhancements to accept cataloged files for input, output, macro library, COPY source code library, and alternate load library files.

The following instructions were deleted: ENQ, DEQ, STEP, and MSS.

All other changes are corrections to, or clarifications of, material applicable prior to release 7.1.

Copies of Updating Package A are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-8913-A. To receive the complete manual, order UP-8913.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ (less DE, GZ and HA) and MZ	Mailing Lists DE, GZ, HA, 28U and 29U (Package A to UP-8913, 83 pages plus Memo)	Library Memo for UP-8913 RELEASE DATE: September, 1981

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS RELEASE

Operating System/3 (OS/3)

Assembler

User Guide

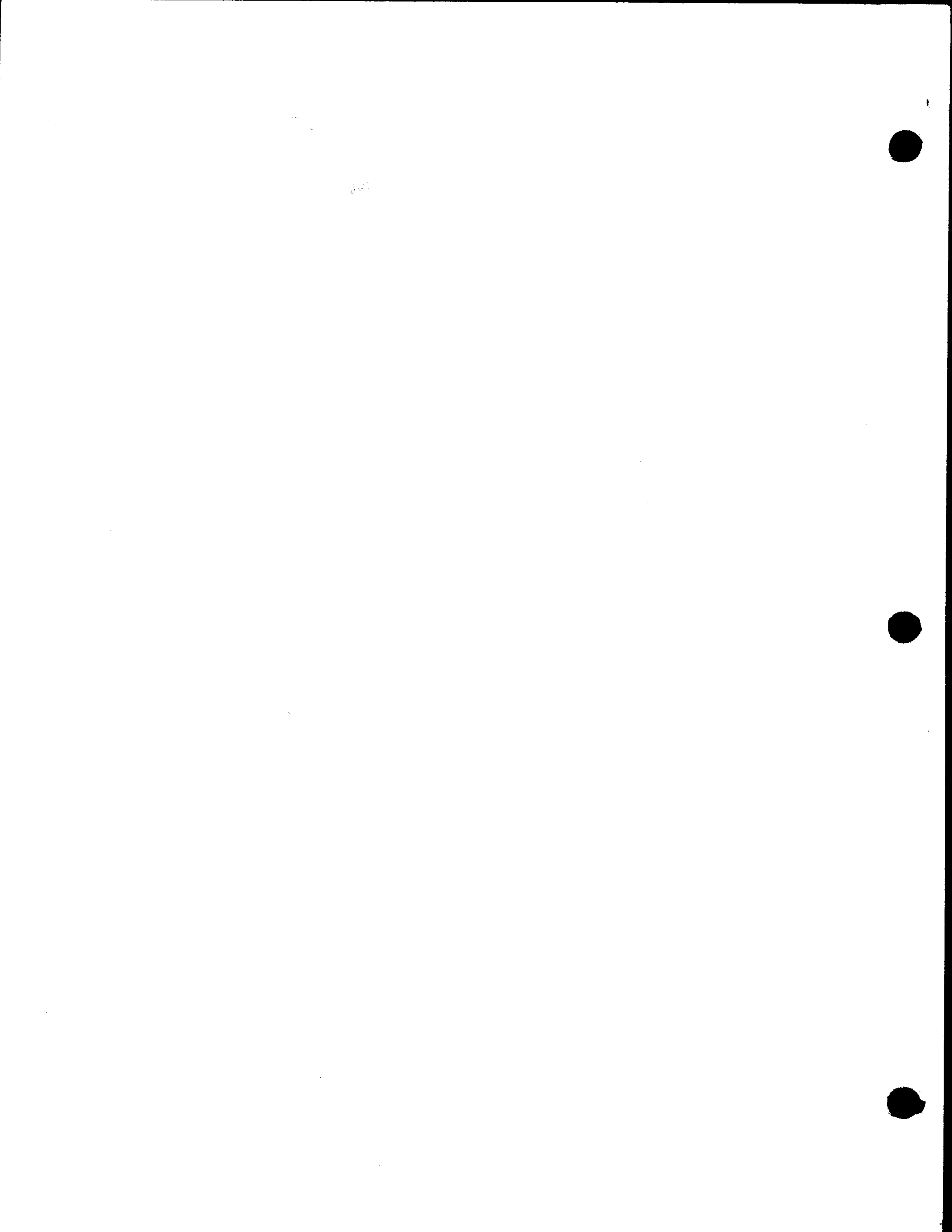
UP-8913

This Library Memo announces the release and availability of "SPERRY UNIVAC® Operating System/3 (OS/3) Assembler User Guide", UP-8913.

This manual describes the assembler. It covers machine instruction and data formats, assembler directives, macro/proc usage, and assembler output.

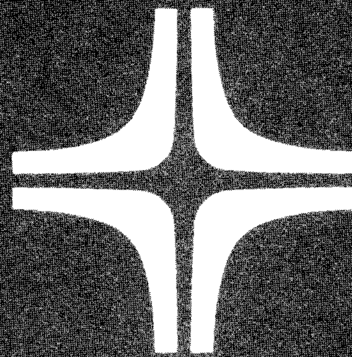
Additional copies may be ordered by your local Sperry Univac representative.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS:
Mailing Lists BZ,CZ (less DE,GZ and HA) and MZ	Mailing Lists DE, GZ, HA, 28U and 29U (Covers and 886 pages)	Library Memo
		RELEASE DATE: October, 1980



Assembler

OS/3



User Guide

Environment: System 80

SPERRY UNIVAC

UP-8913

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please consult the appropriate release documentation or contact your local Sperry Univac representative.

Sperry Univac reserves the right to modify or revise the content of this document. No contractual obligation by Sperry Univac regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry Univac.

Sperry Univac is a division of the Sperry Corporation.

FASTRAND, SPERRY UNIVAC, UNISCOPE, UNISERVO, and UNIVAC are registered trademarks of the Sperry Corporation. ESCORT, PAGEWRITER, PIXIE, and UNIS are additional trademarks of the Sperry Corporation.

This document was prepared by Systems Publications using the SPERRY UNIVAC UTS 400 Text Editor. It was printed and distributed by the Customer Information Distribution Center (CIDC), 555 Henderson Rd., King of Prussia, Pa., 19406.

©1980 — SPERRY CORPORATION

PRINTED IN U.S.A.

PAGE STATUS SUMMARY

ISSUE: Update C – UP-8913
RELEASE LEVEL: 8.2 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1 thru 3	C
Preface	1	Orig.
	2	B
	3	Orig.
Contents	1 thru 3	Orig.
	4	A
	5 thru 7	Orig.
	8 thru 11	B
	12 thru 14	A
	15 thru 18	B
PART 1	Title Page	Orig.
1	1 thru 11	Orig.
	12	B
	13	C
	14 thru 20	Orig.
2	1 thru 10	Orig.
3	1 thru 3	Orig.
4	1	Orig.
	2	C
	3 thru 9	Orig.
	10	C
	11 thru 13	Orig.
	14	B
	15 thru 20	Orig.
PART 2	Title Page	Orig.
5	1, 2	Orig.
	3	C
	4 thru 10	Orig.
	11, 12	B
	13 thru 18	Orig.
	19	C
6	1 thru 3	A
	4	Orig.
PART 3	Title Page	Orig.
7	1 thru 3	B
	4, 5	C
	6	Orig.
8	1 thru 13	Orig.
	14, 15	B
8 (cont)	16, 17	Orig.
	18	B
	19	Orig.
	20	B
	21	C
	22	Orig.
	23	B
9	1 thru 8	Orig.
	9	B
	10 thru 12	Orig.
	13	B
	14 thru 29	Orig.
	30	A
	31 thru 62	A*
	63 thru 80	Orig.
	81	B
	82 thru 100	Orig.
10	1, 2	Orig.
	3 thru 5	B
	6 thru 18	Orig.
	19	C
	20 thru 23	Orig.
	24	C
	25 thru 33	Orig.
	34	B
	35 thru 39	Orig.
	40	C
	41	Orig.
	42	C
	43	Orig.
	44	C
	45, 46	Orig.
	47	B
	48, 49	Orig.
	50	B
	51 thru 54	Orig.
	55	C
	56 thru 72	Orig.
	73	B
	74, 75	Orig.
	76	C
	77	Orig.
	78	C
	79	B
	80	C
	81, 82	Orig.
	83	C
	84 thru 87	Orig.
	88	C
	89, 90	Orig.
11	1, 2	Orig.
	3	C
	4, 5	Orig.
	6	C
11 (cont)	7 thru 11	Orig.
	12	C
	13, 14	Orig.
	15	C
	16	Orig.
	17	C
	18	Orig.
	19	C
	20	Orig.
	21	C
	22	Orig.
	23 thru 25	C
	26	Orig.
	27 thru 29	C
	30	Orig.
	31	C
	32, 33	Orig.
	34	C
	35	Orig.
	36	C
	37	Orig.
	38	C
	39 thru 41	Orig.
	42	C
	43	Orig.
	44	C
	45	Orig.
	46	C
	47	Orig.
	48	C
	49	Orig.
	50	C
	51	Orig.
	52	C
	53	Orig.
	54	C
	55	Orig.
	56	C
	57	Orig.
	58	C
	59	Orig.
	60	C
	61	Orig.
	62	C
	63	Orig.
	64	C
	65	Orig.
	66	C
	67	Orig.
	68	C
	69	Orig.
	70	C
	71	Orig.
	72	B
	73	Orig.
	74	C

*Deleted

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

PAGE STATUS SUMMARY

ISSUE: Update C – UP-8913
RELEASE LEVEL: 8.2 Forward

Part/Section	Page Number	Update Level
11 (cont)	75	Orig.
	76	C
	77	Orig.
	78	C
	79	Orig.
	80	C
	81	Orig.
	82	C
	83	Orig.
	84	C
	85	Orig.
	86	C
	87	Orig.
	88	C
	89 thru 91	Orig.
	92	C
	93	Orig.
	94	C
	95	Orig.
12	1, 2	Orig.
	3	B
	4	Orig.
	5, 6	B
	7	C
	8	B
	9 thru 16	Orig.
	17	B
	18	Orig.
	19	C
	20 thru 43	Orig.
	44	C
	45, 46	Orig.
	47	C
	48	Orig.
	49	C
	50	Orig.
	51	B
	52 thru 59	Orig.
	60	B
	61	Orig.
	62	B
	63	Orig.
	64	B
	65 thru 67	Orig.
	68	B
	69 thru 72	Orig.
	73	B
	74	Orig.
	75	B
	76 thru 81	Orig.
	82	B
	83, 84	Orig.
	85, 86	B
	87	Orig.
	88	B
	89 thru 99	Orig.

Part/Section	Page Number	Update Level
12 (cont)	100, 101	B
	102	Orig.
	103	B
	104, 105	Orig.
	106	B
	107 thru 111	Orig.
	112	B
	113	Orig.
	114, 115	B
	116 thru 118	Orig.
	119	B
	120 thru 131	Orig.
13	1	Orig.
	2	B
	3	Orig.
	4	B
	5	Orig.
	6, 7	B
	8 thru 12	Orig.
	13	B
	14	Orig.
	14a, 14b	B
	15	Orig.
	16	B
	17, 18	Orig.
	19	B
	20 thru 23	Orig.
	24	B
	25	Orig.
	26	A
	26a, 26b	A
	27	B
	28	Orig.
	29	B
	30	Orig.
	31	B
	32	Orig.
14	1	A
PART 4		
	Title Page	Orig.
15	1	Orig.
16	1 thru 3	Orig.
17	1 thru 9	Orig.
18	1 thru 4	Orig.
19	1 thru 9	Orig.
	10, 11	B
	12 thru 14	Orig.

Part/Section	Page Number	Update Level
20	1 thru 6	Orig.
21	1	B
	2 thru 10	Orig.
PART 5		
	Title Page	Orig.
22	1 thru 4	Orig.
23	1 thru 5	Orig.
	6 thru 8	B
	9, 10	Orig.
24	1 thru 5	Orig.
	6	B
	7 thru 12	Orig.
25	1 thru 9	Orig.
	10	A
	11	Orig.
26	1 thru 7	Orig.
27	1 thru 16	Orig.
	17	B
	18, 19	Orig.
	20	B
	21 thru 32	Orig.
PART 6		
	Title Page	Orig.
28	1 thru 4	Orig.
	5	B
PART 7		
	Title Page	Orig.
29	1	Orig.
	2	A
	3	Orig.
	4 thru 6	B
	7, 8	A
	8a	A
	9, 10	B
	10a	B
	11, 12	B
	13	A
	14	B
	14a	B
	15 thru 19	Orig.
30	1 thru 5	Orig.
PART 8		
	Title Page	Orig.

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

PAGE STATUS SUMMARY

ISSUE: Update C – UP-8913
RELEASE LEVEL: 8.2 Forward

Part/Section	Page Number	Update Level
Appendix A	1 thru 4	B
Appendix B	1, 2	B
	3 thru 7	Orig.
Appendix C	1 thru 9	Orig.
Appendix D	1 thru 3	Orig.
Appendix E	1	Orig.
	2 thru 4	B
	5	Orig.
	6, 7	B
	8	A
	9	B
	10	A
	11	B
	12, 13	Orig.
	14	A
	15	B
	16	A
Appendix F	1 thru 3	Orig.
	4	A
	5 thru 8	Orig.
Appendix G	1 thru 5	Orig.
Index	1, 2	Orig.
	3	A
	4 thru 7	Orig.
	8, 9	A
	10	B
	11	Orig.
	12	B
	13	A
	14	Orig.
	15	A
	16 thru 18	Orig.
	19	A
	20, 21	Orig.
	22, 23	B
	24	Orig.
	25, 26	B
	27	Orig.
	28	B
	29	A
	30, 31	Orig.
User Comment Sheet		

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.

Preface

This manual is one of a series designed to instruct and guide the programmer in the use of the SPERRY UNIVAC Operating System/3 (OS/3). This manual specifically describes the OS/3 assembler and its effective use. Its intended audience is the novice programmer with a basic knowledge of data processing, but with limited programming experience, and the assembler programmer whose experience is limited to systems other than Sperry Univac.

Two other manuals are available that cover OS/3 assembler; one is an introductory manual and the other is a programmer reference. The introductory manual briefly describes OS/3 assembler and its facilities. The programmer reference provides the characteristics of OS/3 assembler in skeletal form and is intended as a quick-reference document for the programmer experienced in the use of OS/3 assembler.

This user guide is divided into the following parts:

- PART 1. BASIC DATA AND CONVENTIONS

  Introduces you to what basic assembly language (BAL) is, how the computer stores information (data), how to locate the data required, and what forms mathematic notations assume in computer manipulations. The general rules that a programmer must understand to solve simple BAL problems are stated in this part. Where the content may seem out of context to the more experienced user, he can find such material repeated in greater detail in the following parts of this manual. As the manual progresses, the problems and examples become increasingly complex.

- PART 2. STORAGE AND SYMBOL DEFINITIONS

  Describes and illustrates the use of storage assignments, the constants, and the rules for symbol designations.

- PART 3. BAL APPLICATION INSTRUCTIONS

  Presents the explicit and implicit formats for all the assembly language application instructions, the rules of their use and illustrative examples.

- PART 4. BAL DIRECTIVES

  Describes and illustrates the use of all the assembler control directives.

- PART 5. BAL MACROS

  Explains the macro facility for writing and using this time-saving feature of the assembler.

- PART 6. ASSEMBLY LISTING

  Describes what an assembly listing is, what it means, and how it is of use to the programmer.

- PART 7. PROGRAMMING TECHNIQUES

  A series of programming techniques are presented in this part.

- PART 8. APPENDIXES

  The appendixes contain figures and tables for use in explaining the text and for general programmer reference.

Each of the foregoing parts consists of one or more sections that cover the different aspects of the subject matter covered in each part.

Other current OS/3 publications for the System 80 system, referenced in this manual, will be necessary or useful to the programmer working with the assembler.

Document name and number	Description
General editor user guide/programmer reference, UP-8828	Describes the general editor
System services program user guide, UP-8841	Describes the librarian and linkage editor
Consolidated data management concepts and facilities user guide, UP-8825	Presents an overview of data management
Consolidated data management macro-instructions user guide/programmer reference, UP-8826	Describes the data management macroinstructions
Supervisor concepts and facilities user guide, UP-8831	Presents an overview of the supervisor
Supervisor macroinstructions user guide/programmer reference, UP-8832	Describes the supervisor macroinstructions

Document name and number	Description
Processor programmer reference, UP-8881	Describes the processor
System hardware and software summary, UP-8868	Presents an overview of the system hardware and software
Interactive job control user guide, UP-8822	Describes interactive job control
Workstation user guide, UP-8845	Describes the OS/3 workstation
Assembler programmer reference, UP-8914	Describes the assembler

Contents

PAGE STATUS SUMMARY

PREFACE

CONTENTS

PART 1. BASIC DATA AND CONVENTIONS

1. INTRODUCTION TO ASSEMBLER LANGUAGE PROGRAMMING

1.1.	CODING AN ASSEMBLER PROGRAM	1-1
1.1.1.	Operation Field	1-5
1.1.2.	Operand Field	1-6
1.1.3.	Label Field	1-7
1.1.4.	Comments Field	1-9
1.1.5.	Sequence Numbers	1-10
1.1.6.	Column 72	1-11
1.1.7.	Additional Coding Rules	1-11
1.2.	ASSEMBLING A PROGRAM	1-14
1.3.	CREATING A LOAD MODULE	1-18
1.4.	PROGRAM EXECUTION	1-20

2. DATA FORMS

2.1.	DATA REPRESENTATION	2-1
2.2.	BINARY REPRESENTATION	2-2

2.3.	HEXADECIMAL REPRESENTATION	2-3
2.4.	CHARACTER REPRESENTATION	2-5
2.4.1.	Alphabetic Characters	2-5
2.4.2.	Special Letters	2-6
2.4.3.	Numeric	2-6
2.4.3.1.	Unpacked Format	2-6
2.4.3.2.	Packed Format	2-7
2.4.4.	Special Characters	2-8
2.5.	FIXED-POINT NUMBERS	2-9
2.6.	FLOATING-POINT NUMBERS	2-9
3.	**ADDRESSING**	
3.1.	MAIN COMPUTER STORAGE ADDRESSING	3-1
3.1.1.	Instruction Addressing	3-1
3.1.2.	Data Field Addressing	3-2
3.2.	REGISTER ADDRESSING	3-3
4.	**RULES AND CONVENTIONS**	
4.1.	READING INSTRUCTION NOTATION	4-1
4.1.1.	Assembler Application Instruction Notations	4-1
4.1.2.	Notation Rules and Meanings	4-5
4.2.	TERMS	4-8
4.2.1.	Self-Defining Terms (SDT)	4-9
4.2.2.	Literals	4-10
4.2.3.	Symbols	4-11
4.2.4.	Location Counter References	4-12
4.2.5.	Length Attribute Reference	4-13
4.3.	OPERATORS	4-13
4.3.1.	Arithmetic Operators	4-14
4.3.2.	Logical Operators	4-15
4.3.3.	Relational Operators	4-15
4.4.	EXPRESSIONS	4-16
4.4.1.	Absolute Expressions	4-16
4.4.2.	Relocatable Expressions	4-17
4.4.3.	Complex Relocatable Expressions	4-18
4.4.4.	Character Expressions	4-18
4.4.5.	Length Attribute of Expressions	4-19
4.4.6.	Character Strings	4-19

PART 2. STORAGE AND SYMBOL DEFINITIONS

5. STORAGE DEFINITIONS

5.1.	STORAGE USAGE		5–1
5.1.1.	Define Constant	(DC)	5–4
5.1.2.	Define Storage	(DS)	5–5
5.1.3.	Duplication Factor		5–5
5.1.4.	Definition Type		5–6
5.1.5.	Length Factor	(L_n)	5–6
5.1.6.	Constant Specification		5–7
5.1.7.	Alignment		5–8
5.2.	DEFINITION TYPES		5–8
5.2.1.	Character Constants	(C)	5–8
5.2.2.	Hexadecimal Constants	(X)	5–9
5.2.3.	Binary Constants	(B)	5–9
5.2.4.	Packed Decimal Constants	(P)	5–10
5.2.5.	Zoned Decimal Constants	(Z)	5–10
5.2.6.	Half-Word Fixed-Point Constants	(H)	5–11
5.2.7.	Full-Word Fixed-Point Constants	(F)	5–12
5.2.8.	Half-Word Address Constants	(Y)	5–12
5.2.9.	Full-Word Address Constants	(A)	5–13
5.2.10.	Base and Displacement Constants	(S)	5–13
5.2.11.	External Address Constants	(V)	5–15
5.2.12.	Floating-Point Constants	(E and D)	5–15
5.3.	LITERALS		5–18

6. SYMBOL DEFINITIONS

6.1.	EQUIVALENT SYMBOLS	6–2
6.2.	SYMBOL APPLICATIONS	6–3

PART 3. BAL APPLICATION INSTRUCTIONS

7. INTRODUCTION TO APPLICATION INSTRUCTIONS

7.1.	INSTRUCTION AND FORMAT CONVENTIONS	7–1
7.2.	EXPLICIT FORMS	7–6
7.3.	IMPLICIT FORMS	7–6
7.4.	DEFINITIONS OF FORMAT TERMS	7–6

8. BRANCHING INSTRUCTIONS

8.1.	USE OF BRANCHING INSTRUCTIONS		8-1
8.2.	EXTENDED MNEMONIC CODES		8-2
8.3.	BRANCH AND LINK	(BAL, BALR)	8-5
8.3.1.	Use of the BALR Instruction in Base Register Assignment		8-7
8.4.	BRANCH ON CONDITION	(BC, BCR)	8-9
8.5.	BRANCH ON COUNT	(BCT, BCTR)	8-13
8.6.	BRANCH ON INDEX HIGH	(BXH)	8-15
8.7.	BRANCH ON INDEX LOW OR EQUAL	(BXLE)	8-18
8.8.	EXECUTE	(EX)	8-20

9. DECIMAL AND LOGICAL INSTRUCTIONS

9.1.	USING DECIMAL INSTRUCTIONS		9-1
9.2.	DEFINING PACKED AND UNPACKED CONSTANTS AND MAIN STORAGE AREAS		9-3
9.2.1.	Packed Decimal Constants and Main Storage Areas		9-4
9.2.2.	Unpacked Decimal Constants and Main Storage Areas		9-6
9.3.	ADD DECIMAL	(AP)	9-8
9.4.	COMPARE DECIMAL	(CP)	9-10
9.5.	DIVIDE DECIMAL	(DP)	9-13
9.6.	EDIT	(ED)	9-16
9.6.1.	The Edit Pattern		9-17
9.6.2.	The Resulting Condition Code		9-23
9.6.3.	Examples of General Usage		9-24
9.6.4.	Summary		9-26
9.7.	EDIT AND MARK	(EDMK)	9-27
9.8.	Deleted (MSS)		9-30

9.9.	MOVE CHARACTER	(MVC)	9–63
9.10.	MOVE CHARACTER LONG	(MVCL)	9–67
9.11.	MOVE NUMERICS	(MVN)	9–72
9.12.	MOVE WITH OFFSET	(MVO)	9–75
9.13.	MOVE ZONES	(MVZ)	9–78
9.14.	MULTIPLY DECIMAL	(MP)	9–80
9.15.	PACK DECIMAL	(PACK)	9–82
9.16.	SUBTRACT DECIMAL	(SP)	9–86
9.17.	SHIFT AND ROUND DECIMAL	(SRP)	9–89
9.18.	UNPACK DECIMAL	(UNPK)	9–95
9.19.	ZERO AND ADD DECIMAL	(ZAP)	9–98

10. FIXED-POINT BINARY INSTRUCTIONS

10.1.	USE OF FIXED-POINT BINARY INSTRUCTIONS		10–1
10.1.1.	Half-Word Fixed-Point Constants		10–3
10.1.2.	Full-Word Fixed-Point Constants		10–4
10.1.3.	Address Constants		10–4
10.1.3.1.	Full-Word Address Constants		10–4
10.1.3.2.	Half-Word Address Constants		10–5
10.1.4.	Representation of Positive and Negative Fixed-Point Binary Numbers		10–6
10.2.	ADD	(A)	10–7
10.3.	ADD	(AR)	10–9
10.4.	ADD HALF WORD	(AH)	10–11
10.5.	ADD IMMEDIATE	(AI)	10–13
10.6.	COMPARE	(C)	10–15
10.7.	COMPARE	(CR)	10–17
10.8.	COMPARE HALF WORD	(CH)	10–19
10.9.	CONVERT TO BINARY	(CVB)	10–21
10.10.	CONVERT TO DECIMAL	(CVD)	10–24
10.11.	DIVIDE	(D)	10–27
10.12.	DIVIDE	(DR)	10–31

10.13.	LOAD	(L)	10-33
10.14.	LOAD	(LR)	10-35
10.15.	LOAD AND TEST	(LTR)	10-38
10.16.	LOAD COMPLEMENT	(LCR)	10-40
10.17.	LOAD HALF WORD	(LH)	10-42
10.18.	LOAD MULTIPLE	(LM)	10-44
10.19.	LOAD NEGATIVE	(LNR)	10-50
10.20.	LOAD POSITIVE	(LPR)	10-52
10.21.	MULTIPLY	(M)	10-55
10.22.	MULTIPLY	(MR)	10-59
10.23.	MULTIPLY HALF WORD	(MH)	10-61
10.24.	SHIFT LEFT DOUBLE	(SLDA)	10-64
10.25.	SHIFT LEFT SINGLE	(SLA)	10-67
10.26.	SHIFT RIGHT DOUBLE	(SRDA)	10-70
10.27.	SHIFT RIGHT SINGLE	(SRA)	10-73
10.28.	STORE	(ST)	10-76
10.29.	STORE HALF WORD	(STH)	10-78
10.30.	STORE MULTIPLE	(STM)	10-80
10.31.	SUBTRACT	(S)	10-83
10.32.	SUBTRACT	(SR)	10-85
10.33.	SUBTRACT HALF WORD	(SH)	10-88

11. FLOATING-POINT INSTRUCTIONS

11.1.	INTRODUCTION		11-1
11.2.	ADD NORMALIZED, LONG FORMAT	(AD)	11-3
11.3.	ADD NORMALIZED, LONG FORMAT	(ADR)	11-6
11.4.	ADD NORMALIZED, SHORT FORMAT	(AE)	11-9
11.5.	ADD NORMALIZED, SHORT FORMAT	(AER)	11-12

11.6.	ADD UNNORMALIZED, SHORT FORMAT	(AU)	11–15
11.7.	ADD UNNORMALIZED, SHORT FORMAT	(AUR)	11–17
11.8.	ADD UNNORMALIZED, LONG FORMAT	(AW)	11–19
11.9.	ADD UNNORMALIZED, LONG FORMAT	(AWR)	11–21
11.10.	COMPARE, LONG FORMAT	(CD)	11–23
11.11.	COMPARE, LONG FORMAT	(CDR)	11–25
11.12.	COMPARE, SHORT FORMAT	(CE)	11–27
11.13.	COMPARE, SHORT FORMAT	(CER)	11–29
11.14.	DIVIDE, LONG FORMAT	(DD)	11–31
11.15.	DIVIDE, LONG FORMAT	(DDR)	11–34
11.16.	DIVIDE, SHORT FORMAT	(DE)	11–36
11.17.	DIVIDE, SHORT FORMAT	(DER)	11–38
11.18.	HALVE, LONG FORMAT	(HDR)	11–40
11.19.	HALVE, SHORT FORMAT	(HER)	11–42
11.20.	LOAD COMPLEMENT, LONG FORMAT	(LCDR)	11–44
11.21.	LOAD COMPLEMENT, SHORT FORMAT	(LCER)	11–46
11.22.	LOAD, LONG FORMAT	(LD)	11–48
11.23.	LOAD, LONG FORMAT	(LDR)	11–50
11.24.	LOAD, SHORT FORMAT	(LE)	11–52
11.25.	LOAD, SHORT FORMAT	(LER)	11–54
11.26.	LOAD NEGATIVE, LONG FORMAT	(LNDR)	11–56
11.27.	LOAD NEGATIVE, SHORT FORMAT	(LNER)	11–58
11.28.	LOAD POSITIVE, LONG FORMAT	(LPDR)	11–60
11.29.	LOAD POSITIVE, SHORT FORMAT	(LPER)	11–62
11.30.	LOAD AND TEST, LONG FORMAT	(LTDR)	11–64
11.31.	LOAD AND TEST, SHORT FORMAT	(LTER)	11–66
11.32.	MULTIPLY, LONG FORMAT	(MD)	11–68
11.33.	MULTIPLY, LONG FORMAT	(MDR)	11–70

11.34.	MULTIPLY, SHORT FORMAT	(ME)	11-72
11.35.	MULTIPLY, SHORT FORMAT	(MER)	11-74
11.36.	SUBTRACT NORMALIZED, LONG FORMAT	(SD)	11-76
11.37.	SUBTRACT NORMALIZED, LONG FORMAT	(SDR)	11-78
11.38.	SUBTRACT NORMALIZED, SHORT FORMAT	(SE)	11-80
11.39.	SUBTRACT NORMALIZED, SHORT FORMAT	(SER)	11-82
11.40.	STORE, LONG FORMAT	(STD)	11-84
11.41.	STORE, SHORT FORMAT	(STE)	11-86
11.42.	SUBTRACT UNNORMALIZED, SHORT FORMAT	(SU)	11-88
11.43.	SUBTRACT UNNORMALIZED, SHORT FORMAT	(SUR)	11-90
11.44.	SUBTRACT UNNORMALIZED, LONG FORMAT	(SW)	11-92
11.45.	SUBTRACT UNNORMALIZED, LONG FORMAT	(SWR)	11-94

12. LOGICAL INSTRUCTIONS

12.1.	THE USE OF LOGICAL INSTRUCTIONS		12-1
12.2.	ADD LOGICAL	(AL)	12-2
12.3.	ADD LOGICAL	(ALR)	12-5
12.4.	AND	(N)	12-7
12.5.	AND	(NC)	12-10
12.6.	AND	(NI)	12-13
12.7.	AND	(NR)	12-16
12.8.	COMPARE LOGICAL	(CL)	12-19
12.9.	COMPARE LOGICAL CHARACTERS	(CLC)	12-22
12.10.	COMPARE LOGICAL CHARACTERS LONG	(CLCL)	12-25
12.11.	COMPARE LOGICAL IMMEDIATE	(CLI)	12-29
12.12.	COMPARE LOGICAL IMMEDIATE AND SKIP	(CLIS)	12-34
12.13.	COMPARE LOGICAL CHARACTERS UNDER MASK	(CLM)	12-39
12.14.	COMPARE LOGICAL	(CLR)	12-42
12.15.	COMPARE AND SWAP UNDER MASK	(CSM)	12-44
12.16.	EXCLUSIVE OR	(X)	12-49

12.17. EXCLUSIVE OR (XC) 12-52
12.18. EXCLUSIVE OR (XI) 12-55
12.19. EXCLUSIVE OR (XR) 12-58
12.20. INSERT CHARACTER (IC) 12-61
12.21. INSERT CHARACTERS UNDER MASK (ICM) 12-64
12.22. LOAD ADDRESS (LA) 12-67
12.23. MOVE IMMEDIATE (MVI) 12-69
12.24. OR (O) 12-72
12.25. OR (OC) 12-75
12.26. OR (OI) 12-78
12.27. OR (OR) 12-81
12.28. SHIFT LEFT DOUBLE LOGICAL (SLDL) 12-84
12.29. SHIFT LEFT SINGLE LOGICAL (SLL) 12-87
12.30. SHIFT LOGICAL (SHL) 12-91
12.31. SHIFT RIGHT DOUBLE LOGICAL (SRDL) 12-99
12.32. SHIFT RIGHT SINGLE LOGICAL (SRL) 12-102
12.33. STORE CHARACTER (STC) 12-105
12.34. STORE CHARACTERS UNDER MASK (STCM) 12-108
12.35. SUBTRACT LOGICAL (SL) 12-111
12.36. SUBTRACT LOGICAL (SLR) 12-114
12.37. TEST UNDER MASK (TM) 12-117
12.38. TEST UNDER MASK AND SKIP (TMS) 12-121
12.39. TRANSLATE (TR) 12-126
12.40. TRANSLATE AND TEST (TRT) 12-129

13. PRIVILEGED AND STATUS SWITCHING INSTRUCTIONS

13.1. GENERAL 13-1

13.2. STATUS-SWITCHING PRIVILEGED INSTRUCTIONS 13-1
13.2.1. Halt and Proceed Instruction (HPR) 13-2
13.2.2. Insert Storage Key Instruction (ISK) 13-3

13.2.3.	Load Program Status Word Instruction	(LPSW)	13-4
13.2.4.	Set Storage Key Instruction	(SSK)	13-5
13.2.5.	Set System Mask Instruction	(SSM)	13-6
13.3.	INPUT/OUTPUT PRIVILEGED INSTRUCTIONS		13-6
13.3.1.	Clear Channel Instruction	(CLRCH)	13-7
13.3.2.	Clear Device Instruction	(CLRDV)	13-8
13.3.3.	Enqueue I/O Instruction	(EIO)	13-9
13.3.4.	Halt Device Instruction	(HDV)	13-10
13.3.5.	Load Channel Register Instruction	(LCHR)	13-11
13.3.6.	Load Directive Address Instruction	(LDA)	13-12
13.3.7.	Load I/O Address Instruction	(LIA)	13-13
13.3.8.	Move I/O Instruction	(MIO)	13-14
13.3.8.1.	Put IORB Instruction	PRB)	13-14a
13.3.8.2.	Get IORB Instruction	(GRB)	13-14b
13.3.9.	Start Device Instruction	(SDV)	13-15
13.4.	DIAGNOSTIC PRIVILEGED INSTRUCTIONS		13-15
13.4.1.	Execute Diagnose Instruction	(EXD)	13-16
13.4.2.	RESET Instruction	(RESET)	13-17
13.4.3.	Store Status Instruction	(STS)	13-18
13.5.	INTERVAL TIMER PRIVILEGED INSTRUCTION		13-18
13.5.1.	Service Timer Register Instruction	(STR)	13-19
13.6.	CONTROL REGISTER PRIVILEGED INSTRUCTIONS		13-19
13.6.1.	Load Control Instruction	(LCTL)	13-20
13.6.2.	Store Control Instruction	(STCTL)	13-21
13.7.	RELOCATION REGISTER PRIVILEGED INSTRUCTIONS		13-21
13.7.1.	Load Relocation Register Instruction	(LRR)	13-22
13.7.2.	Store Relocation Register Instruction	(STRR)	13-23
13.8.	GENERAL REGISTER PRIVILEGED INSTRUCTIONS		13-24
13.8.1.	Supervisor Load Multiple Instruction	(SLM)	13-24
13.8.2.	Supervisor Store Multiple Instruction	(SSTM)	13-25
13.9.	DATA CHECKING PRIVILEGED INSTRUCTION		13-25
13.9.1.	Longitudinal Redundancy Check Instruction	(LRC)	13-26
13.10.	PROGRAM LOAD PRIVILEGED INSTRUCTION		13-26
13.10.1.	Initial Program Load Instruction	(IPL)	13-26a
13.11.	SWITCH LIST PRIVILEGED INSTRUCTION		13-26a
13.11.1.	Switch List Scan Instruction	(SWLS)	13-26b
13.12.	SET PROGRAM MASK STATUS-SWITCHING INSTRUCTION	(SPM)	13-27
13.13.	SUPERVISOR CALL STATUS-SWITCHING INSTRUCTION	(SVC)	13-29
13.14.	TEST AND SET STATUS-SWITCHING INSTRUCTION	(TS)	13-31

14. DELETED (LIST PRØCESSING)

PART 4. BAL DIRECTIVES

15. INTRODUCTION TO DIRECTIVES

16. EQUATE AND DELETE OPERATION CODE DIRECTIVES

16.1. EQUATE (EQU) 16–1

16.2. DELETE OPERATION CODE (OPSYM) 16–3

17. ASSEMBLER CONTROL DIRECTIVES

17.1. CONDITION NO OPERATION (CNOP) 17–2

17.2. PROGRAM END (END) 17–4

17.3. GENERATE LITERALS (LTORG) 17–5

17.4. SPECIFY LOCATION COUNTER (ORG) 17–6

17.5. PROGRAM START (START) 17–8

18. BASE REGISTER ASSIGNMENT DIRECTIVES

18.1. UNASSIGN BASE REGISTER (DROP) 18–2

18.2. ASSIGN BASE REGISTER (USING) 18–3

19. PROGRAM LINKING AND SECTIONING DIRECTIVES

19.1. COMMON STORAGE DEFINITION (COM) 19–3

19.2. CONTROL SECTION IDENTIFICATION (CSECT) 19–6

19.3. DUMMY CONTROL SECTION IDENTIFICATION (DSECT) 19–8

19.4. EXTERNALLY REFERENCED SYMBOL DECLARATION (ENTRY) 19–10

19.5. EXTERNALLY DEFINED SYMBOL DECLARATION (EXTRN) 19–11

19.6. SUBROUTINE LINKAGE 19–12

20. LISTING CONTROL DIRECTIVES

20.1. ADVANCE LISTING (EJECT) 20–2

20.2. LISTING CONTENT CONTROL (PRINT) 20–3

20.3. LEAVE BLANK LINES ON LISTING (SPACE) 20-5

20.4. LISTING TITLE DECLARATION (TITLE) 20-6

21. INPUT AND OUTPUT CONTROL DIRECTIVES

21.1. INPUT FORMAT CONTROL (ICTL) 21-2

21.2. INPUT SEQUENCE CONTROL (ISEQ) 21-4

21.3. REPRODUCE FOLLOWING RECORD (REPRO) 21-5

21.4. PRODUCE A RECORD (PUNCH) 21-6

21.5. INCLUDE CODE FROM A LIBRARY (COPY) 21-8

21.6. CHANNEL COMMAND WORD (CCW) 21-9

PART 5. BAL MACROS

22. MACRO FACILITY

22.1. THE MACRO PROCESSOR 22-1

22.2. MACRO SOURCE CODE 22-2

23. MACRO DESIGN

23.1. THE MACRO DEFINITION 23-1

23.2. MACRO DEFINITION STORAGE 23-3

23.3. THE MACRO CALL INSTRUCTION 23-6

24. TWO TYPES OF MACRO DEFINITIONS

24.1. PROCS AND MACROS 24-1

24.2. CALL INSTRUCTION DESIGN 24-2

24.3. PASSING PARAMETERS TO THE BODY 24-7

25. PROC FORMAT

25.1. BASIC PROC DESIGN 25-1

25.2. REFERENCING POSITIONAL PARAMETERS IN THE CALL 25-3

25.3.	REFERENCING KEYWORD PARAMETERS IN THE CALL		25–4
25.4.	REFERENCING SUBPARAMETERS IN THE CALL		25–7
25.5.	MULTIPLE PROC NAMES AND POSITIONAL PARAMETER 0		25–9
25.6.	THE LABEL ARGUMENT		25–11

26. MACRO FORMAT

26.1.	BASIC MACRO DESIGN		26–1
26.2.	REFERENCING POSITIONAL PARAMETERS IN THE CALL		26–2
26.3.	REFERENCING KEYWORD PARAMETERS IN THE CALL		26–4
26.4.	REFERENCING SUBPARAMETERS IN THE CALL		26–5
26.5.	THE LABEL ARGUMENT		26–6

27. CONDITIONAL ASSEMBLY

27.1.	SET SYMBOLS		27–2
27.1.1.	Local Set Symbols		27–3
27.1.2.	Global Set Symbols		27–5
27.1.3.	Set Symbol Value Assignment		27–6
27.1.4.	SET Statement		27–7
27.1.5.	SETA Statement		27–9
27.1.6.	SETB Statement		27–10
27.1.7.	SETC Statement		27–13
27.1.8.	Character Expressions		27–14
27.1.9.	Subscripted SET Symbols		27–14
27.1.9.1.	Defining Subscripted SET Symbols		27–15
27.2.	BRANCHING		27–15
27.2.1.	Sequence Symbols		27–15
27.2.2.	Unconditional Branch	(AGO)	27–16
27.2.3.	Conditional Branch	(AIF)	27–17
27.2.4.	Define Branch Destination	(ANOP)	27–18
27.2.5.	Macro Definition Exit	(MEXIT)	27–19
27.3.	ERROR MESSAGES AND COMMENTS		27–19
27.3.1.	MNOTE Message Statements	(MNOTE)	27–20
27.3.2.	PNOTE Message Statements	(PNOTE)	27–21
27.3.3.	Comments Statement		27–21
27.4.	REPETITIVE CODE GENERATION		27–22
27.4.1.	Define Start of Range	(DO)	27–22
27.4.2.	Define End of Range	(ENDO)	27–23
27.4.3.	Conditional Assembly Control Counter	(ACTR)	27–24

27.5. ATTRIBUTE REFERENCES 27–25
27.5.1. Type Attributes 27–27
27.5.2. Length Attributes 27–28
27.5.3. Scale Attributes 27–30
27.5.4. Integer Attributes 27–30
27.5.5. Count Attributes 27–31
27.5.6. Number Attributes 27–32

PART 6. ASSEMBLY LISTING

28. ORGANIZATION OF LISTING

28.1. HEADER LINES 28–1

28.2. PREFACE 28–1

28.3. CODEDIT 28–2

28.4. EXTERNAL SYMBOL DICTIONARY LISTING 28–3

28.5. CROSS-REFERENCE LISTING 28–4

28.6. DIAGNOSTIC LISTING 28–5

28.7. EXAMPLE OF ASSEMBLY LISTING 28–5

PART 7. PROGRAMMING TECHNIQUES

29. JOB CONTROL PROCEDURES

29.1. HOW TO RUN A JOB 29–1

29.2. INTRODUCING THE SOURCE DECK 29–1
29.2.1. JOB Control Statement 29–2
29.2.2. OPTION Job Control Statement 29–2

29.3. ASSEMBLE; ASSEMBLE AND LINK-EDIT; OR ASSEMBLE, LINK-EDIT, AND EXECUTE 29–3
29.3.1. Assemble (ASM) 29–3
29.3.1.1. ASM Jproc Call Statement 29–4
29.3.2. Assemble and Link-Edit (ASML) 29–10
29.3.2.1. ASML Jproc Call Statement 29–10
29.3.3. Assemble, Link-Edit, and Execute (ASMLG) 29–11
29.3.3.1. ASMLG Jproc Call Statement 29–11

29.4. START-OF-DATA JOB CONTROL STATEMENT (/$) 29–12

29.5. FOLLOWING THE SOURCE DECK 29–13
29.5.1. End-of-Data Job Control Statement (/*) 29–13
29.5.2. End-of-Job Control Statement (/&) 29–13
29.5.3. Terminate-the-Card-Reader Job Control Statement (// FIN) 29–13
29.5.4. Setting the UPSI Byte 29–14

29.6. SUMMARY OF JOB CONTROL PROCEDURE 29–14a
29.6.1. Assembly 29–14a
29.6.2. Assembly and Link-Edit 29–16
29.6.3. Assembly, Link-Edit, and Execution 29–18

29.7. RUNNING ASSEMBLER FROM A WORKSTATION 29–18

30. EXAMPLE MACRO DEFINITIONS

30.1. SMALR/LARGR PROC (POSITIONAL PARAMETER 0) 30–1

30.2. SMALL6/LARGE6 PROC (DO LOOP) 30–2

30.3. BLANK MACRO (VARIABLE INLINE EXPANSION CODE) 30–4

PART 8. APPENDIXES

A. SAMPLE PROGRAM

B. CHARACTER CONVERSION CODES

C. MATH TABLES

C.1. HEXADECIMAL-DECIMAL INTEGER CONVERSION C–1

C.2. HEXADECIMAL FRACTIONS (APPROXIMATE VALUES) C–7

C.3. POWERS OF 2 C–8

C.4. POWERS OF 16 C–9

D. CHECK-OFF TABLE TERMS

E. INSTRUCTION LISTINGS

F. USE OF PARAM STATEMENT

F.1. PARAM STATEMENT F–1

F.2. SOURCE CORRECTIONS F–5
F.2.1. SEQ Statement F–6
F.2.2. REC Statement F–7
F.2.3. SKI Statement F–7

G. SYSTEM VARIABLE SYMBOLS

G.1. &SYSECT G–1

G.2. &SYSLIST G-1

G.3. &SYSNDX G-2

G.4. &SYSDATE G-2

G.5. &SYSTIME G-3

G.6. &SYSJDATE G-4

G.7. &SYSPARM G-5

USER COMMENT SHEET

INDEX

FIGURES

1-1. Writing and Submitting a Program 1-2
1-2. Card Image 1-3
1-3. Assembler Coding Form 1-4
1-4. Coding Form and Card Image Relationship 1-5
1-5. Example of Proper Coding Techniques 1-13
1-6. COBOL Source Code 1-15
1-7. Object Code Generated from COBOL Source Code 1-15
1-8. Assembly Listing 1-16
1-9. OS/3 Object Module Format 1-18
1-10. OS/3 Load Module Format 1-19
1-11. Assemble, Link, and Go Operation 1-20

2-1. Determining Binary Values 2-3
2-2. Fixed-Point Number Formats 2-9

4-1. Assembler Format Relationships 4-4
4-2. Byte and Word Structure 4-7

5-1. Floating-Point Number Formats 5-17

7-1. Instruction Formats 7-2

8-1. Program Status Word Diagram 8-1

→ 9-1 thru 9-8 Deleted (MSS)

10-1. Comparison of Binary Numbers and Values Expressed in Powers of 2 10-6

14-1 thru 14-37 Deleted (LIST PROCESSING)

22-1. Example of Inline Macro Expansion 22-3

23-1. Accessing a Macro Definition Submitted in the Source Deck 23-4
23-2. Accessing a Macro Definition Stored in a Library 23-5

24-1. PROC and MACRO Heading 24-1
24-2. PROC, MACRO, and Call Instruction Comparison 24-6
24-3. Communication between Macroinstruction and Macro Definition 24-8
24-4. Example of MACRO and PROC Definitions 24-12

TABLES

2-1. Comparison of Numeric Expressions 2-2
2-2. Hexadecimal Notation 2-4

4-1. Comparison of Terms 4-9
4-2. Summary of Operators 4-14

5-1. Characteristics of Constant and Storage Definition Types 5-2
5-2. Zero Duplication Area Examples 5-6

8-1. Extended Mnemonics and Functions 8-3
8-2. Operand 1 Mask Combinations 8-10
8-3. Branch-on-Condition Instruction by Usage 8-11

9-1 thru 9-8 Deleted (MSS)

12-1. Shift Logical Mask Bits 12-92

14-1 thru 14-9 Deleted (LIST PROCESSING)

15-1. Assembler Directives 15-1

17-1. Assembler Control Directives 17-1

20-1. Listing Control Directives 20-1

27-1. Conditional Assembly Language Statements 27-1
27-2. Operator Priority 27-9
27-3. Valid Attribute Reference Applications 27-26
27-4. Type Attributes of Symbols 27-27

28-1. CODEDIT Listing Content 28-2
28-2. External Symbol Dictionary (ESD) Listing Content 28-3
28-3. Cross-Reference Content 28-4
28-4. Diagnostic Listing Content 28-5

B–1.	ASCII (American Standard Code for Information Interchange) Character Codes	B–1
B–2.	EBCDIC (Extended Binary Coded Decimal Interchange Code) Character Codes	B–2
B–3.	Punched Card, ASCII, and EBCDIC Codes	B–3
C–1.	Hexadecimal-Decimal Integer Conversion	C–3
C–2.	Hexadecimal Fractions	C–7
E–1.	Mnemonic List of Instructions	E–1
E–2.	Alphabetic Listing of Instructions	E–5
E–3.	List of Instructions by Machine Code	E–11

PART 1. BASIC DATA AND CONVENTIONS

1. Introduction to Assembler Language Programming

1.1. CODING AN ASSEMBLER PROGRAM

An assembler language program goes through several translations from the time it is hand coded by a programmer until it is actually inside the computer and operating. (See Figure 1—1.) The first change is the conversion of code decipherable by people, source code written in basic assembler language (BAL), to data capable of being processed by a computer. Although an assembler source program can reside on several types of storage media, diskette and punched card are the two types used by the BAL programmer operating in a System 80 environment. The source code recorded on either of these media types is in card image format. Therefore, the guidelines for generating your BAL program on diskette are the same as those used for preparing punched cards. Because of this similarity, the descriptions provided in this section are discussed from the standpoint of card images.

While source code is entered onto cards by a card punch, it is recorded onto diskette by either of two methods, both of which involve keyboard entry. One method allows you to prepare the diskette offline by use of the SPERRY UNIVAC Universal Distributed System 2000 (UDS 2000). Basically, you perform the same functions on the UDS 2000 that you would from a card punch except the card images generated from the keyins are recorded on diskette. The second method for recording card images onto diskette is through the use of the system console (or workstation keyboard) and the general editor, information for which is presented in the general editor user guide/programmer reference. In addition to diskette, the general editor can output source code to disk.

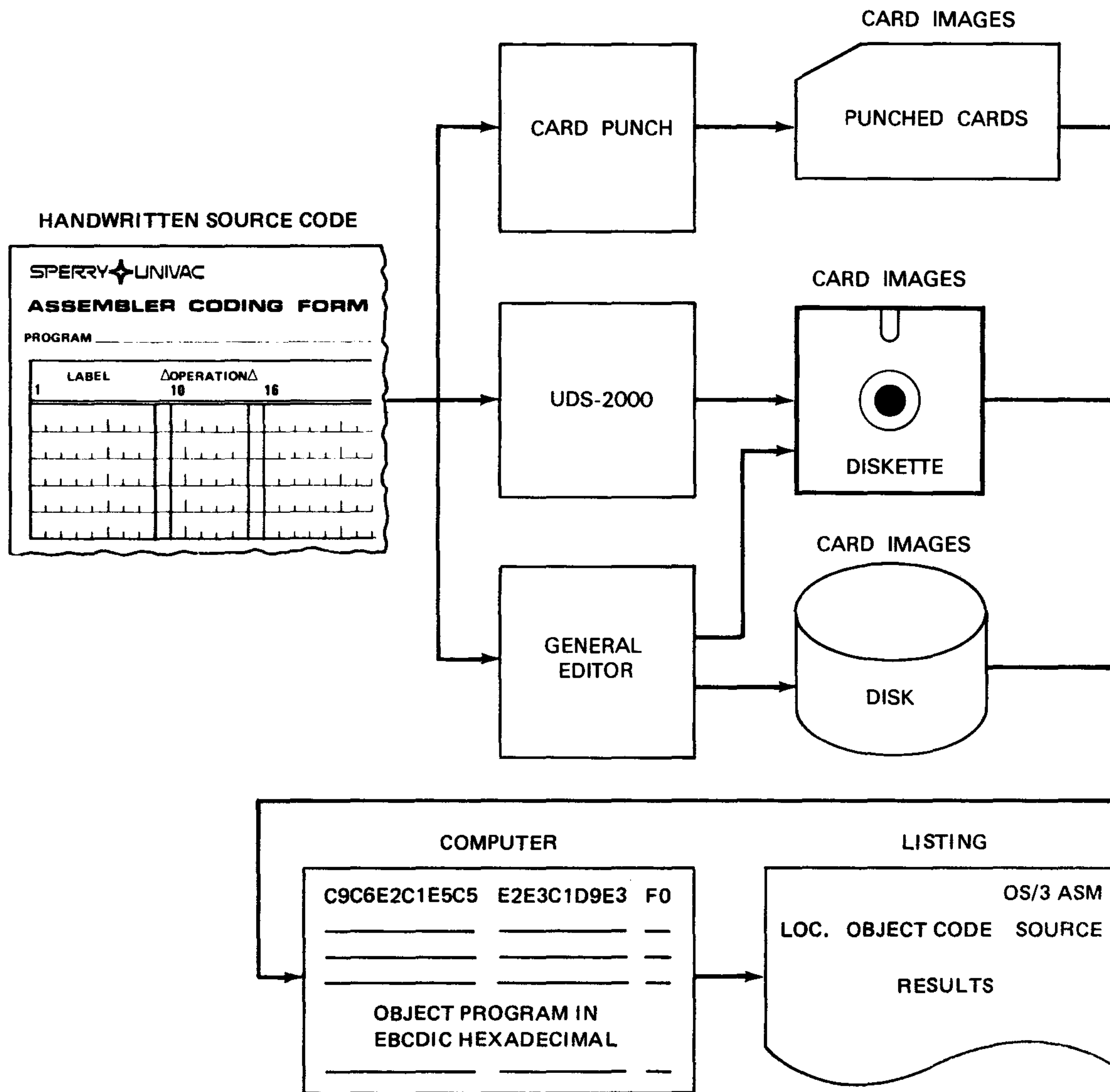


Figure 1—1. Writing and Submitting a Program

The coded entries in a card image are converted on a column-by-column basis to data that can be processed by the computer. Each column represents a single unit of information. Figure 1—2 shows a blank card image. From this figure, you can see that it has 80 vertical columns. Each column has 12 vertical positions called rows (rows are numbered 0 through 9, 11, and 12 which appear at the top portion of the format preceding row 0).

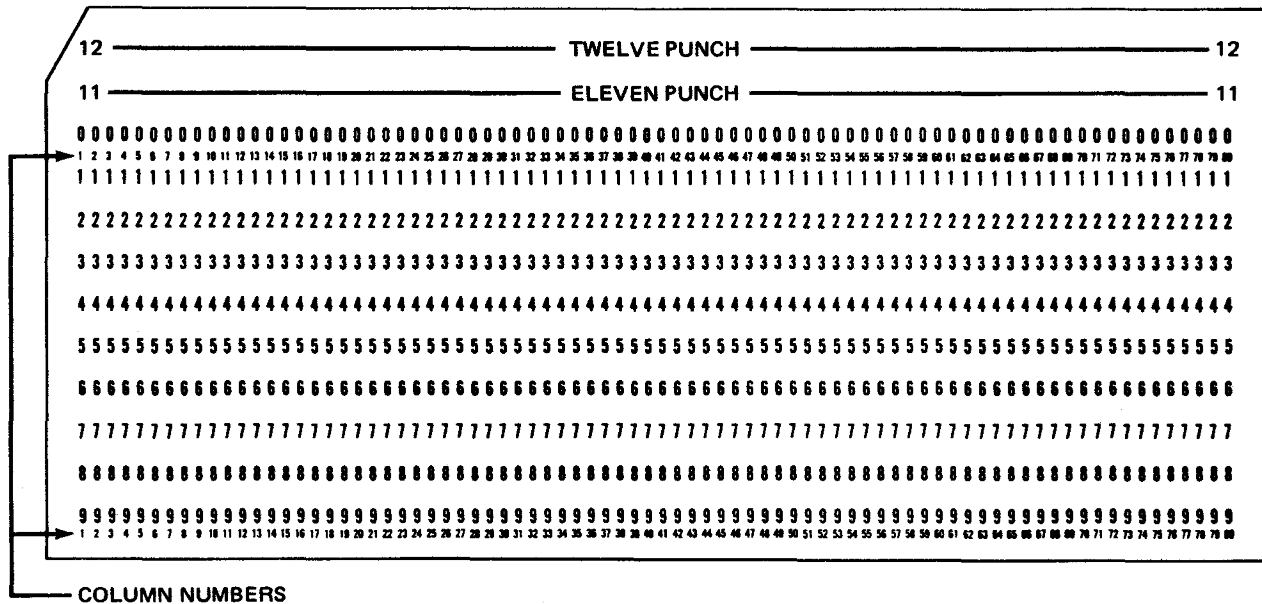


Figure 1—2. Card Image

Different entry configurations in a column represent different characters and numbers. For instance, each decimal number (0 through 9) is represented by a respective position in the card image. If an entry is made only in row 0 of a column, then the image for that column is interpreted as the value zero. Likewise, if an entry is made only in row 1 of a column, then the image for that column is interpreted as the value 1, and so on through row 9. Everything you code in assembly language is based on the 80-column card image.

The card images that make up an assembler language program are entered from code that is handwritten on an assembler coding form. (The Sperry Univac assembler coding form is shown in Figure 1—3.) Each line on the assembler coding form has 80 positions that correspond to the 80 columns of the card image. One card image is entered for every line of code on the assembler coding form (Figure 1—4). The lines of code on the assembler coding form and the card images entered from the form are called source code lines. Collectively, these source lines make a source program.

A BAL source program is written with instructions, directives, conditional statements, and macros. They are the elements of the assembler language and each is usually written on one source line. (Sometimes it may take more than one source line to write a single element, but most of the time it takes only one source line for one instruction, directive, statement, or macro.) The assembler ignores the presence of any blank card images in the source code. A blank line will not be printed nor terminate a sequence of continuation lines. The rules for coding assembler language source lines are reflected on the assembler coding form. Each source line has five fields and the assembler expects specific information to be coded in each field.

SPERRY UNIVAC

ASSEMBLER CODING FORM

UNIVAC 9000 SERIES

PROGRAM ____ PROGRAMMER ____ DATE ____ PAGE ____ OF ____ PAGES

LABEL	△OPERATION△	OPERAND	△	COMMENTS	
1	10	16		72	80

Figure 1—3. Assembler Coding Form

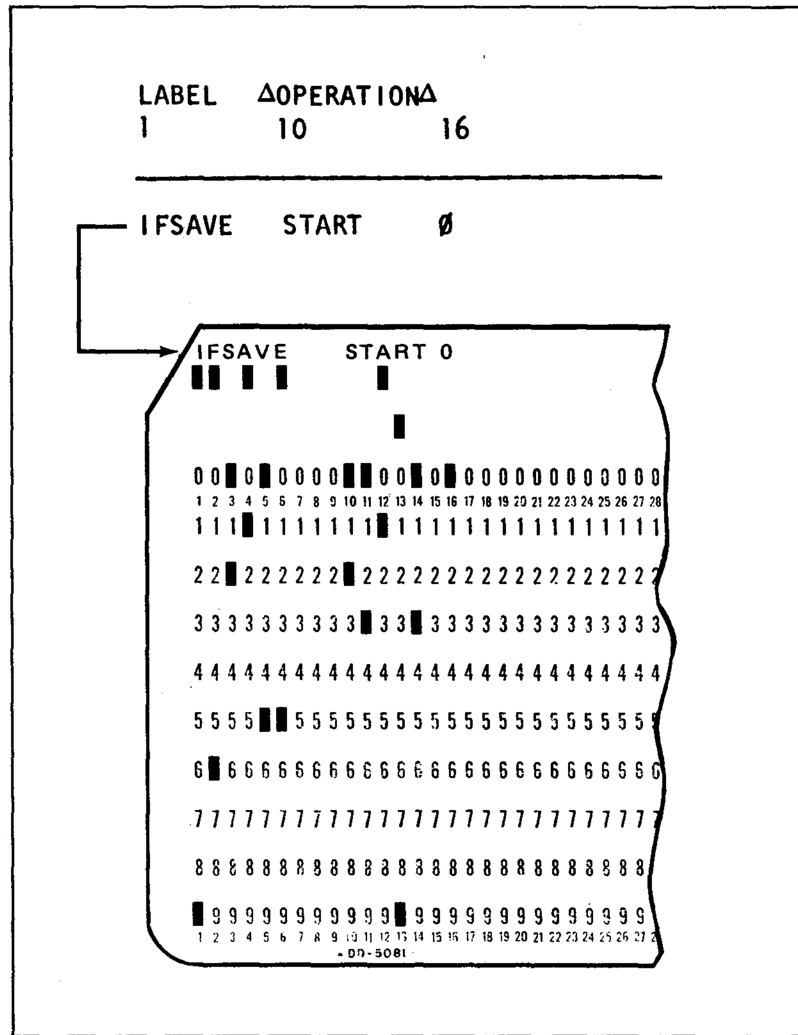


Figure 1—4. Coding Form and Card Image Relationship

1.1.1. Operation Field

The easiest part of an assembler source code line to recognize is the operation field; it begins in column 10 and ends in column 14 of the card image. The operation field is the most restrictive field on the coding form because you must use an established operation code. You cannot arbitrarily assign a name of your own. The operation code you use is a mnemonic code that relates to some function. For example, *A* is for add, *D* is for divide, and *S* is for subtract. The mnemonic code must be written exactly as the instruction, directive, or statement indicates. For example, *A* (not *AD*) causes the add operation to be performed. If you put *AD* in the operation field, the assembler could not relate it to any of the assembler functions, so this would cause an error. Each mnemonic code for instructions, directives, statements, or macros is listed with the description of that function. The rules for using the operation field are:

1. The operation code must not contain embedded blanks.

2. The operation code must be written exactly as shown in the list of mnemonics for instructions, directives, and procs, or macroinstructions.

3. The operation field must be terminated by a blank.

4. The operation code must not start in column 1.

Examples:

```
    LABEL    ΔOPERATIONΔ                    OPERAND
    1        10       16
   ---------------------------------------------------
1. MOVEPAY  MVC       YEARPAY,WORK
2. MOVEPAY  M V C     YEAR,WORK
3.          EOJ
4.          ENJOB
5.          START     Ø
6.          STARTØ
7.          USINC     *,6
8. USING *,6
```

1. Valid
2. Invalid because there are embedded blanks in the operation code MVC
3. Valid
4. Invalid because there is no such mnemonic as ENJOB
5. Valid
6. Invalid because the operation code START is not followed by a blank
7. Valid
8. Invalid because the operation code starts in column 1

1.1.2. Operand Field

The operand field is the object of the operation code. The operand field begins in column 16 and ends in column 71. The operand field holds the data or the location of data that is being operated on. Each item of data in the operand field is an operand, and operands are separated by commas. For instructions, operands can be actual data — like the decimal number 10, the name of an area where data is stored — like STORAREA, or the actual address specifying the number of bytes the assembler must count to get to the data — like 1108(32). Operands for instructions, directives, statements, or macros are whatever parameters are required by the particular operation that is being done. For instance, an *add immediate* instruction has two operands. The first operand is a main storage location, and the second operand is a byte of actual data. An *add immediate* instruction adds the second operand to whatever data is located at the first operand's address.

```
          AI        STORAREA,1Ø
```

The add operation in this example is performed on the actual data, 10, and on whatever data is located at an area named STORAREA.

The rules for using the operand field are:

1. The operand field is terminated by a blank that is not enclosed by an apostrophe.

2. Operands may be continued onto the next line by placing a nonblank character in column 72. Up to two continuation lines are permitted. Caution should be exercised when using a nonblank character in column 72. As shown in the OUTRIB RIB example (1.1.7), a comma must follow the last operand on the continued statement if there are more operands to follow; otherwise, the operands that follow will be treated as comments.

3. Column 16 is where a continuation line starts.

Examples:

```
   LABEL   ΔOPERATIONΔ                      OPERAND
   1        10        16                                                  72

1. NAME     DC        CL9'REBEW R D'  NAME IN 9 BYTES
2. NAME     DC        CL9'REBEW R D'NAME IN 9 BYTES
3.          ENTRY     ILE,AYAHC NAD,NAHS,WNS,WBE,OREG,                     X
                      DNOMYAR,N4543NI1,CONST32,EQUITY,WMC,                 X
                      WDR,WRD32,SGAW
```

1. Valid
2. Invalid because the operand field is not terminated by a blank
3. Invalid because the line has an embedded blank

1.1.3. Label Field

As we mentioned, the operand field can contain data or the name of an area where data is stored. You assign a name to an area in your program by coding a symbolic name in the label field of the area to be accessed. Once a source line is given a label, it can be referenced from any other location in the source program. For example, I can name a line of code and use its name in the operand field of an instruction.

```
1. ROUTINE  AI        STORAREA,1Ø
            .
            .
            .
2.          B         ROUTINE
```

In this example, I labeled an add instruction: ROUTINE. Then, later in my program, I used the symbol ROUTINE to refer to that line of code. On line 2, I said, "Branch to the area called ROUTINE, where the add instruction is located."

A symbol in the label field of a line of code can also be used as an operand to reference data. For example, I can write a line of code to define a constant.

```
LABEL   ΔOPERATIONΔ                        OPERAND
1        10         16
--------------------------------------------------------
TEN      DC         H'1Ø'
```

This line of code says, "Place the number 10 in the location named TEN." Once the symbol TEN is defined, it can be used as an operand to represent the value 10.

```
         AI         TEN,6
```

In this line of coding, I'm requesting that 6 be added to whatever data is stored at location TEN. When you label data as I labeled the data (10), you are associating a symbol with a value. That symbol can then be used in place of the value.

The rules for using the label field are:

1. The symbol must start in column 1.
2. The symbol must begin with an alphabetic character or special letter.
3. The symbol must not exceed eight characters in length.
4. The symbol must not contain embedded blanks or other special characters.
5. The field must be terminated by a blank.

Examples:

```
    LABEL   ΔOPERATIONΔ                        OPERAND
    1        10         20
   ------------------------------------------------------
1. |BEGIN
2. | BEGIN
3. |WEEKS52
4. |52WEEKS
5. |EMPLOYEE
6. |EMPLOYEENO
7. |BLANKNO
8. |BLANK NO
9. |MOVEPAYMVCYEARPAY,WORK
```

1. Valid
2. Invalid because the symbol does not start in column 1
3. Valid
4. Invalid because the symbol starts with a number
5. Valid
6. Invalid because the symbol is longer than eight characters
7. Valid
8. Invalid because the symbol contains an embedded blank
9. Invalid because the symbol MOVEPAY is not followed by a blank (There must also be a blank after the operation code MVC.)

The three fields just discussed are essential for designing an executable BAL program. The remaining two fields, the comment and sequence fields, don't play a role in the actual design of a program but they are useful programming aids. The comment field is a program documentation aid and the sequence field is a program maintenance aid. Program documentation is as important to the programmer writing the program as it is to those who must refer to it later. Operand specification is usually completed by column 40, thus leaving columns 41 through 71 free for comments.

1.1.4. Comments Field

There are two ways to code comments:

1. Comments can be coded on the same line as an instruction, statement, or directive. There must be at least one blank between the end of the operand specification and the start of comments. If your comments exceed one source line, place a nonblank character in column 72 and continue the remaining comments on the next source line (at column 16).

   Examples:

```
LABEL    ΔOPERATIONΔ           OPERAND      ΔCOMMENTS
1         10         16                                                        72
----------------------------------------------------------------------------------
         OPEN       CARDFLE,(PRINT)          OPEN FILES
         BALR       14,HDRTN                 GO TO HEADING ROUTINE
READCARD DMINP      CARDFLE,CARWORK          READ A CARD INTO WORKAREA
         MVI        PRINTOUT,C' '            CLEAR PRINT AREA
         MVC        PRINTOUT+1(131),PRINTOUT
         .
         .
         .

         CLC        NUMBERIN(5),CUSTNO       IS THE CUSTOMER NUMBER DIFFERENX
                    T THAN THE PREVIOUS NUMBER
         BNE        NEWCUST
         .
         .
         .
```

2. Comments can be coded on a separate line. This is done by placing an asterisk (*) in column 1 of a source line. Then your comments can be coded. If your comments exceed one full source line, place another asterisk in column 1 of the next source line and continue coding the remaining comments. Note that a nonblank character is not coded in column 72 for continuation when an asterisk is coded in column 1 of the next source line. However, if your comments exceed one full source line, you can code a nonblank character in column 72 if you continue the remaining comments on the next source line starting in column 16. An asterisk must not be coded in column 1.

   Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10      16
------------------------------------------------------------------------
         BALR     4,Ø
         USING    *,4
         OPEN     CARDSIN,(CARDRIB)
* THIS PROGRAM PREPARES AN ACCOUNTS RECEIVABLE REPORT USING CARD INPUT
* AND PRINTER OUTPUT
```

During assembly, comments are printed but do not affect the resulting object code. The purpose of comments is to make the program listing easier to follow and can also highlight certain portions of the program.

1.1.5. Sequence Numbers

Columns 73 through 80 may be used for entering sequence numbers. This is done by assigning consecutive numbers to each line of coding and is useful for reassembling the card deck, if it should be dropped. It is good practice to number the lines in multiples of 10, or even 100. This allows you to insert additional coding lines without having to renumber the cards when they have been keypunched prior to the modification. Some programmers use letters in addition to the numbers. This is useful in identifying the deck from which cards have come if they have been removed for any reason. Sequence numbers also are important in maintaining a source module. A copy of your source module may be stored on tape, disk, or diskette and the OS/3 librarian can update and correct the source module by using the sequence numbers. (See the system service programs (SSP) user guide.)

1.1.6. Column 72

Another coding feature on the assembler coding form is column 72. This column separates the sequence field from the rest of the source line and is normally blank unless you have to continue an operand field to the next source line. If an operand specification is too lengthy to fit into the columns provided on a single line, the field may be continued onto the next line. An operand field can be continued by coding any nonblank character in column 72 and then continuing the operands on the next line starting in column 16. It is best to avoid using the comma as a continuation character when the comma is being used to separate the operand fields. However, it can be used as a continuation character when it is being used to separate operands. If you have coded up to column 72 and the next character you have to code is a comma separating operands, that comma must appear in column 16 of the next line after you code a nonblank character in column 72 (even another comma may be used).

1.1.7. Additional Coding Rules

The operand fields of an instruction, directive, or conditional statement must completely fill all available space on a source line, starting with the first operand specified up to and including column 71. Then a nonblank character can be placed in column 72 and the remainder of the operand field can be continued onto the next source line (column 16). These operand fields in an instruction or directive can be continued for only two additional lines.

Example:

```
LABEL   ΔOPERATIONΔ                         OPERAND
1        10       16                                                      72
--------------------------------------------------------------------------------
TITLE    DC       C'UNITED STATES GOVERNMENT PRINTING OFFICE STYLE MANUALSX
                  (ABRIDGED)'
         ENTRY    I1234567,J1234567,K1234567,L1234567,M1234567,N1234567,O1X
                  234567
```

The operand fields of macros and procs can be coded in two different ways:

1. The operand fields can be coded in the same manner as instructions, directives, or conditional statements, in which case they must completely fill all available space on a source line, starting with the first operand specified up to and including column 71. Then a nonblank character can be placed in column 72 and the remainder of the operand field can be continued onto the next source line (column 16).

2. The operand fields can be coded to leave space between the last operand specified on that line and the nonblank character in column 72. A comma must be placed immediately following the last operand on that line, thereby separating it from the following operand field on the next source line. However, if you omit the comma immediately following the last operand on that line, and at least one blank exists between the last operand and the nonblank character in column 72, a warning message is issued by the assembler.

The operand fields in a macro or proc can be continued for as many lines as necessary.

Examples:

```
LABEL    ΔOPERATIONΔ              OPERAND
1        10          16                                                 72
_____________________________________________________________________________

OUTRIB   RIB         IOA1=OUTBUF,RCFM=VARBLK,VARBLD=(13),IORG=(12),TYPEFLE=OUX
                     TPUT,FILABL=STD

OUTRIB   RIB         IOA1=OUTBUF,                                       X
                     RCFM=VARBLK,                                       X
                     VARBLD=(13),                                       X
                     IORG=(12),                                         X
                     TYPEFLE=OUTPUT,                                    X
                     FILABL=STD
```

It is wise to develop good coding habits from the start. A neatly coded program is easy to enter, debug, and interpret. Figure 1—5 is an example of such a program. This example program follows the format of the coding form, has plenty of comments, and uses sequence numbers. Don't fall into the bad habit of jotting down instructions and assembling them just to see if your ideas have any substance. It is much better to sit down and evaluate the problem. First flowchart your program, and then code it on the coding form, using plenty of comments and sequence numbers for lengthy programs.

```
LABEL    ΔOPERATIONΔ           OPERAND         ΔCOMMENTS
1        10        16                                                            72      80

         TITLE     'FIRST PROBLEM PROGRAM'                                      DARØØ1ØØ
PROG1    START     Ø                                                            DARØØ2ØØ
BEGIN    BALR      6,Ø                                                          DARØØ3ØØ
         USING     *,6                                                          DARØØ4ØØ
         ZAP       WORKAREA,BONUS          ENTERS BONUS RATE INTO WORKAREA      DARØØ5ØØ
         MP        WORKAREA,WEEKS          MULT BONUS BY 52 WEEKS               DARØØ6ØØ
         AP        WORKAREA,YEARRATE       ADD BONUS TO YEARLY RATE             DARØØ7ØØ
MOVEPAY  MVC       YEARPAY,WORKAREA+2      MOVE TOTAL YEARLY PAY                DARØØ8ØØ
         DP        WORKAREA,WEEKS          DIVIDE TOTAL PAY BY 52 WEEKS         DARØØ9ØØ
         MVC       WEEKPAY,WORKAREA+1      MOVE WEEKLY PAY,HOURLY RATE IS       DARØ1ØØØ
*                                          NOT CALCULATED IN THIS PROGRAM       DARØ11ØØ
         MVC       OUTPUT(29),EMPLOYEE     COMPLETE RECORD MOVED                DARØ12ØØ
         EOJ                               END OF JOB                           DARØ13ØØ
WORKAREA DS        CL6                     RESERVE 6 BYTES OF STORAGE           DARØ14ØØ
BONUS    DC        PL2'5ØØ'                PACKED VALUE OF 5ØØ IN 2 BYTES       DARØ15ØØ
WEEKS    DC        PL2'52'                 PACKED VALUE OF 52 IN 2 BYTES        DARØ16ØØ
YEARRATE DC        PL4'13ØØØØØ'            PACKED 13ØØØØØ IN 4 BYTES            DARØ17ØØ
OUTPUT   DC        23C'Δ'                  23 BYTES OF BLANKS                   DARØ18ØØ
EMPLOYEE DS        ØCL23                   SYMBOL FOR NEXT 23 BYTES             DARØ19ØØ
NAME     DC        CL9'REBEWΔRΔD'          REBEWΔRΔD IN 9 BYTES                 DARØ2ØØØ
WORKNO   DC        C'A1234'                A1234 IN 5 BYTES                     DARØ21ØØ
YEARPAY  DC        PL4'Ø'                  4 BYTES OF PACK ZEROS                DARØ22ØØ
WEEKPAY  DC        PL3'Ø'                  3 BYTES OF PACKED ZEROS              DARØ23ØØ
CG       DC        C'ΔΔΔ'                  3 BYTES OF BLANKS                    DARØ24ØØ
         END       BEGIN                   END OF THE PROGRAM                   DARØ25ØØ
```

Figure 1—5. Example of Proper Coding Techniques

You can, if you wish, code your assembler program in a free-form manner. The operation, operand, and comments fields don't always have to start in column 10, column 16, and column 41. These columns are shown on the coding form as preferred starting positions for each field to promote formalized coding practices. One unbreakable rule is that label field must always start in column 1. Each field after the label field must be separated by at least one blank. So, if you only have a 3-character label, the operation field can be coded starting in column 5 instead of column 10. Also note that the label, operation, and operand fields must all be keypunched on the same card. Another restriction is that the sequence numbers must always appear in columns 73 through 80. Some examples of free-form coding are as follows:

```
LABEL    ΔOPERATIONΔ              OPERAND
1        10        16                                                            72      80

TAG START Ø                                                                     DARØØ1ØØ
BEGIN BALR 6,Ø                                                                  DARØØ2ØØ
 USING *,6                                                                      DARØØ3ØØ
 ZAP WORKAREA,BONUS ENTERS BONUS RATE INTO WORKAREA                             DARØØ4ØØ
 MP WORKAREA,WEEKS, MULT BONUS BY 52 WEEKS                                      DARØØ5ØØ
```

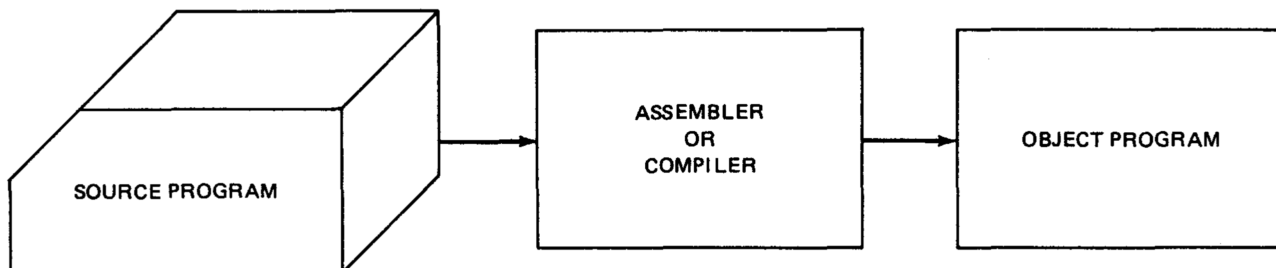
As you can see, the free-form style of coding is much more difficult to interpret than the formalized style.

Another option is available, if the location of the fields on the supplied coding form doesn't suit your particular application. The assembler coding form can be changed by using the ICTL directive (21.1). By using this directive, you can change the location of the beginning, ending, or continuation column.

After a BAL source program is coded, it must first be *assembled* (and also linked) before the program can be *executed* by the computer. These two functions are separate operations and therefore they happen at different times under control of two different computer elements. At assembly time, the assembler translates the source program to machine code instructions, and at execution time, the hardware processor performs the machine code instructions. Although *you* can interpret a source program as if it can actually execute, the hardware processor is incapable of actually executing this source program.

1.2. ASSEMBLING A PROGRAM

Before source code can be executed by a computer, it must be converted to machine code. A BAL source program is converted by an assembler, and a higher-level language, like COBOL or FORTRAN, is converted by a compiler. Whether a source program is assembled or compiled, the output is always the same. An assembler or compiler produces an object program (machine code):



The object program is a binary program that can actuate the electronic logic circuits in the hardware processor to perform specific functions like add, subtract, or divide. Any computer program must be in binary form before it can be stored in the computer and executed by the processor.

Though an assembler or compiler can produce an object program, each operates differently. Figure 1—6 shows five source lines from a COBOL program, and Figure 1—7 shows the object source code generated from the original COBOL source lines. As you can see from Figure 1—7, a single compiler source line produces several object code instructions. This is not true of assembler source lines (excluding macroinstructions); each line is converted to object code on a one-for-one basis. The object code shown in Figure 1—7 is in hexadecimal as is any object code shown in printout form. This is because hexadecimal is easier to read and binary would take up too much room on a printout.

```
LINE NO.  SEQ.    SOURCE STATEMENT                     IDEN.
00088     004019  PROCEDURE DIVISION.                  PROG01
00089     004020  INITIALIZE.                          PROG01
00090     004021      OPEN INPUT CDS.                  PROG01
00091     004022      OPEN OUTPUT NEWFIL, LIST.        PROG01
00092     004023      MOVE SPACES TO OUT.              PROG01
```

Figure 1—6. COBOL Source Code

```
LINE #  BASE/DISPL  ADDRESS  CONTENTS OF MEMORY      OPERAND ADDRESSES  OPCODE  COMMENTS
00089               000938   INITIALIZE                                         PARAGRAPH HEADER
00090               000938   58 10 A 0AC             0000E4             L       OPEN
                    00093C   58 F0 A 0F0             000108             L
                    000940   05 EF                                      BALR
00091               000942   41 10 A 0A4             0000DC             LA      OPEN
                    000946   58 F0 A 0D4             0000FC             L
                    00094A   05 EF                                      BALR
                    00094C   0042                                       DC Y
00091               00094E   58 10 A 0C4             0000EC             L       OPEN
                    000952   58 F0 A 0F0             000108             L
                    000956   05 EF                                      BALR
                    000958   90 77 A 098             0000C0             STM
00092               00095C   92 40 7 000                                MVI     MOVE
                    000960   02 82 7 001 7 000                          MVC
```


Figure 1—7. Object Code Generated from COBOL Source Code

The assembler converts each source instruction directly to a line of object code. Figure 1—8 shows a listing of an assembler program. The source code that was submitted to the assembler is shown at the right of the listing, and the object code generated by the assembler is shown at the left of the listing. Figure 1—8 has a BALR assembler instruction in the second source line that uses register 6 in operand 1 and register 0 in operand 2. As you can see, in the object code part of this listing, the assembler has converted the BALR source instruction to 0560_{16}. The 05_{16} is the machine code for a *branch and link* instruction; when the processor reads an 05_{16}, it will perform the BALR instruction. For a listing of the machine codes for all instruction opcodes, see Table E—1. The register numbers for the BALR source instruction are in the second half of the object instruction.

Very rarely will high-level language programmers read object code. Their concern is mostly with the language and the compiler. While assembler language programmers not only have the assembler and the assembler language to contend with, they also, if not just by sheer exposure, have to tolerate reading object code. This is because the assembler is really only one language step from the hardware processor. The only programming language left after assembler language is the nonsymbolic machine language. Although assembler language is closely related to the processor, it is still a symbolic programming language.

```
LOC.   OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT                                                  OS/3 ASM 80/01/03

000000                                 1 IFSAVE   START 0
000000 0560                            2 BEGIN    BALR  6,0
000002                                 3          USING *,6
000002 4100 60F6              000F8    4          LA    13,SAVE
000006 F263 60CA 60D1 000CC 000D3      5          PACK  PRINP,PRINZ
00000C F212 60D5 60D7 000D7 000D9      6          PACK  INTERP,INTERZ
000012 F272 60DE 60E6 000E0 000E8      7          PACK  TIMEP,TIMEZ
000018 4F40 60DE              000E0    8          CVB   4,TIMEP
00001C FC61 60CA 60D5 000CC 000D7      9 AGAIN    MP    PRINP,INTERP
000022 FA61 60CA 60E9 000CC 000EB     10          AP    PRINP,ROUNDEC
000028 D100 60CF 60D0 000D1 000D2     11          MVN   PRINP+5(1),PRINP+6
00002E D205 60EB 60CA 000ED 000CC     12          MVC   AREA,PRINP
000034 F865 60CA 60EB 000CC 000ED     13          ZAP   PRINP,AREA
00003A 4640 601A              0001C   14          BCT   4,AGAIN
00003E F346 60F1 60CA 000F3 000CC     15          UNPK  ANSWER,PRINP
                                      16          OPEN  OUT,(OUTRIB)
000044                              A 17+         CNOP  0,4                                                      P@000960
000044 4510 604E              00050 A 18+         BAL   1,*+12                                                   P@001150
000048 81                           A 19+         DC    X'81'                                                    P@001160
000049 000080                       A 20+         DC    AL3(OUT)                                                 P@001170
00004C 80                           A 21+         DC    X'80'                                                    P@001180
00004D 0000AC                       A 22+         DC    AL3(OUTRIB)                                              P@001190
000050 0A26                         A 23+         SVC   38 ISSUE SVC                                             P@002170
000052 0700                           24          CNOP  0,4
000054 D208 60BA 60F1 000BC 000F3     25          MVC   BUF(9),ANSWER
00005A 96F0 60C2        000C4         26          OI    BUF+8,X'F0'
                                      27          DMOUT OUT,BUF
00005E                              A 28+         DC    0Y(0) *                    SET ALIGNMENT                 P@I00690
00005E 5810 613E              00140 A 29+         L     1,=A(OUT) *              LOAD R1$, CDIB ADDRESS          P@I00710
000062 5800 6142              00144 A 30+         L     0,=A(BUF) *              LOAD R0$, WORKAREA ADDRESS      P@I00740
000066 9220 1002        00002       A 31+         MVI   2(1),X'20' *            SET FUNCTION CODE                P@I00890
00006A 9200 1003        00003       A 32+         MVI   3(1),0 *              SET FUNCTION CONTROL BYTE 1        P@I00891
                                    A 33+         SCALL 47                                                       P@I00920
00006E                              B 34+         DS    0H                                                       P$S00810
00006E 0AEF                         B 35+         SVC   239                                                      P$S01380
000070 10                           B 36+         DC    YL1(16)                                                  P$S01390
000071 2F                           B 37+         DC    YL1(47)                                                  P$S01395
000072 0A19                         B 38+         SVC   25                                                       P$S01590
000074 0700                         B 39+         NOPR  0                                                        P$S01650
000076 0A1C                         B 40+         SVC   28                                                       P$S01660
000076                              A 41+         ORG   *-2
000076 0A85                         A 42+         SVC   133
                                      43          CLOSE OUT
000078                              A 44+         DC    0Y(0)                                                    P@000280
000078 5810 613E              00140 A 45+         L     1,=A(OUT) LOAD R1 WITH FILENAME ADDRESS                  P@002020
00007C 0A27                         A 46+         SVC   39 ISSUE SVC                                             P@002030
                                      47          EOJ
00007E                              A 48+         DS    0H                                                       EOJ00050
00007E 0A1A                         A 49+         SVC   26                                                       EOJ00070
                                      50 OUT      CDIB
                                    A 51+         ENTRY OUT
```

Figure 1—8. Assembly Listing

The symbolic language for the assembler has two basic types of operation codes. Those that are translated directly to machine codes and those that are not. The mnemonic codes that aren't translated to machine codes are processed only at assembly time and do not become part of the object program. Nonmachine code mnemonics are used to direct the assembler when building an object program, while machine code mnemonics make up the actual object program. Operations codes that are translated to machine codes are called assembler instructions. For a complete listing of mnemonic instruction codes and their counterpart machine codes, see Table E—1.

There are three categories of nonmachine code mnemonics in the OS/3 assembler language: directives, conditional statements, and macros. Table 15—1 is a summary of assembler directives; Table 27—1 is a summary of conditional statements; and information on the macroinstructions available under OS/3 is included in the applicable user guide or programmer reference. The most commonly used macroinstruction types are data management and supervisor. See the consolidated data management concepts and facilities user guide, the consolidated data management macroinstructions user guide, the supervisor concepts and facilities user guide, and the supervisor macroinstructions user guide. The following listing shows the four elements of the assembler language and whether or not they are converted to executable code.

1.	Instructions	Machine Codes/Executable Code
2.	Directives	Nonmachine Codes/Nonexecutable Code
3.	Conditional statements	
4.	Macros	

As stated, the main function of the assembler is to produce an object program from a BAL source program. The object program created by the OS/3 assembler is called an object module and contains other information in addition to the machine code instructions translated from your source program. This other information is generated by the assembler so that OS/3 can recognize and process the object module. Figure 1—9 shows the format of the OS/3 object module. The shaded area indicates where the machine code program is located in the generated object module, and the remaining unshaded areas in the object module are used by the linkage editor, which is an OS/3 system program that creates another module called a load module. For further details about the contents of an object module, see the system service programs (SSP) user guide.

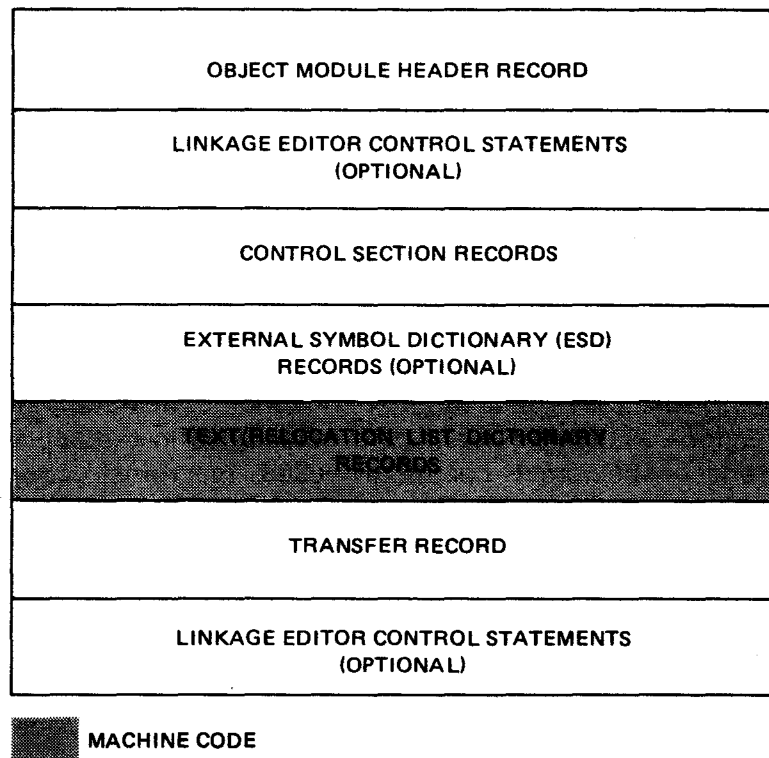


Figure 1—9. OS/3 Object Module Format

1.3. CREATING A LOAD MODULE

Assembling a program is only the first step in generating an executable BAL program. The complete process is a 3-step sequence and is generally called an assemble, link, and go operation. This means you must assemble an object module and create (link) a load module before you can execute (go) a BAL program. To set up an assemble, link, and go operation, or an assemble only (if only an assembly is required), you must use job control statements. Section 29 gives detailed information on how to run a BAL job.

Although the object module created by the assembler contains a BAL program in machine code form, it still isn't an executable program. To be executable (in OS/3) the object module must be changed to a load module. After an object module is generated, the assembler is no longer in control and the object module, left behind by the assembler, is used as input for creating a load module. The next OS/3 system program to gain control builds a load module from the object module. This system program is called the linkage editor.

The format of the load module produced by the linkage editor is illustrated in Figure 1—10. The shaded area indicates where the machine code program is located in the load module. For detailed information about the contents of a load module see the system service programs (SSP) user guide. Segments phase 1 through phase n shown in Figure 1—10 aren't created unless you specifically do so with linkage editor control statements. However, every load module will always have a root phase. After a load module is created, the BAL program is ready for execution.

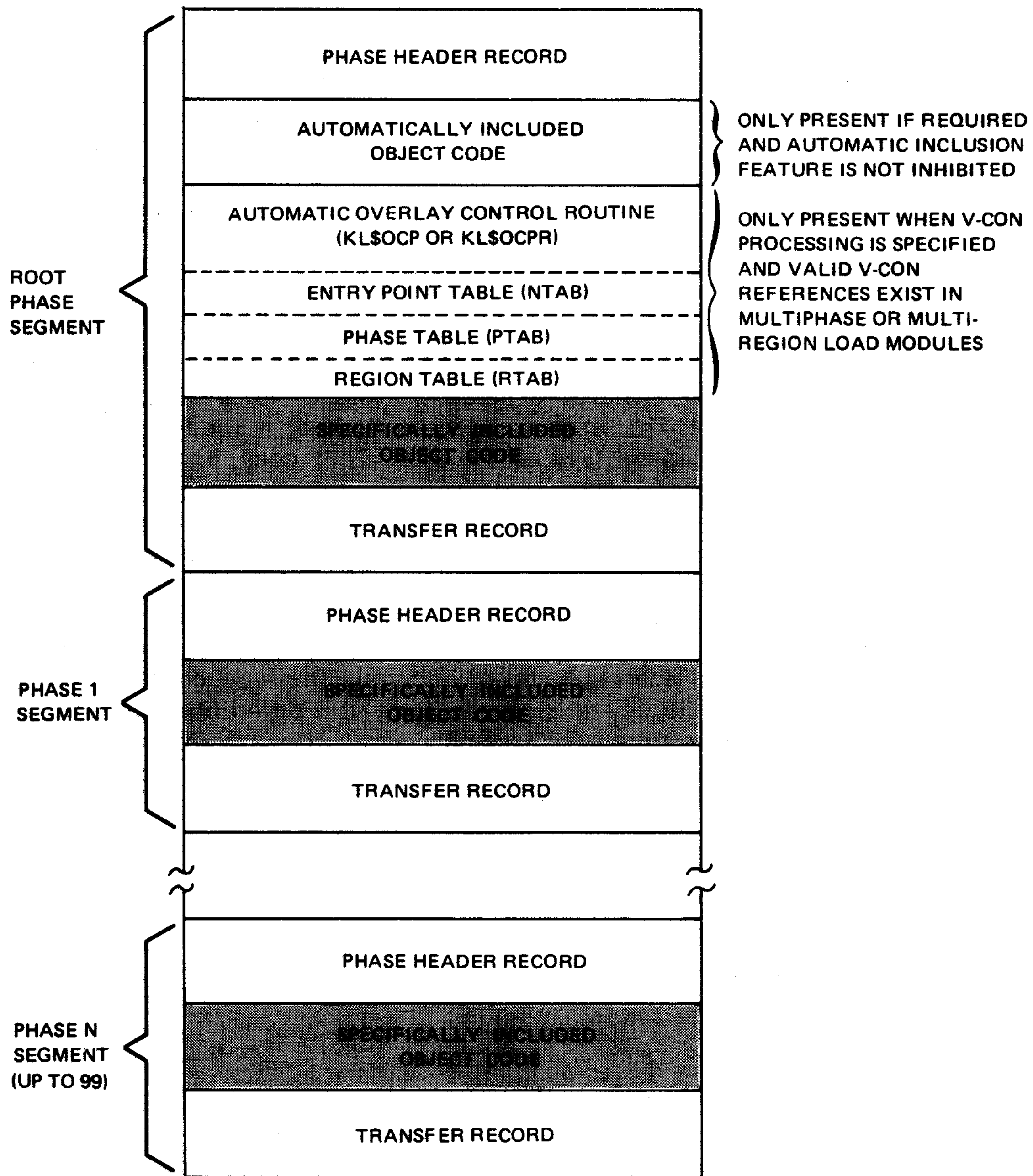


Figure 1—10. OS/3 Load Module Format

1.4. PROGRAM EXECUTION

During the assemble and link phase, each type of BAL module is on disk, while during the program execution phase, the machine program is stored in main storage as a load module. Figure 1—11 shows the location of each module after assembly time and linkage editor time. The source, object, and load modules are stored in a disk file called the job's run library file (YRUN). This file is an OS/3 system file, which is used to hold each BAL module until the assembler, linkage, and execution steps are finished.

The focal point of program execution is main storage. Once the load module is loaded from disk to main storage, the machine instructions are fetched one at a time from main storage by the processor. When the processor fetches an instruction, the opcode is decoded to find out which instruction is to be executed. If the instruction is legal, it is executed and the processor fetches another. This goes on until no machine instructions are left in the load module.

The only codes that the processor can interpret are the machine codes for assembler instructions. Any other codes submitted to the processor will cause an error, and the offending program is aborted. Every machine code instruction in the BAL instruction repertoire is supported by the microcode loaded into the control storage of the system. It is important to realize that machine codes, and only machine codes, can drive the hardware processor. This is the key to understanding the difference between program execution time and assembly time. At assembly time, the assembler processes the source program; the processor cannot execute a source program and doesn't see your program until it is in machine code form.

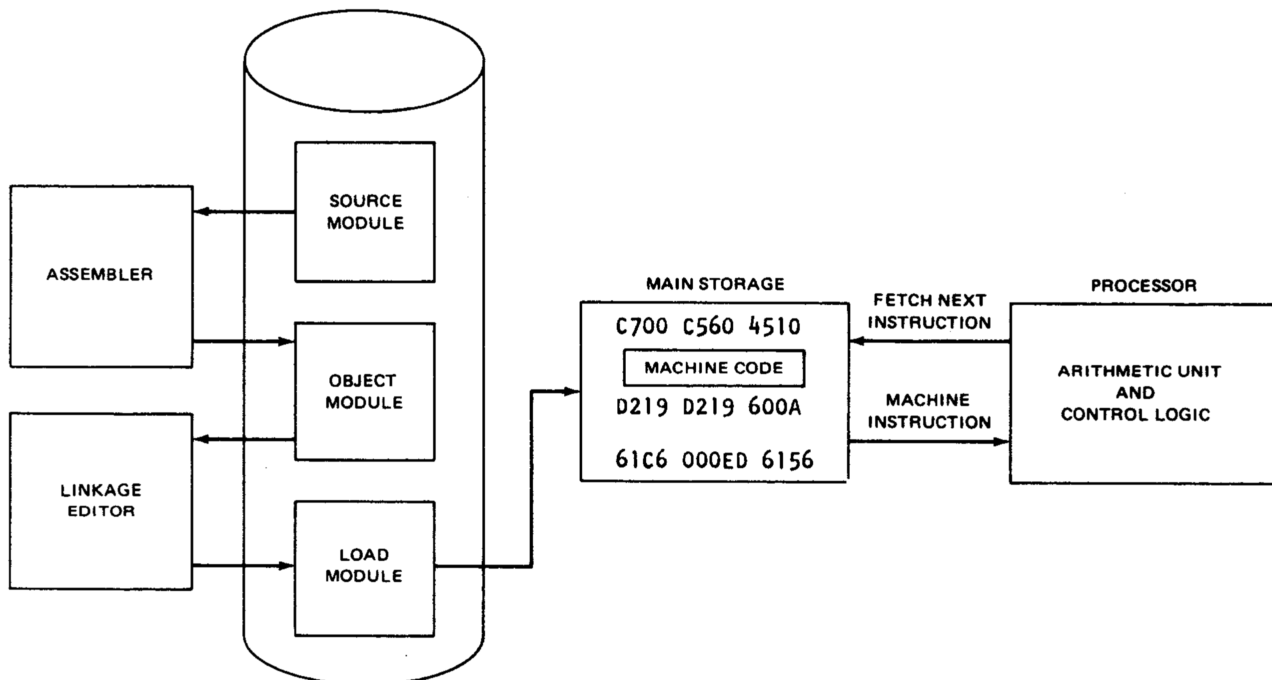
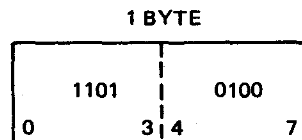


Figure 1—11. Assemble, Link, and Go Operation

2. Data Forms

2.1. DATA REPRESENTATION

Computer data is stored in special code combinations used to represent all the characters and numerical data needed for problem solving. The smallest area the computer can move or manipulate is called a byte, which is composed of eight units called bits. Each bit is either a 1 or a 0; thus, a byte representing the letter M would look like this:



Bits are numbered from left to right, with the leftmost bit referred to as the zero bit or the most significant bit (MSB). The rightmost bit in this byte is the number seven bit. The rightmost bit in any field, no matter how long, is also referred to as the least significant bit (LSB). Two contiguous bytes are called a half word; four are called a full word; and eight are a double word. When you manipulate several bytes as a string, the leftmost byte is called the most significant byte (MSB), and the rightmost byte is the least significant byte (LSB). Additional information on bit and byte structure is given in 4.3.2.

Table 2—1 comprises different methods of organizing and referencing numeric data as you would use them in data processing.

Table 2—1. Comparison of Numeric Expressions

Type of Number	Examples	Decimal Values
Character form (unpacked)	F 5 | F 0 | F 0	500
Zoned decimal (+)	F 5 | F 0 | C 0	+500
Zoned decimal (—)	F 5 | F 0 | D 0	—500
Packed decimal (+ only)	5 0 | 0 F	+500
Packed decimal, signed (+)	5 0 | 0 C	+500
Packed decimal, signed (—)	5 0 | 0 D	—500
Hexadecimal (+ only)	0 1 | F 4	+500
Floating point (+)	4 3 | 1 F | 4 0 | 0 0	+500
Floating point (—)	C 3 | 1 F | 4 0 | 0 0	—500
Binary (+ only)	0000 0001 | 1111 0100	+500
Binary (+ only)	1111 1110 | 0000 1100	+65,036
Fixed point (+)	0000 0001 | 1111 0100	+500
Fixed point (—)	1111 1110 | 0000 1100	—500

2.2. BINARY REPRESENTATION

In binary language, the same principles are followed as in decimal language. In decimal language (base 10), the number 251 is a combination of three values:

Sum	Hundreds	Tens	Units	
	2	5	1	
1			1	a value of one unit
50		5		the value of five 10's
+ 200	2			the value of two 100's
251				

In binary (base 2), the rightmost digit has the decimal value of 1; the digit to its left has a decimal value of 2, the next is 4, then 8, and so on to the most significant bit, which in one byte, has the decimal value of 128. You determine the total value of a binary number by adding the decimal value of each "on" bit (1), as illustrated in Figure 2—1.

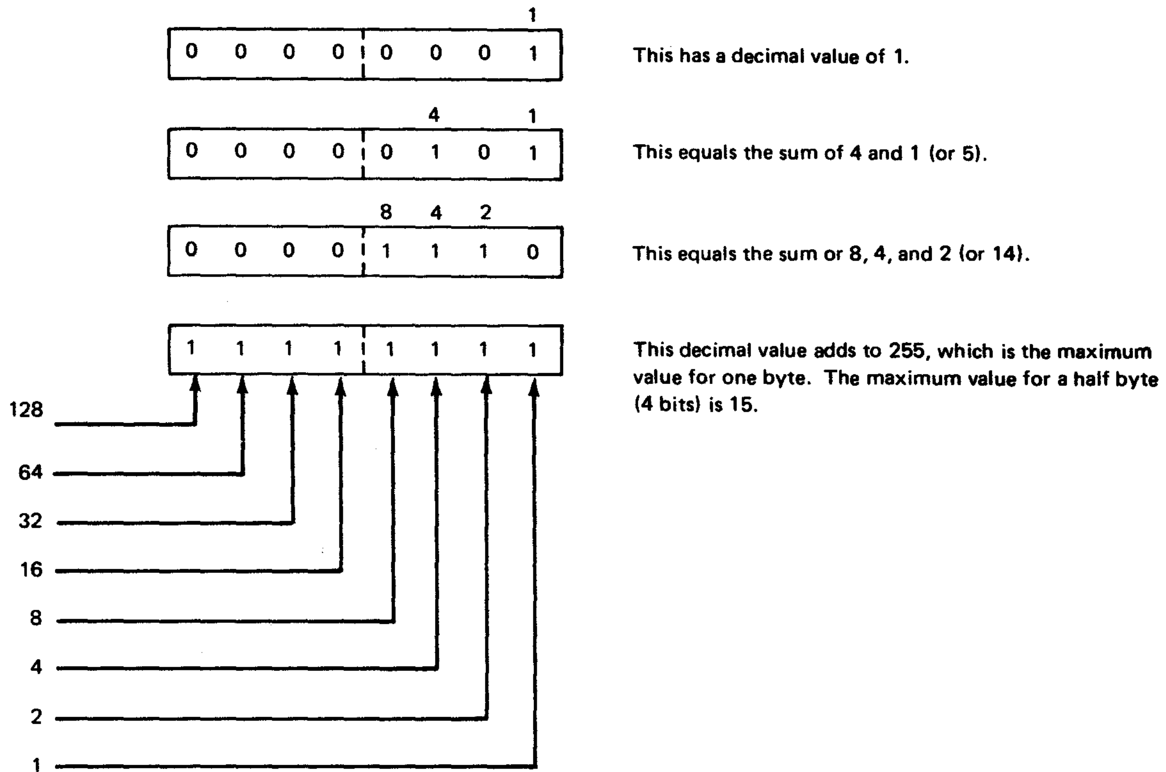


Figure 2—1. Determining Binary Values

Starting with the value of zero, a full byte represents a total of 256 different codes (B.1) and a half byte represents 16 codes. Since binary notation is unwieldy, most notations are written and computer-printed in other forms.

2.3. HEXADECIMAL REPRESENTATION

Using base 16 values, there are 256 hexadecimal codes in one byte. The hexadecimal notations consist of the numbers 0 through 9 and the letters A through F. In this way, we can represent the maximum decimal value of a half byte, which is 15, by one hexadecimal notation, which is F (Table 2—2). In B.1, the relationship of the binary, decimal, and hexadecimal codes for a full byte is shown.

Hexadecimal representation is an outgrowth of decimal and binary representation of data. In decimal, the base used is 10; therefore, the decimal number 251 is in actuality:

$$2\ 5\ 1 = 2 \times 10^2 + 5 \times 10^1 + 1 \times 10^0$$

which is the same as saying:

$$(2 \times 100) + (5 \times 10) + (1 \times 1) = 251$$

If we take this same number, 251, and show it in binary notation (that is, use the base 2), it would look like this for one byte:

1111	1011

This is the same as:

$$(1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

or:

$$(1 \times 128) + (1 \times 64) + (1 \times 32) + (1 \times 16) + (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1) = 251$$

Hexadecimal notation reduces the time and space needed to read or write the codes represented by a full byte of binary information. Because 16 is the base, to convert binary data to hexadecimal data, divide the binary representation of the decimal number into groups of four bits and pad to the left as necessary to obtain a full grouping of four bits. Thus, taking the binary representation of the decimal number 251 and breaking it up into groups as just described, we get:

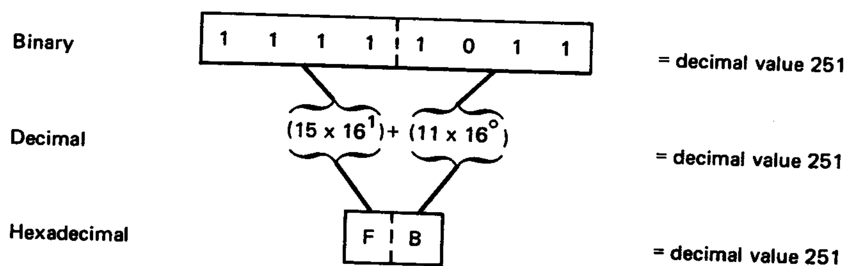


Table 2—2. Hexadecimal Notation

Binary Half Byte	Decimal Value	Hexadecimal Code
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2.4. CHARACTER REPRESENTATION

There are 256 possible bit combinations that can be stored in a byte. By convention, certain bit combinations are used to represent the letters, numerics, and special characters that are used to convey information in written form. In B.1 are listed the hexadecimal equivalents of the characters used to write programs. It is also pointed out in this document that only certain characters are used in statement formats. To aid in the specification of permissible characters, the overall character set of the assembler is divided into the following classes:

- Alphabetic set:
  - Alphabetic characters: the uppercase letters A through Z
  - Special letters: ? $ # @
- Numeric characters: 0 through 9
- Special characters: + - * / , = △ (blank) ( ) . & ' > <

2.4.1. Alphabetic Characters

The letters A through Z are alphabetic characters and part of the alphabetic set. The following table shows the hexadecimal representation, which is one byte long, for each of the uppercase letters. (Also see B.2.)

Alphabetic Character	Hexadecimal (EBCDIC) Code	Alphabetic Character	Hexadecimal (EBCDIC) Code
A	C1	N	D5
B	C2	O	D6
C	C3	P	D7
D	C4	Q	D8
E	C5	R	D9
F	C6	S	E2
G	C7	T	E3
H	C8	U	E4
I	C9	V	E5
J	D1	W	E6
K	D2	X	E7
L	D3	Y	E8
M	D4	Z	E9

2.4.2. Special Letters

The following special letters are part of the alphabetic set and usually follow the same rules as the letters mentioned in 2.4.1. The hexadecimal representation for these special letters are listed in the following table (also see B.2):

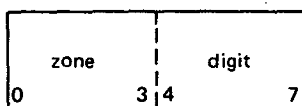
Special Letters	Hexadecimal (EBCDIC) Code
?	6F
$	5B
#	7B
@	7C

2.4.3. Numeric

As previously noted, all characters are coded in a full byte (eight bits), and this is also true for the character forms (unpacked) of numbers. Numbers written in this form, just as letters and other characters, can be moved from one location in main storage to another and can be sequenced, compared, and treated in other ways; but mathematical operations can not be performed on unpacked numerics. To do mathematical operations, the values must be in binary or packed decimal form. Unpacked and packed numeric formats are explained in 2.4.3.1 and 2.4.3.2. All numeric forms are shown in Extended Binary Coded Decimal Interchange Code (EBCDIC). For the American Standard Code for Information Interchange (ASCII), see B.3.

2.4.3.1. Unpacked Format

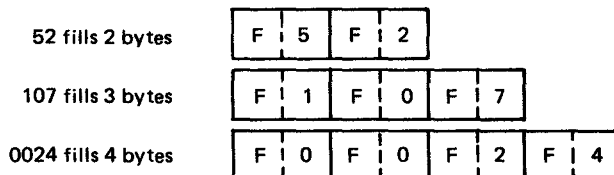
Unpacked (printable) numeric characters are coded in a full byte and are easily recognized because the first half of the coded byte is the hexadecimal code F. The decimal digit 5 is represented in a byte as F5. The F half of the byte (bits 0 through 3) is the zone field, and the 5 (bits 4 through 7) is in the digit field. Numeric data must be in this unpacked format to be output to a printer unit.



The following shows the hexadecimal 1-byte unpacked code for each decimal digit.

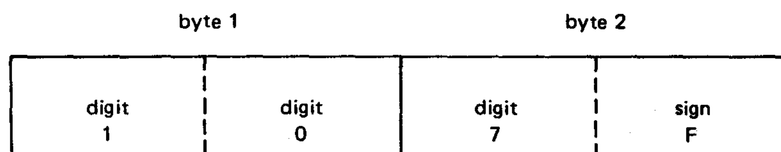
Decimal Digit	Hexadecimal (EBCDIC) Code
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

Examples of decimal digits and their unpacked byte equivalents are shown here.

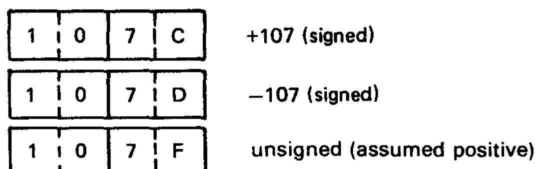


2.4.3.2. Packed Format

As you can see, unpacked format involves a considerable waste of main storage space. When numbers are to be processed, they can be converted to packed format by means of the PACK instruction, prior to processing. In packed format, the zone fields are stripped away and the number is stored as follows:



Thus, in packed format, only two bytes are needed to store the decimal number 107. This results in considerable savings in main storage space. After mathematical operations on a packed number, the sign C indicates a positive value and the sign D a negative value in EBCDIC.



The following program extract shows how the PACK instruction is used. (See 9.14.)

```
              PACK      AREA1,NO1       Area NO1 is packed into AREA1.
              AP        AREA1,ONE       A sign code is produced.
              .
              .
              .
NO1           DC        C'123'
AREA1         DS        CL2
ONE           DC        PL'1'
              .
              .
              .
```

After these operations, the two main storage areas will look like this:

```
NO1      | F ¦ 1 | F ¦ 2 | F ¦ 3 |
AREA1    | 1 ¦ 2 | 4 ¦ C |
```

The hexadecimal code C in AREA1 indicates that the value is positive. If the value is to be indicated as a negative value, a hexadecimal code of D would be in this field.

2.4.4. Special Characters

The following 14 special characters are not part of the alphabetic set (2.4.1), special letters (2.4.2), nor are they numeric (2.4.3). They have special uses, and rules are covered in this user guide when required. Following are listed the special characters with their hexadecimal codes for reference. (Also see B.2.)

Special Character	Hexadecimal (EBCDIC) Code	Special Character	Hexadecimal (EBCDIC) Code
+	4E	(left parenthesis	4D
- (minus)	60	) right parenthesis	5D
*	5C	. (period)	4B
/	61	&	50
, (comma)	6B	' (prime)	7D
=	7E	>	6E
Δ (blank)	40	<	4C

2.5. FIXED-POINT NUMBERS

Each fixed-point number is represented in one of three fixed-length binary formats composed of a single positive or negative sign bit followed by a number field (Figure 2—2). When the sign bit is 0, the number represents a positive value; when 1, the number represents a negative value. Negative numbers are represented in twos complement notation, which is derived by inverting each bit of the binary number and adding 1 to the result of the inversion. For additional information on fixed-point numbers, see 2.1, 5.2.6, 5.2.7, and Section 10.

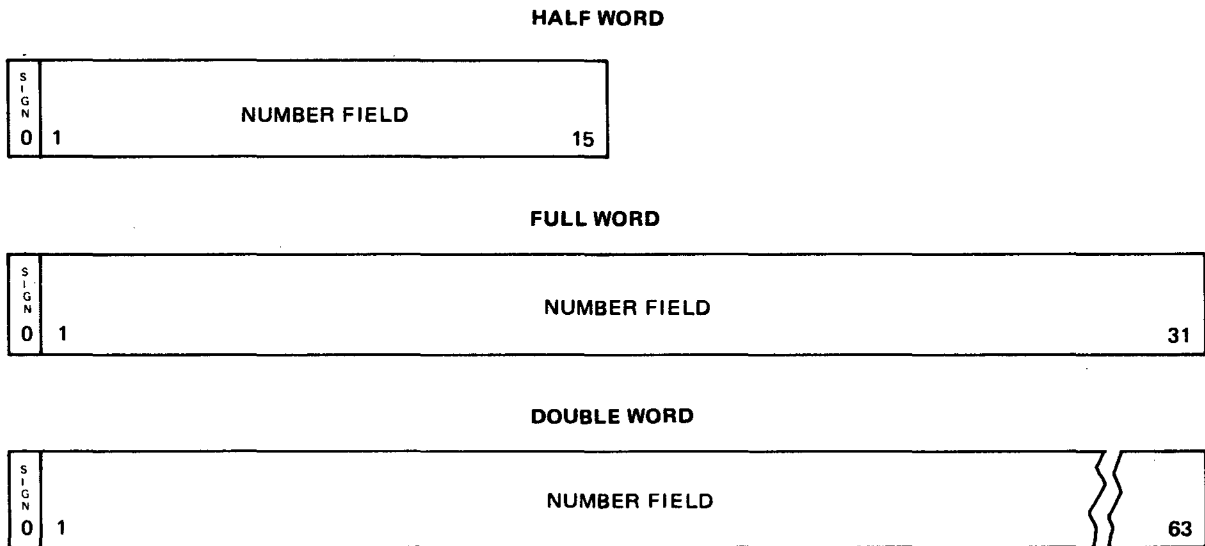


Figure 2—2. Fixed-Point Number Formats

2.6. FLOATING-POINT NUMBERS

The assembler provides floating-point arithmetic operations as an optional hardware feature. Floating-point arithmetic operations involve a fraction and an exponent. For example:

217,000 can be expressed as 0.217×10^6

296,000 can be expressed as 0.296×10^6

In fixed-point arithmetic, add:

```
  217,000
+ 296,000
---------
  513,000
```

In floating-point arithmetic, add:

$$
\begin{array}{r}
0.217 \times 10^{6} \\
+ \quad 0.296 \times 10^{6} \\
\hline
0.513 \times 10^{6}
\end{array}
$$

where:

0.513 is the fraction and 10^6 is the exponent.

In floating-point notation, the fraction is added and the exponent is retained. The example uses decimal floating-point; the assembler uses hexadecimal floating-point. In hexadecimal floating-point notation, the biased exponent is expressed in excess-64 binary notation; the fraction is expressed as a hexadecimal number having an arithmetic point to the left of the most significant 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 biased exponent (fraction x 16^{n-64}). For additional information on floating-point numbers, see 5.2.12 and Section 11.

S I G N	characteristic (exponent)	mantissa (fraction)

3. Addressing

Each full byte (eight bits) of main storage is numbered in sequence starting with 000000. With the assembler, the address of each instruction is calculated and you can refer to it by its real address or by a symbolic notation assigned to it. The assembly listing shows these addresses in their hexadecimal form. The computer also contains 16 registers that can be used for addressing and storage. The many types and uses of addressing are covered in detail in the following parts of this user guide.

3.1. MAIN COMPUTER STORAGE ADDRESSING

If you wish to refer to some other part of your program, you assign a symbol to that location and the assembler translates this to the real main storage address.

3.1.1. Instruction Addressing

Your program may contain the *move* instruction MVC:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1         10        16
---------------------------------------------------------------
MOVE5     MVC       MYAREA,YOURAREA
```

Even though the main storage for this application instruction is 00008A, you could return (branch) to this instruction by writing:

```
          B         MOVE5
```

This type of referencing a location in a program is called symbolic addressing. It is a time saver and helps eliminate many errors.

3.1.2. Data Field Addressing

As noted in 1.2, storage and data areas are defined for later reference. The following list shows assembler-generated addresses, the symbolic addresses assigned by you, and the storage areas.

Assembler-Generated Address	Symbolic Address	Definition	
000048	WKAREA1	DS	CL41
000071	WKAREA2	DS	CL16
000081	MYAREA	DS	0CL121
000081	OUTPUT1	DS	0CL121
000081	NEWAREA1	DS	CL41
0000AA	NEWAREA2	DS	CL80
0000FA	YOURAREA	DS	0CL121
0000FA	INPUT1	DS	0CL121
0000FA	COUNTS	DS	0CL3
0000FA	COUNT5	DC	CL1'5'
0000FB	COUNT12	DC	CL2'12'
0000FD		DS	CL118

The first work area shown, WKAREA1, has the hexadecimal location 000048 and is 41 bytes long. The hexadecimal value of 41 is 29, which added in hexadecimal produces the next hexadecimal location 000071. The next areas, MYAREA and OUTPUT1, show how we can assign different symbols to the same area. They do not take up main storage space and thus have the same address as NEWAREA1, which is 16 bytes from the start of the last address. The hexadecimal value of 16 is 10; thus, the address of NEWAREA1 is 000081. This address plus 41 bytes (hexadecimal 29) produces the next address, 0000AA.

The use of either the symbol MYAREA or OUTPUT1 calls for the same 121 bytes following them in storage. The zero placed in front of the CL instructs the assembler to assign a location for these symbols but not to reserve any storage for them. The remaining six instructions show how this can be done with constants (DC) as well. The symbol COUNTS is an example of a symbol reference within another symbol reference.

3.2. REGISTER ADDRESSING

There are 16 general registers (0 through 15). Each register consists of 32 bits which is equivalent to a full word. Any register can be used in RR, RS, or RX type instructions. Any register can also be used in base register assignment. However, most I/O operations use registers 14, 15, 0, and 1. So, if you use any one of these registers and then perform either input or output, the original data in these registers is destroyed. You can use these registers, though, by saving their contents prior to the execution of an I/O operation and restoring their contents after the execution of an I/O operation.

4. Rules and Conventions

4.1. READING INSTRUCTION NOTATION

Notations are used throughout this manual to describe the general forms of programmer-written and computer-generated formats. A consolidated listing of all the notations is included in Figure 7—1. This section includes the definitions of terms.

4.1.1. Assembler Application Instruction Notations

There are eight forms of assembler applications instructions:

RR — Register-to-Register

RX — Register-to-Indexed-Storage or Storage-to-Indexed-Register

RS — Register-to-Nonindexed-Storage or Storage-to-Nonindexed-Register

SI — Storage Immediate

SS — Storage-to-Storage (Type SS1)

SS — Storage-to-Storage (Type SS2)

S — Storage

SM — Storage Mask

Assembler application instructions provide the format for handwritten coding that, in turn, leads to the assembler format that generates the machine coding. The assembler application move instruction (MVC) illustrated is an SS1 type. The coding follows. Definitions of the explicit and implicit formats are provided in Section 7.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	MVC	$d_1(l_1,b_1),d_2(b_2)$

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	MVC	$s_1(l_1),s_2$

On the coding sheet, it could look like this:

```
LABEL   ΔOPERATIONΔ                       OPERAND
1        10         16
-------------------------------------------------------------
OUT4     MVC        AREA(37),NETPAY
```

where:

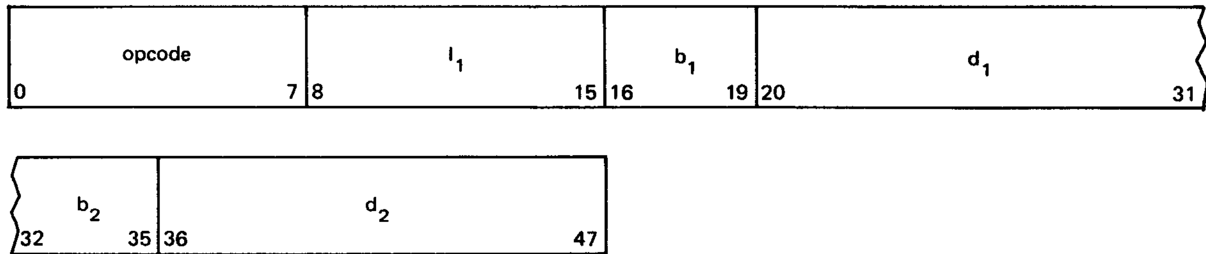
[symbol]
The brackets around *symbol* mean OUT4 is optional.

MVC
Is the mnemonic opcode for the *move* instruction.

AREA(37)
Is the first operand: d_1 (l_1,b_1) or s_1 (l_1) AREA is the address $d_1(b_1)$ or symbol s_1 and (37) is the length (l_1) of the receiving field to be filled.

NETPAY
Is the second operand: $d_2(b_2)$ or s_2.

After this application instruction is assembled, it is in the following form:



And could have the generated machine code:

D2 24 44FC 4AA6

where:

D2
: Is the operation (opcode) for the mnemonic MVC.

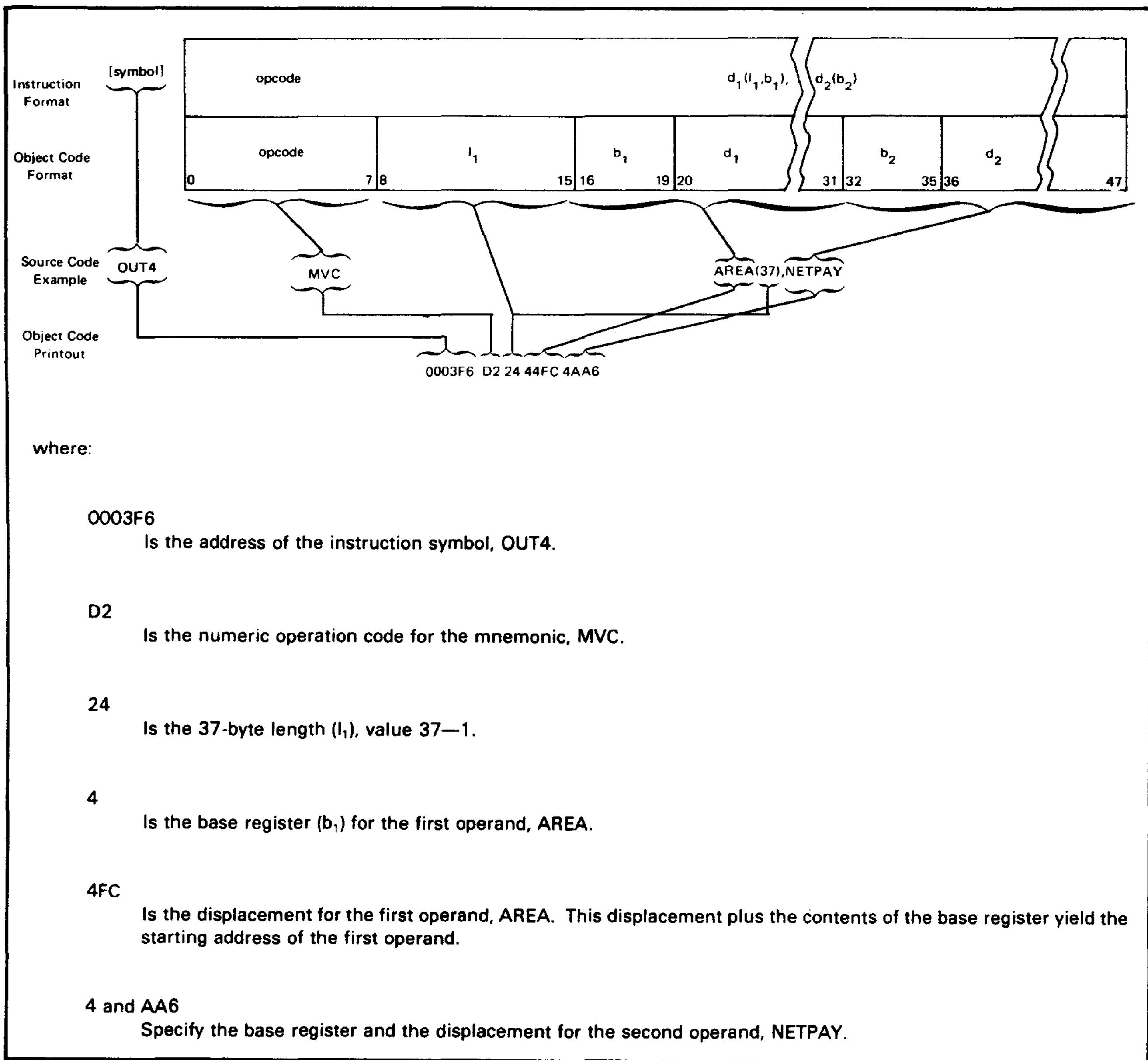
24
: Is the hexadecimal coding for the length (l_1), which is 37 bytes long but assembled as 37—1 or 36.

4
: Is the base register b_1 used for the first operand.

4FC
: Is the displacement d_1 used for the first operand.

4AA6
: Is the base b_2 and displacement d_2 address of the second operand NETPAY.

The generated machine code is expressed in hexadecimal form. Knowing the organization of the machine code format can help you when the written coding does not generate the values you intended. Such knowledge helps in finding errors in the results of a program. (See Figure 4—1).



where:

0003F6
: Is the address of the instruction symbol, OUT4.

D2
: Is the numeric operation code for the mnemonic, MVC.

24
: Is the 37-byte length (l_1), value 37—1.

4
: Is the base register (b_1) for the first operand, AREA.

4FC
: Is the displacement for the first operand, AREA. This displacement plus the contents of the base register yield the starting address of the first operand.

4 and AA6
: Specify the base register and the displacement for the second operand, NETPAY.

Figure 4—1. Assembler Format Relationships

Consider another instruction:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	L	6,GROSSPAY(5)

where:

L
Is the mnemonic opcode for the load instruction.

6
Is the first operand register r_1.

GROSSPAY (5)
Is the second operand in the form $s_2(x_2)$.

After this application instruction is assembled it may generate the following machine code:

58654012

where:

58
Is the opcode for the load instruction.

6
Is the register r_1 used for the first operand.

5
Is called the index register and is part of the second operand (x_2).

4012
Is the base b_2 and displacement d_2 for the second operand address GROSSPAY. For a discussion of how the processor uses the base register, index register, and displacement of an operand to form main storage addresses, refer to the current version of the processor programmer reference.

Figure 7—1 shows the formats of the eight application instructions as generated by the assembler in machine code, as well as the explicit and implicit formats for programmer coding. Examples of the implicit coding format using symbols and the explicit format are included in following sections for each assembler application instruction. More detailed information on the use of the assembly listing is in Section 28.

4.1.2. Notation Rules and Meanings

The following conventions are used in application-instruction, assembler-directive, macroinstruction, proc, and control-statement formats:

- Optional information is enclosed in brackets [ ] and may be specified or omitted as in the use of [symbol].

- Braces { } indicate multiple options, at least one of which must be chosen, as in the following directive.

  For example:

  PRINT {ON / OFF}

- Braces within brackets signify that one of the options must be chosen if that operand is specified.

  For example:

  [{}]

- When given a choice of multiple options, the option that is shaded is the default option and indicates the choice that will be made by the system if you do not specify one of the options.

  For example:

  [{DATE / EXT / ID / PRE}]

- Uppercase letters, terms, and punctuation marks indicate information which must be coded exactly as shown. Also, mnemonic codes (such as MVN, PACK, and CLC) are in uppercase letters.

- Lowercase letters and terms indicate variables (such as [symbol], r, d, b, and e) which are supplied by you.

- An ellipsis, a series of three periods, indicates that a series of entries may be coded, as in the directive DROP r,[, r_2,..., r_n].

- Keyword parameters may be coded in any order.

  For example:

  IOROUT=LOAD,BLKSIZE=512,RECFORM=FIXBLK
  BLKSIZE=512,IOROUT=LOAD,RECFORM=FIXBLK

- Positional parameters must be coded in the order shown. Commas are required after each positional parameter except the last. When a positional parameter is omitted from a series of positional parameters, the comma must be retained to indicate the omission.

  For example:

  X'03',OUTP,X'00',132 (operand field of CCW)
  &P,3,&KEY1=,&KEY2=,&KEY3=(operand field of macro statement in proc format)

- Names of directives and instructions in text are shown in lowercase italics.

  For example:

  add, move, load, branch and link, store

- Throughout this manual, the register notations R0 through R15 represent the registers 0 through 15.

  For example:

  BALR R2,R3

The handwritten program, usually on the assembler coding form, is called the source program; the card images containing this coding are still called the source program. The source program is assembled, and the assembler usually produces a translation of the source program into machine code; this deck is called the object program. A printed listing of the assembled program, called the assembly listing, shows the source coding with its associated assembled machine coding.

The smallest unit of information in basic assembly language (BAL) is the bit. Eight bits make a byte and two bytes form a half word. Four bytes are a full word and eight bytes comprise a double word. Figure 4—2 shows the relationships between bits, bytes, and words. Bits 0 through 7 form the high-order byte or MSB, and bits 56 through 63 form the low-order byte or LSB in a double-word storage area.

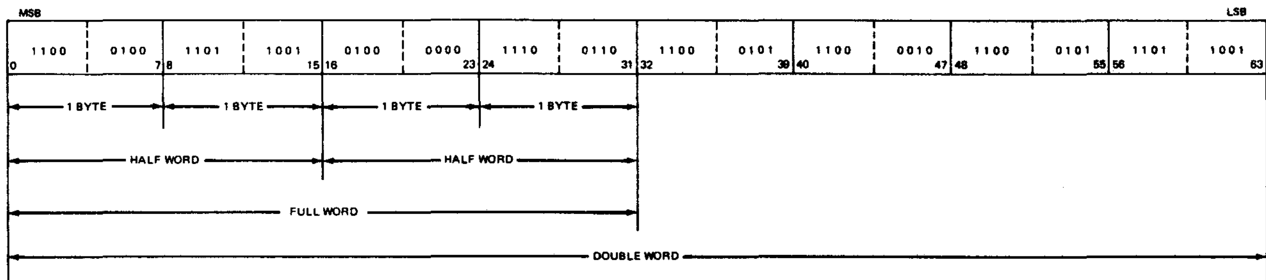


Figure 4—2. Byte and Word Structure

The following short definitions should be useful for the new programmer.

- Source program

  Programmer-produced

- Source cards

  Keypunch output

- Source deck

  Keypunch output

- Source code

  Keypunch, disk, or diskette output

- Machine code

  Assembler-generated

- Object program

  Assembler output

- Assembly listing

  Assembler output to printer

- Bit

  One binary digit

- Byte

  Eight binary digits

- Half word

  Two bytes

- Full word

  Four bytes

- Double word

  Eight bytes

- MSB

  Most significant bit or byte, leftmost

- LSB

  Least significant bit or byte, rightmost

- High order

  Leftmost data, byte, or bit

- Low order

  Rightmost data, byte, or bit

4.2. TERMS

Terms represent values coded by the programmer or computed by the assembler. There are five classes of terms recognized by the assembler:

- Self-defining terms (SDT)
- Literals
- Symbols
- Location counter references
- Length attribute references

Self-defining terms are fixed values the programmer codes such as 33,P'591',X'OF',B'11100110', or C'EBW'. Literals can have their value specified by the programmer and their location decided by the assembler and could look like, =X'FO',=C'A',=P'—1', or =B'00001000' as used in storage-to-storage instructions (e.g., CLC TAGA,=C'A'). Symbols, location counter references, and length attribute references are assigned values by the assembler. (See Table 4—1.)

Table 4—1. Comparison of Terms

Term	Examples	
SDTs ■ Can be used in the 1st or 2nd operands. ■ May be used in application instructions and in assembler directives.	CLI MVI MVC	AREA10, 10 (SDT) AREAB, X'C2' (SDT) 33 (10,R5),3(R8) (SDT SDT SDT)
Literals ■ May not be used in assembler directives. ■ Literals are preceded by an equal (=) sign.	MVC MVC CLC	AREA10,=C'10' (Literal) AREA10,=X'F1F0' (Literal) ONSW,=B'11111111' (Literal)
Symbols for constants ■ May be used in the 1st or 2nd operands. ■ May be used in application instructions and in assembler directives.	AREA10 NO10 MOVE10	DS CL2 DC C'10' MVC AREA10,NO10 (symbols)

4.2.1. Self-Defining Terms (SDT)

Self-defining terms (SDT) are terms that represent fixed values. They are presented by the programmer in a form that is easily recognized and its value is understood without the need for computation. SDTs are not relocatable; they can be used to specify immediate data, registers, addresses, and masks. They can be used in assembler directives as well as in application instructions and can be part of an expression. The size of an SDT depends on where it is used. When used to designate a register you cannot exceed a value of 15. After conversion by the assembler to a binary format, the value is right-justified and filled with binary zeros on the left to fit the designated field. SDTs can be represented in binary, hexadecimal, decimal, or character form. (See 5.2.)

When a 24-bit hexadecimal, binary, or character SDT has a 1 in the sign bit position the SDT will be treated as a negative term in the evaluation of an arithmetic expression.

- A binary SDT consists of a series of up to 24 zeros and ones enclosed in apostrophes and preceded by the letter B (e.g., B'101', B'11110000', B'00101'). The field is filled with high order zeros when necessary.

- A hexadecimal SDT consists of up to six hexadecimal digits enclosed in apostrophes and preceded by the letter X (e.g., X'F0', X'C1', X'F1F0F0'). Each hexadecimal digit represents a half byte of information.

- A decimal SDT is an unsigned decimal number consisting of up to eight digits having a value of 0 through 16,777,215 (2^{24}—1) (e.g., 0, 32, 16000000). This number is converted by the assembler to a binary value occupying one, two, or three bytes.

- A character SDT consists of up to 3 characters of the 256 valid characters of which only 63 are printable. (See Appendix B, Table B—1.) The characters must be enclosed in apostrophes and preceded by the letter C (e.g., C'A', C'ABC', C'123', C'A1'). Each ampersand or apostrophe to be included in a character representation must be indicated by a double ampersand or double apostrophe, respectively. In this case there may be more than three characters within the apostrophes which delimit the SDT (e.g., C'3''S' produces 3'S; C'A&&B' produces A&B).

The following four examples all produce the same internal bit pattern of 11110001 in the one byte area called AREA:

- Decimal CLI AREA, 241
- Hexadecimal CLI AREA, X'F1'
- Character CLI AREA, C'1'
- Binary CLI AREA, B'11110001'

4.2.2. Literals

Literals are terms that represent data in the source coding (see 5.3). The assembler replaces the value of the original literal in the literal table (pool) with the address of the main storage location. In the following example the literal =C'AA' will be replaced in this instruction by the address of a 2-byte area in the literal table containing the binary value 11000001 11000001.

```
MOVEAA    MVC    TESTSW,=C'AA'
```

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 literal may be used in any machine instruction that specifies a storage address, except that the literal may not be specified as the receiving field operand of an instruction that modifies storage. Literals may not be specified in address constants, shift instructions, or I/O instructions. Literals must always appear as the complete operand specification. They cannot be combined with other terms, nor with an explicit base register specification. 'S' type constants may not be used as literals.

4.2.3. Symbols

A symbol is a group of up to eight alphanumeric characters. The left, or leftmost, character must be alphabetic. Special characters or blanks may not be contained within a symbol. (See Section 6.) The following are examples of valid symbols:

V	CARDAREA
GS279	R$INTRN
BOB	BD#4

The following are not valid symbols for the reasons stated:

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

Two other categories of symbols are available in the macro language and conditional assembly statements. They are variables and sequence symbols. These categories of symbols are defined and discussed in detail in Section 6 and in 27.2.1.

The assembler associates three attributes with each symbol it processes. These attributes are value, length, and relocatability. Symbols defined by the EQU directive adopt the attributes of the expression in the operand field of the statement. (See Section 16.)

- 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, full-word, or double-word boundary (5.1.7), 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.

  The value of a symbol must lie in the range -2^{23} through $2^{23}-1$.

- 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 200 bytes would have a length attribute of 200. Symbols equated to location counter references or absolute value representations usually have a length attribute of 1. The duplication factor (constant or storage area) has no effect on the length attribute. (See 5.1.3.)

  The maximum length attribute that can be generated by the assembler is 65,536.

- 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 address 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.

4.2.4. 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 correct location counter. The maximum value that the location counter can achieve is $2^{23}-1$.

Each instruction 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 an instruction, a 1 is added to the location counter before assigning an address to the current statement. Storage locations reserved in this way receive binary 0's when the program is loaded. Certain constants must be aligned to a half-word, full-word, or double-word boundary. Again the location counter is adjusted to the boundary, and the storage locations which were bypassed receive binary 0's when the program is loaded unless the adjustment occurred as a result of a DS or ORG directive.

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 asterisk character (*). If the asterisk is written as a term in an address constant or in an instruction operand expression, this character is replaced by the storage address of the leftmost byte allocated to that instruction or constant. Care must be taken to ensure that all such implied references are specified appropriately in individual expressions since the character asterisk (*) may also be used to indicate the multiply operator during the evaluation of expressions.

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 referencing instruction and the instruction or data being accessed. Relative addressing is always in terms of bytes and not in terms of words or instructions.

A location counter reference may not be made in a statement which requires the use of a predefined symbol, with the exception of the EQU and ORG directives.

4.2.5. Length Attribute Reference

The length attribute of a symbol is referenced as a term in an expression 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. (See 5.1.5.)

4.3. OPERATORS

There are 12 operators in the assembler language (Table 4—2) which designate the method and sequence to be employed in combining terms or expressions. 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 hierarchical order as listed in Table 4—2. The operation with the highest hierarchy number is performed first; operations with the same hierarchy number are performed from left to right.

Parentheses can be used to alter the hierarchy of evaluation. Multiplication 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 4.3.1, 4.3.2, and 4.3.3.

Table 4—2. 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 of A and B.	2
Relation 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

4.3.1. Arithmetic Operators

The symbols +, —, *, /, //, */ represent the six arithmetic operators. The intrinsic meanings of +, —, *, and / are the usual ones; that is, + indicates addition, — indicates subtraction, *indicates multiplication, and / indicates 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 $A/2^B$.

4.3.2. Logical Operators

The symbols **, ++, and —— are the three logical operators. The characters ** represent the logical product (AND), and characters ++ represent the logical sum (OR), and the characters —— represent the symmetric 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. (See Section 12.) 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

4.3.3. 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, than 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 a value of C, then the assembler assigns a value of 1 to the expression; otherwise, a value of 0 is assigned.

A relational expression consists of a relational operator and its two operands. The operands in a relational expression may be either two character expressions (4.4.5) or two arithmetic expressions. A character expression may not be compared to an arithmetic expression. Character expressions are valid only on conditional assembly directives.

Since the evaluation of a relational expression yields an arithmetic result, a relational expression may be used as a term in an arithmetic expression.

4.4. 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. Each term in the expression may be either a relocatable term or an absolute term. A term is absolute if its value is not changed by program relocation. A term is a relocatable term if its value is changed by program relocation. Two relocatable terms may be considered to be paired if they have opposite signs and have the same relocatability attribute (that is, appear in the same control section).

Evaluation of expressions obeys the following rules:

- Multiplication and division of a relocatable term by an absolute 1 or multiplication of an absolute 1 by a relocatable term produces a relocatable term.
- Multiplication of any term by absolute 0 yields absolute 0 as a result.
- If a relocatable term enters any multiply or divide operation other than the preceding, an error flag is given.
- The number of unpaired relocatable terms at any point in the evaluation must not exceed 16.
- Intermediate results of the expression evaluation are full 32-bit values; however, the final result is the truncated rightmost 24 bits.

Three types of expressions, absolute, relocatable, and complex relocatable obtain various characteristics from the term or terms which compose them. These three types of expressions are discussed in 4.4.1 through 4.4.6.

4.4.1. Absolute Expressions

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. Arithmetic operators are permitted between absolute terms.

Examples of absolute terms are: a symbol which has an absolute value, a self-defining term or a length attribute reference.

Relocatable terms alone or relocatable terms in combination with absolute terms can be contained within an absolute expression. This type of absolute expression requires that each relocatable term be paired with another relocatable term which has the opposite sign and the same relocatability attribute. The paired terms need not be contiguous.

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.

4.4.2. Relocatable Expressions

A relocatable expression is an expression whose value changes with program relocation. All relocatable expressions must be positive values.

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 following conditions:

- 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.

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

R
R/1
R+A or A+R
R—R+R
R—A
R*1 or 1*R

where:

A
Is an absolute term.

R
Is a relocatable term.

4.4.3. Complex Relocatable Expressions

A complex relocatable expression is an expression that contains 2 to 16 unpaired relocatable terms or a negative relocatable term in addition to any absolute or paired relocatable terms.

A complex relocatable expression may be written only in the operand field of either an A-type or Y-type address constant. (See 5.2.8 and 5.2.9.)

Some examples of complex relocatable expressions are:

A—R
—R/1
A—R—R+R—R

where:

A
Is an absolute term.

R
Is a relocatable term.

4.4.4. Character Expressions

A character expression is either a character string, a character substring, or a concatenation of strings or substrings. Character expressions are used as the operand of a SET or SETC statement or as terms in a SETB, SET, AIF, or DO relational expression. Any character string is considered to be greater in value than any shorter character string. A character expression may have a length of up to 127 characters.

4.4.5. Length Attribute of Expressions

The length attribute of an expression is determined by the assembler and it 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.

4.4.6. Character Strings

A character string is at least one of the 256 valid characters enclosed by apostrophes. A character string, unlike a character self-defining term, is not converted and treated as a binary value. The value of a character string is determined by its length. Any character string is greater in value than any shorter character string. Rules for writing character strings are:

- Two apostrophes must be written within a character string to represent one apostrophe. The two apostrophes are replaced by a single apostrophe.
- Two ampersands must be written within a character string to represent one ampersand. A single ampersand within the character string is interpreted as the first character of a variable symbol.

A character substring is a valid character string followed by two arithmetic expressions separated by a comma and enclosed in parentheses. The format is:

character string (e_1,e_2)

where:

e_1
Specifies the leftmost character of the original string to be included in the substring.

e_2
Specifies the number of characters to be in the substring.

The expressions e_1 and e_2 must be valid SET expressions. (See 27.1.4.) If there are fewer characters (than the number specified by e_2) remaining after character number e_1 in the string, the resultant substring is shortened to include only valid characters of the original string. A null character string results if e_1 is greater than the number of characters in the original string.

Example:

'PREDEFINED' (4,6)

will produce the character substring

'DEFINE'

Concatenation is the joining together of:

- two character strings;
- two character substrings; or
- a character string and a character substring.

A period designates concatenation into a single string of characters.

Example:

'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.

PART 2. STORAGE AND SYMBOL DEFINITIONS

5. Storage Definitions

In almost all programs, inclusion of constant value is required for mathematical computation, headings for reports, and values or codes for comparisons. You also reserve storage for work areas, record keeping, and save areas. Two methods used to produce constants or reserve storage are:

- *define constant* — DC
- *define storage* — DS

5.1. STORAGE USAGE

There are 13 definition types used to describe the type and format of storage used. Table 5—1 lists the characteristics of each of these storage notations. All the definition types shown are valid for both DC and DS statements. Except for floating-point constants (2.9, 5.2.12, and Section 11), the formats of both statement operands are similar, as follows:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	DC	$[d]t[L_n]\left\{\begin{matrix}'c'\\(c)\end{matrix}\right\}$
[symbol]	DS	$[d]t[L_n]\begin{bmatrix}'c'\\(c)\end{bmatrix}$

where:

[symbol]
Is up to eight characters.

d
Is the duplication factor in decimal.

t
Is the definition type. (See Table 5—1.)

L_n

Is the explicit length factor in decimal.

'c'

Is the constant specification for data.

(c)

Is the constant specification for an address.

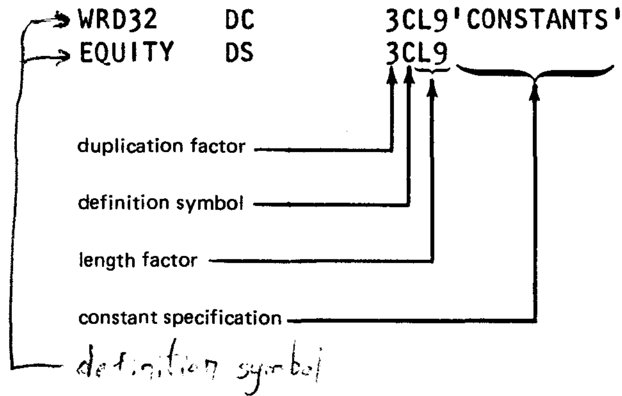
Table 5—1. Characteristics of Constant and Storage Definition Types

Type Code	Constant or Storage Type	Alignment	Source Code Specification		Storage Format	Truncation or Padding	Length in Bytes		
							Implied	Minimum Explicit	Maximum Explicit *
C	Character	None	Characters	C' '	Character	Right	Variable	1	256 (DC) 65,535 (DS)
X	Hexadecimal	None	Hexadecimal digits	X' '	Hexadecimal	Left	Variable	1	256 (DC) 65,535 (DS)
B	Binary	None	Binary digits	B' '	Binary	Left	Variable	1	256
P	Packed decimal	None	Decimal digits	P' '	Packed decimal	Left	Variable	1	16
Z	Zoned decimal	None	Decimal digits	Z' '	Character	Left	Variable	1	16
H	Half word, fixed point	Half word	Decimal digits	H' '	Fixed-point binary	Left	2	1	8
F	Full word, fixed point	Full word	Decimal digits	F' '	Fixed-point binary	Left	4	1	8
Y	Half-word address	Half word	Expression	Y( )	Binary	Left	2	1	2
A	Full-word address	Full word	Expression	A( )	Binary	Left	4	1	4
S	Base and displacement	Half word	One or two expressions	S( )	Base and displacement	None	2	2	2
V	External address	Full word	Relocatable symbol	V( )	Binary	Left	4	3	4
E	Full word, floating point	Full word	Decimal digits	E' '	Floating-point binary normalized	Right	4	1	8
D	Double word, floating point	Double word	Decimal digits	D' '	Floating-point binary normalized	Right	8	1	8

*The maximum explicit length in bytes is that total length produced by the explicit length factor times the duplication factor.

Following are DC-statement and DS-statement examples showing the use of the subfields, which must appear in the order stated and must not be separated by blanks.

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10      16
```



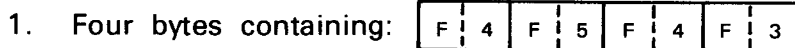
DC

5.1.1. Define Constant (DC)

The *define constant (DC)* statement is processed by the assembler and the constant specification is translated into object code representing the required values. The maximum explicit length for a DC is 256 bytes. (See Table 5—1 for C, X, and B types.)

The following five examples show the use of the subfields in a DC statement.

```
     LABEL    ΔOPERATIONΔ                         OPERAND
     1        10        16

1.   N4543    DC        C'4543'
2.   NAD      DC        2C'4543'
3.   NAHS     DC        CL2'4543'
4.   WRD      DC        2CL2'4543'
5.   L591     DC        5CL1'4543'
```

1. Four bytes containing: 

   N4543 has a length attribute of four bytes, the number of bytes assigned the value '4543'.

2. Eight bytes containing:

   NAD also has a length attribute of four bytes, as called for by the value '4543', even though the duplication factor calls for two such fields of four bytes each.

3. Two bytes containing:

   NAHS has a length attribute of two bytes, as specified by the length modifier, and only two bytes of storage are used. The digits 4, 3 are ignored.

4. Four bytes containing:

   WRD also has a length attribute of two bytes, as specified by the length modifier, but two fields of two bytes each are used because of the duplication factor of 2. The digits 4, 3 are ignored.

5. Five bytes containing:

   L591 has a 1-byte length attribute, as specified by the length modifier of 1. There are five 1-byte fields, as called for by the duplication factor. The digits 5, 4, 3 are ignored.

5.1.2. Define Storage (DS)

The *define storage (DS)* statement is processed by the assembler and the constant specification is translated into reserved storage. The maximum explicit length for a DS is 65,535 bytes. (See Table 5—1 for C and X types.) The following five examples show the use of the subfields in a DS statement. Only the number, not the content, of the bytes reserved by a DS statement is determined by the assembler.

```
      LABEL    ΔOPERATIONΔ                    OPERAND
      1        10       16

  1.  ILE      DS       C'4543'
  2.  AYAHC    DS       CL4
  3.  DNOMYAR  DS       2CL4
  4.  REBEW    DS       5CL1
  5.  OREG     DS       3C'Δ NO.Δ'
```

1. ILE reserves a 4-byte field with a length attribute of 4.

2. AYAHC produces the same result as line 1.

3. DNOMYAR reserves eight bytes composed of two fields of four bytes each. The length attribute of DNOMYAR is 4.

4. REBEW reserves five bytes of storage consisting of five fields of one byte each. The length attribute here is 1.

5. OREG reserves 15 bytes of storage. The constant field defines a 5-byte field, and the duplication factor calls for three of these fields. The length attribute of OREG is 5.

5.1.3. Duplication Factor

The duplication factor designates the number of identical constants or areas to be generated. An unsigned decimal value is used to specify the duplication factor. If no duplication subfield is used, the assembler assumes a factor of 1. A duplication factor of zero generates neither a constant nor a storage area and, if no length factor is specified, the location counter will provide the proper boundary alignment and assign the location counter value to the symbol used. A duplication factor of zero is not permitted with literals. (See Table 5—2 for an example of the use of the zero duplication factor.) Note that, even though the duplication factor can change the size of the storage area used, the use of the duplication factor does not change the length attribute of the field. (See 5.1.5.) The maximum value of the duplication factor is 256.

Table 5—2. Zero Duplication Area Examples

Address	Symbol	Operation	Operand
000D48	WKAREA1	DS	CL41
000D71	WKAREA2	DS	CL16
000D81	WBE	DS	0CL121
000D81	OUTPUT1	DS	0CL121
000D81	NEWAREA1	DS	CL41
000DAA	NEWAREA2	DS	CL80
000DFA	SGAW	DS	0CL121
000DFA	INPUT1	DS	0CL121
000DFA	COUNTS	DS	0CL3
000DFA	COUNT5	DC	CL1'5'
000DFB	COUNT12	DC	CL2'12'
000DFD		DS	CL118

The first work area shown, WKAREA1, has the hexadecimal location 0000D48 and is 41 bytes long. The hexadecimal value of 41 is 29, which is added in hexadecimal to produce the next hexadecimal location, 000D71. (See C.1.) The next areas WBE and OUTPUT1 show how we can assign different symbols to the same area. They do not take up storage space and so would have the same address of NEWAREA1, which you can see is 16 bytes away from the start of the last address. The hexadecimal value of 16 is 10, making the address of NEWAREA1 000D81. Now plus 80 bytes (hexadecimal 50) produces the address 000DFA.

5.1.4. Definition Type

The definition-type symbol is required for both DC and DS statements to determine the alignment (5.1.7), padding, truncation, storage form, and implied length. (See 5.2 and Table 5—1 for the characteristics of the 13 types used.)

5.1.5. Length Factor (L_n)

The length factor designates the explicit value of the length attribute of a field generated by a DS or DC statement. The letter n represents either an unsigned decimal self-defining term or a positive absolute expression enclosed within parentheses. If any symbols are used in the expression, they must be previously defined. The length attribute of a field used in an assembler instruction determines the number of bytes generated for either that constant or reserved field. The maximum value of the length factor (n) is 65,536. Examples follow:

```
   LABEL    ΔOPERATIONΔ                   OPERAND
   1        10          16

1. WNS      DC          C'LENGTH ATTRIBUTE'
2. YAR      DS          CL16
3. WDR      DS          2CL16
4. SGAW     DC          CL16'LENGTH'
5. STOR     DC          CL16
```

Examples 1 through 5 all have a length attribute of 16 bytes. The length factor is not required in example 1 because the constant specified is 16 bytes long. The length attribute of the receiving field in examples 2 through 5 is a vital element of the instruction. (See 12.18.)

When used, the length factor must follow the character L. The maximum and minimum values that may be explicitly specified are shown in Table 5—1 for all definition types. Constants that do not agree with the specified length are padded or truncated to the left or right, as shown in Table 5—1.

NOTE:

Boundary alignment is not provided when a length factor is specified.

5.1.6. Constant Specification

The constant specification determines the constant, or storage to be generated. When an apostrophe or ampersand is included in the constant specification, double apostrophes or ampersands are used to indicate the inclusion of these characters in the constant.

Examples:

```
1. YAR      DC          C'ENTER NUMBER ''4N'' HERE'
2. LG591    DC          C'ENTER THE NUMBER 51&&91'
```

This will produce 22 bytes as follows:

1. ENTER NUMBER '4N' HERE

2. ENTER THE NUMBER 51&91

The constant may take the form of data or an address, as shown in Table 5—1.

Data Constant	Address Constant
'JUNE 15'	(AREA1)

5.1.7. Alignment

Machine instructions are aligned on half-word boundaries; constants may be aligned on a half word, full word, double word, or no boundary. (See Table 5—1.) When a length factor is specified in the DC or DS statement, no alignment is provided. A duplication factor of zero does not generate a constant or storage area but, for some types of constants, it forces a boundary alignment when no length is stated. This provides a method for obtaining boundary alignment before generating a constant that is not automatically aligned by the assembler. Bytes skipped to align constants are zero filled; bytes skipped to align storage areas are not.

5.2. DEFINITION TYPES

Data definition types generate absolute values or storage through the assembler interaction. There are 13 types, as shown in Table 5—1 and described in more detail in 5.2.1 through 5.2.12. (Also see 2.1.)

5.2.1. Character Constants (C)

The character C is used to specify character constants and can produce up to 256 bytes for a DC and 65,535 bytes for a DS statemnent. All of the 256 valid card punch combinations can be used, but only 48 or 64 characters are printable, depending on the print set available. When the length factor does not agree with the constant specification, padding or truncating takes place on the right. Padding takes place with blanks. (See 2.1 and 2.4.)

```
   LABEL   ΔOPERATIONΔ                    OPERAND
   1        10        16

1. PADDING  DC        CL1Ø'CONSTANT'
2. TRUNCAT  DC        CL5'CONSTANT'
3. NORMAL   DC        C'CONSTANT'
```

1. Produces: CONSTANTΔΔ(ten bytes)

2. Produces: CONST (five bytes)

3. Produces: CONSTANT (eight bytes)

A pair of ampersands is needed to specify a single ampersand constant. A pair of apostrophes is needed to specify a single apostrophe constant.

5.2.2. Hexadecimal Constants (X)

The character X is used to specify hexadecimal constants and can produce up to 256 bytes for a DC and 65,535 bytes for a DS statement. Each byte contains two hexadecimal digits. When the length factor does not agree with the constant specification, padding or truncating takes place on the left. Padding takes place with hexadecimal zeros. (See 2.1 and 2.3.)

```
LABEL   ΔOPERATIONΔ                          OPERAND
1        10        16

1. PADDING  DC        XL7'C4CED5F3FA'
2. TRUNCAT  DC        XL4'C4CED5F3FA'
3. NORMAL   DC        X'C4CED5F3FA'
```

1. Produces:	0 0	0 0	C 4	C E	D 5	F 3	F A	7 bytes
2. Produces:				C E	D 5	F 3	F A	4 bytes
3. Produces:			C 4	C E	D 5	F 3	F A	5 bytes

5.2.3. Binary Constants (B)

The character B is used to specify binary constants and can produce up to 256 bytes. When the length factor does not agree with the constant specification, padding or truncating takes place on the left. Padding is done with binary zeros. The constant specification consists of only the numerals 0 and 1. (See 2.1 and 2.2.)

```
1. PADDING  DC        BL2'Ø11Ø'
2. TRUNCAT  DC        BL1'1ØØØ111ØØ11Ø'
3. NORMAL   DC        B'111ØØ11Ø'
```

1. Produces:	0000 0000	0000 0110	2 bytes
2. Produces:		1110 0110	1 byte
3. Produces:		1110 0110	1 byte

5.2.4. Packed Decimal Constants (P)

The character P is used to specify signed packed decimal constants. It can produce up to 16 bytes. When the length factor does not agree with the constant specification, padding or truncating takes place on the left. Padding is done with hexadecimal zeros. The decimal constant is written as a signed or unsigned number. If unsigned, the number is assumed to be plus. A positive number is assembled with a hexadecimal C in the four least significant bits; a negative number has a D in this location. The maximum of 16 bytes can contain 31 decimal digits plus the sign. (See 2.1 and 2.4.3.)

```
     LABEL     ΔOPERATIONΔ                          OPERAND
     1         10         16

 1.  PLUS      DC         P'+4543'
 2.  NEG       DC         P'-4543'
 3.  UNSIGNED  DC         P'4543'
 4.  PADDING   DC         PL4'+4543'
 5.  TRUNCAT   DC         PL2'-4543'
 6.  ODDNUN    DC         P'14543'
```

		Bytes	Size
1.	Produces:	0 4 | 5 4 | 3 C	3 bytes
2.	Produces:	0 4 | 5 4 | 3 D	3 bytes
3.	Produces:	0 4 | 5 4 | 3 C	3 bytes
4.	Produces:	0 0 | 0 4 | 5 4 | 3 C	4 bytes
5.	Produces:	5 4 | 3 D	2 bytes
6.	Produces:	1 4 | 5 4 | 3 C	3 bytes

5.2.5. Zoned Decimal Constants (Z)

The character Z is used to specify zoned decimal constants. It can produce up to 16 bytes. When the length factor does not agree with the constant specification, padding or truncating takes place on the left. Padding is done with zoned zeros (F0). A plus or unsigned number is assembled with a C in the zone half of the rightmost byte; a negative number will have a D in this location. (See 2.1.)

```
 1.  PLUS      DC         Z'+4543'
 2.  NEG       DC         Z'-4543'
 3.  UNSIGNED  DC         Z'4543'
 4.  PADDING   DC         ZL5'+4543'
 5.  TRUNCAT   DC         ZL3'-4543'
```

1. Produces: | F 4 | F 5 | F 4 | C 3 | 4 bytes

2. Produces: | F 4 | F 5 | F 4 | D 3 | 4 bytes

3. Produces: | F 4 | F 5 | F 4 | C 3 | 4 bytes

4. Produces: | F 0 | F 4 | F 5 | F 4 | C 3 | 5 bytes

5. Produces: | F 5 | F 4 | D 3 | 3 bytes

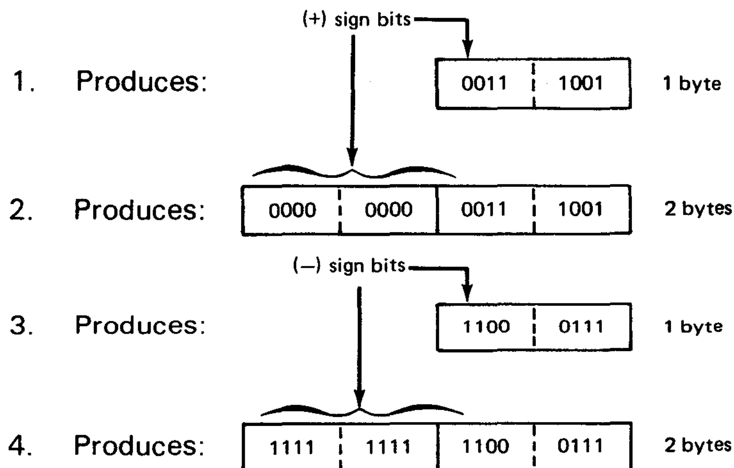
NOTE:

A zoned decimal number is also known as a signed unpacked number. (See 2.4.3.1 and 2.4.3.2.)

5.2.6. Half-Word Fixed-Point Constants (H)

The character H is used to specify half-word fixed-point constants. It can produce up to 8 bytes. If no length factor is specified, the length attribute equals the implied length of 2 bytes. Padding or truncating takes place on the left. Padding is done with the sign of the value, binary 0 for a positive number and binary 1 for a negative number. The constant specification may not contain over five significant decimal digits nor a value greater than +32767 (2^{15}—1) or less than —32768 (—2^{15}). Unsigned values are treated as positive values. The data is aligned on a half-word boundary if no explicit length is specified; otherwise no alignment takes place. (See 2.1, 2.5, and Section 10.)

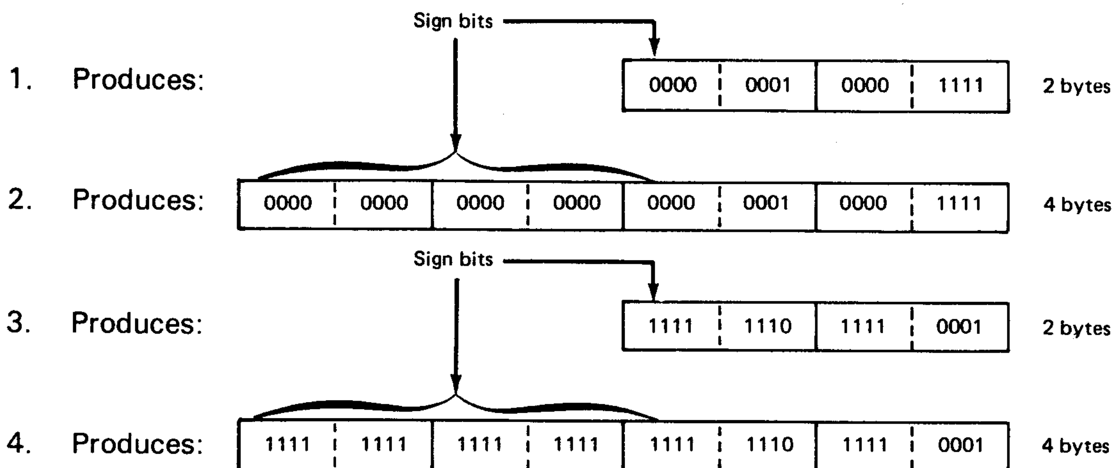
```
   LABEL    ΔOPERATIONΔ                    OPERAND
   1        10         16

1. PLUS1    DC         HL1'+57'
2. PLUS2    DC         H'57'
3. NEG1     DC         HL1'-57'
4. NEG2     DC         H'-57'
```



5.2.7. Full-Word Fixed-Point Constants (F)

The character F is used to specify full-word fixed-point constants. It can produce up to eight bytes. If no length factor is specified, the length attribute equals the implied length of four bytes. Padding or truncating takes place on the left. Padding is done with the sign of the value, binary 0 for a positive number and binary 1 for a negative number. The constant specification may not contain over 10 significant decimal digits nor a value greater than +2,147,483,647 ($2^{31}-1$) or less than −2,147,483,648 (-2^{31}). Unsigned values are treated as positive values. The data is aligned on a full-word boundary if no explicit length is specified; otherwise no alignment takes place. (See 2.1, 2.5, and Section 10.)

```
   LABEL    ΔOPERATIONΔ                     OPERAND
   1        10         16

1. PLUS2    DC         FL2'+271'
2. PLUS4    DC         F'271'
3. NEG2     DC         FL2'-271'
4. NEG4     DC         F'-271'
```

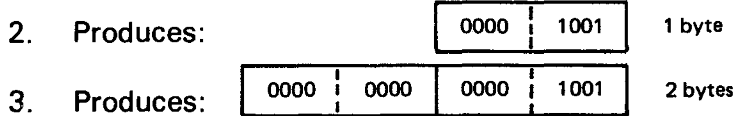


5.2.8. Half-Word Address Constants (Y)

The character Y is used to specify half-word address constants. It can produce up to two bytes. If no length factor is specified, the length attribute equals the implied length of two bytes. Padding or truncating takes place on the left. Padding is done with binary zeros. A length factor of one byte may be specified for absolute expressions only. The expression may be a positive or negative absolute value or a relocatable symbol representing the address of an instruction or item of data within the program. Alignment is on a half-word boundary if no explicit length is stated; otherwise no alignment takes place. The maximum value that can be specified is $2^{15}-1$ (32,767).

```
1. WRD      DC         Y(EQUITY)
2. WBE      DC         YL1(9)
3. WMC      DC         Y(9)
```

1. Produces a 2-byte area containing the address of the instruction EQUITY.
2. Produces:
3. Produces:

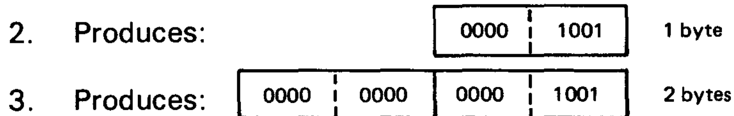


5.2.9. Full-Word Address Constants (A)

The character A is used to specify full-word address constants. It can produce up to four bytes. If no length factor is specified, the length attribute equals the implied length of four bytes. Padding or truncating takes place on the left. Padding is done with binary zeros. Length factors of one or two bytes may be specified for positive or negative absolute values. The maximum value that can be specified is $2^{31}—1$ (2,147,483,647). Alignment is on a full-word boundary if no explicit length is specified; otherwise no alignment takes place.

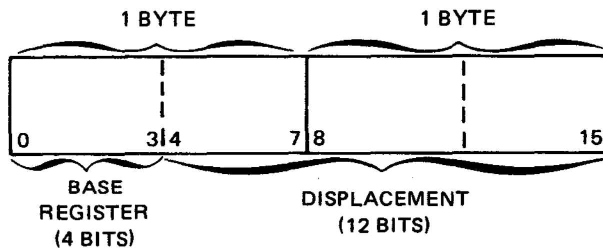
```
    LABEL   ΔOPERATIONΔ                       OPERAND
    1       10          16

1.  WRD     DC          A(VALLEY)
2.  WBE     DC          AL1(9)
3.  WMC     DC          AL2(9)
```

1. Produces a 4-byte area containing the address of the instruction VALLEY.
2. Produces:
3. Produces:



5.2.10. Base and Displacement Constants (S)

The character S is used to specify base and displacement constants. It can produce a 2-byte area. The only length factor that may be specified is 2. No padding or truncating can take place. Alignment is on a half-word boundary when the length factor is not used. Neither negative values nor literals may be used. This instruction produces a 2-byte area, as follows.



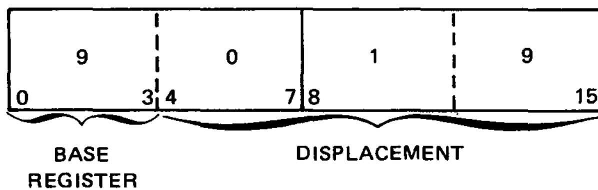
The first four bits (half byte) contain the number of the base register used in this constant. The next 12 bits contain the value of the displacement to be added to the value in the register to produce the full address of the constant.

In the following example, in line 1, the value 5000 will be placed in register number 9 at execution time. (See 19.2 for the USING directive.) In line 2, assume the program has produced the address of 5025 to be assigned to the instruction called ELI, and this instruction is 25 bytes away from the area covered by register number 9. The instruction CHAYA, line 3, specifies the address of ELI, which is register number 9 (value 5000) plus a displacement of 25 bytes to give 5025. The instruction REBEW does not use an address symbol but explicitly states the displacement, 25 bytes, and register number 9.

Examples:

```
LABEL    △OPERATION△                    OPERAND
1         10        16

          START
          .
          .
          .
          USING     5ØØØ,9
          .
          .
          .
ELI       DC        C'CONSTANT'
          .
          .
          .
CHAYA     DC        S(ELI)
REBEW     DC        S(25(9))
          .
          .
          .
```

The constants produced in lines 3 and 4 show the hexadecimal values of the base register and the displacement as follows:



5.2.11. External Address Constants (V)

The character V is used to declare references to special external symbols. The constant must be used to reference an executable instruction which is external to the program. The reference symbol need not be identified by an EXTRN statement. (See 19.5.)

The only length factors that may be specified are 3 or 4. If no length factor is used, the length attribute equals the implied length of four bytes and alignment will be on a full-word boundary.

Padding or truncating takes place on the left. Padding is done with hexadecimal zeros.

The specification of a symbol in the operand field of a type V constant does not constitute a definition of the symbol.

Until the program containing the external symbol is linked to the program with the V type constant, the value of the assembled constant is composed of hexadecimal zeros.

Example:

```
LABEL   ΔOPERATIONΔ                        OPERAND
1        10       16
-----------------------------------------------------
LODGE    DC       V(TRAVEL)
```

As the address value of this DC instruction (TRAVEL) is externally defined, the following constant is generated.

0	0	0	0	0	0	0	0

5.2.12. Floating-Point Constants (E and D)

The format of floating-point constants differs from the standard format of the DC statement (5.1) in that an additional subfield (the scale modifier) may appear. The format for floating-point constants is as follows:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	DC	[d] t [L_n][S+n] 'c[E±n] '

where:

d
Is the duplication factor.

t
Is the definition type (E, full word; D, double word).

L_n
Is the explicit length factor in decimal.

S+n
Is the scale modifier.

'c[E±n]'
Is the constant specification with optional exponent.

The subfields must be written in the order given. The d, t, and L_n subfields are discussed in 5.1. The scale modifier must be a positive signed or unsigned decimal number. If the sign is omitted, a positive value is assumed. The scale modifier is applied to a number after it has been converted to internal format.

Two types of floating-point constants are available: full word (E) and double word (D). The implied length of an E type constant is four bytes; if the length modifier is omitted, full-word boundary alignment is assigned. The implied length of a D type constant is eight bytes; if the length modifier is omitted, double-word boundary alignment is assigned. In either case, an explicit length modifier of from one to eight bytes may be specified.

A floating-point number is written as a decimal number. It can be an integer (110), a fraction (0.75), or a mixed number (110.75). The floating-point number may be followed by an optional exponent represented by an E, a sign, and a decimal number, respectively. In the absence of a sign, a plus sign is assumed.

The exponent for a constant is that power of 10 by which that constant will be multiplied before its conversion to internal format. This exponent value may range from —85 to + 75.

The machine representation of the constant consists of a hexadecimal fraction (mantissa) and a hexadecimal exponent (characteristic). The decimal point is assumed to be at the left of the leftmost digit of the fraction. The characteristic represents the power of 16 by which the fraction must be multiplied to obtain the value of the constant. The machine format is shown in Figure 5—1.

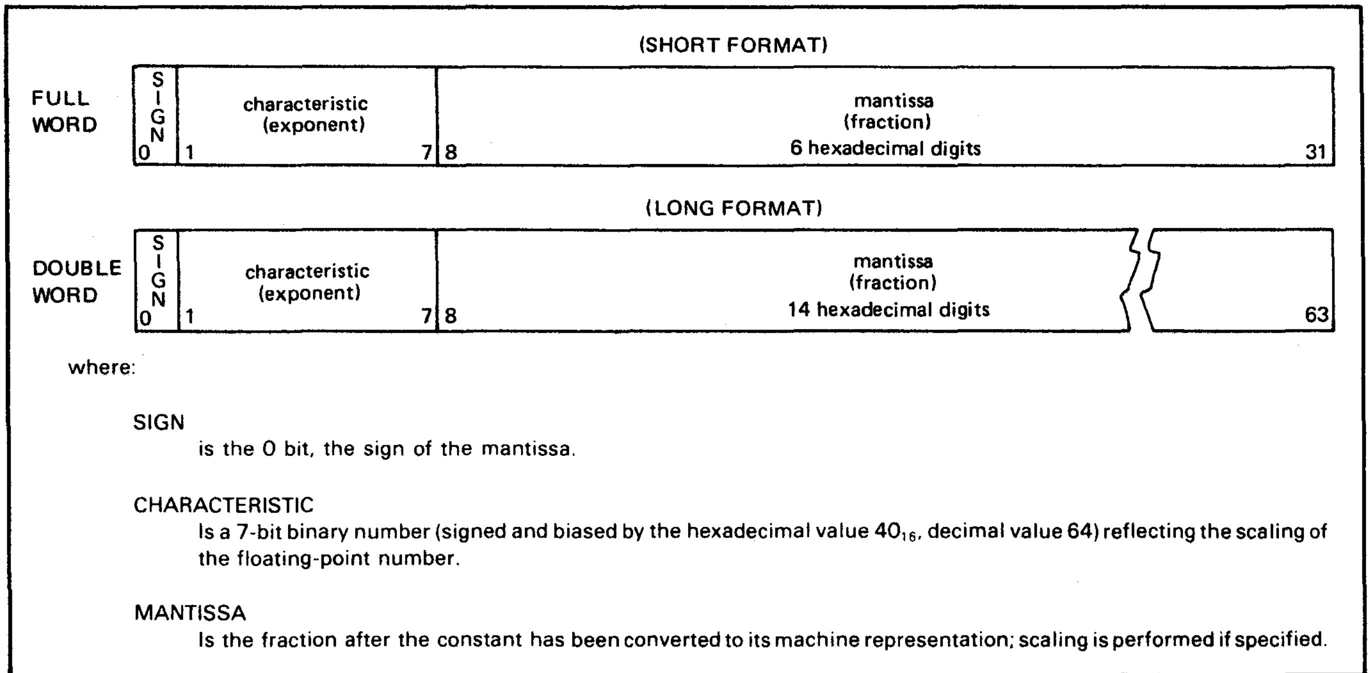


Figure 5—1. Floating-Point Number Formats

Example:

The floating-point value is the product of the mantissa (fraction) and the base 16 raised to the power of the biased characteristic (exponent) after the exponent has been reduced by 64. The decimal number 255 will generate the floating-point number 42FF0000.

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	DC	E'255'

Decimal 255 = the fraction X16. The floating-point number shown in hexadecimal form is 42FF0000.

In this example:

n = hexadecimal 42 (decimal value 66) fraction = .FF0000 (decimal value .9961 from Table C—2). Therefore, 42FF0000 equals:

$.9961 \times 16^{66-64}$
or $.9961 \times 16_2$
or $.9961 \times 256$
or 255

If scaling is not specified, the fraction is hexadecimally normalized; that is, all leading hexadecimal zeros are removed, and the characteristic is adjusted by 1 for each zero removed. Rounding is then performed, and the number is assembled into the field as specified by the explicit or implicit length. Negative fractions are carried, not in twos complement, but in true representation.

The scale modifier must be a positive value from 0 to 14. This modifier specifies the number of hexadecimal positions (four bits) the number is shifted to the right. Scaling provides an unnormalized floating-point number. The characteristic is adjusted to reflect the number of hexadecimal positions the number has been shifted. If any hexadecimal positions are lost, rounding occurs in the rightmost hexadecimal position retained.

Examples:

Normalized number, value 255.	4 ¦ 2	F ¦ F	0 ¦ 0	0 ¦ 0
Unnormalized number, value 255.	4 ¦ 3	0 ¦ F	F ¦ 0	0 ¦ 0
Unnormalized number, value 255.	4 ¦ 4	0 ¦ 0	F ¦ F	0 ¦ 0

See 2.6, Section 11, and Appendix C.

5.3. LITERALS

A literal is a representation of data within a source code statement and can be coded in the sending field of either operand. A literal is simply a constant coded with an equal sign followed by a type code and a nominal value enclosed within single quotation marks.

The method of describing and specifying a constant as a literal is almost identical to the method of specifying it in the operand field of a DC statement. When a literal is assembled, the data is stored in a "literal pool" which is a special area in main storage where all literals are placed. The address of that storage field in the literal pool is then placed in the operand field of the assembled statement.

If two identical literals occur within one literal pool, only the first literal is stored.

The permissible use of literals are:

- Any type of data can be used to specify a literal.

- Only one reference to the same literal in a coding statement can be made.

- A literal is always in the sending field of an operand.

- Literals are relocatable because the address (not the literal itself) is assembled in the coding statement.
- Literals can be self-defining terms which are recognized by the absence of the equal sign, also referred to as immediates.
- Duplication factors can be used in the specification of literals and are expressed only by unsigned decimal values except zero.
- Length attributes can be used in the specification of literals and are expressed only by unsigned decimal values.

The nonpermissible use of literals are:

- A literal can never be used in the receiving field of an operand.
- A literal cannot be combined with other terms.
- It cannot be specified within the parenthesis of an address constant.
- It cannot be specified in a shift instruction, or an I/O instruction.
- A literal cannot have an explicit base or an explicit index.
- Absolute (with all terms previously defined), relocatable, or complex relocatable expressions cannot be used as either duplication factors or length attributes.

Example:

```
 LOC.  OBJECT  CODE     ADDR1  ADDR2  LINE    SOURCE STATEMENT
000000                                   1 STC       START 0
000000 0530                              2 BEGIN     BALR  3,0
000002                                   3           USING *,3
000002 58B0 3016                 00018   4           L     11,AMTIN
000006 5AB0 301E                 00020   5           A     11,=F'5'
00000A 42B0 301A                 0001C   6           STC   11,STOR
                                         7           DUMP
00000E                                A  8+          DS    0H
                                      A  9+*
                                      A 10+*   THE DUMP PARAMETER IS A 1-4 BYTE HEX CODE
                                      A 11+*
00000E 1700                           A 12+          XR    0,0 CLEAR DUMP CODE
000010 0A1B                           A 13*          SVC   27 DUMP SVC
000014                                  14           DS    F
000018 000008BC                         15 AMTIN     DC    F'2236'
00001C F0404040                         16 STOR      DC    CL4'0'
000000                                  17           END   BEGIN
000020 00000005                         18                 =F'5'
```

On line 5 of the sample program, a literal is used in the sending field of operand 2. The equal sign is used followed by the type code (which in this case is F, full word) and the nominal value enclosed in single quotation marks. Note that the object code produced when the literal is assembled is the address (00020) of the field in the literal pool where that literal was placed. Line 18, under object code, shows the literal actually generated.

6. Symbol Definitions

Byte locations in main storage are numbered consecutively starting with zero. Each number is considered the address of the byte of data stored at that location. A group of consecutive bytes is addressed by the leftmost byte. A symbol appearing in the label field of a statement defining an instruction, constant, or storage area is assigned the address value of the first byte of the source statement with which the symbol is associated. The following rules apply to the general use of symbols.

- Must start in column 1
- Must start with an alphabetic character or special letter
- Must consist of only alphabetic characters, numeric characters, and special letters
- Must not be longer than eight characters
- Must not include a space (blank) or other special character
- Must be followed by a blank

Example of valid label field symbols:

```
LABEL    △OPERATION△                    OPERAND
1         10       16
------------------------------------------------------------
W         DC       P'4Ø69'
N4543     DS       PL4
DNOMYARD  DC       C'5ØØ'
CASH$OUT  BALR     R5,Ø
```

Examples of invalid symbols:

```
    LABEL   ΔOPERATIONΔ                OPERAND
    1        10       16

1.  EQUITY  DC        P'4Ø2'
2. 4543     DS        ZL4
3. READ ONE PACK      OPER1,OPER2
4. CONSISTORY DC      C'8Ø'
```

1. Invalid because symbol does not start in column 1
2. Invalid because symbol does not start with an alphabetic character or special letter
3. Invalid because symbol contains a special character (space)
4. Invalid because symbol is longer than eight characters

6.1. EQUIVALENT SYMBOLS

To make a program more meaningful, the programmer may use more than one symbol to represent the same value or location. The same output area could be called NURECORD (in one place) and OUTPUT1 in another part of the program. The EQU directive (Section 16) can be used, as shown in this section, to equate these symbols. An EQU may be used to equate any symbol to any other symbol or to a value. Only the operands may declare expressions.

```
NURECORD EQU      OUTPUT1
RØ       EQU      Ø
R1       EQU      1
R2       EQU      2
R3       EQU      3
 .                .
 .                .
 .                .
R12      EQU      12
R13      EQU      13
R14      EQU      14
R15      EQU      15
```

After the EQU directive, a register instruction could be written in any of the following ways:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10        16
----------------------------------------------------------------
ADD      AR        1,2
ADD      AR        R1,R2
LOAD     LA        5,2(5)
LOAD     LA        R5,2(R5)
MOVE     MVC       WKAREA,REC
MOVE     MVC       15(7,9),5(1Ø)
MOVE     MVC       15(7,R9),5(R1Ø)
```

NOTE:

Throughout, the register notations (R0 through R15) represent the registers 0 through 15.

6.2. SYMBOL APPLICATIONS

Symbols are used not only to identify storage areas and constants but also to locate instructions in the program. In the following example, the branch instruction with the symbol RETURN calls for a branch to the location CONSIS32 under certain conditions. The branch instruction called TRAVEL branches around the nonexecutable DC constants to the location SQUARE.

```
         TITLE     'SYMBOL USE'
         START
BEGIN    BALR      R3,RØ
         USING     *,R3
         .
         .
         .
CONSIS32 MVC       VALLEY,CONSISTO
         .
         .
         .
TRAVEL   B         SQUARE
WMC      DC        C'365'
WRD32    DC        C'32'
         .
         .
         .
SQUARE   LR        R2,R4
         .
         .
         .
```

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10        16

RETURN   BC        7,CONSIS32
         .
         .
         .
MASTER   CLC       WEEKS52,=P'52'
         .
         .
         .
         EOJ
RO       EQU       Ø
R2       EQU       2
R3       EQU       3
R4       EQU       4
REBEW    DC        P'+52'
WEEKS52  EQU       REBEW
VALLEY   DS        CL32
CONSISTO DS        XL32
         END       BEGIN
```

The EQU directives show how to use the symbol WEEKS52 for REBEW; also R0, R2, R3, and R4 for registers 0, 2, 3, and 4.

Through the extensive use of symbols and the assembly listing cross-reference, you can trace every use of a data area or instruction routine. (See 28.5.)

PART 3. BAL APPLICATION INSTRUCTIONS

7. Introduction to Application Instructions

7.1. INSTRUCTION AND FORMAT CONVENTIONS

Certain conventions have been adopted in this manual for specifying instruction, directive, macro, proc, and control-statement formats. The following describe these conventions.

There are eight types of assembler application instructions:

RR — Register-to-register
RX — Register-to-storage-indexed or storage-to-register-indexed
RS — Register-to-storage-nonindexed or storage-to-register-nonindexed
SI — Storage immediate
SS — Storage-to-storage (Type SS1)
SS — Storage-to-storage (Type SS2)
S — Storage
SM — Storage mask

Figure 7—1 illustrates the source code and object code formats for each of these instruction types. (Also see Section 4.)

Instruction Type	Source Code Instruction Format: Explicit Form	Source Code Instruction Format: Implicit Form	Object Code: First Half Word, Byte 1 (bits 0–7)	Byte 2 (bits 8–11)	Byte 2 (bits 12–15)	Second Half Word, Bytes 3 and 4 (bits 16–19)	(bits 20–31)	Third Half Word, Bytes 5 and 6 (bits 32–35)	(bits 36–47)
RR	[symbol] opcode r_1,r_2 (1)	[symbol] opcode r_1,r_2	opcode	r_1 (reg op 1)	r_2 (reg op 2)				
RX	[symbol] opcode $r_1,d_2(x_2,b_2)$ (2)	[symbol] opcode $r_1,s_2(x_2)$	opcode	r_1 (reg op 1)	x_2	b_2	d_2 (address operand 2: x_2, b_2, d_2)		
RS	[symbol] opcode $r_1,r_3,d_2(b_2)$ (3)	[symbol] opcode r_1,r_3,s_2	opcode	r_1 (reg op 1)	r_3 (reg op 3)	b_2	d_2 (address operand 2)		
SI	[symbol] opcode $d_1(b_1),i_2$	[symbol] opcode s_1,i_2	opcode	i_2 (immediate operand)		b_1	d_1 (address operand 1)		
S	[symbol] opcode $d_2(b_2)$	[symbol] opcode s_2	opcode			b_2	d_2 (address operand 2)		
SS	[symbol] opcode $d_1(l,b_1),d_2(b_2)$	[symbol] opcode $s_1(l),s_2$	opcode	$l-1$ (length op 1 and op 2)		b_1	d_1 (address operand 1)	b_2	d_2 (address operand 2)
SS	[symbol] opcode $d_1(l_1,b_1),d_2(l_2,b_2)$	[symbol] opcode $s_1(l_1),s_2(l_2)$	opcode	l_1-1 (length op 1)	l_2-1 (length op 2)	b_1	d_1 (address operand 1)	b_2	d_2 (address operand 2)
SM	[symbol] opcode $d_1(b_1),i_2,m_3,d_4$	[symbol] opcode s_1,i_2,m_3,d_4	opcode	i_2 (immediate operand 2)		m_3 (immediate mask 3)	d_4 (displacement 4)	b_1	d_1 (address operand 1)

NOTES:

(1) The RR instruction has three other forms:

[symbol] opcode i_1 for the SVC instruction;

[symbol] opcode r_1 for the SPM instruction; and

[symbol] opcode m_1,r_2 for the BCR instruction.

(2) The RX instruction BC is written in the form:

[symbol] opcode m_1, d_2 (x_2, b_2).

(3) The RS instruction has two other forms:

the RS shift instructions are written without use of the r_3 operand, in the form:

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

some RS instructions such as ICM and CLM are written in the form:

[symbol] opcode r_1, m_3, d_2 (b_2).

Figure 7—1. Instruction Formats (Part 1 of 2)

Characters	Meaning
OPCODE	The application instruction operation code.
r_1	The number of the general register containing operand 1
r_2	The number of the general register containing operand 2
r_3	The number of the general register containing operand 3
x_2	The number of the general register containing an index number for operand 2 of the RX instruction
i_1	The immediate data used as operand 1 of the SVC instruction
i_2	The immediate data used as operand 2 of an SI instruction
l	The length of the operands as stated in source code*
l_1	The length of operand 1 as stated in source code*
l_2	The length of operand 2 as stated in source code*
b_1	The number of the general register containing the base address for operand 1
b_2	The number of the general register containing the base address for operand 2
d_1	The displacement for the base address of operand 1
d_2	The displacement for the base address of operand 2
d_4	The displacement used as operand 4 of an SM instruction
m_1	The mask used as operand 1
m_3	The mask used as operand 3 of an SM instruction
op_1	Operand 1
op_2	Operand 2
op_3	Operand 3
s_1	The symbol used to identify operand 1 in the implicit format
s_2	The symbol used to identify operand 2 in the implicit format

*In source code, the length you specify is 1 greater than the object code length. The reason for this is that 0 is the first length count, not 1. For example, I can address a maximum length of 256, but in actuality, I get 0 through 255 bytes. The assembler makes a reduction of 1 in the length when converting source code to object code.

Figure 7—1. Instruction Formats (Part 2 of 2)

An instruction is an executable statement for operations involving data. The assembler instructions are two, four, or six bytes in length. (See Figure 7—1.) In a 2-byte (RR) instruction, the registers are referenced for both operands. A 4-byte (RS) instruction references a register for the first operand and main storage for the second operand. A 4-byte (RX) instruction references registers for the first and second operands and main storage for the third operand. A 4-byte immediate operand (SI) instruction references main storage for the first operand and immediate data for the second operand. A 6-byte (SS) instruction references main storage for both operands.

NOTE:

All instructions are aligned by the assembler on a half-word boundary.

The implied length field may be applicable with the SS1 and SS2 type instructions. If no length is specified in an SS1 type instruction, the length attribute of the first operand is assembled into the length field of the instruction. The length attribute of an operand is the length attribute of the expression used to define the storage location. The SS2 type instruction contains a length field for each operand; however, neither, either, or both length fields may be implied. In every case, the assembler puts the operand lengths, implied or specified, into the length fields.

The following are examples of implied and explicitly stated lengths.

```
     LABEL    ΔOPERATIONΔ                          OPERAND
     1        10          16

     NUMBER12 DC          ZL12'+Ø'
     NUMBER7  DC          Z'1234567'
  1. PAD      PACK        NUMBER12,NUMBER7
  2. FILLUP   PACK        NUMBER12(4),NUMBER7
  3. TRUNCATE PACK        NUMBER12(4),NUMBER7(3)
```

Instruction 1 (PAD) packs all seven digits and the sign of operand 2 (NUMBER7) into four bytes of operand 1 (NUMBER12), then zero fills the remaining eight bytes of the implied field of 12 bytes. Instruction 2 (FILLUP) packs all seven digits and the sign of operand 2 into the explicit four bytes of operand 1. Instruction 3 (TRUNCATE) packs only the explicitly stated three digits and the sign of operand 2 into the explicit four bytes of operand 1. Labeled instructions themselves are assigned implied lengths based on instruction type.

There are six basic ways to explain how an assembler application instruction is written: the implicit format, the implicit source code example, the explicit format, the explicit source code example, the object code format, and the object code example. The first four methods are shown for each instruction in this part of the user guide, as the subject of object code formats covered in 4.3.1, and are discussed again in assembly listings (Part 6). The following shows how the *move character* instruction is written.

- Implicit source code:

  Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVC	s_1(l),s_2

  Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1         10            16
------------------------------------------------------------------------
MOVE5     MVC           LODGE,MASTER
MOVE32    MVC           LODGE(32),MASTER
```

- Explicit source code:

  Format:

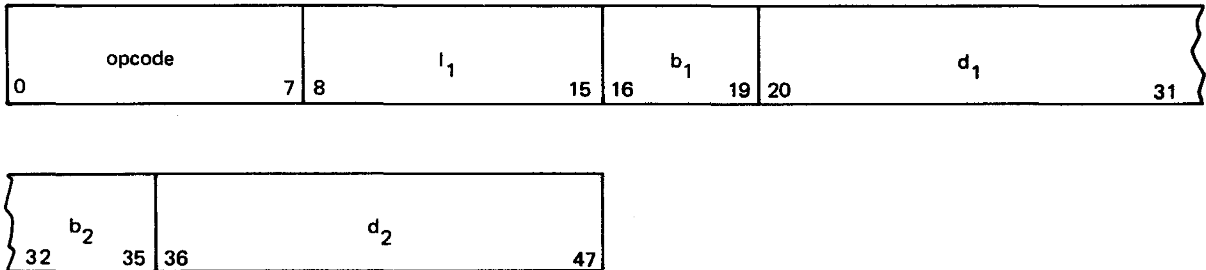
LABEL	△OPERATION△	OPERAND
[symbol]	MVC	d_1(l,b_1),d_2(b_2)

  Example:

```
MOVE32    MVC           32(5,R2),3(R3)
```

- Object code:

  Format:



  Example as shown on an assembly listing:

```
000DF5 D2 1F 20F1 30FC
```

7.2. EXPLICIT FORMS

The first line is how the explicit format is expressed, and the second line is an example of how you might write the explicit source code form of the *add* instruction.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	A	$r_1,d_2(x_2,b_2)$
ADDREC	A	R2,32(R3,R5)

7.3. IMPLICIT FORMS

The first line is how the implicit format is expressed, with the following one or more lines being examples of how you might write the implicit source code forms of the *add* instruction using symbols to represent registers and data areas.

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	A	$r_1,s_2(x_2)$
ADDFOR	A	R2, PAYSUM
ADDREG	A	R2, PAYSUM(R3)

7.4. DEFINITIONS OF FORMAT TERMS

Figure 7—1 explains all the terms used in describing the explicit and implicit forms of the instructions. The following additional explanations will help you to understand the implicit and explicit forms of programming coding.

- The general registers r_1, r_2, or r_3 are shown in the R0 through R15 form.

- The index register x_2 and the base registers b_1 and b_2 are also shown in the R0 through R15 form.

- The terms s_1 and s_2 represent the use of a symbol (4.2.1 and Section 6) in the first or second operand.

- The displacement d_1 or d_2 is a decimal value which is combined with the value in some base register.

- A checkoff table is included for each applicable instruction in the text. Explanations of the program exceptions are provided in Appendix D.

8. Branching Instructions

8.1. USE OF BRANCHING INSTRUCTIONS

Branching instructions are used to alter the normally sequential execution of instructions by branching out of sequence to link to a subroutine, make a decision, or control looping. The operand 2 field of each branching instruction refers to the address (branch to) of the instruction to be executed immediately after the branching instruction. The branch-to address in operand 2 is stored in bits 40—63 of the current program status word (PSW) (Figure 8—1). The PSW is a double word containing the address of the next instruction and various other control fields. In general, the PSW is used to control instruction sequencing and to hold and indicate the status of the system in relation to the program currently being executed. (See the processor programmer reference for a complete description of the PSW.)

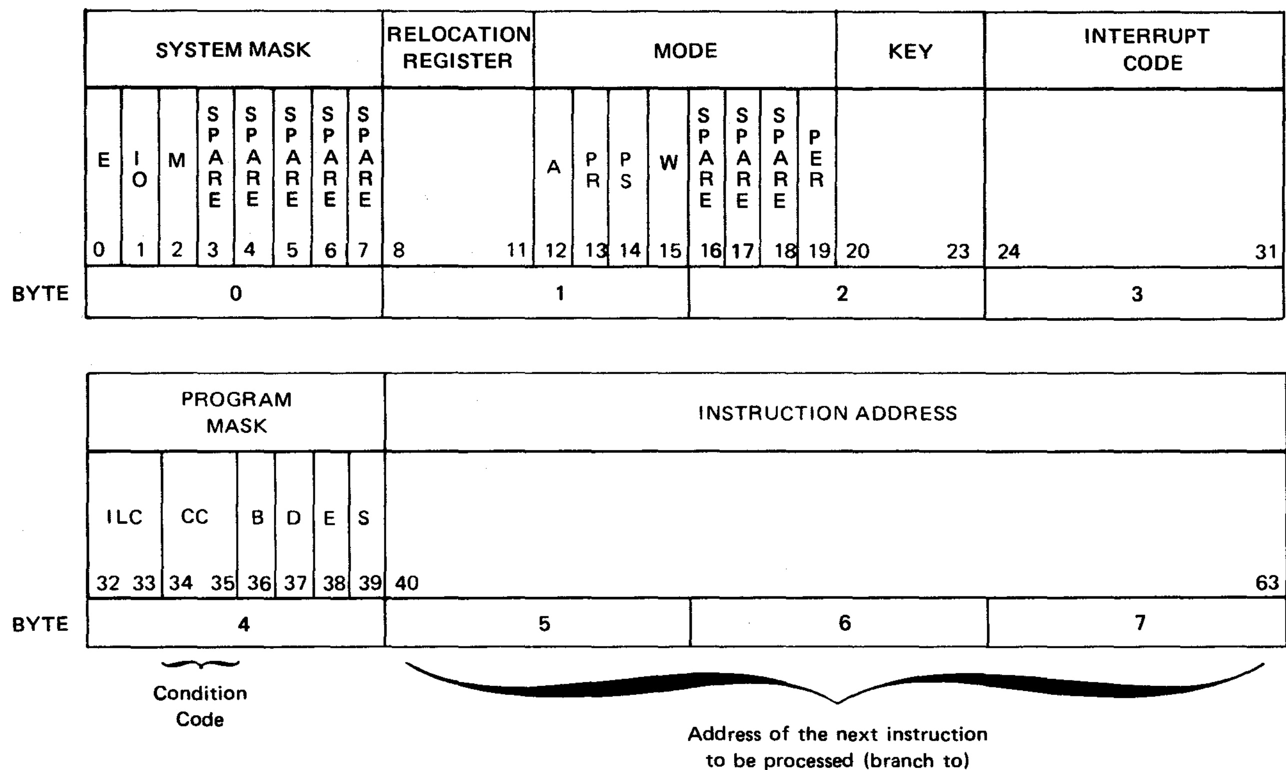


Figure 8—1. Program Status Word Diagram

While the program is executing, it utilizes the PSW (called the current PSW), which is stored in the supervisor portion of the operating system. Before a branch out of the sequence of the program to a new location, the present location of the program is stored in the PSW. That PSW (called the old PSW) is saved, and the program uses a new PSW (current) to keep track of pertinent program information. In other words, the old PSW holds the place in the program if you want to return to where you were before branching to a routine or instruction in another area, and the current PSW keeps track of the running program regardless of where you branched.

For an explanation of the checkoff table exceptions, see Appendix D.

8.2. EXTENDED MNEMONIC CODES

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
See Table 8–1.		RX & RR	2 or 4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The extended mnemonic codes are used like the *branch on condition* (8.4) instruction. Extended mnemonics are the shorthand version of the *branch on condition* (BC) instruction. They are easy to use because you do not need to remember the decimal value that is associated with the operand 1 mask of the branch on condition instruction. You merely remember the mnemonic. The extended mnemonics create the mask value, which tests the condition code. If the specific condition or conditions you want to branch on exist, a branch is made to the address specified in operand 2.

Before a branch is made, the address of the running program is saved, thus enabling you to return to that point if desired. It is a good idea to familiarize yourself with the branch on condition instruction and Table 8—1 before you use the extended mnemonics.

Table 8—1 is a listing of the extended mnemonic codes. The hexadecimal operation codes (with mask) and functions, categorized by instruction type, are grouped according to use. Also included are the *branch on condition* (BC) instruction equivalents. An example of a handwritten coding form follows.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1        10        16

1.                CP        A,B
2.                BE        EQUAL
3.                BL        LOW
                  BH        HIGH
                  .
                  .
                  .
LOW               AP        A,B
                  .
                  .
                  .
A                 DC        P'4'
B                 DC        P'5'
```

1. The *compare decimal* (CP) instruction compares the packed decimal contents of A against the packed decimal contents of B. Based on the results, the condition code in the PSW is set to 1 (operand 1 < operand 2).

2. The next sequential instruction executed is *branch if equal* (BE). The mask (8) tests for condition code 0 (operand 1 = operand 2). Since the condition is not 0, no branch is made, and the next sequential instruction is executed.

3. Since no branch was made, the next sequential instruction executed is *branch if low* (BL). The mask (4) tests for condition code 1 (operand 1 < operand 2). Since the condition code is 1, a branch is made to the operand 2 address (LOW). In this example, LOW is the address of an *add decimal* instruction, which is the instruction executed after the BL instruction.

   Note that the next sequential instruction is *branch if high* (BH), but it is not executed after the BL instruction.

Table 8—1. Extended Mnemonics and Functions (Part 1 of 2)

RR-Type Instructions		RX-Type Instructions		BC Equivalent		Function
Mnemonic Code	Hexadecimal Operation Code/m_1	Mnemonic Code	Hexadecimal Operation Code/m_1	Explicit Form		
Used to Branch Around Nonexecutable Assembler Instructions and Directives						
BR	07 F	–	–	BCR	$15,r_2$	Branch unconditionally
NOPR	07 0	–	–	BCR	$0,r_2$	No operation
–	–	B	47 F	BC	$15,d_2(x_2,b_2)$	Branch unconditionally
–	–	NOP	47 0	BC	$0,d_2(x_2,b_2)$	No operation

Table 8—1. Extended Mnemonics and Functions (Part 2 of 2)

RR-Type Instructions		RX-Type Instructions		BC Equivalent		Function
Mnemonic Code	Hexadecimal Operation Code/m_1	Mnemonic Code	Hexadecimal Operation Code/m_1	Explicit Form		
Used After Comparison Instructions						
BHR	07 2	BH	47 2	BC	$2,d_2(x_2,b_2)$	Branch if high
BLR	07 4	BL	47 4	BC	$4,d_2(x_2,b_2)$	Branch if low
BER	07 8	BE	47 8	BC	$8,d_2(x_2,b_2)$	Branch if equal
BNHR	07 D	BNH	47 D	BC	$13,d_2(x_2,b_2)$	Branch if not high
BNLR	07 B	BNL	47 B	BC	$11,d_2(x_2,b_2)$	Branch if not low
BNER	07 7	BNE	47 7	BC	$7,d_2(x_2,b_2)$	Branch if not equal
Used After Test-Under-Mask Instructions						
BOR	07 1	BO	47 1	BC	$1,d_2(x_2,b_2)$	Branch if all ones
BZR	07 8	BZ	47 8	BC	$8,d_2(x_2,b_2)$	Branch if all zeros
BMR	07 4	BM	47 4	BC	$4,d_2(x_2,b_2)$	Branch if mixed
BNOR	07 E	BNO	47 E	BC	$14,d_2(x_2,b_2)$	Branch if not all ones
BNZR	07 7	BNZ	47 7	BC	$7,d_2(x_2,b_2)$	Branch if not all zeros
BNMR	07 B	BNM	47 B	BC	$11,d_2(x_2,b_2)$	Branch if not mixed
Used After Arithmetic Instructions						
BOR	07 1	BO	47 1	BC	$1,d_2(x_2,b_2)$	Branch if overflow
BZR	07 8	BZ	47 8	BC	$8,d_2(x_2,b_2)$	Branch if zero
BMR	07 4	BM	47 4	BC	$4,d_2(x_2,b_2)$	Branch if minus
BPR	07 2	BP	47 2	BC	$2,d_2(x_2,b_2)$	Branch if positive
BNOR	07 E	BNO	47 E	BC	$14,d_2(x_2,b_2)$	Branch if not overflow
BNZR	07 7	BNZ	47 7	BC	$7,d_2(x_2,b_2)$	Branch if not zero
BNMR	07 B	BNM	47 B	BC	$11,d_2(x_2,b_2)$	Branch if not minus
BNPR	07 D	BNP	47 D	BC	$13,d_2(x_2,b_2)$	Branch if not positive

8.3. BRANCH AND LINK (BAL, BALR)

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
BAL	45	RX	4
BALR	05	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *branch and link* (BAL and BALR) instructions alter the normally sequential execution of instructions by branching to an address you specify in operand 2. The instruction located at that address is the next instruction executed after the *branch and link* instruction. Before the branch is made, the address of the next sequential instruction (current location) is saved in the operand 1 register to enable you to return to the location where you were before branching.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BAL	$r_1, d_2 (x_2, b_2)$
[symbol]	BALR	r_1, r_2

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BAL	$r_1, s_2(x_2)$

Example 1 (BAL):

```
        LABEL    ΔOPERATIONΔ                    OPERAND
        1        10        16

1.               BAL       6,BRANCHTO
2.      NEXTSEQ  AP        A,B
                 .
                 .
                 .
3.      BRANCHTO CP        A,B
                 .
                 .
                 .
        A        DC        P'654'
        B        DC        P'123'
```

1. In this coding example, the BAL instruction alters the sequential execution of instructions by causing a branch to an area in main storage labeled BRANCHTO. Before the branch, the address of the next sequential instruction is stored in register 6 (location of NEXTSEQ).

2. Since a branch took place, the normally sequential execution of the *add decimal* (AP) instruction (NEXTSEQ) is skipped.

3. This *compare decimal* (CP) instruction is processed immediately following the BAL instruction, since its label (BRANCHTO) is the branch-to address (operand 2) of the BAL instruction.

Example 2 (BALR):

```
1.               LA        4,ROUTINE
                 .
                 .
                 .
2.               BALR      6,4
        NEXTSEQ  AP        A,B
                 .
                 .
                 .
        ROUTINE  CP        NUM,=P'15'
                 BL        LOW
                 BH        HIGH
                 .
                 .
                 .
        NUM      DC        P'2Ø'
```

1. This *load address* (LA) instruction puts the address of an instruction labeled ROUTINE (CP) into register 4.

2. The BALR instruction stores the address NEXTSEQ in register 6, then branches to the branch address in register 4. Since register 4 is the address of ROUTINE, the instruction following the BALR instruction is *compare decimal* (CP).

Operational Considerations:

- You may specify any of the general registers (0 through 15) as operand 1 in both the BAL and BALR instructions, and any of these registers as operand 2 of the BALR instruction.

- The address of the instruction following the BAL or BALR instruction is placed in the operand 1 register. Then the branch to the address specified in operand 2 is made.

- If you specify 0 in the operand 2 register of a BALR instruction, it means you are not specifying a branch-to address, and, therefore, no branch takes place. The instruction executed after the BALR is the next instruction in sequence.

8.3.1. Use of the BALR Instruction in Base Register Assignment

The BALR instruction is used in conjunction with the USING directive (see 18.2) to assign a base address to a register. This address becomes the starting address of your program. The BALR instruction and the USING directive must be coded in the following order:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10       16
-------------------------------------------------------------
BASEREG  START     Ø
BEGIN    BALR      3,Ø
         USING     *,3
         OPEN      INFILE,(INRIB)
READ     DMINP     INFILE
```

The BALR instruction in this example stores the address of the next sequential instruction (OPEN) in register 3. No branch takes place since 0 is specified as operand 2. Logically, the address of the USING directive should be stored in register 3, since that is the next instruction. However, USING is a directive and not an executable instruction. Directives are information to the assembler only. They do not generate any object code nor increase the location counter. Therefore, OPEN is the next executable instruction following the BALR instruction.

The USING directive tells the assembler that register 3 is going to be used as the base register for this program. Register 3 can accommodate a program up to 4096 bytes in length. If your program is larger than this, additional base registers can be assigned. (See 18.2.) Operand 1 tells the assembler at what point in your program your base register should start being used. The asterisk (*) means "start now". So, operand 1 indicates starting now, all addresses of the following instructions will use the register specified by operand 2 (in this case, 3) as the base register.

In the following printout example, an LA instruction (line 3) is coded after the BALR instruction, causing an addressability error. The BALR instruction (line 2) stores the address of the next sequential instruction (LA) in register 3. No branch takes place since 0 is specified as the operand 2 register. At location counter 000002, no object code was generated for the LA instruction because the assembler does not assemble an erroneous instruction. It does, however, increase the location counter by the number of bytes that the instruction occupies. Now, register 3 contains the address of the LA instruction at 000002 but the USING directive (line 4) tells the assembler that starting at location counter 000006, all succeeding instructions will use register 3 as the base register. The USING directive assumes that register 3 contains the address at location counter 000006 but in reality contains the address at location counter 000002. So, all the addresses of every instruction and label in this program will be calculated as being four bytes more than its actual location. Therefore, the BALR instruction and USING directive must always refer to the same address so that the base and displacement values can be accurately calculated.

Example:

```
 LOC.  OBJECT CODE    ADDR1 ADDR2  LINE   SOURCE STATEMENT                                  OS/3 ASM 80/01/03
000000                               1 TSTUSING START 0
000000 0530                          2 BEGIN    BALR  3,0
000002 4170 0000          00016      3 LOAD     LA    7,LIST
       *** ERROR ***
000006                               4          USING *,3
                                     5          OPEN  PRINT,(PRINTRIB)
000006 0700                     A    6+         CNOP  0,4                                           P@000960
000008 4510 300E          00014 A    7+         BAL   1,*+12                                        P@001150
00000C 81                       A    8+         DC    X'81'                                         P@001160
00000D 000020                   A    9+         DC    AL3(PRINT)                                    P@001170
000010 80                       A   10+         DC    X'80'                                         P@001180
000011 00004C                   A   11+         DC    AL3(PRINTRIB)                                 P@001190
000014 0A26                     A   12+         SVC   38 ISSUE SVC                                  P@002170
                                                .
                                                .
                                                .
000016                              14 LIST     DS    0CL12
                                                .
                                                .
                                                .
                                    16
                                    25
                                    26 PRINT    CDIB
                                    33 PRINTRIB RIB   IOA1=OUTPUT,BFSZ=120
                                                .
                                                .
                                                .
```

8.4. BRANCH ON CONDITION (BC, BCR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
BC	47	RX	4
BCR	07	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *branch on condition* (BC and BCR) instructions change program sequence based on the condition code setting of the PSW. You specify in the operand 1 mask a decimal value that tests the condition code to see whether the branch-causing condition exists. If the condition of the branch does exist, a branch is made to the operand 2 address you specify in the branch on condition instruction. If the condition of the branch does not exist, no branch takes place, and the next sequential instruction is executed.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BC	$m_1, d_2(x_2, b_2)$
[symbol]	BCR	m_1, r_2

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BC	$m_1, s_2(x_2)$
[symbol]	BCR	m_1, r_2

The condition code, bits 34—35 of the PSW, reflects the result of an instruction executed prior to the branch on condition instruction. There are four possible condition code settings:

If result = 0, set to 0.

If result < 0, set to 1.

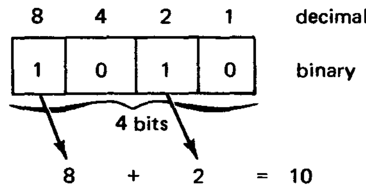
If result > 0, set to 2.

If overflow, set to 3.

The decimal values that can be specified in the operand 1 mask are 0 through 15, each of which has a 4-bit binary representation.

- The decimal value 8 (1000) tests for condition code 0.
- The decimal value 4 (0100) tests for condition code 1.
- The decimal value 2 (0010) tests for condition code 2.
- The decimal value 1 (0001) tests for condition code 3.

Note that only one bit is set for each condition. When more than one bit is set in the 4-bit binary configuration, it is possible to test for multiple conditions.



The decimal value 10 tests for:

1. condition code 0 (result is equal to zero); and
2. condition code 2 (result is greater than zero).

Table 8—2 lists the 16 values and each condition code it tests.

Table 8—2. Operand 1 Mask Combinations

Mask		Possible Combinations of Condition Codes	Branches on Condition Code	Results Causing Branch
Decimal Value	Binary Value			
0	0000	*	no operation	None
1	0001	1 = 1	3	Overflow
2	0010	2 = 2	2	>0
3	0011	3 = 2 + 1	2, 3	>0 or overflow
4	0100	4 = 4	1	<0
5	0101	5 = 4 + 1	1, 3	<0 or overflow
6	0110	6 = 4 + 2	1, 2	≠0
7	0111	7 = 4 + 2 + 1	1, 2, 3	≠0 or overflow
8	1000	8 = 8	0	= 0
9	1001	9 = 8 + 1	0, 3	= 0 or overflow
10	1010	10 = 8 + 2	0, 2	≥0
11	1011	11 = 8 + 2 + 1	0, 2, 3	≥0 or overflow
12	1100	12 = 8 + 4	0, 1	≤0
13	1101	13 = 8 + 4 + 1	0, 1, 3	≤0 or overflow
14	1110	14 = 8 + 4 + 2	0, 1, 2	Any value
15	1111	15 = 8 + 4 + 2 + 1 **	0, 1, 2, 3	Any value or overflow

*No condition code is tested and no branch takes place. The next sequential instruction is executed.

**Unconditional branch

Table 8—3 lists the explicit format of the BC instruction with different mask values and their relation to the condition tested.

Table 8—3. Branch-on-Condition Instruction by Usage

Hexadecimal Operation Code m_1	Mnemonic Code	Explicit Format	Function
Used to Branch Around Nonexecutable Assembler Instructions and Directions			
07 F	BCR	$15,r_2$	Branch unconditionally
07 0	BCR	$0,r_2$	No operation
47 F	BC	$15,d_2(x_2,b_2)$	Branch unconditionally
47 0	BC	$0,d_2(x_2,b_2)$	No operation
Used After Comparison Instructions			
47 2	BC	$2,d_2(x_2,b_2)$	Branch if high
47 4	BC	$4,d_2(x_2,b_2)$	Branch if low
47 8	BC	$8,d_2(x_2,b_2)$	Branch if equal
47 D	BC	$13,d_2(x_2,b_2)$	Branch if not high
47 B	BC	$11,d_2(x_2,b_2)$	Branch if not low
47 7	BC	$7,d_2(x_2,b_2)$	Branch if not equal
Used After Test-Under-Mask Instructions			
47 1	BC	$1,d_2(x_2,b_2)$	Branch if all ones
47 8	BC	$8,d_2(x_2,b_2)$	Branch if all zeros
47 4	BC	$4,d_2(x_2,b_2)$	Branch if mixed
47 E	BC	$14,d_2(x_2,b_2)$	Branch if not all ones
47 7	BC	$7,d_2(x_2,b_2)$	Branch if not all zeros
47 B	BC	$11,d_2(x_2,b_2)$	Branch if not mixed
Used After Arithmetic Instructions			
47 1	BC	$1,d_2(x_2,b_2)$	Branch if overflow
47 8	BC	$8,d_2(x_2,b_2)$	Branch if zero
47 4	BC	$4,d_2(x_2,b_2)$	Branch if minus
47 2	BC	$2,d_2(x_2,b_2)$	Branch if positive
47 E	BC	$14,d_2(x_2,b_2)$	Branch if not overflow
47 7	BC	$7,d_2(x_2,b_2)$	Branch if not zero
47 B	BC	$11,d_2(x_2,b_2)$	Branch if not minus
47 D	BC	$13,d_2(x_2,b_2)$	Branch if not positive

Operational Consideration:

- You can specify any of the general registers (2 through 12) as operand 2 of the BCR instruction.

Example (BC):

```
     LABEL    ΔOPERATIONΔ                        OPERAND
     1        10         16
   ----------------------------------------------------------
1. |          CP         A,B
2. |          BC         2,HIGH
3. |          BC         4,LOW
4. |          BC         8,EQUAL
   |           .
   |           .
   |           .
5. | EQUAL    AP         A,C
   |           .
   |           .
   |           .
   | A        DC         P'65Ø'
   | B        DC         P'65Ø'
   | C        DC         P'1'
   |           .
   |           .
   |           .
```

In this example, two packed decimal values are compared. After the result is obtained, the condition code is stored in the PSW. The branch instructions are executed sequentially, and when the proper condition for the branch-on condition exists, a branch to the operand 2 address takes place.

1. Operands 1 and 2 compare equally since both A and B contained the packed decimal 650. The condition code is set to 0.

2. The operand 1 mask 2 tests for condition code 2. Since the condition code is not 2 (operand 1 > operand 2), the next sequential instruction is processed.

3. The operand 1 mask 4 tests for condition code 1. Since the condition code is not 1 (operand 1 < operand 2), the next sequential instruction is processed.

4. The operand 1 mask 8 tests for condition code 0. Since the condition code is 0 (operand 1 = operand 2), a branch to the address of operand 2 (EQUAL) takes place.

5. Since the *add decimal* instruction has the label EQUAL, that is the instruction executed after the branch-on condition regardless of the sequential instructions in between.

BCT
BCTR

8.5. BRANCH ON COUNT (BCT, BCTR)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
BCT	46	RX	4
BCTR	06	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *branch on count* (BCT and BCTR) instructions branch to the same instruction or routine a number of times (loop). Normally, before you execute a *branch on count* instruction, you load one of the general purpose registers with a value (the count) that refers to the number of times you want to loop to an instruction or routine. You specify the register containing the counter as operand 1 in your branch on count instruction. Each time the branch on count instruction is executed, the operand 1 register is decremented by one. Then the register is checked for a value of zero. If zero is not found, a branch to the instruction at the operand 2 address takes place. If the counter is 0, no additional branching takes place, and the next sequential instruction is executed. You can use the BCTR instruction to decrement the counter register (operand 1) without branching, by specifying the operand 2 register as 0. When BCTR is executed, the value in the operand 1 register is decremented by 1, but since no branch address is supplied, the next sequential instruction is executed.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BCT	$r_1, d_2(x_2, b_2)$
[symbol]	BCTR	r_1, r_2

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BCT	$r_1, s_2(x_2)$
[symbol]	BCTR	r_1, r_2

Example:

```
       LABEL    ΔOPERATIONΔ                    OPERAND
       1        10       16
       
                 .
                 .
                 .
1.               SR       6,6
2.               LA       6,1Ø
3.     BRANCHTO  AP       SUM,AMOUNT
4.               BCT      6,BRANCHTO
       NEXTSEQ   MP       SUM,VALUE
                 .
                 .
                 .
       AMOUNT    DC       P'15Ø'
       SUM       DC       PL3'Ø'
```

This coding example adds AMOUNT (150) to SUM 10 times and stores the result in SUM (3-byte field).

1. The *subtract* (SR) instruction subtracts the operand 2 register value from the operand 1 register value and puts the result in operand 1. In this case, register 6 is subtracted from itself, thus making it 0. It is a good idea to clear a register before using it.

2. The *load address* (LA) instruction loads a count of 10 into register 6.

3. The *add decimal* (AP) instruction adds the packed decimal value in AMOUNT (150) to whatever is stored in SUM and stores the answer in SUM. SUM is defined as a packed decimal constant containing 0. After the AP instruction is executed once, SUM contains 150.

4. The BCT instruction subtracts 1 from register 6. Since register 6 now contains 9, the AP instruction labeled BRANCHTO is executed again. After the AP is executed twice, SUM contains 300. The BCT instruction executes nine more times until the counter (register 6) is 0. On the 10th attempt, no branch takes place, and NEXTSEQ is executed.

Operational Considerations:

- The maximum value you can specify in the operand 1 counter register is 2^{32}.

- You can specify any of the general registers (0 through 15) as operand 1.

- You can specify the operand 2 register of the BCTR instruction as 0 if you want to decrement the operand 1 counter register by 1 without causing a branch. When you specify 0 in operand 2, the next sequential instruction of your program is executed following the BCTR.

- The branch-to address in operand 2 is determined before the operand 1 register is decremented.

8.6. BRANCH ON INDEX HIGH (BXH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
BXH	86	RS	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *branch on index high* (BXH) instruction algebraically compares the sum of the operand 1 register and operand 3 register to either the operand 3 register or one greater than the operand 3 register (comparand register). If the sum is greater than the content of the comparand register, a branch to the instruction located at the operand 2 address takes place. If a greater than condition does not exist, your program continues processing with the instruction following the BXH instruction. The sum is always placed in the operand 1 register after the comparison.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BXH	$r_1,r_3,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BXH	r_1,r_3,s_2

This instruction algebraically adds the content of the operand 1 register to the content of the operand 3 register. The sum is algebraically compared to the content of an odd-numbered register (which can be the same as the operand 3 register) or a register that is one larger than the operand 3 register. If the sum is greater than the content of the odd-numbered register it is being compared to, a branch to the instruction located at the operand 2 address takes place. If the sum is less than or equal to the content of the odd-numbered register it is being compared to, the program continues processing with the instruction following the BXH instruction. Following the comparison, the sum is placed in operand 1. Usually, the BXH instruction is executed several times (depending on program logic) until the content of the operand 1 register is greater than the odd-numbered register it is being compared to. Then the branch to the instruction located at the operand 2 address takes place.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 3.
- Any odd-numbered register either equal to operand 3 or one greater than operand 3 can be used as the comparand register.
- Operand 2 can be any location in main storage.
- The rules of algebra apply to both the addition and the comparison operations.
- The condition code remains unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10      16
--------------------------------------------------------
         LA      3,4
         LA      4,1Ø
         LA      5,11
         BXH     3,4,LOOP
         AP      CARDIN,=P'5ØØ'
         .
         .
         .
LOOP     CP      CARDIN,MAXIMUM
         .
         .
         .
CARDIN   DC      PL3'Ø'
MAXIMUM  DC      PL3'1ØØØØ'
```

Registers 3 and 4 before execution of BXH instruction:

0000	0000	0000	0000	0000	0000	0000	0100	binary
0	0	0	0	0	0	0	4	hex

0000	0000	0000	0000	0000	0000	0000	1010	binary
0	0	0	0	0	0	0	A	hex

Register 5 (comparand register) before and after execution of BXH instruction:

0000	0000	0000	0000	0000	0000	0000	1011	binary
0	0	0	0	0	0	0	B	hex

Registers 3 and 4 after execution of BXH instruction:

0000	0000	0000	0000	0000	0000	0000	1110	binary
0	0	0	0	0	0	0	E	hex

0000	0000	0000	0000	0000	0000	0000	1010	binary
0	0	0	0	0	0	0	A	hex

In this example, the decimal value 4 is loaded into register 3, the decimal value of 10 is loaded into register 4, and the decimal value of 11 is loaded into the comparand register 5. When the BXH instruction is executed, the contents of registers 3 and 4 are algebraically added together, the sum being decimal value 14 (hexadecimal E). The sum is algebraically compared to the content of register 5 and then placed in register 3. Since the sum is greater than the content of register 5, a branch to the instruction labeled LOOP takes place. There, the content of CARDIN is compared to the content of MAXIMUM.

BXLE

8.7. BRANCH ON INDEX LOW OR EQUAL (BXLE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
BXH	87	RS	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *branch on index low or equal* (BXLE) instruction algebraically compares the sum of the operand 1 register and operand 3 register to either the operand 3 register or one greater than the operand 3 register (comparand register). If the sum is less than or equal to the content of the comparand register, a branch to the instruction located at the operand 2 address takes place. If a less than or equal to condition does not exist, the program continues processing with the instruction that follows the BXLE instruction. The sum is always placed in the operand 1 register after the comparison.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BXLE	$r_1, r_3, d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	BXLE	r_1, r_3, s_2

This instruction algebraically adds the content of the operand 1 register to the content of the operand 3 register. The sum is algebraically compared to the content of an odd-numbered register, which can be the same as the operand 3 register, or a register that is one larger than the operand 3 register. If the sum is less than or equal to the content of the odd-numbered register it is being compared to, a branch to the instruction located at the operand 2 address takes place. If the sum is greater than the content of the odd-numbered register it is being compared to, the program continues processing with the instruction following the BXLE instruction. Following the comparison, the sum is placed in operand 1. Usually, the BXLE instruction is executed several times (depending on program logic) until the content of the operand 1 register is less than or equal to the odd-numbered register to which it is being compared. Then the branch to the instruction located at the operand 2 address takes place.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 3.
- Any odd-numbered register either equal to operand 3 or one greater than operand 3 can be used as the comparand register.
- Operand 2 can be any location in main storage.
- The rules of algebra apply to both the addition and the comparison operations.
- The condition code remains unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                           OPERAND
1         10         16
-----------------------------------------------------------------
          L          4,VALUE1
          L          5,VALUE2
          BXLE       4,5,BRANCHTO
NEXTSEQ   AP         A,B
BRANCHTO  CP         A,B
          .
          .
          .
VALUE1    DC         F'-3'
VALUE2    DC         F'1Ø'
A         DC         PL4'2Ø6'
B         DC         PL2'16'
```

In this example, the values —3 and +10 are loaded into registers 4 and 5, respectively. The BXLE instruction compares the sum of the content of registers 4 and 5 (+7) to the content of the comparand register, register 5. Since +7 is less than 10, the branch is taken. The next instruction executed (CP) is located at BRANCHTO.

EX

8.8. EXECUTE (EX)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
EX	44	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3

SEE OPER. CONSIDERATIONS

Possible Program Exceptions

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

The *execute* (EX) instruction is used to branch to an instruction elsewhere in your program, execute it with or without having modified it temporarily, and then branch back to the instruction following the EX instruction.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EX	$r_1, d_2(x_2, b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EX	$r_1, s_2(x_2)$

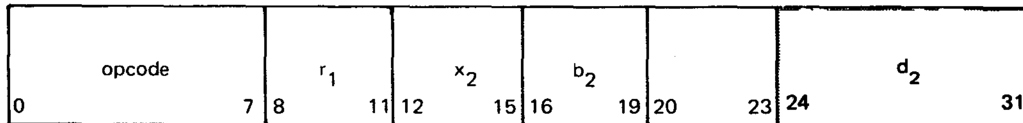
The address specified by operand 2 is the address of the instruction to which you branch following the EX instruction. This instruction, specified by operand 2, is called the subject instruction. The operand 1 register determines whether or not change will be made to the subject instruction before execution. If register 0 is specified as the operand 1 register, no change is made to the subject instruction and it is executed following the EX instruction as if it were the next sequential instruction.

On the other hand, if any register other than zero is specified, bits 8 through 15 of the subject instruction are changed. This change is accomplished by the logical addition OR on the contents of bits 24 through 31 of the operand 1 register that you previously loaded and the contents of bits 8 through 15 of the subject instruction. (See logical OR instruction.) The result is placed in bits 8 through 15 of the subject instruction. The contents of the operand 1 register remain unchanged. Moreover, the change to the subject instruction is temporary and effective only during this execution of the subject instruction.

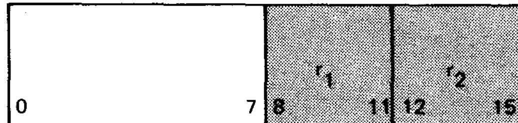
The instruction address and instruction length of the current PSW is changed by the execution of the EX and subject instruction. Normally, instruction sequencing continues with the instruction following the EX instruction. However, if the subject instruction is a successful branch instruction, the instruction address of the current PSW is replaced by the branch address and instruction sequencing resumes with the instruction address specified by the branch. If the subject instruction is a BAL or BALR, ~~instruction sequencing resumes with the instruction address specified by the link register~~. The shaded portion of each instruction shows what portion of it the operand 1 register affects.

the link register will be loaded with the address of the instruction following the EX instruction.

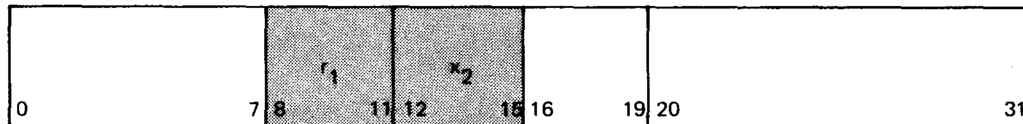
EX instruction:



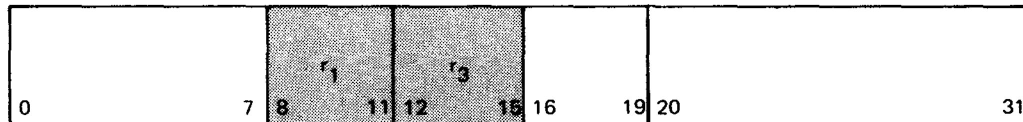
Subject instruction is RR type:



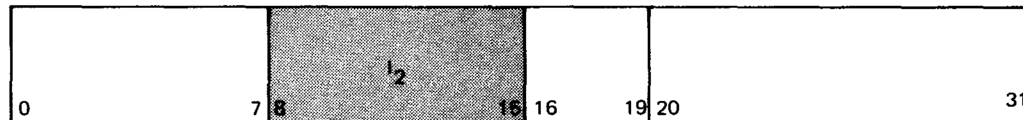
Subject instruction is RX type:



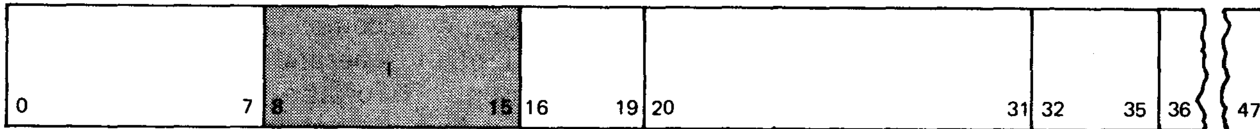
Subject instruction is RS type:



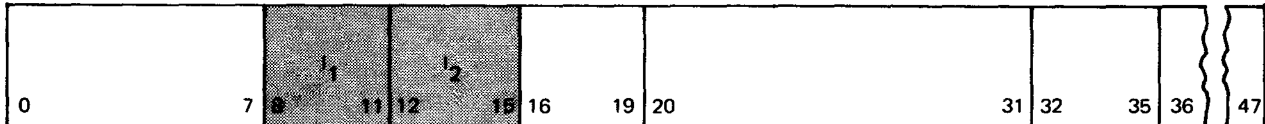
Subject instruction is SI type:



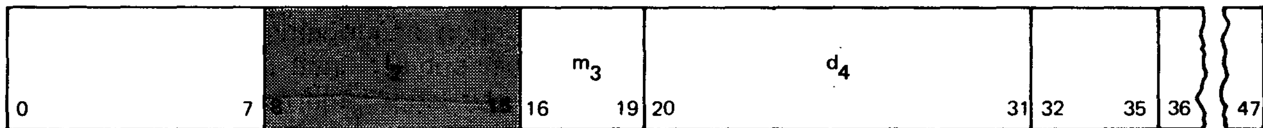
Subject instruction is SS Type 1:



Subject instruction is SS Type 2:



Subject instruction is SM type:



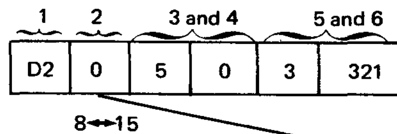
Operational Considerations:

- It isn't a good idea to alter instructions, but if it's absolutely necessary, you can use the EX instruction for that purpose.
- You can specify any of the general registers (0 through 15) as operand 1.
- Before the EX instruction causes a branch to the address you specify in operand 2, the current program address is stored in the PSW. Unless the branch to instruction (operand 2) is a successful branch instruction, sequential instruction execution follows execution of the subject instruction.
- If a program interrupt occurs after completion of the subject instruction, the old PSW contains either the address of the instruction following the EXECUTE or, in the case of a successful branch, the branch address. The current PSW contains the address of the instruction causing the interrupt (i.e., the operand 2 address in the EX instruction, or the branch-to address if a successful branch occurred before the interrupt).
- If the subject instruction is another EX instruction, a program exception occurs.
- A program exception can be caused by either the EX instruction itself or by the subject instruction.
- The condition code can be set by the subject instruction.

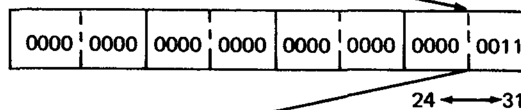
Example:

```
LABEL    ΔOPERATIONΔ                       OPERAND
1        10        16

         L         4,=F'3'
         EX        4,MOVE
         .
         .
         .
MOVE     MVC       Ø(Ø,5),INPUT
         .
         .
         .
         DS        ØH
INPUT    DS        CL8Ø
         .
         .
         .
```

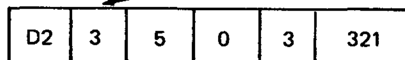
MOVE before execution of EX instruction and after execution of MVC instruction:



Register 4 before and after execution of EX instruction:



MOVE instruction during execution of MVC instruction:



In the preceding coding example, register 4 is loaded with a value of 3. The EX instruction is executed. Register 4 indicates that change will be made to the subject instruction (MOVE). A branch is made to the subject instruction and a logical addition OR is performed on the contents of bits 24 through 31 of register 4 and the contents of bits 8 through 15 of the MVC instruction. The result is placed in bits 8 through 15 of the MVC instruction only for the duration of this execution of the MVC instruction. After execution of the MVC instruction is completed, a branch is made back to the instruction following the EX, and processing continues.

9. Decimal and Logical Instructions

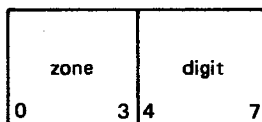
9.1. USING DECIMAL INSTRUCTIONS

Decimal instructions perform arithmetic calculations on data located in main storage, using storage-to-storage instruction format. You must put decimal numbers into main storage before attempting to use them in mathematical calculations. Storage-to-storage instructions do not allow the use of general registers for calculations, since registers handle binary, not decimal, numbers. Decimal instructions are slower than instructions that use general registers (binary arithmetic, floating-point, etc.), because two main storage locations (specified in the operand fields) are accessed each time a decimal instruction is executed.

In assembly language, decimals are expressed in either unpacked or packed format. Format refers to the way bits represent decimal numbers. Unpacked format is the standard form in which numbers are brought in to the system (input), and sent out from the system (output). Packed format is the standard form in which numbers are used in mathematical calculations.

Numbers written in unpacked format are movable from one location in main storage to another and are printable on input and output devices. Arithmetic operations, however, can only make use of packed decimal numbers. Therefore, you must pack each number before you use it. In turn, you must then unpack the number before you output it (either to a printer or any other character sensitive device).

Unpacked format uses eight bits to represent a decimal number. The leftmost four bits are the zone field, and the rightmost four bits are the decimal digit in binary.



The zone portion of a number is always a binary 1111 which is a hexadecimal F. The F in the zone field indicates that any decimal digit (0—9) in the digit field is a numeric character in EBCDIC (Extended Binary Coded Decimal Interchange Code). These relationships are shown in the following chart.

Decimal Digit	Hexadecimal (EBCDIC) Code	Binary Code
0	F0	11110000
1	F1	11110001
2	F2	11110010
3	F3	11110011
4	F4	11110100
5	F5	11110101
6	F6	11110110
7	F7	11110111
8	F8	11111000
9	F9	11111001

Since decimal operations require the number you use to be in packed format, the decimal numbers must be defined as packed constants or converted from unpacked to packed format. To convert from unpacked to packed format, use the pack decimal (PACK) instruction. The PACK instruction removes the zone bits of the unpacked decimal, thus expressing the same value in fewer bytes of main storage.

In both unpacked and packed formats, the sign is expressed in the rightmost byte which is the zone portion in unpacked format and the rightmost digit portion in packed format.

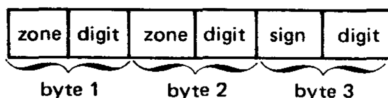
The hexadecimal numbers A through F can be sign values that are either positive or negative, and are used in either ASCII or EBCDIC mode. A hexadecimal A and B are used for output of data in ASCII mode only. A represents a positive value and B represents a negative value.

A hexadecimal C, D, and F are all used for internal processing in EBCDIC mode. C represents a positive value, D represents a negative value, and F represents an unsigned number which is assumed positive. If you attempt to print an unpacked decimal number with hexadecimal C or D as the sign value, an alpha character is printed for the rightmost byte instead of a decimal digit. Hexadecimal C and D must be changed to hexadecimal F either through the ED or OI instruction to print the correct value.

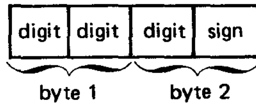
A hexadecimal F is used for output of data in EBCDIC mode and represents an unsigned number which is assumed positive.

The following illustrations represent a 3-digit decimal number in both packed and unpacked format. Notice the positions of the zone and digit portions:

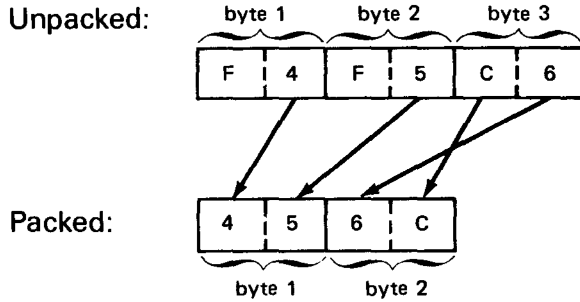
Unpacked Format:



Packed Format:



These illustrations represent the decimal number +456 in both packed and unpacked format.



There is a zone portion for every digit and one sign value in unpacked format and only digits and one sign value in packed format. Notice the number of bytes the unpacked format occupies in contrast to the packed format. The decimal number +456 occupies three bytes when unpacked and only two bytes when packed. The sign value hexadecimal C indicates that 456 is a positive number.

9.2. DEFINING PACKED AND UNPACKED CONSTANTS AND MAIN STORAGE AREAS

You can specify packed or unpacked constants and reserve areas in main storage destined to hold packed decimal values by using the define constant (DC) and define storage (DS) statements. Their format is:

LABEL	△OPERATION△	OPERAND
[symbol]	DC	[d] {P, Z} [Ln] 'c'
[symbol]	DS	[d] {P, Z} [Ln] ['c']

Duplication factor — [d]
Definition type — {P, Z}
Length factor — [Ln]
Constant specification — 'c'

In this format, symbol is an optional predefined label that names the location of the constant or main storage area. The symbol's main storage address is the address of the leftmost byte of the constant or main storage area specified in the operand field. Relative addressing (symbol + 4) is acceptable.

The duplication factor d is a decimal number that tells the assembler how many times you want the constant reproduced or how many areas of the same length you want reserved in main storage. Specifying the duplication factor saves you the time of defining the same constant or area more than once.

The definition type is P for packed or Z for zoned (unpacked), which indicates the type of constant or main storage area you are specifying. There are other definition types available, but are used for other applications (Table 5—1).

The length factor Ln specifies the number of bytes of storage reserved for a constant (DC) or an area to be used in your program (DS). If no length is specified, the assembler assigns the length of the constant specified within apostrophes. By explicitly specifying a length, you can determine the lengths of all the fields in your program regardless of how large or small your constants are.

The constant itself (c) is enclosed in apostrophes. In the case of a DS statement, the constant you enclose in the apostrophes is not actually generated, but its length determines the length of the main storage area allocated. Embedded blanks cannot be used in packed and zoned type constants.

9.2.1. Packed Decimal Constants and Main Storage Areas

When you specify packed decimal constants, the character P is the definition type in the operand field. Packed decimal constants can be up to 31 decimal digits (16 bytes) and can be signed or unsigned. If unsigned, the value is assumed to be positive. The address of the symbol you put in the label field is the address of the constant you define in the operand field. When you specify a packed decimal constant, the actual decimal value you specify is placed into main storage.

Example:

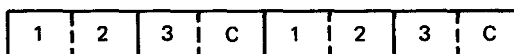
```
   LABEL   ΔOPERATIONΔ                    OPERAND
   1       10        16

1. NUM1    DC        P'+4563'
2. NUM2    DC        PL3'123'
3. NUM3    DC        2PL2'123'
```

1. This coding statement produces this packed decimal constant in main storage. The 3-byte length is implied since three bytes are needed to hold the constant and its sign.

0	4	5	6	3	C

2. This coding statement produces this packed constant in main storage. In this case, the 3-byte length isn't needed to hold the constant, but since a length of three is specified, three bytes are allocated. The number is right-justified and thus the most significant unused bytes are zero-filled.

3. This coding statement produces two consecutive, duplicate 2-byte constants in main storage.



The character P also is the definition type for defining packed decimal storage areas (i.e., areas destined to hold packed decimal data). The address of the symbol you put in the label field is the address of the constant you define in the operand field. No actual constant is placed into the area you reserve, and the area is not cleared of any data it may already contain. You are merely reserving a main storage area for future use.

Example:

```
   LABEL    ΔOPERATIONΔ                     OPERAND
   1        10        16
  ---------------------------------------------------------
1. NUM4     DS        P'+4563'
2. NUM5     DS        PL2
3. NUM6     DS        PL1'6'
```

1. This coding statement reserves a 3-byte area in main storage. The statement does not put the packed decimal +4563 into that area but merely reserves an area large enough to hold it.

2. This coding statement reserves a 2-byte area in main storage. It does not clear the area or put anything into it.

3. This coding statement reserves a 1-byte area in main storage.

If you intend to reserve a packed storage area for mathematical calculations in either of these three ways, move zeros into the specified storage area to clear it of any leftover data from another program. This will ensure that mathematical calculations are performed correctly.

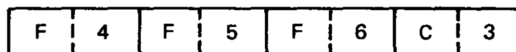
9.2.2. Unpacked Decimal Constants and Main Storage Areas

When you specify unpacked decimal constants, the character Z is the definition type in the operand field. Unpacked decimal constants can be up to 16 decimal digits (16 bytes) and can be signed or unsigned. If unsigned, the value is assumed to be positive. The address of the symbol you put in the label field is the address of the constant you define in the operand field. When you specify an unpacked decimal constant, the actual decimal value you specify is placed into main storage as digits with zone fields of hexadecimal F.

Example:

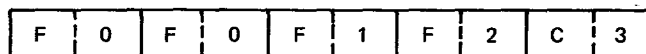
```
   LABEL    ΔOPERATIONΔ                    OPERAND
   1        10        16

1. [symbol] DC        Z'+4563'
2. [symbol] DC        ZL5'123'
3. [symbol] DC        2ZL3'123'
```

1. This coding statement produces this unpacked decimal constant in main storage. The 4-byte length is implied since four bytes are needed to hold the unpacked constant with zones and sign.



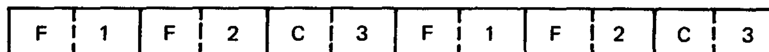
F	4	F	5	F	6	C	3

2. This coding statement produces this unpacked decimal constant in main storage. In this case, the 5-byte length isn't needed to hold the constant, but since a length of five is specified, five bytes are allocated. Note that the C in the rightmost byte represents a signed positive value.



F	0	F	0	F	1	F	2	C	3

3. This coding statement produces two consecutive 3-byte constants in main storage.



F	1	F	2	C	3	F	1	F	2	C	3

The character Z also is the definition type for defining unpacked decimal main storage areas (i.e., area destined to hold unpacked decimal data). The address of the symbol you put in the label field is the address of the main storage area you define in the operand field. No actual constant is placed into the area you reserve, and the area is not cleared of any data it may already contain. You are merely reserving a main storage area.

Example:

```
     LABEL   ΔOPERATIONΔ                    OPERAND
     1       10       16
    ------------------------------------------------------------
 1. ZNUM1    DS       Z'4543'
 2. ZNUM2    DS       ZL4
 3. ZNUM3    DS       2ZL4
```

1. This coding statement reserves a 4-byte area in main storage. The actual unpacked decimal constant 4543 is not placed into the reserved area by this statement.

2. This coding statement also reserves a 4-byte area in main storage.

3. This coding statement reserves two consecutive 4-byte areas in main storage.

AP

9.3. ADD DECIMAL (AP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AP	FA	SS	6

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
■ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *add decimal* (AP) instruction algebraically adds the packed decimal contents of operand 2 (the sending field) to the packed decimal contents of operand 1 (the receiving field). The sum is stored in operand 1 and is filled, a byte at a time, from right to left.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AP	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AP	$s_1(l_1),s_2(l_2)$

Operational Considerations:

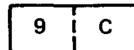
- The value and sign of the sum are algebraically, not logically, calculated.
- Since the sum is stored in the operand 1 location and if the length of the sum is greater than the length of operand 1, the leftmost digits of the sum are truncated.

Example:

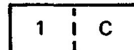
```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10         16
------------------------------------------------------------
         AP         A,B
         .
         .
         .
A        DC         P'2'
B        DC         P'9'
```

A before AP execution:

B before and after AP execution:



A after AP execution:

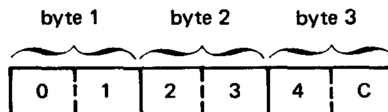


As shown, the entire sum (11) does not fit into the one byte allocated in operand 1, so the leftmost digit of the sum is lost. If the operand field of the DC statement defining A is changed to PL2'2', two bytes are allocated for the sum, and the correct 2-byte sum fits into the allocated area. If the sum does not fill the length specified in operand 1, zeros fill the remaining leftmost bytes of operand 1. A zero sum is positive as long as the length of operand 1 is large enough to hold the entire sum (i.e., no leftmost digits are lost). If the sum is zero and the leftmost digits are lost, the sign is the sign of the sum before the digits were lost. It is possible to double a number (add it to itself) when the rightmost bytes of operands 1 and 2 have overlapping bytes in main storage.

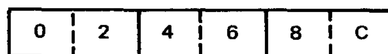
Example:

```
         AP         A,A
         .
         .
         .
A        DC         P'1234'
```

A before AP execution:



A after AP execution:



The entire contents of A is extracted, doubled, and the answer returned to the same field. This destroys the original contents of A.

CP

9.4. COMPARE DECIMAL (CP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CP	F9	SS	6

Condition Codes
■ IF OPI = OP2, SET TO 0
■ IF OPI <OP2, SET TO 1
■ IF OPI >OP2, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *compare decimal* (CP) instruction is used to compare operand 1 to operand 2, byte-by-byte from right to left. The result determines the setting of the condition code. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CP	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CP	$s_1(l_1),s_2(l_2)$

Based on the comparison result, the condition code of the program status word (PSW) is set to 1 if operand 1 is less than operand 2, to 2 if operand 1 is greater than operand 2, and to 0 if operands 1 and 2 are equal.

The condition code is part of the PSW, a double-word register that holds information pertinent to instruction execution. The instruction executed following the CP instruction depends on the condition code setting. The four condition code settings are as follows:

Condition Code (Decimal Value)	Bit Configuration (Bits 34–35 of PSW)
0	00 = test value is binary 8 (1000)
1	01 = test value is binary 4 (0100)
2	10 = test value is binary 2 (0010)
3	11 = test value is binary 1 (0001)

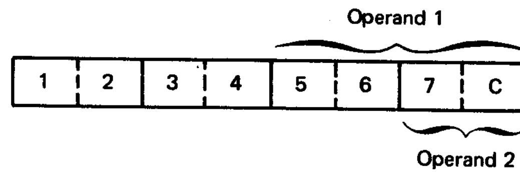
Operational Considerations:

- The CP instruction compares the rightmost byte of the operands first, and then works to the left one byte at a time.
- If operand 1 is shorter than operand 2, or operand 2 is shorter than operand 1, zeros fill the leftmost bytes of the shorter operand, thus making the operands the same length during the comparison. Even though zeros are added, neither operand is permanently changed by the instruction.
- Any zero compares equal to another zero regardless of their signs.
- Positive signs compared to each other compare as equal; and the same holds true for negative signs.
- It is possible to compare a decimal, or part of a decimal to itself, or part of itself, by overlapping the location of the rightmost bytes of the operands in main storage.

Example:

```
LABEL   ΔOPERATIONΔ                        OPERAND
1        10       16
-------------------------------------------------------------
         CP       A+2(2),A+3(1)
         .
         .
         .
A        DC       PL4'1234567'
```

A before and after execution of CP instruction:



In this example, the packed decimal contents of operand 1 are compared to the packed decimal contents of operand 2. Operands 1 and 2 have overlapping rightmost bytes. The processor temporarily adds a byte of zeros to operand 2 since operand 2 has fewer bytes than operand 1. After the CP instruction is executed, the condition code is set to 2 because operand 1 is greater than operand 2.

9.5. DIVIDE DECIMAL (DP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
DP	FD	SS	6

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *divide decimal* (DP) instruction divides the packed decimal contents of operand 1 (dividend) by the packed decimal contents of operand 2 (divisor). The result (quotient and remainder) is stored in operand 1 (the receiving field) which is filled from right to left.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DP	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DP	$s_1(l_1),s_2(l_2)$

Operational Considerations:

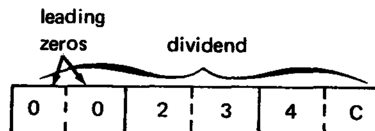
- Operand 1 contains both the quotient and the remainder after the DP instruction is executed. Since operand 1 is the receiving field for the result, it consists of two side-by-side fields. The remainder with sign occupies the rightmost field, and the quotient with sign occupies the leftmost field. The leftmost byte of the quotient is the address specified by operand 1, and the rightmost byte of the remainder is the rightmost byte specified by operand 1.
- The quotient with sign and the remainder with sign are determined algebraically. The sign of the remainder takes its sign value from the sign of the dividend.

- The length of the dividend is restricted to 16 bytes and must have at least one leading zero in the leftmost portion. As a result, the most significant digit is always zero. The length of operand 1 should be sufficient to hold the quotient, the remainder, and their signs.
- The divisor length is restricted to eight bytes. The operand 2 field, which holds the divisor, is unchanged after the DP instruction is executed.
- The length of the quotient is restricted to 15 bytes. This length is equal to the number of bytes needed to hold the dividend with sign and the divisor with sign (operand 1 + operand 2).
- The length of the remainder must be at least one byte. The length of the remainder is the length of the divisor and is therefore restricted to eight bytes.
- If the result is larger than the length specified for operand 1, or if you attempt to divide by zero, a decimal divide program exception occurs.
- If you want to reuse operand 1 for further mathematical calculations, you must move a packed field of zeros into the specified area to clear it of any leftover data.
- In fixed-point instructions, it is your responsibility to keep track of assumed decimal points. To add or delete decimal places, you can multiply or divide by powers of 10. You can also use the move with offset (MVO) instruction (see 9.9) to drop any number of leftmost digits you specify.

Example:

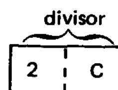
```
LABEL    ΔOPERATIONΔ                     OPERAND
1        10       16
-------------------------------------------------------
         DP       NUM1,NUM2
         .
         .
         .
NUM1     DC       PL3'234'
NUM2     DC       P'2'
```

NUM1 before execution of DP instruction:



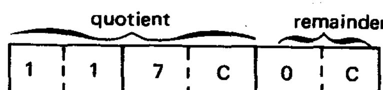
Packed decimal number

NUM2 before and after execution of DP instruction:



Packed decimal number

NUM1 after execution of DP instruction:

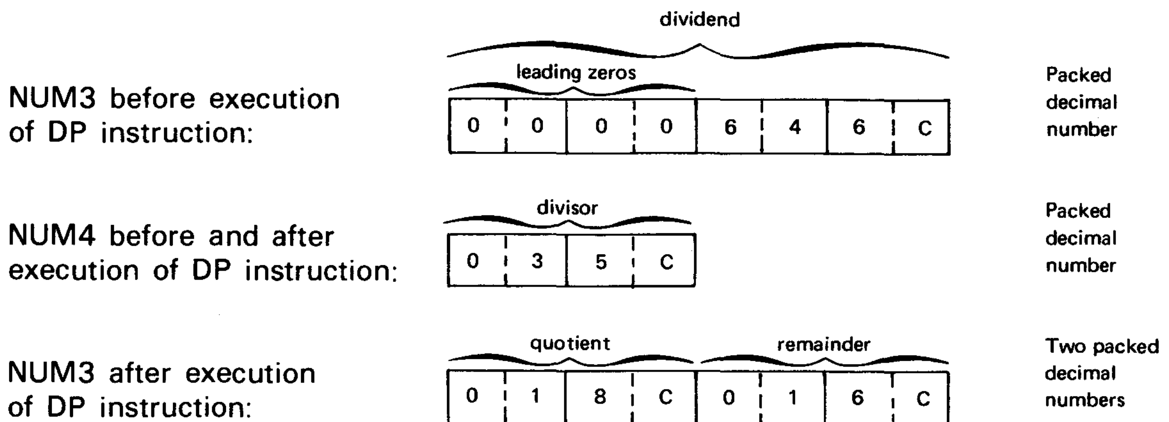


Two packed decimal numbers

In this example, the packed decimal content of NUM1 is divided by the packed decimal content of NUM2. The result (quotient and remainder) replaces NUM1. Note the dividend is a 3-byte field containing leading zeros. Its length is calculated by adding the actual number of bytes required to hold the data in NUM1 (two bytes) and NUM2 (one byte) which gives a total of three bytes for the dividend. Since the divisor is one byte, the remainder also is one byte. Note that the remainder with sign occupies the rightmost byte of NUM1 and the quotient with sign occupies the remaining (leftmost) portion of NUM1.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10       16
-----------------------------------------------------------------
         DP       NUM3,NUM4
         .
         .
         .
NUM3     DC       PL4'646'
NUM4     DC       P'35'
```



In this example, the packed decimal content of NUM3 is divided by the packed decimal content of NUM4. The result is placed in NUM3. The length of operand 1 is calculated by adding the actual number of bytes required to hold the data in NUM3 (2 bytes) and NUM4 (2 bytes) which gives a total of four bytes for the dividend. In this example there also is a remainder of 16 that occupies the same number of bytes as the divisor and is located in the rightmost portion of operand 1.

ED

9.6. EDIT (ED)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ED	DE	SS	6

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
□ SET TO 3
SEE OPER. CONSIDERATIONS

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

The *edit* (ED) instruction unpacks and modifies packed operand 2 data so that the printed output received is printed the way you want it displayed. This modification is controlled by the operand 1 edit pattern.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ED	$d_1(l_1,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ED	$s_1(l),s_2$

The contents of operand 2 must be a packed decimal number. Operand 1 contains the edit pattern which consists of EBCDIC character codes. It is the pattern of EBCDIC character codes you specify that determines how results are displayed. The edit pattern can rearrange, delete, select, or insert any needed data, symbols, or characters in the operand 2 data. The edited result (unpacked and modified operand 2 data) replaces the operand 1 edit pattern.

Operational Considerations:

- The length of the operand 1 edit pattern is almost always longer than operand 2 because operand 1 is in unpacked format while operand 2 is in packed format.
- The edited result replaces operand 1, thus permanently destroying the edit pattern. If you intend to reuse the edit pattern, then it must be saved or moved prior to the execution of the ED instruction.
- The total number of significance starters and digit selectors in operand 1 must equal the total number of digits in operand 2.
- If there is no significance starter in operand 1, all zeros in operand 2, and the fill character is hexadecimal 40, the resultant field is blank.
- The condition code reflects only the last field edited or the field after the last field separator.
- The S switch reflects the sign of the last byte in operand 2. A plus sign detected as the least significant digit turns the S switch off. A minus sign has no effect on the S switch, and a plus or minus sign detected as the most significant digit causes a data exception.
- The sign of operand 2 is converted to hexadecimal F when edited, regardless of whether it is a hexadecimal C or F (positive), or a hexadecimal D (negative).

9.6.1. The Edit Pattern

The operand 1 edit pattern may consist of five types of pattern characters:

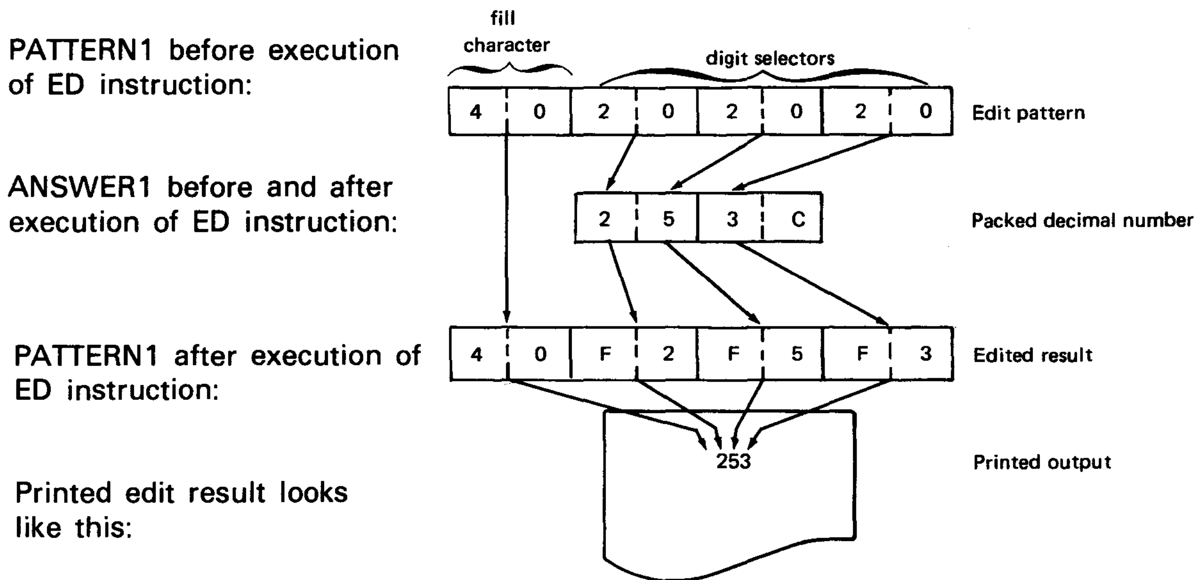
- Fill character
- Digit selector
- Significance starter
- Message character
- Field separator

The *fill character*, in all cases, is the leftmost byte of operand 1. It is any EBCDIC character code you choose. The EBCDIC character code specified is the first byte of the edited result, and replaces (or fills in) certain pattern characters corresponding to any nonsignificant operand 2 digits. (The significant digits are the digits 1 thru 9. Zero is the only nonsignificant digit but becomes significant when it follows a significant digit or the significance starter (hexadecimal 21)). The edited result replaces the operand 1 edit pattern. Some of the more commonly used fill characters are hexadecimal 40 (blank), hexadecimal 5B (dollar sign), and hexadecimal 5C (asterisk).

The *digit selector* is the EBCDIC character code 20. For every digit in operand 2, there must be a corresponding hexadecimal 20 in the operand 1 edit pattern. Every significant digit in operand 2 replaces its corresponding digit selector in operand 1. If there is a nonsignificant digit in operand 2, the fill character replaces its corresponding digit selector.

Example:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1        10       16
_____________________________________________________________
         ED       PATTERN1,ANSWER1
         .
         .
         .
PATTERN1 DC       X'40202020'
ANSWER1  DC       P'253'
```



Note that in PATTERN1 there is a corresponding hexadecimal 20 for every digit in ANSWER1. The edit pattern (operand 1) is examined one byte at a time and operand 2 is examined one digit at a time. The fill character remains as the first byte of the edit result (operand 1), and the succeeding pattern characters (in this example, the digit selectors) are replaced by unpacked operand 2 digits.

Example:

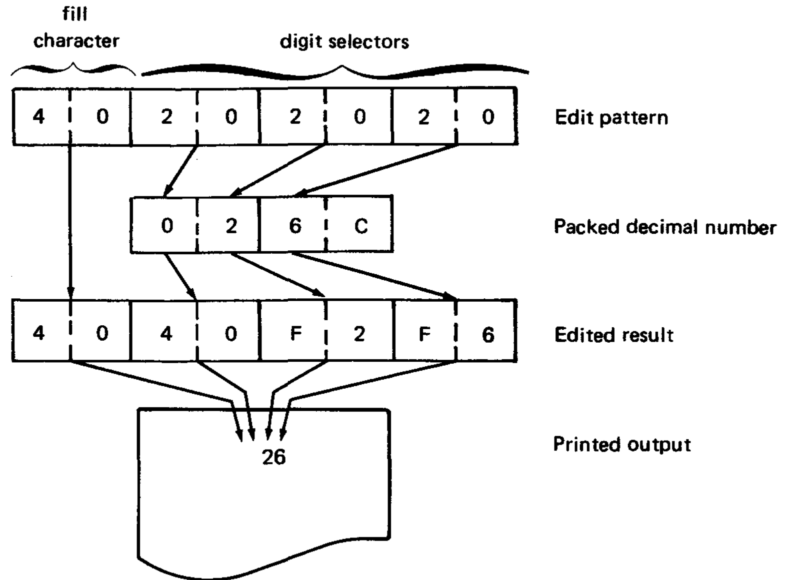
```
         ED       PATTERN2,ANSWER2
         .
         .
         .
PATTERN2 DC       X'40202020'
ANSWER2  DC       P'26'
```

PATTERN2 before execution of ED instruction:

ANSWER2 before and after execution of ED instruction:

PATTERN2 after execution of ED instruction:

Printed edit result looks like this:

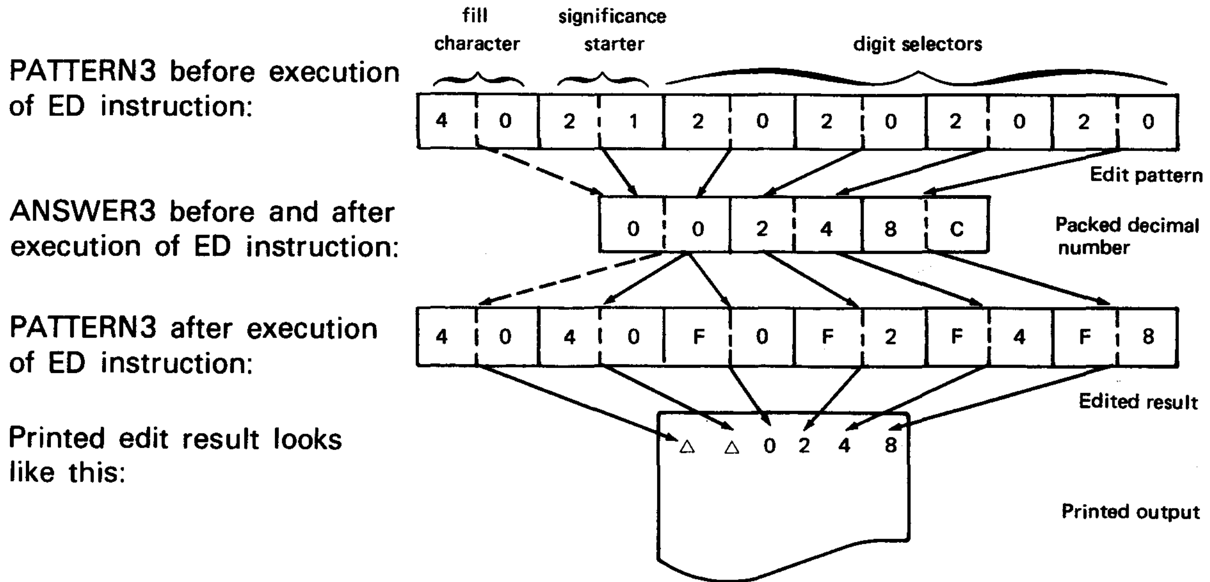


In this example, note that ANSWER2 is padded to the left with a zero. Zero is considered a nonsignificant digit because it precedes any significant digits. Therefore, the fill character hexadecimal 40 replaces the second byte of the edit result, because zero corresponds to the first digit selector. The other packed operand 2 digits are unpacked and replace the remaining digit selectors. This method of changing lead zeros to blanks is called zero suppression.

The *significance starter* is the EBCDIC character code 21. You need to specify only one hexadecimal 21 for each field to be edited. Every hexadecimal 21 must correspond to a digit in operand 2. When a hexadecimal 21 is examined in an edit pattern, it is replaced with its corresponding digit in operand 2 and then turns on the significance start switch (S switch). The significance start switch is an internal switch that when turned on forces all of the following pattern characters either to remain or be replaced in the edit result. In effect, the only conditions that force the S switch on are a hexadecimal 21 or a significant digit. On the other hand, the S switch is turned off after a digit in operand 2 is examined whose sign is positive and located in the rightmost four bit positions of a field. A negative sign does not affect the S switch. Whether the sign is positive or negative, all results are printed as positive values.

Example:

```
LABEL     ΔOPERATIONΔ                     OPERAND
1          10        16
------------------------------------------------------------
           ED        PATTERN3,ANSWER3
           .
           .
           .
PATTERN3  DC        X'4Ø212Ø2Ø2Ø2Ø'
ANSWER3   DC        PL3'248'
```



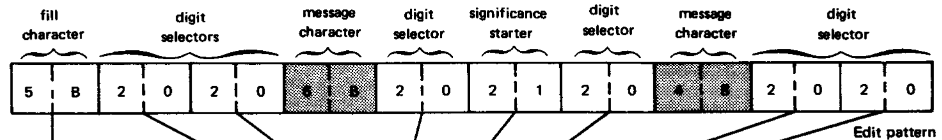
The fill character remains as the first byte of the edit result, and also replaces the significance starter because its corresponding digit in ANSWER3 is a nonsignificant zero. Now, the S switch is turned on and the second zero in ANSWER3 becomes significant. It replaces its corresponding digit selector with a zero. The succeeding operand 2 digits are unpacked and replace the remaining digit selectors.

The *message character* can be any EBCDIC character code except hexadecimal 20, 21, or 22. If the message character follows a significant digit or the significance start switch (hexadecimal 21), the message character remains as part of the edited result (operand 1). On the other hand, if the message character precedes a significant digit or a hexadecimal 21, it is replaced by the fill character. Some of the most commonly used message characters are hexadecimal 6B (comma) and hexadecimal 4B (decimal point).

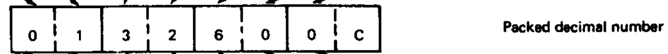
Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10        16
------------------------------------------------------------
          ED         PATTERN4,ANSWER4
          .
          .
          .
PATTERN4  DC         X'5B2Ø2Ø6B2Ø212Ø4B2Ø2Ø'
ANSWER4   DC         P'1326ØØ'
```

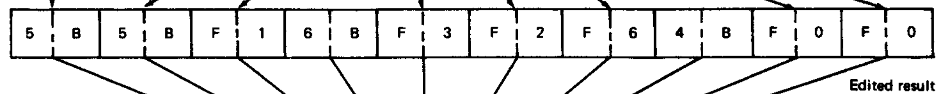
PATTERN4 before execution of ED instruction:



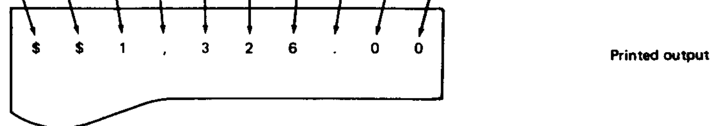
ANSWER4 before and after execution of ED instruction:



PATTERN4 after execution of ED instruction:



Printed edit result looks like this:



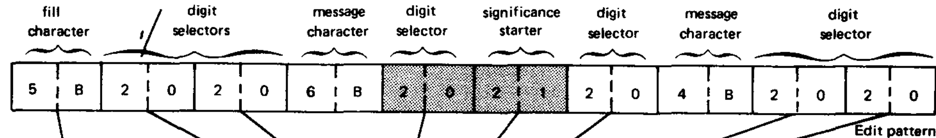
In this example, hexadecimal 5B (dollar sign) is used as the fill character. It remains as the first byte of the edited result and also replaces the second byte because the second byte's digit selector corresponds to a nonsignificant zero. The digit 1 in ANSWER4 replaces the third byte and the message character hexadecimal 6B remains because it follows a significant digit. The digits 3, 2, and 6 in operand 2 replace their corresponding pattern characters, the message character hexadecimal 4B remains and the trailing zeros in ANSWER4 replace their corresponding pattern characters.

Note the position of the significance starter hexadecimal 21. In this example, the S switch is turned on by the first significant digit. Therefore, when this hexadecimal 21 is examined, it is replaced with its corresponding digit in ANSWER4. Now, suppose the edit pattern remains the same and operand 2 is changed to look like this example:

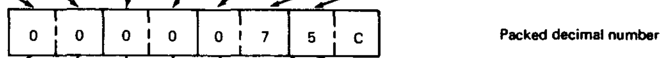
Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10         16
-----------------------------------------------------------------
         ED         PATTERN5,ANSWER5
         .
         .
         .
PATTERN5 DC         X'5B2Ø2Ø6B2Ø212Ø4B2Ø2Ø'
ANSWER5  DC         PL4'75'
```

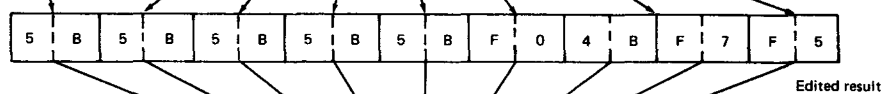
PATTERN5 before execution of ED instruction:



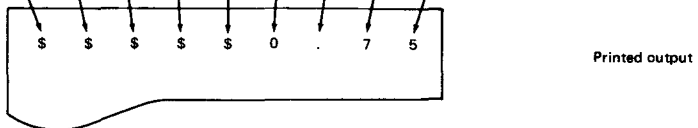
ANSWER5 before and after execution of ED instruction:



PATTERN5 after execution of ED instruction:



Printed edit result looks like this:



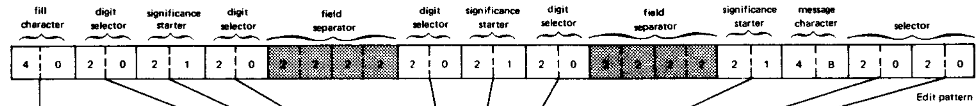
The significance starter (hexadecimal 21) is placed in the sixth byte of PATTERN5 so that the S switch is turned on to force the display of bytes 7 through 10. As a result, the least significant dollar integer, the decimal point, and the cents are always represented no matter how small or large the value of operand 2 is.

The *field separator* is the EBCDIC character code 22. It is used to separate two or more contiguous fields. These fields must be packed decimal numbers in operand 2 and located in consecutive order in main storage. The fill character you specify replaces all field separators. As soon as a hexadecimal 22 is examined, the S switch is turned off and the field separator is replaced with the fill character.

Example:

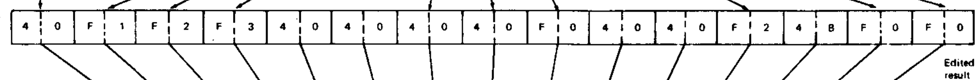
```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10       16
_____________________________________________________________
          ED       PATTERN6,ANSWER6
          .
          .
          .
PATTERN6  DC       X'4Ø2Ø212Ø22222Ø212Ø222221 4B2Ø2Ø'
ANSWER6   DC       P'123CØØØC2ØØC'
```

PATTERN6 before execution of ED instruction:



ANSWER6 before and after execution of ED instruction:

PATTERN6 after execution of ED instruction:



Printed edit result looks like this:

In this example, ANSWER6 contains three packed contiguous fields each separated by two field separators (hexadecimal 22). Since the fill character specified is hexadecimal 40, that character is used as the fill character for all fields, and also replaces each field separator in the edit pattern. Remember that the S switch is turned off as soon as a hexadecimal 22 is examined. This causes any leading zeros in succeeding fields to be nonsignificant digits.

9.6.2. The Resulting Condition Code

All operand 2 digits examined are tested for condition code 0. The sign of the last packed field edited, and whether or not all the digits in that field are zeros, are recorded in the condition code setting when execution of the ED instruction is completed.

The condition code is set to 0 when:

- all digits in the last field edited in operand 2 are zeros;
- the edit pattern has no digit selectors or significance starters causing operand 2 digits not to be examined;
- the last character in the edit pattern is a field separator; and
- the edit pattern has no digit selectors or significance starters after the last field separator.

The condition code is set to 1 when:

- the last field edited is not all zeros but the S switch is on. This indicates the value of the last field edited is less than zero, because a negative sign does not affect the S switch.

The condition code is set to 2 when:

- the last field edited is not all zeros but the S switch is off. This indicates the value of the last field edited is greater than zero, because a positive sign turns the S switch off.

9.6.3. Examples of General Usage

The following examples are more commonly used and can be applied in practical situations. The first example shows how a nonblank fill character is used.

Example:

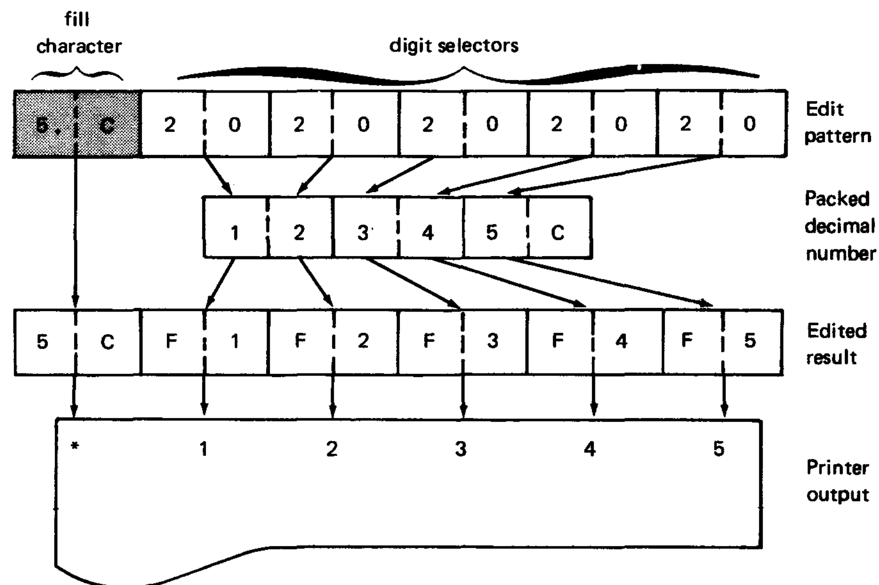
```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10       16
-----------------------------------------------------
         ED       PATTERN7,ANSWER7
         .
         .
         .
PATTERN7 DC       X'5C2Ø2Ø2Ø2Ø2Ø'
ANSWER7  DC       P'12345'
```

PATTERN7 before execution of ED instruction:

ANSWER7 before and after execution of ED instruction:

PATTERN7 after execution of ED instruction:

Printed edit result looks like this:



All results, whether positive or negative, are printed as positive results. By using message characters in the edit pattern, you can indicate whether a field in operand 2 is positive or negative. These message characters should be the last pattern characters in the edit pattern for each corresponding field in operand 2. If the value of operand 2 is negative, message characters placed to the right of the rightmost digit selector remain as part of the edit result. Since a negative sign in the rightmost four bit positions does not affect the S switch, the message characters become significant. However, if the value of operand 2 is positive, the message characters are replaced by the fill character.

Since a positive sign in the rightmost four bit positions turns the S switch off, the message characters become nonsignificant. You can specify any character or any number of characters to indicate a negative number, but the most commonly used are hexadecimal C3D9 (CR), hexadecimal C3C2 (DB), and hexadecimal 60 (—). The following example illustrates editing a negative number using the minus sign.

Example:

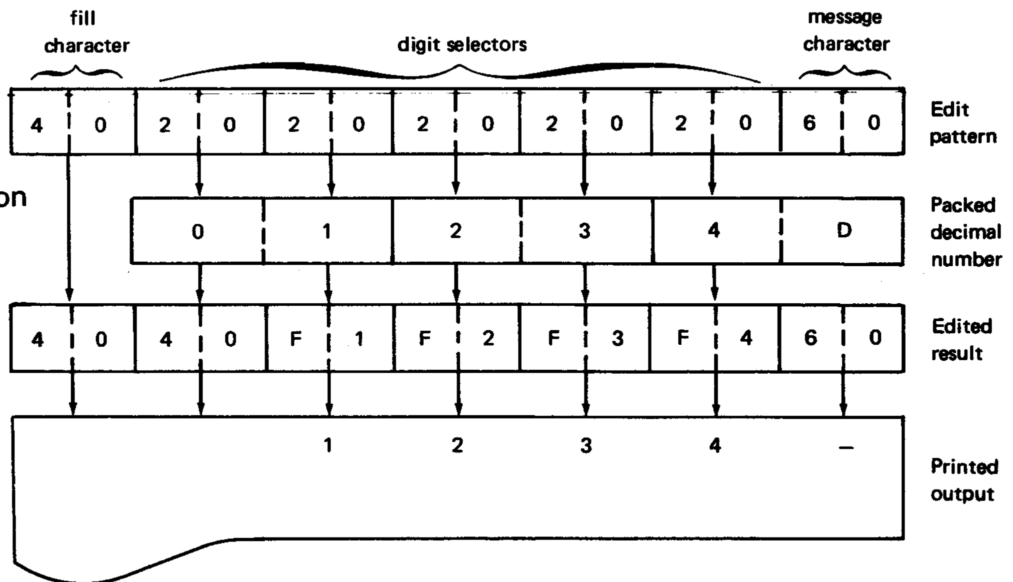
```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10        16
_______________________________________________________________
          ED        PATTERN9,ANSWER9
          .
          .
          .
PATTERN9  DC        X'40202020202060'
ANSWER9   DC        P'-1234'
```

PATTERN9 before execution of ED instruction:

ANSWER9 before and after execution of ED instruction:

PATTERN9 after execution of ED instruction:

Printed edit result looks like this:

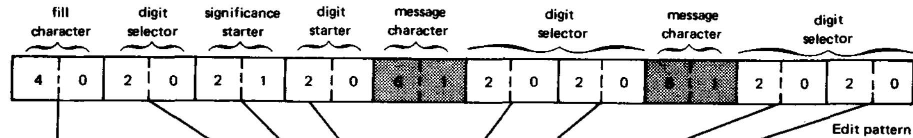


The following example illustrates date-field editing. Here, message characters are inserted into a 5- or 7-digit field. The most commonly used message characters for a date field are hexadecimal 61 (slash), hexadecimal 60 (hyphen), and hexadecimal 40 (blank).

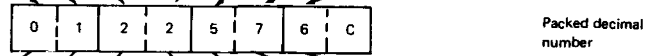
Example:

```
          ED        PATTERNA,ANSWERA
          .
          .
          .
PATTERNA  DC        X'4020212061202061202020'
ANSWERA   DC        P'122576'
```

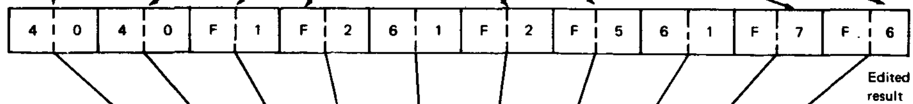
PATTERNA before execution of ED instruction:



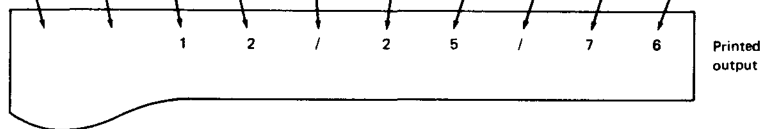
ANSWERA before and after execution of ED instruction:



PATTERNA after execution of ED instruction:



Printed edit result looks like this:



9.6.4. Summary

Table 9—1 summarizes and combines the information in this section to provide a clear and concise picture of the ED instruction and its functions.

Table 9—1. Edit Instruction Operation

Pattern (Operand 1) Character	EBCDIC Code	Previous Switch Status	Decimal (Operand 2) Digit	Sign of Least Significant Byte (Operand 2)	Resulting (Operand 1) Character	Resulting Switch Status
Fill character	Any	Off	Not examined	*	Fill character	Off
Digit selector	20	Off	0	*	Fill character	Off
		Off	1–9	*	Digit	On
		Off	1–9	Positive	Digit	Off
		Off	1–9	Negative	Digit	On
		On	0	*	Digit	On
		On	1–9	*	Digit	On
		On	1–9	Positive	Digit	Off
		On	1–9	Negative	Digit	On
Significance starter	21	Off	0	*	Fill character	On
		Off	1–9	*	Digit	On
		On	0	*	Digit	On
		On	1–9	*	Digit	On
Message character	Any except 20,21,22	Off	Not examined	*	Fill character	Off
		Off	Not examined	Positive	Fill character	Off
		On	Not examined	*	Message character	On
		On	Not examined	Positive	Fill character	Off
		On	Not examined	Negative	Message character	On
Field separator	22	Off	Not examined	*	Fill character	Off
		On	Not examined	*	Fill character	Off

*Not applicable

9.7. EDIT AND MARK (EDMK)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
EDMK	DF	SS	6

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The edit and mark (EDMK) instruction operates like the edit (ED) instruction except that it also saves the address of the first significant byte and places it in register 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EDMK	$d_1(l, b_1), d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EDMK	$s_1(l), s_2$

Operational Considerations:

- The EDMK instruction operates like the ED instruction (see 9.6). After the packed content of operand 2 is edited and the unpacked result stored in operand 1, the address of the first nonzero character is placed in general register 1.

- If the field to be edited contains no significant digits until after the significance starter, no address is moved into register 1, and the move instruction following the EDMK instruction will be using the incorrect address (or whatever value) that is in register 1.

- To avoid having an incorrect address in register 1 because no significant digits exist before the significance starter, load register 1 with the address of the position where you want the insert character to be placed.
- If a field to be edited contains multiple fields, the address of the first significant byte in each field replaces the one before. So in effect, the address of the first significant byte in the last field is the final result.
- The EDMK instruction is a featured instruction. An operation program exception is caused if you use this instruction and the processor does not have the control feature installed.
- This instruction is used to insert a character in several places throughout the output display. For example:

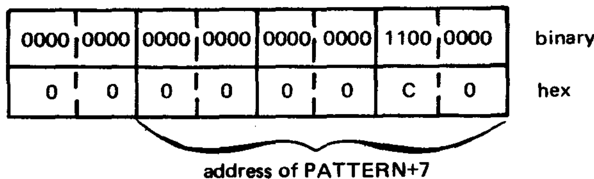
```
  $6.25
 $86.00
  $2.34
$724.11
```

  The location of the dollar sign is predictable in that it appears at the left of the first significant digit on each line. The decimal point position also is predictable as the third character from the right. The proper positioning of a dollar sign or other message character is ensured by using the EDMK instruction.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10        16
--------------------------------------------------------
          MVC       PATTERN,MASK
          LA        1,PATTERN+7
          EDMK      PATTERN,DATA
          S         1,=F'1'
          MVI       Ø(1),C'$'
          .
          .
          .
PATTERN   DS        CL1Ø
MASK      DC        X'4Ø2Ø2Ø6B2Ø2Ø214B2Ø2Ø'
DATA      DC        P'Ø24571Ø'
```

Register 1 before execution of EDMK instruction:



Register 1 after execution of EDMK instruction:

0000	0000	0000	0000	0000	0000	1011	1011	binary
0	0	0	0	0	0	B	B	hex

address of 1st significant digit

Register 1 after execution of S instruction:

0000	0000	0000	0000	0000	0000	1011	1010	binary
0	0	0	0	0	0	B	A	hex

address of byte to the left of 1st significant digit

Edited result after execution of MVI instruction:

$ / 1st significant digit

0100	0000	0101	1010	1111	0010	0110	1011	1111	0100	1111	0101	1111	0111	0100	1011	1111	0001	1111	0000	binary
4	0	5	B	F	2	6	B	F	4	F	5	F	7	4	B	F	1	F	0	hex

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

DELETION

Pages 9—30 through 9—62, Figures 9—1 through 9—8, and Tables 9—1 through 9—7 have been deleted.

MSS

(FOLLOWS THIS [illegible])

9.9. MOVE CHARACTER (MVC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MVC	**D2**	**SS**	**6**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *move character* (MVC) instruction moves the contents of one area in main storage (operand 2) into another area in main storage (operand 1). The length of operand 1 determines the number of bytes moved.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVC	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVC	$s_1(l),s_2$

The move character instruction moves data referenced by operand 2 (the sending field) to the location referenced by operand 1 (the receiving field). Data is moved a byte at a time from left to right. The length of operand 1, whether implied or explicit, determines the number of bytes to be moved.

Operational Considerations:

- The instruction moves one byte at a time, processing from left to right through each field.

- The length of operand 1 determines the number of bytes moved. It can be either implied or explicit.

- When using operands with overlapping bytes, the results are often unpredictable.
- One character can be propagated through an entire field if the sending field begins with the first byte of a field and the receiving field begins with the second byte of that same field.
- Any type of data can be specified in an MVC instruction.

Example:

```
LABEL     ΔOPERATIONΔ                          OPERAND
1         10         16
--------------------------------------------------------------------
          MVC        RECEIVE1,SEND1
          .
          .
          .
RECEIVE1  DC         CL5'DIGIT'
SEND1     DC         CL5'SALES'
```

RECEIVE1 before execution of MVC instruction:

D	I	G	I	T	Alpha characters
C 4	C 9	C 7	C 9	E 3	Hexadecimal (EBCDIC mode)

SEND1 before and after execution of MVC instruction:

S	A	L	E	S	Alpha characters
E 2	C 1	D 3	C 5	E 2	Hexadecimal (EBCDIC mode)

RECEIVE1 after execution of MVC instruction:

S	A	L	E	S	Alpha characters
E 2	C 1	D 3	C 5	E 2	Hexadecimal (EBCDIC mode)

In this example, the content of operand 2 is moved into operand 1. Since it is an even move (a 5-byte field to a 5-byte field), the content of SEND1 completely overlays the content of RECEIVE1. Note that no length is specified for operand 1 and, as a result, the implied length is applied.

Example:

```
          MVC        RECEIVE2(5),SEND2
          .
          .
          .
RECEIVE2  DC         CL7'JANUARY'
SEND2     DC         CL5'MARCH'
```

RECEIVE2 before execution of MVC instruction:

J	A	N	U	A	R	Y	Alpha characters
D 1	C 1	D 5	E 4	C 1	D 9	E 8	Hexadecimal (EBCDIC mode)

SEND2 before and after execution of MVC instruction:

M	A	R	C	H	Alpha characters
D 4	C 1	D 9	C 3	C 8	Hexadecimal (EBCDIC mode)

leftover operand 1 data

RECEIVE2 after execution of MVC instruction:

M	A	R	C	H	R	Y	Alpha characters
D 4	C 1	D 9	C 3	C 8	D 9	E 8	Hexadecimal (EBCDIC mode)

In the preceding example, an explicit length of 5 is specified for operand 1. The 5 determines that RECEIVE2 will accept only five bytes from SEND2. The five bytes from SEND2 are moved to RECEIVE2 filling operand 1 from left to right. As you can see, five bytes of SEND2 (MARCH) are moved to the first five bytes of RECEIVE2 (JANUA). Note that the last two bytes of RECEIVE2 still remain.

Example:

```
LABEL   ΔOPERATIONΔ                    OPERAND
1       10     16

        MVC    TOTAL(2),ANSWER+1
        .
        .
        .
TOTAL   DC     PL4'Ø'
ANSWER  DC     PL3'128'
HEADING DC     CL19'TOTAL SALES FOR MAY'
```

TOTAL before execution of MVC instruction:

0 0	0 0	0 0	0 C	Packed decimal number

ANSWER + 1

ANSWER before and after execution of MVC instruction:

0 0	1 2	8 C	Packed decimal number

leftover operand 1 data

TOTAL after execution of MVC instruction:

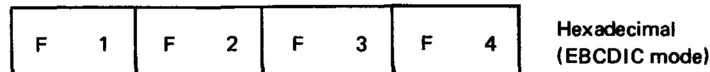
1 2	8 C	0 0	0 C	Two packed decimal numbers

Since the concerned number occupies the second and third bytes of ANSWER, relative addressing (ANSWER + 1) is used to address the second byte, and avoid the first byte of zeros. Note the number of bytes moved is restricted to two by using an explicit length in operand 1. If an explicit length is not specified, four bytes would be moved to TOTAL since it is a 4-byte field. Bytes two and three of operand 2, plus the first two bytes of data contiguous to operand 2 (in this case the letters TO of HEADING), would be moved to TOTAL.

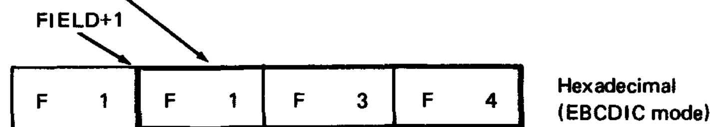
Example:

```
LABEL   ΔOPERATIONΔ                       OPERAND
1       10        16
-----------------------------------------------------------
        MVC       FIELD+1(3),FIELD
        .
        .
        .
FIELD   DC        CL4'1234'
```

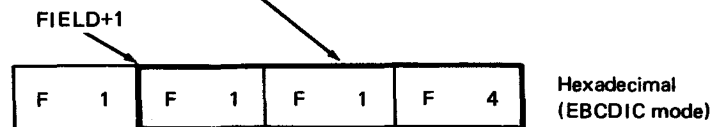
FIELD before execution of MVC instruction:



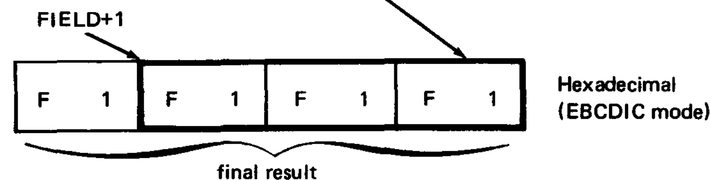
FIELD after byte 1 is moved:



FIELD after byte 2 is moved:



FIELD after byte 3 is moved:



As indicated, F1 is propagated through the entire field. This can be done using any character. If a hexadecimal 40 is used, the resultant field is EBCDIC blanks. If a hexadecimal F0 is used, the resultant field is EBCDIC zeros. To propagate one character through an entire field, the sending field (operand 2) must begin with the first byte of that field and the receiving field (operand 1) must begin with the second byte of that same field. The move is processed from left to right. When operands 1 and 2 overlap, the end result is obtained by processing the operands one byte at a time, and putting each result byte immediately after the byte just obtained.

MVCL

9.10. MOVE CHARACTER LONG (MVCL)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MVCL	**0E**	**RR**	**2**

Condition Codes

- ■ IF OP 1 = OP 2, SET TO 0
- ■ IF OP 1 <OP 2, SET TO 1
- ■ IF OP 1 >OP 2, SET TO 2
- ■ SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

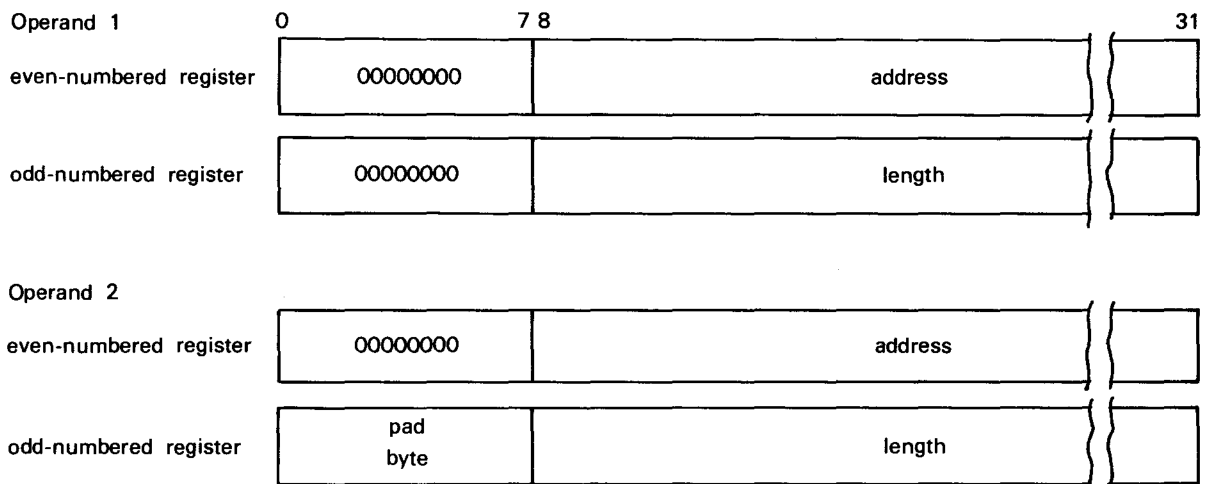
- ■ ADDRESSING
- ☐ DATA (INVALID SIGN/DIGIT)
- ☐ DECIMAL DIVIDE
- ☐ DECIMAL OVERFLOW
- ☐ EXECUTE
- ☐ EXPONENT OVERFLOW
- ☐ EXPONENT UNDERFLOW
- ☐ FIXED-POINT DIVIDE
- ☐ FIXED-POINT OVERFLOW
- ☐ FLOATING-POINT DIVIDE
- ☐ OPERATION
- ■ PROTECTION
- ☐ SIGNIFICANCE
- ■ SPECIFICATION:
  - ☐ NOT A FLOATING-POINT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

The *move character long* (MVCL) instruction moves data from the main storage area specified by operand 2 to the main storage area specified by operand 1. Operands 1 and 2 can have different lengths; where operand 2 is shorter than operand 1, a padding character contained in operand 2 is inserted in all remaining bytes of operand 1.

Explicit and Implicit Formats:

LABEL	△OPERATION△	OPERAND
[symbol]	MVCL	r_1,r_2

The MVCL instruction uses two even-odd register pairs to specify operands 1 and 2. Both have basically the same format: the even-numbered register contains the operand address in bits 8—31, while the odd-numbered register contains the operand length, also in bits 8—31. The operand 2 odd-numbered register differs from the other three registers in that it contains a padding byte in bits 0—7. When operand 1 exceeds operand 2 in length, the padding byte is moved in all remaining low order (rightmost) bytes of operand 1. When the operand 2 length exceeds the operand 1 length, operand 1 determines the number of bytes to be moved. The following chart summarizes register usage with the MVCL instruction:



The MVCL instruction differs from MVC in that it can move more than 256 bytes per instruction. In action, it begins by moving the high order (leftmost) byte of operand 2 to the high order byte of operand 1. Execution continues byte by byte proceeding from left to right. If more than 256 bytes are to be moved, the instruction breaks execution down into units of operation, each unit moving 256 bytes at a time. Interrupts are allowed between units; the MVCL instruction always responds by updating the contents of its registers so that execution can resume exactly where it was interrupted. After execution of the entire instruction is finished, the four registers have the following contents:

- Operand 1 even-numbered register: original operand 1 address incremented by original operand 1 length.
- Operand 1 odd-numbered register: 00000000_{16}.
- Operand 2 even-numbered register: original operand 2 address incremented by number of bytes moved from operand 2 (not including padding bytes).
- Operand 2 odd-numbered register: original operand 2 length decremented by same number of bytes as that added to operand 2 address.

The main storage areas defined by operands 1 and 2 can overlap except for one case: where a byte of main storage is used as an operand 2 source byte after being used as an operand 1 destination byte. This action is called *destructive overlap.* As one of its first actions, the MVCL instruction determines from its operands if destructive overlap is going to occur. If it is, the instruction sets the condition code to 3, moves no data, and terminates.

Operational Considerations:

- Both operands 1 and 2 must be specified as even-numbered registers.
- You can use the MVCL instruction to clear memory. To do so you set both the operand 2 length and padding byte to zero, in effect putting all zeros into the operand 2 odd-numbered register.

- If the destination field contains the MVCL instruction, the processor may attempt to fetch it in mid-execution. If this happens after the instruction has been written over, the results will be unpredictable.
- If you specify an operand 1 length of zero, the MVCL instruction simply sets the condition code and terminates.
- If you specify the same register for r_1 and r_2, the MVCL instruction acts as if you had specified two different register pairs having identical contents. In this case, condition code 0 is set.
- Do not treat the MVCL instruction simply as an extended version of the MVC instruction; certain legal MVC instructions, for example, MVC BYTE+1(2),BYTE, cannot be recoded using MVCL without causing destructive overlap.

Condition Code:

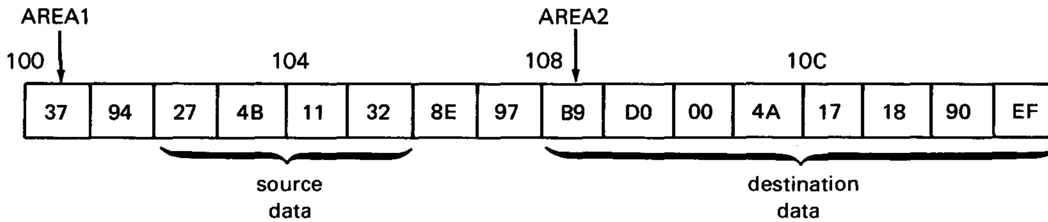
After execution of the MVCL instruction, the condition code is set:

- to 0 if the operand 1 and operand 2 lengths are equal;
- to 1 if the operand 1 length is less than the operand 2 length;
- to 2 if the operand 1 length is greater than the operand 2 length; or
- to 3 if no data movement occurs because of destructive overlap.

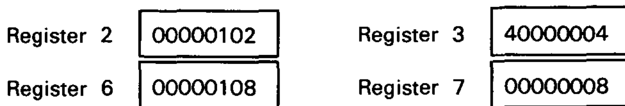
Example:

```
   LABEL    ΔOPERATIONΔ                         OPERAND
   1         10         16

1           LA        2,AREA1+2                 OPERAND 2 ADDRESS
2           LA        3,4                       OPERAND 2 LENGTH
3           LA        6,AREA2                   OPERAND 1 ADDRESS
4           LA        7,8                       OPERAND 1 LENGTH
5           O         3,PADBYTE                 PAD BYTE IN OPERAND 2
6           MVCL      6,2
            .
            .
            .
            DS        OF
   AREA1    DC        XL8'3794274B11328E97'
   AREA2    DC        XL8'B9DØØØ4A171890EF'
   PADBYTE  DC        XL4'4ØØØØØØØ'
```

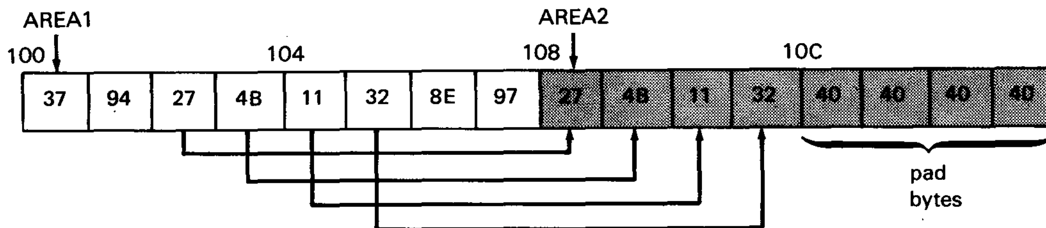
In this example, the MVCL instruction moves data from within the double word at AREA1 to within the double word at AREA2. Assuming that AREA1 is at location 100, the move can be illustrated as follows:



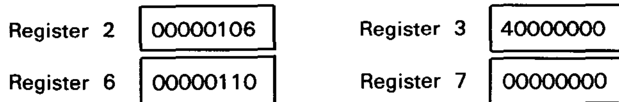
The four LA instructions in lines 1—4 load information into register pairs 2—3 and 6—7, which describe the operands to be used by the MVCL instruction at line 6. The OR instruction at line 5 puts a hexadecimal 40 into the eight high order bits of the operand 2 length register, thus making it the padding character. Before execution of the MVCL instruction the registers contain:



The MVCL instruction at line 6 acts on main storage locations 100—10F as follows:



The MVCL instruction moves four bytes from AREA1+2 to AREA2. Because the destination field is eight bytes long, the remaining four bytes are filled with the pad byte in register 3. After execution of the instruction, the condition code is set to 2 and the registers used contain:



Example:

```
   LABEL    ΔOPERATIONΔ                        OPERAND
   1        10         16

1           LA         4,CSECT1
2           LA         5,2Ø48
3           LA         8,CSECT-1
4           LA         9,Ø
5           MVCL       4,8
```

In this example the MVCL instruction is used to clear a 2048-byte area in main storage starting at CSECT1. After execution of the instructions in lines 1—4, and assuming that CSECT1 is at location X'100', registers 4—5 and 8—9 contain:

Register 4	00000100	Register 5	00000800
Register 8	000000FF	Register 9	00000000

Registers 4 and 5 contain the location and length of the destination field. Register 8 contains the location of the byte immediately preceding CSECT1; in this case it could contain any valid address that does not cause destructive overlap. Register 9 specifies 0 for both the source length and the padding byte. When the MVCL instruction at line 5 is executed, the zero source length causes the instruction immediately to move pad bytes of 0 into CSECT1, CSECT1+1, CSECT1+2, and so on. The destination length of 2048 bytes forces the instruction to move zeros into all 2048 bytes of the destination field, thus clearing it. After execution is finished, the condition code is set to 2 and the registers contain:

Register 4	00000900	Register 5	00000000
Register 8	000000FF	Register 9	00000000

Notice that the operand 2 address in register 8 has not changed. No bytes have been moved from the source area in main storage; rather, the source bytes have come from the pad byte contained in operand 2.

MVN

9.11. MOVE NUMERICS (MVN)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MVN	D1	SS	6

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *move numerics* (MVN) instruction moves the low order four bits (digit portion) of each byte in operand 2 into the corresponding low order four bits of each byte in operand 1. The high order four bits (zone portion) of each byte in operand 1 remain unchanged. This instruction operates from left to right.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVN	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVN	$s_1(l),s_2$

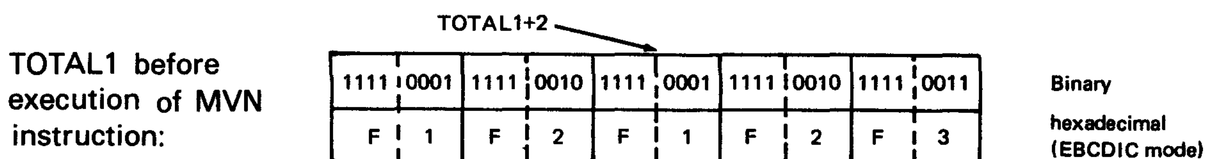
Operational Considerations:

- Any type of data can be specified in both operands 1 and 2.
- The condition code remains unchanged.
- The high order bit positions of each byte in operand 1 remain unchanged.

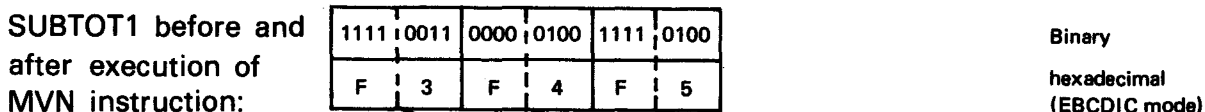
Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10       16
-----------------------------------------------------------
         MVN      TOTAL1+2(3),SUBTOT1
         .
         .
         .
TOTAL1   DC       ZL5'12123'
SUBTOT1  DC       ZL3'345'
```

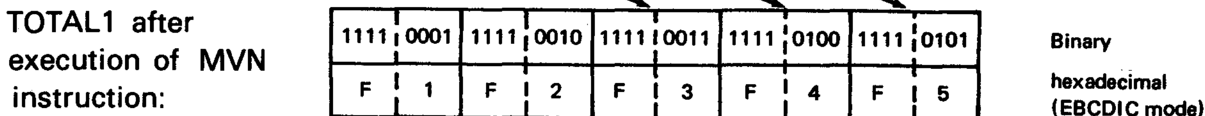
TOTAL1 before execution of MVN instruction:



SUBTOT1 before and after execution of MVN instruction:



TOTAL1 after execution of MVN instruction:

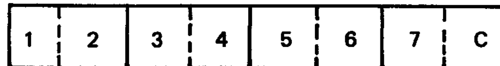


In this example, the low order four bit positions of each byte in SUBTOT1 are moved into the low order four bit positions of bytes 3, 4, and 5 of TOTAL1. The high order bit positions of each byte in TOTAL1 remain unchanged.

Example:

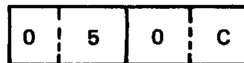
```
1.                AP       RESULT,=P'5Ø'
2.                MVN      RESULT+2(1),RESULT+3
                  MVC      BUFFER,SPACES
3.                MVC      BUFFER(3),RESULT
                  .
                  .
                  .
         RESULT   DC       PL4'1234567'
         BUFFER   DS       CL5
         SPACES   DC       CL5' '
```

RESULT before execution of AP instruction:



Packed decimal number

Operand 2 before and after execution of AP instruction:

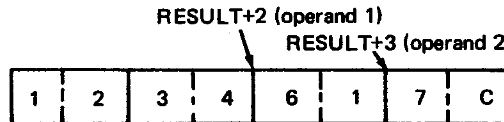


Packed decimal literal

RESULT after execution of AP instruction:

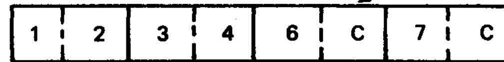
Packed decimal number

RESULT before execution of MVN instruction:



Packed decimal number

RESULT after execution of MVN instruction:

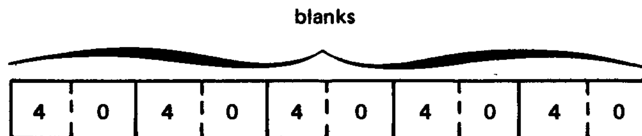


Packed decimal number

In this example, the MVN instruction is used in rounding numbers. In the first line of code, the literal fifty (50) is added to the contents of RESULT to round the number to the first two decimal places.

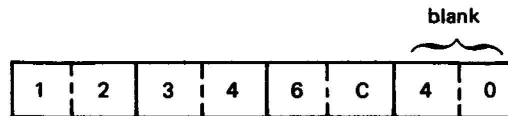
Then, the low order four bits of byte 4 in RESULT are moved to the low order four bits of byte 3 in RESULT. When the MVN instruction is completed, the sign is moved to the right of the first two decimal places that were just rounded. The last byte of RESULT is ignored when the MVC instruction is executed. The location named BUFFER contains the final result.

BUFFER before execution of MVC instruction:



Hexadecimal (EBCDIC mode)

BUFFER after execution of MVC instruction:



Packed decimal number

9.12. MOVE WITH OFFSET (MVO)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MVO	F1	SS	6

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *move with offset* (MVO) instruction moves the contents of operand 2 into operand 1 offsetting the data one half-byte to the left during the move.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVO	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVO	$s_1(l_1),s_2(l_2)$

The MVO instruction operates from right to left. Data from operand 2 (the sending field) is moved into operand 1 (the receiving field) but offset one half-byte to the left. The low order four bits of the rightmost byte in the receiving field remain unchanged. If operand 2 data does not completely fill operand 1, the leftmost unfilled bytes of operand 1 are padded with zeros. However, if the operand 2 field is larger than the operand 1 field, the leftmost bytes of operand 2 are truncated. The MVO instruction is most commonly used in rounding packed decimal numbers to an odd number of digits.

Operational Considerations:

- Usually, the MVO instruction operates on packed decimal fields; however, unpacked fields can be specified.

- Padding of zeros to the left and truncation to the left can occur.
- Condition code remains unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10          16
________________________________________________________________
         MVO         FIELD1,FIELD2
         .
         .
         .
FIELD1   DC          XL4'FFFFFFFF'
FIELD2   DC          XL3'AABBCC'
```

FIELD1 before execution of MVO instruction:

1111	1111	1111	1111	1111	1111	1111	1111	Binary
F	F	F	F	F	F	F	F	hexadecimal characters

FIELD2 before and after execution of MVO instruction:

1010	1010	1011	1011	1100	1100	Binary
A	A	B	B	C	C	hexadecimal characters

FIELD1 after execution of MVO instruction:

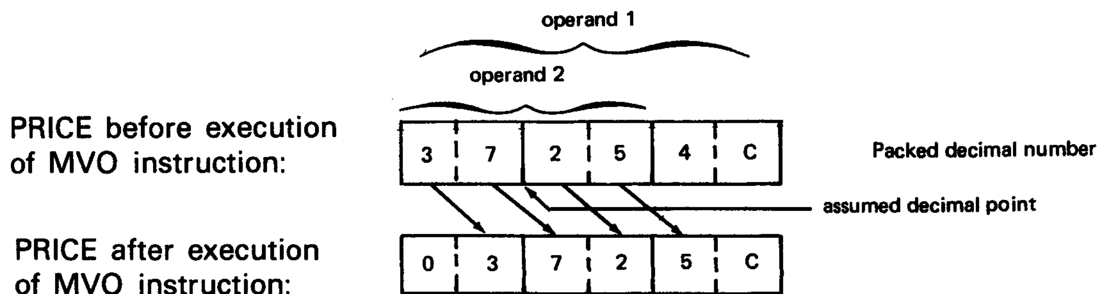
0000	1010	1010	1011	1011	1100	1100	1111	Binary
0	A	A	B	B	C	C	F	hexadecimal characters

In this example, the content of FIELD2 is moved (starting from right to left) into FIELD1 offset by one half-byte to the left. The low order four bits of the rightmost byte of FIELD1 (1111, or hexadecimal F) remain unchanged.

Each half-byte of FIELD2 fills its corresponding half-byte of FIELD1. The high order four bits of the leftmost byte of FIELD1 are padded with binary zeros since the operand 1 field is larger than operand 2.

Example:

```
         MVO         PRICE,PRICE(2)
         .
         .
         .
PRICE    DC          P'37254'
```



In this example, the MVO instruction is used in rounding packed decimal numbers. An explicit length is specified for operand 2 and both operands have overlapping bytes. If all decimal places are needed in the final result, then this rounding technique is not useful. The purpose of this MVO instruction is to move the final result (dollars and cents) next to the sign so that it can be edited and printed. Note the decimal number 4 in the high order four bits of the rightmost byte of operand 1 is replaced with the decimal number 5 and binary zeros are padded in the high order four bits of the leftmost byte of operand 1.

MVZ

9.13. MOVE ZONES (MVZ)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MVZ	D3	SS	6

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *move zones* (MVZ) instruction moves the high order four bits (zone portion) of each byte in operand 2 into the corresponding high order four bits of each byte in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVZ	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVZ	$s_1(l),s_2$

This instruction operates from left to right. The low order four bits of each byte in operand 1 remain unchanged. If the operand 2 field is larger than operand 1, the zone portions of the leftmost bytes of operand 2 are truncated. On the other hand, if the operand 1 field is larger than operand 2, the zone portions of the leftmost bytes in operand 1 remain unchanged.

Operational Considerations:

- The contents of both operands should contain zoned numeric fields; however, any type of data can be specified.

- The low order four bits of each byte in operand 1 remain unchanged.

- Operands 1 and 2 can have overlapping bytes.
- The condition code remains unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10        16
------------------------------------------------------------
         MVZ       NUMPOS,NUMNEG
         .
         .
         .
NUMPOS   DC        ZL3'456'
NUMNEG   DC        XL3'FØFØDØ'
```

NUMPOS before execution of MVZ instruction:

1111	0100	1111	0101	1100	0110
F	4	F	5	C	6

Binary zoned decimal number (positive)

NUMNEG before and after execution of MVZ instruction:

1111	0000	1111	0000	1101	0000
F	0	F	0	D	0

Binary zoned decimal number (negative)

NUMPOS after execution of MVZ instruction:

1111	0100	1111	0101	1101	0110
F	4	F	5	D	6

Binary zoned decimal number (negative)

In this example, the zone portions of each byte in NUMNEG are moved into the corresponding zone portions of each byte in NUMPOS. As a result, the sign is changed from positive to negative by moving a hexadecimal D into the high order four bits of the rightmost byte of NUMPOS. The other two zone portions are replaced with the same hexadecimal value.

MP

9.14. MULTIPLY DECIMAL (MP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MP	**FC**	**SS**	**6**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *multiply decimal* (MP) instruction algebraically multiplies the packed decimal contents of operand 2 (multiplicand) by the packed decimal contents of operand 1 (multiplier) and stores the result (product) in operand 1. The receiving field (operand 1) is filled from right to left.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MP	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MP	$s_1(l_1),s_2(l_2)$

Because the result replaces operand 1, you must ensure that the operand 1 field is large enough to hold the product. This is determined by adding the number of bytes required to hold the multiplicand to the number of bytes required to hold the multiplier.

length of multiplicand	+	length of multiplier	=	length of operand 1 (product)

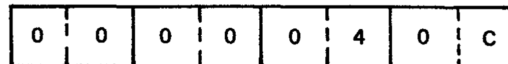
If you use this rule, the multiplicand will have at least as many high order zeros as the number of digits in the multiplier. These high order zeros prevent overflow from occurring in the final result.

The multiplier (operand 2) can be up to 8 bytes long which can consist of 15 digits and a sign. The resultant product (operand 1) can be up to 16 bytes long which can consist of 31 digits and a sign.

Example:

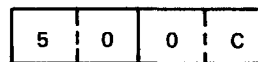
```
LABEL    ΔOPERATIONΔ                      OPERAND
1        10         16

         MP         HOURS,RATE
         .
         .
         .
HOURS    DC         PL4'4Ø'
RATE     DC         PL2'5ØØ'
```

HOURS before execution of MP instruction:



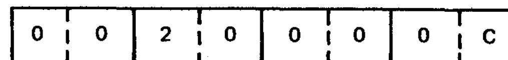
Packed decimal number

RATE before and after execution of MP instruction:



Packed decimal number

HOURS after execution of MP instruction:



Packed decimal number

In this example, the length of operand 1 is determined by adding the number of bytes required to hold the packed decimal 40 (2 bytes) to the number of bytes required to hold the packed decimal 500 (2 bytes) which gives the total of 4 bytes. Then the multiplication operation takes place operating from right to left. The product replaces HOURS and the sign is determined algebraically. (In this example, positive x positive = positive.)

Operational Considerations:

- The operand 1 field must be large enough to hold the product.
- The operand 2 field is limited to 8 bytes in length and the operand 1 field is limited to 16 bytes in length.
- Since a symbolic name references the leftmost or high order portion of a location in main storage, overlapping bytes can exist in the rightmost bytes only.

If overlapping bytes exist in the leftmost portion, the number of bytes required to hold the multiplicand and the multiplier will be larger than the receiving field.

- The condition code remains unchanged.
- Multiplication by powers of 10 adds decimal places to a specified value.

PACK

9.15. PACK DECIMAL (PACK)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
PACK	F2	SS	6

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *pack decimal* (PACK) instruction converts data in operand 2 from unpacked format to packed format. (See 9.1.) The result replaces operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	PACK	d_1 (l_1, b_1), d_2 (l_2, b_2)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	PACK	s_1 (l_1), s_2 (l_2)

Any data that is to be used in decimal arithmetic must be stored in packed decimal format before any arithmetic operations are performed. After your calculations are processed, packed data must be changed back to unpacked (zoned decimal) format to be sent to the printer or any other character sensitive device. Remember, when data is input from an external device (i.e., card reader), the data is stored in zoned decimal format. Operand 2, the sending field, is defined as a character type or zoned type field. Operand 1, the receiving field, is defined as a packed field and should contain enough bytes to receive all digits (plus the sign) from operand 2.

The formula for computing the number of bytes required to receive unpacked operand 2 data is:

$$\frac{\text{(Number of bytes of operand 2)} + 1}{2} = \text{number of bytes required for packed operand 1 field (round upward to the nearest byte)}$$

When the PACK instruction is executed, all zones in operand 2 are ignored except the zone in the rightmost byte. That zone portion (the sign) and the digit portion are reversed and placed in the rightmost byte of operand 1. Each digit in operand 2 is placed in operand 1 next to the rightmost byte, filling in from right to left. Any unfilled bytes or half bytes that are part of the specified length for operand 1 are zero-filled. Any unfilled bytes that are not part of the specified length for operand 1 remain unchanged.

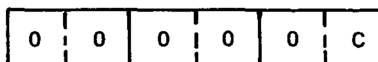
Operational Considerations:

- Operand 2 data should be in zoned decimal format.
- Operand 1 should contain enough bytes to receive all digits (plus the sign) from operand 2.
- This instruction operates from right to left.
- Any unfilled bytes or half bytes that are part of the specified length for operand 1 are zero-filled.
- Any unfilled bytes that are not part of the specified length for operand 1 remain the same.
- Specification of a length attribute for operands 1 and 2 is optional.
- The condition code remains unchanged.

Example:

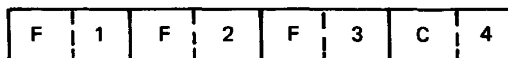
```
LABEL   ΔOPERATIONΔ                     OPERAND
1       10        16
------------------------------------------------------------
        PACK      AMTP(3),AMT(4)
        .
        .
        .
AMT     DC        ZL4'1234'
AMTP    DC        PL3'Ø'
```

AMTP before execution of PACK instruction:



Packed decimal number

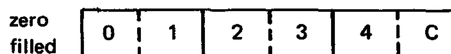
AMT before and after execution of PACK instruction:



Zoned or unpacked decimal number

AMTP after execution of PACK instruction:

zero filled

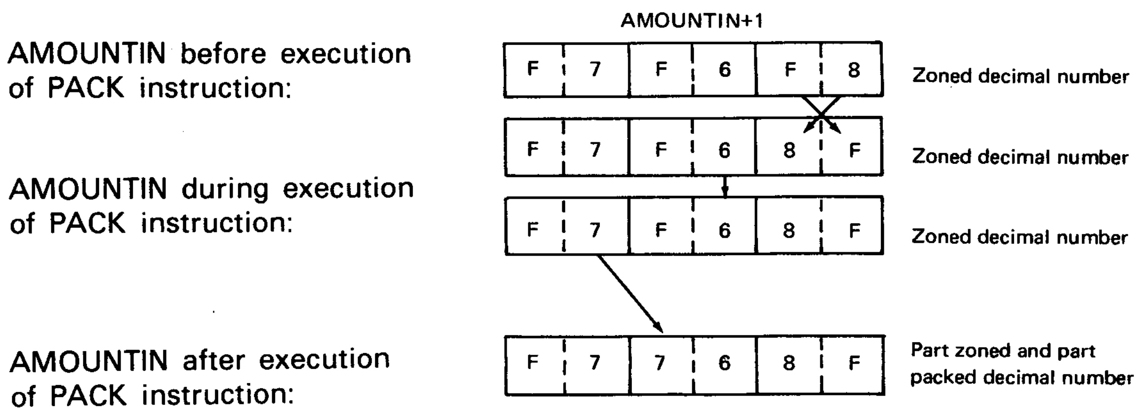


Packed decimal number

In this example, the content of AMT (a 4-byte zoned decimal number) is packed into AMTP (a 3-byte packed field of zeros). The sign and digit portions of the rightmost byte of AMT are reversed and placed in the rightmost byte of AMTP. Then the next digit (3) is placed next to the left of the rightmost byte. Then digit 2 is placed to the left of digit 3 and digit 1 is placed to the left of digit 2. The high order four bits of the leftmost byte are zero-filled. As you can see in this example, two decimal digits occupy a single byte with the exception of the rightmost (sign) byte. Note that a length attribute is specified for both operands in the examples. The length attribute can be omitted but it is suggested it be included for clarity.

Example:

```
LABEL    ΔOPERATIONΔ                  OPERAND
1        10        16
-------------------------------------------------------
         PACK      AMOUNTIN+1(2),AMOUNTIN(3)
         .
         .
         .
AMOUNTIN DC        C'768'
```

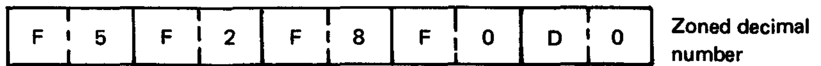


This example shows that the content of AMOUNTIN (a 3-byte zoned decimal field) is packed into part of itself (AMOUNTIN+1, a 2-byte zoned decimal field). The zone portion (F) and digit portion (8) of the rightmost byte of AMOUNTIN are reversed and placed in the rightmost byte of AMOUNTIN+1. The digits 6 and 7 are placed to the left of the rightmost byte, 6 in the low order four bits and 7 in the high order four bits. Because the leftmost byte of AMOUNTIN is not part of the resultant field, that byte remains unchanged. Since AMOUNTIN is now a part zoned, part packed field, you should move the packed decimal number to another field before performing any mathematical calculations. Note that packing a number into itself is not considered good practice since results are often unpredictable.

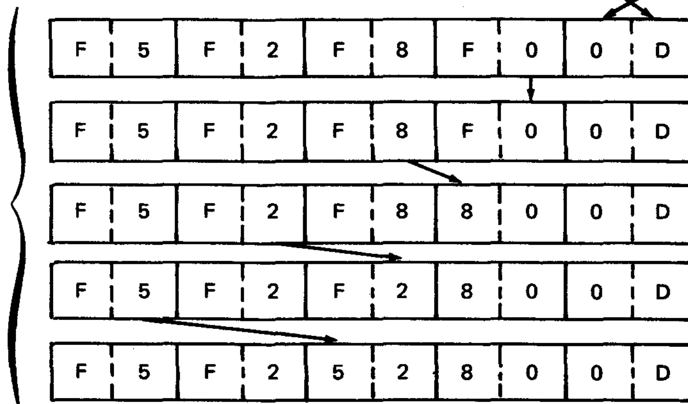
Example:

```
LABEL    ΔOPERATIONΔ                       OPERAND
1        10          16
-----------------------------------------------------------------
         PACK        INVAMT(5),INVAMT(5)
         .
         .
         .
INVAMT   DC          ZL5'-528ØØ'
```

INVAMT before execution of PACK instruction:



INVAMT during execution of PACK instruction:



INVAMT after execution of PACK instruction:

The content of INVAMT (a 5-byte zoned field) is packed into itself. The zone portion (D) and the digit portion (0) of the rightmost byte are reversed and returned to the same byte. The remaining zones are ignored. The remaining digits are placed in INVAMT starting next to the rightmost byte and filling each half-byte from right to left. Because the full length of operand 1 is specified, the remaining unfilled bytes are zero-filled.

SP

9.16. SUBTRACT DECIMAL (SP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SP	FB	SS	6

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *subtract decimal* (SP) instruction algebraically subtracts the packed decimal contents of operand 2 from the packed decimal contents of operand 1 and stores the result in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SP	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SP	$s_1(l_1),s_2(l_2)$

Operand 1 (minuend) and operand 2 (subtrahend) must be in packed decimal format. The operand 1 field should be equal and in most cases larger than the size of operand 2. If operand 1 is too short to contain the result (difference), an overflow condition occurs. Subtraction is algebraic, concerning the signs and digits of both operands. If the sign of operand 2 is negative, it is treated as positive; if positive, it is treated as negative. Then, both operands are added together and the result is placed in operand 1. The sign of the difference is determined by the rules of algebra. If the result is smaller than the operand 1 field, any unfilled leftmost bytes are zero-filled. On the other hand, if the result is larger than the operand 1 field, the leftmost bytes of the result are truncated.

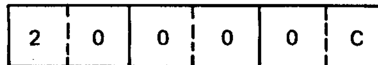
Operational Considerations:

- Operands 1 and 2 must be in packed decimal format.
- The length of operand 1 should be equal to or larger than the length of operand 2.
- Subtraction is algebraic.

Example:

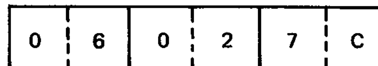
```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10        16
------------------------------------------------------------
         SP        GROSS(3),DEDUCT(3)
         .
         .
         .
GROSS    DC        P'2ØØØØ'
DEDUCT   DC        P'6Ø27'
```

GROSS before execution of SP instruction:



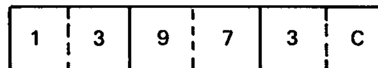
Packed decimal number (assumed decimal point)

DEDUCT before and after execution of SP instruction:



Packed decimal number (assumed decimal point)

GROSS after execution of SP instruction:



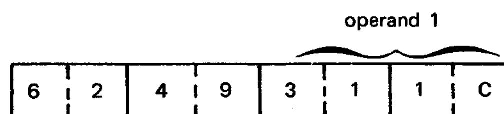
Packed decimal number (assumed decimal point)

In this example, the content of DEDUCT is subtracted from the content of GROSS. The result replaces GROSS and, in this example, completely fills the operand 1 field. The signs of both operands are positive which produces a positive result.

Example:

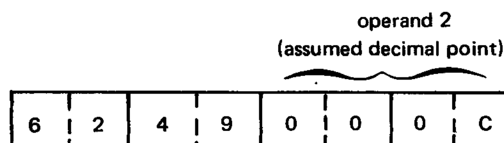
```
         SP        FIELD,FIELD+2(2)
         .
         .
         .
FIELD    DC        P'6249311'
```

FIELD before execution of SP instruction:



Packed decimal number

FIELD after execution of SP instruction:



Packed decimal number

Here is an example of setting the rightmost part of a field to zeros. The contents of FIELD + 2 (a 2-byte field) are subtracted from the contents of FIELD (a 4-byte field). The result replaces the rightmost two bytes of FIELD. The signs of both operands are positive and by the rules of algebra produces a positive result. This instruction operates from right to left. The SP instruction starts with the rightmost bytes of both operands regardless of the differences in length. If you are concerned with whole numbers only, you may want to zero-fill any undesired decimal places. This SP instruction is used as a method to zero-fill any decimal places to the right of the decimal point.

Example:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1        10       16
-------------------------------------------------------
         SP       QTY,ITEMS
         .
         .
         .
QTY      DC       P'-25'
ITEMS    DC       P'12'
```

QTY before execution of SP instruction:

0	2	5	D

Packed decimal number

ITEMS before and after execution of SP instruction:

0	1	2	C

Packed decimal number

QTY after execution of SP instruction:

0	3	7	D

Packed decimal number

The SP instruction subtracts the contents of ITEMS (a 2-byte field) from the contents of QTY (a 2-byte field). The result replaces QTY and a zero fills the leftmost unused half-byte. The signs are different, however. Operand 2 is unsigned and assumed to be positive. Since the sign of operand 2 is positive, it is treated as negative. Now, both operands 1 and 2 are negative and are added together. The sign of the result is negative since the rules of algebra determine that the sign of the operand with the highest absolute value (in addition and subtraction) determines the sign of the result.

9.17. SHIFT AND ROUND DECIMAL (SRP)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SRP	F0	SS	6

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *shift and round decimal* (SRP) instruction shifts a packed decimal number in main storage, specified by operand 1, according to specifications contained in operand 2. For right shifts, the instruction rounds the decimal result according to the byte of immediate data contained in i_3.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRP	$d_1(l_1,b_1),d_2(b_2),i_3$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRP	$s_1(l_1),s_2,i_3$

Operand 1 addresses a packed decimal number, the length (in bytes) of which is specified by 1. The SRP instruction can shift operand 1 left or right. Only the numeric portion of operand 1 participates in the shift; the sign does not change, although a sign digit of F is changed to C by the instruction. The resulting number replaces operand 1; zeros are shifted in to replace vacated digits.

Operand 2 specifies the number of half-bytes (packed decimal digits) to be shifted and the direction of the shift. This information is taken from the low order six bits of operand 2, which are treated together as a signed integer in twos complement form. The integer can range in value from —32 (100000_2) to +31 (011111_2), where a positive number indicates a shift to the left and a negative number a shift to the right. Thus the integer:

000001	(1)

specifies a 1-digit shift to the left while the integer

111101	(—3)

whose twos complement is 3, indicates a 3-digit shift to the right.

Operand 3 contains the rounding factor used during right shifts. The factor is decimally added to the last (or leftmost) digit shifted out of operand 1. Both factor and digit are treated as positive. If this addition generates a carry digit, it is added to the low order digit remaining in operand 1 (not the sign). Depending on the number in operand 1, the carry may in turn generate other carry digits to its left. The rounding factors generally used are: 0 for no rounding, and 5 for rounding. All digits shifted out of operand 1 are lost.

Operational Considerations:

- Operand 1 must be a packed decimal number, the low order half-byte containing a C, D, or F, or else a data exception will result. The sign remains unchanged but the SRP instruction changes a sign digit of F (unsigned positive) to a sign digit of C (signed positive).
- If operand 2 is an explicit address with a base register (b_2) other than 0, the SRP instruction first calculates the address by adding the register contents to the displacement (d_2) then extracts its low order six bits to determine how to shift operand 1. For an explicit address using 0 as its base register, the six bits are taken directly from bits 42—47 of the SRP object code.
- You can specify operand 2 as an explicit address or as a symbol.
- For a right shift, you must specify operand 3 as a self-defining term; the usual values are 0 or 5. For left shifts, operand 3 is ignored.
- For a specified length 1, the object code contains the value 1—1 in bits 8—11. You can specify a maximum length of 16 bytes or 31 packed digits, plus sign.

Condition Code:

After execution of the SRP instruction, the condition code is set:

- to 0 if operand 1 is zero;
- to 1 if operand 1 is less than zero;
- to 2 if operand 1 is greater than zero; or
- to 3 if one or more nonzero digits is shifted out of the high order end of operand 1; this can only occur during a left shift. If the decimal overflow mask bit (bit 37) of the PSW is set to 1, an overflow generates a decimal overflow exception in addition to setting the condition code to 3.

Example:

```
    LABEL   ΔOPERATIONΔ                     OPERAND
    1       10       16

1           LA       9,3
2           SRP      PNUM1(3),Ø(9),Ø
            .
            .
    PNUM1   DC       PL3'41Ø37'
```

In this example, the 3-byte packed decimal field PNUM1 has the value:

PNUM1	41	03	7C

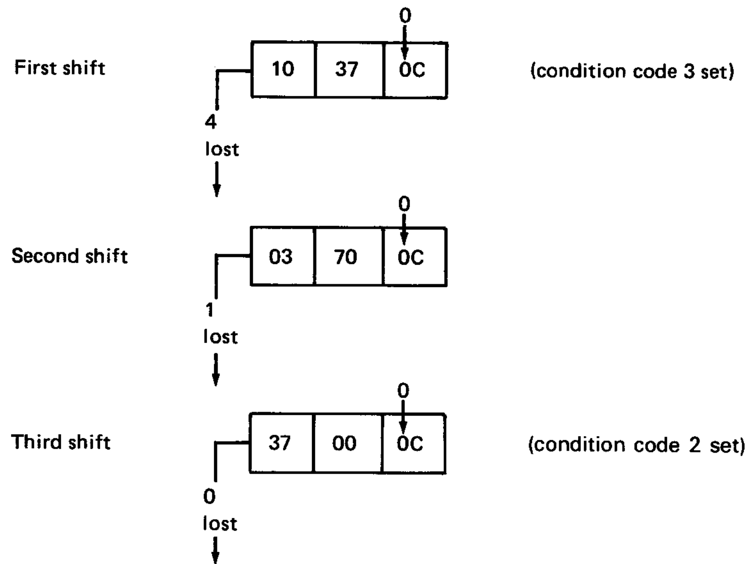
The LA instruction in line 1 puts a value of 3 into register 9. The SRP instruction in line 2 uses operand 2 to form an address of:

Operand 2 address	00000003

of which the low order six bits are:

0	3
00	0011

The value thus obtained is +3 which indicates a left shift of 3 digits. Following PNUM1 through its shifts:

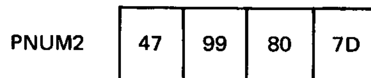


we get the final PNUM1 value of +37000. Notice that the C (positive) sign remains unchanged throughout. Notice also that a decimal overflow sets condition code 3 (assuming here that the decimal overflow exception is prevented) and that a positive result sets condition code 2. Operand 3 plays no part in this shift.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10         16
--------------------------------------------------------
         SRP        PNUM2(4),61,5
         .
         .
PNUM2    DC         PL4'-4799807'
```

In this example, the 4-byte packed decimal field PNUM2 has the value:



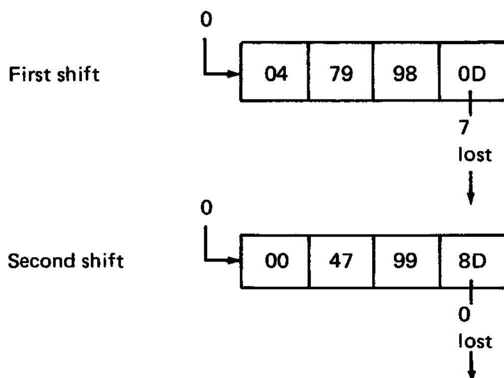
Operand 2 in the SRP instruction yields an effective address of:

0000003D

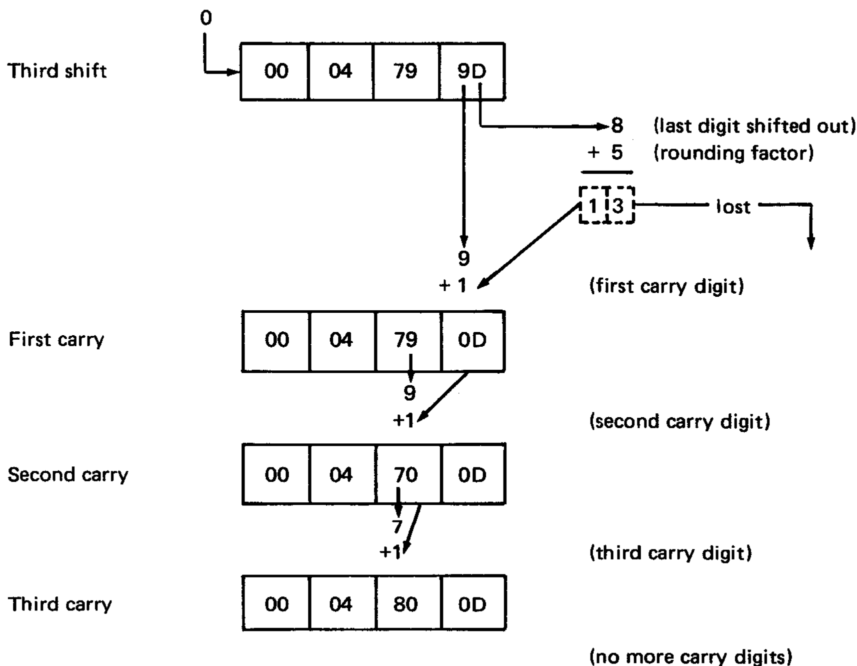
the low order six bits of which are:

3	D
11	1101

The twos complement of this integer is 3 so the value represented is —3, indicating a right shift of 3 digits. First, the SRP instruction performs two digit shifts in which the shifted-out digits are simply lost:



After the third and final shift, the resulting number is rounded according to the rounding factor:



When the final digit (8) is shifted out, it is added to the i_3 rounding factor, 5. The sum of 13 has a carry digit which is added to the low order digit of operand 1. This action generates another carry digit which, when added to the next higher operand 1 digit, generates a third-carry digit. Because that digit, when added to the third operand 1 digit, does not generate another carry digit, the SRP instruction ends there. The condition code is set to 1 to indicate a negative result.

UNPK

9.18. UNPACK DECIMAL (UNPK)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
UNPK	**F3**	**SS**	**6**

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *unpack decimal* (UNPK) instruction converts data in operand 2 from packed format to unpacked (zoned decimal) format. (See 9.1.) The result replaces operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	UNPK	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	UNPK	$s_1(l_1),s_2(l_2)$

Data that is to be printed or sent to any other character-sensitive device must be stored in zoned decimal format. Operand 2, the sending field, is defined as a packed field. Operand 1, the receiving field, is defined as a character type or zoned type field. Operand 1 should contain enough bytes to receive all digits, a zone for each digit, and the sign from operand 2.

The formula for computing the number of bytes required to receive packed operand 2 data is:

(Number of bytes of operand 2) x 2 —1= number of bytes required for unpacked operand 1 field.

The UNPK instruction reverses the zone and digit portion (the sign) of the rightmost byte of operand 2 and places it in the rightmost byte of operand 1. Each half byte of operand 2 is moved to a digit portion and a hexadecimal F (binary 1111) fills each zone portion in operand 1. The move takes place from right to left, consecutively. Any unfilled bytes that are part of the specified length for operand 1 are zero-filled. If the operand 1 field is too short, the leftmost bytes of operand 2 are truncated.

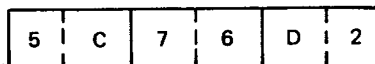
Operational Considerations:

- Operand 2 data should be in packed decimal format.
- Operand 1 should contain enough bytes to receive all digits, a zone for each digit, and the sign from operand 2.
- This instruction operates from right to left.
- Any unfilled bytes that are part of the specified length for operand 1 are zero-filled.
- Specification of a length attribute for operands 1 and 2 is optional.
- The condition code remains unchanged.

Example:

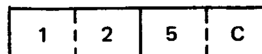
```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10        16
-----------------------------------------------------------------
         UNPK      TOTALU(3),TOTALP(2)
         .
         .
         .
TOTALU   DS        CL3
TOTALP   DC        P'125'
```

TOTALU before execution of UNPK instruction:



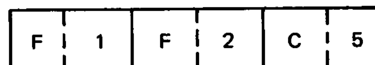
Leftover data from a previous page

TOTALP before and after execution of UNPK instruction:



Packed decimal number

TOTALU after execution of UNPK instruction:



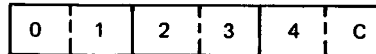
Unpacked decimal number

The UNPK instruction changes the packed format of TOTALP to unpacked format and places the result in TOTALU. The zone (5) and digit (C) portions of TOTALP are reversed and placed in the rightmost byte of TOTALU. The digit 2 fills the digit portion and a hexadecimal F fills the zone portion next to the rightmost byte. Then the digit 1 fills the digit portion and a hexadecimal F fills the zone portion to the left of the byte just filled. As you can see, the field requiring 2 bytes to store the original packed data now requires 3 bytes to store the same data but in unpacked format. Note that a length attribute is specified for both operands, although it can be omitted.

Example:

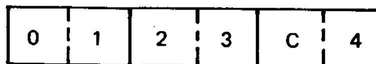
```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10        16
-----------------------------------------------------------
          UNPK      TOTAMT,TOTAMT
           .
           .
           .
TOTAMT    DC        P'1234'
```

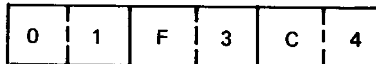
TOTAMT before execution of UNPK instruction:



Packed decimal number

TOTAMT during execution of UNPK instruction:





TOTAMT after execution of UNPK instruction:

The UNPK instruction reverses the zone (4) and digit (C) portions and returns it to the same byte. The next half byte (3) replaces the digit portion and a hexadecimal F fills in the zone portion next to the half byte just filled. No length attributes are specified, so the implied lengths are used. As you can see, the result received is not the result expected. So, remember that unpacking a number into itself is not considered good practice because some results are often unpredictable.

ZAP

9.19. ZERO AND ADD DECIMAL (ZAP)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ZAP	F8	SS	6

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *zero and add decimal* (ZAP) instruction moves a packed field of zeros into operand 1 and then adds the packed contents of operand 2 to the packed field of zeros in operand 1. The result replaces operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ZAP	$d_1(l_1,b_1),d_2(l_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ZAP	$s_1(l_1),s_2(l_2)$

This instruction operates in the same manner as the add decimal (AP) instruction except that a packed field of zeros is moved into operand 1 before the addition occurs. The sign of the packed field of zeros is positive. After the addition takes place the resultant sign is the same as operand 2. If operand 2 does not have a valid sign in the low order four bits, a data exception occurs. If an overflow condition occurs and the leftmost bytes are truncated, a zero result still has the sign of operand 2. In effect, the ZAP instruction replaces operand 1 with the contents of operand 2. The length of operand 1 should be the same as, or larger than, the length of operand 2. If the operand 1 field is not sufficient to receive all of operand 2, an overflow condition occurs. Operands 1 and 2 can have overlapping bytes when the rightmost byte of operand 1 coincides with, or is to the right of, the rightmost byte of operand 2, provided a valid sign exists in the low order four bits of operand 2.

Operational Considerations:

- Operand 2 must be in packed decimal format.
- If the length of operand 2 is larger than the length of operand 1, the leftmost digits of operand 2 are truncated.
- If the length of operand 1 is larger than operand 2, the leftmost digits of operand 1 are zero-filled.
- Operand 2 must have a valid sign in the low order four bits.

Example:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1        10         16
--------------------------------------------------
         ZAP        TOTAMT,YTDAMT
          .
          .
          .
TOTAMT   DC         P'528416'
YTDAMT   DC         P'215'
```

TOTAMT before execution of ZAP instruction:

0	5	2	8	4	1	6	C

Packed decimal number

YTDAMT before and after execution of ZAP instruction:

2	1	5	C

Packed decimal number

TOTAMT after execution of ZAP instruction:

0	0	0	0	2	1	5	C

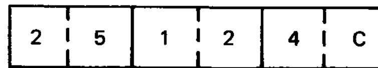
Packed decimal number

In this example, the ZAP instruction moves a packed field of zeros into TOTAMT and then adds the contents of YTDAMT to TOTAMT. As you can see, the contents of YTDAMT now replaces the contents of TOTAMT. In this sample program, TOTAMT contains a year's total amount of sales, while YTDAMT contains the accumulative amount of sales. At the end of 12 months, when the maximum amount of sales for the year is reached, TOTAMT must be cleared to zero, so that the amount of sales for the first month of the next year can be accumulated.

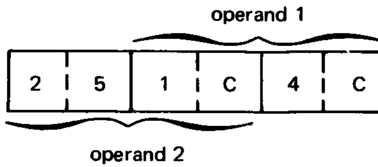
Example:

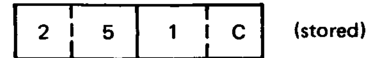
```
         MVC        CALC+1(1),=P'1'
         ZAP        CALC+1(2),CALC(2)
          .
          .
          .
CALC     DC         P'25124'
```

CALC before execution of MVC instruction:

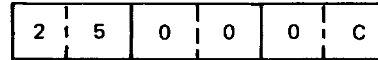


CALC after execution of MVC instruction:

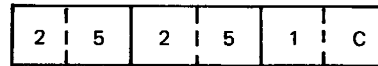




CALC during execution of ZAP instruction:



CALC after execution of ZAP instruction:



In this example, operands 1 and 2 have one overlapping byte. The rightmost byte of CALC+1(2) (operand 1) is to the right of the rightmost byte of CALC(2) (operand 2). When the ZAP instruction is executed, a packed field of zeros with a positive sign is moved into operand 1. Then, the contents of operand 2 that has been saved prior to the execution of the ZAP instruction is now added to the packed field of zeros. In effect, the contents of operand 2 now replace the contents of operand 1.

10. Fixed-Point Binary Instructions

10.1. USE OF FIXED-POINT BINARY INSTRUCTIONS

If the fixed-point binary instruction set (RX, RR, or RS) is compared to the decimal instruction set (SS or SI), you will discover that the difference between storage-to-storage type instructions and register type instructions is the location of the instruction operands. Both operands for decimal instructions are contained in main storage, while the operands for fixed-point instructions are either both in the processor or one in the processor and one in main storage. Any instruction operands located in main storage are transferred to the processor before execution. In fixed-point binary instructions, the RR type requires no transfer of operands, while the RX and RS types require transfer of only one. In decimal instructions, both operands are always transferred. This explains why execution time of fixed-point binary instructions is faster than execution time of decimal instructions.

Execution time gained by arithmetic binary instructions over decimal instructions is lost, however, in the data conversion process. Both instruction sets must convert card input data in zoned decimal format (EBCDIC) to a data format acceptable to the instruction set. Decimal instruction input data must be converted to packed decimal format; fixed-point binary instruction input data must be converted to binary format.

Conversion to packed format is faster than conversion to fixed-point binary format because binary conversion requires an additional instruction that has a slower execution time. To get input data into packed format, you use the PACK instruction; to output packed data, you must first unpack it with the UNPK or ED instruction. When converting input data to binary, the data must be packed first, then converted to binary (using the convert to binary (CVB) instruction). On output, data must be converted to packed decimal (using the convert to decimal (CVD) instruction) and then converted to unpacked or zoned decimal format. For input conversion, fixed-point binary instructions execute slower than decimal instructions. For a comparison of the execution times for decimal and fixed-point instructions, see the system hardware and software summary.

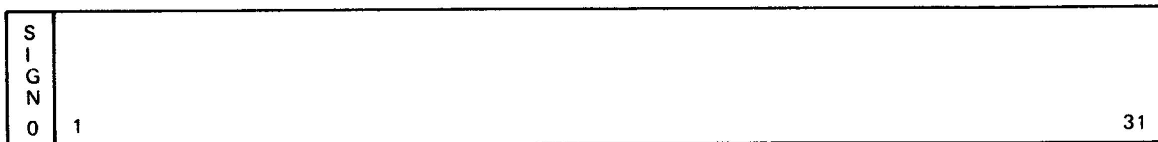
Fixed-point instructions should be used in programs having less input data and more arithmetic calculations, whereas decimal instructions should be used in programs having more input data and less arithmetic calculations. This is why binary instructions are used in the design of FORTRAN compilers and decimal instructions are used in the design of COBOL compilers.

There are 16 general registers located inside the processor that can be used as operands in fixed-point instructions. A decimal number (0 through 15) is used to reference a register. This is similar to using labels when referencing main storage locations.

For all fixed-point instructions, operand 1 always references a register with the exception of the add immediate (AI) instruction, whose operands both reference main storage locations. The operand 1 register is usually the receiving field or resultant field after an instruction is executed. For the store (ST) and convert to decimal (CVD) instructions, however, operand 2 (a main storage location) is the resultant field. In fixed-point instructions, operand 2 references either a register (RR), or a main storage location (RX or RS). The compare instructions do not have a resultant field, since they test already existing conditions and have no effect on operands 1 and 2.

To bring data from main storage into a register, it must be compatible with the structure of the register. A register is four bytes in length and uses 32 binary bits to represent a signed binary number. The high order bit position represents the sign. A binary 1 in the high order bit position represents a negative number, whereas a binary 0 in the high order bit position represents a positive number.

REGISTER (4 bytes)



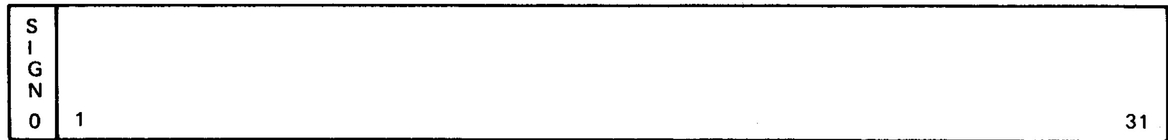
There are two ways to create data in fixed-point binary format:

1. Use the convert to binary (CVB) instruction to convert a packed decimal number to a fixed-point binary number which is placed in a register.

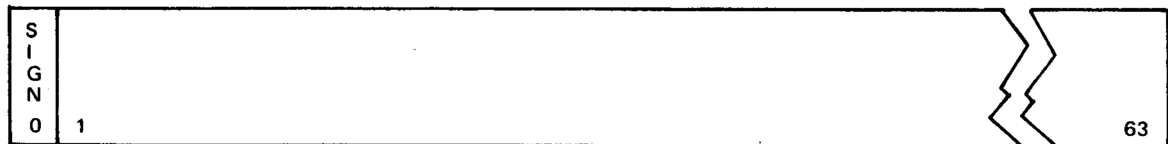
2. Use the define constant (DC) statement to create a constant that is defined as a half word, full word, or double word, or a constant that is aligned on a half-word, full-word, or double-word boundary. This constant is then placed in a register through execution of another instruction [i.e., Add (A), Load (L), Subtract (S)].

HALF WORD (2 bytes)

FULL WORD (4 bytes)



DOUBLE WORD (8 bytes)



As shown, these formats are compatible with the formats of registers. Since registers are full words (4 bytes), only full words or half words in main storage (or other registers) can be specified as operand 2. When a half word value in main storage is specified as operand 2, a full 4 bytes are used when that instruction is executed. Operand 2 is expanded to 32 bits by propagating the sign bit value through the 16 high order bit positions. Expansion occurs after the operand is obtained and before insertion, comparison, or any mathematical calculations are performed with the register.

10.1.1. Half-Word Fixed-Point Constants

The character H is the definition type used for defining half-word fixed-point constants in main storage. The constants associated with this definition type must be enclosed within apostrophes, cannot exceed more than five decimal digits, and cannot have a value greater than +32767 (2^{15}—1) or less than —32768 (—2^{15}). Half-word constants are two bytes in length and aligned on a half-word boundary. If the constant specified does not occupy the full two bytes, it is right-justified and the high order unused bits are filled with the sign bit. Duplication factors can be used and the nominal value can be a signed or unsigned decimal number. Because the length of a half word is always two bytes, no length factor is required. If a length factor is specified, half-word boundary alignment is ignored and the specified length is allocated.

Example:

```
LOC.    OBJECT CODE                         LINE        SOURCE  STATEMENT

000002 39                                   4 PLUS1     DC      HL1'+57'
000003 00
000004 0039                                 5 PLUS2     DC      H'57'
000006 C7                                   6 NEG1      DC      HL1'-57'
000007 00
000008 FFC7                                 7 NEG2      DC      H'-57'
```

10.1.2. Full-Word Fixed-Point Constants

The character F is the definition type used for defining full-word fixed-point constants in main storage. The constant associated with this definition type must be enclosed within apostrophes, cannot exceed more than 10 decimal digits, and cannot have a value greater than +2,147,483,647 ($2^{31}-1$) or less than —2,147,483,648 (-2^{31}). Full-word constants are four bytes in length and aligned on a full-word boundary. If the constant specified does not occupy the full four bytes, it is right-justified and leftmost unused bits are filled with the sign bit. Duplication factors can be used and the nominal value can be a signed or unsigned decimal number. Because the length of a full word is always four bytes, no length factor is required. If a length factor is specified, full-word boundary alignment is ignored and the specified length is allocated.

Example:

```
LOC.    OBJECT CODE                 LINE       SOURCE STATEMENT

00000A 010F                          8 PLUS3    DC     FL2'+271'
00000C 0000010F                      9 PLUS4    DC     F'271'
000010 FEF1                         10 NEG3     DC     FL2'-271'
000012 0000
000014 FFFFFEF1                     11 NEG4     DC     F'-271'
```

10.1.3. Address Constants

Address constants are storage addresses that are stored as constants by using DC statements. Address constants are used to initialize base registers; thereby, providing communication between control sections of a multisection program. Unlike other types of constants, an address constant is enclosed within parentheses. If more than one address constant is specified, they are separated by commas, and the entire sequence is enclosed within parentheses. There are two types of address constants: half word (Y) and full word (A).

10.1.3.1. Full-Word Address Constants

This constant can be specified as an absolute, relocatable, or complex relocatable expression. It has a length of four bytes and is full-word boundary aligned. You cannot specify a value greater than +2,147,483,647 ($2^{31}-1$) or less than —2,147,483,648 (-2^{31}). To generate full-word address constants, use the DC statements with the character A as the definition type and the expressions specified enclosed within parentheses. You can also generate full-word address constants as literals. The address of these expressions are stored in consecutive full words in main storage. However, if a length factor is specified, full-word boundary alignment is ignored and the specified length is allocated.

Example:

```
LOC.    OBJECT CODE         ADDR1 ADDR2       LINE       SOURCE STATEMENT

000002                                          5 TAG      DS     CL3
000005                                          6 HOURS    DS     CL2
000007                                          7 RATE     DS     CL4
00000B                                          8 PAY      DS     CL3
00000E                                          9 LABEL    DS     CL6
000014                                         10 TAG1     DS     CL3
000017                                         11 BUF      DS     CL4
00001B                                         12 TAG2     DS     CL3
00001E                                         13 ADLIST   DS     CL20
000032 0000
000034 0000006A                                14 ADCON1   DC     A(106,TAG,*+526)
000038 00000002
00003C 0000024A
000040
000040 00000007                                15 ADCON2   DC     A(RATE,HOURS,PAY)
000044 00000005
000048 0000000B
00004C
00004C 9858 3056                    00058      16 ADCON3   LM     5,8,=A(88,LABEL,TAG1-TAG2,BUF+641
000050 0000001E                                17 ADCON4   DC     A(ADLIST)
```

10.1.3.2. Half-Word Address Constants

This constant can be specified as an absolute, relocatable, or complex relocatable expression. It has a length of two bytes and is half-word boundary aligned. You cannot specify a value greater than +32767 (2^{15}—1) or less than —32768 (—2^{15}). To generate half-word address constants, use the DC statements with the character Y as the definition type and the expressions specified enclosed within parentheses. You can also generate half-word address constants as literals. The addresses of these expressions are stored in consecutive half words in main storage. However, if a length factor is specified, half-word boundary alignment is ignored and the specified length is allocated.

Example:

```
LOC.    OBJECT CODE                      LINE       SOURCE STATEMENT

000002 500C                                5 VALUE    DC     PL2'500'
000004 00000001                            6 NUM      DC     F'1'
000008 F3F6F2                              7 POS      DC     X'F3F6F2'
00000B 60                                  8 NEG      DC     CL1'-8'
00000C F6F2F4F7C0                          9 ZONE     DC     ZL5'62470'
000011 00
000012 000B                               10 ADCONY1  DC     Y(NEG,POS)
000014 0008
000016
000016 0116                               11 ADCONY2  DC     Y(*+256,600)
000018 0258
00001A 0008                               12 ADCONY3  DC     Y(VALUE+6)
00001C
00001C 0019000A                           13 ADCONY4  DC     Y(25,ZONE-VALUE,NUM,POS+4)
000020 0004
000022 000C
000024
```

10.1.4. Representation of Positive and Negative Fixed-Point Binary Numbers

Binary ones and zeros, with relation to their positions in a string of bits, represent values expressed in powers of two (see Appendix C.3). The powers of two increase from right to left (Figure 10—1). A zero (0) bit indicates no value and a one (1) bit indicates that a value exists. By adding all the powers of two that correspond to one bits, you can determine the decimal equivalence for a positive binary number. A zero bit in the high order bit or any unused high order bits signify a positive binary number.

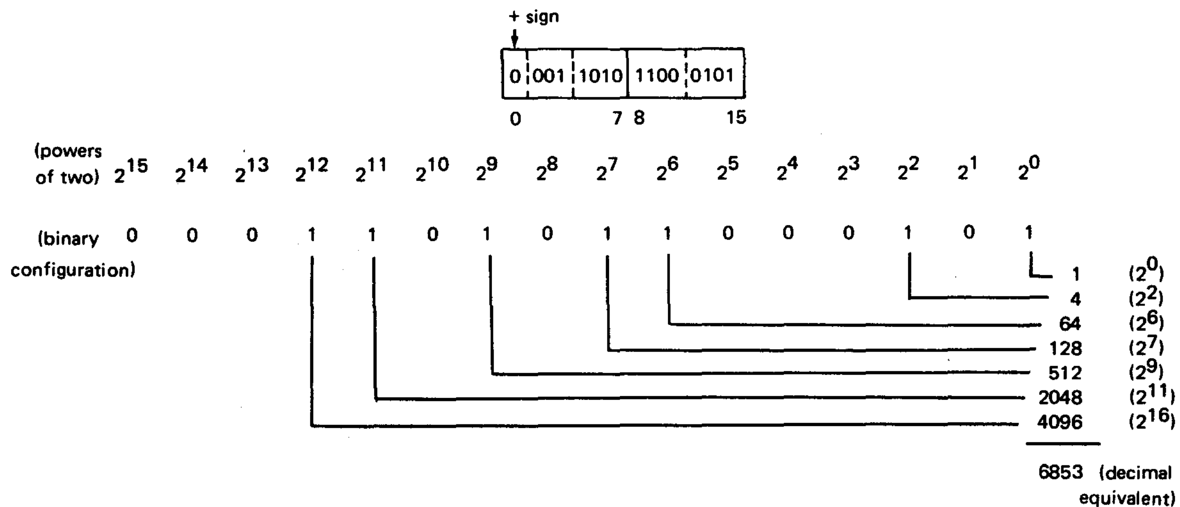


Figure 10—1. Comparison of Binary Numbers and Values Expressed in Powers of 2

Negative binary numbers are indicated by a one bit in the high order bit position or any unused high order bit positions. The remaining portion contains the negative binary number but in twos complement form. To change a positive binary number into twos complement form:

- reverse the bits; and
- add one to the rightmost or low order bit position:

0001101011000101	positive binary number (decimal + 6,853)
1110010100111010 +1	reversed bits add 1
1110010100111011	binary number in twos complement form (decimal — 6,853)

A

10.2. ADD (A)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
A	5A	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
■ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *add* (A) instruction algebraically adds the full-word main storage contents of operand 2 to the contents in the operand 1 register and stores the sum in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	A	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	A	$r_1,s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must either be defined as a full word or aligned on a full-word boundary.
- If the sum exceeds 31 bit positions, an overflow condition occurs.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10          16
-------------------------------------------------------------
         SR          6,6
         A           6,FULLWORD
         .
         .
         .
FULLWORD DC          F'+271'
```

Register 6 before execution of A instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	0	0	hex

FULLWORD before and after execution of A instruction:

0000	0000	0000	0000	0000	0001	0000	1111	binary
0	0	0	0	0	1	0	F	hex

Register 6 after execution of A instruction:

0000	0000	0000	0000	0000	0001	0000	1111	binary
0	0	0	0	0	1	0	F	hex

In this example, the SR instruction subtracts the content of register 6 from itself, clearing it to zero. Then the content of FULLWORD is added to the content of register 6. The result replaces the content of the operand 1 register.

AR

10.3. ADD (AR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AR	**1A**	**RR**	**2**

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
■ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *add* (AR) instruction algebraically adds the contents of the operand 2 register to the contents of the operand 1 register and stores the sum in operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AR	r_1,r_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- If the sum exceeds 31 bit positions, an overflow condition occurs.

Example:

```
LABEL   △OPERATION△                  OPERAND
1         10        16
--------------------------------------------------
          L         5,NUM1
          L         6,NUM2
          AR        5,6
          .
          .
          .
NUM1      DC        F'22'
NUM2      DC        F'16'
```

Register 5 before execution of AR instruction:

0000	0000	0000	0000	0000	0000	0001	0110	binary
0	0	0	0	0	0	1	6	hex

Register 6 before and after AR instruction:

0000	0000	0000	0000	0000	0000	0001	0000	binary
0	0	0	0	0	0	1	0	hex

Register 5 after execution of AR instruction:

0000	0000	0000	0000	0000	0000	0010	0110	binary
0	0	0	0	0	0	2	6	hex

In this example, the contents of NUM1 is loaded into register 5 and the contents of NUM2 is loaded into register 6. Then, the contents of register 6 is added to the contents of register 5. The result is placed in register 5 (operand 1). Notice that both NUM1 and NUM2 are full words.

10.4. ADD HALF WORD (AH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AH	4A	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
■ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *add half-word* (AH) instruction algebraically adds the contents of operand 2 to the contents of the operand 1 register and puts the sum in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AH	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AH	$r_1,s_2(x_2)$

Operand 2 is two bytes in length (16-bit signed integer) and is located in main storage. Before operand 2 is added to the operand 1 register, operand 2 is temporarily expanded to 32 bits by propagating the sign bit through the high order 16 bit positions. Then all 32 bits of operand 2 are added to the 32 bits of operand 1. The result is placed in operand 1. If the sum exceeds 31 bits, an overflow condition occurs.

Operational Considerations:

- Operand 2 must be either defined as a half word or half-word boundary aligned.
- Any of the general registers (0 through 15) can be used as operand 1.
- A fixed-point overflow condition can occur.

Example:

```
LABEL     ΔOPERATIONΔ                          OPERAND
1         10        16
--------------------------------------------------------
          L         5,FULLWORD
          AH        5,HALFWORD
          .
          .
          .
FULLWORD  DC        F'32'
HALFWORD  DC        H'16'
```

Register 5 before execution of AH instruction:

0000	0000	0000	0000	0000	0000	0010	0000	binary
0	0	0	0	0	0	2	0	hex

HALFWORD before and after execution of AH instruction:

before expanding to 32 bits

0000	0000	0000	0000	0000	0000	0001	0000	binary
0	0	0	0	0	0	1	0	hex

after expanding to 32 bits

Register 5 after execution of AH instruction:

0000	0000	0000	0000	0000	0000	0011	0000	binary
0	0	0	0	0	0	3	0	hex

In this example, the content of FULLWORD is loaded into register 5. Then the content of HALFWORD is added to the content of register 5. The result is placed in register 5 (operand 1). If the sum exceeds 31 bits, an overflow condition occurs.

10.5. ADD IMMEDIATE (AI)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AI	9A	SI	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *add immediate* (AI) instruction algebraically adds the 1-byte immediate data in operand 2 to the half word value in operand 1. The sum is placed in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AI	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AI	s_1,i_2

Operand 1 must be either defined as a half word or is half-word boundary aligned. Operand 2 must be a 1-byte self-defining term. Before operand 2 is added to the half word in operand 1, operand 2 is temporarily expanded to 16 bits by propagating the sign bit through the high order 8 bit positions. Then all 16 bits in operand 2 are added to the 16 bits in operand 1. The result is placed in operand 1. If the sum exceeds 15 bit positions, an overflow condition occurs.

Operational Considerations:

- Operand 1 must be either defined as a half word or aligned on a half-word boundary.
- During execution of the AI instruction, operand 2 is temporarily expanded to 16 bit positions. The leftmost eight bits are the same as the sign bit.
- Operand 2 must be a 1-byte, self-defining term (see 4.4).
- You may not specify an immediate value greater than +127 (2^7—1) or less than —128 (—2^7) in operand 2.
- If the sum exceeds 15 bit positions, an overflow condition can occur.

Example:

```
LABEL    ΔOPERATIONΔ          OPERAND
1        10          16
-----------------------------------------------
         AI          STORAGE,1
         .
         .
         .
STORAGE  DC          H'3'
```

STORAGE before execution of AI instruction:

0000	0000	0000	0011	binary
0	0	0	3	hex

Operand 2 immediate before and after execution of AI instruction:

0000	0001	binary
0	1	hex

STORAGE after execution of AI instruction:

0000	0000	0000	0100	binary
0	0	0	4	hex

In this example, the immediate value in operand 2 is added to the half-word value in STORAGE. The result replaces the contents of STORAGE.

10.6. COMPARE (C)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
C	59	RX	4

Condition Codes

- ■ IF r_1 = OPERAND 2, SET TO 0
- ■ IF r_1 < OPERAND 2, SET TO 1
- ■ IF r_1 > OPERAND 2, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *compare* (C) instruction algebraically compares the contents in the operand 1 register to the full word in operand 2. The result determines the setting of the condition code. (See condition code settings, 8.4.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	C	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	C	$r_1,s_2(x_2)$

The operand 1 register is compared to a 32-bit signed integer (operand 2) located on a full-word boundary in main storage. The result of the comparison determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

If operand 1 = operand 2, set to 0.

If operand 1 < operand 2, set to 1.

If operand 1 > operand 2, set to 2.

Usually, a conditional branch instruction tests the resulting condition code for an equal to zero, less than zero, or greater than zero condition. If the condition is met, a branch takes place. If not, the program continues processing as shown in the following coding instruction.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a full word or aligned on a full-word boundary.
- Neither operand is changed by the execution of the instruction.

Example:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1        10       16
---------------------------------------------------------
         SR       7,7
         L        5,AMOUNT
         C        5,FULLWORD
         BE       ROUTINE
         A        6,=F'1'
         .
         .
         .
ROUTINE  AR       7,5
         .
         .
         .
FULLWORD DC       F'32'
AMOUNT   DC       F'32'
```

Register 5 before and after execution of C instruction:

0000	0000	0000	0000	0000	0000	0010	0000	binary
0	0	0	0	0	0	2	0	hex

FULLWORD before and after execution of C instruction:

0000	0000	0000	0000	0000	0000	0010	0000	binary
0	0	0	0	0	0	2	0	hex

In this example, the full word in AMOUNT is loaded into register 5. Then, the content of register 5 is compared to the full word in FULLWORD. Since they compare equally, the condition code is set 0 and a branch to the instruction labeled ROUTINE takes place. If they do not compare equally, the A instruction following the BE instruction is executed and the program continues processing.

10.7. COMPARE (CR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CR	19	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
□ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *compare* (CR) instruction algebraically compares the contents in the operand 1 register to the contents in the operand 2 register. The result determines the setting of the condition code. (See condition code settings, 8.4.)

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CR	r_1, r_2

The 32 bits of operand 1 are compared to the 32 bits of operand 2. The result determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

If operand 1 = operand 2, set to 0.

If operand 1 < operand 2, set to 1.

If operand 1 > operand 2, set to 2.

Usually, a conditional branch instruction tests the resulting condition code for an equal to, less than, or greater than condition. If the condition is met, a branch takes place accordingly. If not, the program continues processing as shown in the following coding instruction.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- Neither operand is changed by the instruction.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10       16
-----------------------------------------------------------------
          SR       7,7
          L        5,AMOUNT
          L        6,VALUE
          CR       5,6
          BH       ROUTINE
          AR       7,5
          .
          .
          .
ROUTINE   AI       ERCNT,1
          .
          .
          .
AMOUNT    DC       F'32'
VALUE     DC       F'32'
ERCNT     DC       H'Ø'
```

Register 5 before and after execution of CR instruction:

0000	0000	0000	0000	0000	0000	0010	0000	binary
0	0	0	0	0	0	2	0	hex

Register 6 before and after execution of CR instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	2	0	hex

In this example, the full word in AMOUNT is loaded into register 5 and the full word in VALUE is loaded into register 6. Then the content of register 5 is compared to the content of register 6. Since they compare equally, the condition code is set to 0. The next branch instruction (BH) tests for a greater than (high) condition. Since both registers compare equally, no branch is taken and the instruction following the BH instruction (AR) is executed and the program continues processing.

CH

10.8. COMPARE HALF WORD (CH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CH	49	RX	4

Condition Codes

- ■ IF r_1 = OPERAND 2, SET TO 0
- ■ IF r_1 < OPERAND 2, SET TO 1
- ■ IF r_1 > OPERAND 2, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *compare half word* (CH) instruction algebraically compares the contents in the operand 1 register to the half word in operand 2. The result of the comparison determines the setting of the condition code.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CH	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CH	$r_1,s_2(x_2)$

Operand 2 is two bytes in length (16-bit signed integer) and is located in main storage. Before operand 2 is compared with the operand 1 register, operand 2 is temporarily expanded to 32 bits by propagating the sign bit through the high order 16 bit positions. Then all 32 bits of operand 1 are compared to the 32 bits in operand 2. The result determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

If operand 1 = operand 2, set to 0.

If operand 1 < operand 2, set to 1.

If operand 1 > operand 2, set to 2.

Usually, a conditional branch instruction tests the condition code for an equal to, less than, or greater than condition. If the condition is met, a branch takes place accordingly. If not, the program continues processing as shown in the following coding instruction.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a half word or aligned on a half-word boundary.
- Neither operand is permanently changed by the execution of the instruction.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10         16
-----------------------------------------------------------
          L          5,AMOUNT
          CH         5,HALFWORD
          BH         ROUTINE
          AR         8,5
ROUTINE   A          6,=F'1'
          .
          .
          .
AMOUNT    DC         F'32'
HALFWORD  DC         H'16'
```

Register 5 before and after executicn of CH instruction:

0000	0000	0000	0000	0000	0000	0010	0000	binary
0	0	0	0	0	0	2	0	hex

HALFWORD before and after execution of CH instruction:

operand 2 before expansion

0000	0000	0000	0000	0000	0000	0001	0000	binary
0	0	0	0	0	0	1	0	hex

operand 2 after expansion

In this example, the full word in AMOUNT is loaded into register 5. Then, the content of register 5 is compared to the half word in HALFWORD. Since the content of register 5 is greater than the content of HALFWORD, the condition code is set to 2. The next branch instruction (BH) tests for a greater than (high) condition. Since a greater than condition exists, a branch to the instruction labeled ROUTINE taken place.

CVB

10.9. CONVERT TO BINARY (CVB)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CVB	**4F**	**RX**	**4**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *convert to binary* (CVB) instruction converts the packed decimal contents of the double word in operand 2 to its binary equivalent and puts the result in the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CVB	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CVB	$r_1,s_2,(x_2)$

The CVB instruction converts a packed decimal number into a binary number. The operand 1 register contains the resulting binary number (4 bytes in length) and operand 2 is a packed number (8 bytes in length) aligned on a double-word boundary. Operand 2 is checked for a valid sign in the low order four bits. The remaining 60 bits represent a decimal value not exceeding 15 decimal digits. The largest decimal number that can be converted is +2,147,483,647 (2^{31}—1) and the smallest is —2,147,483,678 (—2^{31}). Any decimal number outside this range causes a fixed-point decimal divide. The result of the conversion is placed in the operand 1 register. The sign value (low order four bits) of the packed decimal number in operand 2 becomes the sign value (high order bit or bits) of the binary number in the operand 1 register.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must contain a packed decimal number aligned on a double-word boundary.
- If the sign value of the packed decimal number in operand 2 is positive, the sign value of the binary number in operand 1 is positive.
- If the sign value of the packed decimal number in operand 2 is negative, the binary number in operand 1 is represented in twos complement form and the sign value is negative.
- The condition code remains unchanged.
- There is no conversion between EBCDIC and binary. EBCDIC data must first be packed, then converted to binary.

Example:

```
LABEL   ΔOPERATIONΔ                        OPERAND
1       10        16
-------------------------------------------------------
        SR        7,7
        SR        5,5
        L         6,=F'1ØØ'
        PACK      DBLWDP,AMT(3)
        CVB       5,DBLWDP
        CR        5,6
        BH        ERRTN
        AR        7,5
        .
        .
        .
ERRTN   A         4,=F'1'
        .
        .
        .
DBLWDP  DS        D
AMT     DC        ZL3'428'
```

Register 5 before execution of CVB instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	0	0	hex

DBLWDP before and after execution of CVB instruction:

0	0	0	0	0	0	0	0	0	0	0	0	4	2	8	C	Packed decimal number

Register 5 after execution of CVB instruction:

sign bits

0000	0000	0000	0000	0000	0001	1010	1100	binary
0	0	0	0	0	1	A	C	hex

In this example, the SR instruction cleared both register 5 and 7 to zero by subtracting the contents of the registers from themselves. Then, a full-word value of 100 is loaded into register 6. The contents of AMT is packed into DBLWDP and the CVB instruction converts the packed decimal value into its binary equivalent which replaces the contents of register 5. Register 5 is then compared to register 6. Since the value of the contents in register 5 is greater than that of register 6, the condition code is set to 2. The BH instruction then tests for a greater than condition and a branch to the instruction labeled ERRTN takes place. If the condition code is not 2, no branch takes place and the program continues processing with the instruction following the branch.

CVD

10.10. CONVERT TO DECIMAL (CVD)

General				Possible Program Exceptions	
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)	■ ADDRESSING	■ PROTECTION
MNEM.	HEX.			□ DATA (INVALID SIGN/DIGIT)	□ SIGNIFICANCE
CVD	4E	RX	4	□ DECIMAL DIVIDE	■ SPECIFICATION:
Condition Codes				□ DECIMAL OVERFLOW	□ NOT A FLOATING-POINT REGISTER
□ IF RESULT = 0, SET TO 0				□ EXECUTE	□ OP 1 NOT ON HALF-WORD BOUNDARY
□ IF RESULT <0, SET TO 1				□ EXPONENT OVERFLOW	□ OP 2 NOT ON HALF-WORD BOUNDARY
□ IF RESULT >0, SET TO 2				□ EXPONENT UNDERFLOW	□ OP 2 NOT ON FULL-WORD BOUNDARY
□ IF OVERFLOW, SET TO 3				□ FIXED-POINT DIVIDE	■ OP 2 NOT ON DOUBLE-WORD BOUNDARY
■ UNCHANGED				□ FIXED-POINT OVERFLOW	□ OP 1 NOT EVEN NUMBERED REGISTER
				□ FLOATING-POINT DIVIDE	□ OP 1 NOT ODD NUMBERED REGISTER
				□ OPERATION	□ NONE

The *convert to decimal* (CVD) instruction converts the binary number in the operand 1 register into its packed decimal equivalence and stores the result in the double word in operand 2.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	CVD	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	CVD	$r_1,s_2(x_2)$

The CVD instruction converts a binary value into a packed decimal value. Operand 1 register contains the binary value (4 bytes) and operand 2 contains a double-word field in packed decimal format (8 bytes). The largest decimal number that can be represented in binary in the operand 1 register is +2,147,483,647 ($2^{31}-1$) and the smallest is —2,147,483,648 (-2^{31}). Since the number to be converted is a 32-bit signed integer from a register and there are 15 decimal digits available for its decimal equivalent, an overflow condition cannot occur.

The sign value (high order bit or bits) of the binary number in the operand 1 register becomes the sign value (low order four bits) of the packed decimal number in operand 2. The result of the conversion is placed in the double word of operand 2. Note that the CVD instruction is one of the few instructions that has operand 1 as the sending field and operand 2 as the receiving field.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a double word or aligned on a double-word boundary.
- If the sign value of the binary number is positive, the sign value of the decimal number is positive.
- If the sign value of the binary number represented in twos complement form is negative, the sign value of the decimal number is negative.
- The condition code remains unchanged.
- The result is stored in operand 2, unlike most instructions, where operand 1 is the receiving field.

Example:

```
LABEL   ΔOPERATIONΔ                       OPERAND
1        10        16
-----------------------------------------------------------------
         SR        7,7
         PACK      DBLEWRD,CARDIN+9(3)
         CVB       4,DBLEWRD
         AR        7,4
         BZ        NOPAY
         CVD       4,DBLEWRD
         ZAP       TOTHRS,DBLEWRD
         .
         .
         .
NOPAY    A         9,=F'1'
         .
         .
         .
CARDIN   DC        CL8Ø'SMITH,J. 480 WKTOT   EXEMPT X'
DBLEWRD  DS        D
TOTHRS   DS        PL3
```

DBLEWRD before execution of CVD instruction:

0	0	0	0	0	0	0	0	0	0	0	0	4	8	0	C	Packed decimal

Register 4 before execution of CVD instruction:

0000	0000	0000	0000	0000	0001	1110	0000	binary
0	0	0	0	0	1	E	0	hex

DBLEWRD after execution of CVD instruction:

0	0	0	0	0	0	0	0	0	0	0	0	4	8	0	C	Packed decimal

In this example, register 7 is cleared to zero. A field from card input (CARDIN+9), which is EBCDIC and in zoned decimal format, is packed into DBLEWRD. The CVB instruction then converts the packed decimal number in DBLEWRD into its binary equivalent and puts the result into register 4. The content of register 4 is added to register 7. The condition code is set to 2, since the result of the addition is greater than zero. The next branch instruction (BZ) tests for an equal-to-zero condition. Since that condition does not exist, no branch takes place and the instruction following the branch instruction is executed. The CVD instruction then converts the contents of register 4 into its decimal equivalent and puts the result into DBLEWRD. The ZAP instruction clears TOTHRS to zero and adds the packed decimal number in DBLEWRD to TOTHRS. (This is an example where truncation is beneficial.)

10.11. DIVIDE (D)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
D	5D	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *divide* (D) instruction algebraically divides the double word in the operand 1 register pair (dividend) by the full word in operand 2 (divisor) and puts the result (quotient and remainder) in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	D	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	D	$r_1,s_2(x_2)$

Operand 1 consists of an even-odd pair of contiguous registers, where the even-numbered register, since it is the lower numbered register, is specified as operand 1. Every time the even-numbered operand 1 is referenced in the D instruction, both registers are used. The dividend occupies the register pair as a double-word value with the high order bit or bits as the sign value. Operand 2 must be either defined as a full word in main storage, or aligned on a full-word boundary. The resulting quotient occupies the odd-numbered register as a full-word value with its sign determined algebraically. The remainder occupies the even-numbered register, also as a full-word value with its sign the same as the dividend. If the values of the divisor and dividend cause the quotient to be larger than a 32-bit signed integer, a fixed-point divide program exception occurs, no division takes place, and the dividend remains unchanged.

To load a value (dividend) into an even-odd register pair, use the load multiple (LM) instruction (see 10.18). If the value (dividend) can be contained in one register, it must be loaded into the odd-numbered register only. This can be done through the use of the load (L), load register (LR), or load half-word (LH) instructions. The even-numbered register must be cleared before execution of the D instruction.

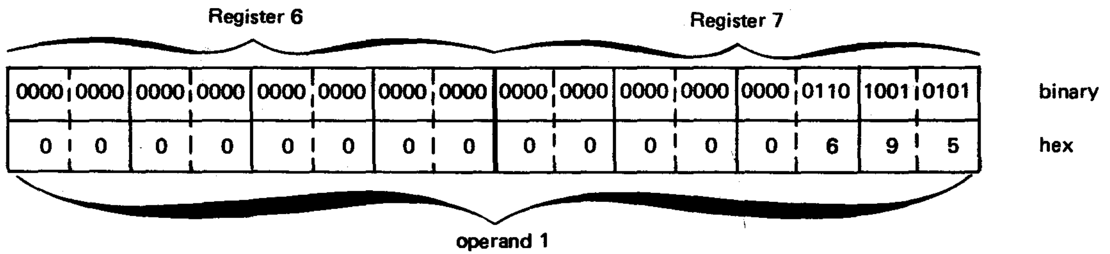
Operational Considerations:

- Operand 1 consists of an even-odd pair of registers located in consecutive order in the processor.
- Operand 1 always references the even-numbered register.
- The dividend occupies both registers. After the D instruction is executed, the quotient occupies the odd-numbered register, and the remainder occupies the even-numbered register.
- Operand 2 must be either defined as a full word or aligned on a full-word boundary.
- If operand 1 does not reference an even-numbered register, a specification exception occurs.
- The condition code remains unchanged.
- Division by zero causes a fixed-point divide program exception.
- Any of the even-numbered general registers (0 thru 14) can be used as the operand 1 register pair.

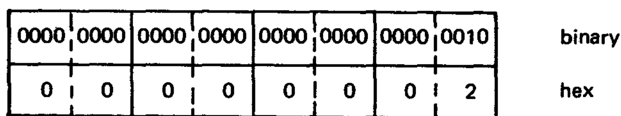
Example:

```
LABEL    ΔOPERATIONΔ                  OPERAND
1        10        16
------------------------------------------------------------
         SR        6,6
         L         7,DIVIDEND
         D         6,DIVISOR
         .
         .
         .
DIVIDEND DC        F'1685'
DIVISOR  DC        F'2'
```

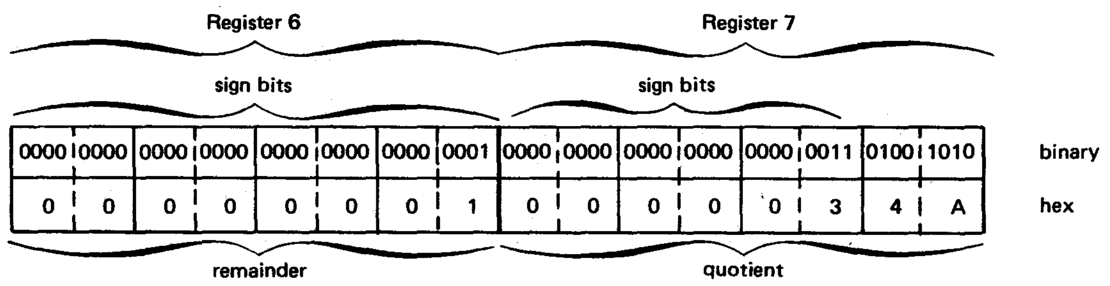
Registers 6 and 7 before execution of D instruction:



DIVISOR before and after execution of D instruction:



Registers 6 and 7 after execution of D instruction:

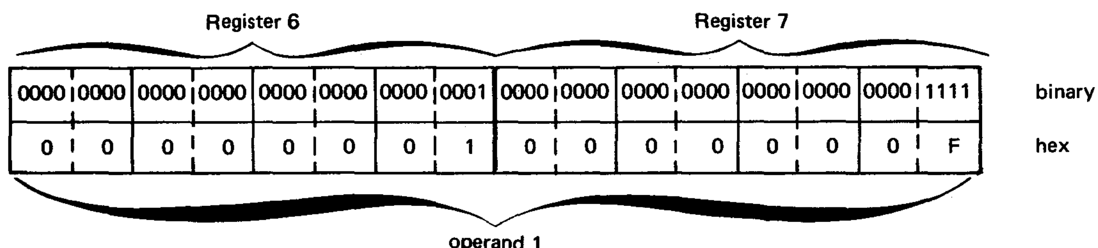


In this example, registers 6 and 7 are the operand 1 register pair and DIVISOR (operand 2) has been defined as a full-word constant. First, the SR instruction clears register 6 to zeros. Since the full word in DIVIDEND can be contained in one register, it is loaded into register 7 (the odd-numbered register) through the use of the L instruction. Then, the register pair 6-7 is divided by the full word in DIVISOR. The resulting remainder occupies register 6 with a positive sign (the same as the dividend) and the resulting quotient occupies register 7 with a positive sign (determined algebraically).

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10       16
-------------------------------------------------------------
          LM       6,7,DIVIDEND
          D        6,DIVISOR
          .
          .
          .
DIVIDEND  DC       F'1'
          DC       F'15'
DIVISOR   DC       F'1ØØØ'
```

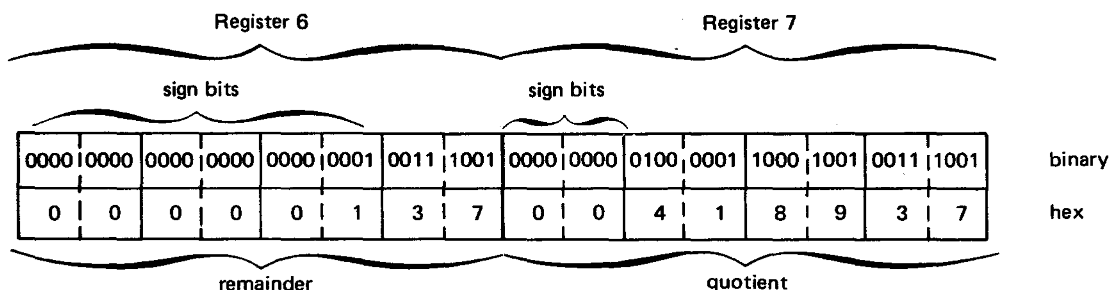
Registers 6 and 7 before execution of D instruction:



DIVISOR before and after execution of D instruction:

0000 0000	0000 0000	0000 0011	1110 1000	binary
0 0	0 0	0 3	E 8	hex

Registers 6 and 7 after execution of D instruction:



In this example, the even-odd register pair is loaded with the contents of DIVIDEND. This is done through the LM instruction because the dividend cannot be contained in one register and has a value of +4,294,967,311 which is greater than

+2,147,483,647 (2^{31}—1)
—2,147,483,648 (—2^{31}).

The content of DIVISOR is then divided into the double-word value in the even-odd register pair. The resulting quotient with sign occupies the odd-numbered register and the resulting remainder with sign occupies the even-numbered register.

10.12. DIVIDE (DR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
DR	1D	RR	2

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *DIVIDE* (DR) instruction algebraically divides the contents of the double word in the operand 1 register pair (dividend) by the full word in the operand 2 register. The result (quotient and remainder) is placed in operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DR	r_1, r_2

Operational Considerations:

- Operand 1 consists of a pair of contiguous registers (64 bits) containing a fixed-point binary value. The registers are even-odd numbered, the lower numbered register being even. You must specify the even-numbered register as operand 1. The odd-numbered registers must contain the dividend before you can use this instruction. You may specify any of the general registers (0 through 15).

- Operand 2 is a 32-bit register (0 through 15) containing a fixed-point binary value (dividend). Operand 2 is not changed by the execution of this instruction.

- After the instruction is executed, the quotient with sign is put into the odd-numbered register, and the remainder with the same sign occupies the even-numbered register. If the quotient and remainder do not fill their respective 32-bit fields, leftmost bit positions are filled by bits having the same value as the sign.

- If you attempt to divide by zero, or if the quotient does not fit into the 32-bit odd-numbered register in operand 1, a fixed-point divide program exception occurs.

Example:

```
LABEL    ΔOPERATIONΔ              OPERAND
1        10       16
--------------------------------------------------
         LM       6,8,DIVEND
         DR       6,8
         .
         .
         .
DIVEND   DC       D'+64'
DIVISOR  DC       F'+32'
```

In this example, registers 6, 7, and 8 are loaded with the main storage contents of DIVEND and DIVISOR, respectively. Then, divide the contents of registers 6 and 7 by the contents of register 8 and place the result (quotient with sign) in register 7.

Note that the use of the LM instruction eliminates the writing of three separate load instructions but still loads three registers. Also note that the quotient and its sign are loaded into register 7 and the remainder with the same sign value as the quotient that occupies register 6.

Registers 6 and 7 before execution of DR instruction:

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0100	0000	binary
0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	hex

Register 8 before and after execution of DR instruction:

0000	0000	0000	0000	0000	0000	0010	0000	binary
0	0	0	0	0	0	2	0	hex

Registers 6 and 7 after execution of DR instruction:

0000	0000	0000	000C	0000	0000	0000	000C	0000	0000	0000	0000	0000	0000	0000	0010	binary
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	hex

10.13. LOAD (L)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
L	58	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *load* (L) instruction places the full word in operand 2 into the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	L	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	L	$r_1,s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a full word or aligned on a full-word boundary.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10         16
_____________________________________________________________
         L          4,XNUM
         .
         .
         .
         DS         ØF
XNUM     DC         X'ØØØØØØ18'
```

Register 4 before execution of L instruction:

0000	0000	0000	0000	0010	1111	1001	1000	binary
0	0	0	0	2	F	9	8	hex

(leftover data from previous program)

XNUM before and after execution of L instruction:

0000	0000	0000	0000	0000	0000	0001	1000	binary
0	0	0	0	0	0	1	8	hex

Register 4 after execution of L instruction:

0000	0000	0000	0000	0000	0000	0001	1000	binary
0	0	0	0	0	0	1	8	hex

In this example, XNUM is defined as a hexadecimal constant aligned on a full-word boundary and register 4 is the operand 1 register. The L instruction places the full word in operand 2 into register 4 replacing any leftover data in register 4 with the contents of XNUM.

10.14. LOAD (LR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LR	18	RR	2

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *load* (LR) instruction places the contents of the operand 2 register into the operand 1 register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LR	r_1,r_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- The contents of the register specified by operand 2 (r_2) are loaded into the register specified by operand 1 (r_1).
- The contents of the register specified by operand 2 (r_2) remain unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1         10       16
-------------------------------------------------------------
          L        7,FINTOT
          L        5,SUBTOT
          LR       6,5
          LR       8,7
          A        5,INTERTOT
          AR       7,5
          CVD      5,SUBTOTP
          CVD      7,FINTOTP
          .
          .
          .
FINTOT    DC       F'5630'
SUBTOT    DC       F'220'
FINTOTP   DS       D
SUBTOTP   DS       D
INTERTOT  DC       F'20'
```

Register 6 before execution of LR instruction:

0000	0000	0000	0000	0000	0111	1100	1011	binary	(leftover data from previous program)
0	0	0	0	0	7	C	B	hex	

Register 5 before and after execution of LR instruction:

0000	0000	0000	0000	0000	0000	1101	1100	binary
0	0	0	0	0	0	D	C	hex

Register 6 after execution of LR instruction:

0000	0000	0000	0000	0000	0000	1101	1100	binary
0	0	0	0	0	0	D	C	hex

In this example, the full word in FINTOT (FINTOT represents final total) is loaded into register 7 and the full word in SUBTOT (SUBTOT represents subtotal) is loaded into register 5. Then the content of register 5 is loaded into register 6 and the content of register 7 is loaded into register 8 so it can be saved prior to the execution of the succeeding add instructions. Then the full word in INTERTOT is added to register 5 (now register 5 has the most current subtotal). The content of register 5 is added to the content of register 7 (now register 7 has the most current final total). The first CVD instruction converts the binary number in register 5 to its decimal equivalent and puts the result into the double word in SUBTOTP. The second CVD instruction converts the binary number in register 7 to its decimal equivalent and puts the result into the double word in FINTOTP.

LTR

10.15. LOAD AND TEST (LTR)

General				
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LTR	12	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *load and test* (LTR) instruction places the contents of the operand 2 register into the operand 1 register. The value and sign of operand 2 determines the setting of the condition code. The actual testing of the condition code is done through the execution of another instruction.

Explicit and Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	LTR	r_1,r_2

If operand 2 = 0, set to 0.

If operand 2 < 0, set to 1.

If operand 2 > 0, set to 2.

Usually, a conditional branch instruction tests the resulting condition code for an equal to zero, less than zero, or greater than zero condition. If the condition specified is met, a branch takes place accordingly. If not, the program continues processing with the following instruction.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- It is your responsibility to test the condition code setting.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10          16
-------------------------------------------------------------
         L           6,=F'25'
         LTR         7,6
         BC          8,ERRTN
         .
         .
         .
ERRTN    AP          ERCNT,=P'1'
         .
         .
         .
ERCNT    DC          PL2'Ø'
```

Register 7 before execution of LTR instruction:

0000	0000	0000	0000	1010	0000	0000	0000	binary
0	0	0	0	A	0	0	0	hex

(leftover data from previous program)

Register 6 before and after execution of LTR instruction:

0000	0000	0000	0000	0000	0000	0001	1001	binary
0	0	0	0	0	0	1	9	hex

Register 7 after execution of LTR instruction:

0000	0000	0000	0000	0000	0000	0001	1001	binary
0	0	0	0	0	0	1	9	hex

In this example, a full word containing the decimal value 25 is loaded into register 6. Then, the LTR instruction loads the contents of register 6 into register 7. The condition code is set to 2, since the value of register 6 (operand 2) is greater than zero. The BC instruction tests for an equal to zero condition which is represented by the decimal value 8 in operand 1. If an equal to condition existed, a branch to the instruction labeled ERRTN would take place. Since that condition does not exist, the program continues processing with the instruction immediately following the BC instruction.

LCR

10.16. LOAD COMPLEMENT (LCR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LCR	13	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *load complement* (LCR) instruction places the twos complement form of the contents of operand 2 register into the operand 1 register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LCR	r_1, r_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- If operand 2 is a positive value, the twos complement of that value is placed into operand 1 when the instruction is executed. If the value in operand 2 is negative, the positive value is placed in operand 1 when the instruction is executed. The maximum value you can specify in operand 2 is +2,147,483,647 (2^{31}—1) or —2,147,483,647 (—2^{31}—1).
- A zero value in operand 2 is not changed when complemented.
- Operand 2 is not changed by the execution of the instruction.
- ~~The LCR instruction is a featured instruction. An operation program exception is caused if you use this instruction and your processor does not have the control feature.~~

Example:

```
LABEL     ΔOPERATIONΔ                     OPERAND
1         10        16

          L         5,FULLWORD
          LCR       6,5
          LTR       6,6
          .
          .
          .
FULLWORD  DC        F'1ØØ'
```

Register 5 before and after execution of LCR instruction:

0000	0000	0000	0000	0000	0000	0110	0100	binary
0	0	0	0	0	0	6	4	hex

Register 6 after execution of LCR instruction:

1111	1111	1111	1111	1111	1111	1001	1100	binary
F	F	F	F	F	F	9	C	hex

In this example, the contents of FULLWORD is loaded into register 5 and the LCR instruction loads the complement of the content of register 5 into register 6. Since the result is less than zero, the condition code is set to 1 and the load and test (LTR) instruction (see 10.15) loads the content of register 6 into itself and tests the condition code. Because the registers of operands 1 and 2 in the LTR instruction are the same, the operation is performed as a test without data movement.

LH

10.17. LOAD HALF WORD (LH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LH	48	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *load half word* (LH) instruction places the half word in operand 2 into the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LH	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LH	$r_1,s_2(x_2)$

Since registers only work in conjunction with full words, the half word in operand 2 is automatically expanded to 32 bits by propagating the sign bit through the 16 high order bit positions. Then, operand 2 is loaded into the operand 1 register.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must either be defined as a half word or aligned on a half-word boundary.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10         16
----------------------------------------------------------------
         LH         4,PRODUCT
         .
         .
         .
PRODUCT  DC         H'256'
```

Register 4 before execution of LH instruction:

0000	0000	0000	0000	0000	1100	0001	1010	binary
0	0	0	0	0	C	1	A	hex

(leftover data from previous program)

PRODUCT before and after execution of LH instruction:

before expansion (last four digits); after expansion (all eight digits)

0000	0000	0000	0000	0000	0001	0000	0000	binary
0	0	0	0	0	1	0	0	hex

Register 4 after execution of LH instruction:

0000	0000	0000	0000	0000	0001	0000	0000	binary
0	0	0	0	0	1	0	0	hex

In this example, the half word in PRODUCT is expanded temporarily to a 32-bit signed integer. Then the LH instructions loads the contents of PRODUCT (now a 32-bit signed integer) into register 4.

LM

10.18. LOAD MULTIPLE (LM)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LM	**98**	**RS**	**4**

Condition Codes
□ IF RESULT = 0, SET TO 0
□ IF RESULT < 0, SET TO 1
□ IF RESULT > 0, SET TO 2
□ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *load multiple* (LM) instruction loads the contents of two or more consecutive registers (operands 1 and 3) with an equal number of consecutive full words in main storage (operand 2).

Explicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	LM	$r_1,r_3,d_2(b_2)$

Implicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	LM	r_1,r_3,s_2

The operand 1 register is the first register loaded and the operand 3 register is the last register loaded. If operands 1 and 3 are not consecutive, any registers consecutive to the operand 1 register up to and including the operand 3 register also are included. If the address of the operand 3 register is less than the address of the operand 1 register, the register addresses wrap around from 15 to 0. The number of full words in main storage to be loaded is determined by the number of consecutive registers specified. The registers are loaded in ascending sequence starting with the operand 1 register up to and including the operand 3 register. The content of operand 2 is loaded into the registers beginning with the byte addressed by the operand 2 label and continuing with as many full words that are needed to fill the registers specified.

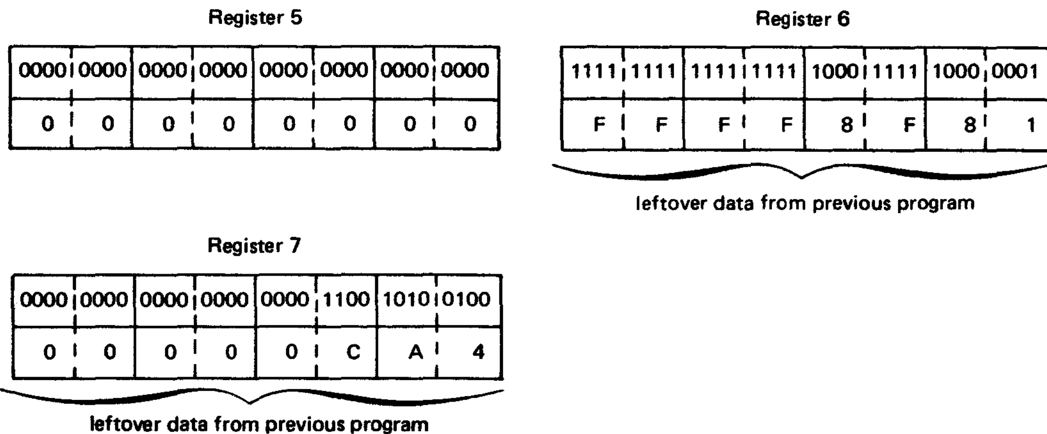
Operational Considerations:

- Any of the general registers (0 thru 15) can be used as operands 1 and 3.
- Operand 2 must either be defined as a full word or aligned on a full-word boundary.
- If operand 2 does not reference the correct number of full words needed to fill all the registers, full words consecutive to the first full word specified by operand 2 are loaded into the registers until the operand 3 register is filled.
- When loading multiple registers, the wraparound concept applies.
- If operand 1 and operand 3 reference the same register, only that register is loaded with the contents of the first full word of operand 2.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10         16
-----------------------------------------------------------------
   .      LM         5,7,VAL1
          .
          .
          .
VAL1      DC         F'1Ø'
VAL2      DC         F'2Ø'
VAL3      DC         F'3Ø'
```

Registers 5, 6, and 7 before execution of LM instruction:



VAL1, VAL2, and VAL3 before and after execution of LM instruction:

VAL1

0000	0000	0000	0000	0000	0000	0000	1010
0	0	0	0	0	0	0	A

VAL2

0000	0000	0000	0000	0000	0000	0001	0100
0	0	0	0	0	0	1	4

VAL3

0000	0000	0000	0000	0000	0000	0001	1110
0	0	0	0	0	0	1	E

Registers 5, 6, and 7 after execution of LM instruction:

Register 5

0000	0000	0000	0000	0000	0000	0000	1010
0	0	0	0	0	0	0	A

Register 6

0000	0000	0000	0000	0000	0000	0001	0100
0	0	0	0	0	0	1	4

Register 7

0000	0000	0000	0000	0000	0000	0001	1110
0	0	0	0	0	0	1	E

In this example, operands 1 and 3 specify that registers 5, 6, and 7 are to be loaded with three consecutive full words from main storage starting with the first full word at VAL1 (operand 2) and continuing until register 7 is filled.

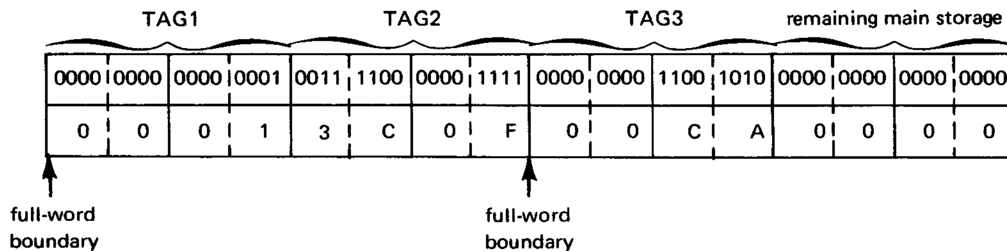
Example:

```
LABEL  ΔOPERATIONΔ                          OPERAND
1      10         16
------------------------------------------------------------
       LM         5,5,TAG1
       .
       .
       .
       DS         ØF
TAG1   DC         XL2'ØØØ1'
TAG2   DC         XL2'3CØF'
TAG3   DC         XL2'ØØCA'
```

Register 5 before execution of LM instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	0	0	hex

TAG1, TAG2, TAG3 before and after execution of LM instruction:



Register 5 after execution of LM instruction:

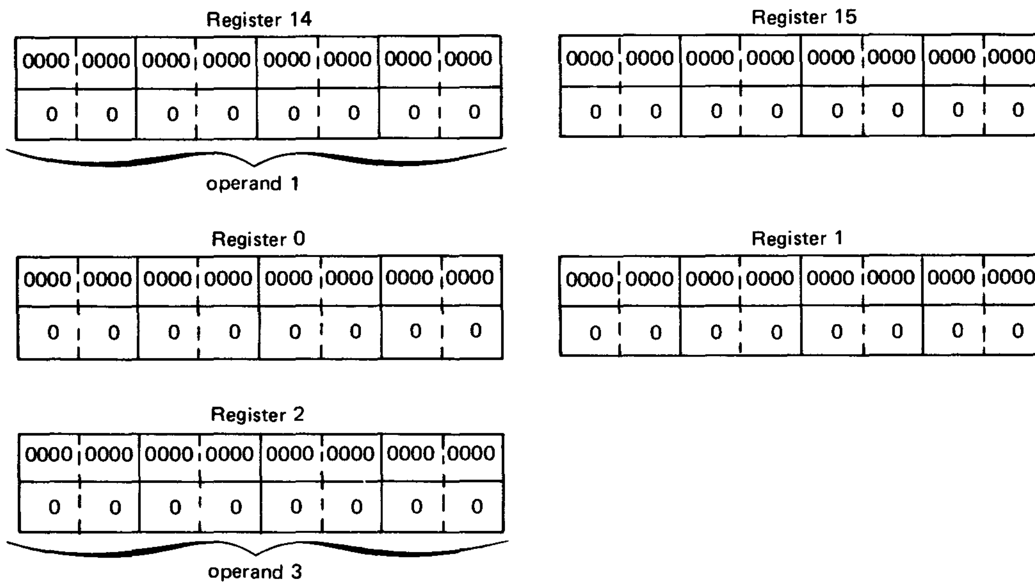
0000	0000	0000	0001	0011	1100	0000	1111	binary
0	0	0	1	3	C	0	F	hex

In this example, operand 1 and operand 3 both refer to register 5. Therefore, the content of operand 2 is loaded into register 5 (operand 1) beginning with the first byte at TAG1 and continuing with as many full words that are needed to fill register 5 (operand 3). Note that only register 5 is filled with the first full word at operand 2.

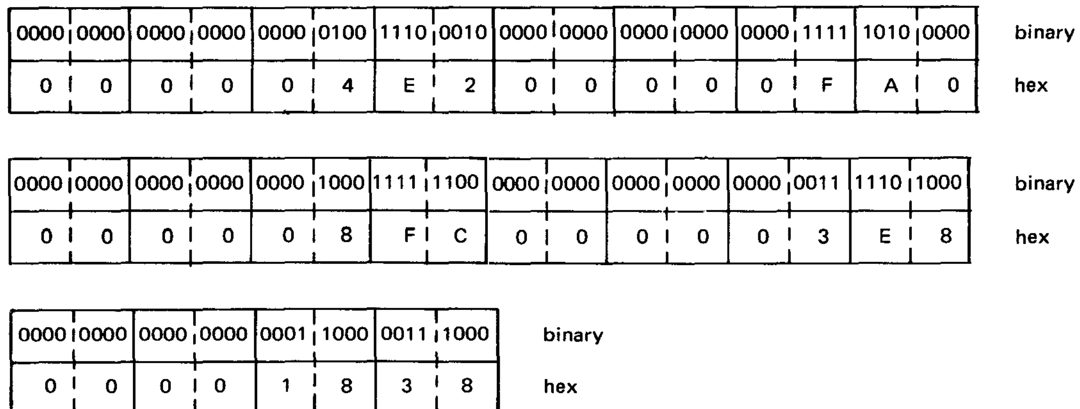
Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10          16
--------------------------------------------------------
         LM          14,2,SECTOR
         .
         .
         .
SECTOR   DC          F'125Ø'
         DC          F'4ØØØ'
         DC          F'23ØØ'
         DC          F'1ØØØ'
         DC          F'62ØØ'
```

Registers before execution of LM instruction:



SECTOR before and after execution of LM instruction:



Registers after execution of LM instruction:

Register 14

0000	0000	0000	0000	0000	0100	1110	0010
0	0	0	0	0	4	E	2

Register 15

0000	0000	0000	0000	0000	1111	1010	0000
0	0	0	0	0	F	A	0

Register 0

0000	0000	0000	0000	0000	1000	1111	1100
0	0	0	0	0	8	F	C

Register 1

0000	0000	0000	0000	0000	0011	1110	1000
0	0	0	0	0	3	E	8

Register 2

0000	0000	0000	0000	0001	1000	0011	1000
0	0	0	0	1	8	3	8

In this example, register 14 is the first register loaded and register 2 is the last register loaded. Since the address of operand 2 (register 2) is less than the address of operand 1 (register 14), the register addresses wrap around from 15 to 0 up to and including 2. Operand 2 is either defined as a full word or aligned on a full-word boundary. The contents of operand 2 is loaded into register 14 starting with the byte addressed by SECTOR and continuing with as many full words until register 2 is filled.

Remember that most I/O operations use registers 14, 15, 0, and 1. So, if you use these registers and then perform some input or output in your program, the original contents of these registers are destroyed. However, you can use these registers if you save the contents prior to every I/O operation, and restore them after completing each I/O operation.

It may be helpful to note that the supervisor usually uses the lower numbered registers and data management usually uses the higher numbered registers.

LNR

10.19. LOAD NEGATIVE (LNR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LNR	11	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- □ IF RESULT > 0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *load negative* (LNR) instruction places the twos complement of the content of the operand 2 register into the operand 1 register. If operand 2 contains a negative value or a value of zero, the instruction places that value unchanged into operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LNR	r_1, r_2

Operational Consideration:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1         10        16
------------------------------------------------------------
          LM        5,7,NUMBERS1
          LNR       5,5
          LNR       6,6
          LNR       7,7
          .
          .
          .
NUMBERS1 DC         F'4'
NUMBERS2 DC         F'5'
NUMBERS3 DC         F'6'
```

In this example registers 5, 6, and 7 are filled with contents of NUMBERS1, NUMBERS2, and NUMBERS3, respectively. Then each of the three LNR instructions, operating respectively on registers 5, 6, and 7, replaces the full word in its respective register with the twos complement of that full word.

Register 5 before execution of the LNR instruction:

0000	0000	0000	0000	0000	0000	0000	0100	binary
0	0	0	0	0	0	0	4	hex

Register 6 before execution of LNR instruction:

0000	0000	0000	0000	0000	0000	0000	0101	binary
0	0	0	0	0	0	0	5	hex

Register 7 before execution of LNR instruction:

0000	0000	0000	0000	0000	0000	0000	0110	binary
0	0	0	0	0	0	0	6	hex

Register 5 after execution of LNR instruction:

1111	1111	1111	1111	1111	1111	1111	1100	binary
F	F	F	F	F	F	F	C	hex

Register 6 after execution of LNR instruction:

1111	1111	1111	1111	1111	1111	1111	1011	binary
F	F	F	F	F	F	F	B	hex

Register 7 after execution of LNR instruction:

1111	1111	1111	1111	1111	1111	1111	1010	binary
F	F	F	F	F	F	F	A	hex

LPR

10.20. LOAD POSITIVE (LPR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LPR	10	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *load positive* (LPR) instruction places the positive value of the content of the operand 2 register in the operand 1 register. If operand 2 contains a positive value or zero, that same value is placed unchanged in operand 1. If operand 2 contains a negative number, the twos complement of that number (its positive value) is loaded into operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LPR	r_1, r_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1 and 2.
- The maximum negative value you can specify in operand 2 is —2,147,483,657 ($-2^{31}-1$). Otherwise, a fixed-point overflow program exception occurs.
- Operand 2 is not changed by the execution of the instruction.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10        16
-----------------------------------------------------------
         LM        5,7,NUMBERS1
         LPR       5,5
         LPR       6,6
         LPR       7,7
         .
         .
         .
NUMBERS1 DC        F'-4'
NUMBERS2 DC        F'-5'
NUMBERS3 DC        F'-6'
```

In this example, registers 5, 6, and 7 are filled with the contents of NUMBERS1, NUMBERS2, and NUMBERS3, respectively. Then each of the three LNR instructions, operating respectively on registers 5, 6, and 7, replaces the full word in its respective register with the twos complement of that full word. The result is their positive values.

Register 5 before execution of LPR instruction:

1111	1111	1111	1111	1111	1111	1111	1100	binary
F	F	F	F	F	F	F	C	hex

Register 6 before execution of LPR instruction:

1111	1111	1111	1111	1111	1111	1111	1011	binary
F	F	F	F	F	F	F	B	hex

Register 7 before execution of LPR instruction:

1111	1111	1111	1111	1111	1111	1111	1010	binary
F	F	F	F	F	F	F	A	hex

Register 5 after execution of LPR instruction:

0000	0000	0000	0000	0000	0000	0000	0100	binary
0	0	0	0	0	0	0	4	hex

Register 6 after execution of LPR instruction:

0000	0000	0000	0000	0000	0000	0000	0101	binary
0	0	0	0	0	0	0	5	hex

Register 7 after execution of LPR instruction:

0000	0000	0000	0000	0000	0000	0000	0110	binary
0	0	0	0	0	0	0	6	hex

10.21. MULTIPLY (M)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
M	5C	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *multiply* (M) instruction algebraically multiplies the operand 1 register pair by the full word in operand 2. The result replaces the operand 1 register pair.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	M	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	M	$r_1,s_2(x_2)$

Operand 1 must be a contiguous pair of even-odd registers. The even-numbered register, since it is a lower numbered register, is specified as operand 1. Both the multiplier (operand 2) and the multiplicand (operand 1) are 32-bit signed integers but the product is always a 64-bit signed integer. Before execution of the M instruction, the multiplicand must be loaded into the odd-numbered register, while the content of the even-numbered register is ignored. The multiplier must either be defined as a full word or aligned on a full-word boundary. After execution of the M instruction, the resulting product replaces the even-odd register pair as a double-word value with the high order bits or bit as the sign value. The sign of the product is determined algebraically; like signs produce positive results and unlike signs produce negative results. If the product is always contained in the odd-numbered register, you can ignore the contents of the even-numbered register and store the contents of the odd-numbered register as the product.

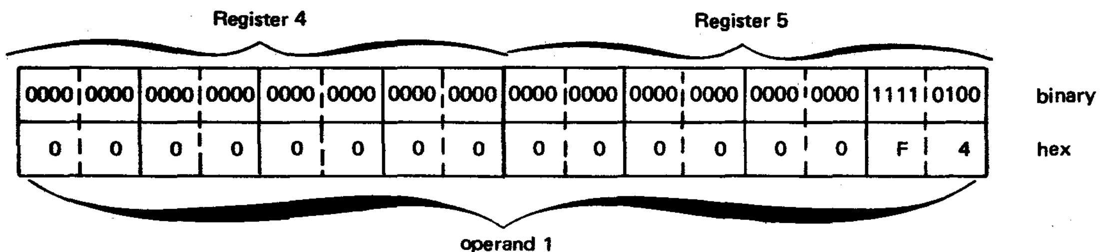
Operational Considerations:

- Operand 1 consists of an even-odd pair of registers located in consecutive order in the processor.
- Operand 1 always references the even-numbered register.
- The multiplicand occupies the odd-numbered register as a full-word value.
- After the M instruction is executed, the product occupies both registers as a double-word value.
- Operand 2 must either be defined as a full word or aligned on a full-word boundary.
- Any of the even-numbered general registers (0 thru 14) can be used as operand 1.

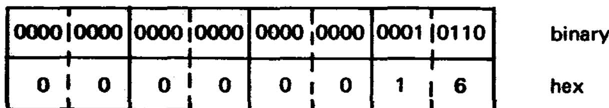
Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10      16
------------------------------------------------------------
         L       5,MULTCAND
         M       4,MULTPLYR
         .
         .
         .
MULTCAND DC      F'244'
MULTPLYR DC      F'22'
```

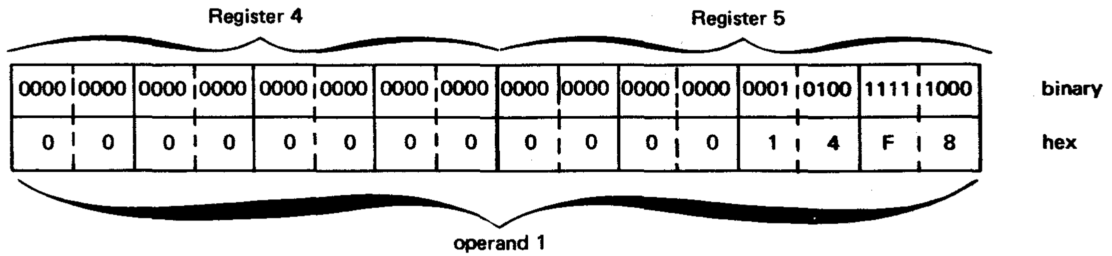
Registers 4 and 5 before execution of M instruction:



MULTPLYR before and after execution of M instruction:



Registers 4 and 5 after execution of M instruction:



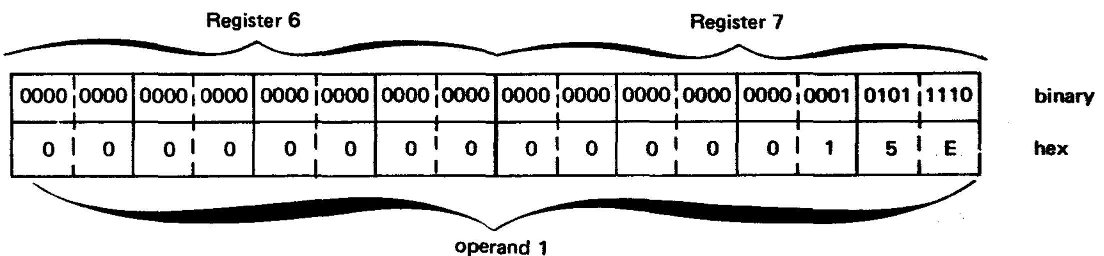
In this example, the full-word value in MULTCAND is loaded into register 5 (odd-numbered register). Then, the full-word value in MULTPLYR is multiplied by register 4 (the even-odd register pair). The even-numbered register is ignored and the content of the odd-numbered register (in this case, 5) is used in the multiplication.

The resulting product replaces the even-odd register pair as a double-word value. Since the value of this product is less than +2,147,483,647, it can be contained in register 5 and register 4 can be ignored.

Example:

```
LABEL    ΔOPERATIONΔ                       OPERAND
1         10        16
-----------------------------------------------------------------
          L         7,MULTCAND
          M         6,MULTPLYR
          ST        7,HOLDAREA
          .
          .
          .
MULTCAND  DC        F'35Ø'
MULTPLYR  DC        F'-5'
HOLDAREA  DS        F
```

Registers 6 and 7 before execution of M instruction:



MULTPLYR before and after execution of M instruction:

sign bits

1111	1111	1111	1111	1111	1111	1111	1011	binary
F	F	F	F	F	F	F	B	hex

(−5 in twos complement form)

Registers 6 and 7 after execution of M instruction:

sign bits

1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1001	0010	1010	binary
F	F	F	F	F	F	F	F	F	F	F	F	F	9	2	A	hex

(−1750 in twos complement form)

10.22. MULTIPLY (MR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MR	1C	RR	2

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *multiply* (MR) instruction algebraically multiplies the content of the operand 1 register pair (multiplicand) by the content of the operand 2 register (multiplier). The result (product) is placed in operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MR	r_1, r_2

Operational Considerations:

- Operand 1 consists of a pair of even-odd registers (64 bits). You must specify the even-numbered register as operand 1, and you must load the odd-numbered operand 1 register with the multiplicand before using this instruction.

- The product fills the odd-numbered register first and then, if necessary, the even-numbered register.

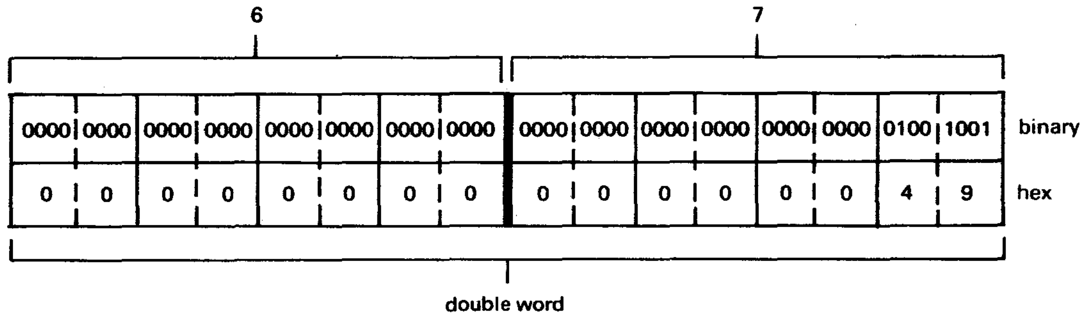
- Any of the general registers (0 through 15) can be used as operands 1 and 2. Operand 2 is not changed by the execution of this instruction.

Example:

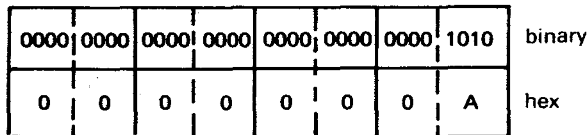
```
LABEL     ΔOPERATIONΔ                    OPERAND
1         10        16

          LH        8,PRICE
          LH        7,NEWBUY
          MR        6,8
          .
          .
          .
NEWBUY    DC        H'73'
PRICE     DC        H'1Ø'
```

In this example, place the contents NEWBUY and PRICE into registers 7 and 8, respectively. Then, multiply the content of the even-odd register pair 6 and 7 by register 8. (You address the pair of registers by using register 6.) The result is placed in register 7. If, however, the result of the multiplication exceeds the capacity of register 7, register 6 is filled with the remainder of the result.

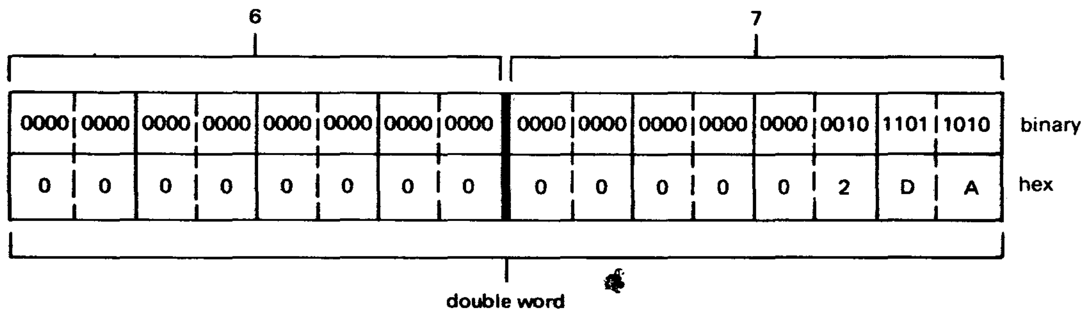
Registers 6 and 7 before execution:



Register 8 before execution:



Registers 6 and 7 after execution:



10.23. MULTIPLY HALF WORD (MH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MH	4C	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *multiply half word* (MH) instruction algebraically multiplies the content of the operand 1 register by the half word in operand 2. The result is placed in the operand 1 register.

Explicit Format:

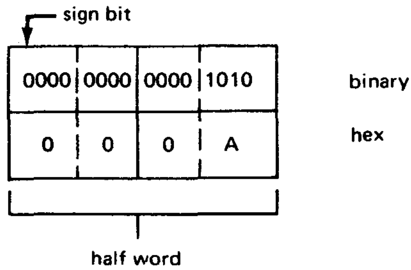
LABEL	△OPERATION△	OPERAND
[symbol]	MH	$r_1, d_2(x_2, b_2)$

Implicit Format:

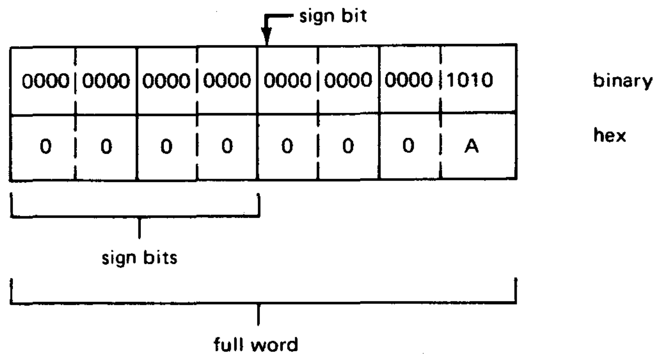
LABEL	△OPERATION△	OPERAND
[symbol]	MH	$r_1, s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Before execution of the MH instruction, operand 2 is expanded from 16 to 32 bits. The 16 high order bits are propagated with the sign bit value. The contents of operand 2 before the 16 high order bits are propagated with the sign bit value.



The contents of operand 2 after the 16 high order bits are propagated with the sign bit value.



Operand 2 is not permanently changed by the execution of the instruction.

- The result (product) fills the 32-bit operand 1 register from right to left. If the product does not fit into the operand 1 field, extra leftmost bits are truncated and the result or sign may be incorrect.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10        16
-----------------------------------------------------------
         L         7,NEWBUY
         MH        7,PRICE
         .
         .
         .
NEWBUY   DC        F'73'
PRICE    DC        H'1Ø'
```

In this example, load the contents NEWBUY into register 7 and multiply the half word of PRICE by the content of register 7. The product is placed in register 7.

Register 7 before execution of MH instruction:

0000	0000	0000	0000	0000	0000	0100	1001	binary
0	0	0	0	0	0	4	9	hex

PRICE before and after execution of MH instruction:

0000	0000	0000	0000	0000	0000	0000	1010	binary
0	0	0	0	0	0	0	A	hex

Register 7 after execution of MH instruction:

0000	0000	0000	0000	0000	0010	1101	1010	binary
0	0	0	0	0	2	D	A	hex

SLDA

10.24. SHIFT LEFT DOUBLE (SLDA)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SLDA	8F	RS	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *shift left double* (SLDA) instruction shifts all of the 63 bits of the operand 1 even-odd register pair to the left the number of bits specified by the low order six bits of the operand 2 address.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLDA	$r_1, d_2(b_2)$

Implicit Format:

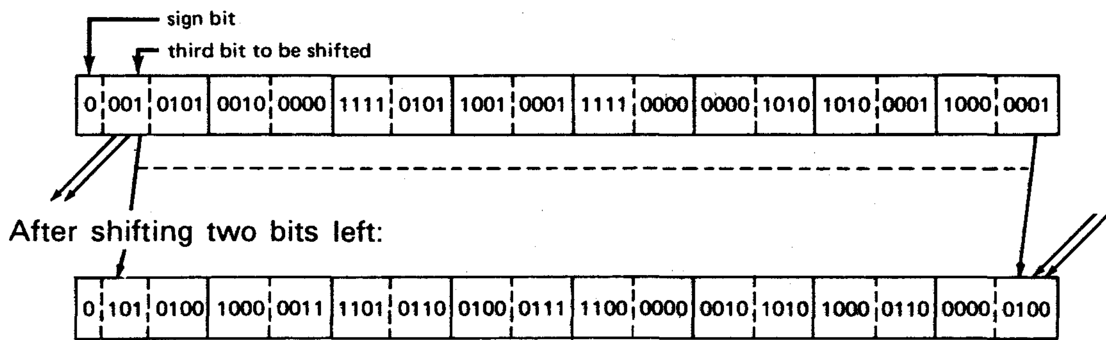
LABEL	△OPERATION△	OPERAND
[symbol]	SLDA	r_1, s_2

Operational Considerations:

- Any pair of general registers (0 through 15) can be used as operand 1. Operand 1 is an even-odd register pair. You must specify the even-numbered register of the pair as operand 1.
- The main storage address or label you specify in operand 2 is not changed by the SLDA instruction execution. Notice the formats indicate that you cannot specify a length in operand 2.

- The sign bit (leftmost bit) of the even-numbered operand 1 register pair is not moved or changed by the execution of this instruction. Only the 63 remaining bits can be shifted by this instruction.
- After the requested number of bits are shifted out of the operand 1 register pair, zeros fill the rightmost bit positions of the register pair that were emptied.
- During the instruction execution, each bit being shifted out is checked when it is the bit adjacent to the sign bit. If the bit differs from the sign, it cannot be shifted out without causing a fixed-point overflow program exception.

Before shifting two bits left:



Shifting left three bits in this register pair causes a fixed-point overflow program exception, since the third bit being shifted is not the same value as the sign bit. In this example, two bits are successfully shifted out, but when the third bit is moved adjacent to the sign bit and tested, it is not like the sign.

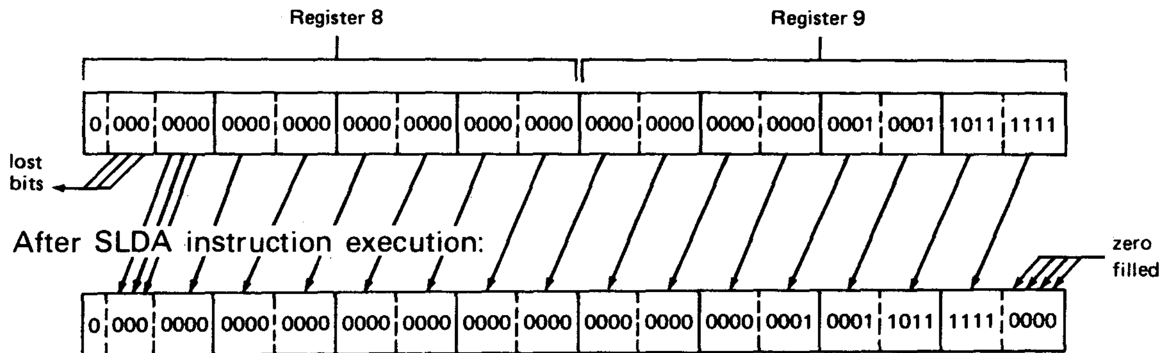
- Each time you shift one digit left, it is the same as multiplying by a power of 2. If you shift one bit left, you multiply by 2, two bits left, you multiply by 2^2, three bits by 2^3, and so forth.
- When the shift value is zero, it causes a double-length sign and magnitude test, and the condition code is set.

Example:

```
LABEL     ΔOPERATIONΔ                    OPERAND
1          10      16
---------------------------------------------------
           L       9,FULLWORD
           SLDA    8,4
           .
           .
           .
FULLWORD   DC      F'4543'
```

In this example, registers 8 and 9 are shifted four bit positions left. Before the SLDA instruction, the content of FULLWORD is placed into register 9. Operand 2 is expressed in explicit format with 4 as the displacement (d_2) and no base register (b_2) representation. Consequently, the addition of base register and displacement values is not performed by the assembler and the operand 2 displacement value becomes the absolute value.

Before execution of SLDA instruction:



Note that the content of register 9 before the shift is 4543, and after shifting four bits left, it contains 72,688 (4543 multiplied by 2^4(16)).

10.25. SHIFT LEFT SINGLE (SLA)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SLA	8B	RS	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *shift left single* (SLA) instruction shifts the 31 bits of the operand 1 register to the left of the number of bits specified by the low order six bits of the operand 2 address. The sign bit (the leftmost high order bit) of register 1 remains unchanged, and zeros fill the vacated positions of the register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLA	$r_1, d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLA	r_1, s_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.

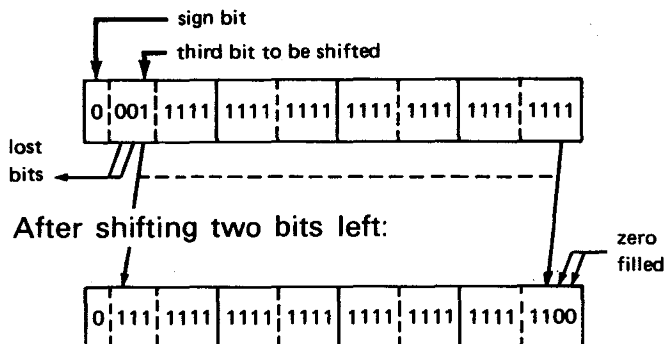
- The main storage address or label you specify in operand 2 is not changed by the SLA instruction execution. Notice the formats indicate that you cannot specify a length in operand 2.

- The sign bit (leftmost bit) of the operand 1 register is not shifted or changed by the execution of this instruction. Only the 31 remaining bits can be shifted by the execution of this instruction.

- After the requested number of bits are shifted out of the operand 1 register, zeros fill the vacated rightmost bit positions.

- During the instruction execution, each bit being shifted out is checked when it is the bit adjacent to the sign bit. If the bit differs from the sign, it cannot be shifted out without causing a fixed-point overflow program exception.

  Before shifting two bits left:



  Shifting left three bits in this register causes a fixed-point overflow program exception, since the third bit being shifted is not the same value as the sign bit. In this example, two bits are successfully shifted out, but when the third bit is moved adjacent to the sign bit and tested, it is not like the sign.

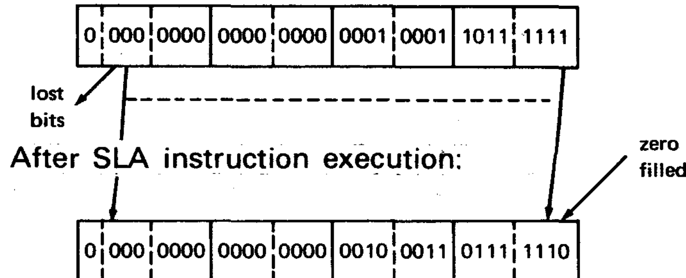
- For numbers with a value of less than 2^{30} (1,073,741,824), each time you shift one digit left, it is the same as multiplying by a power of 2. If you shift one bit left, you multiply by 2^1; two bits left, you multiply by 2^2; three bits by 2^3; and so forth.

Example:

```
LABEL     ΔOPERATIONΔ                                        OPERAND
1          10          16
-----------------------------------------------------------------------
           L           8,FULLWORD
           SLA         8,1
           .
           .
           .
FULLWORD   DC          F'4543'
```

In this example, the content of main storage location FULLWORD is placed in register 8. Register 8 is then shifted one bit position left.

Register 8 before execution of SLA instruction:



Note that register 8 contains 4543 before the SLA instruction and 9086 afterwards. By shifting one bit left, the content of register 8 is multiplied by 2.

SRDA

10.26. SHIFT RIGHT DOUBLE (SRDA)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SRDA	8E	RS	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *shift right double* (SRDA) instruction shifts the 63 bits of operand 1 to the right the number of bits specified by the low order six bits of the operand 2 address. You cannot shift the sign bit. Specify the even-numbered register of the pair as operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRDA	$r_1, d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRDA	r_1, s_2

Operational Considerations:

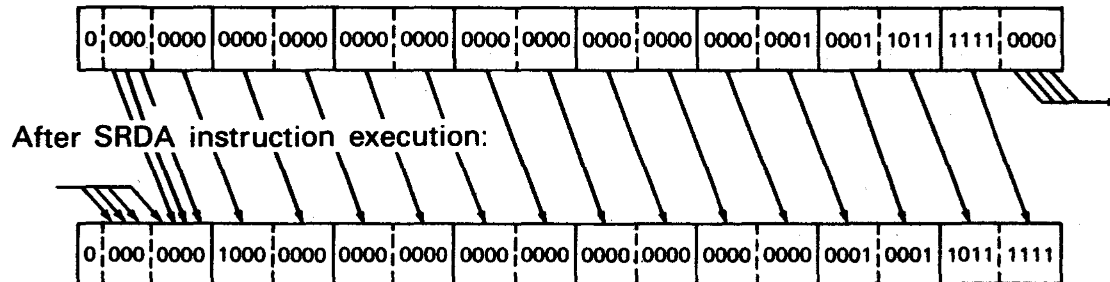
- Any pair of general registers (0 through 15) can be used as operand 1, which is an even-numbered register pair. You must specify the even-numbered register of the pair as operand 1.
- The main storage address or label you specify in operand 2 is not changed by the SRDA instruction execution. Notice the formats indicate that you cannot specify a length in operand 2.
- The sign bit (leftmost bit) of the even-numbered operand 1 register pair is not moved or changed by the execution of this instruction. Only the 63 remaining bits can be shifted by this instruction.
- After the requested number of bits are shifted-out of the operand 1 register pair, the vacated leftmost bit positions are filled with bits that have the same value as the sign bit.
- For positive values, each time you shift one bit position right, it is the same as dividing by a power of 2. If you shift one bit right, you divide by 2^1; two bits right, you divide by 2^2; three bits by 2^3 and so forth. When the value is negative, shifting right causes a divide by 2 on a value one less than the value in the register. For examples, see SRA instruction.
- When the shift value is zero, it causes a double-length sign and magnitude test, and the condition code is set.

Example:

```
LABEL     ΔOPERATIONΔ                    OPERAND
1          10        16
-----------------------------------------------------------
          L          9,FULLWORD
          SRDA       8,4
          .
          .
          .
FULLWORD  DC         F'72688'
```

In this example, registers 8 and 9 are shifted four bit positions right. Before the SRDA instruction, the contents of FULLWORD are placed into register 9.

Before SRDA instruction execution:



Notice that the contents of register 9 before the shift are 72,688, and after shifting four bits right, it contains 4543 (72,688 divided by 2^4(16)).

10.27. SHIFT RIGHT SINGLE (SRA)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SRA	8A	RS	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *shift right single* (SRA) instruction shifts the 31 bits of the operand 1 register to the right the number of bits specified by the low order six bits of the operand 2 address. You cannot shift the sign bit.

Explicit Format:

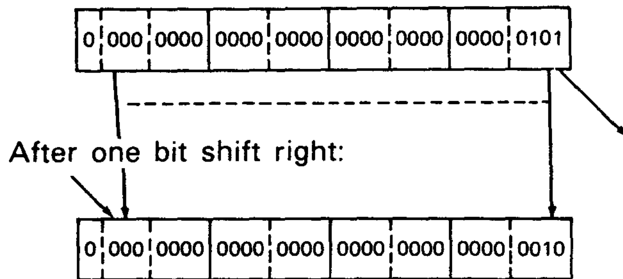
LABEL	△OPERATION△	OPERAND
[symbol]	SRA	$r_1, d_2(b_2)$

Implicit Format:

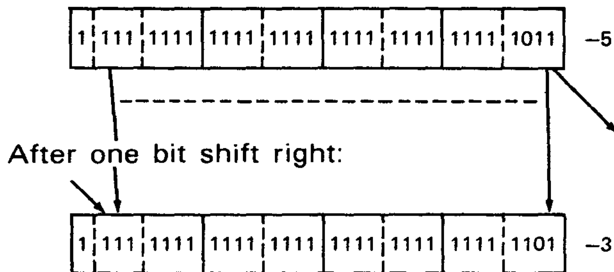
LABEL	△OPERATION△	OPERAND
[symbol]	SRA	r_1, s_2

Operational Considerations:

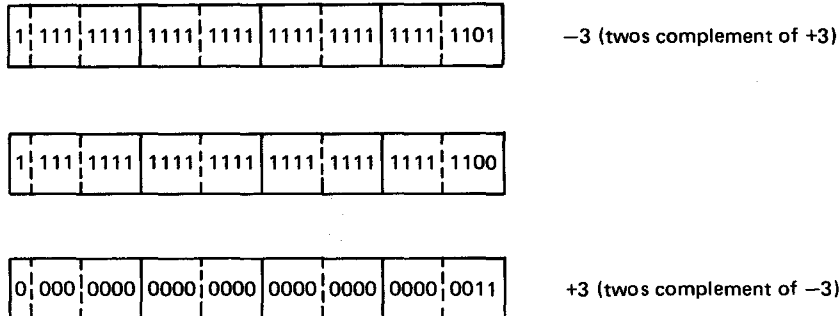
- Any or the general registers (0 through 15) can be used as operand 1.
- The main storage address or label you specify in operand 2 is not changed by the SRA instruction execution. Notice the formats indicate that you cannot specify a length in operand 2.
- The sign bit (leftmost bit) of the operand 1 register is not shifted or changed by the execution of this instruction. Only the 31 remaining bits can be shifted by the execution of this instruction.
- After the requested number of bits are shifted out of the operand 1 register, the vacated leftmost bits are filled with bits that have the same value as the sign bit.
- If the contents of the operand 1 register are a positive value, shifting right one bit divides the value of the register by 2. Any remainders are rounded downward. For example, a value of +5 shifted right one bit produces a +2 in the register after the shift.



When contents of the operand 1 register are negative, shifting right one bit causes a divide by 2 on a value one less than the value in the operand 1 register. For example, a value of —5, when shifted right one bit produces a —3 in the register (—6 divided by 2).



Note that a negative is expressed in twos complement notation. After the shifting operation, you can determine the positive value in the register by taking the twos complement of a negative number.

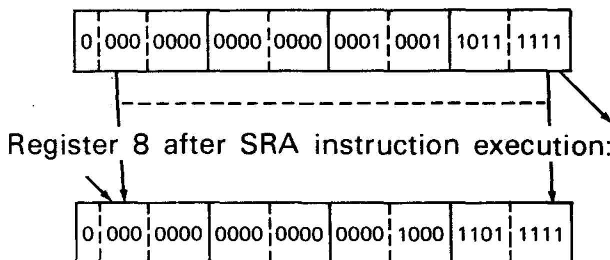


Example:

```
LABEL     ΔOPERATIONΔ                          OPERAND
1          10         16
--------------------------------------------------------------
          L          8,FULLWORD
          SRA        8,1
          .
          .
          .
FULLWORD  DC         F'4543'
```

In this example, the content of main storage location FULLWORD is placed in register 8. Register 8 is then shifted one bit position right.

Register 8 before SRA instruction execution:



Note that register 8 contains 4543 before the SRA instruction and 2271 afterwards. By shifting one bit right, the content of register 8 is divided by 2.

ST

10.28. STORE (ST)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ST	50	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *store* (ST) instruction places the contents of the operand 1 register unchanged into the full word in operand 2.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ST	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ST	$r_1,s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must either be defined as a full word or aligned on a full-word boundary.
- Unlike most instructions, the ST instruction has operand 1 as the sending field and operand 2 as the receiving field.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10        16
--------------------------------------------------------
         L         7,=F'25'
         .
         .
         .
         ST        7,CHART+4
         .
         .
         .
CHART    DC        F'22'
         DC        F'5Ø'
         DC        F'28'
```

CHART+4 before execution of ST instruction:

0000	0000	0000	0000	0000	0000	0011	0010	binary
0	0	0	0	0	0	3	2	hex

Register 7 before and after execution of ST instruction:

0000	0000	0000	0000	0000	0000	0001	1001	binary
0	0	0	0	0	0	1	9	hex

CHART+4 after execution of ST instruction:

0000	0000	0000	0000	0000	0000	0001	1001	binary
0	0	0	0	0	0	1	9	hex

In this example, register 7 is loaded with a full-word value of 25. Then the content of register 7 destroys the content of the second full word in CHART and replaces it with the content of register 7.

STH

10.29. STORE HALF WORD (STH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STH	40	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *store half word* (STH) instruction places bits 16 through 31 of the operand 1 register unchanged into the half word in operand 2.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STH	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STH	$r_1,s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a half word or aligned on a half-word boundary.
- Unlike most instructions, the STH instruction has operand 1 as the sending field and operand 2 as the receiving field.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10       16
--------------------------------------------------------------
         LA       7,HALFBUF
         LH       6,=H'43'
         CH       6,CONSTANT
         BNE      ERROR
         .
         .
         .
ERROR    STH      6,Ø(7)
         A        7,=F'2'
         .
         .
         .
CONSTANT DC       H'5Ø'
         DS       ØH
HALFBUF  DS       CL8Ø
```

HALFBUF (2) before execution of STH instruction:

0000	0000	0000	0001	binary
0	0	0	1	hex

(leftover data from previous program)

Register 6 before and after execution of STH instruction:

0000	0000	0000	0000	0000	0000	0010	1011	binary
0	0	0	0	0	0	2	B	hex

Bits 16 — 31

HALFBUF (2) after execution of STH instruction:

0000	0000	0010	1011	binary
0	0	2	B	hex

In this example, the address of HALFBUF is loaded into register 7, and the half-word decimal value of 43 is loaded into register 6. Then, the content of register 6 is compared to the half-word decimal value in CONSTANT. Since the value 43 is less than 50, the condition code is set to 1 and the branch to the instruction labeled ERROR takes place. There, bits 16 through 31 of register 6 are stored in the first two bytes of HALFBUF. A full word of 2 is then added to the address in register 7 which increases the address by 2 bytes. This makes it possible for the next unequal condition to be stored in the succeeding two bytes and so on.

STM

10.30. STORE MULTIPLE (STM)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STM	**90**	**RS**	**4**

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *store multiple* (STM) instruction places the contents of two or more consecutive registers (operands 1 and 3) into an equal number of consecutive full words in main storage (operand 2).

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STM	$r_1,r_3,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STM	r_1,r_3,s_2

The operand 1 register is the first register stored and the operand 3 register is the last register stored. If operands 1 and 3 are not consecutive, any registers consecutive to the operand 1 register up to and including the operand 3 register are also included. If the address of the operand 3 register is less than the address of the operand 1 register, the register addresses wrap around from 15 to 0. The contents of the registers are stored in ascending sequence into an equal number of consecutive full words in main storage starting with the byte addressed by the operand 2 label, and continuing with as many full words that are needed to receive the contents of the registers specified.

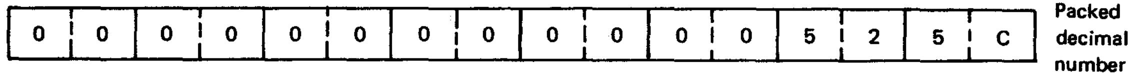
Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1 and operand 3.
- Operand 2 must either be defined as a full word or aligned on a full-word boundary.
- If operand 2 does not reference the correct number of full words needed to receive the contents of all the registers specified, full words consecutive to the first full word specified by operand 2 are filled until the contents of the operand 3 register has been stored.
- When storing multiple registers, the wraparound concept applies.
- If operand 1 and operand 3 reference the same register, only the contents of that register is stored in the first full word of operand 2.

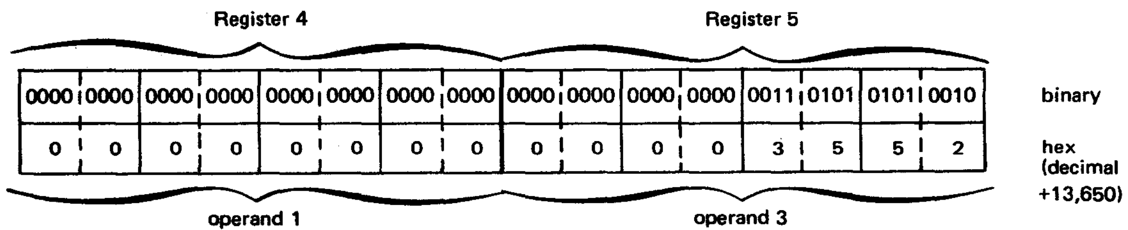
Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10          16
-----------------------------------------------------------------
          ZAP         DWORD,=P'525'
          CVB         5,DWORD
          M           4,MULTPLYR
          STM         4,5,DWORD
          .
          .
          .
DWORD     DS          D
MULTPLYR  DC          F'26'
```

DWORD before execution of STM instruction:



Registers 4 and 5 before and after execution of STM instruction:



DWORD after execution of STM instruction:

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0011	0101	0101	0010	binary
0	0	0	0	0	0	0	0	0	0	0	0	3	5	5	2	hex

In this example, the packed decimal number 525 is added to DWORD which was previously cleared to zero. Then, the double word in DWORD is converted into its binary equivalent and the result is placed in register 5. The contents of registers 4 and 5 are then multiplied by the full word in MULTPLYR and the result replaces registers 4 and 5 as a double-word value. Register 4 (operand 1) and register 5 (operand 3) are stored in the first two full words in DWORD.

10.31. SUBTRACT (S)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
S	5B	RX	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ■ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *subtract* (S) instruction subtracts the contents of the full word in operand 2 from the contents of the operand 1 register. The difference replaces the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	S	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	S	$r_1,s_2(x_2)$

When the actual subtraction takes place, the twos complement form of operand 2 is added to operand 1. The sign of the result is determined algebraically.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a full word or aligned on a full-word boundary.

Example:

```
LABEL    ΔOPERATIONΔ                           OPERAND
1        10          16
--------------------------------------------------------------
         L           5,=F'52'
         S           5,VALUE
         .
         .
         .
VALUE    DC          F'11'
```

Register 5 before execution of S instruction:

0000	0000	0000	0000	0000	0000	0011	0100	binary
0	0	0	0	0	0	3	4	hex

VALUE before and after execution of S instruction:

0000	0000	0000	0000	0000	0000	0000	1011	binary
0	0	0	0	0	0	0	B	hex

Register 5 after execution of S instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	2	9	hex

10.32. SUBTRACT (SR)

General				Possible Program Exceptions	
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)	☐ ADDRESSING	☐ PROTECTION
MNEM.	HEX.			☐ DATA (INVALID SIGN/DIGIT)	☐ SIGNIFICANCE
SR	1B	RR	2	☐ DECIMAL DIVIDE	☐ SPECIFICATION:
Condition Codes				☐ DECIMAL OVERFLOW	☐ NOT A FLOATING-POINT REGISTER
■ IF RESULT = 0, SET TO 0				☐ EXECUTE	☐ OP 1 NOT ON HALF-WORD BOUNDARY
■ IF RESULT <0, SET TO 1				☐ EXPONENT OVERFLOW	☐ OP 2 NOT ON HALF-WORD BOUNDARY
■ IF RESULT >0, SET TO 2				☐ EXPONENT UNDERFLOW	☐ OP 2 NOT ON FULL-WORD BOUNDARY
■ IF OVERFLOW, SET TO 3				☐ FIXED-POINT DIVIDE	☐ OP 2 NOT ON DOUBLE-WORD BOUNDARY
☐ UNCHANGED				■ FIXED-POINT OVERFLOW	☐ OP 1 NOT EVEN NUMBERED REGISTER
				☐ FLOATING-POINT DIVIDE	☐ OP 1 NOT ODD NUMBERED REGISTER
				☐ OPERATION	☐ NONE

The *subtract* (SR) instruction subtracts the contents of the operand 2 register from the contents of the operand 1 register and places the difference in the operand 1 register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SR	r_1,r_2

When the actual subtraction takes place, the twos complement form of operand 2 is added to operand 1. The sign of the result is determined algebraically.

Operational Considerations:

- Any of the general registers can be used as operands 1 and 2.
- This instruction can be used to clear a register by subtracting the content of the register from itself.
- The subtraction is performed by converting the number in operand 2 (r_2) into a signed twos complement binary number and then algebraically adding it to the value in operand 1 (r_1).
- The maximum fixed-point number that can be contained in a 32-bit register is 2,147,483,647($2^{31}-1$); the minimum number is −2,147,483,648(-2^{31}). For decimal numbers outside this range, an overflow condition is produced.
- The contents of operand 2 (r_2) are not changed by the subtract (SR) instruction.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10         16
-----------------------------------------------------------
         LM         5,6,HRS
         SR         4,4
         A          5,HRSOT
         M          4,RATE
         SR         5,6
         ST         5,FLWRD
         .
         .
         .
HRS      DC         F'4Ø'
DEDUC    DC         F'2916'
HRSOT    DC         F'3'
RATE     DC         F'35Ø'
FLWRD    DS         F
```

Register 4 after execution of first SR instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	0	0	hex

Registers 4 and 5 before execution of second SR instruction:

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0011	1010	1100	1010	binary
0	0	0	0	0	0	0	0	0	0	0	0	3	A	C	A	hex

Register 6 before and after execution of SR instruction:

0000	0000	0000	0000	0000	1011	0110	0100	binary
0	0	0	0	0	B	6	4	hex

Registers 4 and 5 after execution of second SR instruction:

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0010	1111	0110	0101	binary
0	0	0	0	0	0	0	0	0	0	0	0	2	F	6	6	hex

In this example, two full words (starting with the byte addressed by HRS and including the full word in DEDUC) are loaded into registers 5 and 6. The SR instruction clears register 4 to zeros and the content of HRSOT is added to the content of register 5. Register 5 now contains the standard 40 working hours per week plus any overtime hours. The content of RATE is multiplied by the even-odd register pair. The product replaces registers 4 and 5 as a double-word value. The content of register 6 is subtracted from register 5 (since the product is contained in one register) and the difference replaces register 5. Register 5 now contains one employee's weekly net pay. The content of register 5 is then stored in FLWRD in main storage.

SH

10.33. SUBTRACT HALF WORD (SH)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SH	4B	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
■ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *subtract half word* (SH) instruction subtracts the contents of the half word in operand 2 from the contents of the operand 1 register. The difference replaces the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SH	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SH	$r_1,s_2(x_2)$

Operand 2 is two bytes in length (a 16-bit signed integer) and is located in main storage. Before operand 2 is subtracted from operand 1, operand 2 is temporarily expanded to 32 bits by propagating the sign bit through the high order 16 bit positions. Then the twos complement of operand 2 is added to operand 1. The difference replaces the content of the operand 1 register.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must either be defined as a half word or aligned on a half-word boundary.
- A fixed-point overflow condition can occur.
- Execution of the SH instruction sets the condition code accordingly.

Example:

```
LABEL   ΔOPERATIONΔ                         OPERAND
1        10        16
--------------------------------------------------------
         LA        5,2
         L         9,BADACTS
         CH        5,CODECD
         BE        GDACTS
         .
         .
         .
GDACTS   SH        9,VALUE1
         ST        9,YTDACTS
         .
         .
         .
CODECD   DC        H'2'
BADACTS  DC        F'3'
VALUE1   DC        H'1'
YTDACTS  DS        F
```

Register 9 before execution of SH instruction:

0000	0000	0000	0000	0000	0000	0000	0011	binary
0	0	0	0	0	0	0	3	hex

VALUE1 before and after execution of SH instruction:

before expansion (last four hex digits)

0000	0000	0000	0000	0000	0000	0000	0001	binary
0	0	0	0	0	0	0	1	hex

after expansion (all eight hex digits)

Register 9 after execution of SH instruction:

0000	0000	0000	0000	0000	0000	0000	0010	binary
0	0	0	0	0	0	0	2	hex

In this example, the decimal value 2 is loaded into register 5 and the content of BADACTS is loaded into register 9. Then the content of register 5 is compared to the half-word value of CODECD. Since an equal to condition exists, the condition code is set to 0. As a result, the following branch to the instruction labeled GDACTS takes place. There, the half word in VALUE1 is expanded to a 32-bit signed integer, and the twos complement form of VALUE1 is added to register 9. The difference occupies register 9 as a full-word value. Finally, the ST instruction stores the contents of register 9 in YTDACTS located in main storage.

11. Floating-Point Instructions

11.1. INTRODUCTION

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. 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 multiple* (ME) instruction, only the most significant word of the double-word register is involved in the operation. The least significant word remains unchanged. Separate instructions are provided for operations with long and short format operands.

Each operand is treated as a floating-point number consisting of a biased exponent (characteristic) and a signed fraction (mantissa). The biased exponent is expressed in excess-64 binary notation; the fraction is expressed as a hexadecimal number having an arithmetic point to the left of the high-order digit. The quantity expressed by the full floating-point number is the product of the fraction and the number 16 raised to the power of the biased exponent minus 64 (fraction times 16^{n-64}).

A quantity may be represented with the greatest precision by a floating-point number of a given fraction length when the number is in a "normalized" form. A normalized floating-point number has a nonzero, high order hexadecimal fraction digit.

An exponent overflow exception develops if, in the result of a floating-point instruction, the characteristic of the result exceeds 127 and the fraction of the result is not zero. An exponent underflow exception develops if the characteristic is less than zero and the fraction of the result is not zero. An exponent overflow exception causes a program interruption. An exponent underflow exception causes a program interruption if the exponent underflow mask bit of the current PSW is 1.

A floating-point number having a zero characteristic, a zero fraction, and a positive (zero) sign is said to be a "true zero" number.

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.

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 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. (See 2.1, 2.3, 2.6, 5.1, 5.2.12, Appendix C, and Appendix D.)

11.2. ADD NORMALIZED, LONG FORMAT (AD)

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
AD	6A	RX	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *add normalized, long format* (AD) instruction causes the contents of the double word in storage specified by operand 2 to be algebraically added to the contents of the double-word register specified by operand 1 (r_1). The sum is normalized and placed in the operand 1 (r_1) register.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	AD	$r_1,d_2(x_2,b_2)$
ADLONG	AD	R4,50(R7,R8)

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	AD	$r_1,s_2(x_2)$
ADLONG	AD	R4,FAM

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-over digit 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, and a possible carry-over digit. If any most signficant digits of the intermediate sum are zero, 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 zero, an exponent underflow condition results. If the exponent underflow mask bit (38) 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 current value. If the exponent underflow mask of the current PSW is zero, the result is a true zero.

- Zero Result

  If the intermediate sum, including the guard digit, is zero, a significance exception exists. If the significance mask bit (39) 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 zero and the intermediate sum is zero, the result is made a true zero. Exponent underflow cannot occur for a zero fraction.

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

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10          16

ADLONG    AD          R4,FAM
FAM       DC          D'1ØØ'
```

Before execution of the *add normalized, long format* (AD) instruction, if we assume a value of +50 in R4, the contents of R4 and storage area FAM will be:

R4 before execution:

4	2	3	2	0	0	0	0	0	0	0	0	0	0	0	0	+50

FAM before and after execution:

4	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	+100

R4 after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

ADR

11.3. ADD NORMALIZED, LONG FORMAT (ADR)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ADR	2A	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *add normalized, long format* (ADR) instruction causes the contents of the double-word register specified by operand 2 (r_2) to be algebraically added to the contents of the double-word register specified by operand 1 (r_1). The sum is normalized and placed in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ADR	r_1,r_2
ADLONG	ADR	R4,R6

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 a 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-over digit 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, and a possible carry-over digit. If any most significant digits of the intermediate sum are zero, 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 zero, an exponent underflow condition results. If the exponent underflow mask bit (38) 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 zero, the result is a true zero.

- Zero Result

  If the intermediate sum, including the guard digit, is zero, a significance exception exists. If the significance mask bit (39) 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 zero and the intermediate sum is zero, the result is made a true zero. Exponent underflow cannot occur for a zero fraction.

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

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10          16
------------------------------------------------------------
ADLONG    ADR         R4,R6
```

Before execution of the *add normalized, long format* (ADR) instruction, if we assume a value of +50 in R4 and +100 in R6, the contents of the R4 and R6 will be:

R4 before execution:

4	2	3	2	0	0	0	0	0	0	0	0	0	0	0	0	+50

R6 before and after execution:

4	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	+100

R4 after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

11.4. ADD NORMALIZED, SHORT FORMAT (AE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AE	7A	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *add normalized, short format* (AE) instruction causes the contents of the full word in storage specified by operand 2 to be algebraically added to the contents of a full word in the register specified by operand 1 (r_1). The sum is normalized and placed in the full word in the operand 1 (r_1) register.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	AE	$r_1,d_2(x_2,b_2)$
ADSHORT	AE	R4,50(R7,R8)

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol	AE	$r_1,s_2(x_2)$
ADSHORT	AE	R4,FAM

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-over 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-over digit. If any most significant digits of the intermediate sum are zero, 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 zero, an exponent underflow condition results. If the exponent underflow mask bit (38) 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 zero, the result is a true zero.

- Zero Result

  If the intermediate sum, including the guard digit, is zero, a significance exception exists. If the significance mask bit (39) 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 zero and the intermediate sum is zero, the result is made a true zero. Exponent underflow cannot occur for a zero fraction.

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

Example:

```
LABEL    ΔOPERATIONΔ                OPERAND
1        10        16
------------------------------------------------
ADSHORT  AE        R4,FAM
FAM      DC        E'1ØØ'
```

Before execution of the *add normalized, short format* (AE) instruction, if we assume a value of +50 in R4, the contents of R4 and storage area FAM will be:

R4 before execution:

4	2	3	2	0	0	0	0	+50

FAM before and after execution:

4	2	6	4	0	0	0	0	+100

R4 after execution:

4	2	9	6	0	0	0	0	+150

AER

11.5. ADD NORMALIZED, SHORT FORMAT (AER)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AER	**3A**	**RR**	**2**

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *add normalized, short format* (AER) instruction causes the contents of a full word in the register specified by operand 2 (r_2) to be algebraically added to a full word in the register specified by operand 1 (r_1). The sum is normalized and placed in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AER	r_1,r_2
ADSHORT	AER	R2,R4

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-over digit 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-over digit. If any most significant digits of the intermediate sum are zero, 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 zero, an exponent underflow condition results. If the exponent underflow mask bit (38) 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 zero, the result is a true zero.

- Zero Result

  If the intermediate sum, including the guard digit, is zero, a significance exception exists. If the significance mask bit (39) 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 zero and the intermediate sum is zero, the result is made a true zero. Exponent underflow cannot occur for a zero fraction.

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

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10        16
-------------------------------------------------------
ADSHORT  AER        R2,R4
```

Before execution of the *add normalized, short format* (AER) instruction, if we assume a value of +50 in R2 and +100 in R4, contents of R2 and R4 will be:

R2 before execution:

4	2	3	2	0	0	0	0	+50

R4 before and after execution:

4	2	6	4	0	0	0	0	+100

R2 after execution:

4	2	9	6	0	0	0	0	+150

11.6. ADD UNNORMALIZED, SHORT FORMAT (AU)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AU	7E	RX	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *add unnormalized, short format* (AU) instruction causes the contents of the full word in storage specified by operand 2 to be algebraically added to the contents of a full word in the register specified by operand 1 (r_1). The sum is placed in the operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AU	$r_1,d_2(x_2,b_2)$
ADSHORT	AU	R4,50(R7,R8)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AU	$r_1,s_2(x_2)$
ADSHORT	AU	R4,FAM

Operational Consideration:

- The execution of the AU instruction is identical to the AE instruction (11.4) except that the sum is not normalized before being placed in operand 1.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10        16
---------------------------------------------------------
ADSHORT  AU         R4,FAM
FAM      DC         E'1ØØ'
```

Before execution of the *add unnormalized, short format* (AU) instruction, if we assume a value of +900 in R4, the contents of R4 and main storage area FAM will be:

R4 before execution:

4	4	0	3	8	4	0	0	+900

FAM before and after execution:

4	2	6	4	0	0	0	0	+100

R4 after execution:

4	4	0	3	E	8	0	0	+1000

11.7. ADD UNNORMALIZED, SHORT FORMAT (AUR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AUR	3E	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *add unnormalized, short format* (AUR) instruction causes the contents of a full word in the register specified by operand 2 (r_2) to be algebraically added to a full word in the register specified by operand 1 (r_1). The sum is placed in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	AUR	r_1,r_2
ADSHORT	AUR	R2,R4

Operational Consideration:

- The execution of the AUR instruction is identical to the AER instruction (11.5), except that the sum is not normalized before being placed in operand 1.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1        10         16
-------------------------------------------------------------
ADSHORT  AUR        R2,R4
```

Before execution of the *add unnormalized, short format* (AUR) instruction, if we assume a value of +900 in R2 and +100 in R4, the contents of R2 and R4 will be:

R2 before execution:

4	4	0	3	8	4	0	0

+900

R4 before and after execution:

4	4	0	0	6	4	0	0

+100

R2 after execution:

4	4	0	3	E	8	0	0

+1000

AW

11.8. ADD UNNORMALIZED, LONG FORMAT (AW)

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
AW	6E	RX	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *add unnormalized, long format* (AW) instruction causes the contents of a double word in storage specified by operand 2 to be algebraically added to the contents of the double word in the register specified by operand 1 (r_1). The sum is placed in the double word in the register specified by operand 1 (r_1).

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AW	$r_1,d_2(x_2,b_2)$
ADLONG	AW	R4,50(R7,R8)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AW	$r_1,s_2(x_2)$
ADLONG	AW	R4,FAM

Operational Consideration:

- The execution of the AW instruction is identical to the AD instruction (11.2) except that the sum is not normalized before being placed in operand 1 (r_1).

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1        10        16
-----------------------------------------------------------
ADLONG   AW        R4,FAM
FAM      DC        D'1ØØ'
```

Before execution of the *add unnormalized, long format* (AW) instruction, if we assume a value of +900 in R4, the contents of R4 and storage area FAM will be:

R4 before execution:

4	4	0	3	8	4	0	0	0	0	0	0	0	0	0	0	+900

FAM before and after execution:

4	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	+100

R4 after execution:

4	4	0	3	E	8	0	0	0	0	0	0	0	0	0	0	+1000

AWR

11.9. ADD UNNORMALIZED, LONG FORMAT (AWR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AWR	2E	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *add unnormalized, long format* (AWR) instruction causes the contents of the double-word register specified by operand 2 (r_2) to be algebraically added to the double-word contents of operand 1 (r_1). The sum is placed in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AWR	r_1,r_2
ADLONG	AWR	R4,R6

Operational Consideration:

- The execution of the AWR instruction is identical to the ADR instruction (11.3) except that the sum is not normalized before being placed in operand 1 (r_1).

Example:

```
LABEL    △OPERATION△                         OPERAND
1         10          16
-------------------------------------------------------------
ADLONG    AWR         R4,R6
```

Before execution of the *add unnormalized, long format* (AWR) instruction, if we assume a value of +900 in R4 and +100 in R6, the contents of R4 and R6 will be:

R4 before execution:

4	4	0	3	8	4	0	0	0	0	0	0	0	0	0	0	+900

R6 before and after execution:

4	4	0	0	6	4	0	0	0	0	0	0	0	0	0	0	+100

R4 after execution:

4	4	0	3	E	8	0	0	0	0	0	0	0	0	0	0	+1000

11.10. COMPARE, LONG FORMAT (CD)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CD	69	RX	4

Condition Codes
■ IF OP1 = OP2, SET TO 0
■ IF OP1 < OP2, SET TO 1
■ IF OP1 > OP2, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *compare, long format* (CD) instruction causes the contents of a double word in the register specified by operand 1 (r_1) to be algebraically compared with the contents of a double word in storage specified by operand 2. The condition code is set by this instruction.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CD	$r_1,d_2(x_2,b_2)$
COMPAR	CD	R2,50(R7,R9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CD	$r_1,s_2(x_2)$
COMPAR	CD	R2,FAM

Operational Considerations:

- Comparison is accomplished by the rules for normalized ~~fixed~~ floating-point subtraction. The operands are equal when the intermediate sum, including the guard digit, is zero.
- Operands with zero fractions compare as equal even when their signs or exponents are different.
- The condition code is set:
  - — to zero when operand 1 equals operand 2;
  - — to 1 when operand 1 is less than operand 2; and
  - — to 2 when operand 1 is greater than operand 2.

  Code 3 is not used.

Examples:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10       16
---------------------------------------------------------------
COMPAR1  CD       R2,FAM3
COMPAR2  CD       R2,FAM32
COMPAR3  CD       R2,FAM33
FAM3     DC       D'3'
FAM32    DC       D'32'
FAM33    DC       D'33'
```

Before execution of the *compare, long format* (CD) instruction, if we assume a value of +32 in R2, then:

- Example 1 will set a condition code of 2.
- Example 2 will set a condition code of zero.
- Example 3 will set a condition code of 1.

11.11. COMPARE, LONG FORMAT (CDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CDR	29	RR	2

Condition Codes

- ■ IF OP1 = OP2, SET TO 0
- ■ IF OP1 < OP2, SET TO 1
- ■ IF OP1 > OP2, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *compare, long format* (CDR) instruction causes the contents of a double word in the register specified by operand 1 (r_1) to be algebraically compared with the contents of a double word in the register specified by operand 2 (r_2). The condition code is set by this instruction.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CDR	r_1,r_2
COMPAR	CDR	R2,R6

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 zero.

- Operands with zero fractions compare as equal even when their signs or exponents are different.

- The condition code is set:

  — to zero when operand 1 equals operand 2;

  — to 1 when operand 1 is less than operand 2; and

  — to 2 when operand 1 is greater than operand 2.

  Code 3 is not used.

Examples:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10        16
------------------------------------------------------------
COMPAR1  CDR       R2,R6
COMPAR2  CDR       R2,R4
COMPAR3  CDR       R2,RØ
```

Before execution of the *compare, long format* (CDR) instruction, if we assume values of +32 in R2, +33 in R6, +32 in R4, and +0 in R0, then:

- Example 1 will set a condition code of 1.
- Example 2 will set a condition code of zero.
- Example 3 will set a condition code of 2.

11.12. COMPARE, SHORT FORMAT (CE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CE	79	RX	4

Condition Codes
■ IF OP1 = OP2, SET TO 0
■ IF OP1 < OP2, SET TO 1
■ IF OP1 > OP2, SET TO 2
□ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *compare, short format* (CE) instruction causes the contents of a full word in the register specified by operand 1 (r_1) to be algebraically compared with the contents of a full word in storage specified by operand 2. The condition code is set by this instruction.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CE	$r_1,d_2(x_2,b_2)$
COMPAR	CE	R2,50(R5,R7)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CE	$r_1,s_2(x_2)$
COMPAR	CE	R2,FAM

Operational Considerations:

- Comparison is accomplished by the rules for normalized ~~fixed~~ floating-point subtraction. The operands are equal when the intermediate sum, including the guard digit, is zero.
- Operands with zero fractions compare as equal even when their signs or exponents are different.
- The condition code is set:
  - to zero when operand 1 equals operand 2;
  - to 1 when operand 1 is less than operand 2; and
  - to 2 when operand is greater than operand 2.

  Code 3 is not used.

Examples:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1        10        16
---------------------------------------------------
COMPAR1  CE        R2,FAM3
COMPAR2  CE        R2,FAM32
COMPAR3  CE        R2,FAM33
FAM3     DC        E'3'
FAM32    DC        E'32'
FAM33    DC        E'33'
```

Before execution of the *compare, short format* (CE) instructions, if we assume a value of +32 in R2, then:

- Example 1 will set a condition code of 2.
- Example 2 will set a condition code of zero.
- Example 3 will set a condition code of 1.

11.13. COMPARE, SHORT FORMAT (CER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CER	39	RR	2

Condition Codes

- ■ IF OPI = OP2, SET TO 0
- ■ IF OPI < OP2, SET TO 1
- ■ IF OPI > OP2, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *compare, short format* (CER) instruction causes the full-word contents of the register specified by operand 1 (r_1) to be algebraically compared with the contents of a full word in the register specified by operand 2 (r_2). The condition code is set by this instruction.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CER	r_1,r_2
COMPAR	CER	R4,R6

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 zero.
- Operands with zero fractions compare as equal even when their signs or exponents are different.
- The condition code is set:
  - — to zero when operand 1 equals operand 2;
  - — to 1 when operand 1 is less than operand 2; and
  - — to 2 when operand 1 is greater than operand 2.

  Code 3 is not used.

Examples:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10      16

COMPAR1  CER      R4,RØ
COMPAR2  CER      R4,R2
COMPAR3  CER      R4,R6
```

Before execution of the *compare, short format* (CER) instructions, if we assume values of +32 in R4, +3 in RO, +32 in R2, and +33 in R6, then:

- Example 1 will set a condition code of 2.
- Example 2 will set a condition code of zero.
- Example 3 will set a condition code of 1.

DD

11.14. DIVIDE, LONG FORMAT (DD)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
DD	6D	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *divide, long format* (DD) instruction causes the double-word contents of the operand 1 (r_1) register to be divided by the contents of the double word in storage specified by operand 2. The normalized quotient is placed in the register specified by operand 1 (r_1). Any remainder is not preserved.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DD	$r_1,d_2(x_2,b_2)$
DIVLONG	DD	R4,33(R7,R10)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DD	$r_1,s_2(x_2)$
DIVLONG	DD	R4,FAM

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 (r_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 zero, 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 zero, the result is made a true zero. 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 of the current PSW is 1.

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

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10         16
----------------------------------------------------------
DIVIDLG  DD          R4,FAM
FAM      DC          D'5'
```

Before execution of the *divide, long format* (DD) instruction, if we assume a value of +1000 in R4, the contents of R4 and storage area FAM will be:

R4 before execution:

4	3	3	E	8	0	0	0	0	0	0	0	0	0	0	0	+1000

FAM before and after execution:

4	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	+5

R4 after execution:

4	2	C	8	0	0	0	0	0	0	0	0	0	0	0	0	+200

DDR

11.15. DIVIDE, LONG FORMAT (DDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
DDR	2D	RR	2

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *divide, long format* (DDR) instruction causes the double-word contents of the operand 1 (r_1) register to be divided by the double-word contents of the operand 2 (r_2) register. The normalized quotient is placed in the operand 1 (r_1) register. Any remainder is not preserved.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DDR	r_1,r_2
DIVLONG	DDR	R2,R6

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 or one digit position may be required. The exponent of the intermediate result is increased by 1 if the shift is necessary. All operand 1 (r_1) fraction digits are used in forming the quotient even if the normalized operand 1 (r_1) fraction is larger than the normalized operand 2 (r_2) fraction.

- If the final quotient exponent exceeds 127, an exponent overflow 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 zero, 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 zero, the result is made a true zero. Underflow does not apply to the intermediate result or the operands during normalization.

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

Example:

```
LABEL    ΔOPERATIONΔ                     OPERAND
1        10         16
_____________________________________________________
DIVIDLG  DDR        R2,R6
```

Before execution of the *divide, long format* (DDR) instruction, if we assume values of +1000 in R2 and +5 in R6, the contents of R2 and R6 will be:

R2 before execution:

4	3	3	E	8	0	0	0	0	0	0	0	0	0	0	0	+1000

R6 before and after execution:

4	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	+5

R2 after execution:

4	2	C	8	0	0	0	0	0	0	0	0	0	0	0	0	+200

DE

11.16. DIVIDE, SHORT FORMAT (DE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
DE	7D	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *divide, short format* (DE) instruction causes the full-word contents of the operand 1 (r_1) register to be divided by the full-word contents of a full word in storage specified by operand 2. The normalized quotient is placed in a full word in the operand 1 (r_1) register. Any remainder is not preserved.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DE	$r_1,d_2(x_2,b_2)$
DIVSHORT	DE	R4,32(R8,R9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DE	$r_1,s_2(x_2)$
DIVSHORT	DE	R4,FAM

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 (r_1) fraction digits are used in forming the quotient even if the normalized operand 1 (r_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 zero, 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 zero, the result is made a true zero. Underflow does not apply to the intermediate result or the operands during normalization.

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

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10        16
-----------------------------------------------------------
DIVIDESH DE        R4,FAM
FAM      DC        E'5'
```

Before execution of the *divide, short format* (DE) instruction, if we assume a value of +1000 in R4, the contents of R4 and storage area FAM will be:

R4 before execution:

4	3	3	E	8	0	0	0	+1000

FAM before and after execution:

4	1	5	0	0	0	0	0	+5

R4 after execution:

4	2	C	8	0	0	0	0	+200

DER

11.17. DIVIDE, SHORT FORMAT (DER)

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
DER	3D	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *divide, short format* (DER) instruction causes the full-word contents of the operand 1 (r_1) register to be divided by the full-word contents of the operand 2 (r_2) register. The normalized quotient is placed in a full word in the operand 1 (r_1) register. Any remainder is not preserved.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DER	r_1,r_2
DIVSHORT	DER	R4,R6

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 (r_1) fraction digits are used in forming the quotient even if the normalized operand 1 (r_1) fraction is larger than the normalized operand 2 (r_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 zero, 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 zero, the result is made a true zero. Underflow does not apply to the intermediate result or the operands during normalization.

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

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10        16
--------------------------------------------------------------
DIVIDESH DER        R4,R6
```

Before execution of the *divide, short format* (DER) instruction, if we assume values of +1000 in R4 and +5 in R6, the contents of R4 and R6 will be:

R4 before execution:

4	3	3	E	8	0	0	0	+1000

R6 before and after execution:

4	1	5	0	0	0	0	0	+5

R4 after execution:

4	2	C	8	0	0	0	0	+200

HDR

11.18. HALVE, LONG FORMAT (HDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
HDR	24	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *halve, long format* (HDR) instruction causes the double-word contents of the operand 2 (r_2) register to be divided by 2. The normalized quotient is placed in the double-word operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	HDR	r_1,r_2
HALVE	HDR	R4,R6

Operational Considerations:

- The fraction of operand 2 (r_2) is shifted right one bit position. The least significant bit of the fraction is placed into the most significant bit position of the guard digit, and the vacated fraction bit position is filled with a zero. The intermediate result is normalized and placed in the operand 1 (r_1) location.

- When normalization causes the exponent to become less than zero, an exponent underflow condition exists. If the exponent underflow mask bit of the current program status word (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 zero, the result is made true zero.

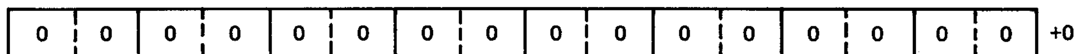
- When the fraction of operand 2 (r2) is zero, the result is made a true zero, a normalization is not attempted, and a significance exception does not occur.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1         10          16

HALF      HDR         R4,R6
```

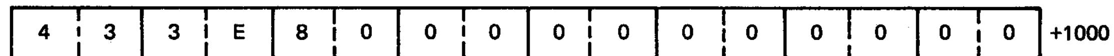
Before execution of the *halve, long format* (HDR) instruction, if we assume values of +0 in R4 and +1000 in R6, the contents of R4 and R6 will be:

R4 before execution:



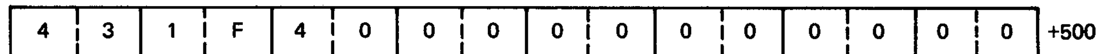
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	+0

R6 before and after execution:



4	3	3	E	8	0	0	0	0	0	0	0	0	0	0	0	+1000

R4 after execution:



4	3	1	F	4	0	0	0	0	0	0	0	0	0	0	0	+500

HER

11.19. HALVE, SHORT FORMAT (HER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
HER	34	RR	2

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *halve, short format* (HER) instruction causes the full-word contents of the operand 2 (r_2) register to be divided by 2. The normalized quotient is placed in the full word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	HER	r_1,r_2
HALVE	HER	R4,R6

Operational Considerations:

- The fraction of operand 2 (r_2) is shifted right one bit position. The least significant bit of the fraction is placed into the most significant bit position of the guard digit, and the vacated fraction bit position is filled with a zero. The intermediate result is normalized and placed in the operand 1 (r_1) location.

- When normalization causes the exponent to become less than zero, an exponent underflow condition exists. If the exponent underflow mask bit of the current program status word (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 zero, the result is made true zero.

- When the fraction of operand 2 (r_2) is zero, the result is made a true zero, normalization is not attempted, and a significance exception does not occur.

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1        10        16
_______________________________________________________________
HALF     HER       R4,R6
```

Before execution of the *halve, short format* (HER) instruction, if we assume values of +1000 in R4 and +0 in R6, the contents of R6 and R4 will be:

R4 before execution:

0	0	0	0	0	0	0	0	+0

R6 before and after execution:

4	3	3	E	8	0	0	0	+1000

R4 after execution:

4	3	1	F	4	0	0	0	+500

LCDR

11.20. LOAD COMPLEMENT, LONG FORMAT (LCDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LCDR	23	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *load complement, long format* (LCDR) instruction causes the sign of the double-word contents of the operand 2 (r_2) register to be reversed. The result is placed in the double word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LCDR	r_1,r_2
SIGN	LCDR	R6,R4

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 (r_2) remain unchanged.
- The condition code is set:
  - — to zero if result is zero;
  - — to 1 if result is less than zero; and
  - — to 2 if result is greater than zero.

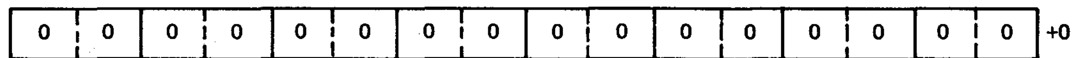
  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10       16
-----------------------------------------------------------
SIGN      LCDR     R6,R4
```

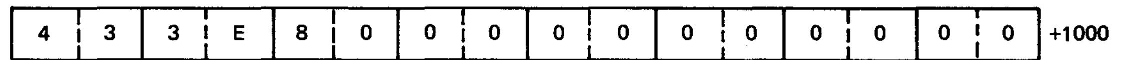
Before execution of the *load complement, long format* (LCDR) instruction, if we assume values of +1000 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:



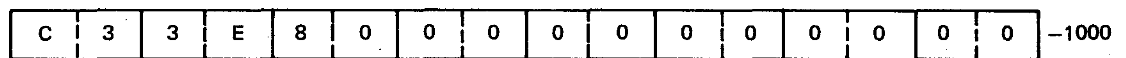
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	+0

R4 before and after execution:



4	3	3	E	8	0	0	0	0	0	0	0	0	0	0	0	+1000

R6 after execution:



C	3	3	E	8	0	0	0	0	0	0	0	0	0	0	0	–1000

LCER

11.21. LOAD COMPLEMENT, SHORT FORMAT (LCER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LCER	33	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *load complement, short format* (LCER) instruction causes the sign of the full-word contents of the operand 2 (r_2) register to be reversed. The result is placed in the full word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LCER	r_1,r_2
SIGN	LCER	R6,R4

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 (r_2) remain unchanged.
- The condition code is set:
  - to zero if result is zero;
  - to 1 if result is less than 0; and
  - to 2 if result is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10          16
-------------------------------------------------------------
SIGN     LCER        R6,R4
```

Before execution of the *load complement, short format* (LCER) instruction, if we assume values of +1000 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:

0	0	0	0	0	0	0	0	+0

R4 before and after execution:

4	3	3	E	8	0	0	0	+1000

R6 after execution:

C	3	3	E	8	0	0	0	–1000

LD

11.22. LOAD, LONG FORMAT (LD)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LD	68	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *load, long format* (LD) instruction causes the contents of a double word in storage specified by operand 2 to be placed in the double word in the operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LD	$r_1,d_2(x_2,b_2)$
LOAD	LD	R4,33(R8,R9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LD	$r_1,s_2(x_2)$
LOAD	LD	R4,FAM

Operational Consideration:

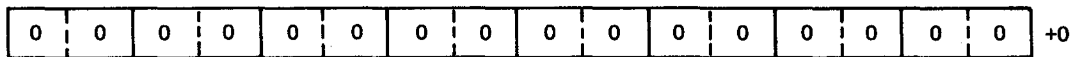
- The contents of operand 2 remain unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10        16
_______________________________________________
LOAD     LD        R4,FAM
FAM      DC        D'15Ø'
```

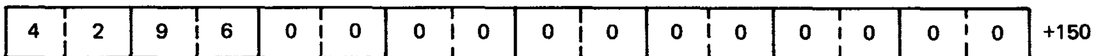
Before execution of the *load, long format* (LD) instruction, if we assume a value of +0 in R4, the contents of R4 and main storage area FAM will be:

R4 before execution:



FAM before and after execution:

R4 after execution:



LDR

11.23. LOAD, LONG FORMAT (LDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LDR	28	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *load, long format* (LDR) instruction causes the contents of the double word in the operand 2 (r_2) register to be placed in the double word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	LDR	r_1,r_2
LOAD	LDR	R6,R4

Operational Consideration:

- The contents of operand 2 (r_2) remain unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10          16
LOAD      LDR         R6,R4
```

Before execution of the *load, long format* (LDR) instruction, if we assume values of +150 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	+0

R4 before and after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

R6 after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

LE

11.24. LOAD, SHORT FORMAT (LE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LE	78	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *load, short format* (LE) instruction causes the contents of a full word in storage specified by operand 2 to be placed in a full word in the operand 1 (r_1) register.

Expicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LE	$r_1,d_2(x_2,b_2)$
LOAD	LE	R6,33(R8,R9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LE	$r_1,s_2(x_2)$
LOAD	LE	R6,FAM

Operational Consideration:

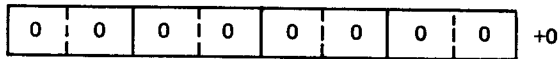
- The contents of operand 2 remain unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10        16
---------------------------------------------------------
LOAD      LE        R6,FAM
FAM       DC        E'15Ø'
```

Before execution of the *load, short format* (LE) instruction, if we assume a value of +0 in R6, the contents of R6 and main storage area FAM will be:

R6 before execution:



0	0	0	0	0	0	0	0

+0

FAM before and after execution:

4	2	9	6	0	0	0	0

+150

R6 after execution:

4	2	9	6	0	0	0	0

+150

LER

11.25. LOAD, SHORT FORMAT (LER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LER	38	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *load, short format* (LER) instruction causes the contents of a full word in the operand 2 (r_2) register to be placed in a full word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LER	r_1,r_2
LOAD	LER	R6,R4

Operational Consideration:

- The contents of operand 2 (r_2) remain unchanged.

Example:

```
LABEL    ΔOPERATIONΔ                   OPERAND
1          10          16
-----------------------------------------------
LOAD       LER          R6,R4
```

Before execution of the *load, short format* (LER) instruction, if we assume values of +150 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:

0	0	0	0	0	0	0	0	+0

R4 before and after execution:

4	2	9	6	0	0	0	0	+150

R6 after execution:

4	2	9	6	0	0	0	0	+150

LNDR

11.26. LOAD NEGATIVE, LONG FORMAT (LNDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LNDR	**21**	**RR**	**2**

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *load negative, long format* (LNDR) instruction causes the sign of the double word in the operand 2 (r_2) register to be made negative. The result is placed in the double-word register specified by operand 1 (r_1).

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LNDR	r_1,r_2
LOAD	LNDR	R2,R6

Operational Considerations:

- Operand 2 (r_2) is made negative even if the fraction is zero.
- The exponent and fraction are not changed.
- The contents of operand 2 (r_2) remain unchanged.
- The condition code is set:
  - — to zero if result is zero; and
  - — to 1 if result is less than zero.

  Codes 2 and 3 are not used.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10       16
_________________________________________________________
LOAD      LNDR     R2,R6
```

Before execution of the *load negative, long format* (LNDR) instruction, if we assume values of +150 in R6 and +0 in R2, the contents of R2 and R6 will be:

R2 before execution:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	+0

R6 before and after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

R2 after execution:

C	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	−150

LNER

11.27. LOAD NEGATIVE, SHORT FORMAT (LNER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LNER	31	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *load negative, short format* (LNER) instruction causes the sign of a full word in the operand 2 (r_2) register to be made negative. The result is placed in a full word in the register specified by operand 1 (r_1).

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LNER	r_1,r_2
LOAD	LNER	R6,R4

Operational Considerations:

- Operand 2 (r_2) is made negative even if the fraction is zero.
- The exponent and fraction are not changed.
- The contents of operand 2 (r_2) remain unchanged.
- The condition code is set:
  - to zero if result is zero; and
  - to 1 if result is less than zero.

  Codes 2 and 3 are not used.

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1         10        16
------------------------------------------------------------
LOADNEG  LNER      R6,R4
```

Before execution of the *load negative, short format* (LNER) instruction, if we assume values of +150 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:

0	0	0	0	0	0	0	0	+0

R4 before and after execution:

4	2	9	6	0	0	0	0	+150

R6 after execution:

C	2	9	6	0	0	0	0	–150

LPDR

11.28. LOAD POSITIVE, LONG FORMAT (LPDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LPDR	20	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- □ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *load positive, long format* (LPDR) instruction causes the sign of the double word in the operand 2 (r_2) register to be positive, and the result is placed in the double word of the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	LPDR	r_1,r_2
LOADN	LPDR	R4,R6

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 (r_2) remain unchanged.
- The condition code is set:
  - — to zero if result is zero; and
  - — to 2 if result is greater than zero.

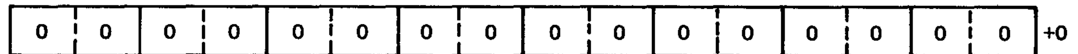
  Codes 1 and 3 are not used.

Example:

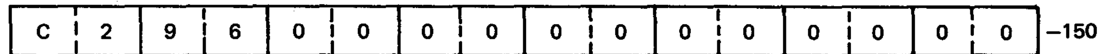
```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10         16
-------------------------------------------------------
LOADPOS  LPDR        R6,R4
```

Before execution of the *load positive, long format* (LPDR) instruction, if we assume value of —150 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:



R4 before and after execution:



R6 after execution:

LPER

11.29. LOAD POSITIVE, SHORT FORMAT (LPER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LPER	30	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *load positive, short format* (LPER) instruction causes the sign of a full word in the operand 2 (r_2) register to be positive. The result is placed in a full word of the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LPER	r_1,r_2
LOADP	LPER	R6,R4

Operational Considerations:

- The exponent and fraction are not changed.
- The contents of operand 2 (r_2) remain unchanged.
- The condition code is set:
  - to zero if result is zero; and
  - to 2 if result is greater than zero.

  Codes 1 and 3 are not used.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10        16
---------------------------------------------------------------
LOADPOS  LPER      R6,R4
```

Before execution of the *load positive, short format* (LPER) instruction, if we assume values of —150 in R4 and +0 in R6, the contents of R6 and R4 will be:

R6 before execution:

0	0	0	0	0	0	0	0	+0

R4 before and after execution:

C	2	9	6	0	0	0	0	–150

R6 after execution:

4	2	9	6	0	0	0	0	+150

LTDR

11.30. LOAD AND TEST, LONG FORMAT (LTDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LTDR	22	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
■ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *load and test, long format* (LTDR) instruction causes the double-word contents of the operand 2 (r_2) register to be placed in the double-word operand 1 (r_1) register. The condition code is set by this instruction.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LTDR	r_1,r_2
TEST	LTDR	R2,R6

Operational Considerations:

- The contents of operand 2 (r_2) remain unchanged.
- When the same register is specified by operand 1 (r_1) and operand 2 (r_2), the operation is equivalent to a test without data movement.
- The condition code is set:
  - to zero if result is zero;
  - to 1 if result is less than zero; and
  - to 2 if result is greater than zero.

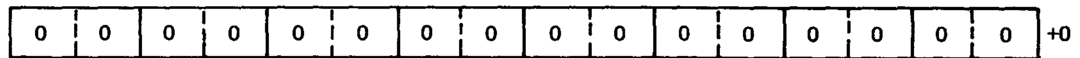
  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10        16
_______________________________________________
TEST      LTDR      R2,R6
```

Before execution of the *load and test, long format* (LTDR) instruction, if we assume values of +150 in R6 and +0 in R2, the contents of R6 and R2 will be:

R2 before execution:



R6 before and after execution:

R2 after execution:

The condition code is set to 2 because the result is greater than zero.

LTER

11.31. LOAD AND TEST, SHORT FORMAT (LTER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LTER	32	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
□ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *load and test, short format* (LTER) instruction causes the contents of a full word in the operand 2 (r_2) register to be placed in a full word in the operand 1 (r_1) register. The condition code is set by this instruction.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LTER	r_1,r_2
TEST	LTER	R6,R4

Operational Considerations:

- The contents of operand 2 (r_2) remain unchanged.

- When the same register is specified by operand 1 (r_1) and operand 2 (r_2), the operation is equivalent to a test without data movement.

- The condition code is set:

  - to zero if result is zero;

  - to 1 if result is less than zero; and

  - to 2 if result is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10        16
-------------------------------------------------------
TEST      LTER      R6,R4
```

Before execution of the *load and test, short format* (LTER) instruction, if we assume values of +150 in R4 and +0 in R6, the contents of R4 and R6 will be:

R6 before execution:

0	0	0	0	0	0	0	0	+0

R4 before and after execution:

4	2	9	6	0	0	0	0	+150

R6 after execution:

4	2	9	6	0	0	0	0	+150

The condition code is set to 2 because the result is greater than zero.

MD

11.32. MULTIPLY, LONG FORMAT (MD)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MD	6C	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *multiply, long format* (MD) instruction causes the contents of the double word in the operand 1 (r_1) register to be multipled by the contents of a double word in main storage specified by operand 2. The normalized product is placed in the double word of the operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MD	$r_1,d_2(x_2,b_2)$
MULT	MD	R4,32(R9,R10)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MD	$r_1,s_2(x_2)$
MULT	MD	R4,FAM

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.

- If the final product exponent is less than zero, an exponent underflow condition exists. If the exponent underflow mask bit (38) 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 zero, the result is made a true zero. When an underflow characteristic becomes less than zero during normalization before multiplication, an underflow exception is not recognized.

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

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

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10        16
------------------------------------------------------------
MULTLG   MD        R4,FAM
FAM      DC        D'5Ø'
```

Before execution of the *multiply, long format* (MD) instruction, if we assume a value of —100 in R4, the contents of R4 and main storage area FAM will be:

R4 before execution:

C	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	–100

FAM before and after execution:

4	2	3	2	0	0	0	0	0	0	0	0	0	0	0	0	+50

R4 after execution:

C	4	1	3	8	8	0	0	0	0	0	0	0	0	0	0	–5000

MDR

11.33. MULTIPLY, LONG FORMAT (MDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MDR	2C	RR	2

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *multiply, long format* (MDR) instruction causes the contents of the double word in the operand 1 (r_1) register to be multiplied by the contents of the double word in the operand 2 (r_2) register. The normalized product is placed in the double word of the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MDR	r_1,r_2
MULT	MDR	R4,R6

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 of 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.

- If the final product exponent is less than zero, an exponent underflow condition exists. If the exponent underflow mask bit (38) 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 zero, the result is made a true zero. When an underflow characteristic becomes less than zero during normalization before multiplication, an underflow exception is not recognized.

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

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

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10        16
---------------------------------------------------------------
MULTREG   MDR       R4,R6
```

Before execution of the *multiply, long format* (MDR) instruction if we assume values of —100 in R4 and +50 in R6, the contents of R4 and R6 will be:

R4 before execution:

C	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	—100

R6 before and after execution:

4	2	3	2	0	0	0	0	0	0	0	0	0	0	0	0	+50

R4 after execution:

C	4	1	3	8	8	0	0	0	0	0	0	0	0	0	0	—5000

ME

11.34. MULTIPLY, SHORT FORMAT (ME)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ME	7C	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *multiply, short format* (ME) instruction causes the contents of a full word in the operand 1 (r_1) register to be multiplied by the contents of a full word in main storage specified by operand 2. The normalized product is placed in a full word of the operand 1 (r_1) register.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	ME	$r_1,d_2(x_2,b_2)$
MULT	ME	R6,32(R8,R12)

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	ME	$r_1,s_2(x_2)$
MULT	ME	R6,FAM

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 zero, 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.
- If the final product exponent is less than zero, an exponent underflow condition exists. If the exponent underflow mask bit (38) 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 zero, the result is made a true zero. When an underflow characteristic becomes less than zero during normalization before multiplication, an underflow exception is not recognized.
- When all digits of the intermediate product are zero, the result is made a true zero.
- When the resulting fraction is zero, a program exception exponent underflow or overflow does not occur.

Example:

```
LABEL   ΔOPERATIONΔ                         OPERAND
1       10         16
-----------------------------------------------------------
MULT    ME         R6,FAM
FAM     DC         E'5Ø'
```

Before execution of the *multiply, short format* (ME) instruction, if we assume a value of —100 in R6, the contents of R6 and main storage area FAM will be:

R6 before execution:

C	2	6	4	0	0	0	0	—100

FAM before and after execution:

4	2	3	2	0	0	0	0	+50

R6 after execution:

C	4	1	3	8	8	0	0	—5000

MER

11.35. MULTIPLY, SHORT FORMAT (MER)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MER	**3C**	**RR**	**2**

Condition Codes

☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

Possible Program Exceptions

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

The *multiply, short format* (MER) instruction causes the contents of a full word in the operand 1 (r_1) register to be multiplied by the contents of a full word in the operand 2 (r_2) register. The normalized product is placed in a full word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	MER	r_1,r_2
MULT	MER	R6,R4

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 zero, 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.

- If the final product exponent is less than zero, an exponent underflow condition exists. If the exponent underflow mask bit (38) 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 zero, the result is made a true zero. When an underflow characteristic becomes less than zero during normalization before multiplication, an underflow exception is not recognized.
- When all digits of the intermediate product are zero, the result is made a true zero.
- When the resulting fraction is zero, a program exception for exponent underflow or overflow does not occur.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1         10          16
-------------------------------------------------------------
MULT      MER         R6,R4
```

Before execution of the *multiply, short format* (MER) instruction, if we assume values of —100 in R6 and +50 in R4, the contents of R6 and R4 will be:

R6 before execution:

C	2	6	4	0	0	0	0	–100

R4 before and after execution:

4	2	3	2	0	0	0	0	+50

R6 after execution:

C	4	1	3	8	8	0	0	–500

SD

11.36. SUBTRACT NORMALIZED, LONG FORMAT (SD)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SD	6B	RX	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *subtract normalized, long format* (SD) instruction causes the contents of a double word in main storage, specified by operand 2, to be algebraically subtracted from the contents of the double-word register specified by operand 1 (r_1). The normalized difference is placed in the operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SD	$r_1,d_2(x_2,b_2)$
SUB	SD	R4,32(R7,R8)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SD	$r_1,s_2(x_2)$
SUB	SD	R4,FAM

Operational Considerations:

- The execution of the SD instruction is identical to that of the AD instruction (11.2), except that the sign of operand 2 is reversed before addition.
- The condition code is set:
  - to zero if result fraction is zero;
  - to 1 if result fraction is less than zero; and
  - to 2 if result fraction is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                       OPERAND
1        10         16
--------------------------------------------------------
SUBLONG  SD         R4,FAM
FAM      DC         D'1ØØ'
```

Before execution of the *subtract normalized, long format* (SD) instruction, if we assume a value of +250 in R4, the contents of R4 and main storage area FAM will be:

R4 before execution:

4	2	F	A	0	0	0	0	0	0	0	0	0	0	0	0	+250

FAM before and after execution:

4	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	+100

R4 after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

SDR

11.37. SUBTRACT NORMALIZED, LONG FORMAT (SDR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SDR	2B	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *subtract normalized, long format* (SDR) instruction causes the contents of the double-word register, specified by operand 2 (r_2) to be algebraically subtracted from the contents of the double-word register, specified by operand 1 (r_1). The normalized difference is placed in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SDR	r_1,r_2
SUBTR	SDR	R6,R4

Operational Considerations:

- The execution of the SDR instruction is identical to that of the ADR instruction (11.3), except that the sign of operand 2 (r_2) is reversed before addition.
- The condition code is set:
  - — to zero if result fraction is zero;
  - — to 1 if result fraction is less than zero; and
  - — to 2 if result fraction is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10       16
--------------------------------------------------------
SUBTR    SDR      R6,R4
```

Before execution of the *subtract normalized, long format* (SDR) instruction, if we assume values of +250 in R6 and +100 in R4, the contents of R6 and R4 will be:

R6 before execution:

4	2	F	A	0	0	0	0	0	0	0	0	0	0	0	0	+250

R4 before and after execution:

4	2	6	4	0	0	0	0	0	0	0	0	0	0	0	0	+100

R6 after execution:

4	2	9	6	0	0	0	0	0	0	0	0	0	0	0	0	+150

SE

11.38. SUBTRACT NORMALIZED, SHORT FORMAT (SE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SE	**7B**	**RX**	**4**

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *subtract normalized, short format* (SE) instruction causes the contents of a full word in main storage, specified by operand 2, to be algebraically subtracted from a full word in the register specified by operand 1 (r_1). The normalized difference is placed in the operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SE	$r_1,d_2(x_2,b_2)$
SUB	SE	R2,32(R9,R10)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SE	$r_1,s_2(x_2)$
SUB	SE	R2,FAM

Operational Considerations:

- The execution of the SE instruction is identical to that of the AE instruction (11.4), except that the sign of operand 2 is reversed before addition.
- The condition code is set:
  - — to zero if result fraction is zero;
  - — to 1 if result fraction is less than zero; and
  - — to 2 if result fraction is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                            OPERAND
1        10         16
-----------------------------------------------------------
SUBSHORT SE         R2,FAM
FAM      DC         E'1ØØ'
```

Before execution of the *subtract normalized, short format* (SE) instruction, if we assume a value of +250 in R2, the contents of R2 and main storage area FAM will be:

R2 before execution:

4	2	F	A	0	0	0	0	+250

FAM before and after execution:

4	2	6	4	0	0	0	0	+100

R2 after execution:

4	2	9	6	0	0	0	0	+150

SER

11.39. SUBTRACT NORMALIZED, SHORT FORMAT (SER)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SER	3B	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *subtract normalized, short format* (SER) instruction causes the contents of a full word in the operand 2 (r_2) register to be algebraically subtracted from a full word in the operand 1 (r_1). The normalized difference is placed in a full word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SER	r_1,r_2
SUB	SER	R2,R4

Operational Considerations:

- The execution of the SER instruction is identical to that of the AER instruction (11.5), except that the sign of operand 2 is reversed before addition.
- The condition code is set:
  - — to zero if result fraction is zero;
  - — to 1 if result fraction is less than zero; and
  - — to 2 if result fraction is greater than zero.

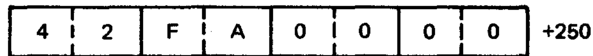
  Code 3 is not used.

Example:

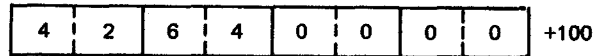
```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10          16
_______________________________________________________________
SUB      SER         R2,R4
```

Before execution of the *subtract normalized, short format* (SER) instruction, if we assume values of +250 in R2 and +100 in R4, the contents of R2 and R4 will be:

R2 before execution:



R4 before and after execution:



R2 after execution:

STD

11.40. STORE, LONG FORMAT (STD)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STD	60	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *store, long format* (STD) instruction causes the contents of the register, specified by operand 1 (r_1), to be placed in a double word in main storage, specified by operand 2.

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	STD	$r_1,d_2(x_2,b_2)$
STORE	STD	R4,32(R5,R6)

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	STD	$r_1,s_2(x_2)$
STORE	STD	R4,FAM

Operational Consideration:

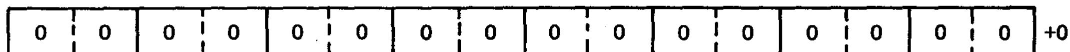
- The contents of the operand 1 (r_1) register remain unchanged.

Example:

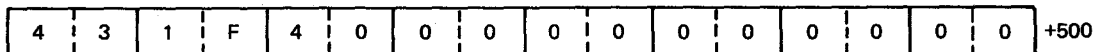
```
LABEL    ΔOPERATIONΔ                            OPERAND
1        10       16
______________________________________________________
STORELG  STD      R4,FAM
FAM      DC       D'Ø'
```

Before execution of the *store, long format* (STD) instruction, if we assume a value of +500 in R4, the contents of R4 and main storage area FAM will be:

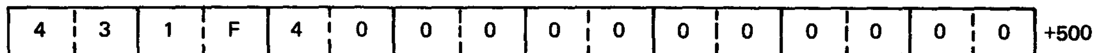
FAM before execution:



R4 before and after execution:



FAM after execution:



STE

11.41. STORE, SHORT FORMAT (STE)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STE	70	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *store, short format* (STE) instruction causes the contents of a full word in the register, specified by operand 1 (r_1), to be placed in a full word in main storage, specified by operand 2.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STE	$r_1,d_2(x_2,b_2)$
STORE	STE	R4,32(R5,R6)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STE	$r_1,s_2(x_2)$
STORE	STE	R4,FAM

Operational Consideration:

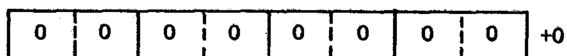
- The contents of the operand 1 (r_1) register remain unchanged.

Example:

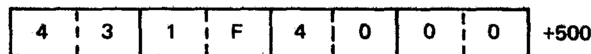
```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10        16
_________________________________________________________
STORE    STE       R4,FAM
FAM      DC        E'Ø'
```

Before execution of the *store, short format* (STE) instruction, if we assume a value of +500 in R4, the contents of R4 and main storage FAM will be:

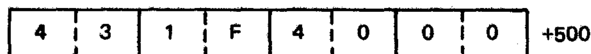
FAM before execution:



R4 before and after execution:



FAM after execution:



SU

11.42. SUBTRACT UNNORMALIZED, SHORT FORMAT (SU)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SU	7F	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT <0, SET TO 1
■ IF RESULT >0, SET TO 2
□ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *subtract unnormalized, short format* (SU) instruction causes the contents of a full word in main storage specified by operand 2 to be algebraically subtracted from the contents of a full word in the register specified by operand 1 (r_1). The difference is placed in a full word in the operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SU	$r_1,d_2(x_2,b_2)$
SUB	SU	R6,32(R7,R9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SU	$r_1,s_2(x_2)$
SUB	SU	R6,FAM

Operational Considerations:

- The execution of the SU instruction is identical to that of the AU instruction (11.6), except that the sign is reversed before addition.

- The condition code is set:

  — to zero if result fraction is zero;

  — to 1 if result fraction is less than zero; and

  — to 2 if result fraction is greater than zero.

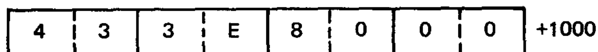
  Code 3 is not used.

Example:

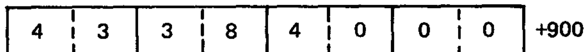
```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10         16
________________________________________________________
SUBUNNOR SU         R6,FAM
FAM      DC         E'9ØØ'
```

Before execution of the *subtract unnormalized, short format* (SU) instruction, if we assume a value of +1000 in R6, the contents of R6 and main storage area FAM will be:

R6 before execution:



FAM before and after execution:



R6 after execution:

SUR

11.43. SUBTRACT UNNORMALIZED, SHORT FORMAT (SUR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SUR	3F	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *subtract unnormalized, short format* (SUR) instruction causes the contents of a full word in the operand 2 (r_2) register to be algebraically subtracted from a full word in the operand 1 (r_1) register. The difference is placed in a full word in the operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SUR	r_1,r_2
SUB	SUR	R6,R4

Operational Considerations:

- The execution of the SUR instruction is identical to that of the AUR instruction (11.7), except that the sign is reversed before addition.
- The condition code is set:
  - — to zero if result fraction is zero;
  - — to 1 if result fraction is less than zero; and
  - — to 2 if result fraction is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10          16
_____________________________________________________
SUBSHORT SUR         R6,R4
```

Before execution of the *subtract unnormalized, short format* (SUR) instruction, if we assume values of +1000 in R6 and +900 in R4, the contents of R6 and R4 will be:

R6 before execution:

4	3	3	E	8	0	0	0	+1000

R4 before and after execution:

4	3	3	8	4	0	0	0	+900

R6 after execution:

4	3	0	6	4	0	0	0	+100

SW

11.44. SUBTRACT UNNORMALIZED, LONG FORMAT (SW)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SW	**6F**	**RX**	**4**

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT < 0, SET TO 1
- ■ IF RESULT > 0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *subtract unnormalized, long format* (SW) instruction causes the contents of a double word in main storage specified by operand 2 to be algebraically subtracted from the contents of the double word in the register specified by operand 1 (r_1). The difference is placed in the double-word operand 1 (r_1) register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SW	$r_1,d_2(x_2,b_2)$
SUB	SW	R4,32(R5,R9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SW	$r_1,s_2(x_2)$
SUB	SW	R4,FAM

Operational Considerations:

- The execution of the SW instruction is identical to that of the AW instruction (11.8), except that the sign is reversed before addition.
- The condition code is set:
  - to zero if result fraction is zero;
  - to 1 if result fraction is less than zero; and
  - to 2 if result fraction is greater than zero.

  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10        16
-------------------------------------------------------------
SUBUNNOR SW         R4,FAM
FAM      DC         D'9ØØ'
```

Before execution of the *subtract unnormalized, long format* (SW) instruction, if we assume a value of +1000 in R4, the contents of R4 and main storage area FAM will be:

R4 before execution:

4	3	3	E	8	0	0	0	0	0	0	0	0	0	0	0	+1000

FAM before and after execution:

4	3	3	8	4	0	0	0	0	0	0	0	0	0	0	0	+900

R4 after execution:

4	3	0	6	4	0	0	0	0	0	0	0	0	0	0	0	+100

SWR

11.45. SUBTRACT UNNORMALIZED, LONG FORMAT (SWR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SWR	2F	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ■ IF RESULT >0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *subtract unnormalized, long format* (SWR) instruction causes the contents of the double word in the operand 2 (r_2) register to be algebraically subtracted from the double-word contents of the operand 1 (r_1) register. The difference is placed in the double-word operand 1 (r_1) register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SWR	r_1,r_2
SUB	SWR	R4,R6

Operational Considerations:

- The execution of the SWR instruction is identical to that of the AWR instruction (11.9), except that the sign is reversed before addition.
- The condition code is set:
  - — to zero if result fraction is zero;
  - — to 1 if result fraction is less than zero; and
  - — to 2 if result fraction is greater than zero.

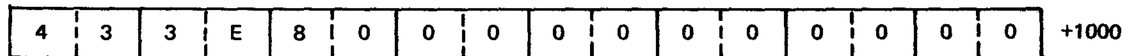
  Code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10        16
_____________________________________________________
SUBLONG  SWR       R4,R6
```

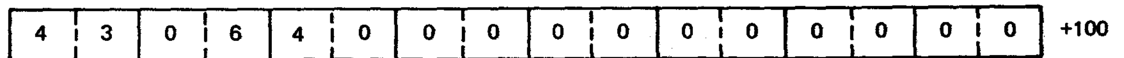
Before execution of the *subtract unnormalized, long format* (SWR) instruction, if we assume values of +1000 in R4 and +900 in R6, the contents of R4 and R6 will be:

R4 before execution:



R6 before and after execution:

R4 after execution:



12. Logical Instructions

12.1. THE USE OF LOGICAL INSTRUCTIONS

All operations performed by logical instructions are executed according to the rules of logic. Unlike decimal and fixed-point binary instructions, logical instructions disregard arithmetic signs. Most of these instructions manipulate data bit by bit and operate from left to right.

12.2. ADD LOGICAL (AL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
AL	5E	RX	4

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
■ SET TO 3
SEE OPER. CONSIDERATIONS

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

The *add logical* (AL) instruction logically adds the content of operand 2 to the content of the operand 1 register and places the sum in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AL	$r_1, d_2(x_2, b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	AL	$r_1, s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- The main storage location you specify in operand 2 must refer to a main storage area that is on a full-word boundary. Operand 2 is not changed by the execution of this instruction.
- The addition is performed by logically adding the 32 bits of operand 2 to the 32 bits of operand 1.
- Neither operand has a sign bit.
- The condition code of the program status word (PSW) is set as follows:
  - to 0 if result is 0 (no carry of most significant bit);
  - to 1 if result is not 0 (no carry of most significant bit);
  - to 2 if result is 0 (carry of most significant bit); or
  - to 3 if result is not 0 (carry of most significant bit).

Example:

```
LABEL     ΔOPERATIONΔ                    OPERAND
1          10       16
--------------------------------------------------------
          L         3,HEXVALU
          AL        3,FULLWORD
          .
          .
          .
          DS        ØF
HEXVALU   DC        X'ØØØØØØ19'
FULLWORD  DC        X'ØØØØØ79C'
```

In this example, the main storage content of HEXVALU is placed into register 3. The AL instruction logically adds the full-word content of main storage location FULLWORD to the content of register 3 and places the sum in register 3.

Register 3 before execution of AL instruction:

0000	0000	0000	0000	0000	0000	0001	1001	binary
0	0	0	0	0	0	1	9	hex

FULLWORD before execution of AL instruction:

0000	0000	0000	0000	0000	0111	1001	1100	binary
0	0	0	0	0	7	9	C	hex

Register 3 after execution of AL instruction:

0000	0000	0000	0000	0000	0111	1011	0101	binary
0	0	0	0	0	7	B	5	hex

12.3. ADD LOGICAL (ALR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ALR	1E	RR	2

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
■ SET TO 3
SEE OPER. CONSIDERATIONS

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

The *add logical* (ALR) instruction logically adds the content of the operand 1 register to the content of the operand 2 register and places the sum in operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ALR	r_1, r_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- The addition is performed by logically adding the 32 bits of operand 2 to operand 1.
- Neither operand has a sign bit.
- The condition code of the program status word (PSW) is set as follows:
  - to 0 if result is 0 (no carry of most significant bit);
  - to 1 if result is not 0 (no carry of most significant bit);
  - to 2 if result is 0 (carry of most significant bit); or
  - to 3 if result is not 0 (carry of most significant bit).

Example:

```
LABEL     ΔOPERATIONΔ                          OPERAND
1          10          16
-----------------------------------------------------------
           L           3,VALU1
           L           4,VALU2
           ALR         3,4
           .
           .
           .
           DS          ØF
VALU1      DC          X'ØØØØØØ19'
VALU2      DC          X'ØØØØØ79C'
```

In this example, the hexadecimal contents of main storage locations VALU1 and VALU2 are placed in registers 3 and 4, respectively. Then, the contents of registers 3 and 4 are added and the sum placed in register 3.

Register 3 before execution of ALR instruction:

0000	0000	0000	0000	0000	0000	0001	1001	binary
0	0	0	0	0	0	1	9	hex

Register 4 before execution of ALR instruction:

0000	0000	0000	0000	0000	0111	1001	1100	binary
0	0	0	0	0	7	9	C	hex

Register 3 after execution of ALR instruction:

0000	0000	0000	0000	0000	0111	1011	0101	binary
0	0	0	0	0	7	B	5	hex

12.4. AND (N)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
N	54	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠ 0, SET TO 1
□ IF RESULT > 0, SET TO 2
□ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *and* (N) instruction performs a logical AND operation on the contents of the operand 1 register and the contents of the full word in operand 2. The result is placed in the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	N	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	N	$r_1,s_2(x_2)$

When the N instruction is executed, a logical AND operation is performed on a bit in operand 1 and a bit in operand 2. The result of the AND operation replaces the bit just accessed in operand 1. This instruction operates from left to right starting with the logical AND operations of bit 0 in both operands up to and including the logical AND operation of bit 31 in both operands.

The N instruction is used to turn off selected bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

When coding patterns used as operands in AND instructions, code a 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to off (0), and a 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.

After the N instruction is executed, the condition code is set to 0 if the result is all 0's; or the condition code is set to 1 if the result is all 1's or a combination of 1's and 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be defined as either a full word or aligned on a full-word boundary.
- The logical AND operation executes upon all 32 bit positions of operands 1 and 2.
- A zero in a bit position in operand 2 sets its corresponding bit position in operand 1 to 0.
- A 1 in a bit position in operand 2 allows its corresponding bit position in operand 1 to remain the same.
- The condition code is set accordingly.

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1           10         16
----------------------------------------------------------------
            L          8,HEXVALUE
            N          8,ANDPATRN
            .
            .
            .
            DS         ØF
ANDPATRN    DC         X'ØØØØØØFF'
HEXVALUE    DC         X'ØØFØØØ7D'
```

Register 8 before execution of N instruction:

0000	0000	1111	0000	0000	0000	0111	1101	binary
0	0	F	0	0	0	7	D	hex

ANDPATRN before and after execution of N instruction:

0000	0000	0000	0000	0000	0000	1111	1111	binary
0	0	0	0	0	0	F	F	hex

Register 8 after execution of N instruction:

0000	0000	0000	0000	0000	0000	0111	1101	binary
0	0	0	0	0	0	7	D	hex

only bits actually changed

In this example, the hexadecimal value in HEXVALUE is loaded into register 8. Then a logical AND operation is performed on the hexadecimal pattern in ANDPATRN (operand 2) and the contents of register 8. Ones in bit positions 24 through 31 of ANDPATRN allow the corresponding bit positions in register 8 to remain the same. Zeros in bit positions 0 through 23 of ANDPATRN set the corresponding bit positions in register 8 to 0. As the high order four bit positions of byte 2 in register 8 are all 1's, they are set to 0; and as the remaining bit positions are already 0, they remain 0. The condition code is set to 1 because the result is a combination of 1's and 0's.

NC

12.5. AND (NC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
NC	D4	SS	6

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The and (NC) instruction performs a logical AND operation on the contents of operand 1 and the contents of operand 2 which are both located in main storage. The result is placed in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	NC	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	NC	$s_1(l),s_2$

When the NC instruction is executed, a logical AND operation is performed on a bit in operand 1 and a bit in operand 2. The result of the AND operation replaces the bit accessed in operand 1. This instruction operates from left to right. The length of operand 1, whether implied or explicit, determines the length of operand 2. Therefore, when the NC instruction is executed, the lengths of operands 1 and 2 are the same.

The NC instruction is used to turn off selected bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

When coding patterns used as operands in AND instructions, code a 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to off (0), and a 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.

After the NC instruction is executed, the condition code is set to 0 if the result is all 0's; or the condition code is set to 1 if the result is all 1's or a combination of 1's and 0's.

Operational Considerations:

- Operands 1 and 2 must be main storage locations.
- The length of operand 1, whether implied or explicit, determines the length of operand 2.
- A zero in a bit position in operand 2 sets its corresponding bit position in operand 1 to 0.
- A 1 in a bit position in operand 2 allows its corresponding bit position in operand 1 to remain the same.
- The condition code is set accordingly.
- Operands 1 and 2 can have overlapping bytes.

Example:

```
LABEL   ΔOPERATIONΔ                    OPERAND
1        10       16
-------------------------------------------------------
         NC       LOCAT1,LOCAT2
         .
         .
         .
LOCAT1   DC       PL2'-217'
LOCAT2   DC       X'FFFC'
```

LOCAT1 before execution of NC instruction:

0010	0001	0111	1101	binary
2	1	7	D	packed decimal

LOCAT2 before and after execution of NC instruction:

1111	1111	1111	1100	binary
F	F	F	C	hex

LOCAT1 after execution of NC instruction:

0010	0001	0111	1100	binary
2	1	7	C	hex

In this example, LOCAT1 is defined as a negative packed decimal number and LOCAT2 is defined as a field containing a hexadecimal value. A logical AND operation is performed on the contents of LOCAT1 and LOCAT2. The result is placed in LOCAT1. The 1's in bit positions 0 through 13 of LOCAT2 allow the corresponding bit positions in LOCAT1 to remain the same. Zeros in bit positions 14 and 15 of LOCAT2 set the corresponding bit positions in LOCAT1 to 0. Because the low order bit position of LOCAT1 is 1, it is set to 0; and the bit position adjacent to the low order bit position remains 0, since it is already 0. The condition code is set to 1 because the result is a combination of 1's and 0's. Note the sign value is changed from negative to positive.

12.6. AND (NI)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
NI	94	SI	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠ 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *and* (NI) instruction performs a logical AND operation on the contents of operand 1 located in main storage and the one byte of immediate data in operand 2. The result is placed in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	NI	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	NI	s_1,i_2

When the NI instruction is executed, a logical AND operation is performed on a bit in operand 1 and a bit in operand 2. The result of the AND operation replaces the bit just accessed in operand 1. This instruction operates from left to right. The length of operand 1 can vary but the length of operand 2 is always one byte. Although operands 1 and 2 may have differing lengths, only one byte in operand 1 is used in conjunction with the one byte of immediate data in operand 2. The result replaces the one byte in operand 1 that was just accessed. If you do not specify the exact byte in operand 1 you want used in the execution with the one byte of data in operand 2, the first byte of operand 1 is used.

The NI instruction is used to turn off selected bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

When coding patterns used as operands in AND instructions, code a 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to off (0), and a 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.

After the NI instruction is executed, the condition code is set to 0 if the result is all 0's, or the condition code is set to 1 if the result is all 1's or a combination of 1's and 0's.

Operational Considerations:

- Operand 1 must be a main storage location.
- Operand 2 must be a 1-byte, self-defining term.
- The length of operand 1 can vary.
- A zero in a bit position in operand 2 sets its corresponding bit position in operand 1 to 0.
- A 1 in a bit position in operand 2 allows its corresponding bit position in operand 1 to remain the same.
- The condition code is set accordingly.
- You can specify the exact byte in operand 1 you want used in the execution with the one byte in operand 2 through relative addressing.

Example:

```
LABEL   ΔOPERATIONΔ                 OPERAND
1        10        16
-----------------------------------------------------------
         NI        RESULT+1,B'10001011'
         .
         .
         .
RESULT   DC        BL2'0000111101101100'
```

RESULT before execution of NI instruction:

RESULT		RESULT + 1		
0000	1111	0110	1100	binary
0	F	6	C	hex
0	7	8	15	

Operand 2 before and after execution of NI instruction:

1000	1011	binary
8	B	hex
0	7	

RESULT after execution of NI instruction:

RESULT		RESULT + 1		
0000	1111	0000	1000	binary
0	F	0	8	hex
0	7	8	15	

In this example, the content of RESULT is a 2-byte binary string of 0's and 1's and the immediate operand 2 is a 1-byte binary string of 0's and 1's. A logical AND operation is performed on the contents of the second byte of RESULT and the one byte in operand 2. The result replaces the second byte of RESULT. The 1's in bit positions 0, 4, 6, and 7 of the immediate operand allow the corresponding bit positions in the second byte of RESULT to remain the same. Zeros in bit positions 1, 2, 3, and 5 of the immediate operand set the corresponding bit positions in the second byte of RESULT to 0. As a result, the second byte of RESULT has been changed from a hexadecimal 6C to a hexadecimal 08.

NR

12.7. AND (NR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
NR	14	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *and* (NR) instruction performs a logical AND operation on the contents of the operand 1 and operand 2 registers. The result is placed in the operand 1 register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	NR	r_1,r_2

When the NR instruction is executed, a logical AND operation is performed on a bit in the operand 1 register and a bit in the operand 2 register. The result replaces the bit accessed in operand 1. This instruction operates from left to right starting with the logical AND operation of bit 0 in both registers up to and including the logical AND operation of bit 31 in both registers.

The NR instruction is used to turn off selected bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
0	1	0
1	0	0
1	1	1

When coding patterns used as operands in AND instructions, code 0's in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to off (0), and a 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.

After the NR instruction is executed, the condition code is set to 0 if the result is all 0's, or the condition code is set to 1 if the result is all 1's or a combination of 1's and 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- The logical AND operation executes upon all 32 bit positions of the operand 1 and 2 registers.
- A zero in a bit position in operand 2 sets its corresponding bit position in operand 1 to 0.
- A 1 in a bit position in operand 2 allows its corresponding bit position in operand 1 to remain the same.
- The condition code is set accordingly.

Example:

```
LABEL   ΔOPERATIONΔ                       OPERAND
1        10        16
-----------------------------------------------------------
         LM        5,6,INFOIN
         NR        5,6
         .
         .
         .
         DS        ØF
INFOIN   DC        X'FFCCBBAA'
         DC        X'CCBBEEDD'
```

Register 5 before execution of NR instruction:

1111	1111	1100	1100	1011	1011	1010	1010	binary
F	F	C	C	B	B	A	A	hex

Register 6 before and after execution of NR instruction:

1100	1100	1011	1011	1110	1110	1101	1101	binary
C	C	B	B	E	E	D	D	hex

Register 5 after execution of NR instruction:

1100	1100	1000	1000	1010	1010	1000	1000	binary
C	C	8	8	A	A	8	8	hex

In this example, the LM instruction loads the two consecutive hexadecimal values aligned on a full-word boundary in main storage into registers 5 and 6. Then, a logical AND operation is performed on all 32 bits of registers 5 and 6. The result replaces register 5. The 1's in respective bit positions in register 6 allow the corresponding bit positions in register 5 to remain the same. The zeros in the remaining bit positions in register 6 set the corresponding bit positions in register 5 to 0. In effect, the content of register 5 is completely changed.

CL

12.8. COMPARE LOGICAL (CL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CL	**55**	**RX**	**4**

Condition Codes

- ■ IF r_1 = OPERAND 2, SET TO 0
- ■ IF r_1 < OPERAND 2, SET TO 1
- ■ IF r_1 > OPERAND 2, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *compare logical* (CL) instruction logically compares the contents of the operand 1 register to the full word in operand 2. The result of the comparison determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CL	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CL	$r_1,s_2(x_2)$

Both operands 1 and 2 are considered unsigned binary values with all codes valid. That is, the comparison takes place regardless of data format. This instruction operates from left to right starting with the logical comparison of bit 0 in both operands and ending as soon as an inequality is reached, or the logical comparison of bit 31 in both operands is reached.

After execution of the CL instruction, the condition code is set:

To 0 if operand 1 = operand 2

To 1 if operand 1 < operand 2

To 2 if operand 1 > operand 2

Usually, a conditional branch instruction tests the resulting condition code for an equal to, less than, or greater than condition. If the condition is met, a branch takes place accordingly. If not, the program continues processing as shown in the following coding instruction.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must either be defined as a full word or aligned on a full-word boundary.
- Both operands 1 and 2 are considered unsigned binary values.
- The condition code is set accordingly.
- Condition code 3 is not used.
- Operands 1 and 2 remain unchanged after execution of this instruction.

Example:

```
LABEL   ΔOPERATIONΔ                     OPERAND
1        10        16
-----------------------------------------------------
         SR        4,4
         L         8,=F'75'
         CL        8,FULVAL
         BH        LOOP1
         ST        8,LOW
         .
         .
         .
LOOP1    AR        4,8
         .
         .
         .
FULVAL   DC        F'64'
LOW      DS        F
```

Register 5 before and after execution of CL instruction.

0000	0000	0000	0000	0000	0000	0100	1011	binary
0	0	0	0	0	0	4	B	hex

FULVAL before and after execution of CL instruction:

0000	0000	0000	0000	0000	0000	0100	0000	binary
0	0	0	0	0	0	4	0	hex

In this example, register 4 is cleared to 0 and the full-word value coded as a literal is loaded into register 8. Then the content of register 8 is logically compared to the full-word value in FULVAL. As the content of register 8 is greater than the content of FULVAL, the condition code is set to 2 and the branch to the instruction labeled LOOP1 takes place. If the result of the comparison was other than a greater than condition, no branch takes place and the ST instruction following the branch instruction is executed.

CLC

12.9. COMPARE LOGICAL CHARACTERS (CLC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLC	D5	SS	6

Condition Codes

- ■ IF OP1 = OP2, SET TO 0
- ■ IF OP1 < OP2, SET TO 1
- ■ IF OP1 > OP2, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *compare logical characters* (CLC) instruction logically compares the contents of operand 1 located in main storage to the contents of operand 2 located in main storage. The result of the comparison determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLC	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLC	$s_1(l),s_2$

Both operands 1 and 2 are considered unsigned binary values with all codes valid. That is, the comparison takes place regardless of data format. This instruction operates from left to right starting with the logical comparison of bit 0 in both operands and ending as soon as an inequality is reached (or the end of the field is reached). The length of operand 1, whether implied or explicit, determines the length of operand 2. Therefore, when the CLC instruction is executed, the length of operands 1 and 2 are the same.

After execution of the CLC instruction, the condition code is set:

To 0 if operand 1 = operand 2

To 1 if operand 1 < operand 2

To 2 if operand 1 > operand 2

Usually, a conditional branch instruction tests the resulting condition code for an equal to, less than, or greater than condition. If the condition is met, a branch takes place accordingly. If not, the program continues processing as shown in the following coding instruction.

Operational Considerations:

- Operands 1 and 2 must be located in main storage.
- Both operands 1 and 2 are considered unsigned binary values.
- The length of operand 1, whether implied or explicit, determines the length of operand 2.
- The condition code is set accordingly.
- Operands 1 and 2 remain unchanged after the execution of this instruction.
- Condition code 3 is not used.

Example:

```
LABEL   ΔOPERATIONΔ                     OPERAND
1        10          16
--------------------------------------------------------------
         CLC         MONTH1(8),MONTH2
         BE          ADRTN
         MVC         MONTH1(8),MONTH2
         .
         .
         .
ADRTN    AP          TOTAL,MTD
         .
         .
         .
MONTH1   DC          CL8'NOVEMBER'
MONTH2   DC          CL8'DECEMBER'
TOTAL    DC          PL3'28ØØ'
MTD      DC          P'524'
```

MONTH1 before and after execution of CLC instruction:

	N		O		V		E		M		B		E		R		
binary	1101	0101	1101	0110	1110	0101	1100	0101	1101	0100	1100	0010	1100	0101	1101	1001	
hex	D	5	D	6	E	5	C	5	D	4	C	2	C	5	D	9	

MONTH2 before and after execution of CLC instruction:

	D		E		C		E		M		B		E		R	
binary	1100	0100	1100	0101	1100	0011	1100	0101	1101	0100	1100	0010	1100	0101	1101	1001
hex	C	4	C	5	C	3	C	5	D	4	C	2	C	5	D	9

In this example, the content of MONTH1 is logically compared to the content of MONTH2. Since the content of MONTH1 (its binary value) is greater than the contents of MONTH2, the condition code is set to 2. The following branch instruction tests for an equal to condition (condition code of 0). Because that condition does not exist, no branch is taken, the MVC instruction following the BE branch instruction is executed, and the program continues processing.

12.10. COMPARE LOGICAL CHARACTERS LONG (CLCL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLCL	0F	RR	2

Condition Codes

- ■ IF OP 1 = OP 2, SET TO 0
- ■ IF OP 1 < OP 2, SET TO 1
- ■ IF OP 1 > OP 2, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

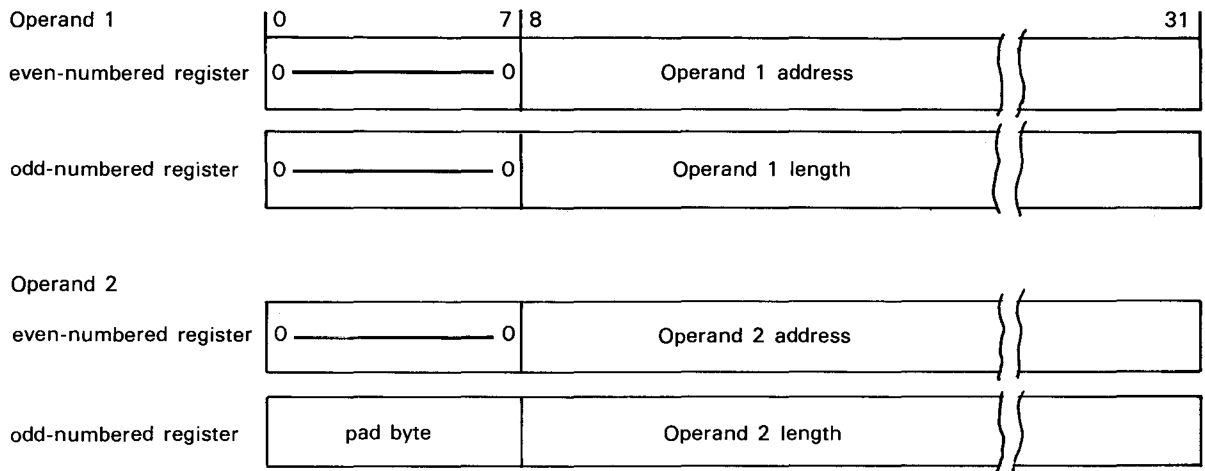
- ■ ADDRESSING
- ☐ DATA (INVALID SIGN/DIGIT)
- ☐ DECIMAL DIVIDE
- ☐ DECIMAL OVERFLOW
- ☐ EXECUTE
- ☐ EXPONENT OVERFLOW
- ☐ EXPONENT UNDERFLOW
- ☐ FIXED-POINT DIVIDE
- ☐ FIXED-POINT OVERFLOW
- ☐ FLOATING-POINT DIVIDE
- ☐ OPERATION
- ■ PROTECTION
- ☐ SIGNIFICANCE
- ■ SPECIFICATION:
  - ☐ NOT A FLOATING-POINT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

The *compare logical characters long* (CLCL) instruction logically compares operands 1 and 2, both of which are areas in main storage, and sets the condition code according to the result. The two operands need not be the same length. If they are not, a padding character specified by operand 2 is used to extend the shorter operand; this character takes part in the comparison.

Explicit and Implicit Formats:

LABEL	△OPERATION△	OPERAND
[symbol]	CLCL	r_1, r_2

The CLCL instruction uses two even-odd register pairs to specify the operands to be compared:



In action, the CLC instruction logically compares the byte at the operand 1 address with the byte at the operand 2 address. If the bytes are equal, the next bytes in both main storage areas are compared. This process continues byte by byte, proceeding left to right, until either an inequality is found or until the longer of the two operands has been completely scanned and the two operands found equal. For comparison purposes all codes and digits are considered valid. If an inequality is found, the condition code is set accordingly and the instruction terminates.

If one operand is shorter than the other, comparison proceeds as previously described until all of the shorter operand has been scanned (up to which point the two operands must have been equal). Beyond this point, comparison continues between the remaining bytes of the longer operand and a single pad byte that you specify in bits 0—7 of the odd-numbered register in operand 2. Comparison ends when an inequality is found or when all remaining bytes have been scanned and found equal to the pad byte.

When at least one operand exceeds 256 bytes in length, the CLCL instruction breaks execution down into units of operation, each of which compares 256 bytes. Interruptions are allowed between units; the CLCL instruction always responds by updating its registers so that execution can resume exactly where it left off.

If execution of the CLCL instruction ends because of an inequality, the operand registers contain information about the operation as follows:

- Odd-numbered register, both operands:

  Contain their original operand lengths reduced by the number of equal bytes scanned. If the pad byte is being used at the time, the length of the shorter operand is 0.

- Even-numbered registers, both operands:

  Contain their original addresses increased the same number of bytes by which their corresponding length registers are reduced. For operands that are equal (the pad byte included if used), both length registers have a zero value when execution finishes.

Programming Considerations:

- Both r_1 and r_2 must be specified as even registers.

- When operand lengths differ, padding always occurs no matter which register pair specifies the shorter operand. Nevertheless, the pad byte must be specified when needed in the high order byte of the operand 2 odd-numbered register.

- If the contents of the r_1 and r_2 registers are identical, condition code 0 is set.

- One or both operands can have zero length. If only one operand has it, all comparisons take place between the other operand and the pad byte. If both lengths are 0, condition code 0 is set and the instruction terminates.

Condition Code:

After execution of the CLCL instruction, the condition code is set:

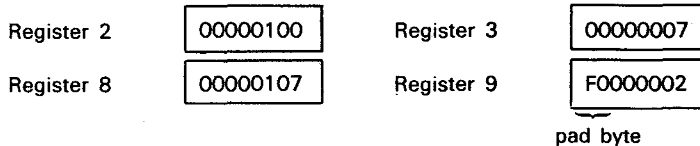
- to 0 if the two operands are equal, or if both operand lengths are zero;
- to 1 if operand 1 is less than operand 2; or
- to 2 if operand 1 is greater than operand 2.

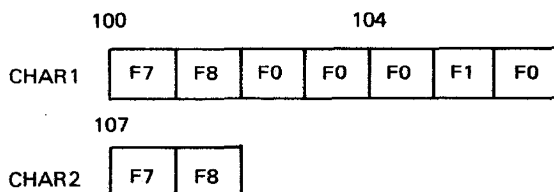
Condition code 3 is not used.

Example:

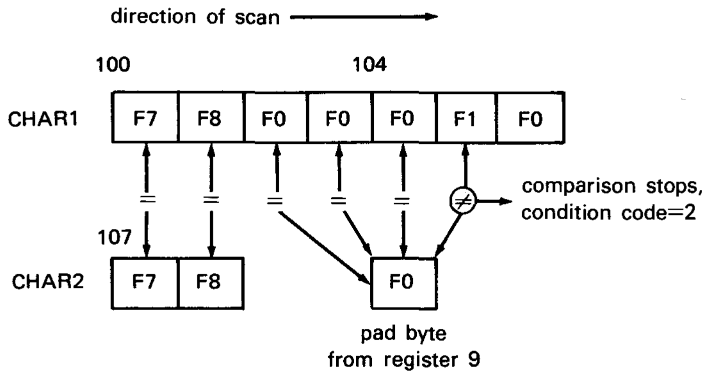
```
     LABEL     ΔOPERATIONΔ                          OPERAND
     1         10        16
1.             LA        2,CHAR1
2.             LA        3,7
3.             LA        8,CHAR2
4.             LA        9,2
5.             O         9,PADBYTE
6.             CLCL      2,8
                .
                .
     CHAR1     DC        CL7'78ØØØ1Ø'
     CHAR2     DC        CL2'78'
               DS        ØF
     PADBYTE   DC        XL4'FØØØØØØØ'
```

In this example, a 7-byte area in main storage starting at CHAR1 is compared to a 2-byte area, also in main storage, starting at CHAR2. The starting addresses for the two operands are loaded in registers 2 and 8. The respective operand lengths are loaded in registers 3 and 9. At line 5, a logical OR operation on register 9 moves the pad byte F0 into the high order (leftmost) eight bits of register 9. At line 6, the CLCL instruction is executed. When execution begins, the registers and main storage used are as follows (assuming CHAR1 to be at location 100):





Comparison proceeds as follows:



After the first two comparisons, all bytes are equal and the instruction runs out of operand 2 (CHAR2) data. This causes all further comparisons to be made using the pad byte contained in register 9, which is the odd-numbered register of operand 2. The next three bytes of CHAR1 are found to be equal to the pad byte. However, the byte at operand 1 location CHAR1+5 is greater than the pad byte, so the condition code is set to 2 and CLCL execution stops there.

The CLCL registers, after execution is finished, contain:

Register 2	00000105	Register 3	00000002
Register 8	00000109	Register 9	F0000000

Notice that register 2 gives the exact location of the operand 1 byte that caused the inequality.

CLI

12.11. COMPARE LOGICAL IMMEDIATE (CLI)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLI	95	SI	4

Condition Codes

- ■ IF OPERAND 1 = i_2, SET TO 0
- ■ IF OPERAND 1 < i_2, SET TO 1
- ■ IF OPERAND 1 > i_2, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *compare logical immediate* (CLI) instruction logically compares the content of operand 1 located in main storage to the 1-byte immediate data of operand 2. The result of the comparison determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLI	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLI	s_1,i_2

Both operands 1 and 2 are considered unsigned binary values with all codes valid. That is, the comparison takes place regardless of data format. The length of operand 1 can vary in length but the length of operand 2 is always one byte. Although the lengths of operands 1 and 2 differ, only one byte of operand 1 is compared to the one byte of immediate data in operand 2. If you don't specify the exact byte in operand 1 you want logically compared to the one byte of data in operand 2, the first byte of operand 1 is used. This instruction operates from left to right starting with the logical comparison of bit 0 of the byte specified in operand 1 and bit 0 of operand 2, and ending as soon as an inequality is found, or the logical comparison of bit 7 of the byte specified in operand 1 and operand 2 is reached.

After the execution of the CLI instruction, the condition code is set:

To 0 if operand 1 = operand 2

To 1 if operand 1 < operand 2

To 2 if operand 1 > operand 2

Usually, a conditional branch instruction tests the condition code for an equal to, less than, or greater than condition. If that condition is met, the branch takes place. If not, no branch takes place and the program continues processing as shown in the following coding instruction.

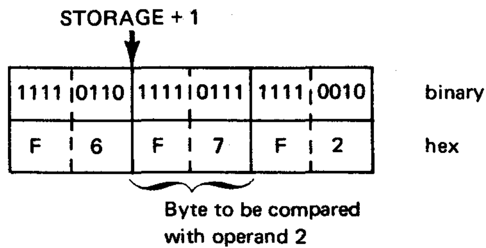
Operational Considerations:

- Operand 1 must be located in main storage.
- Operand 2 must be a 1-byte self-defining term.
- You can specify the exact byte in operand 1 that you want logically compared to the one byte in operand 2 through relative addressing.
- Operands 1 and 2 remain unchanged after the execution of this instruction.
- The condition code is set accordingly.
- The length of operand 1 can vary.
- Condition code 3 is not used.

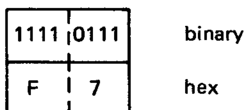
Example:

```
LABEL   ΔOPERATIONΔ                     OPERAND
1        10        16
-----------------------------------------------------
         CLI       STORAGE+1,X'F7'
         BE        EQUALITY
         PACK      STORAGEP(2),STORAGE(3)
         .
         .
         .
EQUALITY MVC       BUF(1),STORAGE+1
         .
         .
         .
BUF      DC        CL2'Ø'
STORAGE  DC        X'F6F7F2'
STORAGEP DC        PL2'Ø'
```

STORAGE before and after execution of CLI instruction:



Operand 2 immediate before and after execution of CLI instruction:



In this example, the second byte in STORAGE is compared to the 1-byte immediate data in operand 2. As the content of the second byte of STORAGE is equal to operand 2, the condition code is set to 0, and the branch to the instruction labeled EQUALITY takes place. If the result of the comparison is not equal, no branch takes place, the PACK instruction following the branch instruction is executed, and the program continues processing.

Example:

```
LABEL    △OPERATION△                    OPERAND
1        10       16
------------------------------------------------------------
         LA       8,526
         CLI      AREA,C'T'
         BE       TOOLRTN
         MVI      HOLD,C'T'
         .
         .
         .
TOOLRTN  S        8,=F'1'
         .
         .
         .
AREA     DC       CL3'T12'
HOLD     DC       CL1' '
```

AREA before and after execution of CLI instruction:

1110	0011	1111	0001	1111	0010	binary
E	3	F	1	F	2	hex

Byte to be compared with operand 2

Operand 2 immediate before and after execution of CLI instruction:

1110	0011	binary
E	3	hex

In this example, register 8 is loaded with a value of 526. Then the contents of the first byte of AREA is logically compared to operand 2. Because no one byte is specified in AREA, the first byte is used. As the content of byte 1 of AREA is equal to the content of operand 2, the condition code is set to 0, and the branch to the instruction labeled TOOLRTN takes place. If the result of the comparison is not equal, no branch takes place, the MVI instruction following the branch instruction is executed, and the program continues processing.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1        10        16
-----------------------------------------------------------
         LM        3,4,LOADREG
         CLI       NUMIN,X'C1'
         BE        STOCKNO
         MVI       NEWHOLD,NUMIN
         .
         .
         .
STOCKNO  AR        3,4
         .
         .
         .
NEWHOLD  DC        CL1' '
NUMIN    DC        CL4'A256'
LOADREG  DC        F'5264'
         DC        F'1'
```

NUMIN before and after execution of the CLI instruction:

1100	0001	1111	0010	1111	0101	1111	0110	binary
C	1	F	2	F	5	F	6	hex

Operand 2 immediate before and after execution of the CLI instruction:

1100	0001	binary
C	1	hex

In this example, two consecutive full words in main storage are loaded into registers 3 and 4. The first byte of NUMIN is logically compared to the 1-byte immediate in operand 2. Because no one byte is specified in NUMIN, the first byte is moved. As the content of byte 1 of NUMIN is equal to the content of operand 2, the condition code is set to 0, and the branch to the instruction labeled STOCKNO takes place. If the result of the comparison is not equal, no branch takes place, the MVI instruction following the branch instruction is executed, and the program continues processing.

CLIS

12.12. COMPARE LOGICAL IMMEDIATE AND SKIP (CLIS)

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
CLIS	E1	SM	6

Condition Codes

- ■ IF OP 2 = OP 3, SET TO 0
- ■ IF OP 2 < OP 3, SET TO 1
- ■ IF OP 2 > OP 3, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

- ■ ADDRESSING
- ☐ DATA (INVALID SIGN/DIGIT)
- ☐ DECIMAL DIVIDE
- ☐ DECIMAL OVERFLOW
- ☐ EXECUTE
- ☐ EXPONENT OVERFLOW
- ☐ EXPONENT UNDERFLOW
- ☐ FIXED-POINT DIVIDE
- ☐ FIXED-POINT OVERFLOW
- ☐ FLOATING-POINT DIVIDE
- ☐ OPERATION
- ■ PROTECTION
- ☐ SIGNIFICANCE
- ■ SPECIFICATION:
  - ☐ NOT A FLOATING-POINT REGISTER
  - ☐ OP 1 NOT ON HALF-WORD BOUNDARY
  - ☐ OP 2 NOT ON HALF-WORD BOUNDARY
  - ☐ OP 2 NOT ON FULL-WORD BOUNDARY
  - ☐ OP 2 NOT ON DOUBLE-WORD BOUNDARY
  - ☐ OP 1 NOT EVEN NUMBERED REGISTER
  - ☐ OP 1 NOT ODD NUMBERED REGISTER
  - ■ LOW-ORDER BIT OF OP 4 DISPLACEMENT MUST BE ZERO

The *compare logical immediate and skip* (CLIS) instruction logically compares a byte in main storage, addressed by operand 1, with a byte of immediate data in operand 2, setting the condition code accordingly. A 4-bit mask that you specify in operand 3 uses the condition code to determine if program control goes to the next sequential instruction or branches to another location in the program. That location is specified by operand 4 as an offset from the instruction immediately following the CLIS instruction.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLIS	$d_1(b_1),i_2,m_3,d_4$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLIS	s_1,i_2,m_3,d_4

The CLIS instruction operates much like a combined COMPARE LOGICAL IMMEDIATE and BRANCH ON CONDITION instruction. It compares the byte in main storage addressed by operand 1 to operand 2, a byte of immediate data. This comparison sets the condition code to 0, 1, or 2. At this point, the instruction calls on a 4-bit mask, specified by m_3, to determine where program control goes. Each bit in the mask corresponds to a condition code as follows:

Mask Value	8	4	2	1
Object bit position	16	17	18	19
Condition code	0	1	2	3

If a condition code is set and the mask bit corresponding to that code has a value of 1, program control branches to the address specified in the operand 4 displacement. If the mask bit has a 0 value, program control unconditionally passes to the next sequential instruction.

When a condition code mask bit is set to 1, the CLIS instruction algebraically adds the 12-bit displacement value contained in operand 4 to the value of the current program status word (PSW), effectively causing a branch to the new address. The displacement can be positive or negative, signifying, respectively, a branch forward or backward. The sign of the displacement is determined by its high order (leftmost) bit: 0 for positive, 1 for negative. Addition of the displacement to the current PSW takes place using the rules of twos complement arithmetic.

You can specify the m_3 mask either with an absolute value or by coding an extended mnemonic in place of CLIS in your assembler source program. The assembler provides six of these mnemonics for the CLIS instruction: all assemble into CLIS object instructions but each generates a different m_3 mask value as shown in the following list.

Mnemonic	Opcode	m_3 mask	Resulting action
CLIBH	E1	2	Branch if operand 1 is greater than operand 2
CLIBL	E1	4	Branch if operand 1 is less than operand 2
CLIBE	E1	8	Branch if operand 1 is equal to operand 2
CLIBNH	E1	D	Branch if operand 1 is not greater than operand 2
CLIBNL	E1	B	Branch if operand 1 is not less than operand 2
CLIBNE	E1	7	Branch if operand 1 is not equal to operand 2

When using an extended mnemonic you use only three operands: the operand 1 address, the operand 2 immediate byte, and the operand 4 displacement, in that order.

You can specify the displacement value yourself or you can let the assembler do it for you:

- You can code the displacement as an absolute expression. In this case the assembler inserts the expression, unchanged, into the displacement field.
- You can code the displacement as a relocatable expression. In this case the assembler simulates a branch to the specified location by calculating its offset from the address of the instruction immediately following CLIS and inserting the offset in the displacement field.

Operational Considerations:

- The displacement field can range from —2048 decimal bytes to +2046 bytes.
- The displacement must always be an even number of bytes because destination instructions must always lie on a half-word boundary.
- A mask of 0 (0000_2) causes the instruction always to branch to the next sequential instruction regardless of the condition code set. Likewise, a displacement value of 0 causes an unconditional branch to the next instruction.
- A mask of 15 (1111_2) causes the CLIS instruction always to branch to the instruction specified by the displacement field.
- You must specify both the mask and the immediate byte as self-defining terms.

Condition Code:

After execution of the CLIS instruction, the condition code is set:

- to 0 if the operand 1 byte is equal to the i_2 byte;
- to 1 if the operand 1 byte is less than the i_2 byte; or
- to 2 if the operand 1 byte is greater than the i_2 byte.

Condition code 3 is not used.

Example:

```
    LABEL     ΔOPERATIONΔ                        OPERAND                    ADDRESS
    1         10        16
   ------------------------------------------------------------------------------
1. |          PACK      PACKSTR(3),CHARSTR+1(5)                             ØØØ1ØØ
2. |          CLIS      CHARSTR,X'6Ø',7,POS                                 ØØØ1Ø6
3. | NEG      NI        PACKSTR+2,X'FD'                                     ØØØ1ØC
4. |          B         COMSTEP                                             ØØØ11Ø
5. | POS      NI        PACKSTR+2,X'FC'                                     ØØØ114
6. | COMSTEP  .                                                             ØØØ118
   |          .
   |          .
   | CHARSTR  DC        CL6'-44Ø31'
   | PACKSTR  DS        CL3
```

In this example, a 5-byte EBCDIC number at CHARSTR is packed into a 3-byte number at PACKSTR, and the leading sign of CHARSTR is attached to PACKSTR. The object code addresses of the instructions are shown in the right margin in the preceding example. Before execution of the CLIS instruction in line 2, CHARSTR and PACKSTR contain:

	–	4	4	0	3	1
CHARSTR	60	F4	F4	F0	F3	F1
PACKSTR	44	03	1F			

In the PACK operation, PACKSTR becomes an unsigned positive number; its actual sign is subsequently set by the CLIS instruction which:

- compares the first byte of CHARSTR against an immediate byte, X'60' (EBCDIC minus sign);
- sets up a branch to location POS if conditions are met, by adding the displacement between the next sequential instruction (at NEG) and POS (114—10C=8 bytes) to the current PSW; and
- meets those conditions, causing the branch, if condition codes 1, 2, or 3 are set (mask value 4+2+1=7).

In action, the CLIS instruction finds CHARSTR equal to X'60', setting condition code 0. Since the mask bit for condition code 0 is not set, program control passes to the next sequential instruction, which is a NI instruction at location NEG that, in effect, attaches a packed negative sign to PACKSTR:

PACKSTR	44	03	1D

If CHARSTR were anything but X'60', program control would pass to location POS, which attaches a packed positive sign to PACKSTR.

12.13. COMPARE LOGICAL CHARACTERS UNDER MASK (CLM)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLM	BD	RS	4

Condition Codes
■ IF OP 1 = OP 2, SET TO 0
■ IF OP 1 < OP 2, SET TO 1
■ IF OP 1 > OP 2, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *compare logical characters under mask* (CLM) instruction compares some or all of the bytes within the operand 1 register with a main storage location starting at the second operand address. The bytes to be compared are determined by a mask derived from operand 3. The condition code is set according to the result.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLM	$r_1,m_3,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLM	r_1,m_3,s_2

The CLM instruction compares all or part of operand 1 with contiguous main storage data starting at the operand 2 address. The main storage bytes to be compared are contiguous, but you can select the bytes to be compared using a 4-bit mask (m_3) defined by operand 3. Each bit in the mask corresponds to a byte within the operands. If a bit equals 1 its byte participates in the comparison; if the bit equals 0 its byte does not participate. The mask value of the bits and their corresponding bytes are as follows:

Mask value	8	4	2	1
Object code bit	12	13	14	15
Byte (left to right)	1	2	3	4

Thus, a mask value of 10 (8+2) specifies that the first and third bytes of the register are to take part in the comparison. The number of bytes in the operand 2 field equals the number of 1-bits in the mask. In operation, the leftmost register byte whose mask bit equals 1 is logically compared to the byte addressed by operand 2. Then, the next byte within the register having a mask bit of 1 is compared to the main storage byte immediately following the one used in the preceding comparison. This process is repeated for all 1-bits within the mask. Each comparison treats its bytes as unsigned binary data, the comparisons proceeding from left to right.

Operational Considerations:

- The operand 3 mask must be a self-defining term ranging from 0 to 15.
- With a mask of 15 (1111_2), the CLM instruction acts like a COMPARE instruction, comparing all four bytes within the operand 1 register to four bytes in main storage. The only difference between the two instructions is that the four bytes addressed by CLM need not reside on a full-word boundary.

Condition Code:

After execution of CLM, the condition code is set:

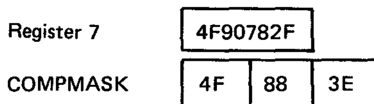
- to 0 if the mask is all 0's or if the selected bytes are equal;
- to 1 if the selected bytes of operand 1 are less than the corresponding bytes of operand 2; or
- to 2 if the selected bytes of operand 1 are greater than the corresponding bytes of operand 2.

Condition code 3 is not set.

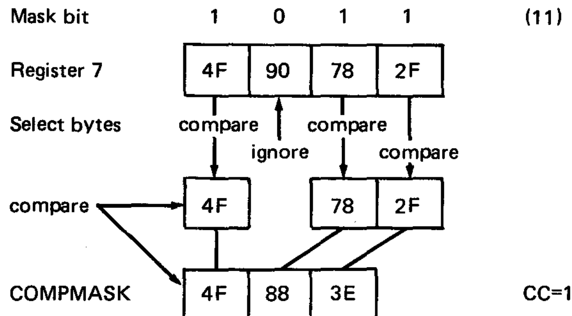
Example:

```
     LABEL     ΔOPERATIONΔ                         OPERAND
     1         10          16
   ---------------------------------------------------------------
1.             L           7,FWORD
2.             CLM         7,11,COMPMASK
               .
               .
               .
               DS          ØF
     FWORD     DC          XL4'4F9Ø782F'
     COMPMASK  DC          XL3'4F883E'
```

In this example, register 7 is compared to three bytes in main storage starting at COMPMASK. At the beginning of the CLM instruction, register 7 and COMPMASK have the following contents:



The CLM instruction has a mask of 11 that causes the instruction to operate as follows:



The mask bits cause the CLM instruction to compare the first, third, and fourth bytes of register 7 with the three bytes at COMPMASK. The first operand 2 byte equals the first operand 1 byte; therefore, no conclusions can yet be drawn. However, the second operand 2 byte (88) exceeds the second operand 1 byte (78), thus making the entire quantity at COMPMASK greater than the selected bytes in register 7. As a result, CLM sets the condtion code to 1.

CLR

12.14. COMPARE LOGICAL (CLR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLR	15	RR	2

Condition Codes

- ■ IF $r_1 = r_2$, SET TO 0
- ■ IF $r_1 < r_2$, SET TO 1
- ■ IF $r_1 > r_2$, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *compare logical* (CLR) instruction logically compares the content of the operand 1 register to the content of the operand 2 register. The result of the comparison determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLR	r_1,r_2

Both operands 1 and 2 are considered unsigned binary values with all codes valid. That is, the comparison takes place regardless of data format. This instruction operates from left to right starting with the logical comparison of bit 0 in both operands and ending as soon as an inequality is found, or the logical comparison of bit 31 in both operands is reached.

After execution of the CLR instruction, the condition code is set:

To 0 if operand 1 = operand 2

To 1 if operand 1 < operand 2

To 2 if operand 1 > operand 2

Usually, a conditional branch instruction tests the resulting condition code for an equal to, less than, or greater than condition. If the condition is met, a branch takes place accordingly. If not, the program continues processing as shown in the following coding instruction.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- Both operands 1 and 2 are considered unsigned binary values.
- The condition code is set accordingly.
- Operands 1 and 2 remain unchanged after the execution of this instruction.
- Condition code 3 is not used.

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1        10        16
------------------------------------------------------------
         SR        7,7
         L         5,=F'1250'
         A         7,=F'875'
COMPARE  CR        5,7
         BH        ADD2
         CVD       7,DBLWD
         B         END
ADD2     AH        7,=H'375'
         B         COMPARE
END      MVC       BUF,DBLWD+5
         .
         .
         .
DBLWD    DS        D
BUF      DC        PL3'Ø'
```

In this example, register 7 is cleared to 0. A full word containing the decimal value 1250 is loaded into register 5. Another full word containing the decimal value 875 is added to register 7. Then the content of register 5 is logically compared to the content of register 7. Since the content of register 5 is greater than the content of register 7, the condition code is set to 2, and the branch to the instruction labeled ADD2 takes place. There, a half word containing the decimal value 375 is added to register 7. An unconditional branch to the instruction labeled COMPARE takes place and registers 5 and 7 are logically compared again. This time, the content of register 5 is equal to the content of register 7. Because an equal to condition exists, the condition code is set to 0, and no branch takes place. The CVD instruction following the branch instruction is executed and the program continues processing.

CSM

12.15. COMPARE AND SWAP UNDER MASK (CSM)

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CSM	**B9**	**RS**	**4**

Condition Codes

- ■ IF OP 1 = OP 2, SET TO 0
- ■ IF OP 1 < OP 2, SET TO 1
- ■ IF OP 1 > OP 2, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

■ ADDRESSING	■ PROTECTION
□ DATA (INVALID SIGN/DIGIT)	□ SIGNIFICANCE
□ DECIMAL DIVIDE	■ SPECIFICATION:
□ DECIMAL OVERFLOW	□ NOT A FLOATING-POINT REGISTER
□ EXECUTE	□ OP 1 NOT ON HALF-WORD BOUNDARY
□ EXPONENT OVERFLOW	□ OP 2 NOT ON HALF-WORD BOUNDARY
□ EXPONENT UNDERFLOW	■ OP 2 NOT ON FULL-WORD BOUNDARY
□ FIXED-POINT DIVIDE	□ OP 2 NOT ON DOUBLE-WORD BOUNDARY
□ FIXED-POINT OVERFLOW	■ OP 1 NOT EVEN NUMBERED REGISTER
□ FLOATING-POINT DIVIDE	■ OP 3 NOT EVEN NUMBERED REGISTER
□ OPERATION	□ NONE

The *compare and swap under mask* (CSM) instruction logically compares a full word of main storage addressed by operand 2 with some or all of the odd-numbered register of an even-odd register pair specified by operand 1. If the two operands are not equal the condition code is set and execution ends there. If the two operands are equal the condition code is set, and then the instruction replaces some or all of the operand 2 field with data contained in the odd-numbered register of an even-odd register pair specified by operand 3. You select the bits to be compared using a 32-bit mask in the even-numbered register of operand 1. You select the operand 2 bits to be replaced using a 32-bit mask in the even-numbered register of operand 3.

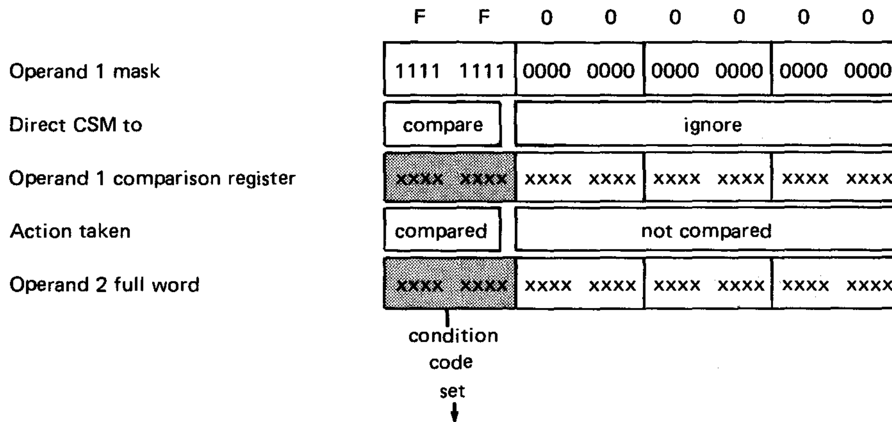
Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CSM	$r_1,r_3,d_2(b_2)$

Implicit Format:

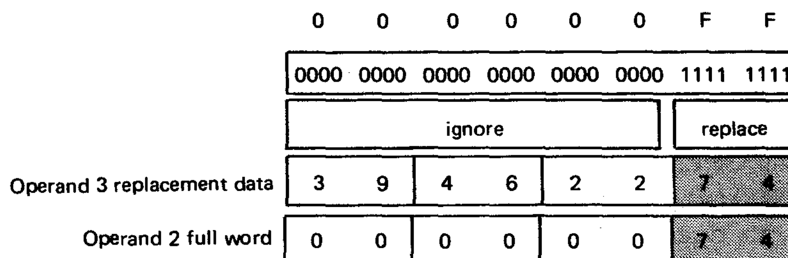
LABEL	△OPERATION△	OPERAND
[symbol]	CSM	r_1,r_3,s_2

The CSM instruction acts basically like a sequence of the COMPARE, BRANCH ON CONDITION, and STORE instructions. What separates the CSM instruction from the sequence is the use of 32-bit register masks in both the comparison and replacement steps. The comparison step uses an even-odd register pair, the even-numbered member of which is specified as operand 1. Within this pair, the odd-numbered register provides the data to be logically compared against the operand 2 full word, and the even-numbered register determines, using a combination of 0 and 1 bits, which bits actually take part in the comparison. Only operand 2 bits whose corresponding bits in the mask are 1 take part. A zero bit in the mask causes its corresponding bit to be ignored, to have no effect on the result of the comparison. Thus, an operand 1 mask containing $FF000000_{16}$ causes comparison to take place as follows:



With this mask, comparison takes place using only the first byte of operands 1 and 3. The operands as a whole may be unequal, but in this instance equality is determined solely by comparing their respective first bytes. All signs and codes are considered valid for comparison purposes. The comparison step sets the condition code, which remains unchanged for the remainder of the instruction.

If the masked operand 1 register does not equal the corresponding data in the operand 2 full word, the CSM instruction terminates and control passes to the next instruction in sequence. If the operands are equal, the instruction proceeds to its second step, using another even-odd register pair, the even-numbered member of which is specified as operand 3. The odd-numbered register makes available 32 bits of data for replacement of data in the operand 2 full word. The bits that the instruction actually replaces are determined by the mask contained in the even-numbered register. If a mask bit position has a value of 1, the corresponding bit in the operand 3 odd-numbered register replaces the corresponding bit in the operand 2 full word. If a mask bit position has a value of 0, its corresponding bit in operand 2 remains unchanged. Thus, an operand 3 mask of $00FFFFFF_{16}$ causes bit replacement to take place as follows:



The condition code is set according to the result of the comparison step and remains unchanged through the rest of the instruction.

Operational Considerations:

- Both r_1 and r_3 must be even-numbered registers.
- Operand 2 must reside on a full-word boundary.
- An operand 1 mask of zero (00000000_{16}) sets the condition code to 0 and causes bit replacement to take place according to the operand 3 registers.
- Using r_1 and r_3 masks of $FFFFFFFF_{16}$ causes CSM to act like a COMPARE-BRANCH ON CONDITION-STORE instruction sequence in that it deals with whole registers without masking out any bits.

Condition Code:

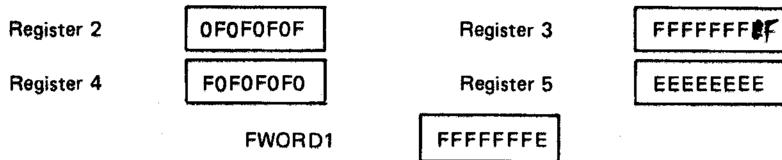
After execution of the CSM instruction, the condition code is set:

- to 0 if the masked bits of the operand 1 odd-numbered register equal their corresponding operand 2 bits (causing operand 3 bit replacement to take place);
- to 1 if the masked operand 1 data is less than the operand 2 data (preventing operand 3 bit replacement); or
- to 2 if the masked operand 1 data is greater than the operand 2 data (preventing operand 3 bit replacement).

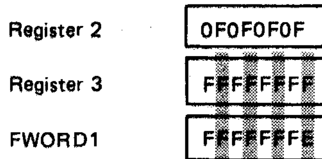
Example:

```
        LABEL    ΔOPERATIONΔ                      OPERAND
        1        10         16

1.               LM         2,5,EX1
2.               CSM        2,4, FWORD1
3.               LM         2,5,EX2
4.               CSM        2,4,FWORD2
                 .
                 .
        FWORD1   DC         F'-2'
        FWORD2   DC         F'-1'
        EX1      DC         XL8'ØFØFØFØFFFFFFFFF'
                 DC         XL8'FØFØFØFØEEEEEEEE'
        EX2      DC         XL8'ØØØ8ØØØ8FFFFFFFF'
                 DC         XL8'FØFØFØFØDDDDDDDD'
```

In this example, two CSM instructions both use full-word FWORD1 as the main storage operand 2. After the LM instruction in line 1 is executed, registers 2 through 5 and FWORD1 have the following contents:

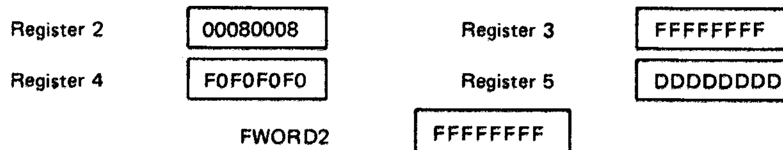


When the CSM instruction in line 2 is executed, it first uses the even-numbered operand 1 register, register 2, to mask a comparison between register 3 and FWORD1.



Only the shaded data shown takes part in the comparison. Although three of the four half bytes thus compared are equal, the low order half byte of FWORD1 is less than its corresponding register 3 half byte. This result sets the condition code to 2 and terminates CSM at once, leaving FWORD1 unchanged.

Next, the LM instruction at line 3 loads the following data into registers 2 through 5:



Upon execution of the CSM instruction in line 4, the register 2 mask governs comparison of register 3 with FWORD2 as follows:

	0	0	0	8	0	0	0	8
Register 2	0000	0000	0000	1000	0000	0000	0000	1000
	F	F	F	F	F	F	F	F
Register 3	1111	1111	1111	1111	1111	1111	1111	1111
	F	F	F	F	F	F	F	F
FWORD2	1111	1111	1111	1111	1111	1111	1111	1111

As a result of the operand 1 mask, only shaded bits 12 and 28 are compared. Because all bits involved are equal, the condition code is set to 0 and CSM execution continues with the bit replacement step. In that step, the operand 3 mask in register 4 governs bit replacement from register 5 to FWORD2 as follows:

Register 4	F0F0F0F0
Register 5	DDDDDDDD
FWORD2	DFDFDFDF

As a result of the mask, every other half byte in FWORD2 is replaced by its corresponding half byte in register 5. The condition code remains set to 0.

12.16. EXCLUSIVE OR (X)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
X	57	RX	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠ 0, SET TO 1
□ IF RESULT > 0, SET TO 2
□ IF OVERFLOW, SET TO 3
□ UNCHANGED

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

The *exclusive or* (X) instruction performs an exclusive OR operation on the content of the operand 1 register and the full word in operand 2. The result is placed in operand 1 and also determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	X	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	X	$r_1,s_2(x_2)$

When the X instruction is executed, an exclusive OR operation is performed on a bit in operand 1 and a bit in operand 2. The result of the exclusive OR operation replaces the bit just accessed in operand 1. This instruction operates from left to right starting with the exclusive OR operation of bit 0 in both operands up to and including the exclusive OR operation of bit 31 in both operands.

The X instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

When coding patterns are used as operands in exclusive OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 0's in operand 1 that you want set to 1.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 1's in operand 1 that you want set to 0.

After the X instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a full word or aligned on a full-word boundary.
- The condition code is set accordingly.
- Condition codes 2 and 3 are not used.

Example:

```
LABEL   ΔOPERATIONΔ                    OPERAND
1        10        16
---------------------------------------------------
         L         6,VAL
         X         6,XPATTERN
         .
         .
         .
         DS        ØF
VAL      DC        X'ØØØØCAF2'
XPATTERN DC        X'ØØØØC5ØD'
```

Register 6 before execution of X instruction:

0000	0000	0000	0000	1100	1010	1111	0010	binary
0	0	0	0	C	A	F	2	hex

XPATTERN before and after execution of X instruction:

0000	0000	0000	0000	1100	0101	0000	1101	binary
0	0	0	0	C	5	0	D	hex

Register 6 after execution of X instruction:

0000	0000	0000	0000	0000	1111	1111	1111	binary
0	0	0	0	0	F	F	F	hex

In this example, the content of VAL is loaded into register 6. Then the exclusive OR operation is performed on the contents of register 6 and the contents of XPATTERN. The resultant modified binary string is placed in register 6. Since the result is a combination of 0's and 1's, the condition code is set to 1.

XC

12.17. EXCLUSIVE OR (XC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
XC	D7	SS	6

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT ≠ 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *exclusive or* (XC) instruction performs an exclusive OR operation on the contents of operand 1 and operand 2, both located in main storage. The result is placed in operand 1 and also determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	XC	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	XC	$s_1(l),s_2$

When the XC instruction is executed, an exclusive OR operation is performed on a bit in operand 1 and a bit in operand 2. The result of the exclusive OR operation replaces the bit accessed in operand 1. This instruction operates from left to right. The length of operand 1, whether implied or explicit, determines the length of operand 2. Therefore, when the XC instruction is executed, the lengths of operands 1 and 2 are the same.

The XC instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

When coding patterns are used as operands in exclusive OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 0's in operand 1 that you want set to 1.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 1's in operand 1 that you want set to 0.

After the XC instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Operands 1 and 2 must be located in main storage.
- The condition code is set accordingly.
- Operands 1 and 2 can have overlapping bytes.

Example:

```
LABEL   ΔOPERATIONΔ                       OPERAND
1       10        16
------------------------------------------------------------
        XC        A,B
        XC        B,A
        .
        .
        .
A       DC        BL1'ØØØ11Ø11'
B       DC        BL1'ØØØ1ØØØ1'
```

A before execution of first XC instruction:

0001	1011	binary
1	B	hex

B before and after execution of first XC instruction:

0001	0001	binary
1	1	hex

A after execution of first XC instruction:

0000	1010	binary
0	A	hex

B before execution of second XC instruction:

0001	0001	binary
1	1	hex

A before and after execution of second XC instruction:

0000	1010	binary
0	A	hex

B after execution of second XC instruction:

0001	1011	binary
1	B	hex

In this example, the exclusive OR operation is performed on the contents of A and B. The resultant modified binary string of 1's and 0's is placed in A. Then another exclusive OR operation is performed on the contents of B and A (now modified). That resultant modified binary string of 1's and 0's is placed in B. Note that the sequence of executions of the exclusive OR operation on A and B, then B and A, results in A containing the resultant modified binary string and B containing the original contents of A. Note that the original contents of A is saved without use of another area in main storage.

12.18. EXCLUSIVE OR (XI)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
XI	97	SI	4

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠ 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *exclusive or* (XI) instruction performs an exclusive OR operation on the contents of one byte of operand 1 located in main storage and the one byte of immediate data in operand 2. The result is placed in operand 1 and also determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	XI	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	XI	s_1,i_2

When the XI instruction is executed, an exclusive OR operation is performed on a bit in operand 1 and a bit in operand 2. The result of the exclusive OR operation replaces the bit accessed in operand 1. This instruction operates from left to right. The length of operand 1 can vary but the length of operand 2 is always one byte. Although operands 1 and 2 may have differing lengths, only one byte in operand 1 is used in the exclusive OR operation. The result replaces the one byte in operand 1 that was accessed. If you do not specify the exact byte in operand 1 you want used in the exclusive OR operation, the first byte of operand 1 is used.

The XI instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

When coding patterns are used as operands in exclusive OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 0's in operand 1 that you want set to 1.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 1's in operand 1 that you want set to 0.

After the XI instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Operand 1 must be a main storage location.
- Operand 2 must be a 1-byte self-defining term.
- The length of operand 1 can vary.
- The condition code is set accordingly.
- You can specify the exact byte in operand 1 you want used with the one byte in operand 2 through relative addressing.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10          16
-------------------------------------------------------------
          CLC         ITEMNO(1),STNDNO
          BE          SWTCHON
          .
          .
          .
SWTCHON   XI          ITEMNO+1,'Ø1'
          MVC         PROCESS,ITEMNO
          .
          .
          .
ITEMNO    DC          X'F2ØØ'
STNDNO    DC          X'F2'
PROCESS   DS          CL2
```

ITEMNO before execution of XI instruction:

1111	0010	0000	0000	binary
F	2	0	0	hex

Operand 2 immediate before and after execution of XI instruction:

0000	0001	binary
0	1	hex

ITEMNO after execution of XI instruction:

1111	0010	0000	0001	binary
F	2	0	1	hex

In this example, the first byte of ITEMNO is logically compared to the content of STNDNO. Since they compare equally, the condition code is set to 0 and the branch to the instruction labeled SWTCHON takes place. There, the exclusive OR operation is performed on the first byte of ITEMNO and the one byte of data in operand 2. The result replaces the first byte in ITEMNO. The only change to the content of ITEMNO is the setting of the low order bit. This is an example of how the XI instruction can be used in setting programmed binary bit switches that are useful in testing for existing conditions within the logic of the program.

XR

12.19. EXCLUSIVE OR (XR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
XR	17	RR	2

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT ≠ 0, SET TO 1
- □ IF RESULT > 0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *exclusive or* (OR) instruction performs an exclusive OR operation on the contents of the operand 1 register and operand 2 register. The result is placed in the operand 1 register and also determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	XR	r_1,r_2

When the XR instruction is executed, an exclusive OR operation is performed on a bit in the operands of 1 and 2 registers. The result of the exclusive OR operation replaces the bit just accessed in operand 1. This instruction operates from left to right starting with the execution of the exclusive OR operation on bit 0 in both registers up to and including bit 31 in both registers.

The XR instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	0

When coding patterns are used as operands in exclusive OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 0's in operand 1 that you want set to 1.
- 1 in all bit positions in operand 2 that correspond to bit positions containing 1's in operand 1 that you want set to 0.

After the XR instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- The condition code is set accordingly.

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10       16
-----------------------------------------------------------------
          SR       4,4
          L        7,CONTENTS
          A        7,=F'25'
          CVD      7,DBLWD
          MVC      AREA,DBLWD+5
          XR       7,7
          .
          .
          .
CONTENTS  DC       F'5Ø'
          .
          .
          .
DBLWD     DS       D
AREA      DS       PL3
```

Register 7 before execution of XR instruction:

0000	0000	0000	0000	0000	0000	0011	0010	binary
0	0	0	0	0	0	4	B	hex

Register 7 after execution of XR instruction:

0000	0000	0000	0000	0000	0000	0000	0000	binary
0	0	0	0	0	0	0	0	hex

In this example, the full word in CONTENTS is loaded into register 7. A full word containing the decimal value of 25 is added to register 7, and that result is converted into its decimal equivalent and placed in DBLWD in main storage. Then the content of the last three bytes of DBLWD are moved into a smaller field, and the exclusive OR operation is performed on the contents of register 7 and itself. The result is a field of 0's. This is another method of clearing a field to 0's.

12.20. INSERT CHARACTER (IC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
IC	43	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *insert character* (IC) instruction places one byte of data from operand 2 into the rightmost byte of the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	IC	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	IC	$r_1,s_2(x_2)$

The data in operand 2 can be defined in any format. The length of operand 2 can vary but the length of operand 1 is always four bytes. Although operands 1 and 2 can have differing or equal lengths, only one byte of operand 2 is inserted into the rightmost byte of the operand 1 register. The remaining three bytes of operand 1 remain the same. If you do not specify the exact byte in operand 2 you want inserted into operand 1, the first byte of operand 2 is used.

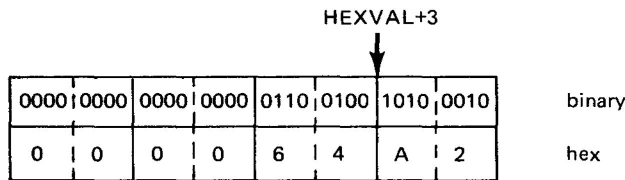
Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- The length of operand 2 can vary.
- You can specify the exact byte in operand 2 you want inserted into the rightmost byte of operand 1 through relative addressing.
- The condition code remains unchanged.

Example:

```
LABEL    △OPERATION△                          OPERAND
1         10        16
______________________________________________________________
          L         4,HEXVAL
          CLC       HEXVAL+3(1),NEWVAL
          BNE       INSERT
          .
          .
          .
INSERT    IC        4,NEWVAL
          .
          .
          .
          DS        ØF
HEXVAL    DC        X'ØØØ064A2'
NEWVAL    DC        X'F4'
```

Register 4 before execution of IC instruction:



NEWVAL before and after execution of IC instruction:

1111	0100	binary
F	4	hex

Register 4 after execution of instruction:

0000	0000	0000	0000	0110	0100	1111	0100	binary
0	0	0	0	6	4	F	4	hex

In this example, the content of HEXVAL is loaded into register 4. Then the fourth byte of HEXVAL is logically compared to the one byte in NEWVAL. Since the content of HEXVAL+3 is greater than NEWVAL, the condition code is set to 2 and the branch to the instruction labeled INSERT takes place because a not equal to condition exists. There, the 1-byte field in NEWVAL is inserted into the rightmost byte of register 4.

ICM

12.21. INSERT CHARACTERS UNDER MASK (ICM)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ICM	BF	RS	4

Condition Codes

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

Possible Program Exceptions

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

The *insert characters under mask* (ICM) instruction inserts contiguous bytes from storage starting at the operand 2 address into the operand 1 register, according to a pattern determined by a 4-bit mask in operand 3.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ICM	$r_1,m_3,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ICM	r_1,m_3,s_2

The ICM instruction replaces any or all bytes within the operand 1 register with bytes located at the operand 2 address in main storage. The bytes to be replaced are determined by the 4-bit operand 3 mask, in which each bit corresponds to a byte within the register into which data is to be inserted:

Mask value	8	4	2	1
Object code bit	12	13	14	15
Register byte (left to right)	1	2	3	4

Thus, a mask value of 9 (8+1) replaces the two end bytes in the register while a value of 6 (4+2) replaces the two middle bytes.

The number of bytes in operand 2 equals the number of bits in the operand 3 mask. In operation, the first byte at operand 2 is inserted into the leftmost byte of the register for which a 1 bit exists in the mask. The process continues from left to right within the mask, the storage field of operand 2, and the operand 1 register. Bytes within the register whose mask bits equal 0 are left unchanged.

Operational Considerations:

- Operand 3 must be a self-defining term between 0 and 15.
- Operand 2 need not reside on a full-word boundary.
- For a mask of 15 (1111_2), the ICM instruction acts like a combination of a LOAD instruction and a LOAD AND TEST instruction, moving four bytes from storage into all four bytes of the operand 1 register and setting the condition code according to the result.

Condition Code:

After execution of the ICM instruction, the condition code is set:

- to 0 if the operand 3 mask is zero, or if all bits inserted are zero;
- to 1 if the high order bit of operand 2 is 1 (making operand 2 algebraically negative); or
- to 2 if the high order bit of operand 2 is 0 (making operand 2 algebraically positive).

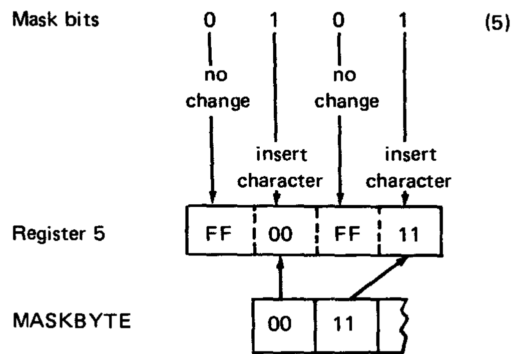
Condition code 3 is not used. Note that the condition code is set according to operand 2, not operand 1.

Example:

```
     LABEL    ΔOPERATIONΔ                     OPERAND
     1        10       16

1.            L        5,=F'-1'
2.            ICM      5,5,MASKBYTE
              .
              .
              .
     MASKBYTE DC       XL2'ØØ11'
```

In this example, the LOAD instruction in line 1 puts a value of —1 into register 5:

The ICM instruction at line 2, using a mask of 5 and main storage starting at location MASKBYTE, operates on register 5 as follows:



The first and third bits of the mask are 0; therefore, the first and third bytes of register 5 remain unchanged. But the second mask bit is 1, so the byte at MASKBYTE is inserted into the second byte of the register. Likewise, the fourth mask bit is 1, so the fourth byte of register 5 is replaced by the byte at MASKBYTE+1.

12.22. LOAD ADDRESS (LA)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LA	41	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *load address* (LA) instruction places the address of the main storage location of operand 2 into bit positions 8 through 31 (rightmost 3 bytes) of the operand 1 register. Bits 0 through 7 (leftmost byte) of the operand 1 register are set to 0's.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LA	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LA	r_1,s_2

Operand 2 can be any byte in main storage and does not have to be aligned on a full-word boundary. Operand 2 can also be a self-defining term. The three rightmost bytes of operand 1 are filled and the leftmost byte of operand 1 is set to 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Any of the general registers (1 through 15) can be used as operand 2. These registers are used as self-defining terms.
- Operand 2 can be any label in main storage.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10          16

         SR          4,4
         LA          6,CARDIN+38
         ZAP         WKLYHRS,=P'Ø'
ADDLOOP  PACK        Ø(3,6),Ø(3,6)
         AP          WKLYHRS,Ø(3,6)
         A           4,CON1
         A           6,CON3
         C           4,CON7
         BL          ADDLOOP
         .
         .
         .
CARDIN   DS          CL8Ø
WKLYHRS  DS          PL3
CON1     DC          F'1'
CON3     DC          F'3'
CON7     DC          F'7'
```

In this example, register 4 is cleared to 0. Then, the LA instruction loads the address of CARDIN+38 into register 6. The ZAP instruction sets the field labeled WKLYHRS to a packed field of 0's. Since the address of the byte located at card column 39 is in register 6, the PACK instruction packs the 3-byte field (defined in explicit format) into itself. Note that there is a displacement value of 0. Therefore, the base address is not modified through displacement values. The 3-byte packed field is now added to WKLYHRS which will eventually contain the total number of hours an employee works in one week. A full word containing the decimal value of 1 is added to register 4 each time a 3-byte field is packed and added to WKLYHRS. Register 4 acts as a counter to keep track of the number of times the ADDLOOP routine has been executed. A full word containing the decimal value of 3 is added to register 6 modifying the address by increasing it 3 bytes each time the A instruction is executed. This allows the successive fields on the input card to be processed. Then, the content of register 4 is compared to the decimal value of 7 in CON7. The branch if low (BL) instruction tests the condition code for a less than condition. Since the content of register 4 is less than the content of CON7, a branch to the instruction labeled ADDLOOP takes place. The ADDLOOP routine is executed seven times. After the seventh execution, the content of register 4 is not less than CON7 and the instruction following the BL instruction is executed.

12.23. MOVE IMMEDIATE (MVI)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MVI	92	SI	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *move immediate* (MVI) instruction places the one byte of immediate data in operand 2 into one byte of operand 1 located in main storage.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVI	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MVI	s_1,i_2

The data in operands 1 and 2 can be defined in any format. The length of operand 1 can vary but the length of operand 2 is always one byte. Although operands 1 and 2 can have differing lengths, only one byte in operand 1 receives the immediate data from operand 2. If you do not specify the exact byte in operand 1 you want to receive the operand 2 data, the first byte of operand 1 is used.

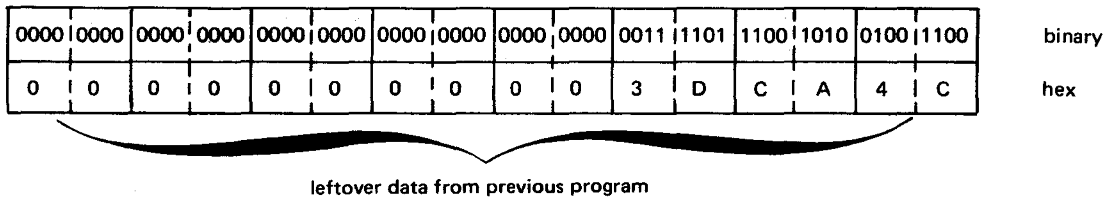
Operational Considerations:

- Operand 1 must be a main storage location.
- Operand 2 must be a 1-byte, self-defining term.
- The length of operand 1 can vary.
- You can specify the exact byte in operand 1 you want to receive the immediate data in operand 2 by relative addressing.

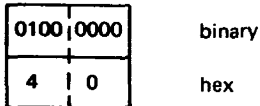
Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10          16
--------------------------------------------------------------
          MVI         OUTPUT,X'40'
          MVC         OUTPUT+1(7),OUTPUT
          .
          .
          .
OUTPUT    DS          CL8
```

OUTPUT before execution of MVI instruction:



Operand 2 immediate before and after execution of MVI instruction:



OUTPUT after execution of MVI instruction:

0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0011	1101	1100	1010	0100	1100	binary
4	0	0	0	0	0	0	0	0	0	3	D	C	A	4	C	hex

only byte changed

OUTPUT after execution of MVC instruction:

0100	0000	0100	0000	0100	0000	0100	0000	0100	0000	0100	0000	0100	0000	0100	0000	binary
4	0	4	0	4	0	4	0	4	0	4	0	4	0	4	0	hex

In this example, the one byte of immediate data in operand 2 is placed in the first byte of OUTPUT since no exact byte is specified. Then that first byte of OUTPUT is propagated through that entire field. The length attribute (in this example, 7) can be either implied or explicit and determines the number of bytes that the first byte is propagated through.

O

12.24. OR (O)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
O	56	RX	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT ≠ 0, SET TO 1
- □ IF RESULT > 0, SET TO 2
- □ IF OVERFLOW, SET TO 3
- □ UNCHANGED

Possible Program Exceptions

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

The *or* (O) instruction performs a logical OR operation (sometimes referred to as an inclusive OR operation) on the contents of the operand 1 register and the full word in operand 2. The result is placed in the operand 1 register.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	O	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	O	$r_1,s_2(x_2)$

When the O instruction is executed, a logical OR operation is performed on a bit in operand 1 and a bit in operand 2. The result of that OR operation replaces the accessed bit in operand 1. This instruction operates from left to right starting with the logical OR operation of bit 0 in both operands up to and including the logical OR operation of bit 31 in both operands.

The O instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	1

When coding patterns are used as operands in logical OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to 1.

After the O instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be either defined as a full word or aligned on a full-word boundary.
- The condition code is set accordingly.

Example:

```
LABEL   ΔOPERATIONΔ                   OPERAND
1        10        16
-------------------------------------------------------------
         L         11,NUMX
         O         11,PATRNO
         .
         .
         .
         DS        ØF
NUMX     DC        X'ØØØØFØFØ'
PATRNO   DC        X'ØØFFØFØF'
```

Register 11 before execution of O instruction:

0000	0000	0000	0000	1111	0000	1111	0000	binary
0	0	0	0	F	0	F	0	hex

PATRNO before and after execution of O instruction:

0000	0000	1111	1111	0000	1111	0000	1111	binary
0	0	F	F	0	F	0	F	hex

Register 11 after execution of O instruction:

0000	0000	1111	1111	1111	1111	1111	1111	binary
0	0	F	F	F	F	F	F	hex

In this example, the content of NUMX is loaded into register 11 and then a logical OR operation is performed on the content of register 11 and the content of PATRNO. The resultant modified binary string replaces register 11.

12.25. OR (OC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
OC	D6	SS	6

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠ 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *or* (OC) instruction performs a logical OR operation (sometimes referred to as an inclusive OR operation) on the contents of operands 1 and 2 located in main storage. The result is placed in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	OC	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	OC	$s_1(l),s_2$

When the OC instruction is executed, a logical OR operation is performed on a bit in operand 1 and a bit in operand 2. The result of the logical OR operation replaces the accessed bit in operand 1. This instruction operates from left to right. The length of operand 1, whether implied or explicit, determines the length of operand 2. Therefore, when the OC instruction is executed, the lengths of operands 1 and 2 are the same.

The OC instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	1

When coding patterns are used as operands in logical OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to 1.

After the OC instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Operands 1 and 2 must be located in main storage.
- The condition code is set accordingly.
- Operands 1 and 2 can have overlapping bytes.

Example:

```
LABEL    △OPERATION△                     OPERAND
1         10        16
---------------------------------------------------------------
          OC        CONSTANT,CONDIT1
          OC        CONSTANT,CONDIT2
          .
          .
          .
CONSTANT  DC        B'ØØØØØØØØ'
CONDIT1   DC        B'ØØØØØØ11'
CONDIT2   DC        B'ØØ1Ø1ØØØ'
```

CONSTANT before execution of OC instruction:

0000	0000	binary
0	0	hex

Before and after execution of OC instruction:

CONDIT1

0000	0011	binary
0	3	hex

CONDIT2

0010	1000	binary
2	8	hex

CONSTANT after execution of first OC instruction:

0000	0011	binary
0	3	hex

CONSTANT after execution of second OC instruction:

0010	1011	binary
2	B	hex

In this example, a logical OR operation is performed on the contents of CONSTANT and CONDIT1 and the result is placed in CONSTANT. Then, another logical OR operation is performed on the contents of CONSTANT (now modified) and CONDIT2. That result replaces the contents of CONSTANT. This is an example of the way programmed switches can be used to set several conditions.

OI

12.26. OR (OI)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
OI	96	SI	4

Condition Codes

- ■ IF RESULT = 0, SET TO 0
- ■ IF RESULT ≠ 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ☐ UNCHANGED

Possible Program Exceptions

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

The *or immediate* (OI) instruction performs a logical OR operation on the contents of operand 1 located in main storage and the one byte of immediate data in operand 2. The result replaces one byte in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	OI	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	OI	s_1,i_2

When the OI instruction is executed, a logical OR instruction is performed on a bit in operand 1 and a bit in operand 2. The result of the logical OR operation replaces the accessed bit in operand 1. This instruction operates from left to right. The length of operand 1 can vary but the length of operand 2 is always one byte. Although operands 1 and 2 may have differing lengths, only one byte in operand 1 is used in the logical OR operation. The result replaces the one byte in operand 1 that was accessed. If you do not specify the exact byte in operand 1 you want used in the logical OR operation, the first byte of operand 1 is used.

The OI instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table.

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	1

When coding patterns are used as operands in logical OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to 1.

After the OI instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Operand 1 must be a main storage location.
- Operand 2 must be a 1-byte self-defining term.
- The length of operand 1 can vary.
- The condition code is set accordingly.
- You can specify the exact byte in operand 1 you want used in the logical OR operation through relative addressing.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1        10         16
------------------------------------------------------------
         AP         AMTP,VAL
         UNPK       AMT(5),AMTP
         OI         AMT+4,X'FØ'
         .
         .
         .
AMTP     DC         PL3'652'
VAL      DC         P'522'
AMT      DS         ZL5
```

AMT before execution of OI instruction:

AMT+4 (arrow points to fifth byte)

1111	0000	1111	0001	1111	0001	1111	0111	1100	0100	binary
F	0	F	1	F	1	F	7	C	4	hex

Operand 2 immediate before and after execution of OI instruction:

1111	0000	binary
F	0	hex

AMT after execution of OI instruction:

1111	0000	1111	0001	1111	0001	1111	0111	1111	0100	binary
F	0	F	1	F	1	F	7	F	4	hex

In this example, the packed decimal contents of AMTP and VAL are added together and the result is placed in AMTP. Then the UNPK instruction changes the packed format of AMTP to the zoned decimal format and puts the result in AMT. In order to print a decimal number, it must be in zoned decimal format and each number must be preceded by a hexadecimal F. Otherwise, an alpha character will be printed as the rightmost byte. Note that the last byte in AMT has a hexadecimal C in its zone portion. The OI instruction allows a logical OR operation to be performed on the contents of byte 5 in AMT and the one byte of data in operand 2. The result replaces byte 5 of AMT. Now the decimal number in AMT can be printed.

12.27. OR (OR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
OR	**16**	**RR**	**2**

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT ≠ 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The *or* (OR) instruction performs a logical OR operation on the contents of the operand 1 register and operand 2 register. The result is placed in the operand 1 register and also determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	OR	r_1,r_2

When the OR instruction is executed, a logical OR operation is performed on a bit in the operand 1 and operand 2 registers. The result of the logical OR operation replaces the accessed bit in operand 1. This instruction operates from left to right starting with the execution of the logical OR operation on bit 0 in both registers up to and including bit 31 in both registers.

The OR instruction is used to modify bits in the receiving field. The procedure is shown in the following truth table:

Operand 1	Operand 2	Result (Operand 1)
0	0	0
1	0	1
0	1	1
1	1	1

When coding patterns are used as operands in logical OR instructions, the following codes are set:

- 0 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to remain the same.
- 1 in all bit positions in operand 2 that correspond to bit positions in operand 1 that you want to set to 1.

After the OR instruction is executed, the condition code is set as follows:

To 0 if result is all 0's.

To 1 if result is a combination of 1's and 0's.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- The condition code is set accordingly.

Example:

```
LABEL   ΔOPERATIONΔ                    OPERAND
1        10       16
------------------------------------------------------------
         L        4,HEX#1
         L        8,HEX#2
         OR       4,8
         .
         .
         .
         DS       ØF
HEX#1    DC       X'ØØØ1Ø1CØ'
HEX#2    DC       X'ØCØAØFF5'
```

Register 4 before execution of OR instruction:

0000	0000	0000	0001	0000	0001	1100	0000	binary
0	0	0	1	0	1	C	0	hex

Register 8 before and after execution of OR instruction:

0000	1100	0000	1010	0000	1111	1111	0101	binary
0	C	0	A	0	F	F	5	hex

Register 4 after execution of OR instruction:

0000	1100	0000	1011	0000	1111	1111	0101	binary
0	C	0	B	0	F	F	5	hex

In this example, the content of HEX#1 is loaded into register 4 and the content of HEX#2 is loaded into register 8. Then a logical OR operation is performed on the contents of registers 4 and 8. The result replaces the content of register 4.

SLDL

12.28. SHIFT LEFT DOUBLE LOGICAL (SLDL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SLDL	8D	RS	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *shift left double logical* (SLDL) instruction shifts all of the 63 bits of operand 1 to the left by the number of bits specified in the low order six bits of the operand 2 address. Specify the even-numbered register of the pair as operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLDL	$r_1, d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLDL	r_1, s_2

Operational Considerations:

- Any pair of general registers (0 through 15) can be used as operand 1, which is an even-odd register pair. You must specify the even-numbered register of the pair as operand 1.
- The main storage address or label you specify in operand 2 is not changed by the SLDL instruction execution. Notice the formats indicate that you cannot specify a length in operand 2.
- After the requested number of bits are shifted out of the operand 1 register pair, zeros fill the rightmost bit positions of the register pair.

Example 1:

```
LABEL     ΔOPERATIONΔ                 OPERAND
1          10       16
------------------------------------------------------------
           LM       4,5,VALUE
           SLDL     4,32
           .
           .
           .
           DS       ØF
VALUE      DC       X'FFFFFFFF'
           DC       X'FFFFFFFF'
```

In this example, register 4 is loaded with the content of main storage location VALUE. Register 5 is loaded with the next 32 bits of main storage following VALUE. The SLDL instruction causes the contents of the registers to be shifted left 32 bits.

Registers 4 and 5 before execution of SLDL instruction:

Register 4								Register 5								
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	hex
1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	binary

Registers 4 and 5 after execution of SLDL instruction:

F	F	F	F	F	F	F	F	0	0	0	0	0	0	0	0	hex
1111	1111	1111	1111	1111	1111	1111	1111	0000	0000	0000	0000	0000	0000	0000	0000	binary

Example 2:

This example using the SLDL has the same shift result except that a label is used as operand 2 of the SLDL instruction.

```
LABEL     ΔOPERATIONΔ                          OPERAND
1          10        16
--------------------------------------------------------------
           LM        4,5,VALUE
           SLDL      4,STORAGE
           .
           .
           .
           DS        ØF
VALUE      DC        2XL4'FFFFFFFF'
STORAGE    EQU       X'2Ø'
```

In this example, the absolute value of the main storage location STORAGE determines the number of bits that operand 1 should be shifted left. The operand 2 label is equated (EQU instruction) with a hexadecimal value that indicates how many bit positions you want to move. This is necessary so that the assembler takes the absolute value that you equated with your label and not its relocatable address. Note that a hexadecimal 20 is equivalent to a decimal 32.

Registers 4 and 5 before execution of SLDL instruction:

F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	hex
1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	binary

STORAGE before and after SLDL instruction execution:

2	0	hex
0010	0000	binary

2^5(32)

Registers 4 and 5 after SLDL instruction execution:

F	F	F	F	F	F	F	F	0	0	0	0	0	0	0	0	hex
1111	1111	1111	1111	1111	1111	1111	1111	0000	0000	0000	0000	0000	0000	0000	0000	binary

12.29. SHIFT LEFT SINGLE LOGICAL (SLL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SLL	89	RS	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *shift left single logical* (SLL) instruction shifts all of the 32 bits in the operand 1 register to the left by the number of bits specified in the low order six bits of the operand 2 address.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLL	$r_1, d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLL	r_1, s_2

The operand 2 address is not used to address data. The low order six bits are used as the shift count and the remainder of the address is ignored. When the SLL instruction is executed, the high order bits that are shifted out of the register are lost and replaced with subsequent bits within the register also being shifted. Zeros fill the vacated low order bit positions.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- If you already know the displacement value associated with a label in your program, you can use the low order six bits as the shift count.
- A self-defining term can be specified for operand 2.
- Zeros fill the vacated low order bit positions in the operand 1 register.
- The condition code remains unchanged.
- A length attribute cannot be specified for operand 2.
- The shift count cannot exceed 32 bits because the highest value that can be represented in the low order six bits is +32 (2^5).

Example:

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10        16
------------------------------------------------------------
          L         4,FLWRD
          SLL       4,STORAGE
          .
          .
          .
          DS        ØF
FLWRD     DC        X'89ABCDEF'
STORAGE   EQU       8
```

Register 4 before execution of SLL instruction:

1000	1001	1010	1011	1100	1101	1110	1111	binary
8	9	A	B	C	D	E	F	hex

STORAGE before and after execution of SLL instruction:

low order six bits

0000	0000	0000	0000	0000	1000	binary
0	0	0	0	0	8	hex

location counter (address) not contents

Register 4 after execution of SLL instruction:

1010	1011	1100	1101	1110	1111	0000	0000	binary
A	B	C	D	E	F	0	0	hex

In this example, the content of FLWRD is loaded into register 4. Then the SLL instruction uses the low order six bits of the address of STORAGE (not content) as the shift count. In this case, STORAGE has no content but has been equated with or contains the address of the absolute value of 8. When the SLL instruction is executed, eight high order bits of register 4 are shifted out of the register, and replaced with subsequent bits within the register also being shifted. Zeros fill the vacated eight low order bit positions.

Example:

```
LABEL   ΔOPERATIONΔ                          OPERAND
1       10        16
--------------------------------------------------------
         SR        4,4
         L         4,=F'25ØØ'
         L         5,=F'1Ø'
         SLL       4,Ø(5)
         .
         .
         .
```

Register 4 before execution of SLL instruction:

0000	0000	0000	0000	0000	1001	1100	0100	binary
0	0	0	0	0	9	C	4	hex

Register 5 before and after execution of SLL instruction:

0000	0000	0000	0000	0000	0000	0000	1010	binary
0	0	0	0	0	0	0	A	hex

actual contents

Register 4 after execution of SLL instruction:

0000	0000	0010	0111	0001	0000	0000	0000	binary
0	0	2	7	1	0	0	0	hex

In this example, register 4 is cleared to 0. The decimal value of 2500 is loaded into register 4, and the decimal value of 10 is loaded into register 5. The actual content of register 5 is used (not its address) because register 5 is being used as a base register. So, the value loaded into register 5 is treated as an address. This can only be done in the explicit format.

12.30. SHIFT LOGICAL (SHL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SHL	9B	RS	4

Condition Codes

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

Possible Program Exceptions

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

The *shift logical* (SHL) instruction shifts all 32 bits in the operand 1 register or all 64 bits in the operand 1 even-odd register pair right or left by the number of bits specified in the low order six bits of the operand 2 address.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SHL	$r_1,m_3,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SHL	r_1,m_3,s_2

The SHL instruction is a versatile shift instruction that can operate in a number of different ways. With it you can:

- shift the contents of the operand 1 register left or right up to 63 bits per operation;
- fill the vacated bits of the operand 1 register with 1's or 0's at your option;
- shift a single register or shift two contiguous registers; and
- allow bits shifted out of operand 1 to be lost or move them into the opposite end of operand 1, a process known as *circular* shifting.

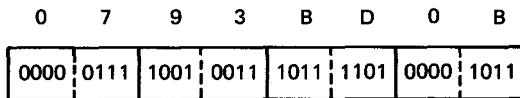
When an SHL instruction is executed, the low order six bits of operand 2 determine the number of bits to be shifted. This permits you to specify a maximum shift of 63 bits ($3F_{16}$).

You use the 4-bit operand 3 mask (m_3) to further specify the shift. The mask specifies 4 options so that you can use the instruction in 1 of 16 different ways. Individual bits within the mask are used as shown in Table 12—1:

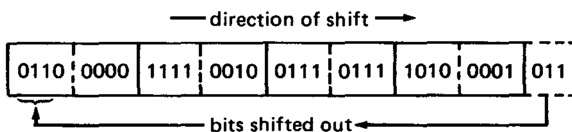
Table 12—1. Shift Logical Mask Bits

Object Code Bit Position				Mask Value (Decimal)
12	13	14	15	
Vacated Bits 0=Lost 1=Circular	Shift Direction 0=Left 1=Right	Register 0=Single 1=Even/Odd Pair	Digit Shifted In 0=0 1=1	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

A circular shift (bit 12 = 1) moves bits shifted out of one end of operand 1 into the opposite end, one bit at a time. Assume that register 11 contains 0793BD0B:

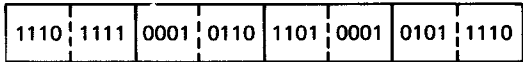


Now register 11 undergoes a circular shift of three bits to the right:



The three bits shifted out of the low order end of register 11 are placed, their sequence unchanged, into the high order bit positions just vacated by the instruction.

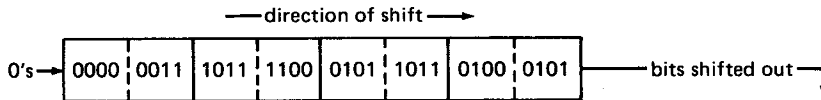
You can use the SHL instruction to specify which binary digit is to be shifted into the operand 1 register to replace vacated bit positions. Specifying a 1 in bit 15 causes 1's to be shifted in. Specifying a zero in bit 15 causes 0's to be shifted in. If register 8 contains



the instruction

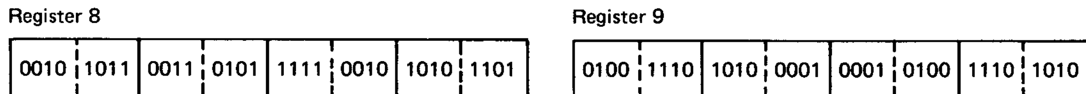
```
SHL 8,4,6
```

specifies a 6-bit shift to the right with 0's shifted in to the left or high order end (see Table 12—1):

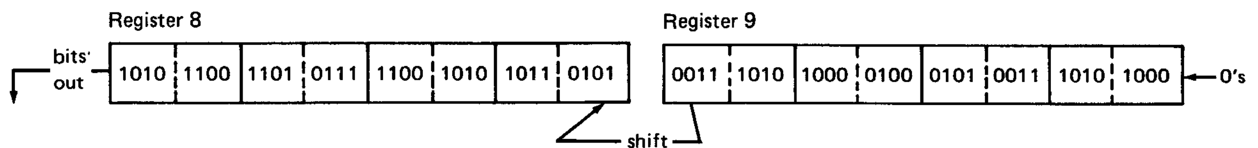


The SHL instruction can shift two contiguous registers under all the conditions described. The first register must be an even-numbered register, the second the next higher odd-numbered register. For double-register operation, all 64 bits of the pair are shifted:

- for left shifts each bit shifted out of the high order end of the odd-numbered register is put in the low order end of the even-numbered register;
- for right shifts, bits shifted out of the low order end of the even-numbered register are put in the high order end of the odd-numbered register. Assume that the register pair 8 and 9 contains:



A 2-bit left shift with 0's moved in the vacated bits yields:



Operational Considerations:

- You can specify operand 2 as an address ($d_2(b_2)$ form only) or as a symbol equated (EQU instruction) to an absolute expression. You should not use an operand 2 symbol (s_2) as an address because you may get unpredictable results.
- You can specify a single register in operand 1 using any of the general-purpose registers 0 through 15.
- You can specify a register pair in operand 1, in which case the operand must be the even-numbered register, the first register of the pair.
- You must specify the operand 3 mask as a self-defining term.
- During circular shifts bit 15 (the bit to be shifted in) is ignored.

Condition Code:

After execution of SHL, the condition code is set:

- to 0 if operand 1 contains 0 and the bits shifted out are all 0's;
- to 1 if operand 1 contains 0 and one or more 1's are shifted out;
- to 2 if operand 1 is not zero and the bits shifted out are all 0's; or
- to 3 if operand 1 is not zero and one or more 1's are shifted out.

Example:

```
      LABEL     ΔOPERATIONΔ                      OPERAND
      1         10        16
 1.             LA        9,Ø                    CLEAR R9
 2.             SHL       8,6,4                  SHIFT 1ST DIGIT
 3.             SHL       9,5,4                  ZONE
 4.             SHL       8,6,4                  SHIFT 2ND DIGIT
 5.             SHL       9,5,4                  ZONE
 6.             SHL       8,6,4                  SHIFT 3RD DIGIT
 7.             SHL       9,5,4                  ZONE
 8.             SHL       8,6,4                  SHIFT 4TH DIGIT
 9.             SHL       9,5,4                  ZONE
10.             LR        11,9                   MOVE LOW-ORDER DIGITS TO R11
11.             LA        9,Ø                    CLEAR R9
12.             SHL       8,6,4                  SHIFT 5TH DIGIT
13.             SHL       9,5,4                  ZONE
14.             SHL       8,6,4                  SHIFT 6TH DIGIT
15.             SHL       9,5,4                  ZONE
16.             SHL       8,6,4                  SHIFT 7TH DIGIT
17.             SHL       9,5,4                  ZONE
18.             SHL       8,6,4                  SHIFT 8TH DIGIT
19.             SHL       9,5,4                  ZONE
20.             LR        1Ø,9                   MOVE HIGH-ORDER DIGITS TO R1Ø
21.             SHL       1Ø,14,8                SHIFT SIGN AROUND TO TOP OF R1Ø
22.             STM       1Ø,11,EBCNO            STORE NUMBER IN EBCNO
23.             CLI       EBCNO,X'FD'            TEST SIGN
24.             BNE       POS                      IF POSITIVE GO TO POS
25.             MVI       EBCNO,C'-'               OTHERWISE ATTACH '-'
26.             B         NEG
27.   POS       MVI       EBCNO,C''                ATTACH SPACE FOR POSITIVE
28.   NEG        .
                 .
                 .
      EBCNO     DS        2F                     EBCDIC RESULT
```

The code in this example expands a packed decimal number in register 8 to an EBCDIC number of 7 digits plus leading sign in field EBCNO.

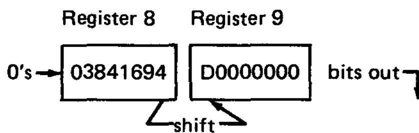
Assume that register 8 contains

Register 8 | 3841694D |

The LA instruction in line 1 puts 0's in register 9. The SHL instruction in line 2 has a mask of 6 that is interpreted as follows (see Table 12—1):

Mask 6	0	1	1	0
	lose vacated bits	shift right	shift register pair	shift in 0's

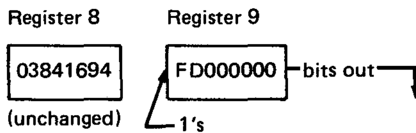
Together with the rest of the operands, the SHL instruction specifies that register pair 8 and 9 is to be shifted 4 bits to the right. Doing this results in:



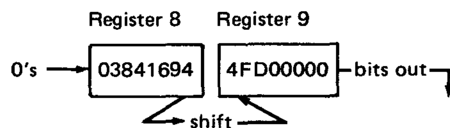
(Since a hexadecimal digit comprises four bits, a 4-bit shift is equivalent to a shift of a single hexadecimal digit.) The SHL instruction in line 3 has a mask of 5 that is interpreted as follows:

Mask 5	0	1	0	1
	lose vacated bits	shift right	shift single register	shift in 1's

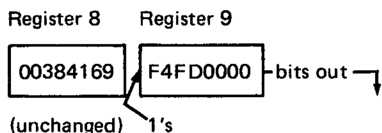
As a result of this instruction, the contents of register 9 are shifted four bits to the right, and 1's are shifted in from the high order end:



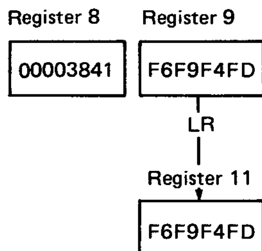
Notice that the four "1" bits shifted into the high order end of register 9 form, in effect, the zone half of an EBCDIC digit. The SHL instruction in line 4 shifts another packed digit into the high order end of register 9:



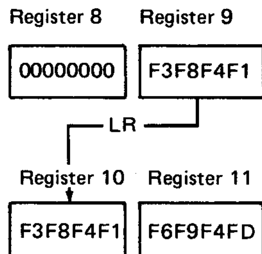
The SHL instruction of line 5, like that of line 3, shifts 1's into register 9; this operation attaches a zone to the digit just shifted from register 8:



Lines 6 through 9 repeat this process for two more packed digits, and line 10 moves the contents of register 9 to register 11 for later processing:



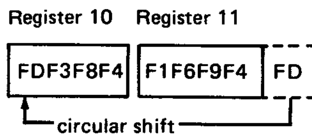
Line 11 fills register 9 with 0's and lines 12 through 19 repeat the action of lines 2 through 9, but this time on the high order four digits orignally in register 8. After line 20, which moves the contents of register 9 to register 10, the result is:



The SHL instruction in line 21 has a mask of 14 (hexadecimal E) that is interpreted as follows:

Mask E_{16}	1	1	1	0
	circular shift	shift right	shift register pair	not used

The instruction shifts registers 10 and 11 right by 8 bits, moving the shifted bits out of register 11 directly back into the high order end of register 10:



In line 22, the contents of registers 10 and 11 are stored into a double-word area in storage named EBCNO. Lines 23 through 28 test the leading byte: if it contains 'FD' it is replaced with a minus sign to indicate a negative number; otherwise, the byte is replaced with a blank to indicate a positive number. Since the leading byte of EBCNO is 'FD', EBCNO ends up as:

EBCNO	60F3F8F4	F1F6F9F4
	– 3 8 4	1 6 9 4

The number in EBCNO can now be printed if desired.

SRDL

12.31. SHIFT RIGHT DOUBLE LOGICAL (SRDL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SRDL	**8C**	**RS**	**4**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *shift right double logical* (SRDL) instruction shifts the 64 bits of operand 1 to the right by the number of bits specified in the low order six bits of the operand 2 address. You specify the even-numbered register of the pair as operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRDL	$r_1, d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRDL	r_1, s_2

Operational Considerations:

- Any pair of the general registers (0 through 15) can be used as operand 1. Operand 1 is an even-odd register pair. You must specify the even-numbered register of the pair as operand 1.
- The main storage address or label you specify in operand 2 is not changed by the SRDL instruction execution. Notice the formats indicate that you cannot specify a length in operand 2.
- After the requested number of bits are shifted out of the operand 1 register pair, 0's fill the leftmost bit positions of the register pair.

Example 1:

```
LABEL     ΔOPERATIONΔ                        OPERAND
1          10        16
-------------------------------------------------------------
           LM        4,5,VALUE
           SRDL      4,32
           .
           .
           .
           DS        ØF
VALUE      DC        2XL4,'FFFFFFFF'
```

In this example, registers 4 and 5 are loaded with the content of main storage location VALUE. The SRDL instruction causes the content of a register to be shifted right 32 bits.

Registers 4 and 5 before execution of SRDL instruction:

F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	hex
1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	binary

Registers 4 and 5 after SRDL instruction execution:

0	0	0	0	0	0	0	0	F	F	F	F	F	F	F	F	hex
0000	0000	0000	0000	0000	0000	0000	0000	1111	1111	1111	1111	1111	1111	1111	1111	binary

Example 2:

This example using the SRDL instruction has the same shift result, but a label is used as operand 2 of the SRDL instruction.

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10        16
-----------------------------------------------
          LM        4,5,VALUE
          SRDL      4,STORAGE
          .
          .
          .
          DS        ØF
VALUE     DC        2XL4'FFFFFFFF'
STORAGE   EQU       X'2Ø'
```

In this example, the absolute value of the main storage location STORAGE determines the number of bits that operand 1 should be shifted right. The operand 2 label is equated (EQU instruction) with a hexadecimal value that indicates how many bit positions you want to move. This is necessary so that the assembler takes the absolute value that you equated with your label and not its relocatable address. Note that a hexadecimal 20 is equivalent to a decimal 32.

Registers 4 and 5 before execution of SRDL instruction:

F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	hex
1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	binary

STORAGE before and after SRDL instruction execution:

2	0	hex
0010	0000	binary

2^5(32)

Registers 4 and 5 after instruction execution:

0	0	0	0	0	0	0	0	F	F	F	F	F	F	F	F	hex
0000	0000	0000	0000	0000	0000	0000	0000	1111	1111	1111	1111	1111	1111	1111	1111	binary

SRL

12.32. SHIFT RIGHT SINGLE LOGICAL (SRL)

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
SRL	88	RS	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

The *shift right single logical* (SRL) instruction shifts all of the 32 bits in the operand 1 register to the right by the number of bits specified in the low order six bits of the operand 2 address.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRL	$r_1,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SRL	r_1,s_2

The operand 2 address is not used to address data. The low order six bits are used as the shift count and the remainder of the address is ignored. When the SRL instruction is executed, the low order bits that are shifted out of the register are lost and replaced with subsequent bits within the register also being shifted. Zeros fill the vacated high order bit positions.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Using a label as operand 2 can cause unpredictable results.
- A self-defining term can be used as operand 2.
- Zeros fill the vacated high order bit positions in the operand 1 register.
- The condition code remains unchanged.
- A length attribute cannot be specified for operand 2.
- The shift count cannot exceed 32 bits because the highest value that can be represented in the low order six bits is +32 (2^5).

Example:

```
LABEL    △OPERATION△                        OPERAND
1         10        16
------------------------------------------------------------
          L         4,WORD
          SRL       4,LABEL
          .
          .
          .
          DS        ØF
WORD      DC        X'3A6ØØØØØ'
LABEL     DC        PL2'Ø'
```

Register 4 before execution of SRL instruction:

0011	1010	0110	0000	0000	0000	0000	0000	binary
3	A	6	0	0	0	0	0	hex

Address of LABEL:

low order six bits

0000	0000	0000	0000	0000	0000	0001	0100	binary
0	0	0	0	0	0	1	4	hex

location counter

Register 4 after execution of SRL instruction:

0000	0000	0000	0000	0000	0011	1010	0110	binary
0	0	0	0	0	3	A	6	hex

In this example, the content of WORD is loaded into register 4. Then the SRL instruction uses the low order six bits of the address location counter of LABEL as the shift count. You should already know the displacement value of LABEL. That value should be the number of bits you want to shift to the right. When the SRL instruction is executed, 20 low order bits of register 4 are shifted out of the register, and replaced with subsequent bits within the register also being shifted. Zeros fill the vacated 20 high order bits. This is what LABEL looks like when assembled:

```
LOC.   OBJECT CODE                                LINE  SOURCE STATEMENT

000014 000C                                       14 LABEL    DC     PL2'0'
```

12.33. STORE CHARACTER (STC)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STC	42	RX	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *store character* (STC) instruction places the low order eight bits (bit positions 24 through 31) of the operand 1 register into one byte of operand 2 that is located in main storage.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STC	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STC	$r_1,s_2(x_2)$

The data in operand 2 can be defined in any format. The length of operand 2 can vary but the length of operand 1 is always four bytes. Although operands 1 and 2 can have the same or differing length, only one byte in operand 2 receives the rightmost byte from the operand 1 register. If you do not specify the exact byte in operand 2 to receive the operand 1 data, the first byte of operand 2 is used.

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- Operand 2 must be a main storage location.
- The length of operand 2 can vary.
- You can specify the exact byte in operand 2 to receive the rightmost byte of the operand 1 register through relative addressing.
- This instruction is one of the few in which operand 1 is the sending operand.

Example:

```
LABEL   ΔOPERATIONΔ                   OPERAND
1        10        16
--------------------------------------------------
         L         11,AMTIN
         A         11,=F'5'
         STC       11,STOR
         .
         .
         .
AMTIN    DC        F'2236'
STOR     DC        CL4'Ø'
```

STOR before execution of STC instruction:

1111	0000	0100	0000	0100	0000	0100	0000	binary
F	0	4	0	4	0	4	0	hex

Register 11 before and after execution of STC instruction:

0000	0000	0000	0000	0000	1000	1100	0001	binary
0	0	0	0	0	8	C	1	hex

STOR after execution of STC instruction:

1100	0001	0100	0000	0100	0000	0100	0000	binary
C	1	4	0	4	0	4	0	hex

In this example, the content of AMTIN is loaded into register 11. Then a decimal value of 5 is added to the value already in register 11. This is a method of rounding numbers located in registers that will eventually be printed. The STC instruction then places the content of the rightmost byte in register 11 into the first byte of STOR. Since the exact byte of STOR that is to receive the data from register 11 is not specified, the first byte is used.

STCM

12.34. STORE CHARACTERS UNDER MASK (STCM)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STCM	**BE**	**RS**	**4**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *store characters under mask* (STCM) instruction permits you to store some or all of the contents of the operand 1 register in contiguous bytes of main storage starting at the operand 2 address. You select the bytes to be stored by specifying a 4-bit mask in operand 3 (m_3).

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STCM	$r_1,m_3,d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STCM	r_1,m_3,s_2

Operand 3 specifies a 4-bit mask, each bit of which corresponds to a byte within the operand 1 register, going left to right. The value of the mask corresponds to its bits as follows:

Mask value	8	4	2	1
Object code bit	12	13	14	15
Register byte (left to right)	1	2	3	4

Thus, a mask of 5 (4+1) specifies the second and fourth register bytes. Each bit thus specified gets a value of 1, while remaining mask bits get a value of 0. In operation, the STCM instruction scans the mask left to right. The first register byte whose corresponding mask bit equals 1 is stored in the main storage byte located at operand 2. The next register byte whose mask bit equals 1 is stored in the byte location following the operand 2 address. This process continues until all mask bits are scanned. Register bytes whose mask bits equal 0 are not stored. The number of bytes in the operand 2 field equals the number of 1-bits in the mask, up to a limit of four bytes.

Operational Considerations:

- The operand 3 mask must be a self-defining term ranging from 0 to 15.
- With a mask of 15 (1111_2) the STCM instruction acts much like a STORE instruction; the only difference is that the STCM operand 2 need not reside on a full-word boundary.
- The STCM instruction is one of the few instructions whose source operand precedes its destination operand.

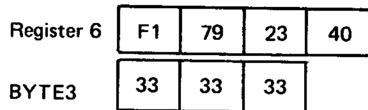
Condition Code:

The condition code is left unchanged by the STCM instruction.

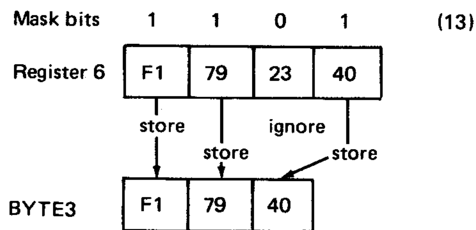
Example:

```
LABEL     ΔOPERATIONΔ                          OPERAND
1          10          16
---------------------------------------------------------------
           L           6,FWORD
           STCM        6,13,BYTE3
           .
           .
           DS          ØF
FWORD      DC          XL4'F17923400'
BYTE3      DC          XL3'333333'
```

In this example, register 6 is partially stored into a 3-byte area named BYTE3. At the start of the STCM instruction, the data areas contain:



The STCM instruction has a mask of 13 which causes the instruction to operate as follows:



The STCM mask of 13 causes the first byte of register 6 to be stored at BYTE3, the second byte of register 6 to be stored at BYTE3+1, the third byte of register 6 to be ignored, and the fourth byte of register 6 to be stored at BYTE3+2.

12.35. SUBTRACT LOGICAL (SL)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SL	5F	RX	4

Condition Codes

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

SEE OPER. CONSIDERATIONS

Possible Program Exceptions

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

The *subtract logical* (SL) instruction logically subtracts the content of operand 2 from the content of the operand 1 register and places the result in operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SL	$r_1,d_2(x_2,b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SL	$r_1, s_2(x_2)$

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operand 1.
- The label or address you specify in operand 2 must refer to a main storage location that is on a full-word boundary. Operand 2 is not changed by the execution of this instruction.
- The logical subtraction is performed by adding the twos complement of operand 2 to operand 1. All 32 bits of each operand are used.
- Neither operand has a sign bit.
- The condition code of the program status word (PSW) is set as follows:
  - to 1 if result is not 0 (no carry of most significant bit);
  - to 2 if result is 0 (carry of most significant bit); or
  - to 3 if result is not 0 (carry of most significant bit).

  Zero code is not used.

Example:

```
LABEL     ΔOPERATIONΔ                     OPERAND
1          10       16
-------------------------------------------------------
           L        3,HEXVALU
           SL       3,FULLWORD
           .
           .
           .
           DS       ØF
HEXVALU    DC       X'ØØØØØFF8'
FULLWORD   DC       X'ØØØØØEØ8'
```

In this example, the hexadecimal content of HEXVALU is placed in register 3. Then the twos complement of the content of main storage location FULLWORD is added to the content of register 3.

FULLWORD before twos complement:

0	0	0	0	0	E	0	8	hex
0000	0000	0000	0000	0000	1110	0000	1000	binary

Twos complement of FULLWORD:

1111	1111	1111	1111	1111	0001	1111	1000	binary
F	F	F	F	F	1	F	8	hex

Register 3 before execution of SL instruction:

0	0	0	0	0	F	F	8	hex
0000	0000	0000	0000	0000	1111	1111	1000	binary

Register 3 after execution of SL instruction:

0	0	0	0	0	1	F	0	hex
0000	0000	0000	0000	0000	0001	1111	0000	binary

The twos complement of FULLWORD is added to the content of register 3. The result replaces the content of register 3. The condition code is set to 3, since the result is not 0, and there is carryout (leftover 1 bit) of the leftmost bit. The carryout does not cause an overflow condition as would the subtract (S) instruction.

SLR

12.36. SUBTRACT LOGICAL (SLR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SLR	1F	RR	2

Condition Codes

- ☐ SET TO 0
- ■ SET TO 1
- ■ SET TO 2
- ■ SET TO 3
- SEE OPER. CONSIDERATIONS

Possible Program Exceptions

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

The *subtract logical* (SLR) instruction logically subtracts the content of the operand 2 register from the content of the operand 1 register and places the result in operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLR	r_1,r_2

Operational Considerations:

- Any of the general registers (0 through 15) can be used as operands 1 and 2.
- The logical subtraction is performed by adding the twos complement of operand 2 to operand 1. All 32 bits of each operand are used.
- Neither operand has a sign bit.
- The condition code of the program status word (PSW) is set as follows:
  - to 1 if result is not 0 (no carry of most significant bit);
  - to 2 if result is 0 (carry of most significant bit); or
  - to 3 if result is not 0 (carry of most significant bit).

  Zero code is not used.

Example:

```
LABEL     ΔOPERATIONΔ                        OPERAND
1          10        16
_____________________________________________________
           L         3,VALU1
           L         5,VALU2
           SLR       3,5
           .
           .
           .
           DS        ØF
VALU1      DC        X'ØØØØØFF8'
VALU2      DC        X'ØØØØØEØ8'
```

In this example, the hexadecimal contents of main storage locations VALU1 and VALU2 are loaded into registers 3 and 5, respectively. Then, the SLR instruction logically subtracts the content of register 5 from the content of register 3.

Register 5 before twos complement:

0	0	0	0	0	E	0	8	hex
0000	0000	0000	0000	0000	1110	0000	1000	binary

Twos complement of register 5:

1111	1111	1111	1111	1111	0001	1111	1000	binary
F	F	F	F	F	1	F	8	hex

Register 3 before execution of SLR instruction:

0	0	0	0	0	F	F	8	hex
0000	0000	0000	0000	0000	1111	1111	1000	binary

Register 3 after execution of SLR instruction:

0	0	0	0	0	1	F	0	hex
0000	0000	0000	0000	0000	0001	1111	0000	binary

The twos complement of register 5 is added to the content of register 3. The result replaces the content of register 3. The condition code is set to 3, since the result is not zero and there is a carryout (leftover 1 bit) of the leftmost bit. The carryout does not cause an overflow condition as would the subtract (SR) instruction.

12.37. TEST UNDER MASK (TM)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
TM	91	SI	4

Condition Codes
■ SET TO 0
■ SET TO 1
□ SET TO 2
■ SET TO 3
SEE OPER. CONSIDERATIONS

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

The *test under mask* (TM) instruction uses the 1-byte mask in operand 2 to test the bit pattern of one byte in operand 1 that is located in main storage. The result of the test determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TM	$d_1(b_1),i_2$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TM	s_1,i_2

Operand 2 is one byte of immediate data that is used as the mask. Within that byte is an 8-bit binary testing pattern. The bits of the mask correspond one for one to bits in one byte of operand 1. The length of operand 1 can vary but the length of operand 2 is always one byte. Although the lengths of operands 1 and 2 can differ, only one byte of operand 1 is used. If you do not specify the exact byte of operand 1 you want tested, the first byte is used.

A mask bit of one in operand 2 tests its corresponding bit position in operand 1 for the presence of one bit or a zero bit.

A mask bit of 0 in operand 2 causes its corresponding bit position in operand 1 not to be tested. If the mask pattern is all 0's, no testing takes place; whereas, if the mask pattern is all 1's, every corresponding bit position in operand 1 is tested.

The condition code is set as follows:

- to 0 if the mask pattern is all 0's; or all the corresponding tested bit positions in operand 1 are 0's;
- to 1 if all the corresponding tested bit positions in operand 1 are a combination of 0's and 1's; or
- to 3 if all the corresponding tested bit positions in operand 1 are 1's.

Some of the more common uses of the test under mask instruction are checking the setting of program switches and checking for valid characters. These switches are usually set by one of the logical OR instructions. After the switches are set, the TM instruction uses the mask to test the bit pattern in operand 1 and set the condition code. Then the resulting condition code can be used to alter the processing sequence of a program by using one of these branch instructions:

Mnemonic Code		Meaning	Remarks	Branches on Condition Code
RR-Type Instruction	RX-Type Instruction			
BOR	BO	Branch if 1's.	The branch is taken if all the bits tested are on.	3
BMR	BM	Branch if mixed.	The branch is taken if some of the bits tested are on, some off.	1
BZR	BZ	Branch if 0's.	The branch is taken if all of the bits are off, or the mask is 0.	0
BNOR	BNO	Branch if not all 1's.	The branch is taken if at least one of the bits tested is not on.	0,1
BNZR	BNZ	Branch if not all 0's.	The branch is taken if at least one of the bits tested is not off.	1,3
BNMR	BNM	Branch if not mixed.	The branch is taken if all the bits tested are off or if all are on.	0,3

Operational Considerations:

- Operand 1 must be a main storage location.
- Operand 2 is the mask and is a 1-byte self-defining term.
- The condition code is set accordingly but condition code 2 is not used.
- The length of operand 1 can vary but a length attribute cannot be specified.

- You can specify the exact byte in operand 1 you want the operand 2 mask to test through relative addressing.
- The contents of operand 1 and operand 2 remain unchanged after the execution of the TM instruction.

Example:

```
LABEL    △OPERATION△                          OPERAND
1         10        16
----------------------------------------------------------------
          LA        5,PAYHRS
          TM        Ø(5),X'FØ'
          BNO       ERROR1
          TM        1(5),X'FØ'
          BNO       ERROR2
          .
          .
          .
ERROR1    MVC       OUTPUT(2Ø),MSG1
          .
          .
          .
ERROR2    MVC       OUTPUT(21),MSG2
          .
          .
          .
PAYHRS    DC        CL2'4A'
          DS        ØH
OUTPUT    DS        CL132
MSG1      DC        CL2Ø'FIRST NUMBER INVALID'
MSG2      DC        CL21'SECOND NUMBER INVALID'
```

PAYHRS before and after execution of first and second TM instruction:

1111	0100	1010	0001	binary
F	4	C	1	hex

Operand 2 mask before and after execution of first and second TM instruction:

1111	0000	binary
F	0	hex

In this example, the address of PAYHRS is loaded into register 5. The TM instruction then uses the mask in operand 2 to test the first byte in operand 1. The four high order bits of the mask test the four high order bits of operand 1. (The low order four bits of the mask cause no testing of the low order four bits of operand 1 to take place.) Since the result of the test is all 1's, the condition code is set to 3 and no branch takes place.

The second TM instruction uses the mask in operand 2 to test the second byte in operand 1. The four high order bits of the mask test the four high order bits of the second byte in operand 1. (The low order four bits of the mask cause no testing of the low order four bits of operand 1 to take place.) Since the result of the test is a combination of 0's and 1's, the condition code is set to 1 and the branch to the instruction labeled ERROR2 takes place. There, an error message is moved to the output area for printing.

12.38. TEST UNDER MASK AND SKIP (TMS)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
TMS	E2	SM	6

Condition Codes

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

Possible Program Exceptions

- ■ ADDRESSING
- ☐ DATA (INVALID SIGN/DIGIT)
- ☐ DECIMAL DIVIDE
- ☐ DECIMAL OVERFLOW
- ☐ EXECUTE
- ☐ EXPONENT OVERFLOW
- ☐ EXPONENT UNDERFLOW
- ☐ FIXED-POINT DIVIDE
- ☐ FIXED-POINT OVERFLOW
- ☐ FLOATING-POINT DIVIDE
- ☐ OPERATION
- ■ PROTECTION
- ■ SIGNIFICANCE
- ☐ SPECIFICATION:
  - ☐ NOT A FLOATING-POINT REGISTER
  - ☐ OP 1 NOT ON HALF-WORD BOUNDARY
  - ☐ OP 2 NOT ON HALF-WORD BOUNDARY
  - ☐ OP 2 NOT ON FULL-WORD BOUNDARY
  - ☐ OP 2 NOT ON DOUBLE-WORD BOUNDARY
  - ☐ OP 1 NOT EVEN NUMBERED REGISTER
  - ☐ OP 1 NOT ODD NUMBERED REGISTER
  - ■ LOW-ORDER BIT OF OP 4 MUST BE ZERO
- ☐ NONE

The *test under mask and skip* (TMS) instruction tests selected bits of a byte of main storage specified by operand 1 and sets the condition code. Together with a 4-bit mask specified by operand 3, the condition code determines whether program control then passes to the next sequential instruction or to another location determined by a displacement specified in operand 4.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TMS	$d_1(b_1),i_2,m_3,d_4$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TMS	s_1,i_2,m_3,d_4

Operand 2 is an 8-bit mask which corresponds, bit for bit, with the 8 bits at the operand 1 location in main storage. The TMS instruction scans the mask and, depending on the setting of its bits, takes the following action with each bit:

- If a mask bit is 0, its corresponding operand 1 bit is ignored.
- If a mask bit is 1, its corresponding operand 1 bit is selected and tested to determine if it is set to 1.

After the mask scan is finished, the condition code is set:

- to 0 if the mask consists of all 0's or if all selected bits are set to 0;
- to 1 if some selected bits are set to 0 and some to 1; or
- to 3 if all selected bits are set to 1.

After the condition code is set, it is compared against the 4-bit mask specified by operand 3:

Mask value	8	4	2	1
Mask bit position	16	17	18	19
Corresponding condition code	0	1	2	3

The TMS instruction then takes one of the following two actions:

- If the mask bit corresponding to the condition code is set to 0, program control passes to the next sequential instruction.
- If the mask bit corresponding to the condition code is set to 1, the TSM instruction adds the 12-bit displacement in bits 20—31 of the object code to the current program status word (PSW). This, in effect, causes a branch to the new PSW address.

The TMS instruction acts like a sequence of TEST UNDER MASK and BRANCH ON CONDITION instructions. The only difference is that a TMS branch, when called for, is really a displacement from the current PSW; the normal address-formation routines play no role in this part of the instruction. Branching can take place forwards or backwards. A 1 in bit 20 (the high order displacement bit) means a negative displacement and a backward branch, while a zero in bit 20 means a positive displacement and a forward branch.

You can specify the m_3 mask either with an absolute value or by coding an extended mnemonic in place of TMS in your assembler source program. The assembler provides six of these mnemonics for the TMS instruction: all assemble into TMS object instructions but each generates a different m_3 mask value as shown in the following table.

Extended Mnemonic	Mnemonic Code	Mask Value*	Function
TMBO	E2	1	Branch if all ones.
TMBZ	E2	8	Branch if all zeros.
TMBM	E2	4	Branch if mixed.
TMBNO	E2	E	Branch if not all ones.
TMBNZ	E2	7	Branch if not all zeros.
TMBNM	E2	B	Branch if not mixed.

*Bits 16—19 of object instruction

When using an extended mnemonic, you use only three operands: the operand 1 address, the operand 2 immediate byte, and the operand 4 displacement, in that order.

You can specify the displacement value yourself or you can let the assembler do it for you:

- You can code the displacement as an absolute expression. In this case, the assembler inserts the expression, unchanged, into the displacement field.
- You can code the displacement as a relocatable expression. In this case, the assembler simulates a branch to the specified location by calculating its offset from the address of the instruction immediately following TMS and inserting the offset in the displacement field.

Operational Considerations:

- The displacement field can range from —2048 decimal bytes to +2046 bytes.
- The displacement must always be an even number of bytes since destination instructions must always lie on a half-word boundary.
- A mask of 0 (0000_{16}) causes the instruction always to branch to the next sequential instruction regardless of the condition code set. Likewise, a displacement value of 0 causes an unconditional branch to the next sequential instruction.
- A mask of 15 (1111_{16}) causes the TMS instruction always to branch to the instruction specified by the displacement field.
- You must specify both the mask and the immediate byte as self-defining terms.

Example:

```
LABEL     ΔOPERATIONΔ                 OPERAND                   OBJECT ADDRESS
1         10         16
---------------------------------------------------------------------------------
          MVI        TESTBYTE,X'F4'                             ØØØ11C
          TMS        TESTBYTE,X'FØ',1,NUM                       ØØØ12Ø
NEXTINS   L          7,Ø                                        ØØØ126

NUM        .                                                    ØØØ142
           .
           .
```

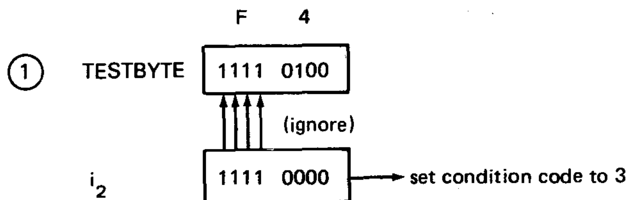
In this example, the TMS instruction tests the byte in main storage at location TESTBYTE. Before execution of the TMS instruction, TESTBYTE contains:

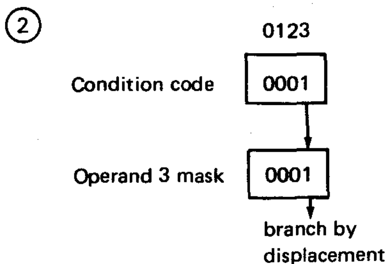
TESTBYTE	F4

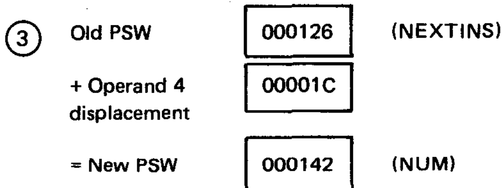
At the time it is assembled, the TMS instruction:

- specifies that only the four high order bits of operand 1 are to be tested (using a mask of X'F0' or B'11110000');
- sets up a branch to location NUM if conditions are met by calculating the displacement from the next sequential instruction (at NEXTINS) to NUM (142—126=$1C_{16}$ bytes) and putting that value in bits 20—31 of the object code; and
- defines a condition code mask of 1, meaning that a branch to NUM will take place only if the condition code is set to 3.

When executed, the instruction operates as follows:







In ①, the i_2 mask causes only the four high order bits of TESTBYTE to be tested. The test takes place and all four bits are found to contain 1, therefore, condition code 3 is set. According to ②, the operand 3 mask has a value of 1 that, corresponding to condition code 3, causes a program branch to NUM. The branch occurs in ③ when the displacement of X'1C' is added to the current PSW address, originally pointing at NEXTINS. This addition forces a branch to location 142, the location of NUM, and the TMS instruction then terminates.

You could have recoded the TMS instruction using the appropriate extended mnemonic, TMBO:

```
TMBO TESTBYTE,X'F0',NUM
```

You would, in this case, omit the m_3 mask, as TMBO automatically supplies the correct mask value, 1.

TR

12.39. TRANSLATE (TR)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
TR	DC	SS	6

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

The *translate* (TR) instruction gets a byte from a table of characters in operand 2 and places it in its corresponding byte of operand 1.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TR	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TR	$s_1(l),s_2$

The translate instruction can translate the bit pattern of each byte in a field (operand 1) to any other bit pattern. This instruction works in conjunction with a table (operand 2) that has been previously defined within the program, and contains the bit patterns that correspond to each byte in operand 1. This instruction operates from left to right, starting with the replacement of the leftmost byte in operand 1, and ending with the replacement of the rightmost byte in operand 1. To find the appropriate code in the table (operand 2), each byte in the operand 1 field is used as an unsigned binary value that is added to the address of the first byte in the table. (The sum of the addition is similar to the addition of base register and displacement values.)

The result of the addition is another address which should be somewhere within the table. The one byte of data that is located at that address replaces the 8-bit binary value of operand 1 that was used in the addition to arrive at that location. After the first byte is replaced, the second byte is operated upon and replaced, and so on until the end of the operand 1 field is reached.

Since there are 256 different combinations of an 8-bit byte (EBCDIC), the maximum size of the translate table is 256 bytes. However, you can define a table smaller than that because input data is normally restricted to a smaller range.

The translate instruction can be used to convert data from one code to another code (i.e., octal to hexadecimal) or it can be used to rearrange data to be stored in a specific sequence.

Operational Considerations:

- Operand 1 must be a main storage location and can be defined in any format.
- Operand 2 must be a table that is previously defined within the program.
- Operand 2 cannot exceed 256 bytes in length.
- One byte in operand 2 replaces one byte in operand 1.
- Each byte in operand 1 is treated as an unsigned binary value which is added to the address of the first byte in operand 2.

Example:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1         10          16
--------------------------------------------------------------
          TR          FIELD,TABLE
          .
          .
          .
FIELD     DC          X'66A4C5F7'
TABLE     DS          ØCL256
          DC          1ØØX'ØØ'
          DC          X'Ø1Ø2Ø3Ø4Ø5Ø6Ø7Ø8Ø9ØA'
          DC          5ØX'ØØ'
          DC          X'ØBØCØDØEØF1Ø111213'
          DC          1ØX'ØØ'
          DC          X'1415161718191A1B1C'
          DC          5ØX'ØØ'
          DC          X'1D1E1F2Ø212223242526'
          DC          4X'ØØ'
```

FIELD before execution of TR instruction:

0110	0110	1010	0100	1100	0101	1111	0111	binary
6	6	A	4	C	5	F	7	hex

FIELD after execution of TR instruction:

0000	0011	0000	1111	0000	0000	0010	0110	binary
0	3	0	F	0	0	2	6	hex

In this example, FIELD (operand 1) contains four addresses to be added to the address of the first byte of TABLE (operand 2), one at a time, to access a specific byte within the table. The table with the label TABLE is defined as part of this program.

The first byte in operand 1, a hexadecimal 66, has a decimal equivalence of 102. When a decimal value of 102 is added to the address of the first byte in TABLE, the 103rd byte in the table (a hexadecimal 03) is accessed. That byte replaces the hexadecimal 66 and the second byte in operand 1 is processed.

The second byte in operand 1 (a hexadecimal A4) has a decimal equivalence of 164. When a decimal value of 164 is added to the address of the first byte in TABLE, the 165th byte in the table (a hexadecimal 0F) is accessed. That byte replaces the hexadecimal A4 and the third byte in operand 1 is processed.

The third byte in operand 1 (a hexadecimal C5) has a decimal equivalence of 197. When a decimal value of 197 is added to the address of the first byte in TABLE, the 198th byte in the table (a hexadecimal 00) is accessed. That byte replaces the hexadecimal C5 and the fourth byte in operand 1 is processed.

The fourth byte in operand 1 (a hexadecimal F7) has a decimal equivalence of 247. When a decimal value of 247 is added to the address of the first byte in TABLE, the 248th byte in the table (a hexadecimal 26) is accessed. That byte replaces the hexadecimal F7 and the TR instruction terminates.

12.40. TRANSLATE AND TEST (TRT)

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
TRT	DD	SS	6

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
☐ SET TO 3
SEE OPER. CONSIDERATIONS

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

The *translate and test* (TRT) instruction gets a byte from a table of characters in operand 2 and examines it for the presence of a hexadecimal 00. If found, it continues processing with the next byte in operand 1. If not found, execution of the TRT instruction is terminated and the address of that byte in operand 1 and the nonzero byte in the table are saved. The result of the byte examination in the table determines the setting of the condition code, bits 34 and 35 of the PSW. (See 8.1.)

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TRT	$d_1(l,b_1),d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TRT	$s_1(l),s_2$

This instruction works in conjunction with a table (operand within the program) and contains bit patterns that correspond to each byte in operand 1. This instruction operates from left to right starting with the first byte in operand 1 and ending with the last byte in operand 1 or when a nonzero byte is found in the table. To find the appropriate code in the table (operand 2), each byte in the operand 1 field is used as an unsigned binary value which is added to the address of the first byte in the table. This method of accessing a byte from a table is performed in the same manner as the translate (TR) instruction.

The result of the addition of an unsigned binary value to the address of the first byte is another address which should be somewhere within the table. The one byte of data that is located at that address is examined for the presence of a hexadecimal 00. If found, processing continues with the next byte in operand 1. If a hexadecimal 00 is not found, execution of the TRT instruction is terminated.

The address of that byte in operand 1 is inserted in the low order 24 bits of register 1, with the high order eight bits remaining unchanged. The nonzero byte from the table is inserted in the low order eight bits of register 2, with the high order 24 bits remaining unchanged.

The condition code is set as follows:

- to 0 if all the bytes examined in the table are zero;
- to 1 if a nonzero character is found in the table before the last byte in operand 1 is processed; or
- to 2 if a nonzero character is found in the table that corresponds to the last byte in operand 1.

The translate and test instruction is used to find certain characters in an input stream. You can set up operand 2 (table) with all zero bytes for those characters to be skipped over and all nonzero bytes for those characters to be detected and used.

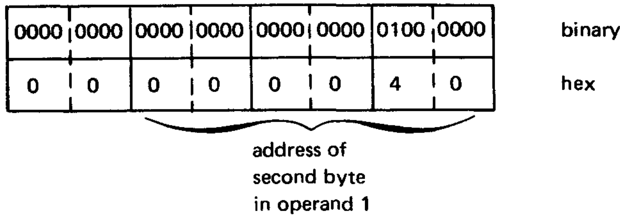
Operational Considerations:

- Operand 1 must be a main storage location and can be defined in any format.
- Operand 2 must be a table that is previously defined within the program.
- Operand 2 cannot exceed 256 bytes in length.
- Each selected byte in the table (operand 2) is examined for the presence of a hexadecimal 00.
- Each byte in operand 1 is treated as an unsigned binary value which is added to the address of the first byte in operand 2.
- The condition code is set accordingly.

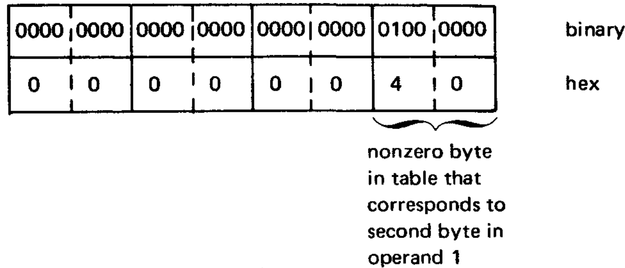
Example:

```
LABEL    ΔOPERATIONΔ                            OPERAND
1         10      16
-------------------------------------------------------------
          TRT     AREAIN,TRTTABLE
           •
           •
           •
AREAIN    DC      X'324Ø48'
TRTTABLE  DS      ØCL256
          DC      64X'ØØ'
          DC      X'4Ø'
          DC      191X'ØØ'
```

Register 1 after execution of TRT instruction:



Register 2 after execution of TRT instruction:



In this example, the only nonzero byte in the table is the 65th byte which is the address of the first byte in the table +64. The hexadecimal 40 at that location causes the TRT instruction to terminate. Then the address of the blank, also a hexadecimal 40, is put into register 1. The nonzero character in the table, a blank, is put in the rightmost byte of register 2. The condition code is set to 1 since a nonzero character is found in the table before the last byte in operand 1 has been processed.

13. Privileged and Status Switching Instructions

13.1. GENERAL

A privileged instruction is an instruction used by the operating system. A privileged instruction *cannot* be used in a program operating under the SPERRY UNIVAC Operating System/3 (OS/3). If a program operating under OS/3 uses a privileged instruction, a privileged operation program exception causes the program to terminate without executing.

The privileged instructions are included in this book because they can be assembled under OS/3 even though they can't be executed.

The checkoff table used with each instruction is explained in Appendix D.

Some of the status-switching instructions are also privileged, but three are not. These instructions are *set program mask* (SPM), *supervisor call* (SVC), and *test and set* (TS).

Since the status switching instructions manipulate portions of the program status word (PSW), it might be helpful to read the PSW field description in 8.1.

13.2. STATUS-SWITCHING PRIVILEGED INSTRUCTIONS

The status-switching instructions can change the program status word (PSW), the contents of the protect key storage, and the current relocation register.

HPR

13.2.1. Halt and Proceed (HPR) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
HPR	**99**	**SI**	**4**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	HPR	$d_1(b_1), i_2$
HALTHERE	HPR	0(5), 81

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	HPR	s_1, i_2
	HPR	TAG, X'FF'

13.2.2. Insert Storage Key (ISK) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
ISK	09	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ISK	r_1, r_2
INKEY	ISK	3, 4

LPSW

13.2.3. Load Program Status Word (LPSW) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LPSW	82	S	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ SEE NOTE

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LPSW	$d_2(b_2)$

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LPSW	s_2

NOTE:

Condition code is set as specified in the new PSW loaded.

SSK

13.2.4. Set Storage Key (SSK) Instruction

General

OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SSK	08	RR	2

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit and Implicit Format:

LABEL	△ OPERATION △	OPERAND
[symbol]	SSK	r_1, r_2
SETKEY	SSK	2,3

SSM

13.2.5. Set System Mask (SSM) Instruction

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
SSM	80	S	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SSM	$d_2(b_2)$
SETSM	SSM	6(32)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SSM	s_2
SETSM	SSM	SYSMASK

13.3. INPUT/OUTPUT PRIVILEGED INSTRUCTIONS

This set of privileged instructions initiates, controls, and ends all input/output operations in System 80.

CLRCH

13.3.1. Clear Channel (CLRCH) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLRCH	9F02	S	4

Condition Codes

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

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLRCH	$d_2(b_2)$
STCLR	CLRCH	48(6)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLRCH	s_2
STCLR	CLRCH	ADD16

CLRDV

13.3.2. Clear Device (CLRDV) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
CLRDV	9DX2	RS	4

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

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLRDV	$r_1,d_2(b_2)$
CLEAR1	CLRDV	8,0(6)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLRDV	r_1,s_2
CLEAR1	CLRDV	8,CLRDSC

Operational Consideration:

- The r_1 field you specify goes into bits 8—11 of the object code.

13.3.3. Enqueue I/O (EIO) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
EIO	E0	SS	6

Condition Codes

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

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EIO	$d_1(i_1,b_1),d_2(r_1,b_2)$
QUEUE1	EIO	0(3,4),8(3,6)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EIO	$s_1(i_1),s_2(r_1)$
QUEUE1	EIO	OP2(3),ENQ(3)

HDV

13.3.4. Halt Device (HDV) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
HDV	**9E01**	**S**	**4**

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

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	HDV	$d_2(b_2)$
HALT10	HDV	8(10)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	HDV	s_2
HALT10	HDV	STOPCD

13.3.5. Load Channel Register (LCHR) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LCHR	9F03	S	4

Condition Codes

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

Possible Program Exceptions

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

Explicit Format:

LABEL	△ OPERATION △	OPERAND
[symbol]	LCHR	$d_2(b_2)$
OP2	LCHR	8(2)

Implicit Format:

LABEL	△ OPERATION △	OPERAND
[symbol]	LCHR	s_2
OP2	LCHR	CHR3

LDA

13.3.6. Load Directive Address (LDA) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LDA	51	RX	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LDA	$r_1,d_2(x_2,b_2)$
LOAD8	LDA	8,16(4,5)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LDA	$r_1,s_2(x_2)$
LOAD8	LDA	8,ADDR(4)

LIA

13.3.7. Load I/O Address (LIA) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LIA	**61**	**RX**	**4**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LIA	$r_1,d_2(x_2,b_2)$
LDADDR	LIA	7,8(2,3)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LIA	$r_1,s_2(x_2)$
LDADDR	LIA	7,DATA1

MIO

13.3.8. Move I/O (MIO) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
MIO	81	RS	4

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

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MIO	$r_1,r_3,d_2(b_2)$
ADDR5	MIO	6,10,24(3)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	MIO	r_1,r_3,s_2
ADDR5	MIO	6,10,BUFF1

PRB

13.3.8.1. Put IORB (PRB) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
PRB	**0C**	**RR**	**2**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	PRB	r_1,r_2
PUTIT	PRB	3,4

GRB

13.3.8.2. Get IORB (GRB) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
GRB	0B	RR	2

Condition Codes
■ IF RESULT = 0, SET TO 0
■ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	GRB	r_1, r_2
GETIT	GRB	3,4

13.3.9. Start Device (SDV) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SDV	9C02	S	4

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

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SDV	$d_2(b_2)$
START1	SDV	16(4)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SDV	s_2
START1	SDV	STDATA

13.4. DIAGNOSTIC PRIVILEGED INSTRUCTIONS

The diagnostic instructions are *execute diagnose* (EXD), *reset* (RESET), and *store status* (STS) instruction.

EXD

13.4.1. Execute Diagnose (EXD) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
EXD	**8300**	**S**	**4**

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ SEE NOTE

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EXD	$d_2(b_2)$
DIAG1	EXD	8(4)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EXD	s_2
DIAG1	EXD	LOCN2

NOTE:

Condition code may be set by the subject diagnostic or special function.

13.4.2. RESET Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
RESET	8301	S	4

Condition Codes

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

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	RESET	$d_2(b_2)$
INST3	RESET	44(3)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	RESET	s_2
INST3	RESET	PLACE1

STS

13.4.3. Store Status (STS) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STS	8302	S	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STS	$d_2(b_2)$
ADDSTORE	STS	0(11)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STS	s_2
ADDSTORE	STS	STDATA

13.5. INTERVAL TIMER PRIVILEGED INSTRUCTION

The OS/3 hardware contains an interval timer register that is controlled by the *service timer register* (STR) instruction.

STR

13.5.1. Service Timer Register (STR) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STR	**03**	**RR**	**2**

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

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

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STR	r_1, r_2
TIMREG	STR	5, 6

13.6. CONTROL REGISTER PRIVILEGED INSTRUCTIONS

These instructions operate on the control registers maintained as part of System 80 hardware.

LCTL

13.6.1. Load Control (LCTL) Instruction

General

OPCODE MNEM.	OPCODE HEX.	FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
LCTL	B7	RS	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT <0, SET TO 1
- ☐ IF RESULT >0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LCTL	$r_1,r_3,d_2(b_2)$
CONT1	LCTL	4,6,12(3)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LCTL	r_1,r_3,s_2
CONT1	LCTL	4,6,CNTLSTOR

13.6.2. Store Control (STCTL) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STCTL	B6	RS	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STCTL	$r_1,r_3,d_2(b_2)$
STOR1	STCTL	3,5,0(6)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STCTL	r_1,r_3,s_2
STOR1	STCTL	3,5,STORDATA

13.7. RELOCATION REGISTER PRIVILEGED INSTRUCTIONS

These instructions operate on the relocation registers maintained as part of System 80 hardware.

LRR

13.7.1. Load Relocation Register (LRR) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LRR	A3	RS	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	LRR	$r_1,d_2(b_2)$
DOLOAD	LRR	3,0(6)

Implicit Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	LRR	r_1,s_2
DOLOAD	LRR	3,RELOC3

STRR

13.7.2. Store Relocation Register (STRR) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
STRR	**A2**	**RS**	**4**

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT <0, SET TO 1
☐ IF RESULT >0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STRR	$r_1,d_2(b_2)$
STOR12	STRR	4,8(3)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STRR	r_1,s_2
STOR12	STRR	4,STLOC

13.8. GENERAL REGISTER PRIVILEGED INSTRUCTIONS

These instructions operate on the problem general register set.

SLM

13.8. GENERAL REGISTER PRIVILEGED INSTRUCTIONS

13.8.1. Supervisor Load Multiple (SLM) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SLM	B8	RS	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLM	$r_1,r_3,d_2(b_2)$
LOAD36	SLM	3,6,20(9)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SLM	r_1,r_3,s_2
LOAD36	SLM	3,6,DATAFIVE

13.8.2. Supervisor Store Multiple (SSTM) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SSTM	B0	RS	4

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
■ UNCHANGED

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SSTM	$r_1,r_3,d_2(b_2)$
STORE47	SSTM	4,7,0(2)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SSTM	r_1,r_3,s_2
STORE47	SSTM	4,7,AREATWO

13.9. DATA CHECKING PRIVILEGED INSTRUCTION

Bit checking on main storage data is performed by the *longitudinal redundancy check* privileged instruction.

LRC

13.9.1. Longitudinal Redundancy Check (LRC) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
LRC	830E	S	4

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

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

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LRC	$d_2(b_2)$
PRM1	LRC	0(2)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LRC	s_2
PRM1	LRC	CHECK1

13.10. PROGRAM LOAD PRIVILEGED INSTRUCTION

The initial program load (IPL) instruction performs the initial program load (IPL) function in the same manner as if the IPL key were pressed on the console workstation.

IPL

13.10.1. Initial Program Load (IPL) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
IPL	8303	S	4

Condition Codes

- ☐ IF RESULT = 0, SET TO 0
- ☐ IF RESULT < 0, SET TO 1
- ☐ IF RESULT > 0, SET TO 2
- ☐ IF OVERFLOW, SET TO 3
- ■ UNCHANGED

Possible Program Exceptions

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

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	IPL	$d_2(b_2)$
IPL01	IPL	2(3)

13.11. SWITCH LIST PRIVILEGED INSTRUCTION

One privileged instruction scans the supervisor switch list: the switch list scan (SWLS) instruction.

SWLS

13.11.1. Switch List Scan (SWLS) Instruction

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SWLS	**830F**	**S**	**4**

Condition Codes

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

Possible Program Exceptions

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

Explicit and Implicit Format:

The bit pattern is the format of the instruction.

SPM

13.12. SET PROGRAM MASK (SPM) STATUS-SWITCHING INSTRUCTION

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SPM	04	RR	2

Condition Codes
■ SET TO 0
■ SET TO 1
■ SET TO 2
■ SET TO 3
SEE OPER. CONSIDERATIONS

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

The *set program mask* (SPM) instruction replaces bits 34 through 39 of the current PSW with bits 2 through 7 of the operand 1 register.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SPM	r_1
SETM	SPM	3

Operational Considerations:

- Bits 0, 1, and 8 through 31 of the operand 1 register are ignored by the OS/3 hardware.
- The condition code is set equal to bit positions 2 and 3 of operand 1.

Example:

```
LABEL   ΔOPERATIONΔ                   OPERAND
1        10        16
________________________________________________
            .
            .
            .
LODREG   L         3,MASK
SETM     SPM       3
            .
            .
            .
MASK     DC        F'1'
```

In this example, I loaded register 3 with the contents of a main storage area called MASK. MASK contains a full word of binary 1's. When the SPM instruction is executed, bits 34 through 39 of the PSW are replaced with bits 2 through 7 of register 3.

SVC

13.13. SUPERVISOR CALL (SVC) STATUS-SWITCHING INSTRUCTION

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
SVC	**0A**	**RR**	**2**

Condition Codes

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

SEE OPER. CONSIDERATIONS

Possible Program Exceptions

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

The *supervisor call* (SVC) instruction causes an interrupt and replaces bits 24 through 31 of the current PSW with the 1-byte contents of operand 1.

Explicit and Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SVC	i_1
SUPCALL	SVC	38

Operational Considerations:

- The operand you specify is an *immediate* byte of data, which is a 1-byte absolute term.
- Once the SVC instruction is executed, the PSW with its new contents is stored, and a new PSW is controlling your program.
- The condition code is equal to bits 34 and 35 of the PSW after the supervisor call is granted.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10       16
-----------------------------------------------------------
CALL      SVC       X'OF'
```

A supervisor call interrupt is generated, and the value X'00001111' is stored in the old PSW.

13.14. TEST AND SET (TS) STATUS-SWITCHING INSTRUCTION

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		
TS	93	S	4

Condition Codes

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

SEE OPER. CONSIDERATIONS

Possible Program Exceptions

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

The test and set (TS) instruction tests the zero bit of the operand 1 main storage area for a 1 or a 0 and sets the condition code according to the result.

Explicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TS	$d_2(b_2)$
TEST	TS	32(16)

Implicit Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TS	s_2
TEST	TS	STORAGE

Operational Considerations:

- Only the first bit of the operand is tested to determine the condition code.
- All eight bits of the operand are set to binary 1's after the condition code is set.
- The condition code is set as follows:
  - — 0 if bit position 0 is zero; or
  - — 1 if bit position 0 is 1.

Operational Considerations:

- Only the leftmost bit of the operand is tested to determine the condition code setting.
- All eight bits of the operand byte are set to 1 after the condition code is set.
- The condition code is set as follows:
  - to 0 if the tested bit is 0; or
  - to 1 if the tested bit is 1.
- This instruction can be used by two programs referencing the same main storage byte. A condition code setting of 0 indicates that the area is available for use by the testing program. A condition code setting of 1 indicates that the area is not available.

Example:

```
LABEL     △OPERATION△                    OPERAND
1          10        16
---------------------------------------------------------------
TESTSW     TS        BYTE
            .
            .
            .
BYTE       DC        B'11110000'
```

When the TS instruction is executed, the leftmost bit of BYTE is tested. Since the bit is 1, the condition code is set to 1.

DELETION

Section 14 has been deleted.

PART 4. BAL DIRECTIVES

15. Introduction to Directives

The OS/3 assembly language includes assembler directives (Table 15—1) that enable the user to control assembler operation. Assembler directives control the assembler at assembly time just as application instructions control the processor at execution time. Housekeeping, program organization, assembly listing modification, and I/O control are the tasks of the directives.

The assembler application instructions the programmer uses to control the processor operation are discussed in Part 3 of this manual. The major portion of the program statements consists of these instructions. Just as there are mnemonics to direct the generation of the instructions, there are directives to control the operation of the software language processor (the assembler). These are called assembler directives.

Table 15—1. Assembler Directives

Types of Directives	Basic Function	Where Discussed
EQUATE OPSYM	Symbol definitions Delete operation code	Section 16
ASSEMBLER CONTROL	Control program name and organization	Section 17
BASE REGISTER ASSIGNMENT	Directs registers to be used and when	Section 18
LINKING AND SECTIONING	Control of modules to be linked	Section 19
LISTING CONTROL	Control of the assembly listing	Section 20
I/O CONTROL	Control of input/output data	Section 21

16. Equate and Delete Operation Code Directives

EQU

16.1. EQUATE (EQU)

The *equate* (EQU) directive defines the length and value of a symbol using another symbol as all or part of the definition.

The format is as follows:

LABEL	ΔOPERATIONΔ	OPERAND
symbol	EQU	e[,a]

where:

e
Is an absolute or relocatable expression.

a
Is an absolute expression.

All symbols must be predefined.

The symbol in the label field is defined as the value of the first expression in the operand. The maximum values are -2^{23} to $2^{23}-1$. The length attribute of the symbol is equal to the second expression (a) if explicitly stated. If the second expression (a) is omitted, the symbol will have the length attribute of the first term in the first expression (e). If the first term is an * or a self-defining term, the length attribute of the symbol is 1. (See the following coding examples.)

Examples:

```
     LABEL    ΔOPERATIONΔ                      OPERAND
     1        10       16

1.  TAG       DS       25CL1Ø
2.  HIDE      EQU      1ØØ+TAG,15Ø
3.  SEEK      EQU      TAG+127Ø-*
4.  GO        EQU      TAG+127Ø-*,2ØØ
5.  RØ        EQU      Ø
6.  R1        EQU      1
```

If the value of the location counter is 2000 when instructions 1 through 4 are encountered, the symbols have the following location counter 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. If line 4 is substituted in place of line 3, then GO has an absolute value of 1020 (2000 + 1270 — 2250) and a length attribute of 200. (The 200 overrides the length of TAG.)

5. and 6. The registers 0 and 1 are equated to R0 and R1. (See 6.1.)

16.2. DELETE OPERATION CODE (OPSYM)

The delete operation code (OPSYM) directive allows you to tell the assembler not to accept a certain mnemonic operation code.

The format is as follows:

LABEL	ΔOPERATIONΔ	OPERAND
mnemonic operation code	OPSYM	unused

After you use the OPSYM directive to declare a mnemonic code as unacceptable, the assembler will not generate the normal object code for that mnemonic if it appears after the OPSYM. You are then free to use the declared mnemonic another way, for example, as the mnemonic code of a macro prototype statement.

Examples:

```
    LABEL    ΔOPERATIONΔ                         OPERAND
    1        10        16

 1.          MACRO
 2.          A         &QUANT,&Q2,&SUM
 3.          L         13,&QUANT
 4.          A         13,&Q2
 5.          ST        13,&SUM
 6.          MEND
 7.          START     Ø
 8. A        OPSYM
             .
             .
             .
 9. CALCU    A         PAY,RAISE,TOTAL
             .
             .
             .
10.          END
```

In this example, the program is preceded by a macro definition which is used in my program. Line 2 contains the mnemonic code A, which is the mnemonic operation code for an *add full word* instruction. Before you can call the A macro into your program, you must use an OPSYM directive to tell the assembler not to recognize A as the *add full word* mnemonic. The OPSYM directive must code before the line of code which references the macro, that is, line 8 must precede line 9.

The OPSYM directive cannot be used from within a PROC/MACRO or from within code generated as a result of conditional assembly statements.

17. Assembler Control Directives

Assembler control directives are available to name the program and specify an initial location counter, section the program, alter the location counter to a specified value, indicate the end of a program, and designate the instruction the program will begin with. Table 17—1 is a summary of the assembler control directives available to the user of the OS/3 assembler.

Table 17—1. Assembler Control Directives

Directives	Basic Function	Where Discussed
CNOP	Condition no operation	17.1
END	Program end	17.2
LTORG	Generate literal pool	17.3
ORG	Specify location counter	17.4
START	Program start	17.5

CNOP

17.1. CONDITION NO OPERATION (CNOP)

The *condition no operation* (CNOP) directive adjusts the location counter to a half-word, full-word, or double-word storage boundary. The format of the CNOP directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	CNOP	a_1,a_2

where:

a_1 and a_2
Are absolute expressions consisting of predefined terms.

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 full-word boundary alignment, and 0, 2, 4, and 6 for double-word boundary alignment.

- 0 indicates a full-word or double-word boundary;
- 2 indicates the second byte (first half word) past the boundary;
- 4 indicates the fourth byte (second half word) past a double-word boundary; and
- 6 indicates the sixth byte (third half word) past a double-word boundary.

Permissible values for the second expressions 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, the 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    ΔOPERATIONΔ                              OPERAND
   1        10        16
  ---------------------------------------------------------------------------
1.|         CNOP      Ø,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.

END

17.2. PROGRAM END (END)

The *program end* (END) directive indicates the end of a source program or macro definition written in PROC format. (See Appendix A and Section 25.)

The format of the END directive is:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	END	[e]

where:

e
: Is a relocatable expression.

The END directive must be the last statement in the source program. An expression in the operand field designates the point in the 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. If the END directive terminates a proc, the label and operand fields are not used.

Examples:

```
   LABEL   ΔOPERATIONΔ                      OPERAND
   1        10      16

1. FOX      END     BEGN
2.          END     GO+324
3.          END
```

All three of the END statements halt assembly, but each transfers control to a different address in the program.

1. Control is transferred to a statement labeled BEGN in the program. The label FOX is assigned the address associated with the last byte of the assembly.

2. If GO has a value of 1000, control is transferred, and the next instruction to be processed is located at address 1324.

3. If no operand is specified, control is transferred to the first address of the program loaded.

17.3. GENERATE LITERALS (LTORG)

The *generate literal pool* (LTORG) directive generates all literals previously defined into a data pool within the source program. The format of the LTORG directive is:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	LTORG	unused

The literals are pooled following the occurrence of the LTORG directive. A symbol in the label field represents the first byte of the generated literal pool and is assigned a length attribute of 1. LTORG directives may not appear within a dummy control section (19.3) 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 pooled at the end of the first control section. The programmer then must ensure that a valid base register is available to address the locations in the literal pool.

Literals are placed in the literal pool according to their total length (duplication factor multiplied by the length of the constant). The literal pool consists of four sections:

1. Literals with total lengths that are multiples of double words (eight bytes)

2. Literals with total lengths that are multiples of full words (four bytes)

3. Literals with total lengths that are multiples of half words

4. Any remaining literals

Within each pool section, the literals are stored in order of occurrence. Before the literal pool is generated, the location counter is adjusted to a double-word boundary. If two control sections are assembled together and a LTORG is not included in the second or following sections, then all the literals defined in all the sections will be pooled in the first control section and may subsequently be available only to that first section. To ensure that each linked control section can use the literals declared by it, an LTORG should be used within each control section.

ORG

17.4. SPECIFY LOCATION COUNTER (ORG)

The *specify location counter* (ORG) directive sets or resets the location counter to a specified value. The format of the ORG directive is:

LABEL	ΔOPERATIONΔ	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 no expression is present, the location counter is set to the highest location previously assigned in that control section. A symbol in the label field has the same value as the expression in the operand field and is assigned a length attribute of 1. The expression in the operand field must be relocatable. Its value must represent an address in the same control section in which the ORG occurs. This address 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. All terms in the expression must be predefined.

The ORG directive permits the location counter to be set to a value not on a half-word boundary.

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

Example:

```
LABEL    ΔOPERATIONΔ                                OPERAND
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.

Additional examples of the ORG directive follow.

Examples:

```
     LABEL    ΔOPERATIONΔ              OPERAND                ΔCOMMENTS
     1        10         16

1.  INPUT     DS         CL80                                 INPUT FIELD 80 BYTES
2.            ORG        INPUT                                INPUT FOR REC1
    REC1      DS         CL20
    FLD1      DS         CL60
3.            ORG        INPUT                                INPUT FOR REC2
    REC2      DS         CL30
    FLD2      DS         CL30
    FLD22     DS         CL20
4.            ORG        INPUT                                INPUT FOR REC3
    REC3      DS         CL15
    FLD3      DS         CL25
    FLD33     DS         CL40
```

1. An input area for an 80-byte card is defined with no subfields.

2. The input field is redefined in place to show two subfields.

3. Redefine INPUT for different organizations of the field.
and
4.

Instructions 1 through 4 define four different types of cards or other 80-byte records.

START

17.5. PROGRAM START (START)

The *program start* (START) directive defines the program name, the name of the first control section, and the initial location counter value. The format of the START directive is:

LABEL	△OPERATION△	OPERAND
[symbol]	START	[a]

where:

a

Is an absolute expression.

A symbol in the label field becomes the name of the first or only control section in the program. If the label field is blank, an unnamed control section is begun. All statements following the START directive are assembled as part of the control section until another unique control section definition is encountered.

The label field of a CSECT directive, which can contain the same name as the label field of the START directive, identifies the continuation of the control section. A blank label field in the CSECT directive identifies the continuation of an unnamed control section that began with an unnamed START directive.

The symbol in the label field of the START directive also identifies or names the object program. If the START directive is unnamed, the object module is assigned the name ASMOBJ. The symbol must be a valid symbol. It is an automatic entry point and has a length attribute of 1. The START directive must not be preceded by any statements which would initiate a control section.

The self-defining term in the operand field of the START directive establishes the initial location counter value for the first control section. If the self-defining term represents a value which is not a multiple of 8, the START directive is flagged and the location counter set to the next higher multiple of 8. If the operand is omitted, the initial control section is assigned a location counter value of zero.

Examples:

```
LABEL    △OPERATION△                                OPERAND
1         10         16
------------------------------------------------------------------
TEST      START      1Ø63
TEST      START      X'427'
```

The location counter contents for either of these statements would be 1064, which is the next higher multiple of 8 from 1063.

18. Base Register Assignment Directives

The OS/3 assembler converts 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 assembler directives USING and DROP are available for this purpose.

- The *unassign base register* (DROP) directive informs the assembler that certain registers are no longer to be used for base registers.
- The *assign base register* (USING) directive informs the assembler that the specified registers are available for use as base registers.

DROP

18.1. UNASSIGN BASE REGISTER (DROP)

The *unassign base register* (DROP) directive informs the assembler that the registers specified are no longer available for base register assignment. The format of the DROP directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	DROP	r_1 [,...,r_n]

where:

r_1[,...,r_n]
: Specifies that the declared registers (0 through 15) are no longer available for base register assignment.

Registers previously made available for base register assignment may be dropped and made available again in a USING directive. (See 18.2.) The value assumed to be in a base register may be changed by coding another USING directive without an intervening drop of that register.

Examples:

```
   LABEL   ΔOPERATIONΔ                        OPERAND
   1         10       16

1.           DROP     1
2.           DROP     1,3,4
```

1. This directive specifies that register 1 is no longer available to the assembler for base register assignment.

2. This directive specifies that registers 1, 3, and 4 are no longer available for base registers.

18.2. ASSIGN BASE REGISTER (USING)

The *assign base register* (USING) directive informs the assembler that a specified register is available for base register assignment and will contain a specific value at execution time. The value must be loaded by the program into the base register that the USING directive specifies. The assembler maintains a USING table of the specified registers. The format of the USING directive is:

LABEL	△ OPERATION △	OPERAND
unused	USING	v,r_1 [$,\ldots,r_n$]

where:

v
: Is the value assumed to be in the first specified register at execution time. This value may be relocatable or absolute. Literals are not permitted.

r_1[$,\ldots,r_n$]
: Specifies that the declared registers (0 through 15) will be used as base registers loaded at execution time. These register numbers do not necessarily have to be assigned in ascending sequence.

The first register specified after 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.

Register 0 may be specified as a valid base register; however, the assembler assumes that it always contains the value 0 and calculates displacements as if the operands were zero. Register 0 must be the operand specified by r_1, and any registers specified in the operand field following register 0 are assumed to contain increments of 4096 from zero.

When v is absolute, the indicated registers may be used to process only absolute effective addresses.

When v is relocatable, the indicated registers can be used to process only relocatable effective addresses. The registers $r_1,\ldots,r_n$ are used to process only those addresses in the same control section as the address represented by v.

The value specification in a USING directive sets the lower limit of an address range; the upper limit 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 zero, and the line is flagged with an error indication. If more than one register contains the value yielding the smallest displacement, the highest numbered register is selected.

Examples:

```
    LABEL   ΔOPERATIONΔ                       OPERAND
    1        10       16

1.           USING    4ØØØ,8
2.           USING    8ØØØ,1,2,3,6,7,8,12
3.           USING    *,5
4.           USING    TAG,R9
```

1. A range of 4096 bytes is covered by register 8 at location 4000 through 8095. The value 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. Register 5 is used as the base register, with the value of the location counter contained in register 5.

4. The register declared by the symbol R9 is assumed to contain the base address of the symbol "TAG".

19. Program Linking and Sectioning Directives

A program or a portion of a program assembled as a single unit is called a module. A complex program may consist of many modules; some may be standard subroutines that can be used in any program.

The assembler provides, as part of its output, information that allows 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 the module that 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 must be externally defined by an ENTRY directive in module A and defined by an EXTRN directive in module B. 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 an optional capability of dividing one module into different sections. A control section is a group of instructions, constants, and storage areas, the positions of which, relative to each other, are fixed and must remain fixed to ensure proper coding. Proper execution of instructions and data in one control section must not depend on their positions relative to instructions or data in any other control section. Because the assembler maintains a separate location counter for each section, control sections may appear in any order for input to the assembled. Statements belonging to one control section may be intermixed with statements belonging to one or more other sections. If the first statement of a control section is a START directive, its label names the control section.

Each module may have a maximum of 255 external symbol identification (ESID) items. An ESID item contains special information 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 unique symbol used in a V-type address constant
- Each symbol used in a V-type address constant
- Each control section
- Each dummy control section
- Each common storage definition section

COM

19.1. COMMON STORAGE DEFINITION (COM)

The *common storage definition* (COM) directive enables the programmer to define a control section which is a common storage area for two or more separately assembled routines. The format of the common section may be described by DS and DC directives. Labels appearing within the sections are defined. Like a dummy control section, no data or instructions are assembled in a common section. It has a separate location counter with an initial value of zero. Data may be entered into a common section only by execution of a program which refers to it, or by loading a control section of the same name. Such CSECTS are called block data sections. DC instructions act as DS instructions in the COM area because neither instructions nor constants in a common storage area are assembled. Labels defined in a common section are not subject to the restrictions imposed on dummy section labels.

One assembly can define only one blank (unnamed) common section. Several like-named COM directives may appear among the source statements. Each COM directive after the first defines a continuation of the common section previously described. When several routines defining like common storage are linked, the resulting module contains only one section corresponding to the like common sections in the input modules. The length of this section is the length of the largest like common section in the input modules. The format of the COM directive is:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	COM	unused

If the common section is unlabeled, the area is addressed by referencing the label of a statement within the common section with a USING directive. (See 18.2.)

Examples:

MODULE 1:

```
    LABEL    ΔOPERATIONΔ                      OPERAND
    1        10         16

1.  MOD1     CSECT
             .
             .
             .
2.  ACOM     COM
    REBEW    DS         CL125
    CHAYA    DS         CL8Ø
             END
```

MODULE 2:

```
   LABEL    ΔOPERATIONΔ                     OPERAND
   1         10        16
3. MOD2      CSECT
             .
             .
             .
4. ACOM      COM
   ELI       DS        CL26Ø
             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

The common storage area for these examples is 260 bytes long (see following listing). The fields REBEW and CHAYA are the same storage area as the first 205 bytes of the field ELI.

Byte Number	Hexadecimal Address	Module 1			Hexadecimal Address	Module 2		
		ACOM	COM			ACOM	COM	
0	00000	REBEW	DS	CL125	00000	ELI	DS	CL260
125	0007D	CHAYA	DS	CL80				
205			END					
260					00104		END	

If more than one object module element refers to a common storage area with the same name, the references are to the same storage area. Only one common storage area is allocated within a load module to satisfy all object module requests for common storage areas with the same name. The size of a common storage area in a load module is determined by the maximum size requested by any object module for common storage with that name. Blank common storage areas are allocated in the same way.

In a multiphase load module, common storage areas are not normally overlaid.

The following rules apply to the use of common storage:

- An entry point cannot have the same name as a labeled common storage area included in the load module.
- When the linkage editor includes module elements (CSECT or COM) with the same name as a labeled common storage area, that section is treated as a block data subprogram (i.e., to initialize values of labeled common blocks) and is loaded into all or a portion of the common storage area. A block data subprogram is loaded when the phase in which it was included is loaded. Blank common cannot be initialized during loading unless the text encountered is for that COM ESD.
- If an object module has requested common storage, the partial inclusion of a single control section from that object module will cause the common storage area defined to be included also, regardless of whether or not the included control section refers to that common storage name. For further information, see the linkage editor portion in the system service programs (SSP) user guide.

CSECT

19.2. CONTROL SECTION IDENTIFICATION (CSECT)

The *control section identification* (CSECT) directive indicates to the assembler the initiation or continuation of a control section. The format of the CSECT directive is:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	CSECT	unused

The symbolic name of the control section defines an entry point of the program being assembled. This symbol must not appear as a symbol for any other source statement except the START directive of its control section or another CSECT directive to indicate continuation of the coding in the same control section.

Each control section is adjusted to begin on a double-word boundary. The value of the symbol is the address of the first byte of the control section and has a length attribute of 1.

If the symbol is blank, the CSECT directive is a continuation of coding for an unnamed control section. If the symbol is blank and is not preceded by an unnamed control section, the CSECT initiates an unnamed control section. Only one unnamed control section is permitted in a module.

Examples:

```
   LABEL   ΔOPERATIONΔ                        OPERAND
   1        10        16
   ------------------------------------------------------------
1. GROSS    START
            .
            .
            .
            .
            .
2. DEDUCT   CSECT
            .
            .
            .
            .
            .
3. GROSS    CSECT
            .
            .
            .
            .
            .
           END
```

(continued)

```
    LABEL    ΔOPERATIONΔ                         OPERAND
    1        10        16
   ------------------------------------------------------------
4. GROSS2X  CSECT
            .
            .
            .
            .
            .
5. DEDUCTX  CSECT
            .
            .
            .
            .
6. GROSS2X  CSECT
            .
            .
            .
            .
            .
            END
```

1. The first control section of coding is labeled GROSS.
2. The second control section of coding is labeled DEDUCT.
3. The coding beginning at line 3 is a continuation of the section labeled GROSS.
4. The first control section of coding is labeled GROSS2X.
5. The second control section of coding is labeled DEDUCTX.
6. The coding beginning at line 3 is a continuation of the section labeled GROSS2X.

DSECT

19.3. DUMMY CONTROL SECTION IDENTIFICATION (DSECT)

A program may contain references to areas that have been defined in other modules. Addressing such areas is facilitated by describing the area and its format to the assembler as a dummy control section. Any statement following a *dummy control section identification* (DSECT) directive is identified as belonging to the dummy control section. The format of the DSECT directive is:

LABEL	∆ OPERATION ∆	OPERAND
[symbol]	DSECT	unused

Storage is not reserved by a DS directive within a dummy control section, and the data and instructions appearing in a dummy control section do not become part of the assembled program. A separate location counter with an initial value of zero is kept for each dummy control section. More than one DSECT directive with the same symbol may appear in a module. The first DSECT directive initiates the dummy control section; the remaining DSECT directives continue it.

Symbols of statements in a dummy control section are called dummy section symbols. The following rules must be observed in using and assigning dummy section symbols:

- An unpaired dummy section symbol may appear only in an expression defining a storage address for a machine instruction or an S-type constant.
- A base register may not be specified for an address field containing an unpaired dummy section symbol.
- 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 user should align the storage area to a double-word boundary.

Coding examples utilizing the DSECT directive follow.

Examples:

```
    LABEL    △OPERATION△                    OPERAND          △COMMENTS
    1         10         16

    BEGIN    BALR        R3,Ø
             USING       *,R3
             L           R4,AREA
             USING       SECTION,R4                          REGISTER 4 FOR DSECT
             .
             .
             .
    MOVE     MVC         WORK(3),CODE
             MVC         WORK1(28),NAME
    AREA     DC          A(TABLE)
1.  SECTION  DSECT
    NAME     DS          CL28
    NUMBER   DS          CL15
    CODE     DS          CL3
    WORK     DS          CL3
    WORK1    DS          CL28
2.           CSECT
    TABLE    DS          CL5Ø
             END
```

1. The coding following DSECT is assigned to a dummy control section.
2. CSECT begins a new control section or continues the current control section.

ENTRY

19.4. EXTERNALLY REFERENCED SYMBOL DECLARATION (ENTRY)

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 referenced and is declared by the ENTRY directive. The format of the ENTRY directive is:

LABEL	ΔOPERATIONΔ	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 reference records. (See 19.5.)

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10        16
------------------------------------------------------------
          ENTRY     WRD32,REBEW,ILE,CHAYA
```

WRD32, REBEW, ILE, and CHAYA are symbols defined in module 1 for the use of other modules. ENTRY permits other modules to reference the symbol defined by the ENTRY directive declaring it.

19.5. EXTERNALLY DEFINED SYMBOL DECLARATION (EXTRN)

The assembler must be informed of all symbols used in the module being assembled that are defined in some other module. References to these symbols are called external definitions; these symbols are declared in the *externally defined symbol declaration* (EXTRN) directive. The format of the EXTRN directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	EXTRN	symbol[,symbol,...,symbol]

Each symbol in the operand field is declared to be a symbol defined in some other module. (See 19.4.) The symbolic name and the external symbol identification assigned by the assembler are input to the linkage editor as an external definition record. Each reference to the externalized symbol creates an appropriate relocation mask to allow reference resolution at linkage editor time. When an EXTRN and a definition for an identical symbol appear in the same assembly, the EXTRN reference is discarded automatically, and the definition is accepted regardless of the order of appearance of either item.

Examples:

```
MODULE A:

LABEL   ΔOPERATIONΔ                     OPERAND
1        10        16
--------------------------------------------------------------
FOX      MVO       DEST(5),ORIG(3)
         DC        A(CAT)
         DC        A(DOG)
JOE      BC        8,1Ø48
         .
         .
         .
MAT      BCT       1Ø,SET
         DC        A(PIG)
         .
         .
         .
         ENTRY     FOX,JOE,MAT
         EXTRN     CAT,DOG,PIG
```

```
MODULE B:

LABEL    ΔOPERATIONΔ                      OPERAND
1         10        16
----------------------------------------------------------------
          DC        A(FOX)
          .
          .
          .
CAT       PRINT     DATA
          DC        A(JOE)
DOG       MVC       NAHS,NAD
          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.

19.6. SUBROUTINE LINKAGE

In addition to writing the code in your external subroutines, you must provide for certain conventions that link your subroutines to your program. The conventions are:

- Saving and restoring the contents of the registers
- Establishing a new base register
- Branching back to the program

Each of these conventions uses a specific register. The table that follows lists the registers and their use.

Register	Use
1	Contains the address of the table holding variables being passed to the subroutine.
13	Contains the address of an 18-word area that will save the contents of the registers as they were before your subroutine began execution.
14	Contains the address that your program branches to after it finishes its execution.
15	Used as the base register

The format that follows shows how you should use these registers to meet these conventions.

```
        LABEL      △OPERATION△      OPERAND
1.  symbol        CSECT
2.                STM               14,12,12(13)
3.                BALR              15,0
4.                USING             *,15
                    .
                    .
                    .
                 processing code here
                    .
                    .
                    .
5.                LM                14,12,12(13)
6.                BR                14
                    .
                    .
                    .
                 storage definition (if any)
                    .
                    .
                    .
7.                END
```

Explanation:

1. This line uses the CSECT directive to name the subroutine.

2. This line saves the contents of the registers in an 18-word save area located at the address stored in register 13.

3. and 4. These two lines establish register 15 as the base register for the execution of the subroutine.

5. This line restores the contents of the registers from an 18-word save area located at the address stored in register 13.

6. This line returns control to your program by branching to the address stored in register 14.

7. This statement must be the last line in the subroutine.

20. Listing Control Directives

One of the outputs of the assembler process is a listing of source and object codes. The assembler directives that control the format of the listing have the following functions:

- Provide headings for each page
- Eject or skip to a new page
- Space for extra blank lines
- Provide for printing or nonprinting of the output

Table 20—1 is a summary of the assembler listing control directives available to the user of the OS/3 assembler.

Table 20—1. Listing Control Directives

Directives	Basic Function	Where Discussed
EJECT	Advance listing	20.1
PRINT	Listing content control	20.2
SPACE	Leave blank lines on listing	20.3
TITLE	Listing title declaration	20.4

EJECT

20.1. ADVANCE LISTING (EJECT)

The *advance listing* (EJECT) directive causes the assembler to continue the assembly listing (Part 6, Section 28) on the top of the next printout page. The format of the EJECT directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	EJECT	unused

If the next line of the listing causes a page change, the EJECT directive has no effect.

When the EJECT directive is encountered, the printing form is skipped to the next page. If a title has been previously specified, the title is printed on the new page. An EJECT directive appearing in a source code macro definition causes the form to be skipped whenever the definition is listed and each time the macro is generated.

The assembler will advance the assembly listing to a new sheet whenever a sheet is full. However, if the programmer would like each new logical part or subroutine to start at the top of a new sheet, he can use the EJECT directive whenever he wants a new sheet to start.

The EJECT directive itself is never printed.

20.2. LISTING CONTENT CONTROL (PRINT)

The *listing content control* (PRINT) directive enables the programmer to control the contents of the assembly listing. The format of the PRINT directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	PRINT	[{ON / OFF}] [,{GEN / NOGEN}] [,{DATA / NODATA}] [,{SINGLE / DOUBLE}]

where:

ON
Specifies the listing is to be printed.

OFF
Specifies that no listing is printed.

GEN
Specifies that lines generated by a macroinstruction are printed.

NOGEN
Specifies that lines generated by a macroinstruction are not printed, except that the macroinstruction and any MNOTE or PNOTE messages generated 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.

SINGLE
Specifies that the source listing is single-spaced.

DOUBLE
Specifies that the source listing is double-spaced.

If a PRINT directive specifies OFF plus other parameters, the other specifications are not effective until a PRINT directive is encountered that specifies the listing is to be turned ON. The options provided by the PRINT directive are keyword and not positional parameters; therefore, the comma is not required if a parameter is omitted. The initial print condition of assembly printing is ON, GEN, NODATA, SINGLE. This condition remains until the first PRINT directive changes it. PRINT directives may change from only one to all of the parameters; any unspecified parameters remain in their previous condition. A PRINT directive may not appear in a macro definition.

Examples:

```
   LABEL    ΔOPERATIONΔ                    OPERAND
   1         10     16
  ------------------------------------------------------------
1.          PRINT   DATA
2.          PRINT   OFF
3.          PRINT   ON,GEN,DATA
```

1. Data is printed in full.
2. Assembly listing is suppressed.
3. Assembly list printing is restored with complete printing of data constants.

20.3. LEAVE BLANK LINES ON LISTING (SPACE)

The *leave blank lines on listing* (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 specifying the number of lines the paper is to be advanced. If no operand is coded, one line will be spaced.

LABEL	△OPERATION△	OPERAND
unused	SPACE	[i]

where:

i
: Is an unsigned decimal integer.

Examples:

```
LABEL   △OPERATION△                              OPERAND
1        10        16

1.               SPACE     6
2.               SPACE     22
```

1. The assembler advances the print form six lines before printing the next line.

2. The assembler advances the print form 22 lines before printing the next line.

TITLE

20.4. LISTING TITLE DECLARATION (TITLE)

The *listing title declaration* (TITLE) directive provides data for the heading of each page of the assembler listing and advances the printer form to a new page. The format of the TITLE directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	TITLE	'c'

where:

c
: Is a heading of up to 100 characters enclosed in apostrophes.

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.

Examples:

```
   LABEL  ΔOPERATIONΔ              OPERAND              ΔCOMMENTS
   1        10    16                                                        72
1.          TITLE ' WEEKLY PAYROLL SOURCE AND OBJECT LISTING -- ASSEMBLED Z
                  ON &SYSDATE AT &SYSTIME'
2.          TITLE ' PAYROLL SUBSECTION -- &SYSDATE'
```

1. The Z in column 72 specifies that the title is continued on the next line. At assembly time, the assembler replaces the system variable symbols &SYSDATE and &SYSTIME with the current date and time, respectively. (See Appendix G.)

2. The assembler puts the system date in &SYSDATE at assembly time.

21. Input and Output Control Directives

The OS/3 assembler input and output control directives provide the necessary control for sequence checking, formatting, and reproducing data. The directives in this section help you in writing the source code program and controlling the source code punched cards. The six directives are:

- ICTL

  Controls the format of the program instructions.

- ISEQ

  Controls the sequence of the punched cards in the source deck.

- REPRO

  Controls the production of linkage editor control statements in the object module.

- PUNCH

  Produces a specified record at assembly time.

- COPY

  Controls the inclusion of prefiled source statements into your source programs.

- CCW

  Initiates input and output operations.

ICTL

21.1. INPUT FORMAT CONTROL (ICTL)

The *input format control* (ICTL) directive specifies new values for the begin, end, and continue columns. Normally, a source statement begins in column 1 of the coding form and ends in column 71. If a continuation statement is needed, a character is written in column 72, and the statement continues in column 16 of the following line. The format of the ICTL directive is:

LABEL	△ OPERATION △	OPERAND
unused	ICTL	[b] [,e] [,c]

where:

b

Is an unsigned decimal integer specifying the beginning column. It must be between 1 and 75.

e

Is an unsigned decimal integer specifying the ending column. It must be greater than or equal to b+5 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 than e. The line is continued starting in the column specified by c.

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. If e is specified and e is less than 80, a continuation statement is signalled by putting a nonblank character in column e+1 of the line to be continued.

There can be only one ICTL directive in a source code module and it must immediately precede or follow any program-defined macro definitions. The ICTL directive applies only to those source statements that follow it. All library macro definitions are assumed to have normal output format. If the ICTL appears before the START card and it is incorrect, the assembly is terminated. When an ICTL appears out of sequence (must be first card following START card), the ICTL terminates the assembly.

Examples:

```
LABEL   ΔOPERATIONΔ                     OPERAND
1        10        16

1.       ICTL      2,79,1Ø
2.       ICTL      2,,16
```

1. Coding is to follow a new format by starting in column 2, ending in column 79, and continuing on the following line in column 10.

2. Coding is to follow standard format except that it is to start in column 2.

ISEQ

21.2. INPUT SEQUENCE CONTROL (ISEQ)

The *input sequence control* (ISEQ) directive informs the assembler which columns of the source statement contain the field used for checking the sequence of statements and controls the initiation and termination of sequence checking. The format of the ISEQ directive is:

LABEL	ΔOPERATIONΔ	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 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 macro definitions or on statements inserted into the source code via a COPY directive.

If no ISEQ directive is supplied, no sequence checking occurs.

Example:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10        16
-------------------------------------------------------
          ISEQ      75,79
```

Input record sequence is to be checked using the sequence numbers found in columns 75 through 79.

REPRO

21.3. REPRODUCE FOLLOWING RECORD (REPRO)

The *reproduce following record* (REPRO) directive is used to reproduce a record in its entirety (columns 1 through 80) during assembly time. This directive is used to produce statements to precede or succeed the object module and eliminates the necessity of manually inserting them. The format of the REPRO directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	REPRO	unused

This directive causes the contents of the following source record to be reproduced as a record in the assembler output. Each REPRO directive produces one record; up to 80 bytes are reproduced.

A REPRO directive prior to the first control section of the program produces records prior to the first control section.

All REPRO directives following the declaration of the first CSECT (START) produce records which appear after the object module transfer record. Although this directive may be included anywhere in the program, it cannot be used before a macro definition.

No substitution for variable symbols occurs in the record thus produced.

Example:

```
LABEL    ΔOPERATIONΔ                       OPERAND
1         10         16
------------------------------------------------------------
          .
          .
          .
          REPRO
          INCLUDE XYZ,USERLIB
          .
          .
          .
          START      Ø
          .
          .
          .
          END        BEGIN
```

PUNCH

21.4. PRODUCE A RECORD (PUNCH)

The *produce a record* (PUNCH) directive produces a record at assembly time. This directive is used to produce job control card images to precede or succeed the object module; it eliminates the necessity of manually inserting them. 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 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 must be used to separate fields.
- In counting characters for the limit of 80, a pair of apostrophes or ampersands written to express a single apostrophe or ampersand counts as one character.

A PUNCH directive prior to the first control section of the program produces records prior to the first control section, and all others produce records after the last control section.

Although this directive may be included anywhere in the program, it cannot be used before a macro definition.

Variable symbol substitution is performed within the operand field.

Example:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10         16
-------------------------------------------------------------
          PUNCH      'INCLUDE XYZ,USERLIB'
```

The record *XYZ* is included from *USERLIB* at assembly time.

COPY

21.5. INCLUDE CODE FROM A LIBRARY (COPY)

The *include code from a library* (COPY) directive causes the source module identified in the operand field of the COPY directive to be included directly into the source program being assembled. The format of the COPY directive is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	COPY	symbol

where:

symbol
Identifies the source module to be copied by the assembler. Only one symbol may be used.

The assembler places the source code, identified by the operand, immediately after the COPY directive. This source module may not include any COPY, END, ICTL, MACRO, or MEND directives. Also, the last statement in the source module may not be continued into the source program being assembled. Statements included in the program by a COPY directive are assumed to be in standard format regardless of any ICTL directives in the program.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10        16
-----------------------------------------------------------------
          COPY      SUBRUT
```

SUBRUT is copied from a source library and placed into the calling program.

21.6. CHANNEL COMMAND WORD (CCW)

The *channel command word* (CCW) defines an 8-byte field aligned on a double-word boundary and located in main storage. The CCW is used to initiate I/O operations such as reading and writing. It has four operands which specify the contents of the channel command word. Each operand is separated by a comma and all four operands must appear in the operand field.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CCW	op_1,op_2,op_3,op_4

where:

op_1
: Is an absolute expression that specifies the command code. The command code defines the I/O operation to be performed. This value is right-justified in byte 1.

op_2
: Is an expression that specifies the address of the first byte of data in main storage to be controlled. This value is located in bits 13 through 31. Bits 8 through 12 are set to zero.

op_3
: Is an absolute expression that specifies the flags for bits 33 and 34, and zeros for bits 32 and 35 through 47. Flag bits are set if a specific option is being used.

op_4
: Is an absolute expression representing the byte count which specifies the number of bytes to be controlled. This value is right-justified in bytes 7 and 8.

Operational Considerations:

- If a symbol is used in the label field, it references the address of the leftmost byte of the CCW. Its length attribute is eight.
- All four operands must be specified.
- For more detailed information on the use of the CCW, see the processor programmer reference manual.

Example:

```
LABEL   ΔOPERATIONΔ                      OPERAND
1        10        16
------------------------------------------------------------
         CCW       2,INAREA,X'80',80
CCW1     CCW       X'03',LOC+24,X'90',55
         CCW       5,8,X'00',128
```

PART 5. BAL MACROS

22. Macro Facility

22.1. THE MACRO PROCESSOR

The OS/3 macro facility processes macro call instructions. This macro processor functions somewhat like a compiler and provides BAL users with a higher level basic assembler language. The OS/3 macro call instructions that make up this language are stored in the macro library file (YMAC). Each macro call instruction provided by Sperry Univac (data management, sort/merge, etc) generates an open subroutine each time it is used in a program. An open subroutine is a set of BAL source instructions, designed to perform a particular function, that must be inserted into a program at each place desired. (The set of BAL instructions that make up an open subroutine is also called *inline expansion code)*. The macro facility expands the OS/3 macro definitions from YMAC and inserts them into a program in place of a macro call instruction.

Although the macro processor is far from being a high-level language compiler like COBOL or FORTRAN, it has language statements that must be interpreted and reduced to machine instructions just like any compiler. We no longer have an assembler that just converts one source instruction to one machine instruction. We now have an assembler that accepts one source statement in the form of a macro call instruction and converts this one statement into as many BAL source instructions as required to perform the particular function.

The macro processor is a valuable tool for the BAL programmer. Any programmer who writes his programs in assembly language quickly discovers the existence of macroinstructions (colloquially known as *macros*). Most programmers use data management macros to define their files and to process them (OPEN, DMINP, DMOUT, etc). These macros were created for a specific reason. For example, to open a file might take 15 instructions in a particular sequence. If you want to open five files, you have to code this 15-instruction sequence five times. The only differences in these instructions that you would have to code would be the instruction parameters that generally vary from file to file. To avoid this boring and repetitive process, which also provides opportunity for making coding errors, data management macroinstructions are provided for your use to define and process the necessary instruction sequences. These sequences are known to be error free, and you can generate your specific instruction sequences merely by filling in parameters in two or three lines of coding.

If you are an experienced BAL programmer, you should be familiar with macroprogramming because, to write any type of worthwhile BAL program, you have to use data management macroinstructions. This means you've studied the data management instructions, and possibly others, and used them when writing BAL programs. You know what inline expansion code looks like because you are a user of macro calls and have seen inline code in your source listings.

For instance, look at the listing shown in Figure 22—1. This program has five macro call instructions, and the inline code that immediately follows each call instruction is marked with a plus sign. The inline code shown in this listing is generated via macro call instructions designed by Sperry Univac. Each call is designed to produce a sequence of source instructions that will perform a specific function. The CDIB macroinstruction, shown in Figure 22—1, is designed to generate all the DC statements required to define a file for data management. If a CDIB didn't exist, you would have to code all the DC statements needed to create a printer file. The purpose of this part of the user guide is to teach you how to become more proficient at using macro call instructions. We are going to teach you how to design your own macros, not how to call macros. You can learn about the macroinstructions Sperry Univac provides by reading the related user guides. If you are experienced in macro design, you would be better off referring to the assembler programmer reference, and not this user guide, because the discussion in this part is meant for novice macro designers.

22.2. MACRO SOURCE CODE

Although you've probably seen a lot of inline expansion code, chances are you have probably never seen macro source code. Inline expansion code originates from macro source code. Whenever you use an OS/3 macro call instruction, the macro facility retrieves the macro source code from YMAC, which contains a macro definition for each macro call instruction provided by Sperry Univac. Each macro definition holds the BAL source instructions that are to be generated inline. If you have a BAL program that has become popular and is recurring in other programs and you want to make this code available via a macro call instruction, you have to design a macro definition. You use the statements provided with the assembler macro facility to transform your BAL program into a macro definition. It is the responsibility of the macro facility, which is part of the assembler, to process your macro definition. The macro facility works entirely with macro source code while a conventional assembly recognizes and processes program source code. Macro source code consists of macro facility source statements and BAL source statements.

```
LOC.   OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT

000000                                 1            START 0
000000 0560                            2 BEGIN      BALR  6,0
000002                                 3            USING *,6
000002 DC06 607F 6086 00081 00088      4 EXAMPLE    TR    INDEX,TABLE
000008 D006 607F 6086 00081 00088      5 EXAMPLE2   TRT   INDEX,TABLE
                                       6            OPEN  OUT,(OUTRIB)  <------ MACROINSTRUCTION
00000E 0700                            7+           CNOP  0,4
000010 4510 601A             0001C     8+           BAL   1,*+12
000014 81                              9+           DC    X'81'
000015 000048                         10+           DC    AL3(OUT)
000018 80                             11+           DC    X'80'
000019 000074                         12+           DC    AL3(OUTRIB)
00001C 0A26                           13+           SVC   38 ISSUE SVC
00001E D206 60AA 607F 000AC 00081     14            MVC   BUF(7),INDEX
                                      15            DMOUT OUT,BUF
000024                                16+           DC    0Y(0) *                  SET ALIGNMENT
000024 5810 60BE             000C0    17+           L     1,=A(OUT) *             LOAD R1$, CDIB ADDRESS
000028 5800 60C2             000C4    18+           L     0,=A(BUF) *             LOAD R0$, WORKAREA ADDRESS
00002C 9220 1002       00002          19+           MVI   2(1),X'20' *           SET FUNCTION CODE
000030 9200 1003       00003          20+           MVI   3(1),0 *             SET FUNCTION CONTROL BYTE 1
                                      21+           SCALL 47
000034                                22+           DS    0H
000034 0AEF                           23+           SVC   239
000036 10                             24+           DC    YL1(16)
000037 2F                             25+           DC    YL1(47)
000038 0A19                           26+           SVC   25
00003A 0700                           27+           NOPR  0
00003C 0A1C                           28+           SVC   28
00003C                                29+           ORG   *-2
00003C 0A85                           30+           SVC   133
                                      31            CLOSE OUT
00003E                                32+           DC    0Y(0)
00003E 5810 60BE             000C0    33+           L     1,=A(OUT) LOAD R1 WITH FILENAME ADDRESS
000042 0A27                           34+           SVC   39 ISSUE SVC
                                      35            DUMP
000044                                36+           DS    0H
                                      37+*
                                      38+*  THE DUMP PARAMETER IS A 1-4 BYTE HEX CODE TO BE DISPLAYED BY DUMP
                                      39+*
000044 1700                           40+           XR    0,0 CLEAR DUMP CODE
000046 0A1B                           41+           SVC   27 DUMP SVC
                                      42 OUT        CDIB
                                      43+           ENTRY OUT
000048                                44+           DS    0F
000048 112C                           45+OUT        DC    X'112C' *                 CDIB ID AND LENGTH
00004A 0000                           46+           DC    2X'0'
00004C D6E4E34040404040               47+           DC    CL8'OUT' *                FILENAME
000054 0000000000000000               48+           DC    8F'0'
                                      49 OUTRIB     RIB   IOA1=BUF,BFSZ=16
                                      50+           PRINT OFF
                                     566+           PRINT ON
```

Figure 22—1. Example of Inline Macro Expansion

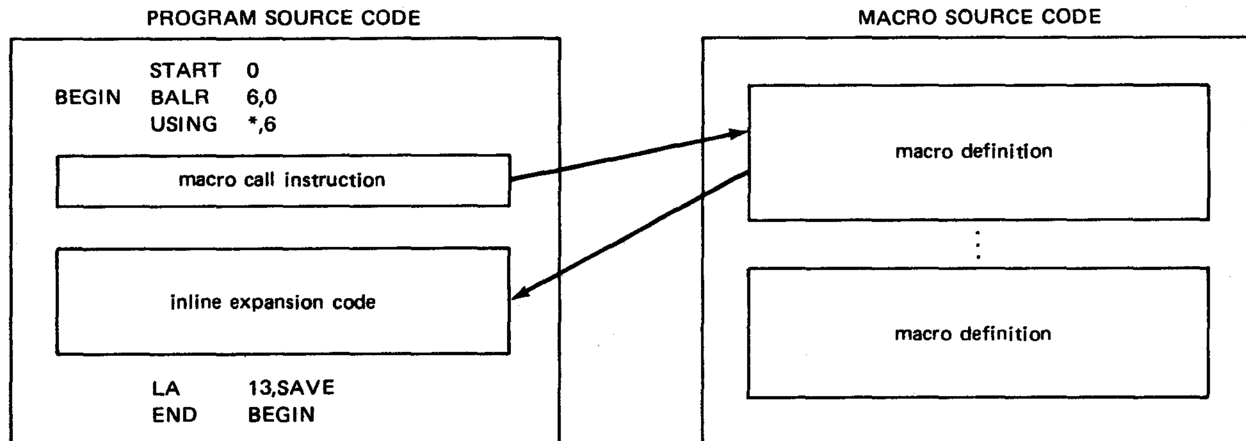
There are three types of source code that are always associated with the macro facility:

1. Macro source code

2. Macro call instruction

3. Inline expansion code

The order in which these different types of source code are listed is the order of their evolution. First, you must have a macro definition (*macro source code)* before you can use a *macro call instruction* to generate *inline expansion code*. An important fact to keep in mind is that all of this code is source code. The macro facility works entirely at the source code level, from the macro call instruction, to the YMAC, to the inline expansion code. The following diagram shows the interactions of each type of code when a macro call instruction is used.



The macro facility performs preassemble processing. It has nothing to do with turning source code into object code. The basic function of the macro facility is to search for the proper macro definition when a macro call instruction is used in a program, and generate the requested inline expansion code. This is done before the assembler starts creating an object module. When the assembler detects a pseudo-operation code (a mnemonic code that is not a machine instruction), that code is turned over to the macro facility. Each macro definition has a unique *call-name* that is identified in the operation field of the macro call instruction. The macro facility searches for the macro definition that matches the *call-name* and generates the requested inline expansion code. The macro facility expands the code inline before the assembler starts converting the program source code to object code.

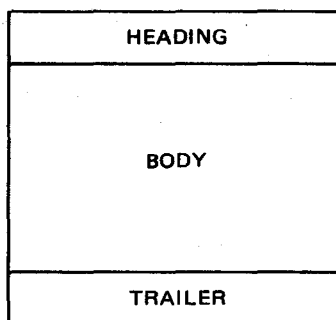
The macro facility has capabilities other than just inserting the BAL source instructions that are contained inside a macro definition inline in place of the macro call instruction. There are other elements of the expansion that you can control. You can use the macro facility to perform variable parameter replacement and variable inline expansion code. If you've used macro call instructions supplied by Sperry Univac, then you are familiar with positional and keyword parameters. The values you code as positional or keyword parameters in the call instruction replace variable symbols coded in the macro definition. Using variable parameter replacement, you can use the values given in the call instruction to replace variable symbols coded in the *label, operation,* or *operand* field of any BAL instruction in the macro definition.

Variable inline expansion code is another level of control that allows you to design a macro definition that will vary the pattern of BAL instructions generated from within the macro definition. Conditional assembly language statements are used to design the logic for variable inline expansion code. Variable parameter replacement and variable inline expansion code allow you to give the user of your macro call instruction more control over the code that is generated. If you use these coding techniques when designing macro definitions, the user can control calculations performed by the open subroutine and select the functions that are to be performed.

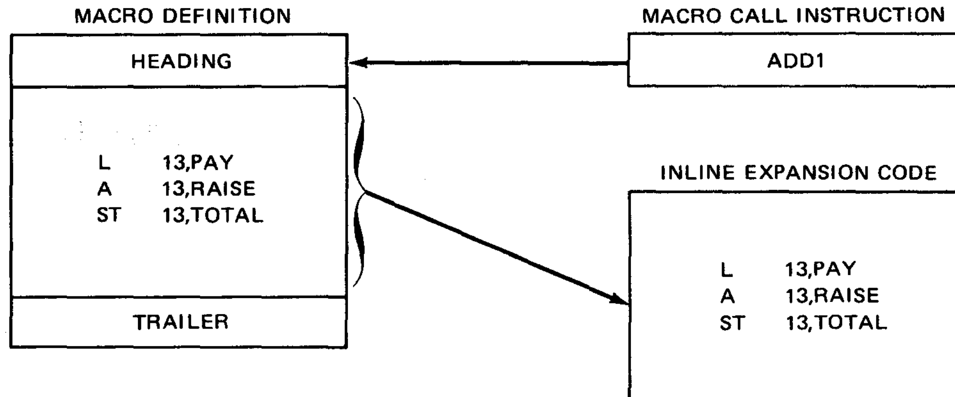
23. Macro Design

23.1. THE MACRO DEFINITION

You can define your own macro call instructions by using the statements provided with the assembler macro facility. When you define a call, it must be in a formalized pattern called a *macro definition.* Each macro definition is organized into the following parts:

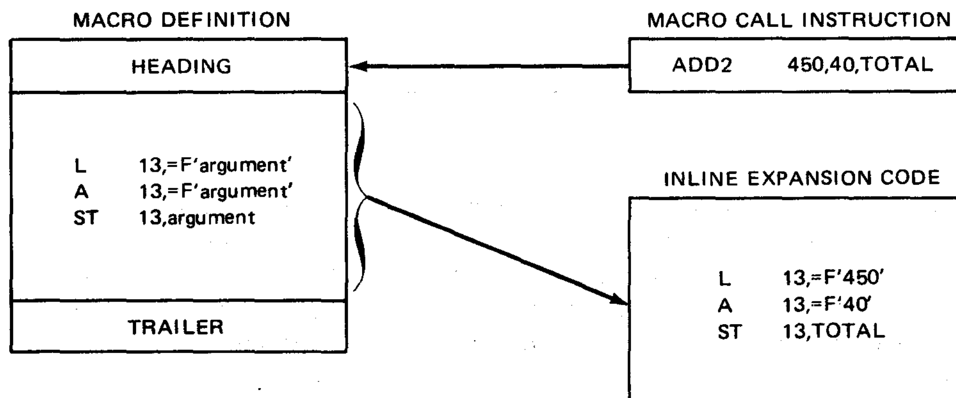


The two major areas to consider when designing a macro definition are the heading and the body; the trailer is merely a single statement indicating the end of the macro definition. The heading is always the first part of the macro, and it consists of statements you use to design the macro call instruction. The body, which immediately follows the heading, is where you design the inline expansion code. Model statements are used in the body to construct a model of the inline expansion code you want generated by the macro call instruction. If you design a *basic* macro definition, the model statements are merely a copy of the BAL source instructions that are to be expanded inline. A basic macro definition is one that does not require any parameters from the macro call instruction and will generate the same sequence of source instructions, with no modifications, each time it is called. The following diagram shows the operation of a basic macro definition.



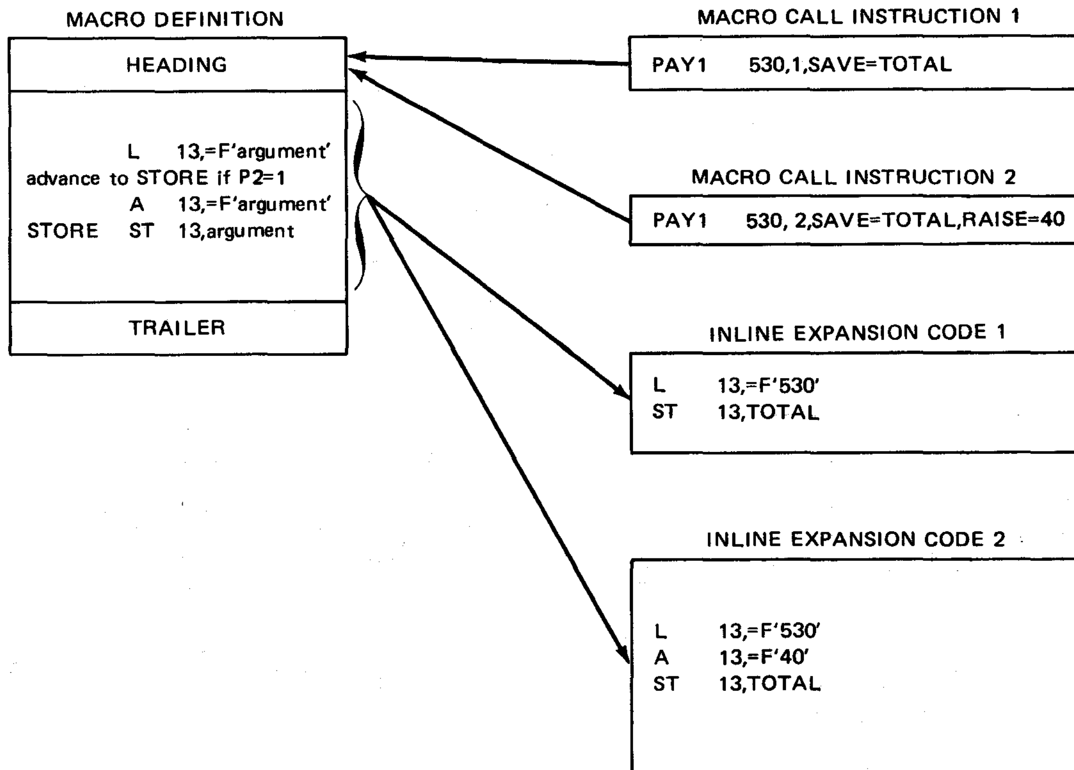
The only real design consideration for the basic macro definition is the *call-name*, which is the mnemonic that appears in the operation field of the macro call instruction. You must give the *call-name* in the heading.

If you want variable parameter replacement in the body, you must give further consideration to coding the heading and body. Variable parameter substitution is substituting parameter values coded in the macro call instruction in place of arguments given in the *label, operation,* or *operand* fields of model statements. The following diagram shows the operation of a macro definition designed to perform parameter substitution.



You must design the body to indicate where the arguments are and design the heading to indicate how the parameter values are to be coded in the macro call instruction and how the body is to reference the parameter values in the macro call instruction.

If you want variable inline expansion code, you must include model statements other than the BAL source statements that you want expanded inline. These other model statements are called conditional assembly language statements, and they enable you to vary the pattern of the inline expansion code produced by the macro definition. The pattern of code generated depends on a value given in the macro call instruction. The following diagram shows the operation of a macro definition designed to perform variable inline expansion code.



Parameter 2 in the call indicates which pattern of inline expansion code is to be generated. The second statement in the body is a conditional assembly that tests the value of parameter 2. If the value of parameter 2 is 1, then a branch is made to STORE and the A instruction is not included in the inline expansion code. If the value of parameter 2 is 2, no branch is made and all of the instructions are generated.

23.2. MACRO DEFINITION STORAGE

After you've designed a macro definition there are several things you can do with it. If it is for your program and not to be used by anyone else, you can put it in your source program when it is assembled. A macro definition is placed in the source program immediately following the start-of-data (/$) job control statement and before the START assembler directive (Figure 23—1). The macro definition is stored in the temporary job run library file (YRUN), and it is only available during execution of the current job. To make a macro definition available to anyone at anytime, it must be stored in a library other than YRUN (either YMAC or your own library). Figure 23—2 shows how a macro definition is obtained from YMAC. To add a macro definition from cards to a disk file, you can use the ELE librarian control statement. (For more information on the system library and creating library files, see the system service programs (SSP) user guide.) Whether a macro definition is stored in a library or is part of a source program depends on whether you want the macro to be temporary or permanent.

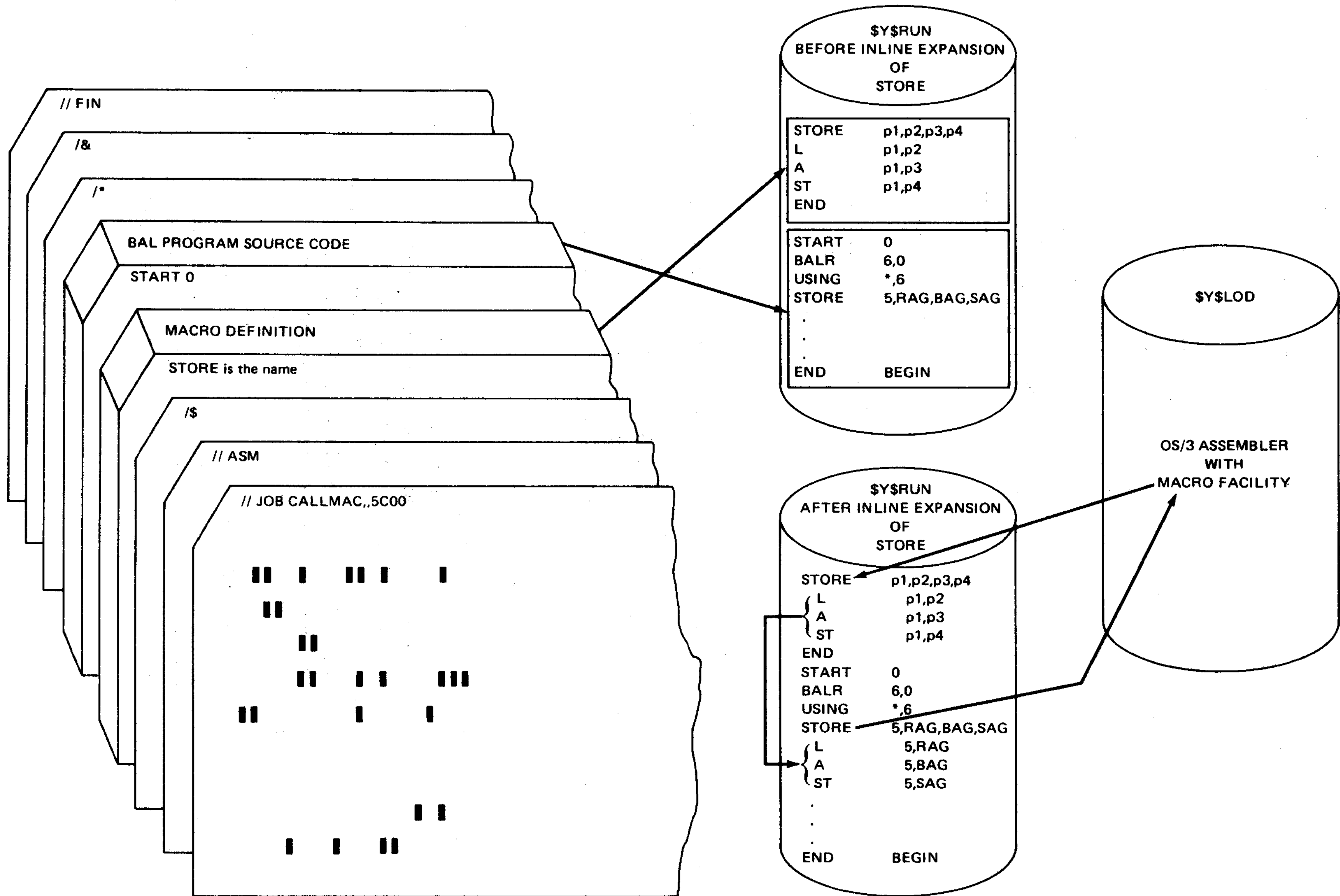
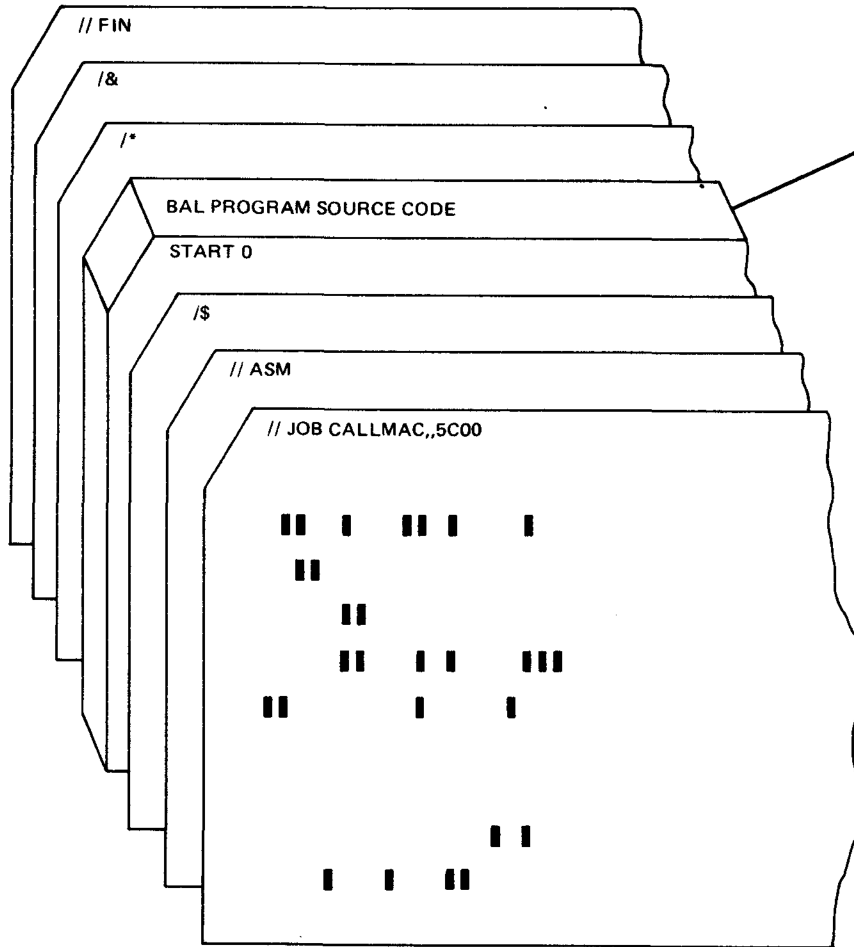
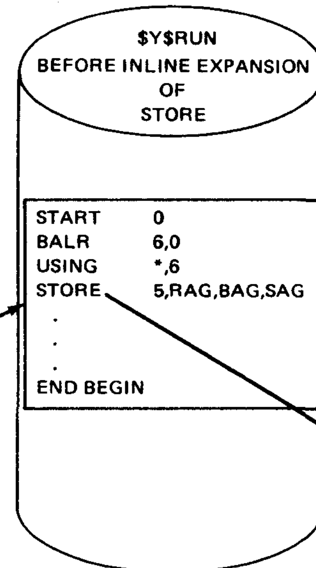


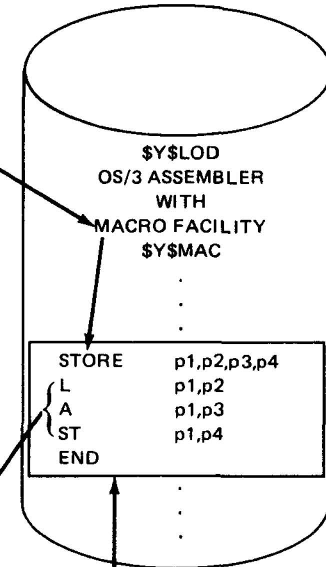
Figure 23—1. Accessing a Macro Definition Submitted in the Source Deck

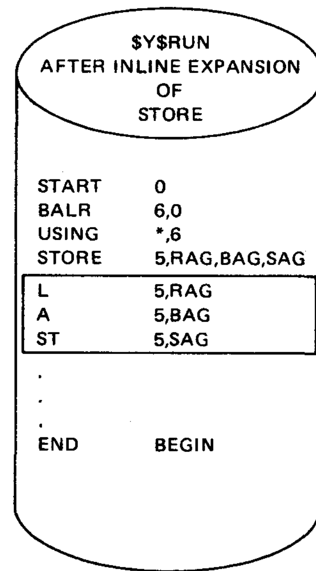
SOURCE DECK



SYSRES DISK PACK





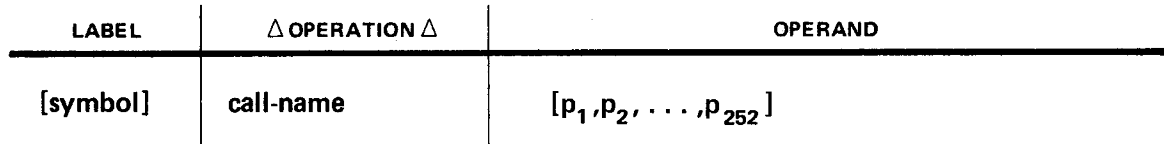


THIS CODE COULD
ALSO EXIST IN
A USER LIBRARY

Figure 23—2. Accessing a Macro Definition Stored in a Library

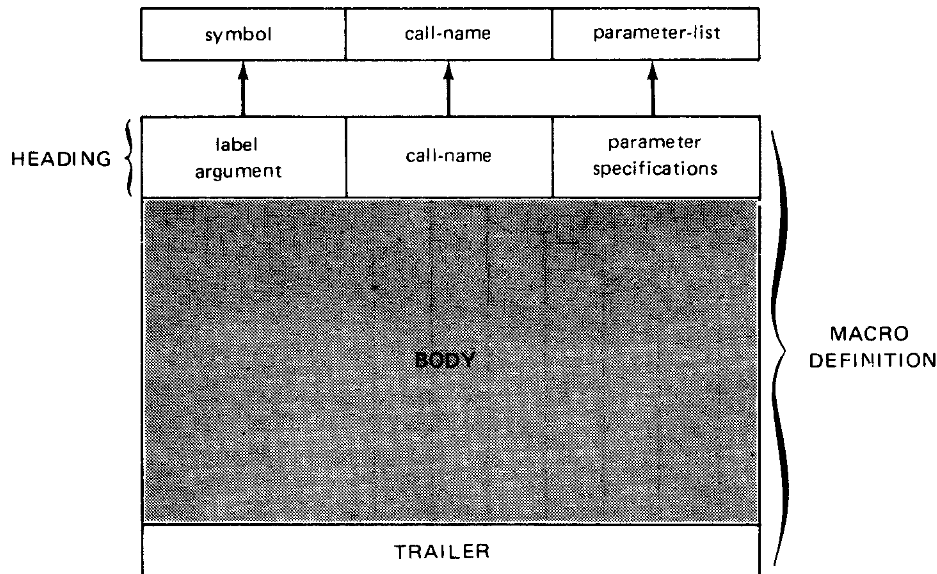
23.3. THE MACRO CALL INSTRUCTION

The macro call instruction has two functions: to call a macro definition and to pass parameters to the macro definition, if required. The *label* and *operand* fields of the macro call instructions are used for passing parameters, and the *operation* field is used to call the macro definition:



LABEL	△OPERATION△	OPERAND
[symbol]	call-name	$[p_1,p_2,\ldots,p_{252}]$

Just as the inline expansion code originates from the macro definition body, the macro call instruction originates from the heading. The contents of the *label, operation,* and *operand* fields of the macro call instruction are specified in the heading of the macro definition. You actually use the heading of the macro definition as a dummy call line to design the format of the macro call instruction. The following diagram shows an abstract representation of how the heading represents each field of the macro call instruction. (The fields shown in the diagram of the heading do not necessarily appear exactly where they are shown; the diagram is provided to illustrate the association the heading has with the call. The exact format of the heading is described in the following discussion.)



Everything you want the user to code in the macro call instruction must be represented in the heading. If you design a *basic* macro definition, the only thing that you require from the macro call instruction is the *call-name* in the operation field. The *call-name* that you want the user to use to call your basic macro definition is duplicated in the heading. All *call-names:*

1. Must begin with an alphabetic character or special letter.

2. Must not exceed eight characters in length.

3. Must not contain embedded blanks or other special characters.

4. Must not be the same name as any of the Sperry Univac mnemonic operation codes or any mnemonic operation codes you have in your library. This restriction is normally true unless you use the OPSYM directive to override a valid mnemonic code (16.2).

A macro definition that is designed to perform variable parameter replacement must indicate, in the heading, the type and number of parameters to expect from the macro call instruction. Values may be passed to the macro definition from the *label* field or the *operand* field of a macro call instruction. If you want to use the symbol in the *label* field of a call instruction, you must have a *label-argument* in the heading. Before the macro definition can pick up the values from the *parameter-list* in the macro call instruction, they must be coded according to the *parameter-specifications* given in the heading. You use the heading to specify how many parameters are to be coded in the macro call instruction and whether the parameters are keyword or positional, or both. The macro call instruction must be coded in accordance with the heading before the values in the *parameter-list* can properly replace the arguments in the body.

The heading can specify a *parameter-list* of up to 252 parameters. All parameters must be separated by commas and each parameter can be from 0 to 127 characters in length. In order to be properly picked up by the macro definition, the string of characters comprising a macroinstruction operand must satisfy the following conditions:

- May include one or more sequences of characters enclosed in single apostrophes. The apostrophes enclosing each character sequence are paired. Paired apostrophes may appear within paired apostrophes.

- May include a single apostrophe outside paired apostrophes if written as part of the following sequence: any special character except an ampersand, the letter L, an apostrophe, and a letter.

- May include an ampersand as the first character of a variable symbol if the ampersand is a single ampersand or the last ampersand of a string containing an odd number of ampersands.

- May include paired parentheses outside paired apostrophes. To determine pairing, a left parenthesis is paired with the immediately following right parenthesis (that is, no parentheses between them). Additional pairs are determined by ignoring the first pair and reapplying the rule.

- May include an equal sign only as the first character of an operand or within paired parentheses or paired apostrophes.

- May include a comma as a character in a string if the comma is enclosed in paired parentheses or paired apostrophes. A comma standing alone is interpreted as the end of an operand.

- May include a blank within paired apostrophes. A blank not enclosed in apostrophes terminates the operand field.

NOTE:

Operands can be coded on more than one line through the use of a continuation character in column 72. If a line is to be continued, the last operand on that line must be followed by a comma. A warning message is issued if a comma is not included.

The specifications for the *parameter-list* of a macro call instruction should be thoroughly documented for the user of the macro. He should know the range of values each parameter is to have and the type of parameters.

There are two ways the macro definition can recognize values in the *parameter-list*: by the position of the value in the list or by the name associated with the value in the list. A value identified by the position it holds in the list is called a *positional* parameter, and one that is identified by a name is called a *keyword* parameter. When the parameter specifications in the heading indicate that the *parameter-list* is to contain only positional parameters, the corresponding values in the operand field of the macro call instruction must appear in the same operands each time the call is used. If any positional parameters are omitted in the call, this omission must be indicated by retaining the comma in the parameter's place. For instance, if a macro has the capacity to accept four positional parameters, the call doesn't necessarily have to give all the parameters because some of the parameters may be optional to the macro's function. The proper coding of some of the possible combinations for four positional parameters is:

```
LABEL   ΔOPERATIONΔ                        OPERAND
1        10        16
-----------------------------------------------------------
NAME1    CALL      P1,P2,P3,P4
NAME2    CALL      ,P2,P3,P4
NAME3    CALL      P1,,P3,P4
NAME4    CALL      P1,P2,,P4
NAME5    CALL      P1,P2,P3,
NAME6    CALL      P1,,,P4
NAME7    CALL      P1
```

It is not necessary to retain the commas for trailing positional parameters; if you are not going to code any of the remaining positional parameters in a call line, you do not have to code the commas for those parameters. But it doesn't matter if you do code the commas. I could have left out the last comma for CALL in NAME5 or I could have left the commas in for CALL in NAME7.

The following are typical examples of positional parameters with their calls:

```
READIN   DMINP    INFILE,WORK1
WRIT1    DMOUT    PRINTER,HEDR
ENDIN    CLOSE    INFILE
ACTION   OPR      TEXT,,,REPLY,AREA
```

Keyword parameters, unlike positional parameters, are not referenced by the position they hold in the call line, but by the name of the keyword in the call line. A keyword parameter consists of three parts: a keyword, an equal sign, and a value:

keyword=value

The keyword is an alphanumeric string from one to seven characters in length. Actually, a keyword may be longer than six, but the macro facility recognizes only the first seven characters as the true name and truncates the remaining characters. The value is dependent upon the arguments in the macro.

Keyword parameters, like positional parameters, must be separated by commas but aren't restricted to being in the same position each time they are coded. This nonpositional characteristic eliminates the inconvenience of comma counting because keyword parameters can be coded in any order and, when a keyword is omitted, a comma does not have to be retained in the parameter's place. Keyword parameters also provide another coding choice: default values for omitted parameters. The macro may be designed to automatically provide a preselected value for a keyword parameter that is omitted from a call line. To illustrate, suppose we had a keyword parameter called CHOICE. This parameter could be assigned optional values, such as ONE, NONE, or ALL. We could then write the parameter in the macro with a default value equal to ALL. Now when we call the macro, the ALL option is used unless the call specifically states another parameter such as CHOICE=ONE or CHOICE=NONE. This is true even if we omit the keyword parameter in the call. However, if a keyword is not given a default value within a macro and it is omitted in the call, it receives the value of a null character string.

Following are examples of typical macros with keyword parameters:

LABEL	ΔOPERATIONΔ	OPERAND
TAPRIB	RIB	TYPEFLE=OUTPUT,FILABL=STD, RCFM=FIXBLK,RCSZ=220,IOA1=TAPAREA, WORK=YES
PRINTRIB	RIB	RCFM=FIXUNB,BFSZ=120,IOA1=LIST,WORK=YES, PRAD=2,PRINTOV=YES

Both positional and keyword parameters may appear in the same call line. This is known as a *mixed-mode* macro call. In a mixed-mode macro call, the positional parameter string must be coded before the keyword parameter string:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	call-name	$p_1,\ldots,p_n,k_1,\ldots,k_n$

Positional and keyword parameters may have a subordinate list of parameters called *subparameters*. This sublist of parameters permits the macro call line to provide more than one value in a single parameter position. A sublist for positional parameters is coded as follows:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	call-name	$(p_{1,1},p_{1,2},...,p_{1,n}),...,(p_{n,1},...,p_{n,n})$

A sublist for keyword parameters is coded as follows:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	call-name	$k_1=(p_1,...,p_n),...,k_n=(p_1,...,p_n)$

The parameter sublist must always be enclosed by parentheses, and the subparameters are coded as positional parameters (parameter omission is indicated by retaining commas).

When you design a macro definition that requires parameters from the macro call instruction, the heading is the means to pass values from the call to the body. There are two methods available for designing the call-to-heading-to-body communications cycle that generates inline expansion code. You can use a macro definition in PROC format or a macro definition in MACRO format. Each uses a different technique in the heading for designing the call instruction. If a SET symbol appears in the operand entry of a macroinstruction, attribute information is not provided and the operand may not be accessed as a sublist.

24. Two Types of Macro Definitions

24.1. PROCS AND MACROS

The OS/3 macro facility can process two types of macro definitions. One type is called a *procedure* (PROC) and has been the standard type of macro definition for Sperry Univac systems for many years. The other type is called a *macro* (MACRO) and is available primarily to be compatible with the IBM 360/20 system. You can design a macro definition in PROC format or MACRO format; the OS/3 macro facility will accept and process either one. Although the statements inside a PROC are a little different from those inside a MACRO, both types of definitions will always have a heading, a body, and a trailer and will always be in that order. Actually, the only differences between the two types of definitions occur in the heading and the trailer. The body, which contains the model statements, is the same for both PROCs and MACROs. Let's take a look at a macro definition heading in MACRO format and compare it to the same macro definition heading in PROC format and discuss their differences without going into a lot of detail about the operation of each. Look at Figure 24—1 and you'll see the differences between the headings for the PROC and the MACRO.

PROC HEADING

	LABEL	△ OPERATION △	OPERAND
PROC STATEMENT	label-argument	PROC	parameter-specifications
NAME STATEMENT	call-name	NAME	pos-O

MACRO HEADING

	LABEL	△ OPERATION △	OPERAND
MACRO STATEMENT	unused	MACRO	unused
PROTOTYPE STATEMENT	label-argument	call-name	parameter-specifications

CALL INSTRUCTION

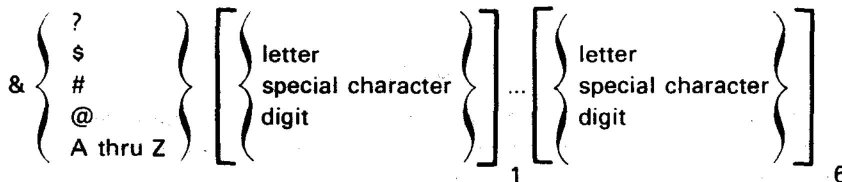
LABEL	△ OPERATION △	OPERAND
symbol	call-name	parameter-list

Figure 24—1. PROC and MACRO Heading

24.2. CALL INSTRUCTION DESIGN

Even through the format for the PROC and MACRO headings is different, it doesn't affect the coding rules for the call instruction format. When a user issues a call instruction, he is completely unaware of whether the call communicates with a PROC or MACRO. This is because both headings have the same counterpart fields that represent each field of the call. The *label-argument* in each heading represents the *symbol* in the label field of the call instruction, the *call-name* in each heading is the *call-name* used in the call instruction, and the *parameter-specifications* in each heading define the specifications for the *parameter-list* in the call. The fields representing the call instruction are in different positions in the heading for the PROC and MACRO, but they serve the same function for each.

Variable symbols are used to create the label-argument and parameter-specifications in the PROC and MACRO heading. The variable symbol is a macro language symbol used as a dummy argument in a macro definition. It's a dummy argument because the variable symbol will be replaced with a value when the macro definition is called by the macro call instruction. Since a variable symbol is recognized and processed only by the macro facility, it is distinct from the symbols used in program source code. The macro facility requires that any symbol to be used as a variable symbol must have an ampersand (&) as an identifier in the first character position. There are seven character positions after the ampersand that are used to construct the variable symbol. Because a variable symbol must always be identified with an ampersand in the first character position, a variable symbol will always be at least two characters in length (an ampersand and a character) and eight characters at the most (an ampersand and seven characters). The character position after the ampersand can contain a letter (A through Z) or a special letter (?$#@) and each of the remaining seven positions can contain a letter, special character (see 2.4 for character types), or a digit (0 through 9):



Some examples of legal variable symbols are:

&ABCDEFG
&#BCDE67
&$6
&@

Illegal variable symbols:

ABCDEFGHI
&&CDEFGHI
&=KEY
&95&95
&

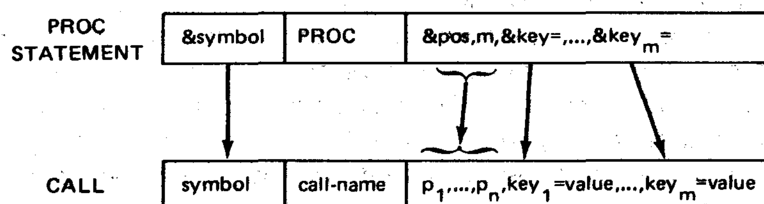
When variable symbols are used, the following restrictions must be considered:

- A variable symbol may not be used to generate a new sequence symbol, a SET symbol, a parameter, or a system variable symbol.
- A variable symbol may not be used in the label or operand field of an END, ICTL, ISEQ, COPY, or PRINT directive.
- A variable symbol may not be used in the operation field of a statement.
- No variable symbol replacement is performed on the line following a REPRO directive.
- Variable symbol replacement must not produce leading blanks in the label or operand fields.

The OS/3 assembler provides system variable symbols. When a system variable symbol is used in a model statement, a value is automatically provided by the macro facility. The system variable symbols available with the macro facility are given in Appendix G.

Both the PROC and the MACRO use two statements in the heading; the PROC may have more than two but must have at least two. When designing a MACRO, you use the first statement strictly for indicating MACRO format; the entire second statement, called the prototype statement, is used to design the call instruction. In a PROC, the first statement is not only used to indicate a PROC format but is also used, along with the second statement, for call instruction design. The heading of the PROC consists of the PROC and NAME statement, and the heading of a macro consists of a MACRO and prototype statement.

In MACRO format, each field of the prototype statement is used for designing each corresponding field in the macro call instruction. If you want to reference the *label* field in the call instruction, you must indicate a variable symbol for the *label-argument* in the *label* field of the prototype statement. The *call-name* for the call instruction is indicated in the *operation* field of the prototype statement, and the *parameter-specifications* for the *parameter-list* to be coded in the call instruction are indicated by variable symbols in the *operand* field of the prototype statement. The variable symbols for indicating a *label-argument* and for specifying positional and keyword parameters are coded in the prototype statement as follows:

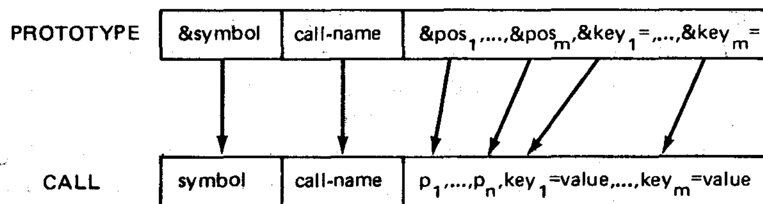


If you want the call instruction to pass values as positional parameters, variable symbols representing these positional parameters are coded starting in operand 1 of the prototype statement and are separated by commas. Each positional parameter is represented by a unique variable symbol that must be coded in the prototype statement in the same position its corresponding values are to be coded in the *operand* field of the call. If you want a mixed-mode call instruction, you must code variable symbols representing the keyword parameters immediately following the variable symbols for positional parameters. The format of the keyword parameter variable symbol is:

&key=

where *key* is a 1- to 6-character keyword to be used in the call. The keyword in the call may be longer than six characters, but the macro facility will only use the first six characters as the keyword name. If you want the call to pass values using only keyword parameters, then you code only keyword parameter variable symbols starting in operand 1 of the prototype statement. When designing a call instruction from a prototype statement, you can match each field of the statement with each field of the call, and you can also match value in the *parameter-list* of the call with each variable symbol in the operand field of the prototype statement. The prototype statement is the dummy call instruction used for designing the macro call instruction.

Even though you can compare the prototype statement with the call instruction on a field-by-field basis, you cannot use a single statement in the PROC heading for a similar comparison. This is because you must use both the PROC and the NAME statement to design the call instruction. The *label* field of the PROC statement is used to indicate the variable symbol for referencing the symbol in the *label* field of the call instruction, and the *operand* field of the PROC statement is used to indicate the variable symbols used for referencing the *parameter-list* in the operand field of the call. The coding of *parameter-specifications* in the *operand* field of the PROC statement is different from the coding in the prototype statement:



You specify positional parameters by indicating a variable symbol in operand 1, and you indicate the number of positional parameters to appear in the call in operand 2. If you want a mixed-mode call, the keyword parameters are indicated by coding keyword variable symbols starting in operand 3. You specify keyword parameters only, by coding commas in operand 1 and 2 and then coding the keyword parameter variable symbols. The PROC statement is used to indicate the *label-argument* and the *parameter-specifications*, and the NAME statement is used for specifying the *call-name*.

In effect, what the NAME statement does is to make the *call-name* an independent entity in the PROC heading. Since the *call-name* is disassociated from the *parameter-specifications*, it becomes easy to introduce another method of parameter submission without having any effect on the variable symbols in the heading. This order parameter is positional parameter zero and is submitted in the *operand* field of the NAME statement. Each value you want positional parameter zero to have must be coded in the operand field (pos-O) of a separate NAME statement. And each NAME statement must have a different *call-name* in the *label* field. Thus, you can vary the value of positional parameter zero by varying the *call-name* used in the call instruction. At any point in time, positional parameter zero has the value of the *pos-0* coded in the *operand* field of the NAME statement that matches the *call-name* used in the call instruction. The PROC makes positional parameter zero possible through efficient use of two fields that are unused in the MACRO heading. These two fields are used in the PROC to disassociate the *call-name* from the *parameter-specifications* and split the call design responsibility between the two heading statements.

So, where the MACRO uses one statement in the heading to communicate directly with the call line, the PROC uses both statements, one statement for associating parameter replacement and another for naming the *call-name*. Using the one statement in the heading to interface the call-line instead of two, as the PROC does, seems like a straightforward way of doing things. But when you read about how to use the PROC, you'll find the split-heading characteristic of the PROC heading works to your advantage. The PROC offers one more additional parameter reference (positional parameter zero) and intrinsically allows the body to reference the parameters in the call line by the number of the position and not by the symbolic name of the position.

A complete picture of the PROC and MACRO construction compared with the call instruction format is shown in Figure 24—2. It shows the format of the heading, body, and trailer for each type of macro definition. The rules for coding the model statements in the body are the same for the PROC and MACRO. These rules are:

- The label field may contain a symbol, a variable symbol, or a sequence symbol, depending on the operation defined. Comments statements may not be created by substitution for variable symbols.

- The operation field may contain any machine, assembler, or macroinstruction mnemonic code except END, ICTL, or ISEQ.

- Either ordinary symbols or variable symbols may be written in the operand field. The size of this field may not exceed 240 characters after substitution.

- The comments field may contain any combination of characters; however, substitution for variable symbols is not performed on this field by the assembler. Comments are written in the format of the statement the model represents.

- A macroinstruction that is a model statement within a macro definition is called an inner macroinstruction, while a macroinstruction in the program source module is called an outer macroinstruction. A macroinstruction that appears in a macro corresponding to an outer macroinstruction is called a second-level macroinstruction. Macroinstructions within macro definitions are nested. The number of levels to which macroinstructions may be nested in an assembly depends upon the amount of main storage available to the assembler.

- Because COPY statements within a macro definition are processed prior to the generation of code from a macro definition, they are not considered to be model statements nor are they ever processed as such.

The trailer indicates the end of a macro definition. The mnemonic code END is used in the PROC and MEND is used in the MACRO. The *label* and *operand* fields are not used.

PROC CONSTRUCTION

	LABEL	ΔOPERATIONΔ	OPERAND
HEADING	[&symbol] call-name	PROC NAME	[&pos,n] [,$\&key_1$=,...,$\&key_m$=] [pos-0]
BODY	[{symbol / &symbol / .symbol}]	mnemonic-code . . . mnemonic-code	operands . . . operands
TRAILER	unused	END	unused

MACRO CONSTRUCTION

	LABEL	ΔOPERATIONΔ	OPERAND
HEADING	unused	MACRO	unused
	[&symbol]	call-name	[$\&pos_1$,...,$\&pos_n$][,$\&key_1$=,...,$\&key_m$=]
BODY	[{symbol / &symbol / .symbol}]	mnemonic-code . . . mnemonic-code	operands . . . operands
TRAILER	unused	MEND	unused

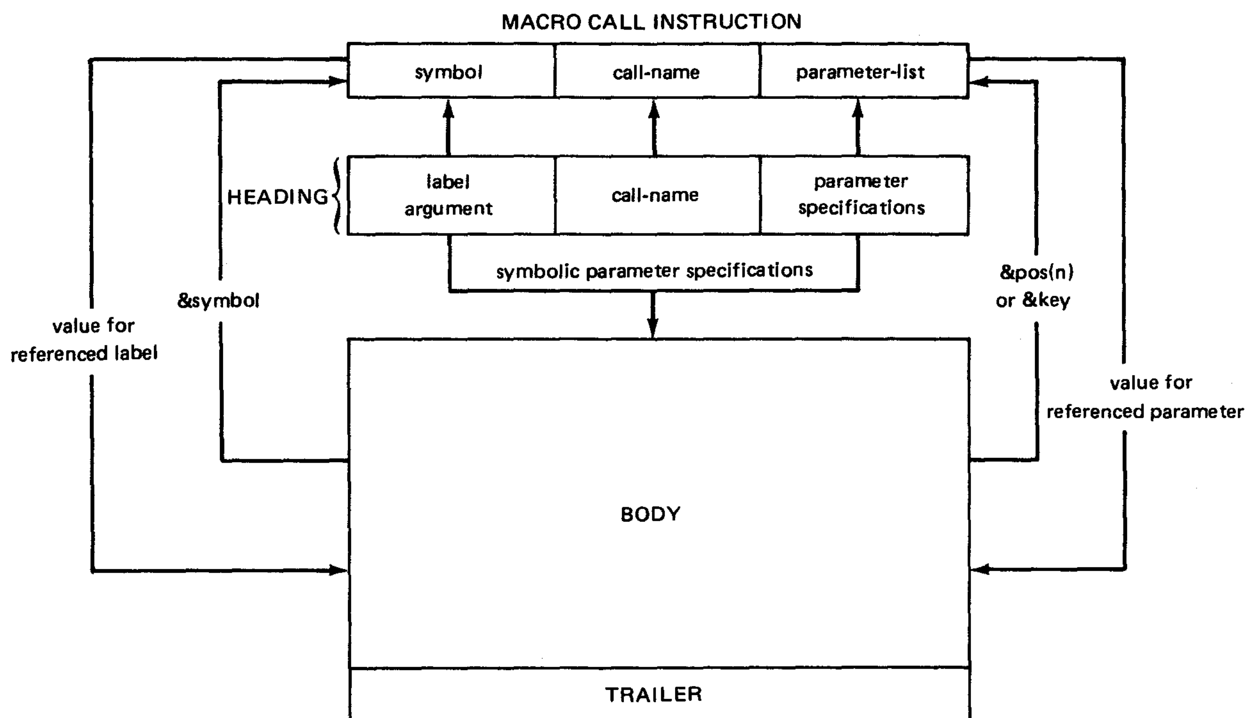
CALL INSTRUCTION FORMAT

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	call-name	[p_1,p_2,...,p_{252}]

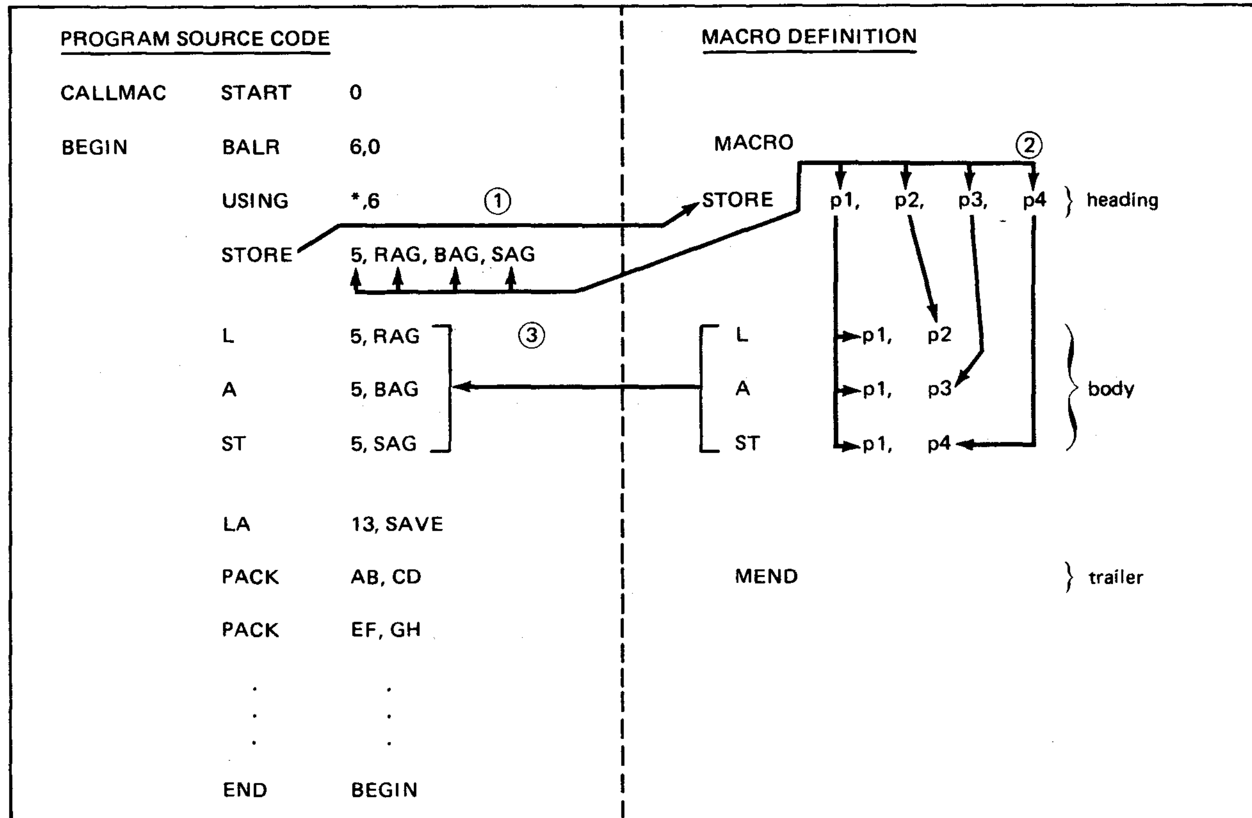
Figure 24—2. PROC, MACRO, and Call Instruction Comparison

24.3. PASSING PARAMETERS TO THE BODY

The discussion of the heading thus far has been from the call instruction design point of view, but the heading also establishes a homologous relationship between the call instruction and the body. Body references to the values in the call instruction stem from the variable symbols used for designing the call instruction. The variable symbols used in the heading to represent the *label* and *operand* fields of the call instruction are used as arguments in the body to reference the values in the call instruction. Variable symbols in the body that directly reference values in the call instruction are called symbolic parameters. When you use a symbolic parameter in the body, you are directly referencing its associated value in the call instruction and, when you call the definition, the values in the call are substituted in place of the symbolic parameters. This call-to-body communications cycle is as follows:



The heading of the macro definition thus serves not only as the design medium for the macro call instruction but also dictates how the arguments in the body are to reference the values in the call instruction. Coordination of values and arguments is accomplished by using variable symbols in the *label-argument* and *parameter-specifications* fields of the heading. The variable symbols used in the heading to represent the *symbol*, and the values in the *parameter-list*, are the variable symbols used in the body to reference the *label* and *operand* fields of the macro call instruction. Communications between a macro call instruction and a macro definition that is capable of variable parameter replacement is shown in Figure 24—3.



LEGEND:

① Call the macro.

② Substitute the parameters.

③ Inline expansion

p Variable symbol

Figure 24—3. Communication between Macroinstruction and Macro Definition

The following sequence of events occurs when a macro call is issued: The macro facility finds the macro definition called upon, matches the parameters given in the call to the variable symbols in the heading, and (wherever the body references a variable symbol in the heading), plugs the matched value into the body. Then, the body, with the substituted parameters, is put in the source program where the macro call instruction appeared. A variable symbol in a macro definition body represents either the label in an issued macro call instruction or one of the operands in that macroinstruction.

When a variable symbol appears as a symbolic parameter in a model statement, it may be concatenated (joined) with other variable symbols or characters. Some combinations of variable symbols and characters require a period as a separator between the variable symbol and the joined character to distinguish where the variable symbol stops and the concatenation begins. Other combinations do not require a period as a connector because the concatenation is implied by certain characters that are recognized as character string terminators. When the period is properly used as a concatenator, it will not appear in the generated inline expansion code. The replacement value for the variable symbol and the concatenated string appear as one string in the inline expansion code.

The period must be used to concatenate a character string that immediately follows a variable symbol if that character string starts with a letter, digit, left parenthesis, or period. Some examples of this are:

```
&Z.BC
&Z.12
&Z.(5,6)
&Z..BC
```

If the replacement value for the variable symbol &Z is 12, the values generated in the inline expansion code would be:

```
12BC
1212
12(5,6)
12.BC
```

The opposite combination reproduces the intended concatenation period in the inline expansion code. If you code a character before a variable symbol, don't use a period to concatenate the character with the variable symbol. The period, in this particular combination, is not considered to be a concatenator. A character coded before a variable symbol does not have to be concatenated with the variable symbol; the period is considered to be part of the character code and is generated as such. Take each previous example and switch the variable symbol and the characters:

```
BC.&Z
12.&Z
(5,6).&Z
BC..&Z
```

If &Z has a replacement value of 12, the following would be generated inline:

```
BC.12
12.12
(5,6).12
BC..12
```

The period will not be generated, however, if the character coded before the variable symbol is contained within quotes. The following is the proper way to concatenate a variable symbol prefixed with a character string if the period is used:

```
'BC'.'&Z'
'12'.'&Z'
'(5,6)'.'&Z'
'BC'.'&Z'
```

However, it isn't necessary to concatenate a variable symbol with a prefixed character. If you code:

```
BC&Z
12&Z
(5,6)&Z
BC.&Z
```

and &Z equals 12, then the following would be generated:

```
BC12
1212
(5,6)12
BC.12
```

You do not need a period after a variable symbol to indicate concatenation, if the variable symbol is followed by another variable symbol or a special character other than a left parenthesis or a period. Some examples of this are:

```
&A&B
&A+23
&A△B
&A=23
```

You could use periods as concatenators in these examples with no adverse side effects.

If &A equals TAG and &B equals 1, then the following is generated inline:

```
TAG1
TAG+23
TAG△B
TAG=23
```

There are really only two items that you can concatenate with variable symbols: characters and other variable symbols. When concatenating variable symbols with variable symbols, the period is optional and, when certain combinations of characters are concatenated with a variable symbol, the period is necessary.

Each format uses a different method for coding symbolic parameters to reference positional parameters, and this relates back to differences in the *parameter-specifications* between the PROC and MACRO. The PROC specifies positional parameters by indicating a variable symbol in operand 1 and the number of positional parameters in the call in operand 2. The variable symbol in operand 1 of the PROC statement is the symbolic parameter used in the body to reference positional parameters in the call. In addition to coding the symbolic parameters, you must also give the position of the parameter in parentheses immediately following the symbolic parameter: &pos(n). You do not give a positional reference after the symbolic parameter when referencing positional parameters in a MACRO body. The MACRO works differently because the prototype statement has to give a different variable symbol for every positional parameter in the call. To reference a positional parameter from a MACRO body, the symbolic parameter that you use is the variable symbol in the prototype statement that represents the desired positional value. This means you must keep track of all the variable symbols used to represent positional parameters, while with the PROC, you only require one variable symbol for the symbolic parameter. However, you can use the system variable symbol (&SYSLIST) in a macro to reference by position. (See 26.2.)

Symbolic parameter references to keyword parameters and symbolic parameter references to the label field of the call are the same for the PROC and MACRO. The symbolic parameter used to reference a keyword parameter in the call is the *&key* portion of the variable symbol used in the heading to represent the keyword parameter. And the symbolic parameter used to reference the *label* field of a call instruction is the variable symbol used as a *label-argument* in the heading. These symbolic parameters are used the same way for both formats because the mechanics for coding them in each heading are the same.

The details of how to use symbolic parameters in model statements in the PROC and the MACRO are given in Sections 25 and 26. PROC design of call instructions is encouraged because it is easier and because most of the call instructions designed by Sperry Univac are PROCs. Model statements using symbolic parameters are shown in Figure 24—4. More detailed examples of PROCs and MACROs are given in Section 30.

PROC Format

To write the macro in PROC format:

Statements		LABEL	ΔOPERATIONΔ	OPERAND
Proc	Heading	&NO1	PROC	&TAG,3
Name		ADD1	NAME	
Model	Body	&NO1	L	13,&TAG(1)
Model			A	13,&TAG(2)
Model			ST	13,&TAG(3)
End	Trailer		END	

To call the macro (macro call instruction):

LABEL	OPERATION	OPERAND
CAL1	ADD1	PAY,RAISE,TOTAL

Generates this pattern of coding:

LABEL	OPERATION	OPERAND
CAL1	L	13,PAY
	A	13,RAISE
	ST	13,TOTAL

MACRO Format

To write the macro in MACRO format:

Statements		LABEL	ΔOPERATIONΔ	OPERAND
Macro	Heading		MACRO	
Prototype		&NO2	ADD2	&TAG1,&TAG2,&TAG3
Model	Body	&NO2	L	13,&TAG1
Model			A	13,&TAG2
Model			ST	13,&TAG3
Mend	Trailer		MEND	

To call the macro (macro call instruction):

LABEL	OPERATION	OPERAND
CAL2	ADD2	PAY,RAISE,TOTAL

Generates this pattern of coding:

LABEL	OPERATION	OPERAND
CAL2	L	13,PAY
	A	13,RAISE
	ST	13,TOTAL

Figure 24—4. Example of MACRO and PROC Definitions

25. PROC Format

25.1. BASIC PROC DESIGN

The most basic type of PROC that you can design is one that requires no parameters from the call, no label, and no positional or keyword parameters in the operand field. All that is required is a mnemonic in the operation field.

LABEL	ΔOPERATIONΔ	OPERAND
	ADD1	

When this type of call is used, it generates the same sequence of instructions with no parameter replacement. The call instruction ADD1 is designed to generate the following code every time it is used in a source program:

LABEL	ΔOPERATIONΔ	OPERAND
	L	13,PAY
	A	13,RAISE
	ST	13,TOTAL

The design structure of a PROC that accomplishes this type of basic inline expansion is:

LABEL	ΔOPERATIONΔ	OPERAND
	PROC	
call-name	NAME	
	mnemonic-code	operands
	.	.
	.	.
	.	.
	mnemonic-code	operands
	END	

The PROC and NAME statements make up the heading of the PROC and must always be coded in the order shown. You use the PROC statement to identify the beginning of a macro definition in PROC format and the NAME statement to assign a *call-name* to the PROC. The *call-name* is coded in the *label* field of the NAME statement and is a 1- to 8-character symbol (it cannot be a variable symbol) defining the mnemonic operation code by which the macro definition may be referenced. The *call-name* must be unique. It may not be the same as any Sperry Univac mnemonic operation code or any *call-name* in your own library. (The only way that you can duplicate a *call-name* is if you override the established *call-name* with the OPSYM directive. (See 16.2.)

After the NAME statement is the body that contains the model statement and then the trailer, an END statement that indicates the end of the PROC. The model statements in the body are the source code statements that are generated when you use the *call-name* in your program. If you don't require parameter replacement, the model statements could be any BAL instruction, assembler directive, or call instruction except END, ICTL, or ISEQ.

The model statements for the ADD1 call are the BAL instructions that are generated every time ADD1 is used. A completely designed ADD1 PROC is as follows:

LABEL	△OPERATION△	OPERAND
	PROC	
ADD1	NAME	
	L	13,PAY
	A	13,RAISE
	ST	13,TOTAL
	END	

Although the basic PROC design that accomplishes direct instruction substitution saves the programmer time and effort, the most valuable role of the macro definition in BAL is variable parameter replacement. Variable parameter replacement allows you to vary the value of the *label, operation,* or *operand* field of any model statement by using the parameters submitted with each call. This means that you can design the ADD1 example PROC so that it accepts values from the call line and replaces the PAY, RAISE, and TOTAL operands of the model statements with the call line values. To accomplish parameter replacement in model statements, you indicate the fields that are variable by using symbolic parameters.

The symbolic parameter is the type of variable symbol used to indicate variable parameter replacement directly from the call line to the body. You can use symbolic parameters in the *label, operation,* or *operand* field of model statements to indicate parameter replacement. If you wanted the PAY, RAISE, and TOTAL operands of the ADD1 example to be variable, you would code symbolic parameters for those operands. The symbolic parameters in the model statements reference the values in the call through the PROC statement, and no symbolic parameter may be used in a model statement unless it also appears in the PROC statement. The value referenced in the call line replaces the symbolic parameter, and the manner in which you reference the parameters in the call depends on the kinds of parameters you design the call to have.

As shown in Figure 22—1, comments or instructions within a PROC call will be shifted one space beyond the last operand when they are assembled. This permits a maximum amount of space for comments on instructions that generate variable symbols.

25.2. REFERENCING POSITIONAL PARAMETERS IN THE CALL

You can define the call to submit values to the PROC body via positional parameters. This is done by indicating a variable symbol in operand 1 of the PROC statement and by indicating the total number of positional parameters that can appear in the call in operand 2 of the PROC statement:

LABEL	∆ OPERATION ∆	OPERAND
	PROC	&pos,n

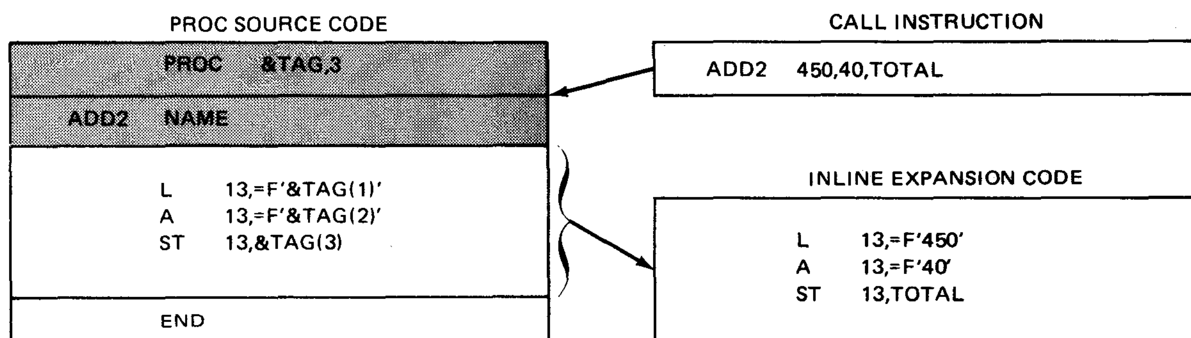
The *&pos* variable symbol is the symbolic parameter used in the body of the PROC to reference positional parameters in the call instruction. Following the symbolic parameter is a decimal number in parentheses (&pos(n)) that references the positional parameter in the call. For instance:

&pos(1) references positional parameter 1 in the call.

&pos(2) references positional parameter 2 in the call.

&pos(3) references positional parameter 3 in the call.

You can reference a positional parameter from the *label, operation*, or *operand* field of a model statement, and the value coded in the referenced position in the call line is generated in place of the symbolic parameter. If an omitted positional parameter is referenced, a null character string is generated in place of the symbolic parameter that made the reference. The following example shows the PROC source code and inline expansion code for an ADD2 call instruction that is designed to submit three positional parameters.



Operand 1 of the PROC statement indicates the variable symbol &TAG is used as the symbolic parameter reference in the body, and operand 2 indicates that there can be three positional parameters in the call. The NAME statement assigns the *call-name* ADD2 as the mnemonic to call the PROC. The body is a model for a procedure that adds two numbers and stores the result in a main storage location. The two numbers are picked up from positional parameter 1 of the call instruction, which is someone's pay, and positional parameter 2, which is their raise. The main storage location is picked up from positional parameter 3 of the call instruction. (One of the requirements for using this call instruction is that the user supply a DS statement with a *symbol* the same as that of positional parameter 3 for the PROC to store the result.) The first model statement is a *load* instruction that uses a variable symbol in operand 2 to reference positional parameter 1 in the call instruction. The PROC generates the *load* instruction with a full-word fixed-point literal value of 450 in operand 2. The *add* instruction is generated with a 40 in operand 2, and the main storage address TOTAL is generated in operand 2 of the *store* instruction.

25.3. REFERENCING KEYWORD PARAMETERS IN THE CALL

Instead of having the call submit values to the PROC body via positional parameters, you can use keyword parameters. This is done by using keyword parameter variable symbols starting in operand 3 of the PROC statement. Commas are used in operands 1 and 2 to indicate there are no positional parameters in the call (and *key* is the keyword name):

LABEL	ΔOPERATIONΔ	OPERAND
	PROC	,,$\&key_1$=,...,$\&key_m$=

Since a variable symbol can only be eight characters long and we've used two positions with the ampersand and equal sign, the keyword name portion of the variable symbol can only be six characters long. But the keyword name in the call can exceed six characters and still be accepted by the PROC. However, anything after six characters is ignored by the PROC; it recognizes only the first six characters as the true keyword name.

The symbolic parameter used in the body of the PROC to reference keyword parameters in the call is the variable symbol indicated in the PROC statement, without the equal sign (&key).

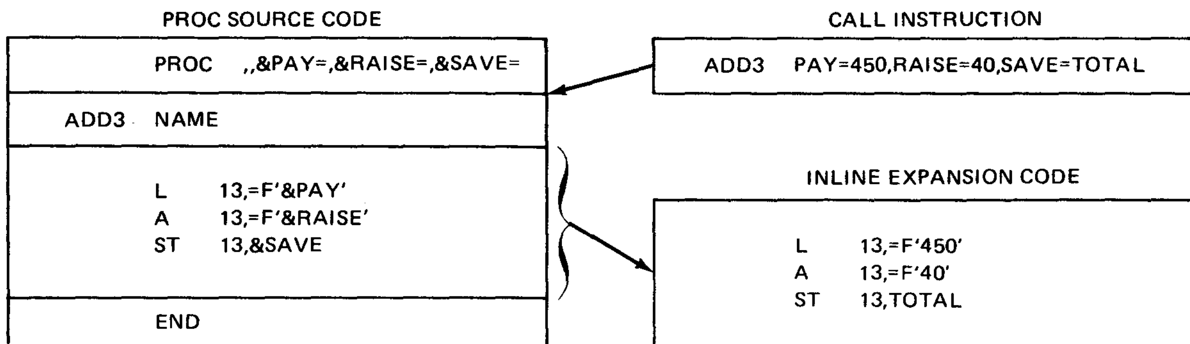
For instance:

&PAY references keyword parameter PAY=value in the call.

&RAISE references keyword parameter RAISE=value in the call.

&SAVE references keyword parameter SAVE=value in the call.

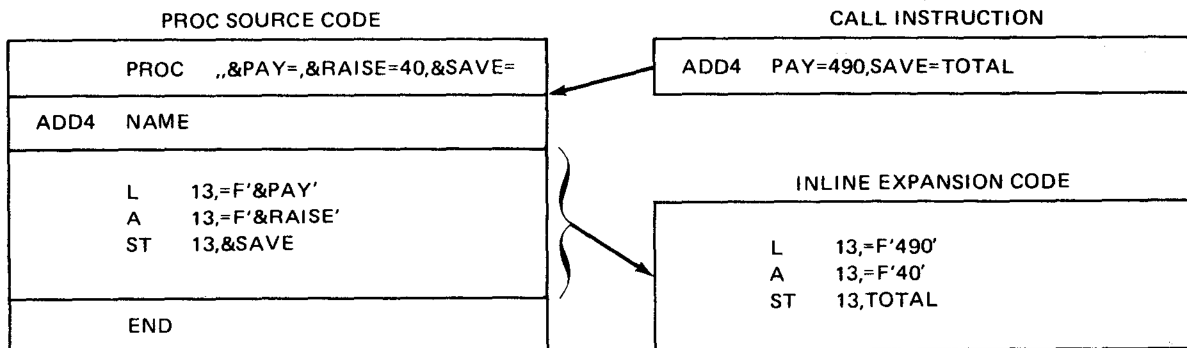
You can reference a keyword parameter from the *label, operation,* or *operand* field of a model statement, and the value after the equal sign in the call line will be generated in place of the symbolic parameter. The following example shows the PROC source code and inline expansion code for an ADD3 call instruction that is designed to submit three keyword parameters:



Operands 3 through 5 of the PROC statement indicate that there can be three different keyword parameters in the call with the keyword names of PAY, RAISE, and SAVE. The NAME statement assigns the *call-name* ADD3 as the mnemonic to call the PROC. (The body of this PROC is designed to perform the same function as the ADD2 example PROC.) The symbolic parameter &PAY in operand 2 of the first model statement references the keyword parameter PAY in the call, and since PAY=450, the value 450 is substituted in operand 2 of this model statement. The same processing takes place with symbolic parameters &RAISE and &SAVE.

You can design a PROC so that a preselected value is generated for a symbolic parameter that references an omitted keyword parameter in the call. Otherwise, symbolic parameters that reference omitted keyword parameters receive the value of a null character string. The default value for a keyword parameter is coded after the equal sign of the variable symbol in the PROC statement.

The following example shows a PROC with a default value indicated in the PROC statement:

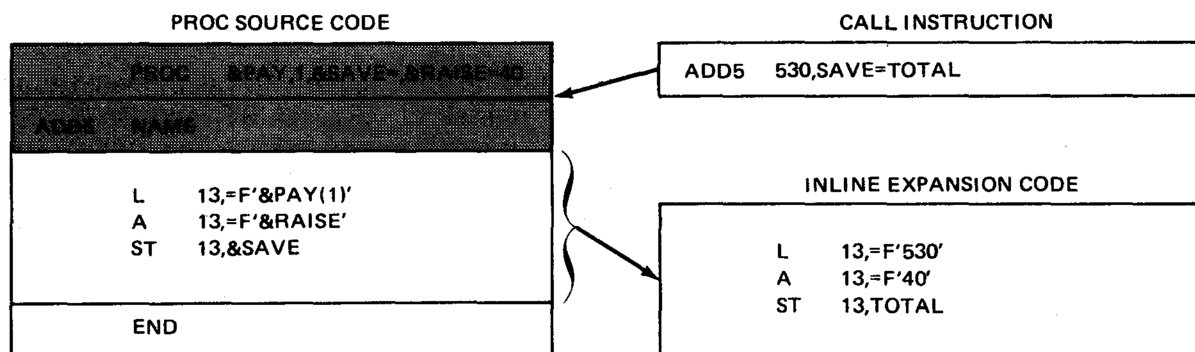


The call instruction doesn't use the RAISE keyword parameter so the default value of 40 indicated in the PROC statement is generated in the inline expansion code.

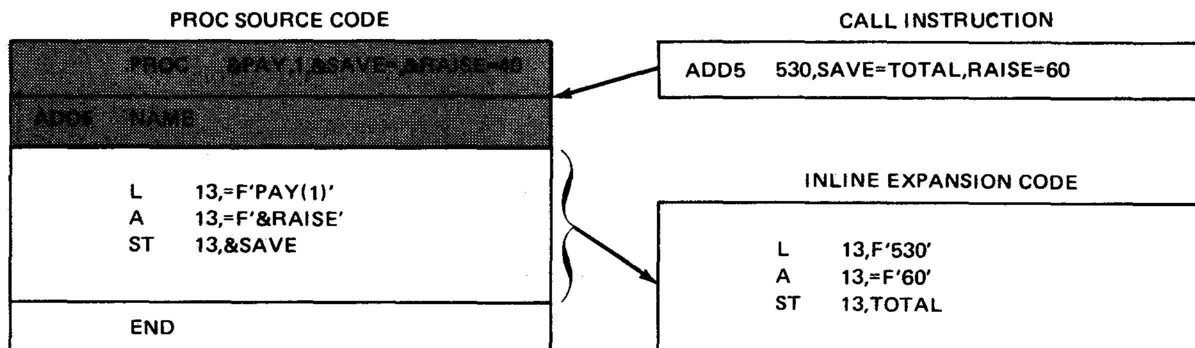
Another type of PROC is one with mixed-mode parameters. This PROC is designed to accept both positional and keyword parameters from the call. The format of the PROC statement for designing mixed-mode calls is:

LABEL	△OPERATION△	OPERAND
	PROC	$\&pos,n,\&key_1=,...,\&key_m=$

We could design the ADD4 example PROC to have the call submit the &PAY symbolic parameter as a positional parameter and the &SAVE and &RAISE symbolic parameters as keyword parameters, with RAISE having a default value of 40:



The following example shows what happens if you have a default value indicated in the PROC statement and submit a value for the keyword in the call instruction; the value in the call overrides the value in the PROC heading:



25.4. REFERENCING SUBPARAMETERS IN THE CALL

Another way to generate values in a field of a model statement is by referencing values in a parameter sublist. The parameter sublist is a list of subparameters that are subordinate to either a positional or keyword parameter. When you use a sublist, you can submit multiple parameter values from a single operand in the call. No coding is required in the PROC statement to support a sublist in a call line. It is a matter of referencing the sublist from a model statement by using a symbolic parameter with a sublist reference. The symbolic parameter reference to a subparameter in a positional parameter sublist is:

&pos(n,x)

where:

&pos
: Is the variable symbol used in operand 1 of the PROC statement to represent positional parameters.

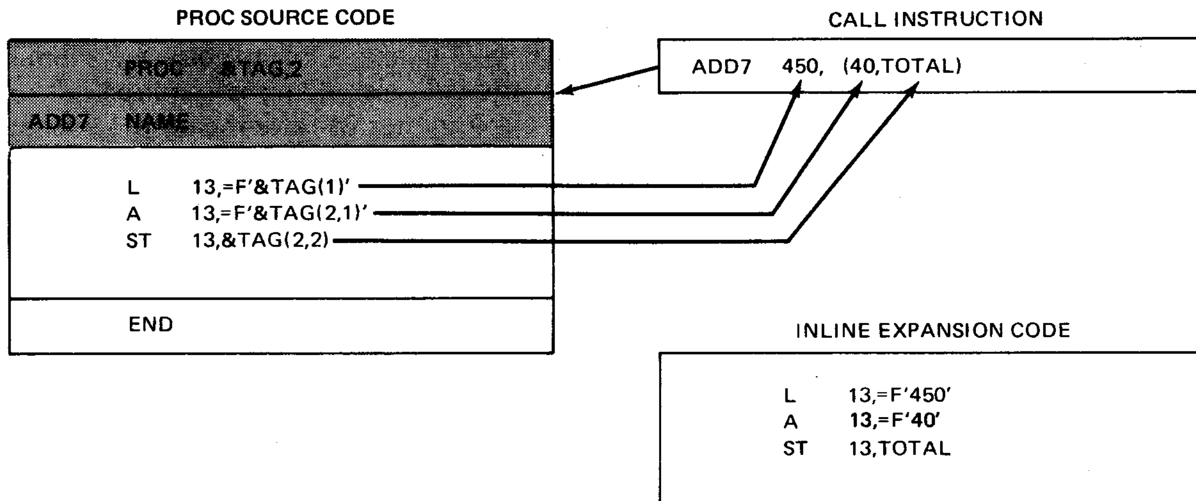
n
: Is the number of the positional parameter in the call.

x
: Is the position of the subparameter in the sublist. (All subparameters are referenced by position.)

Sublists for positional parameters must be coded within parentheses in the operand field of the call:

$(P_{1,1}, P_{1,2},\ldots, P_{1,n}),\ldots,(P_{n,1},\ldots,P_{n,x})$

The following example shows a PROC referencing a positional parameter sublist in the call line:



The symbolic parameter reference to a subparameter in a keyword parameter sublist is:

&key(x)

where:

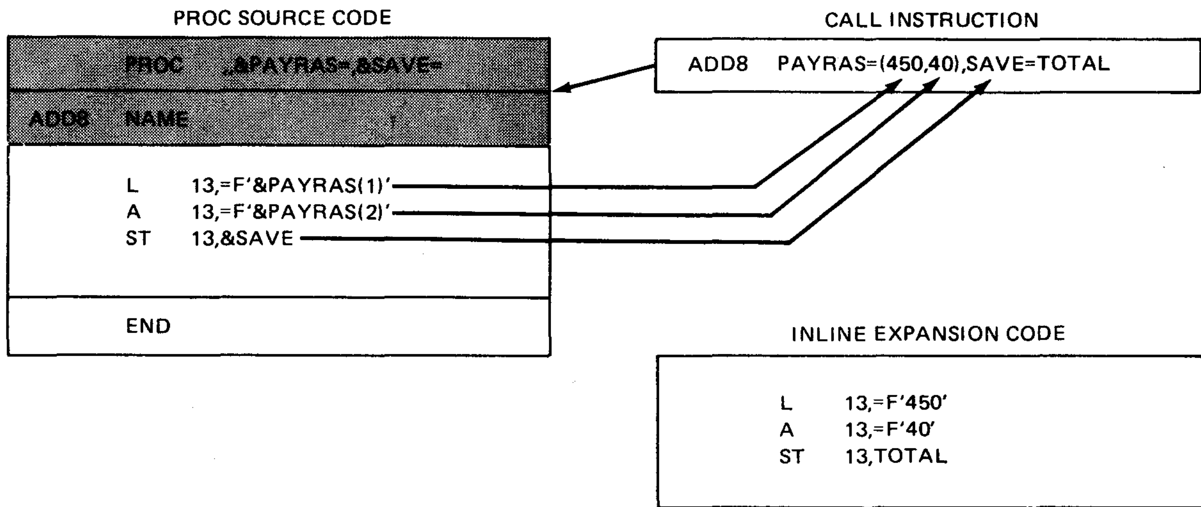
&key
Is the variable symbol in the PROC statement.

x
Is the number of the position of the subparameter in the sublist.

Sublists for keyword parameters must be coded within parentheses after the equal sign in the operand field of the call:

$k_1=(p_1,\ldots,p_x),\ldots,k_n=(p_1,\ldots,p_x)$

The following example shows a PROC referencing a keyword parameter sublist in the call line:



In addition to having the individual subparameters in a call generated in model statements, you can have the entire sublist generated. You do this by referencing the positional or keyword parameter with no reference to its subparameters. Simply use the symbolic parameter as you normally would reference a keyword or positional parameter; its associated sublist, including the parentheses, will be generated. If a SET symbol appears in the operand entry of a macroinstruction, attribute information is not provided and the operand may not be accessed as a sublist.

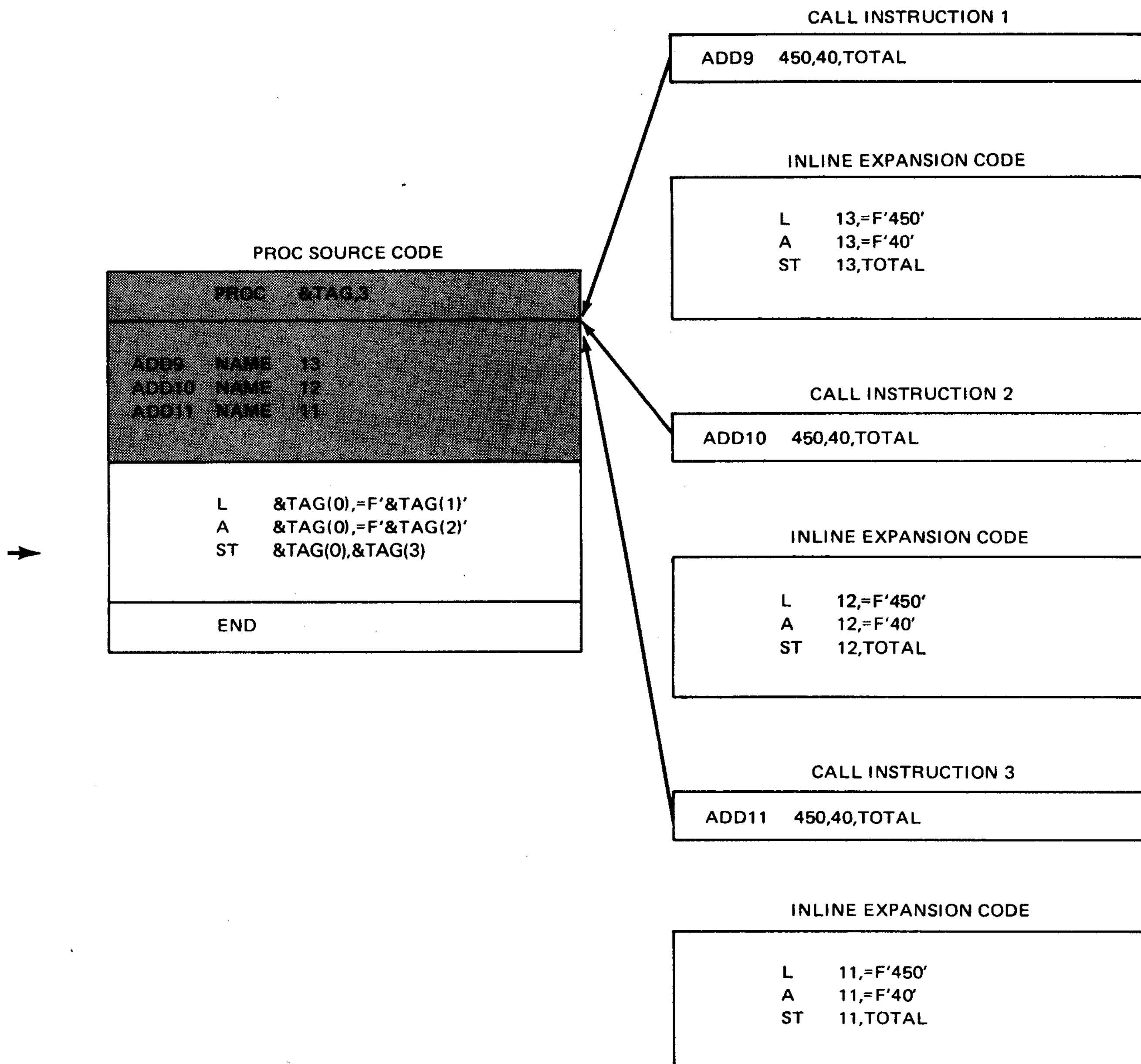
25.5. MULTIPLE PROC NAMES AND POSITIONAL PARAMETER 0

The split heading characteristic of the PROC permits another means of parameter modification. The parameter value is submitted by varying the mnemonic name in the operation field of the call. You can design a PROC so it may be called by many mnemonic names, and each name represents a different parameter value. This is easily done in the PROC because the NAME statement separates the *call-name* from the parameters in the PROC statement and therefore leaves the operand field of the NAME statement open for use. The PROC utilizes the operand field of the NAME statement for assigning a value to the *call-name:*

LABEL	△OPERATION△	OPERAND
call-name	NAME	pos-0

pos-0 can be a decimal or alphanumeric value but it cannot be a variable symbol. The value in the *operand* field of the NAME statement is referenced as positional parameter 0 by using the same symbolic parameter you indicated in operand 1 of the PROC statement (&pos(0)). You can vary the value for positional parameter 0 by using multiple NAME statements. Each NAME statement has a different *call-name* and a value in the operand field for positional parameter 0. A symbolic parameter referencing positional parameter 0 receives the *pos-0* value from the NAME statement whose *call-name* is used in the operation field of the call. All NAME statements must appear directly after the PROC statement and before any model statements, including comments.

We could design the ADD2 example PROC to have several different *call-names* and, each time a different *call-name* is used, the register number for operand 1 of the model statements is changed:



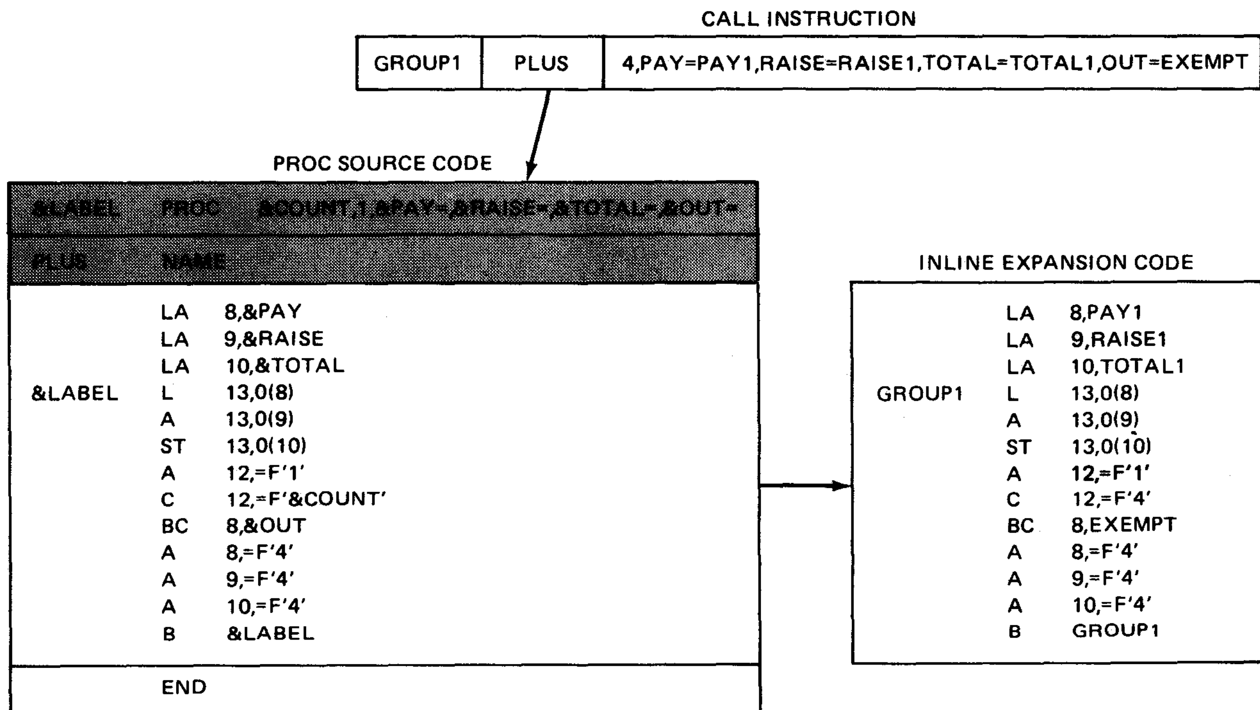
25.6. THE LABEL ARGUMENT

The *label-argument* is used to reference the symbol coded in the *label* field of a call instruction. To establish a *label-argument*, you must code a variable symbol in the *label* field of the PROC statement. The format of the PROC statement with a *label-argument* is as follows:

LABEL	△ OPERATION △	OPERAND
&symbol	PROC	&pos,n,&key$_1$=,...,&key$_m$=

The variable symbol that is coded in the *label* field of the PROC statement is used as the reference to the symbol in the *label* field of the call instruction. This variable symbol is used to take the label field from a call instruction and generate it in a model statement. Any model statement can have a *label-argument*. A *label-argument* is useful when you use a label in a PROC and you expect several copies of the inline expansion code to be in one program. The user can change the name of the label by changing the symbol in the label field of the call.

The following PROC uses a *label-argument*. The purpose of this PROC is to add the number of salaries indicated in positional parameter 1 and located where indicated by keyword parameter RAISE. The results are stored in the location indicated by keyword parameter TOTAL, and control is returned to the user program at the location indicated by keyword parameter OUT.



26. MACRO Format

26.1. BASIC MACRO DESIGN

A basic MACRO does not perform variable parameter substitution and, therefore, does not require any values to be passed from the call instruction. The design structure of a MACRO that accomplishes this type of basic inline expansion is:

LABEL	ΔOPERATIONΔ	OPERAND
	MACRO	
	call-name	
	mnemonic-code	operands
	.	.
	.	.
	.	.
	mnemonic-code	operands
	MEND	

The statements in a MACRO must always be coded in the order shown. First is the MACRO statement which indicates the beginning of a macro definition in MACRO format. Next is the prototype statement, which is where you code the *call-name.* Then, you code the model statements, which can be any BAL instruction, assembler directive, or call instruction except END, ICTL, or ISEQ. The last statement in a MACRO is the MEND statement, which indicates the end of the definition.

If we take the ADD1 PROC shown in Section 25 and design an ADD1 MACRO, it appears as follows:

LABEL	ΔOPERATIONΔ	OPERAND
	MACRO	
	ADD1	
	L	13,PAY
	A	13,RAISE
	ST	13,TOTAL
	MEND	

The ADD1 example MACRO produces the same inline expansion code as the ADD1 example PROC:

LABEL	△OPERATION△	OPERAND
	L	13,PAY
	A	13,RAISE
	ST	13,TOTAL

As shown in Figure 22—1, comments on instructions within a MACRO call will be shifted one space beyond the last operand when they are assembled. This permits a maximum amount of space for comments on instructions that generate variable symbols.

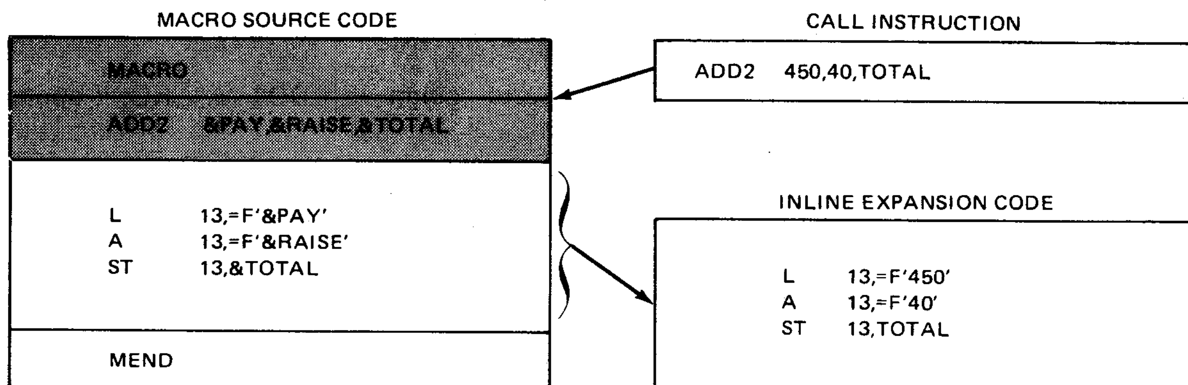
26.2. REFERENCING POSITIONAL PARAMETERS IN THE CALL

Designing a MACRO to reference positional parameters in the call is a little different from the way it is done in the PROC. In MACRO design, the prototype statement must indicate a variable symbol for each positional parameter to be coded in the call. Positional parameter variable symbols are coded in the prototype statement as follows:

LABEL	△OPERATION△	OPERAND
	call-name	&pos_1,...,&pos_n

The variable symbol representing a positional parameter in the call is used as the symbolic parameter to reference that positional parameter. You can reference a positional parameter from the *label, operation,* or *operand* field of a model statement, and the value coded in the referenced position in the call is generated in place of the symbolic parameter. If an omitted positional parameter is referenced, a null character string is generated in place of the symbolic parameter that made the reference.

The following example shows the ADD2 example PROC given in Section 25 recoded in MACRO format:



You can also reference the positional parameter values in the call by position rather than by using the symbolic parameters coded in the prototype statement. To do this, you must use the system variable symbol &SYSLIST. Instead of coding the positional parameter variable symbol for the symbolic parameter reference, you code:

&SYSLIST(n)

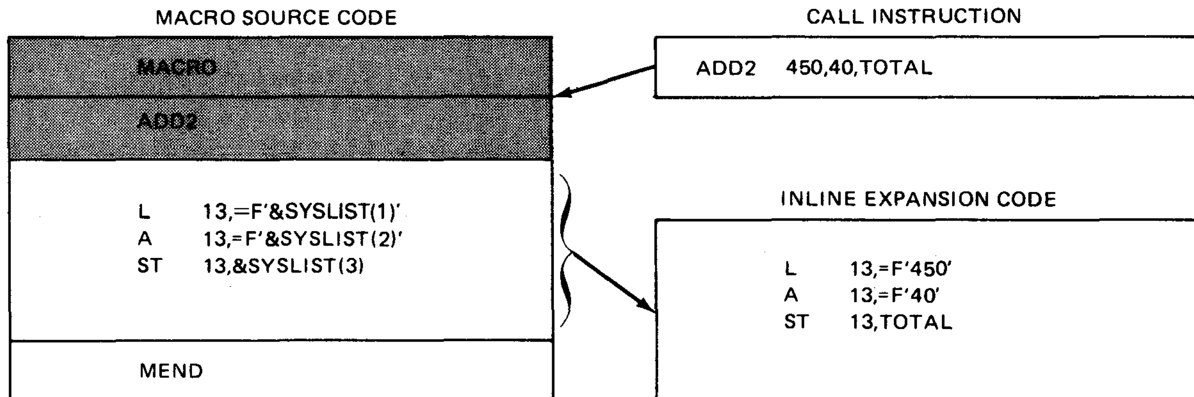
where:

n

Is the number of the positional parameter in the call and can be an expression that is a self-defining term or a SETA symbol. (SETAs are discussed in Section 27.)

If you miscount the positional parameter string in the call and n is greater than the actual number of parameters that are supposed to be in the call, then a null character string is generated in place of &SYSLIST(n). When you use &SYSLIST, it isn't necessary to have positional parameter variable symbols in the heading. &SYSLIST references the call instruction, not the heading. The function of &SYSLIST is to provide the MACRO designer with the option to reference positional parameters by position, similar to the way it is done in the PROC. However, &SYSLIST does not allow you to reference keyword parameters by position as you could do in the PROC. In the PROC, you could use &pos(n+1) and continue right on through into the keyword parameters in the parameter-list. This won't work with &SYSLIST because &SYSLIST only references positional parameters.

The following example shows the ADD2 example MACRO redesigned with &SYSLIST symbolic parameter references:



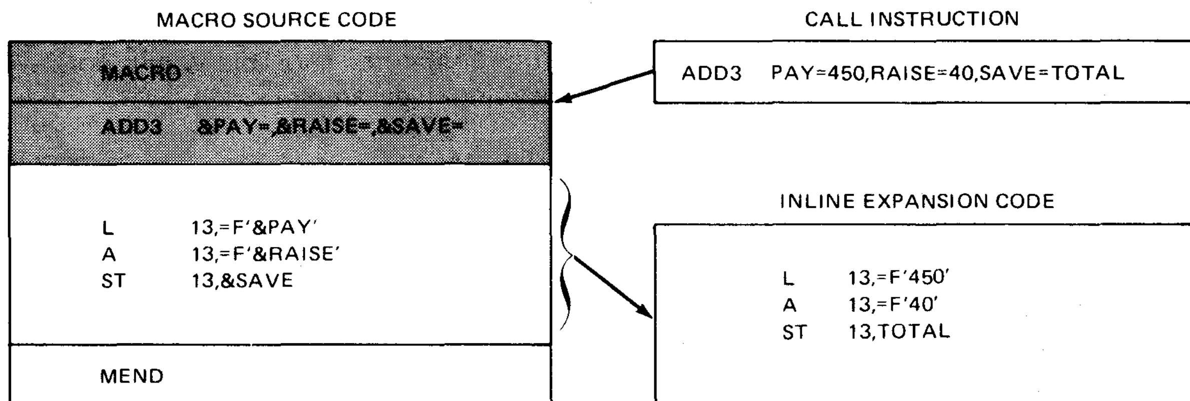
26.3. REFERENCING KEYWORD PARAMETERS IN THE CALL

The MACRO references keyword parameter values in the call in the same way as the PROC. A keyword parameter reference is made by a symbolic parameter that consists of the *&key* portion of the keyword variable symbol. If the call is to have only keyword parameters, the prototype statement is coded as follows:

LABEL	△OPERATION△	OPERAND
	call-name	$\&key_1=,...,\&key_m=$

You can reference a keyword parameter from the *label, operation,* or *operand* field of a model statement and the value after the equal sign in the call will be generated in place of the symbolic parameter.

The following is the ADD3 example PROC, shown in Section 25, recoded in MACRO format:

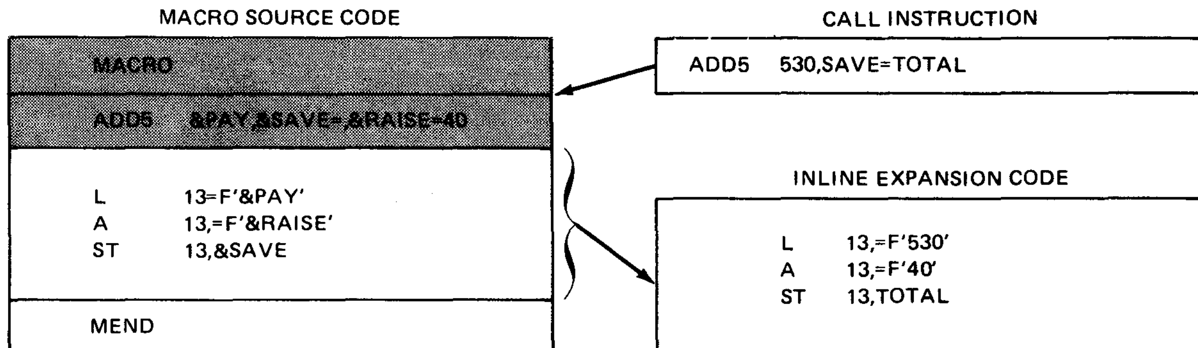


If you want any of the keyword parameters in the call to have a default value, you code that value after the equal sign of the keyword variable symbol in the prototype statement. Whenever a keyword parameter in the call is omitted, the symbolic parameter receives the value given the keyword variable symbol. A symbolic parameter referencing a keyword parameter that doesn't have a default value receives the value of a null character string.

To design a MACRO that references positional and keyword parameters in the call, you code the positional symbolic parameters before the keyword symbolic parameters in the prototype statement. The prototype statement for a mixed-mode MACRO is as follows:

LABEL	△OPERATION△	OPERAND
	call-name	$\&pos_1,...,\&pos_n,\&key_1=,...,key_m=$

The following example shows the ADD5 example PROC, shown in Section 25, redesigned in MACRO format:



26.4. REFERENCING SUBPARAMETERS IN THE CALL

Like the PROC, the MACRO does not require support from the heading to use sublists in the call. It is a matter of referencing the sublist from a model statement by using a symbolic parameter or &SYSLIST with a sublist reference. The &SYSLIST reference to a subparameter in a positional parameter sublist is:

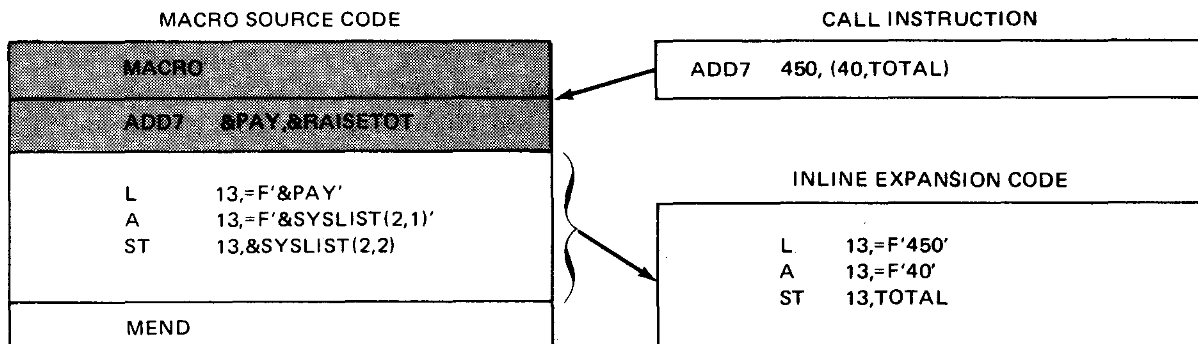
&SYSLIST(n,x)

where:

n
: Is the number of the positional parameter in the call.

x
: Is the number of the subparameter in the sublist.

The following example shows the ADD7 example PROC, shown in Section 25, recoded in MACRO format:



The symbolic parameter reference to a subparameter in a keyword parameter sublist for a MACRO is the same as for the PROC:

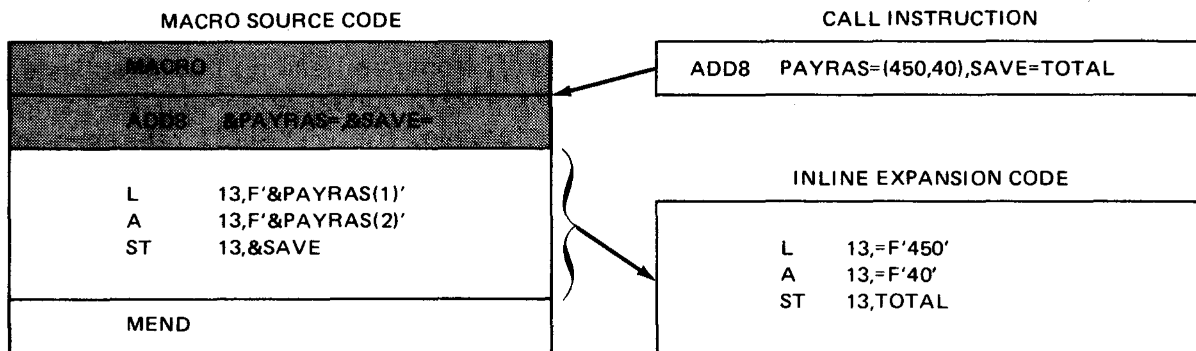
&key(x)

where:

&key
Is the variable symbol in the prototype statement.

x
Is the number of the subparameter in the sublist.

The following example shows the ADD8 example PROC, shown in Section 25, redesigned in MACRO format:



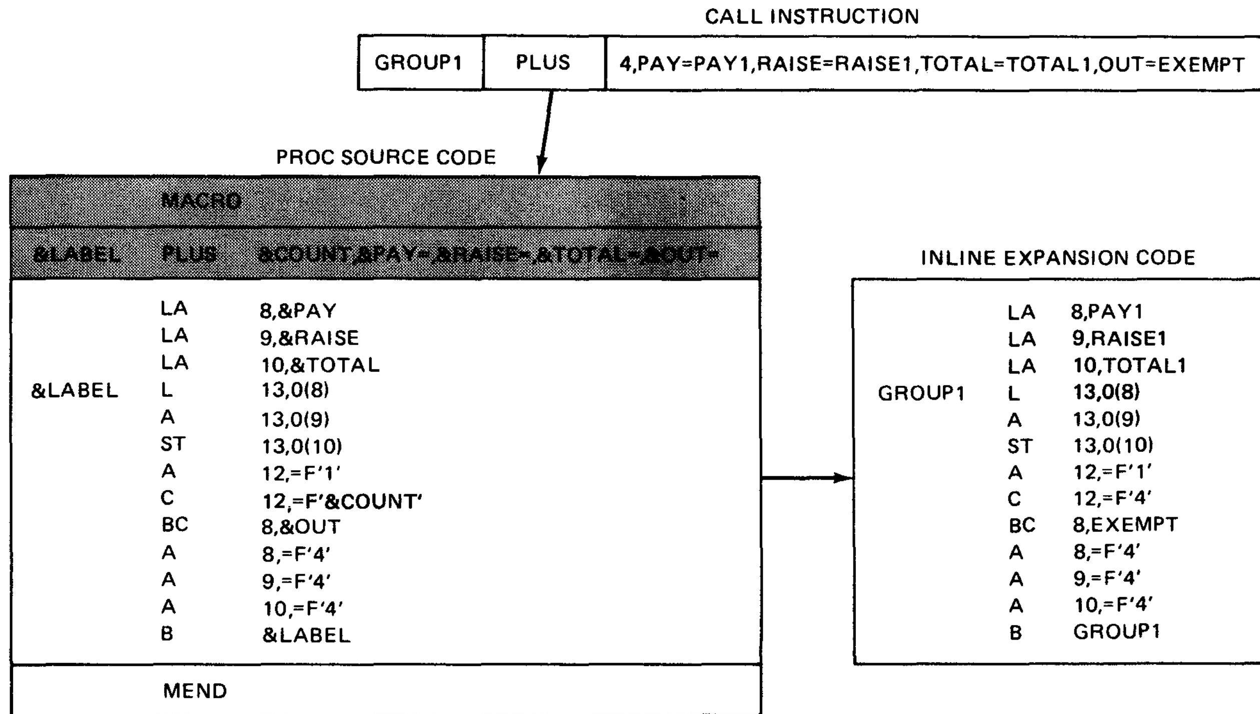
If you reference a positional or keyword parameter with a sublist and do not include a reference to a subparameter within the list, you will generate the entire sublist, including parentheses, in place of the symbolic parameter. The same thing happens in a PROC. If a SET symbol appears in the operand entry of a macroinstruction, attribute information is not provided and the operand may not be accessed as a sublist.

26.5. THE LABEL ARGUMENT

The label argument for a MACRO is used the same way it is used in the PROC. To establish the label-argument, you must code a variable symbol in the label field of the prototype statement. The format of the prototype statement with a label argument is as follows:

LABEL	△ OPERATION △	OPERAND
&symbol	**call-name**	**$\&pos_1,\ldots,\&pos_n,\&key_1=,\ldots,\&key_m=$**

The following MACRO uses a label argument. This example MACRO is the PLUS example PROC recoded in MACRO format (25.6).



27. Conditional Assembly

Conditional assembly statements are used to control the pattern of coding generated within a macro definition, and to define and assign values to set symbols that can be used to vary parts of generated statements. Conditional assembly statements are used by the programmer to direct the asssembler to:

- exclude lines of code from the assembler output;
- include a set of lines more than once in the assembly output; or
- establish and alter values to determine whether a set of lines should be included in the output listing.

Table 27—1 lists the function of each conditional assembly statement.

Table 27—1. Conditional Assembly Language Statements (Part 1 of 2)

Statement	Function	General Usage
ACTR	Sets a conditional assembly loop counter	Branching
AGO *AGOB *GOTO	An unconditional branch	
AIF *AIFB	A conditional branch	
ANOP *LABEL	Provides a branch destination point for a location that already contains a symbol or variable symbol	
MEXIT	Stops processing of macro definitions	
DO	Defines starting point of the code and the number of times it is to be generated	Used for defining the range for repetitive code
ENDO	Defines the end of the code to be repeated	
MNOTE PNOTE	Generates messages in macro definitions or program source code	Used for generating messages

Table 27—1. Conditional Assembly Language Statements (Part 2 of 2)

Statement	Function	General Usage
LCL	Declares a general-purpose local set symbol	Used for declaring set symbols that are to be used only inside the macro definition that is declaring the set symbol
LCLA	Declares an arithmetic local set symbol	
LCLB	Declares a Boolean local set symbol	
LCLC	Declares a character local set symbol	
GBL	Declares a general-purpose global set symbol	Used for declaring set symbols that are to be used not only in the macro definition in which the set symbol is declared but also outside the macro definition in other macro definitions
GBLA	Declares an arithmetic global set symbol	
GBLB	Declares a Boolean global set symbol	
GBLC	Declares a character global set symbol	
SET	Assigns an arithmetic or character string value to a set symbol	Used to assign values to arithmetic, Boolean, or character set symbol
SETA	Assigns an arithmetic value to a set symbol	
SETB	Assigns a binary value of 0 (false) or 1 (true) to a set symbol	
SETC	Assigns a character value to a set symbol	

*Alternate mnemonic

27.1. SET SYMBOLS

Set symbols are a type of variable symbol (Appendix G). The rules for writing set symbols are the same as for other variable symbols:

- An ampersand (&) is followed by an alphabetic character followed by up to six additional characters (total maximum characters is eight).
- If the ampersand is omitted, the assembler interprets the character string as a symbol and not as a set symbol.

The following are valid set symbols:

```
&C
&A1
&PARAM
```

The following are not valid set symbols for the reasons stated:

CAT
: Valid for an ordinary symbol but not as a set symbol; no leading ampersand.

&1
: First character after & is not alphabetic.

&S12345678
: There are too many characters in the string (maximum length, including &, is eight characters).

Because set symbols are evaluated in the macro generation phase of the assembler, they may be used as counters, switches, or values to control the sequence of code generated. Unlike an ordinary symbol, the value assigned to a set symbol may be altered during assembly.

A set symbol may be either global or local. A global set symbol, once declared and given a value by a SET statement, retains the same value until that value is changed by another SET statement. A local set symbol is defined only within the macro definition in which it is declared. The value of a local set symbol within one macro definition is not affected by the declaration of either a global or local set symbol with the same name in another macro definition.

Set symbols must be declared after macro prototype or NAME statements and before being referenced.

27.1.1. Local Set Symbols

A local set symbol is available for use only in the macro definition in which it is declared. Four statements are available for declaring local set symbols. The declarative chosen determines the values to which the set symbol may be set and the type of SET statement used to assign the values.

The basic format for a local set symbol declaration is:

LABEL	△OPERATION△	OPERAND
unused	{ LCL LCLA LCLB LCLC }	s_1 [, s_2,..., s_n]

where:

LCL
Declares a general-purpose local set symbol.

LCLA
Declares an arithmetic local set symbol.

LCLB
Declares a Boolean local set symbol.

LCLC
Declares a character local set symbol.

$s_1, s_2, \ldots, s_n$
Are set symbol names.

The operand field of the local set declaration may contain one or more set symbol names. A local symbol is considered defined when declared. A set symbol declared by an LCLA or LCLB statement is assigned an initial value of zero.

A set symbol declared by an LCLC or LCL statement is assigned an initial value of a null character string.

Examples:

```
     LABEL    ΔOPERATIONΔ                         OPERAND
     1          10        16

1.              LCLA      &A,&B
2.              LCLB      &BIG
3.              LCLC      &BIG3
```

1. Declares arithmetic local set symbols &A and &B.
2. Declares a Boolean local set symbol &BIG.
3. Declares a character local set symbol &BIG3.

27.1.2. Global Set Symbols

Global set symbols are initialized only once and are used to pass values back and forth between macro definitions. A global set symbol is available to all macro definitions in which it is also declared.

Four statements are available for declaring global set symbols. The declarative chosen determines the range of values to which the set symbol may be set and the type of set statement used to assign the values.

The basic format for a global symbol declaration is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	{GBL GBLA GBLB GBLC}	s_1 [, s_2,..., s_n]

where:

GBL
Declares a general-purpose global set symbol.

GBLA
Declares an arithmetic global set symbol.

GBLB
Declares a Boolean global set symbol.

GBLC
Declares a character global set symbol.

$s_1, s_2, \ldots, s_n$
Are set symbol names.

The operand field of the global set declaration may contain one or more set symbols. A global set symbol is considered defined when declared. It is initialized only once; that is, the first time it is declared. With subsequent declarations in other contexts, the global set symbol is available for use but is not reinitialized. A set symbol must be declared before it is available for use. A set symbol declared by a GBLA or GBLB statement is assigned an initial value of zero. A set symbol declared by a GBLC or GBL statement is assigned an initial value of a null character string.

If a set symbol is declared as a global set symbol in more than one macro definition, it must be declared with the same statement code in each macro definition.

Examples:

```
   LABEL    ΔOPERATIONΔ                       OPERAND
   1         10      16
  ---------------------------------------------------------
1.           GBL     &BR
2.           GBLA    &SET1,&SET2
3.           GBLB    &BUT
4.           GBLC    &GLB
```

1. Declares a general-purpose global set symbol.

2. Declares arithmetic global set symbols.

3. Declares a Boolean global set symbol.

4. Declares a character global set symbol.

27.1.3. Set Symbol Value Assignment

Four statements are provided to assign values to set symbols: SETA, SETB, SETC, and SET. The statement used depends on the statement chosen to declare the set symbol.

- SETA

  Assigns values to set symbols declared in either LCLA or GBLA.

- SETB

  Assigns values to set symbols declared in either LCLB or GBLB.

- SETC

  Assigns values to set symbols declared in either LCLC or GBLC.

- SET

  Assigns values to set symbols declared in either LCL or GBL.

27.1.4. SET Statement

The SET statement can be used to assign either an arithmetic or character string value to a variable symbol declared by an LCL or GBL statement.

The format of the SET statement is:

LABEL	ΔOPERATIONΔ	OPERAND
&s	SET	$\left\{\begin{matrix}a\\c\end{matrix}\right\}$

where:

&s
Is a set symbol declared by LCL or GBL.

SET
Defines the operation.

a
Is a valid arithmetic expression.

c
Is a valid character expression.

When the operand of the SET statement contains an arithmetic expression, the value of the expression may range from -2^{23} to $+2^{23}-1$. When the operand of the SET statement contains a character expression, the maximum length that may be specified is eight characters.

If a SET variable symbol is assigned a character value, a reference to the SET symbol yields the same result as a reference to a SETC symbol assigned the same character value. Similarly, if a SET variable symbol is assigned an arithmetic value, a reference to the SET symbol yields the same result as a reference to a SETA symbol assigned the same value. A SET variable symbol with a character value may be reassigned an arithmetic value, and vice versa.

A SET expression is a SETA expression allowing the use of the operators >,<, */,//,=,**, — —, and ++ in the SET expression when an arithmetic operator is valid. The two characters ** represent the logical product AND, the two characters ++ represent the logical sum OR, and the two characters — — represent the logical difference XOR.

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

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

where:

=

Compares the value of two terms or expressions. If the two values are equal, the assembler assigns a value of 1 to the expression. If the values are not equal, a zero value is assigned.

>

Compares two terms or expressions. If the value of the first (left) term is greater than the value of the second (right) term, a value of 1 is assigned to the expression. If the value of the second term is greater than the value of the first term, a zero value is assigned.

<

Compares the value of the first (left) expression or term with the second (right) expression or term. If the value of the first expression or term is less than the value of the second, a value of 1 is assigned to the expression. If the value of the second expression or term is less than the value of the first, a zero value is assigned.

Given the expression A+B>C, if the expression A+B has a greater value than the value of C, the assembler assigns a value of 1 to the expression. If the value of C is greater than the value of A+B, a zero value is assigned.

Since the value of a relational character or logical expression is arithmetic, the expression may be used as a term in an arithmetic expression.

Operator priority is shown in Table 27—2.

Table 27—2. Operator Priority

Operator	Hierarchy
*/	6
//, *, /	5
+, −	4
**	3
− −, ++	2
<>=	1

Examples:

```
   LABEL    ΔOPERATIONΔ                          OPERAND
   1          10          16
  ------------------------------------------------------------
1. &ARK      SET         'ANIM'
2. &NUM      SET         6
```

1. The SET symbol &ARK is assigned the value of ANIM.

2. The SET symbol &NUM is assigned the value of 6.

27.1.5. SETA Statement

The SETA statement assigns an arithmetic value to a variable symbol that was declared by an LCLA or GBLA statement.

The format of the SETA statement is:

LABEL	Δ OPERATION Δ	OPERAND
&s	SETA	a

where:

&s
Is a set symbol declared by either LCLA or GBLA.

SETA
Defines the operation.

a
Is a valid SETA term or an arithmetic combination of valid SETA terms.

A valid SETA term is:

- a self-defining term;
- a variable symbol with an arithmetic value; or
- a character value consisting of one to eight decimal digits.

The arithmetic operators used in writing SETA expressions are +, —, *, and /. The expression may not begin with an operator. Two operators or two terms may not succeed one another.

The rules of precedence for the evaluation of a SETA arithmetic expression are the same as stated in Table 27—2. The value of a SETA expression may range from -2^{23} to $2^{23}-1$.

When the SETA symbol is used in an arithmetic expression, the arithmetic value of the symbol is substituted for the symbol. If the SETA symbol is used in another context, the arithmetic value of the SETA symbol is converted to an unsigned decimal integer with leading zeros removed. This decimal value is then substituted for the SETA symbol. If the value of the SETA symbol is zero, a single zero is substituted.

Examples:

```
   LABEL    ΔOPERATIONΔ                      OPERAND
   1        10         16
  ---------------------------------------------------
1. &ART     SETA       5
2. &LOC     SETA       7
3. &HER     SETA       &ART+&LOC
```

1. The SETA symbol &ART is assigned a value of 5.

2. The SETA symbol &LOC is assigned a value of 7.

3. The SETA symbol &HER is assigned a value of 12.

27.1.6. SETB Statement

The SETB or set Boolean value statement may be used to assign a binary value of zero or one to a variable symbol which was declared by an LCLB or GBLB statement. The format of the SETB statement is:

LABEL	ΔOPERATIONΔ	OPERAND
&s	SETB	b

where:

&s
Is a set symbol declared in either LCLB or GBLB.

SETB
Defines the operation.

b
Is a valid logical expression that must be enclosed in parentheses or a 0 or 1 enclosed in parentheses.

The logical expression in the operand field may have a value of either 0 (false) or 1 (true), and the set symbol specified in the name field of the set statement is assigned the resultant binary value. The logical expressions may consist of a single term or logical combination of terms.

The permissible terms are:

- a SETB arithmetic relational expression;
- a SETB character relational expression; and
- a SETB symbol.

The SETB logical operators that may be used to combine the terms are AND, OR, and NOT. The logical expression must not contain two terms in succession. Two operators may appear in succession if the first operator is either AND or OR, and the second operator is NOT. Only the operator NOT is allowed prior to the first term of the expression.

A SETB arithmetic relational expression consists of two arithmetic expressions connected by a SETB relational operator. A SETB character relational expression consists of two character strings connected by a SETB relational operator. The SETB relational operators are:

- NE

  Not equal

- EQ

  Equal

- LT

  Less than

- LE

  Less than or equal

- GT

  Greater than

- GE

  Greater than or equal

The arithmetic expression that may be used as a term in the SETB arithmetic relational expression is defined under the SETA statement. The rules under the SETC statement define the format of the character string that may be used in a SETB character relational expression. If two character strings are of unequal length, the shorter will always compare less than the longer, regardless of actual value. The maximum length of character strings that may be compared is 127 characters.

In writing SETB expressions, the SETB relational or logical operators must be preceded and followed by at least one blank or other special character. The relational expression may be optionally enclosed in parentheses.

The procedure for evaluating a SETB expression is:

- Each term (SETB symbol, SETB arithmetic expression, or SETB character expression) is evaluated and given a value of either 1 (true) or 0 (false).

- Evaluation is from left to right. The weight of the logical operators is:

  OR = 1

  AND= 2

  NOT= 3

  Therefore, NOT is performed prior to AND, and AND is performed prior to OR.

If a SETB variable symbol is used in the operand field of a SETA or DO statement, or in an arithmetic relation (in either a SETB or AIF term), the binary values 0 and 1 are converted to the arithmetic value +0 and +1.

If the SETB variable symbol is used in the operand field of a SET statement, the value substituted is dependent on the context. In an arithmetic expression, +1 or +0 are substituted. In a character expression, the character values 1 and 0 are substituted.

Examples:

```
   LABEL    ΔOPERATIONΔ                        OPERAND
   1        10         16
  ---------------------------------------------------------
1. &CONT    SETB       (L'&TO EQ 4)
2. &EXP     SETB       (1)
```

1. If the expression L'&TO EQ 4 is true, the symbol &CONT is assigned a value of 1; otherwise, it is assigned a value of 0.

2. The symbol &EXP is assigned a value of 1.

27.1.7. SETC Statement

The SETC statement may be used to assign a character value to a variable symbol that was declared by an LCLC or GBLC statement.

The format of the SETC statement is:

LABEL	ΔOPERATIONΔ	OPERAND
&s	SETC	c

where:

&s
Is a set symbol declared by either LCLC or GBLC.

SETC
Defines the operation.

c
Is a valid SETC operand.

A SETC operand must be a character expression. (See 27.1.8.)

The maximum length of the value that may be specified for a SETC symbol is eight characters. If more than eight characters are specified, only the leftmost eight characters are used by the assembler.

Examples:

```
     LABEL    ΔOPERATIONΔ                      OPERAND
     1         10       16
   ----------------------------------------------------------
1. &TYPE      SETC      '&&AID'
2. &CITY      SETC      'CINN'
```

1. The symbol &TYPE is assigned the value of &AID.

2. The symbol &CITY is assigned the value of CINN.

27.1.8. Character Expressions

A character expression is either a character string, a character substring, or a concatenation of strings or substrings. Character expressions are used as the operand of a SET or SETC statement or as terms in a SETB, SET, AIF, or DO relational expression. Any character string is considered to be greater in value than any shorter character string. A character expression may have a length of up to 127 characters.

27.1.9. Subscripted SET Symbols

Subscripted SET symbols may be defined as both global and local SET symbols. The local SET symbols previously defined were all nonsubscripted SET symbols. Subscripted SET symbols provide you with a convenient way to use a SET symbol plus a subscript to refer to many binary, arithmetic, or character values. The subscript may be any arithmetic expression that is allowed in the operand of a SETA statement in the range of 1 to the specified dimension. The subscripted SET symbol consists of a SET symbol immediately followed by a subscript enclosed in parentheses.

A SETA or SETB operand permits only five levels of parentheses.

The following are valid subscripted SET symbols:

&INPUT(30)
&B45723(&A1)
&A6B9(2+&B1)

The following are invalid subscripted SET symbols:

&AB	No subscript
(300)	No SET symbol

27.1.9.1. Defining Subscripted SET Symbols

To use a subscripted SET symbol, you must write in a GBLA, GBLB, GBLC, LCLA, LCLB, or LCLC instruction with a SET symbol immediately followed by an unsigned decimal integer enclosed in parentheses. The number of SET variables associated with the SET symbol is indicated by the decimal integer and is called the dimension. Every variable associated with a SET symbol is assigned an initial value that is the same as the initial value assigned to the corresponding type of nonsubscripted SET symbol. If a subscripted SET symbol is defined as global, the same dimension (decimal integer) must be used with the SET symbol each time it is defined as global.

The maximum dimension of 255 can be used with a SETA, SETB, or SETC statement. A subscripted SET symbol may be used only if the declaration was subscripted, while a nonsubscripted SET symbol may be used only if the declaration had no subscript.

27.2. BRANCHING

The sequence of processing macro source code statements may be altered by branching. The assembler provides for conditional and unconditional branching.

27.2.1. Sequence Symbols

A sequence symbol is used to define a branch destination point. A sequence symbol may appear in the label field of any statement that does not contain a symbol or a set symbol, except for a macro prototype statement, a local or global symbol declaration statement (LCL, LCLA, LCLB, LCLC, GBL, GBLA, GBLB, GBLC), or MACRO, PROC, NAME, ICTL, or ISEQ statement.

A sequence symbol is written in the following form: a period followed by at least one alphabetic character followed by up to six alphanumeric characters. The following are valid sequence symbols:

```
.D

.D3

.BRNCPNT
```

When a sequence symbol is written in the label field of a macroinstruction, statement, and the prototype (MACRO format) or PROC statements (PROC format) for that macro definition that contain a variable symbol in their label fields, the sequence symbol does not replace the variable symbol.

27.2.2. Unconditional Branch (AGO)

The *unconditional branch* (AGO) statement unconditionally alters the sequence of source statement processing. The format is:

LABEL	△OPERATION△	OPERAND
[$.s_1$]	{ AGO / AGOB / GOTO }	$.s_2$

where:

AGO
Defines the operation.

$.s_1$
Is a sequence symbol.

$.s_2$
Is a sequence symbol defined in a source code statement.

The label field of the AGO statement may contain a sequence symbol. AGOB or GOTO may be used in lieu of AGO in the operation field. The sequence symbol in the operand field is the symbol of the next statement to be processed. Branching forward or backward from the AGO statement is permitted.

When an AGO statement is used in a macro definition, the sequence symbol specified in the operand field must appear in the label of another statement in that macro definition.

Examples:

```
   LABEL    ΔOPERATIONΔ                     OPERAND
   1         10        16

1.           AGO       .STOP
2. .BRANCH   AGO       .BUG
```

1. An unconditional branch is made to another statement in the source code labeled .STOP.

2. An unconditional branch is made to an instruction labeled .BUG somewhere else in your program. Notice a sequence symbol is used in the label field.

27.2.3. Conditional Branch (AIF)

The *conditional branch* (AIF) statement conditionally alters the sequence of source statement processing. The format is:

LABEL	△ OPERATION △	OPERAND
[$.s_1$]	{ AIF / AIFB }	(b) $.s_2$

where:

$.s_1$
: Is a sequence symbol.

AIF
: Defines the operation.

(b)
: Is a SETB logical expression enclosed in parentheses.

$.s_2$
: Is a sequence symbol defined in a source code statement.

The label field of the AIF statement may contain a sequence symbol. AIFB is permitted in lieu of AIF in the operation code field.

Any logical expression permitted in the operand field of a SETB statement (27.1.6) is valid in the operand field of the AIF statement except a 0 or a 1 enclosed in parentheses. The sequence symbol in the operand field must be written immediately after the parenthesis terminating the logical expression.

If, after the logical expression has been evaluated, the condition is true (a value of 1), you branch to the statement specified by the operand. If the condition is false (a value of 0), the statement in the source code following the AIF statement would be the next statement to be processed. Branching either forward or backward from the AIF statement is permitted. When an AIF statement is written in a macro definition, the sequence symbol specified in the operand field must appear in the label field of another statement within that macro definition.

Examples:

```
   LABEL    ΔOPERATIONΔ                   OPERAND
   1         10       16
  ------------------------------------------------------------
1.           AIF      (&BRO NE Ø).SIS
2. .IF       AIF      (L'&TO EQ L'&FROM).END
```

1. If the value of the symbol &BRO is zero, the next statement to be processed is &IF.

2. If the length attributes of the symbols &TO and &FROM are equal, a true (1) results and a branch is made to a statement in the source code labeled .END.

27.2.4. Define Branch Destination (ANOP)

The *define branch destination* (ANOP) statement is provided to facilitate branching. If a branch is necessary and no statement within the source code supplies the branch destination in its label field, an ANOP statement can be coded to provide a label to which to branch. The format is:

LABEL	Δ OPERATION Δ	OPERAND
.s	{ANOP LABEL}	unused

where:

.s
Is a sequence symbol.

ANOP
Defines the operation.

The label field must contain a sequence symbol.

When the label field of a statement which is desired as a branch destination point already contains a symbol or variable symbol, the branch destination is indicated by preceding the statement by an ANOP statement.

LABEL is an acceptable synonym for ANOP in the operation field.

Example:

```
LABEL    ΔOPERATIONΔ                              OPERAND
1         10        16
---------------------------------------------------------------
          .
          AIF       (&TYPE NE L) .DEST
          .
          .
.DEST     ANOP
&STATE    SETC      'PENN'
          .
          .
```

If the Boolean expression in the AIF operand is true, a branch is made to the ANOP statement with the label .DEST. Since no operations are performed by ANOP, control passes immediately to the following statement, a SETC whose label field is already occupied by the variable symbol &STATE.

27.2.5. Macro Definition Exit (MEXIT)

The *macro definition exit* (MEXIT) statement is used when it is necessary to process only one section or operation of a macro definition rather than the entire macro definition. This statement indicates to the assembler that the processing of a macro definition should be terminated before ending normally with a MEND statement.

The format of the MEXIT statement is:

LABEL	Δ OPERATION Δ	OPERAND
unused	MEXIT	unused

When MEXIT is used, the assembler terminates processing the macro definition and processes the statement in the source program following the macro call instruction that called the macro definition containing the MEXIT. A coding example of the MEXIT statement is included in 27.3.1.

27.3. ERROR MESSAGES AND COMMENTS

PNOTE or MNOTE statements are used to generate error messages or comments in a macro definition or in source code statements.

27.3.1. MNOTE Message Statements (MNOTE)

A MNOTE message statement is used to generate an error message. It indicates how dangerous the error is or generates a comment that supplies information. A MNOTE statement is used in a macro definition or in source code statements.

The MNOTE statement source code statement format is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	MNOTE	{ 'm' / Δ, 'm' / S, 'm' / *, 'm' }

This format can be used to specify a message enclosed in apostrophes, a comma followed by a message enclosed in apostrophes, a severity code followed by a message, or an asterisk followed by a message. In all cases, the message is printed in the assembly listing source code. The severity code indicates the danger of the error that occurred. The severity code is a decimal value of zero to 255.

If you want to indicate a severity code of 1, you leave a blank space (Δ) followed by the error message enclosed in apostrophes. An asterisk used as the severity code indicates that the message following it is informational and not an error. Any of these specifications causes the message to be printed in the assembly listing. Also, MNOTE lines are flagged as errors and listed in the diagnostics portion of the assembly listing if they do not have an asterisk in operand 1. Messages which are preceded by an asterisk are not flagged or listed in the diagnostics because they are not errors.

Variable symbols can be used as operands in a MNOTE statement.

The following example contains a MNOTE statement that generates a message in the source code of the assembly listing, and causes the line of code to be flagged in error. The error also is listed in the diagnostics portion of an assembly listing produced by this code.

Example:

```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10          16
---------------------------------------------------------------------
          MACRO
          MOVE        &A,&B,&C
          AIF         (&A GT 0 AND &A LT 257).TAG
          MNOTE       'INVALID LENGTH SPECIFIED'
          MEXIT
.TAG      MVC         &B(&A),&C
          MEND
```

27.3.2. PNOTE Message Statements (PNOTE)

A PNOTE message statement is used to generate an error message or a comment. A PNOTE statement is used in a macro definition or a source code statement.

The PNOTE source code statement format is:

LABEL	△OPERATION△	OPERAND
unused	PNOTE	{ * / 'e' } ,'m'

In this format, there are two operand fields. In the first field, you can specify an asterisk to indicate that the message is informational and not an error; or you can specify a character expression containing up to six characters. The second operand field contains the message. It can contain up to 79 characters. Regardless of the choice you make for the first operand, the message is printed in the assembly listing source code. If it does not contain an asterisk as operand 1, a PNOTE statement is flagged as an error and listed in the diagnostics portion of the assembly listing. If there is an asterisk in the first operand field, the line is not flagged or listed in diagnostics. This is done because the asterisk indicates that the message is not an error.

Variable symbols can be used as operands in a PNOTE statement.

27.3.3. Comments Statement

A comments statement written within a macro definition causes the assembler to generate comments on the output listing. This type of comments statement is written with an asterisk in column 1 of the assembler coding form followed by the comment.

A special form of the comments statement also is available for use within macro definitions. It is used to include comments in a macro definition that are not to be generated in the output listing. This comments statement is written with a period in column 1 of the assembler coding form, followed by an asterisk (*) in column 2, followed by the comment.

Neither comments statement form can be created by substitution for variable symbols. Substitution for variable symbols is not performed on comment lines.

Examples:

```
LABEL   ΔOPERATIONΔ                  OPERAND
1        10      16
-------------------------------------------------------------
         MACRO
         GEN
*        THIS COMMENT WILL BE GENERATED ON THE LISTING
.*       THIS COMMENT IS FOR INTERNAL USE, NOT GENERATED
         MEND
```

27.4. REPETITIVE CODE GENERATION

A section of code can be generated one or more times. The statements DO and ENDO specify the code you want and the number of times you want it to be generated.

27.4.1. Define Start of Range (DO)

The *define start of range* (DO) statement defines the starting point of the code and the number of times it is to be generated. The format is:

LABEL	ΔOPERATIONΔ	OPERAND
[&varisymb]	DO	a

where:

&varisymb
Is an optional variable symbol.

DO
Defines the operation.

a
Is a valid SET expression (27.1) written in a macro definition.

The expression in the operand field indicates the number of times the source code statements following the DO statement are produced in the object code. All lines of coding appearing between a DO statement and its associated ENDO statement (27.4.2) are generated. The value of the expression in the operand field may be any value from 0 to $2^{23}-1$. If the value of the expression is negative, the DO statement is flagged and ignored (that is, treated as if the value has been 1).

The set of statements between the DO statement and its associated ENDO statement are said to be within the range of the DO statement. Any valid source code statement may be within the range of a DO statement including other DO statements with their corresponding ENDO statements. DO statements may be nested up to 10 levels.

A variable symbol may be declared in the label field of the DO statement. The rules for writing variable symbols are described in Section 26. When the variable symbol in the label field is specified, it is used as a counter for the number of times a set of lines within the range of a DO statement has been generated. The value of this variable symbol is 1 the first time through the set of statements; 2 the second time through; and so forth. It is referenced in the same manner as a SETA symbol (27.1.5). An example of a DO statement is shown in 27.4.2.

27.4.2. Define End of Range (ENDO)

The *define end of range* (ENDO) statement is used to indicate the end of the range of a DO statement. The format is:

LABEL	ΔOPERATIONΔ	OPERAND
unused	ENDO	unused

DO and ENDO statements must be paired. For every DO statement, there must be an ENDO statement to define the end of the range.

Examples:

```
      LABEL   ΔOPERATIONΔ                   OPERAND
      1        10        16

 1.   &DO1     DO        5
 2.            .
 3.            .
 4.            .
 5.            ENDO
 6.            .
 7.            .
 8.   &DO2     DO        10
 9.            .
10.            .
11.            .
12.   &DO3     DO        3
13.            .
14.            .
15.   &DO4     DO        5
16.            .
17.            .
18.            .
19.            ENDO
20.            ENDO
21.            .
22.            ENDO
```

Lines 2, 3, and 4 are produced in the assembler output five times.

Lines 9, 10, 11, 21, and the lines produced by the operation of the DO directives on lines 12 and 15 are produced in the assembler output 10 times.

Within each of the 10 sets of code produced by the DO directive on line 8, lines 13, 14, and the lines produced by the operation of the DO directive on line 15 are generated in the assembler output three times.

Within each of the 30 sets of code produced by the DO directives on lines 8 and 12, lines 16, 17, and 18 are generated in the assembler output five times.

27.4.3. Conditional Assembly Control Counter (ACTR)

You use the ACTR statement to limit the number of AGO, AIF, GOTO, AGOB, AIFB, and GO statements that may be processed by the assembler either within a macro or within the source program.

The ACTR statement source code format is:

LABEL	△ OPERATION △	OPERAND
unused	ACTR	SETA expression

The ACTR statement must be written immediately following the local and global symbol declarations in either the source program or in a macro definition. There can be a separate ACTR statement in the source program and in each macro definition.

The value of the expression in the operand field may be any positive value from 1 to $2^{23}-1$. The value specified in the operand field causes a counter to be set to that value. This counter is decremented by 1 for each AGO, AGOB, or GOTO statement that is processed for each AIF or AIFB statement whose evaluation resulted in a true condition and for each time that the range of a DO statement is generated.

If the counter is zero prior to decrementing, the following occurs. If a macro is being processed, its processing and that of any macros above it in a nest are terminated. The next statement to be processed is in the source code following the macroinstruction that initiated the nest. If the source code is being processed (outside a macro definition), an END directed is generated. The assembly continues with only that portion of the program generated thus far.

If an ACTR statement is not written, the value of the counter is 4096_{10}.

27.5. ATTRIBUTE REFERENCES

The assembler assigns certain attributes to symbols and macro call operands that you may refer to in conditional assembly statements. These attributes are type (T), length (L), scale (S), integer (I), count (K), and number (N).

You can specify attributes in conditional assembly statements to control logic, which in turn can control the sequence and contents of the inline expansion code generated from model statements. Each kind of attribute has a specific purpose, which determines when you use it. The format of an attribute reference is as follows:

LABEL	△OPERATION△	OPERAND
[symbol]	conditional assembly operation code	{ T, L, S, I, K, N } { symbol, &symbol }

The attribute notation (T, L, S, I, K, or N) denotes which attribute of a symbol or parameter you are using. The symbol or parameter is a reference to the data or field which possesses the attribute. The operation code must be a conditional assembly operation code except when you are using the length attribute. (See 4.4.5 for a discussion of the use of length-attribute references in program source code.)

The origin of an attribute value is always either a symbol or a parameter. Table 27—3 gives the restrictions for using a symbol or parameter as the reference to obtain a particular data attribute. Whether a symbol or parameter can be used in an attribute reference depends on where it is referenced. If an attribute reference is made in macro source code (from inside a macro definition), a symbol may be referenced for any data attribute except K or N. A symbol cannot be used in a count or number attribute reference in macro source code because, when K or N is used inside a macro definition, the only data that can be referenced is an operand field in the macroinstruction call. Any one of the valid attributes can be acquired for a symbol or &SYSLIST. A SET symbol and the system variable symbols listed in Table 27—3 can only be used in the T and K attribute references when in macro source code. You can get all but K or N attributes of a symbol in program source code by using the symbol in the attribute reference. Macroinstruction operands cannot be referenced from program source code so a symbolic parameter or a &SYSLIST cannot be part of an attribute reference in program source code. However, A SET symbol and the system variable symbols listed in Table 27—3 can be used in an attribute reference in program source code.

Table 27—3. Valid Attribute Reference Applications

Attribute						Reference	Location
T	L	S	I	K	N		
√	√	√	√			Symbol	Macro source code
√				√		Set symbol	
√	√	√	√	√	√	Symbolic parameter	
√	√	√	√	√	√	&SYSLIST	
√				√		&SYSNDX, &SYSPARM, &SYSJDATE, &SYSECT, and &SYSTIME	
√	√	√	√			Symbol	Program source code
√				√		SET symbol	
√				√		&SYSPARM, &SYSDATE, &SYSJDATE, and &SYSTIME	

√ = valid application

There are two requirements that must be met before using symbols in attribute references. First, the symbol must appear either in the operand field of an EXTRN directive used outside of a macro or in the label field of at least one assembler directive or instruction outside a macro. Second, there must not be any variable symbol in the source line in whose label field the symbol appears. In regard to the call operand attributes, you must abide by the same criteria in addition to the following. The operand must be a symbol and it may not be one generated by variable symbol replacement. The attributes of the operand are really the attributes of the symbol itself. A nested call operand may be a symbolic parameter whose attributes are then the same as the corresponding outer operand. You cannot use a length attribute if the type attribute is J, M, N, O, T, or U (see 27.5.1).

Since a call operand may be a sublist, you can also refer to attributes of a sublist or each individual parameter in the sublist. When you refer to these attributes, they will be assigned the same value as the first parameter in the sublist.

You can refer to attributes on conditional directives both inside and outside of macros. Symbols that appear in the label field of instructions generated by a macro are not assigned attributes. If a SET symbol appears in the operand entry of a macroinstruction, attribute information is not provided and the operand may not be accessed as a sublist.

27.5.1. Type Attributes

You can use the type attribute to test for the characteristic of the operand or symbol. This is done by writing a T followed by the symbol or symbolic parameter to be tested. This can also be used in SETC directive operand fields or as character expressions in SETB and AIF directive operand fields. Table 27—4 summarizes the type attributes and the circumstances under which they are produced.

Table 27—4. Type Attributes of Symbols (Part 1 of 2)

Type	Symbol Definition	Length Specification	Alignment
A	Type A address constant	Implied	Full-word
B	Binary constant	Implied or explicit	Not applicable
C	Character constant	Implied or explicit	Not applicable
D	Double-word floating-point constant	Implied	Double-word
E	Full-word floating-point constant	Implied	Full-word
F	Full-word fixed-point constant	Implied	Full-word
G	Fixed-point constant	Explicit	Not applicable
H	Half-word fixed-point constant	Implied	Half-word
I	Machine instruction	Implied	Half-word
J	Control section name	Not applicable	Double-word
K	Floating-point constant	Explicit	Not applicable
M	Macroinstruction	Not applicable	Not applicable
N ①	Self-defining term	Not applicable	Not applicable
O ①	Omitted operand	Not applicable	Not applicable
P	Packed decimal constant	Implied or explicit	Not applicable

Table 27—4. Type Attributes of Symbols (Part 2 of 2)

Type	Symbol Definition	Length Specification	Alignment
R	Unaligned address constant (A, S, V, or Y)	Explicit	Not applicable
S	Type S address constant	Implied	Half-word
T	External symbol	Not applicable	Not applicable
U ②	Type not available	Not applicable	Not applicable
V	Type V address constant	Implied	Full-word
W	CCW statement	Implied	Double-word
X	Hexadecimal constant	Explicit or implied	Not applicable
Y	Type Y address constant	Implied	Half-word
Z	Zoned decimal constant	Explicit or implied	Not applicable

① This type attribute is produced only for macroinstruction operands.

② Type cannot be assigned. It is produced for inner and outer macroinstruction operands that cannot be assigned any other attribute, as well as for literals appearing as macroinstruction operands. symbols appearing in the label field of LTORG, ORG, or EQU directives, symbols appearing more than once in a source statement label field, and symbols appearing in the label field of DC or DS directives containing expressions or variable symbols in the modifier subfields. The latter is true even if the modifier subfield expression consists solely of self-defining terms.

27.5.2. Length Attributes

You can reference the length attribute by writing an L' followed by the symbol or parameter whose attribute you want. The length attribute has a numeric value, which refers to the number of bytes assigned by the assembler to a data field. If the length-attribute value is required for conditional preassembly processing, the symbol you specify in the attribute reference must appear in the label field of a statement in open source code. The operand field of that statement must contain a self-defining term.

The length modifier or length field must not be coded as a multiterm expression because the assembler does not evaluate this expression until assembly time.

```
LABEL    △OPERATION△                          OPERAND
1         10          16
________________________________________________________________

          .
          .
          .
DATA      DC          FL7'7E+9'
```

When the length attribute is used in conditional assembly statements, it can be specified only within an expression. Examples: L'&P(4),L'&VARY(1,2),L'&SYSLIST(5). These could be written in conditional statements such as:

```
          .
          .
          .
          AIF         (L'&P(4) LT &P(3)).PE
          .
          .
          .
&LBL      SETA        L'&TAG
          .
          .
          .
          DO          L'&NAME=6
```

An L' cannot be generated directly by a macro/proc; it can be done indirectly as follows:

```
          LCLC        &A,&B
&A        SETC        'Z'
&B        SETC        'L'''
          MVC         &A.(&B&A),X
```

After generation this would result in:

```
MVC Z(L'Z),X
```

27.5.3. Scale Attributes

You can reference scale attributes of variable symbols by coding an S' followed by the desired symbol. Scaling attributes are available only for labels of statements defining fixed-point constants. This restricts them to H, F, D, E, P, type Z, type K, and type G constants in the OS/3 assembler. The scaling attribute is the value you have assigned for the scale modifier of a fixed- or floating-point constant. This modifier is an integer used to assign a number of bits in an unnormalized constant for the fractional portion of the constant. For example, the scale modifier of a DC statement such as HF8Δ'—19.788' would be 8, since it is specifying 8 bits for the fractional part of the number. For decimal type constants the scaling attribute is the number of decimal digits to the right of the decimal point.

The following examples illustrate typical usages of scale attributes:

```
LABEL   ΔOPERATIONΔ                     OPERAND
1        10      16
------------------------------------------------------------
         .
         .
         .
         AIF     (S'&S(1) EQ S'&S(2)).S1
         .
         .
         .
&SCALE   SET     S'&P(1)
         .
         .
         .
         DO      S'&VARY 6
```

27.5.4. Integer Attributes

An integer attribute can be written with an I' followed by the symbol you wish. An integer attribute is computed from length and scaling attributes and is thus also applicable only to a symbol which is the label of a statement defining a fixed-point or floating-point constant (F, H, D, E, P, type Z, type K, and type G). A fixed-point integer attribute is equal to 8 times the length attribute, minus the scaling attribute, minus 1 (I'=8*L—S'—1). For floating point, you obtain the integer attribute, multiplying by 2, and subtracting the scaling attribute I'=2*(L'—1)—S'. Typical fixed-point constants and their computed integer attributes are:

```
         .
         .
         .
HLFWRD   DC      HS4'97.65'
FULLWRD  DC      FS12'47.8959'
```

A half-word fixed-point constant (H) would have a length attribute of 2 (L'=2) and a scale attribute specified as 4 (S'=4). Therefore, the integer attribute would be (8x2)—8—1=7. A full-word fixed-point constant would have a length of 4 (L'=4) and a scale attribute specified here as 12 (S'=12). The integer attribute in this case would be (8x4)—12—1=19.

Some floating-point constants and their computed integer attributes are:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1         10       16
-------------------------------------------------------------
          .
          .
          .
FLTHFWRD DC         ES3'64.495'
FLTFLWRD DC         DS6'17.99.2'
```

Since E is a floating-point full word, its length attribute is 4 (L'=4). The scale attribute is specified to be 3 (S'=3). Thus, the integer attribute is 2 (4—1)—3=3. When we have a floating-point double-word constant (D), its length attribute is 8 (L'=8). The scale attribute is shown to be 6. We can then compute the integer attribute to be 2(8—1)—6=8. For decimal constants, the integer attribute is the number of decimal digits to the left of the decimal point.

27.5.5. Count Attributes

You can use the count attribute of a call operand to reference the number of characters in the operand, excluding commas. This attribute is determined after substitution of any variable symbols; that is, it uses the replacement characters rather than the variable symbol to determine the count attribute. You can use the count attribute in SETA or DO operand fields, and in relational expressions of SETB and AIF operands that are within a macro.

If the operand selected is a sublist, the count attribute will include the parentheses and commas within the sublist. Examples using count attribute references in statements are:

```
          .
          .
          .
OPCT      SETA      K'&SYSLIST(1)
          .
          .
          .
&GBLB     SETB      (K'&P(3) NE Ø)
          .
          .
          .
          AIF       (K'&P(2) EQ Ø).NG
```

27.5.6. Number Attributes

For call operands you can also reference the number of operands in an operand sublist. You reference the number attribute by writing an N' followed by the symbol or parameter whose attribute you want. This number is equal to 1 plus the number of commas separating or indicating the omission of operands in the sublist. This attribute is available in SETA, DO, SETB, or AIF directives.

Examples of number attribute usage are:

```
LABEL    ΔOPERATIONΔ                        OPERAND
1        10         16
-------------------------------------------------------------------
         .
         .
         .
&NUM     SETA       N'&P
&COUNT   SETB       (N'&SYSLIST NE 2)
         AIF        (N'&SYSLIST NE 3).ERR1
         DO         N'&SYSLIST-2
         DO         N'&P>2
```

If an operand is not a sublist, the number attribute is 1. If an operand is omitted, its value is 0.

The following is an example showing all the attribute references available, along with the related constants and local directives that a SET directive requires in a program environment.

```
         PROC       &PARAM,1
DATTR    NAME
*DISPLAY ATTRIBUTES OF MACRO INSTRUCTION OPERAND
.* THIS COMMENT IS NOT GENERATED
         LCLA       &SQ,&I0,&KQ,&NQ,&LQ
         LCLC       &TQ
&I0      SETA       I'&PARAM(1)
&SQ      SETA       S'&PARAM(1)
&KQ      SETA       K'&PARAM(1)
&NQ      SETA       N'&PARAM(1)
&LQ      SETA       L'&PARAM(1)
&TQ      SETC       T'&PARAM(1)
         DC         C'&PARAM(1)' THIS IS THE OPERAND
         DC         Y(&LQ)        LENGTH ATTRIBUTE OF PARAM
         DC         Y(&KQ)        COUNT ATTRIBUTE OF PARAM
         DC         Y(&I0)        INTEGER ATTRIBUTE OF PARAM
         DC         Y(&SQ)        SCALE ATTRIBUTE OF PARAM
         DC         Y(&NQ)        NUMBER OF OPERANDS IN SUBLIST
         DC         C'&TQ'        TYPE ATTRIBUTE OF PARAM
         END
```

PART 6. ASSEMBLY LISTING

28. Organization of Listing

28.1. HEADER LINES

The assembly listing produced by the OS/3 assembler consists of five sections, each with its own headings. The five sections are:

- PREFACE

  Lists of options and assembler identification.

- CODEDIT

  Object code and source code entries

- EXTERNAL SYMBOL DICTIONARY LISTING
- CROSS-REFERENCE LISTING
- DIAGNOSTIC LISTING

28.2. PREFACE

The first section or preface section identifies the assembler variant (if any), its version and update number, and the time and date of the assembly. The format is:

UNIVAC SYSTEM OS/3 ASSEMBLER

DATE yy/mm/dd TIME — hh.mm

The preface also lists parameters indicating the assembler options selected in the job control stream, if any.

28.3. CODEDIT

In this section, source coding is printed beside the object code generated for it. The first header line in this section contains the operand field used in the TITLE statement. The header, except for the page number, will be blank. The headings in the second line are shown in Table 28—1.

Table 28—1. CODEDIT Listing Content

Second Heading Line	Field Contents
LOC	Assembler address of the object code in hexadecimal.
OBJECT CODE	Contains the object code produced from the source statement. This field is left-justified and is in hexadecimal. Machine instructions are printed in the format: mmmmmmmmmmmm Constants are printed in the form: C– –C
ADDR1	Contains the effective address in hexadecimal for the first operand of an instruction referencing main storage.
ADDR2	Contains the effective address in hexadecimal for the second operand of an instruction referencing main storage.
LINE	Contains the sequential record number in decimal. If the statement is macro-generated, each line of generated code indicates its nest level in the leftmost portion of the column containing the line number. This macro level indicator is an alphabetic character (A through Z) that represents the nest level at which the coding was generated. If more than 26 levels are nested, the indicator wraps around from Z to A. The line counter columns can record 10,000 lines of code (from 0 to 9999). If line 10,001 of code is nested, the line number will wrap around from 9999 to 0000. The next level indicator enables you to see the nest level on any line of source code.
SOURCE STATEMENT	Contains the source program statement. The listing also contains any macro-generated statements, following the source statement that called it. The assembly listing also prints embedded macro or proc call lines when one macro or proc calls another into the program. The embedded lines contain the original keyword and positional parameters supplied by the caller.

Although a sequence heading does not print in the assembly listing, you are permitted to have a sequence field that does print. Lines of code generated by a macro or proc call line retain any sequence number they had when they were originally coded. That is, the original sequence number is printed in the assembly listing on the same line as the generated statement.

28.4. EXTERNAL SYMBOL DICTIONARY LISTING

This section is optional; you can request its omission at assembly time. Entries are generated in the external symbol dictionary listing for named and unnamed section definitions (defined by CSECT statement), entry points (defined by an ENTRY statement), external symbols (EXTRN statement and type V constants), and common sections (COM statement). A description of the information which is contained in this section is given in Table 28—2.

Table 28—2. External Symbol Dictionary (ESD) Listing Content

Second Heading Line	Field Comments
SYMBOL	Contains the symbol that caused the ESD entry to be generated.
TYPE	Defines the type of entry: CSECT — Section definition (CSECT or START) ENTRY — Symbol appeared in operand field of ENTRY statement. EXTRN — External reference (symbol appeared in EXTRN operand field or defined as type V address constant) COM — Common control section definition
ESID	Two-digit external symbol identification number (in hexadecimal) of item.
ADDRESS	Contains the address of the symbol in hexadecimal for ENTRY ESDS, the starting address of the control section for CSECT and COM items, and blank for EXTRN ESDS.
LENGTH	Contains a hexadecimal value which is the assembled length (in bytes) of control on common section. Blank for ENTRY and EXTRN ESD items.

28.5. CROSS-REFERENCING LISTING

This section is optional; you can request its omission at assembly time. When this section is included, each symbol in the object program is listed in alphanumeric sequence with the statement number of the statement defining it, and the statement numbers of all references to it. Also included in the data for the symbol are the length attribute of the symbol and the value assigned to the symbol in the assembly.

The first heading line of every page contains the following:

CROSS REFERENCE

The information on each page of the cross-reference listing appears in one column on the left half of the page. If more space is needed, the assembler prints a second column of cross-reference data on the right half of the page. Each column has a heading line followed by one or more data lines. The formats of the column heading and data lines are given in Table 28—3.

Table 28—3. Cross-Reference Content

Second Heading Line	Field Comments
SYMBOL	Symbol to which cross-reference data pertains.
LENGTH	Length (in decimal) of data associated with symbol (i.e., implied length of symbol).
VALUE	Hexadecimal value of symbol.
DEFN	Statement number of statement in which symbol is defined.
(Actual line numbers of reference)	May contain from 0 to 5 entries of the form nnnnΔΔ, where nnnn is a statement number, and ΔΔ represents two blanks. Each entry represents a statement number of a line in which the symbol is referenced. Succeeding lines will be used as necessary to list all the references or duplicate definitions. Leading zeros are suppressed. Printing of the symbol cross-reference listing is in double-column format. Continuation occurs from the bottom of the left-hand column to the top of the right-hand column and from the bottom of the right-hand column to the top of the left-hand column on the next page. The last page of the listing may contain any part of a partial page.

28.6. DIAGNOSTIC LISTING

Statements containing errors are flagged and appear in the diagnostic listing. The diagnostic listing follows the assembly listing and contains a detailed accounting of any errors that occurred in the assembly. The listing contains the line number of the statement in which the error occurred, the error code, and a message indicating the cause of the error. The messages are listed in the order in which they occurred. A diagnostic listing is optional and can be suppressed by using the PARAM statement (Appendix F) with the LST=ND option in its operand field. The PARAM statement also provides the LST=DBG option for debugging a macro definition.

When a macro definition is retrieved from a library, those of its statements that contain errors are listed and flagged immediately after the END statement. If the macro definition is part of your source program, source statements in error are flagged both within the definition itself and wherever they appear in inline expansion code. In all cases macro errors are listed in the diagnostic listing.

The first heading line contains the following:

DIAGNOSTICS

The format of the second heading line and the data line contents are given in Table 28—4.

Table 28—4. Diagnostic Listing Content

Second Heading Line	Field Contents
STMT	May contain from 1 to 10 entries of the form nnnn△△, where nnnn represents a statement number of a line in which the error occurred and △△ represents two blanks. Leading zeros are suppressed.
ERROR CODE	Error code of the error in form annn.
MESSAGE	An actual error message giving details about the error.

After the listing of the diagnostics, the assembler prints the total number of statements that were in error as follows:

nnnn STATEMENTS FLAGGED IN THE ASSEMBLY — yy/mm/dd hh.ss

The final error statement message is also displayed on the console or master workstation upon completion of the assembly. This lets you know immediately if there are any diagnostic errors in your source program.

28.7. EXAMPLE OF ASSEMBLY LISTING

Section 29 contains a sample assembly listing.

PART 7. PROGRAMMING TECHNIQUES

29. Job Control Procedures

29.1. HOW TO RUN A JOB

To assemble, link edit, and execute your program, you must tell the computer what you want it to do for you. You assign peripheral devices and then request other programs and routines for use in your program. Job control is your means of communicating with the computer. Job control procedures are designed to enable you to get your program into the computer in the most efficient way. These procedures are similar to macro definitions. They generate a series of job control statements by using one calling line of code. This section includes the job control procedures you need to assemble your program into an object module, to link-edit your program into a load module, and finally, to execute it. There are job control procedures available for many other functions but they are not discussed here. The minimum number of job control statements needed to run your program are provided. For additional information on job control, refer to the interactive job control user guide.

When using a multisectioned program or unfamiliar instructions, assemble the program and correct any syntax, addressability, or other errors. Then, add the job control cards needed to link-edit and execute, and resubmit the job. Assemble your program and store the object module in a private library for execution at another time or store the object module and link it to another object module for combined execution. The system service programs user guide contains information about link-editing and how to create and access files of stored information.

29.2. INTRODUCING THE SOURCE DECK

To assemble your source program into an object code module, you need to surround your source code deck with job control statements. The job control statements needed to introduce the source deck are discussed in 29.2.1 and 29.2.2.

29.2.1. JOB Control Statement

The first job control statement in your deck is the JOB control statement, which assigns a unique name to your job. It is the only required parameter, as you can see from the format. See the interactive job control user guide for an explanation of the other parameters.

```
//[symbol]  JOB  jobname [,{P|H|N}] [,min] [,max] [,{tasks|1}] [,max-time] [,(op-list-1,...,op-list-n)]

    [,acct-no] [,nXm] [,{NOACT|NOLOG|NONE|BOTH}] [,{NOHDR|HDR}]
```

The *jobname* can have up to eight alphanumeric characters. The name you specify on the JOB control statement has no bearing on the name you assign on the START card (see 17.5) within your assembly program. The jobname parameter distinguishes one job from another. Use a unique name, since only one job can be scheduled for processing by the operating system under a name. (If two jobs have the same name, the second job would replace the first job.)

29.2.2. OPTION Job Control Statement

Following the JOB control statement, you can include an OPTION job control statement to cause a program dump at the end of your assembly listing. There are three kinds of dumps you can request, depending on the parameter you choose:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10        16

1. // OPTION DUMP

2. // OPTION JOBDUMP

3. // OPTION SYSDUMP
```

These dumps are explained in the system service programs user guide. When a program terminates normally, the OPTION job control statement alone does not produce a dump. You must have a corresponding DUMP or SNAP card within your assembly program. DUMP and SNAP are supervisor macros created to dump portions or all of your assembly program. DUMP and SNAP are explained in the supervisor macroinstructions user guide.

If your program terminates *abnormally* and you've included an OPTION job control statement, you'll get a dump following the assembly listing.

If an OPTION job control statement is not present in the control stream, the DUMP macroinstruction acts as an EOJ macroinstruction. The OPTION job control statement must also be in the job step in which you want the dump to occur. For example, if you assemble, link-edit, and execute your load module, and you want the dump to occur when you execute your load module, you place the OPTION job control statement in the job step that executes your load module, not in the one that assembles or link-edits.

29.3. ASSEMBLE; ASSEMBLE AND LINK-EDIT; OR ASSEMBLE, LINK-EDIT, AND EXECUTE

You can assemble, link-edit, and execute your program in steps or do it *all* at once. Each of these functions requires a different job control procedure (jproc) call statement. You can use any one of these jproc call statements, depending on what you want to do:

- ASM — Assembles your source deck.
- ASML — Assembles and then link-edits.
- ASMLG — Assembles, link-edits, and then executes the generated load module.

29.3.1. Assemble (ASM)

When you assemble, you create and name (either directly or indirectly) an object module. Errors incurred during assembly are flagged and listed on the printout in the diagnostics following the assembly listing. Once you have an error-free assembly, you are ready to execute. To execute, you must add the job control statements to your deck (and ASM jproc call statement) that link-edit and execute your program. Or, you can replace the ASM jproc call statement with the jproc call statement to assemble and link-edit (ASML) or assemble, link-edit, and execute (ASMLG). The latter approach is suggested. It is more practical to let the prewritten job control procedures do the work rather than having to keypunch the additional cards. If you do not use ASML or ASMLG jprocs, you will have to consult the interactive job control user guide for the additional cards needed.

Remember, the object module you produce during assembly is not saved (unless you say so with a parameter). After the assembly is complete, the object module is removed from the temporary job YRUN file. If you want to save it for later use, you must store it in a system library, or a library of your own. The librarian portion of the system service programs user guide describes system and user libraries.

29.3.1.1. ASM Jproc Call Statement

The format of the jproc call statement that generates only an assembly follows. Except for the OUT parameter, the format also applies to the ASML and ASMLG statements discussed in 29.3.2 and 29.3.3.

```
//[symbol] △ASM△  [PRNTR= { lun[,dest]
                            N[,dest]
                            20[,dest] } ]

                  [,IN= { (vol-ser-no,label)
                          (RES)
                          (RES,label)
                          (RUN,label)
                          (*,label) } ]

                  [,OUT= { (vol-ser-no,label)
                           (RES,label)
                           (RUN,label)
                           (*,label)
                           (N)
                           (RUN,$Y$RUN) } ]

                  [,LIN= ( { vol-ser-no-1,label-1      [, { vol-ser-no-2,label-2
                             RES,label-1                    RES,label-2
                             RUN,label-1                    RUN,label-2
                             *,label-1                      *,label-2
                             N                              N
                             RES,$Y$MAC }                   RES,$Y$MAC } ] ) ]

                  [,COPY= ( { vol-ser-no-1,label-1     [, { vol-ser-no-2,label-2
                              RES,label-1                   RES,label-2
                              RUN,label-1                   RUN,label-2
                              *,label-1                     *,label-2
                              N                             N
                              RES,$Y$SRC }                  RES,$Y$SRC } ] ) ]

                  [,LST= { option
                           (opt-1,...,opt-n) } ]

                  [,SCR1= { vol-ser-no
                            RES } ]

                  [,SCR2= { vol-ser-no
                            RUN } ]

                  [,ALTLOD= { (vol-ser-no,label)
                              (RES,label)
                              (RUN,label)
                              (*,label)
                              (RES,$Y$LOD) } ]
```

The symbol field of the ASM jproc call statement is an optional field. There is no space between the // and the symbol. Normally, your input (the source module) is in the form of punched cards. But, possibly, you may have stored the source module in a library. The symbol field supplies the name of the source module (one to eight alphanumeric characters). (The symbol field is only needed when you use an IN parameter.)

If no name is specified on the START directive, ASMOBJ is assigned. You can have only one unnamed assembly (the default, ASMOBJ) within a job unless all other assemblies are continuations of the first. For example, if performing two individual assemblies within the same job, proceed as follows:

```
LABEL    ΔOPERATIONΔ                         OPERAND
1         10        16                                                72
------------------------------------------------------------------------
// JOB COMSTOR
// ASM
/$
PROGA     START     Ø
           .
           .
           .
          END
/*
// ASM
/$
PROGB     START     Ø
           .
           .
           .
          END
/*
/&
// FIN
```

In this example, the first assembly is named PROGA, and the second is named PROGB. The jobname is COMSTOR. When you use the IN keyword parameter, you must also use the symbol field. (This is discussed when the IN keyword parameter is discussed.)

The keyword parameters of the ASM jproc call statement are optional. The shaded areas indicate the default values generated if you do not use the parameter. You can use statement continuation to contain all the parameters you want to specify. Following are examples of correct and incorrect coding:

Correct Example:

```
// JOB ASSEMBLE
//PROGNM ASM        PRNTR=21,IN=(DSC1,U$SRC),                    X
//1                 OUT=(DSC2,U$OBJ),                            X
//2                 LIN=(DSC1,U$MAC1,DSC2,U$MAC2)
/&
```

Incorrect Example:

You cannot break a parameter specification. The keyword and its value must be on the same card. You cannot code the IN parameter as follows:

```
LABEL    ΔOPERATIONΔ                          OPERAND
1         10          16                                                72
_____________________________________________________________________________

// JOB ASSEMBLE
//PROGNM ASM          PRNTR=21,IN=DSC1,
                      U$SRC)
```

This coding would cause an error. To continue on another card you need a nonblank character (we used X) in 72, and, in the next card, a // in columns 1 and 2 followed by a number in column 3 as shown in the correct example. The column 3 numbers must be in ascending order in the deck, in the range of 1 through 9, or they can all have the same number. If there are two continuation cards, as shown in the correct example, the first card cannot be //2 and the second card //1. But it could be //2 and //2.

The parameter definitions are as follows:

PRNTR=

Allocates a specific printer for the assembly listing.

N

If you specify N, the device assignment set for the printer is not generated by the jproc. Instead, the device assignment set for the printer is manually inserted by the user in the control stream (prior to the placement of the jproc call). This allows for the creation of specific load code and vertical format buffers (the LCB and VFB job control statements) by the user. For example:

```
//DVC 26 // VFB LENGTH=66,DENSITY=6
// LFD PRNTR
// ASM PRNTR=N
```

NOTE:

When this parameter is used, the file name for the device assignment set for the printer must be PRNTR.

lun

If you want a logical unit number other than 20, you must specify the logical unit number associated with the printer you select.

20

If you specify 20 as the logical unit number, you will get the next available printer. This parameter is optional because 20 is the default value.

dest

If you spool the printed output and want to send it to a device at a remote site, you use this parameter to specify the device's 1- to 6-alphanumeric-character destination identifier as it is defined by remote batch processing (RBP).

IN=

You only use the IN keyword parameter when the source program is not on cards. If you have stored your unassembled program on a disk, you use the IN parameter to retrieve it for assembly. When using this parameter, you must have the name of the source program you are retrieving in the symbol field of the ASM jproc call statement. The options are:

(vol-ser-no,label)

Specifies the volume serial number and the file identifier where the source module is located. For example, it could be on a disk whose volume serial number is DSC1. The disk is assigned *that* number. On the disk, the source program, named PROGNM, is stored in a library called US$SRC. To assemble the source program, specify the IN keyword parameter to define the input. This parameter must be used when making source corrections via the SKI, REC, and SEQ statements. (See F.2.)

```
LABEL    ΔOPERATIONΔ                        OPERAND
1         10        16
_____________________________________________________________

// JOB ASSEMB2A
//PROGNM ASM        IN=(DSC1,U$SRC)
              .
              .
              .
```

(RES)

Indicates you want to retrieve a source program from the system resident device (SYSRES) in the source library file (YSRC).

(RES,label)

Indicates you want to retrieve your source program from SYSRES. But it is not in YSRC; it is in a file identified by the label.

(RUN,label)

Indicates that you want to retrieve a source program from the volume containing the job's run library file (YRUN). The label is the file identifier.

(*,label)

Indicates that you want to retrieve a source program from a cataloged file. The label is the file identifier, which is all that is necessary to identify the file to the system.

OUT=

You use this parameter to store assembled object module in a library other than the job's YRUN file to save your assembly. Remember that the ASM jproc only stores your program until the job is complete. To permanently save an object module, you must put it somewhere with the OUT parameter. This also generates a PARAM OUT job control statement. The options are:

(vol-ser-no,label)

Specifies the volume serial number and file identifier where you want to store the object module. It is assumed that this file has been already allocated. If it is not, you have to supply a device assignment set to allocate this file.

(RES,label)

Indicates that you put your object module on SYSRES, but you didn't use the reserved YSRC file; you named your own file. The label is the file identifier.

(RUN,label)

Indicates you put the object module on the volume containing the job's YRUN file in the file identified by label.

(*,label)

Indicates that you want to store the object module in a cataloged file. The label is the file identifier, which is all that is necessary to identify the file to the system.

(N)

Indicates you do not want the object module stored in your own file, or retrieved from YRUN.

LIN=

You can use the macro library file (YMAC) to store the macro definitions or you can use a nonsystem library. The LIN keyword parameter identifies the library you want searched. If you do not specify this parameter, YMAC is searched. The options are:

(vol-ser-no-1,label-1[,vol-ser-no-2,label-2])

Provides the volume serial numbers and file identifiers. You can specify up to two volumes and files where macros are stored. YMAC is searched if the desired macros are not found elsewhere.

(RES,label-1[,RES,label-2])

Specifies two library files to be searched for macros; both files are on SYSRES.

(RUN,label-1[,RUN,label-2])

Specifies two library files to be searched for macros; both files are on the volume containing the job's run library file (YRUN).

(*,label-1[,*,label-2])

Specifies two library files to be searched for macros; you use this format when the files are cataloged, so you need only specify the file identifiers.

([N])

Indicates that no macros should be retrieved.

COPY=

Identifies stored source programs that are to be copied into another source program. You can specify two volumes and two files. If you do not specify this parameter, YSRC is searched for any source programs named in an assembler COPY directive. This parameter works with the COPY directive, which names the source programs you want to copy into your program. The options are:

(vol-ser-no-1,label-1[,vol-ser-no-2,label-2])
Provides the volume serial numbers and file identifiers. You can specify up to two volumes and files where source programs are stored.

(RES,label-1[,RES,label-2])
Specifies two library files containing stored source programs; both files are on SYSRES.

(RUN,label-1[,RUN,label-2])
Specifies two library files containing stored source programs; both files are on the volume containing the job's run library file (YRUN).

(*,label-1[,*,label-2])
Specifies two library files containing stored source programs; you use this format when the files are cataloged, so you need only specify the file identifiers.

({N})
Specifies that no source programs should be copied.

LST=

Alters the normal assembly listing and generates a PARAM LST job control statement. If you do not use this keyword parameter, the assembly listing contains a source, object, cross-reference, and diagnostic listing. You can specify LST options in either of two ways:

option

Specifies a single option.

(opt-1,...,opt-n)

Specifies more than one option. The parentheses are required.

The options are:

N

Specifies that no assembly listing is produced.

NC

Specifies that no cross-reference listing is produced.

ND

Specifies that no diagnostic listing is produced.

NR

Specifies that the cross-reference listing is to contain only symbols that each have at least one reference. The NC option, if specified with NR, always overrides it.

DBG

Specifies the proc/macro debug mode feature, which shows the expansion of any macro or proc called within the use program. Source code is listed twice and shows any appropriate substitutions. Any statements causing error diagnostics show the exit line in error. For more information, see the LST keyword parameter discussion in Appendix F.

SCR1= and SCR2=

The assembler needs two scratch work areas to perform its calculations. Normally, the SYSRES device is used for one file and the volume containing the job YRUN file for the other file. This is what is generated by default. But, you can use a different volume if desired.

SCR1=vol-ser-no and **SCR2=vol-ser-no**

Specify the volume serial numbers of the work files. The default for SCR1 is SYSRES, and the default for SCR2 is the job's YRUN file.

ALTLOD=

Identify the library file from which the assembler will be loaded if it is other than the YLOD file on SYSRES.

(vol-ser-no,label)

Specifies the name of the library file from which the assembler will be loaded and the volume serial number of the volume containing that file.

(RES,label)
Indicates that the assembler is to be loaded from the file identified by label on SYSRES.

(RUN,label)
Indicates that the assembler is to be loaded from the file identified by label on volume containing the job's run library file (YRUN).

(*,label)
Indicates that the assembler is to be loaded from a cataloged file specified by label.

29.3.2. Assemble and Link-Edit (ASML)

When you assemble and link-edit your program, you create and name an object module and a load module. The load module is *temporarily* stored in the job's YRUN file. The job is not executed. You only assemble and link-edit without executing if you are referencing something in your program that is defined in another program. For example, you may have external references (EXTRN) and supervisor routines (DUMP), etc. At link-edit time, cross-referencing between object modules is completed and loose ends are tied together. If you can link-edit without error, you are one step closer to completing the job.

The load module is saved temporarily in the job YRUN file, thus enabling all separate object modules to communicate while the job is being run. Once the link edit is complete, the load module is removed from YRUN. The load module can be stored permanently as discussed in the linkage editor portion of the system service programs user guide. It is important to realize that you are using more main storage, for a longer period of time, when you assembly and link than when you just assemble. When you use the ASML jproc call statement, you cannot use the OUT parameter to define an output library and save the generated object module.

29.3.2.1. ASML Jproc Call Statement

The format of the ASML jproc call statement generates an assembly and then automatically link-edits the object module. The options shown are described in 29.3.1.1. Again note that the OUT option is not included.

```
//[symbol] ΔASMLΔ  [PRNTR= { lun[,dest] } ]
                          { N[,dest]   }
                          { 20[,dest]  }

                   [,IN= { (vol-ser-no,label) } ]
                         { (RES)              }
                         { (RES,label)        }
                         { (RUN,label)        }
                         { (*,label)          }
```

(continued)

```
[      ( ( vol-ser-no-1,label-1 )   [ ( vol-ser-no-2,label-2 ) ] ) ]
[      ( ( RES,label-1          )   [ ( RES,label-2          ) ] ) ]
[ ,LIN=( ( RUN,label-1          )   [ ( RUN,label-2          ) ] ) ]
[      ( ( *,label-1            ) , [ ( *,label-2            ) ] ) ]
[      ( ( N                    )   [ ( N                    ) ] ) ]
[      ( ( RES,$Y$MAC           )   [ ( RES,$Y$MAC           ) ] ) ]

[       ( ( vol-ser-no-1,label-1 )   [ ( vol-ser-no-2,label-2 ) ] ) ]
[       ( ( RES,label-1          )   [ ( RES,label-2          ) ] ) ]
[ ,COPY=( ( RUN,label-1          )   [ ( RUN,label-2          ) ] ) ]
[       ( ( *,label-1            ) , [ ( *,label-2            ) ] ) ]
[       ( ( N                    )   [ ( N                    ) ] ) ]
[       ( ( RES,$Y$SRC           )   [ ( RES,$Y$SRC           ) ] ) ]

[ ,LST= { option              } ]
[       { (opt-1,...,opt-n)   } ]

[ ,SCR1= { vol-ser-no } ]
[        { RES        } ]

[ ,SCR2= { vol-ser-no } ]
[        { RUN        } ]

[         { (vol-ser-no,label) } ]
[         { (RES,label)        } ]
[ ,ALTLOD={ (RUN,label)        } ]
[         { (*,label)          } ]
[         { RES,$Y$LOD         } ]
```

This jproc call statement is useful when you are still testing your program, since it lets you see the output of your job without reserving a file for it. Once the job is executing properly, you can allocate a file and store the load module by using the linkage editor. This jproc call statement is also useful for infrequently run jobs.

The functions and use of the linkage editor are explained in the system service programs user guide. There also is a jproc for executing the linkage editor, just as there is one for executing the assembler. This jproc call allows you to do more with the generated load module than either the ASML or ASMLG jprocs, such as storing the load module in a library. (This does not mean you cannot store your load module when you use either the ASML or ASMLG jproc call statements; it only means you cannot do it by the jproc call itself). You also can use the LINKOP linkage editor control statement but this involves more coding, and the jprocs are designed to reduce coding. The jproc call statement for the linkage editor is included in the interactive job control user guide.

29.3.3. Assemble, Link-Edit, and Execute (ASMLG)

When you use the ASMLG jproc call statement, you create and name both an object module and a load module, temporarily store it in the job YRUN file, and then execute it. The load module is stored in the job YRUN until execution of the job is completed. You cannot use the ASMLG jproc call when the system has the shared data management feature because job control must first scan the load modules in YLOD for this feature. The GO option associated with the ASMLG jproc call cannot be used. If you use the ASM jproc call with a separate LINK jproc call or the ASML jproc call, and want to execute the program using the shared data management feature, you must provide a separate EXEC statement.

29.3.3.1. ASMLG Jproc Call Statement

The format of the ASMLG jproc call statement generates an assembly, creates a load module, and executes your program. The options shown in this format are described in 29.3.1.1. Notice, however, that the ALTLOD parameter default is RUN,YRUN when using the ASMLG jproc call statement. The OUT keyword parameter does not apply to the ASMLG jproc call statement, only to the ASM jproc call statement.

```
//[symbol] ΔASMLGΔ  [PRNTR= { lun[,dest] } ]
                             { N[,dest]   }
                             { 20[,dest]  }

                    [,IN= { (vol-ser-no,label) } ]
                          { (RES)              }
                          { (RES,label)        }
                          { (RUN,label)        }
                          { (*,label)          }

                    [,LIN= ( { vol-ser-no-1,label-1 } [, { vol-ser-no-2,label-2 } ] ) ]
                           ( { RES,label-1          }    { RES,label-2          }   )
                           ( { RUN,label-1          }    { RUN,label-2          }   )
                           ( { *,label-1            }    { *,label-2            }   )
                           ( { N                    }    { N                    }   )
                           ( { RES,$Y$MAC           }    { RES,$Y$MAC           }   )

                    [,COPY= ( { vol-ser-no-1,label-1 } [, { vol-ser-no-2,label-2 } ] ) ]
                            ( { RES,label-1          }    { RES,label-2          }   )
                            ( { RUN,label-1          }    { RUN,label-2          }   )
                            ( { *,label-1            }    { *,label-2            }   )
                            ( { N                    }    { N                    }   )
                            ( { RES,$Y$SRC           }    { RES,$Y$SRC           }   )
```

(continued)

```
[,LST={option          }]
      {(opt-1,...,opt-n)}

[,SCR1={vol-ser-no}]
       {RES       }

[,SCR2={vol-ser-no}]
       {RUN       }

[         {(vol-ser-no,label)}]
[         {(RES,label)       }]
[,ALTLOD= {(RUN,label)       }]
[         {(*,label)         }]
[         {(RUN,$Y$RUN)      }]
```

NOTE:

Calling the ASML or ASMLG jproc more than once in a single job may create unpredictable results. As called by either of these jprocs, the linkage editor includes all object modules currently residing in the job YRUN file, even those modules generated by ASML or ASMLG jprocs called earlier in the job. In this way, you may accidentally include object code that has no place in your intended program. To avoid this problem you should:

1. *Call only one ASML or ASMLG jproc per job; or*
2. *Assemble individual object modules separately using the ASM jproc, then link the modules together with one of the linkage editor jprocs described in the current version of the job control and system service programs user guides; these linkage editor jprocs give you more control over the generated load module.*

29.4. START-OF-DATA JOB CONTROL STATEMENT (/$)

A start-of-data job control statement must precede the first card of the source program or any macros being submitted with the source program.

```
LABEL    ΔOPERATIONΔ                      OPERAND
1         10        16
-------------------------------------------------------------
// JOB MYPROG
// ASM
/$
PROGRAM  START    0
          .
          .
          .
          END
/*
```

29.5. FOLLOWING THE SOURCE DECK

Following the END card in the source deck, you need job control cards to tell the computer that you have completed part or all of your job.

29.5.1. End-of-Data Job Control Statement (/*)

An end-of-data job control statement follows the END directive of the source program.

```
// JOB MYPROG
// ASMLG
/$
PROGNM   START     Ø
         .
         .
         .
         END
/*
```

29.5.2. End-of-Job Control Statement (/&)

An end-of-job control statement terminates the job which was started by the last JOB control statement. It indicates that all job steps have been completed.

```
LABEL    ΔOPERATIONΔ                          OPERAND
1        10        16
---------------------------------------------------------------
// JOB COMSTOR
// ASM
/$
PROGA    START     Ø
         .
         .
         .
         END
/*
// ASM
/$
PROGB    START     Ø
         .
         .
         .
         END
/*
/&
// FIN
```

29.5.3. Terminate-the-Card-Reader Job Control Statement (//FIN)

A terminate-the-card-reader job control statement ends a card reader operation. This statement follows the end-of-job control statement as shown in the coding form in 29.5.2.

29.5.4. Setting the UPSI Byte

If any errors are detected in your program while it is being assembled, the assembler sets the User Program Switch Indicator (UPSI) byte according to OS/3 system standards to indicate the type of errors that occurred.

UPSI Byte	Setting	Meaning
Bit 0	1 (X'80')	Catastrophic errors were detected in the source program that prevented completion of the requested function. An object module was not generated.
	0	No catastrophic errors were detected.
Bit 1	1 (X'40')	Serious errors were detected that may have affected (but not prevented) the completion of the requested function. An object module was generated but the results could be unpredictable.
	0	No serious errors were detected.
Bit 2	1 (X'20')	Diagnostic errors were detected but the completion of the requested function was not affected. The source program contains a legal but potentially undesirable situation. An object module was generated.
	0	No diagnostic errors were detected.

NOTE:

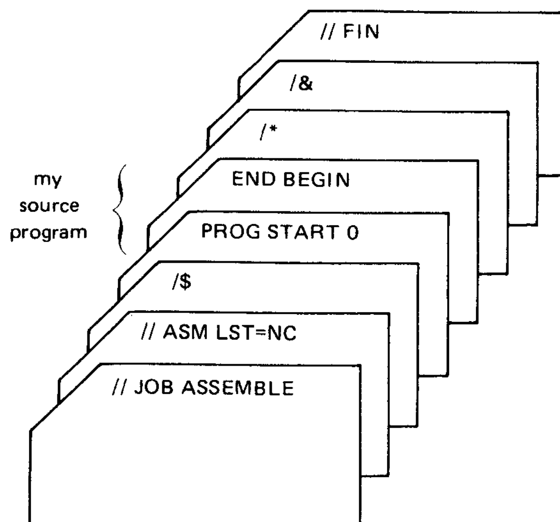
In the event of a program check, the UPSI byte setting X'80', in combination with a supervisor macro (STXIT), provides continuation to the next job step rather than complete termination of the job stream. The job step that resulted in the program check is cancelled and a dump is produced. For further details about STXIT, see the supervisor macroinstructions user guide/programmer reference.

29.6. SUMMARY OF JOB CONTROL PROCEDURE

The following card deck sketches and sample printouts demonstrate the difference in an output listing when the same source program is assembled; assembled and link-edited; and assembled, link-edited, and executed in three separate steps.

29.6.1. Assembly

The following source deck requests an assembly:

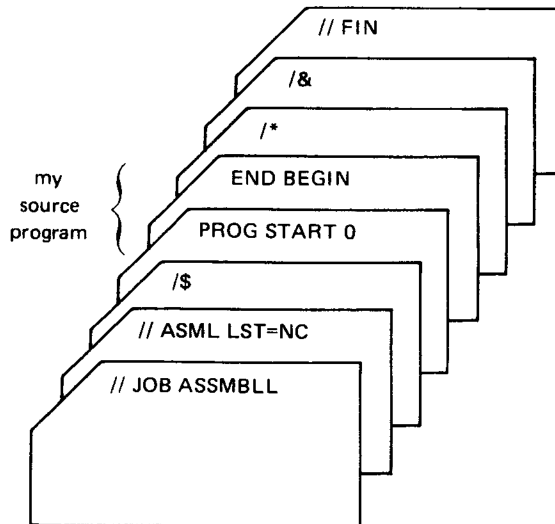


The listing produced by the assembly source deck is as follows. The headings are explained in Section 28.

```
UNIVAC SYSTEM  OS/3 ASSEMBLER                                                                                         VER791026
DATE- 80/01/07 TIME- 09.45


ASSEMBLER CONTROL STATEMENTS ENCOUNTERED AND PROCESSED AS FOLLOWS-
                                          EXTERNAL SYMBOL DICTIONARY                                                  PAGE    1
  SYMBOL.  TYPE.  ESID.  ADDRESS.  LENGTH.


  PROG     CSECT  01     000000    000060
  OUT      ENTRY  01     000054
  OUTRIB   ENTRY  01     000080
                                                                                                                      PAGE    1


 LOC.  OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT                                                  OS/3 ASM 80/01/07
000000                                1 PROG     START 0
000000 0560                           2 BEGIN    BALR  6,0
000002                                3          USING *,6
000002 47F0 6010             00012    4 BRANCH   B     *+16
000006 C1C2C3C440404040               5          DC    CL8'ABCD'
00000E C5C6C7C840404040               6          DC    CL8'EFGH'
000016 D203 6008 600C 0000A 0000E     7          MVC   BRANCH+8(4),BRANCH+12
                                      8          OPEN  OUT,(OUTRIB)
00001C                                9+         CNOP  0,4                                                       P@000960
00001C 4510 6026             00028   10+         BAL   1,*+12                                                    P@001150
000020 81                            11+         DC    X'81'                                                     P@001160
000021 000054                        12+         DC    AL3(OUT)                                                  P@001170
000024 80                            13+         DC    X'80'                                                     P@001180
000025 000080                        14+         DC    AL3(OUTRIB)                                               P@001190
000028 0A26                          15+         SVC   38 ISSUE SVC                                              P@002170
00002A D207 6096 6004 00098 00006    16          MVC   BUF(8),BRANCH+4
                                     17          DMOUT OUT,BUF
000030                               18+         DC    0Y(0) *                   SET ALIGNMENT                   P@I00690
000030 5810 60A6             000A8   19+         L     1,=A(OUT) *             LOAD R1$, CDIB ADDRESS            P@I00710
000034 5800 60AA             000AC   20+         L     0,=A(BUF) *             LOAD R0$, WORKAREA ADDRESS        P@I00740
000038 9220 1002       00002         21+         MVI   2(1),X'20' *           SET FUNCTION CODE                  P@I00890
00003C 9200 1003       00003         22+         MVI   3(1),0 *           SET FUNCTION CONTROL BYTE 1            P@I00891
                                     23+         SCALL 47                                                        P@I00920
000040                               24+         DS    0H                                                        P$S00810
000040 0AEF                          25+         SVC   239                                                       P$S01380
000042 10                            26+         DC    YL1(16)                                                   P$S01390
000043 2F                            27+         DC    YL1(47)                                                   P$S01395
000044 0A19                          28+         SVC   25                                                        P$S01590
000046 0700                          29+         NOPR  0                                                         P$S01650
000048 0A1C                          30+         SVC   28                                                        P$S01660
000048                               31+         ORG   *-2
000048 0A85                          32+         SVC   133
                                     33          CLOSE OUT
00004A                               34+         DC    0Y(0)                                                     P@000280
00004A 5810 60A6             000A8   35+         L     1,=A(OUT) LOAD R1 WITH FILENAME ADDRESS                   P@002020
00004E 0A27                          36+         SVC   39 ISSUE SVC                                              P@002030
                                     37          EOJ
000050                               38+         DS    0H                                                        EOJ00050
000050 0A1A                          39+         SVC   26                                                        EOJ00070
                                     40 OUT      CDIB
                                     41+         ENTRY OUT
000054                               42+         DS    0F
000054 112C                          43+OUT      DC    X'112C' *                 CDIB ID AND LENGTH
000056 0000                          44+         DC    2X'0'
000058 D6E4E34040404040              45+         DC    CL8'OUT' *              FILENAME
000060 0000000000000000              46+         DC    8F'0'
                                     47 OUTRIB   RIB   BFSZ=16,IOA1=BUF,RCSZ=16,TYPEFLE=OUTPUT,RCFM=FIXUNB
                                     48+         PRINT OFF                                                       RIB00430
                                    564+         PRINT ON                                                        RIB00450
000080                              565+OUTRIB   DS    0F                                                        RIB00460
                                    566+         ENTRY OUTRIB                                                    RIB00480
                                                                                                                 PAGE    2


 LOC.  OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT                                                  OS/3 ASM 80/01/07
000080 A000                         567+         DC    AL2(RB$STRT) *                  RIB START                 RIB00500
000082 0402                         568+         DC    AL2(RB$BFSZ) *                   BFSZ HIC                 RIB01410
000084 0010                         569+         DC    AL(L'RB$BFSZ)(16) *          BUFFER SIZE                  RIB01420
000086 0803                         570+         DC    AL2(RB$IOA1) *                   IOA1 HIC                 RIB02000
000088 000098                       571+         DC    AL(L'RB$IOA1)(BUF-0) *  IOA1 ADDRESS                      RIB02010
                                    572+         EXTRN BUF                                                       RIB02040
00008B 2202                         573+         DC    AL2(RB$RSZ) *                    RCSZ HIC                 RIB05100
00008D 0010                         574+         DC    AL(L'RB$RSZ)(16) *           RCSZ                         RIB05110
00008F 2101                         575+         DC    AL2(RB$RCFM) RCFM HIC                                     RIB05660
000091 04                           576+         DC    AL(L'RB$RCFM)(RB$FXUN) *         RCFM=FIXUNB              RIB05670
000092 5601                         577+         DC    AL2(RB$TYPE) *                   TYPEFLE HIC              RIB19970
000094 80                           578+         DC    AL(L'RB$TYPE)(RB$OUT) *          TYPEFLE=OUTPUT           RIB19990
000095 A100                         579+         DC    AL2(RB$REND) *                 RIB END                    RIB31330
000098                              580          DS    0F
000098                              581 BUF      DS    ZL16
                                    582          END
0000A8 00000054                     583                =A(OUT)
0000AC 00000098                     584                =A(BUF)
NO STATEMENTS FLAGGED IN THIS ASSEMBLY        -80/01/07  09.53-
```

29.6.2. Assembly and Link-Edit

The following source deck requests an assembly and link-edit:



The listing produced by this deck is as follows:

Assembly Listing:

```
UNIVAC SYSTEM  OS/3 ASSEMBLER
DATE- 80/01/21 TIME- 02.17                                                                                  VER791026

ASSEMBLER CONTROL STATEMENTS ENCOUNTERED AND PROCESSED AS FOLLOWS-

                                                  EXTERNAL SYMBOL DICTIONARY                                PAGE    1
   SYMBOL.  TYPE.  ESID.  ADDRESS.  LENGTH.

   PROG     CSECT  01     000000    000080
   OUT      ENTRY  01     000054
   OUTRIB   ENTRY  01     000080

                                                                                                            PAGE    1

 LOC.  OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT                                       OS/3 ASM 80/01/21
 000000                                1 PROG     START 0
 000000 0560                           2 BEGIN    BALR  6,0
 000002                                3          USING *,6
 000002 47F0 6010            00012     4 BRANCH   B     *+16
 000006 C1C2C3C440404040               5          DC    CL8'ABCD'
 00000E C5C6C7C840404040               6          DC    CL8'EFGH'
 000016 D203 6008 600C 0000A 0000E     7          MVC   BRANCH+8(4),BRANCH+12
                                       8          OPEN  OUT,(OUTRIB)
 00001C                          A     9+         CNOP  0,4                                               Pa000960
 00001C 4510 6026            00028 A  10+         BAL   1,*+12                                            Pa001150
 000020 81                       A    11+         DC    X'81'                                             Pa001160
 000021 000054                   A    12+         DC    AL3(OUT)                                          Pa001170
 000024 80                       A    13+         DC    X'80'                                             Pa001180
 000025 000080                   A    14+         DC    AL3(OUTRIB)                                       Pa001190
 000028 0A26                     A    15+         SVC   38 ISSUE SVC                                      Pa002170
 00002A D207 6096 6004 00098 00006    16          MVC   BUF(8),BRANCH+4
                                      17          DMOUT OUT,BUF
 000030                          A    18+         DC    CY(0) *                 SET ALIGNMENT             PaI00690
 000030 5810 60A6            000A8 A  19+         L     1,=A(OUT) *           LOAD R1$, CDIB ADDRESS      PaI00710
 000034 5800 60AA            000AC A  20+         L     0,=A(BUF) *           LOAD R0$, WORKAREA ADDRESS  PaI00740
 000038 9220 1002      00002     A    21+         MVI   2(1),X'20' *         SET FUNCTION CODE            PaI00890
 00003C 9200 1003      00003     A    22+         MVI   3(1),0 *           SET FUNCTION CONTROL BYTE 1    PaI00891
                                 A    23+         SCALL 47                                                PaI00920
 000040                          B    24+         DS    0H                                                P$S00810
 000040 0AEF                     B    25+         SVC   239                                               P$S01380
 000042 10                       B    26+         DC    YL1(16)                                           P$S01390
 000043 2F                       B    27+         DC    YL1(47)                                           P$S01395
 000044 0A19                     B    28+         SVC   25                                                P$S01590
 000046 0700                     B    29+         NOPR  0                                                 P$S01650
 000048 0A1C                     B    30+         SVC   28                                                P$S01660
 000048                          A    31+         ORG   *-2
 000048 0A85                     A    32+         SVC   133
                                      33          CLOSE OUT
 00004A                          A    34+         DC    CY(0)                                             Pa000280
 00004A 5810 60A6            000A8 A  35+         L     1,=A(OUT) LOAD R1 WITH FILENAME ADDRESS           Pa002020
```

```
00004E 0A27                     A  36+        SVC    39 ISSUE SVC                                   PA002030
                                   37         EOJ
000050                          A  38+        DS     CH                                             EOJ00050
000050 0A1A                     A  39+        SVC    26                                             EOJ00070
                                   40 OUT     CDIB
                                A  41+        ENTRY  OUT
000054                          A  42+        DS     0F
000054 112C                     A  43+OUT     DC     X'112C' *             CDIB ID AND LENGTH
000056 0000                     A  44+        DC     2X'0'
000058 D6E4E34040404040         A  45+        DC     CL8'OUT' *           FILENAME
000060 0000000000000000         A  46+        DC     8F'0'
                                   47 OUTRIB  RIB    BFSZ=16,IOA1=BUF,RCSZ=16,TYPEFLE=OUTPUT,RCFM=FIXUNB
                                A  48+        PRINT  OFF                                            RIB00510
                                A 586+        PRINT  ON                                             RIB00530
000080                          A 587+OUTRIB  DS     0F                                             RIB00540
                                A 588+        ENTRY  OUTRIB                                         RIB00560

                                                                                                    PAGE  2

 LOC.  OBJECT CODE    ADDR1 ADDR2  LINE    SOURCE STATEMENT                                    OS/3 ASM 80/01/21
000080 A000                     A 589+        DC     AL2(RBSSTRT) *              RIB START          RIB00580
000082 0402                     A 590+        DC     AL2(RBSBFSZ) *               BFSZ HIC          RIB01580
000084 0010                     A 591+        DC     AL(L'RBSBFSZ)(16) *     BUFFER SIZE            RIB01590
000086 0803                     A 592+        DC     AL2(RBSIOA1) *               IOA1 HIC          RIB02440
000088 000098                   A 593+        DC     AL(L'RBSIOA1)(BUF-0) *   IOA1 ADDRESS          RIB02450
                                A 594+        EXTRN  BUF                                            RIB02480
00008B 2202                     A 595+        DC     AL2(RBSRSZ) *                RCSZ HIC          RIB05890
00008D 0010                     A 596+        DC     AL(L'RBSRSZ)(16) RCSZ                          RIB05900
00008F 2101                     A 597+        DC     AL2(RBSRCFM) RCFM HIC                          RIB06810
000091 04                       A 598+        DC     AL(L'RBSRCFM)(RBSFXUN) *     RCFM=FIXUNB       RIB06820
000092 5601                     A 599+        DC     AL2(RBSTYPE) *               TYPEFLE HIC       RIB21210
000094 80                       A 600+        DC     AL(L'RBSTYPE)(RBSOUT) *      TYPEFLE=OUTPUT    RIB21230
000095 A100                     A 601+        DC     AL2(RBSREND) *              RIB END            RIB33980
000098                            602         DS     0F
000098                            603 BUF     DS     ZL16
                                  604         END
0000A8 00000054                   605                =A(OUT)
0000AC 00000098                   606                =A(BUF)

                                              DIAGNOSTICS                                           PAGE  1
   LINE   ERROR CODE    MESSAGE


   NO STATEMENTS FLAGGED IN THIS ASSEMBLY        -80/01/21  02.27-
```

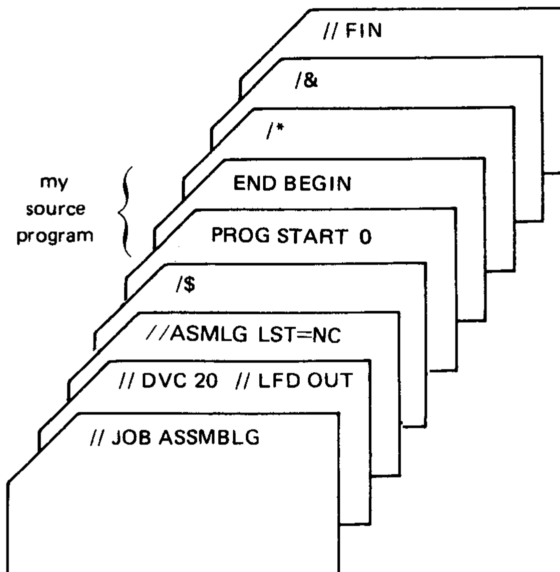
Linkage Editor Listing:

```
UNIVAC SYSTEM OS/3  LINKAGE EDITOR                                                              VER800110
DATE- 80/01/21 TIME- 02.27

CONTROL STREAM ENCOUNTERED AND PROCESSED AS FOLLOWS-

*GENERATED*  LOADM
PROG      *RUN LIBE MODULE*


                                        *DEFINITIONS DICTIONARY*

SYMBOL.  TYPE.  PHASE. ADDRESS.      SYMBOL.  TYPE.  PHASE. ADDRESS.      SYMBOL.  TYPE.  PHASE. ADDRESS.

KESALP   ENTRY  ABS    00000080      KESRES   ENTRY  ABS    00000080      OUT      ENTRY  ROOT   00000054
OUTRIB   ENTRY  ROOT   00000080      PROG     CSECT  ROOT   00000000


                                         ** ALLOCATION MAP **

                     LOAD MODULE -    LNKLOD          SIZE -    00000080

 PHASE NAME  TRANS ADDR    FLAG     LABEL     TYPE      ESID       LNK ORG     HIADDR     LENGTH      OBJ ORG
 LNKLOD00        NODE - ROOT                                       00000000    000000AF   00000080
 *** START OF AUTO-INCLUDED ELEMENTS -
 *** END OF AUTO-INCLUDED ELEMENTS -
        - 80/01/21 02.27 -         PROG       OBJ
                                   PROG      CSECT       01        00000000    000000AF   00000080    00000000
                                   OUT        ENTRY      01        00000054                           00000054
                                   OUTRIB     ENTRY      01        00000080                           00000080
              00000000


                                             FLAG CODES -
B - BLK DATA CSECT     D - AUTO-DELETED       E - EXCLUSIVE 'A' REF     G - GENERATED EXTRN      I - INCLUSIVE 'V' REF
L - DEFERRED LENGTH    M - MULTIPLY DEFINED   N - NOT INCLUDED          P - PROMOTED COMMON      R - SHARED REC PRODUCED
S - SHARED ITEM        U - UNDEFINED REF      V - VCON ITEM
*ANY OTHER CODES REPRESENT PROCESS ERRORS*


LINK EDIT OF 'LNKLOD'   COMPLETED
DATE- 80/01/21 TIME- 02.28
ERRORS ENCOUNTERED- 0000  UPSI- X'00'
```

The definitions dictionary and allocation map are explained in the system service programs user guide.

29.6.3. Assembly, Link-Edit, and Execution

The following source deck requests an assembly, link-edit, and execution:



The listing produced by this service deck is the same as the assembler and link listing, but it includes the results of the execution. This program moved letters and printed out the field containing ABCDEFGH.

29.7. RUNNING ASSEMBLER FROM A WORKSTATION

OS/3 provides you with the capability of assembling, link-editing, and executing your program interactively. This means two things:

1. you can build a control stream to execute the assembler, linkage editor, and user programs at a workstation, as opposed to punching them on cards or writing them to a diskette; and

2. you can initiate the running of the control stream from the workstation, as opposed to asking the system operator to run your job for you.

The easiest way to build a control stream from a workstation is by using the job control dialog. The job control dialog is an interactive facility of OS/3 that allows you to describe your job's requirements in English, in response to a series of questions, and then produces as its output, the job control stream needed by OS/3 to run your job. The control stream produced by the job control dialog is virtually identical to the control stream that you would need to produce if your were running your job in a batch environment. Only now, you do not have to be concerned with the intricacies of the job control language. The job control dialog eliminates this requirement.

After you have answered all the questions presented to you by the job control dialog, it builds a control stream and stores it in a permanent library file for you. From here, you can initiate its running by simply keying in the appropriate system RUN command, or if you'd rather, you can change the contents of the control stream using another interactive facility of OS/3 called the general editor.

The procedures for activating the job control dialog, initializing the running of a job, and activating the general editor are described in detail in the OS/3 workstation user guide.

More detailed descriptions of the job control dialog and the general editor are presented in the interactive job control user guide and the general editor user guide, respectively.

30. Example Macro Definitions

This section gives examples of both PROCs and MACROs. The explanation of these examples places the primary emphasis on the macro source code instead of on the resulting inline expansion source code. Descriptions of the macro definitions resolve around variable parameter replacement and variable inline expansion code caused by conditional assembly statements. Each description is accompanied by the macro source code and an example call with the inline expansion code.

30.1. SMALR/LARGR PROC (POSITIONAL PARAMETER 0)

The following example PROC selects either the smallest or largest of three positional parameters submitted in the call instruction. Two different mnemonics are provided for indicating whether the smallest or largest value is selected. The call SMALR is used for smallest value selection and the call LARGR is used for largest value selection. This PROC is a good example of using positional parameter 0.

PROC Source Code:

```
&DMY     PROC   &#,4
SMALR    NAME   BNH
LARGR    NAME   BNL
         AIF    (N'&# NE 4).ERR           Test for 4 parameters. Branch to print error message if untrue.
&DMY     ZAP    &#(1),&#(2)               Select first value.
         CP     &#(1),&#(3)               Compare to next value.
         &#(0)  *+10                      BNH or BNL or
         ZAP    &#(1),&#(3)               Select next value.
         CP     &#(1),&#(4)               Compare to next value.
         &#(0)  *+10                      BNH or BNL or
         ZAP    &#(1),&#(4)               Select next value
         MEXIT
.ERR     MNOTE  'IMPROPER PARAMETERS--NO GENERATION'
         END
```

Inline Expansion Code (smallest value):

```
SMALR  SELECT,VAL1,VAL2,VAL3
ZAP    SELECT,VAL1
CP     SELECT,VAL2
BNH    *+10
ZAP    SELECT,VAL2
CP     SELECT,VAL3
BNH    *+10
ZAP    SELECT,VAL3
```

Inline Expansion Code (largest value):

```
LARGR  SELECT,VAL1,VAL2,VAL3
ZAP    SELECT,VAL1
CP     SELECT,VAL2
BNL    *+10
ZAP    SELECT,VAL2
CP     SELECT,VAL3
BNL    *+10
ZAP    SELECT,VAL3
```

Operand 2 of the PROC statement indicates there are four positional parameters that can be passed from the call instruction, and &# is the symbolic parameter that references the positional parameters. This PROC requires all four positional parameters from the PROC. Otherwise, no inline expansion takes place and the message coded in the MNOTE is printed.

The AIF conditional assembly statement tests to see whether the four parameters are present. Whether the smallest or largest value is selected from the call instruction is dependent upon generation of the BNH or BNL instruction in the inline expansion code. This is controlled by the *call-names* SMALR or LARGR used in the two NAME statements. SMALR implements BNH for positional parameter 0 and LARGR implements BNL.

The two model statements referencing positional parameter 0 (&#(0)) are the instructions that determine smallest or largest value selection. The inline expansion code shows that the SMALR call instruction generates BNH in place of &#(0) and the LARGR call instruction generates BNL.

Positional parameter 1 of the call instructions indicates an area to receive the selected value. It is referenced in the model statements (&#(1)) in operand 1 of the add instructions and compare instructions. Positional parameters 2 through 4 of the call instructions are the values to be selected. The model statements that perform the calculations for finding the smallest or highest value have references to positional parameters 2 through 4 (&#(2) through &#(4)).

The MEXIT statement ends PROC processing at this point in the PROC, so the message isn't printed. The MNOTE is printed only if there aren't four positional parameters in the call instruction.

30.2. SMALL6/LARGE6 PROC (DO LOOP)

The following example PROC selects either the smallest or largest of the positional parameters in the call instruction, just like the SMALR/LARGR PROC. This PROC broadens the usage range by allowing the caller to specify from 3 to 100 values instead of limiting the caller to only 3 as did the SMALR/LARGR example PROC. There are two call mnemonics provided: the SMALL6 is used for small value selection and the LARGE6 is used for large value selection. The two mnemonics are provided via positional parameter 0. The SMALL6/LARGE6 PROC shows a design using DO loops to provide variable inline expansion code. The DO range is determined by the number of values in the call instruction; the more values in the call, the more lines of source code generated.

PROC Source Code:

```
&DMY     PROC  &P,100
SMALL6   NAME  BNH
LARGE6   NAME  BNL
         DO    N'&P>2
&DMY     ZAP   &P(1),&P(2)
&CT      DO    N'&P-2             )
         CP    &P(1),&P(&CT+2)    (  Inner DO loop. The instructions CP, BNH/BNL,
         &P(0) *+10               >  and ZAP will be generated the number of positional
         ZAP   &P(1),&P(&CT+2)    (  parameters in the call instruction minus two.
         ENDO                     )
         ENDO
         DO    N'&P<3
         PNOTE 'NOGEN','MINIMUM OF THREE PARAMETERS REQUIRED'
         ENDO
         END
```

Inline Expansion Code (smallest value):

```
SMALL6 SELECT,VAL1,VAL2,VAL3,VAL4,VAL5          Number of parameters = 6
ZAP    SELECT,VAL1
CP     SELECT,VAL2  )                         \
BNH    *+10         > First inner DO generation |
ZAP    SELECT,VAL2  )                          |
CP     SELECT,VAL3  )                          |
BNH    *+10         > Second inner DO generation |  DO iteration = 6-2 = 4
ZAP    SELECT,VAL3  )                          >
CP     SELECT,VAL4  )                          |  DO N'&P-2
BNH    *+10         > Third inner DO generation |
ZAP    SELECT,VAL4  )                          |
CP     SELECT,VAL5  )                          |
BNH    *+10         > Fourth inner DO generation |
ZAP    SELECT,VAL5  )                         /
```

Inline Expansion Code (largest value):

```
LARGE6 SELECT,VAL1,VAL2,VAL3,VAL4,VAL5
ZAP    SELECT,VAL1
CP     SELECT,VAL2  )      \
BNL    *+10         > 1    |
ZAP    SELECT,VAL2  )      |
CP     SELECT,VAL3  )      |
BNL    *+10         > 2    |
ZAP    SELECT,VAL3  )      >  Outer DO processed because number of parameters
CP     SELECT,VAL4  )      |  are more than 2 (DO N'&P>2).
BNL    *+10         > 3    |
ZAP    SELECT,VAL4  )      |
CP     SELECT,VAL5  )      |
BNL    *+10         > 4    |
ZAP    SELECT,VAL5  )      /
```

Operand 2 of the PROC statement indicates that the user can code up to 100 parameters in the call instruction while the symbolic parameter is &P. Two mnemonic codes can be used to call this PROC, as coded in the label field of the two NAME statements. Each is used to implement different BAL instruction codes via positional parameter 0 (&P(0)). In this PROC, positional parameter 0 is referenced only one time, while in SMALR/LARGR it was referenced twice. This is because the inner DO loop generates the required number BNH/BNL instructions to process the number of values coded in the call instruction. Of course, the inner DO is never processed unless the outer DO is.

In order for the outer DO to be processed, there must be more than two positional parameters in the call (DO N'&P>2). If there are two parameters or fewer, a zero is generated in the outer DO operand and the DO with the PNOTE is generated.

A 1 is generated in the operand field of the PNOTE DO because there are less than three parameters (DO N'&PL3), and the message in the PNOTE is generated. The call instruction for the SMALL6/LARGE6 example PROC must have a minimum of 3 positional parameters and can have a maximum of 100. This gives a range of 1 to 98 values to be tested for the highest or lowest value. If you compare this with the SMALR/LARGR example PROC, you can see that the SMALL6/LARGE6 PROC provides much more than SMALR/LARGR and has only one more line of code.

30.3. BLANK MACRO (VARIABLE INLINE EXPANSION CODE)

In the following example, MACRO blanks (X'00') the content of a specified number of bytes. There are only two positional parameters that can be submitted with the BLANK call instruction, and one is optional. Positional parameter 1 indicates the starting address of the area to be cleared, and positional parameter 2 specifies the number of bytes to be cleared. If the number of bytes in the area is less than 257, parameter 2 is optional. The BLANK MACRO is an example of variable inline expansion code. There are three basic sets of code that can be generated from this MACRO. Positional parameter 2 is used to determine which set is generated.

MACRO Source Code:

```
         MACRO
&DMY     BLANK  &#1,&#2
         LCLA   &LA1                                        }
         DO     (N'&SYSLIST>2)++(N'&SYSLIST<1)              }
         MNOTE  7,'IMPROPER PARAMETERS - NO GENERATION'     }  Error check
         MEXIT                                              }
         ENDO                                               }
         DO     N'&SYSLIST=1                                }
         AIF    (L'&#1 GT 256).A1                           }
&DMY     XC     &#1,&#1                                     }  Code generated if
         MEXIT                                              }  positional parameter 2
.A1      MNOTE  6,'LENGTH OF &#1>256 - PARAM: 2 MUST BE USED'   }  is omitted
         MEXIT                                              }
         ENDO                                               }
         AIF    (T'&#2 NE 'N').A2
&LA1     SETA   &#2
&DMY     L      15,=A(&#1)
         DO     &#2/256                                     }
         XC     0(256,15),0(15)                             }  Code generated for
         LA     15,256(15,0)                                }  more than 256 bytes
&LA1     SETA   &LA1-256                                    }
         ENDO                                               }
         DO     &LA1>0                                      }
         XC     0(&LA1,15),0(15)                            }  Code generated for less
         ENDO                                               }  than 256 bytes
         MEXIT
.A2      MNOTE  8,'PARAMETER 2 NOT NUMERIC'
         MEND
```

Inline Expansion Code:

```
BLANK  PRTBF,80
L      15,=A(PRTBF)       These two instructions were generated from the last DO because
XC     0(80,15),0(15)     positional parameter 2 is less than 256.
```

The prototype statement establishes the call mnemonics as BLANK and indicates there can be two positional parameters in the call that are referenced in the body as and . The LCLA declares &LA1 as an arithmetic set symbol. Set symbols must be declared following the heading and preceding any other model statements.

The rest of the body of this MACRO is sectioned by four DO statements. The first DO is an error exit. If either or both expressions on each side of the OR (++) operator are true, then no code is generated and the MNOTE message is printed. That is, the numeric attribute of the parameter list (N'&SYSLIST) is anything other than 1 or 2, then the blank MACRO will not work.

The second DO is processed only if there is one parameter in the parameter list (N'&SYSLIST=1). And if the area indicated by positional parameter 1 is greater than 256, the AIF statement will shunt the generation of the XC instruction (the blanking operation) and print the MNOTE message. If the numeric attribute of the parameter list is 2, then the second DO loop is not processed and the AIF after this DO is processed. This AIF tests positional parameter 2 () for a self-defining term; if it isn't, no code is generated and the MNOTE is printed.

The &LA1 set symbol is set to the value of positional parameter 2, which is the number of bytes to be cleared. Register 15 is loaded with the address of area (). The next DO loop is processed once for every multiple of 256 bytes indicated in positional parameter 2 (DO /256). If positional parameter 2 is less than 256, the next DO is processed and the last one is not.

There are three sets of codes that can be generated from the BLANK example MACRO. One set is generated if positional parameter 2 is omitted; another if positional parameter 2 is more than 256; and another if positional parameter 2 is less than 256.

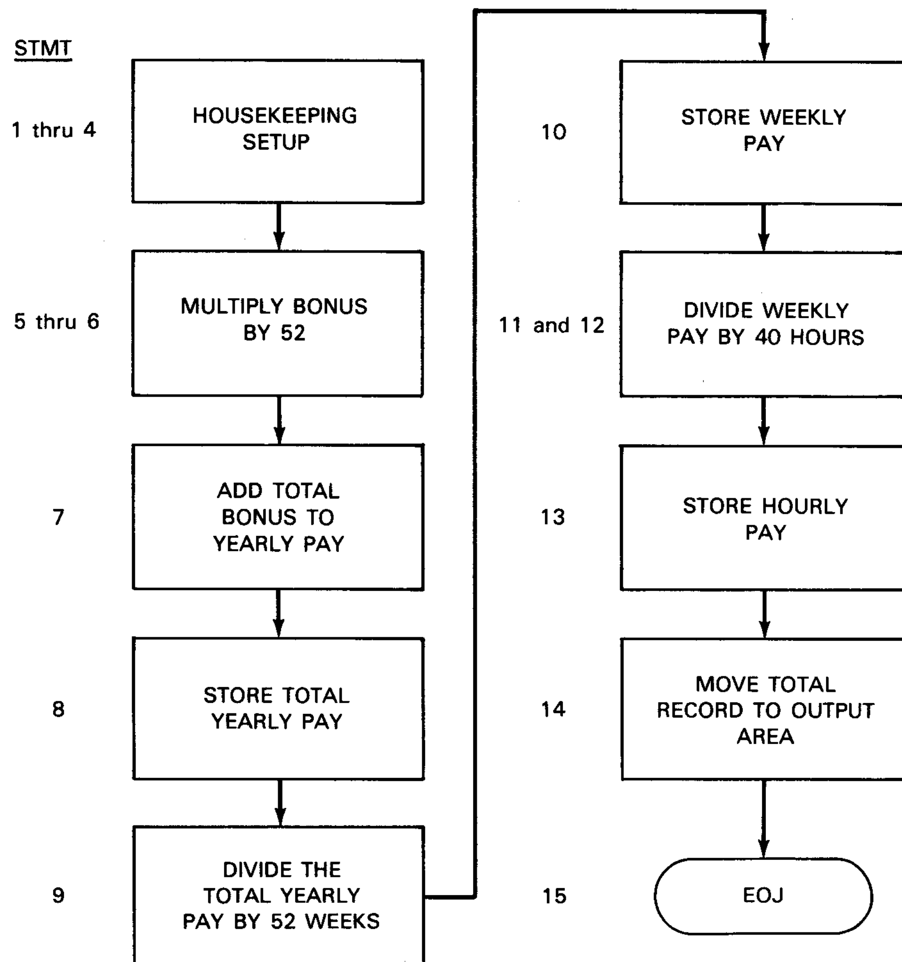
PART 8. APPENDIXES

Appendix A. Sample Program

The following list illustrates the steps taken to create, code, and execute a simple assembler program. The sample problem is designed to calculate the effect of a $5.00 a week bonus on an employee's yearly, weekly, and hourly pay.

- Flowchart

  The flowchart provides a graphic representation of the logic steps used to solve the problem.



- Source Code

Source code is created to implement the logic flow set forth in the flowchart. In the following list, comments have been included to explain what each source statement does.

STMT	SYMBOL	OPERATION CODE	STATEMENT OPERANDS	COMMENTS
1		TITLE	'FIRST SAMPLE PROGRAM'	Provides the assembler listing with a heading.
2	PROGRAM1	START	0	Provides a starting point for the program.
3	BEGIN	BALR	R6,0	Assigns a base address to the register.
4		USING	*,R6	Assigns a base register to the program.
5		ZAP	WORKAREA,BONUS	Enter bonus rate into work area.
6		MP	WORKAREA,WEEKS	Multiply bonus rate by 52 weeks.
7		AP	WORKAREA,YEARRATE	Add yearly rate to total bonus.
8		MVC	YEARPAY,WORKAREA+2	Move calculated total to yearly pay area.
9		DP	WORKAREA,WEEKS	Divide total pay by 52 weeks.
10		MVC	WEEKPAY,WORKAREA+1	Move results to weekly pay area.
11		ZAP	WORKAREA,WEEKPAY	Move weekly pay into work area.
12		DP	WORKAREA,HOURS	Divide by 40 hours week.
13		MVC	HOURPAY,WORKAREA+2	Move results to hourly pay area.
14		MVC	OUTPUT(23),EMPLOYEE	Completed record moved to output area.
15		EOJ		End of job.
16	WORKAREA	DS	CL6	Reserve 6 bytes of storage, contents unknown.
17	BONUS	DC	PL2'500'	Place value 5.00 in two bytes of storage.
18	HOURS	DC	PL2'40'	Place value 40 in two bytes of storage.
19	WEEKS	DC	PL2'52'	Place value 52 in two bytes of storage.
20	YEARRATE	DC	PL4'1300000'	Place value 13000.00 in four bytes of storage.
21	OUTPUT	DC	23C' '	Produces 23 bytes of blanks only.
22	EMPLOYEE	DS	OCL23	Symbol "EMPLOYEE" represents next 23 bytes.
23	NAME	DC	CL9'REBEW R D'	Produces nine bytes containing "REBEWRD".
24	WORKNO	DC	C'N4543'	Produces five bytes containing "N4543".
25	YEARPAY	DC	PL4'0'	Produces four bytes of zeros.
26	WEEKPAY	DC	PL3'0'	Produces two bytes of zeros.
27	HOURPAY	DC	PL2'0'	Produces two bytes of zeros.
28	R6	EQU	6	This instruction equates register 6 with R6.
29		END	BEGIN	THIS IS THE END OF THE PROGRAM.

- Job Control Procedure to Assemble

The following job control statements assemble the source code into an object code module.

Job Control Statements	Comments
// JOB ASSEMBLE	Assigns a unique name to the program.
// ASM	Assembles the source code.
/$	Signifies the start of the source statements.
. Assembler	
. source	
. code	
/*	Signifies the end of the source statements.
/&	Signifies the end of the job.
// FIN	Terminates card reader operation.

- **Output Generated by Assembly**

  The program is now assembled and an assembly listing is generated. The assembly listing contains warning messages if any errors are detected in the source code.

```
UNIVAC SYSTEM  OS/3 ASSEMBLER                                                                                   VER800922
DATE- 81/12/30 TIME- 23.09

ASSEMBLER CONTROL STATEMENTS ENCOUNTERED AND PROCESSED AS FOLLOWS-

          /$

                                                      EXTERNAL SYMBOL DICTIONARY                                PAGE    1
SYMBOL.  TYPE.  ESID.  ADDRESS.  LENGTH.

PROGRAM1 CSECT  01     000000    00007E

      FIRST SAMPLE PROGRAM                                                                                      PAGE    2

 LOC.  OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT                                                  OS/3 ASM 81/12/30

000000                                 2 PROGRAM1 START 0
000000 0560                            3 BEGIN    BALR  R6,0
000002                                 4          USING *,R6
000002 F851 603E 6044 00040 00046      5          ZAP   WORKAREA,BONUS
000008 FC51 603E 6048 00040 0004A      6          MP    WORKAREA,WEEKS
00000E FA53 603E 604A 00040 0004C      7          AP    WORKAREA,YEARRATE
000014 D203 6073 6040 00075 00042      8          MVC   YEAPPAY,WORKAREA+2
00001A FD51 603E 6048 00040 0004A      9          DP    WORKAREA,WEEKS
000020 D202 6077 603F 00079 00041     10          MVC   WEEKPAY,WORKAREA+1
000026 F852 603E 6077 00040 00079     11          ZAP   WORKAREA,WEEKPAY
00002C FD51 603E 6046 00040 00048     12          DP    WORKAREA,HOURS
000032 D201 607A 6040 0007C 00042     13          MVC   HOURPAY,WORKAREA+2
000038 D216 604E 6065 00050 00067     14          MVC   OUTPUT(23),EMPLOYEE
                                      15          EOJ
00003E                             A  16+         DS    0H                                                        EOJ00050
00003E 0A1A                        A  17+         SVC   26                                                        EOJ00070
000040                                18 WOPKAREA DS    CL6
000046 500C                           19 BONUS    DC    PL2'500'
000048 040C                           20 HOURS    DC    PL2'40'
00004A 052C                           21 WEEKS    DC    PL2'52'
00004C 1300000C                       22 YEARRATE DC    PL4'1300000'
000050 4040404040404040               23 OUTPUT   DC    23C' '
000067                                24 EMPLOYEE DS    0CL23
000067 D9C5C2C5E6400940               25 NAME     DC    CL9'REBEW R D'
000070 D5F4F5F4F3                     26 WORKNO   DC    C'N4543'
000075 0000000C                       27 YEARPAY  DC    PL4'0'
000079 00000C                         28 WEEKPAY  DC    PL3'0'
00007C 000C                           29 HOURPAY  DC    PL2'0'
000006                                30 R6       EQU   6
000000                                31          END   BEGIN

                                                     CROSS-REFERENCE                                            PAGE    1

SYMBOL   LENGTH VALUE  DEFN                                               SYMBOL   LENGTH VALUE   DEFN

BEGIN    00002 000000 00003     0031
BONUS    00002 000046 00019     0005
EMPLOYEE 00023 000067 00024     0014
HOURPAY  00002 00007C 00029     0013
HOURS    00002 000048 00020     0012
NAME     00009 000067 00025
OUTPUT   00001 000050 00023     0014
PROGRAM1 00001 000000 00002
R6       00001 000006 00030     0003  0004
WEEKPAY  00003 000079 00028     0010  0011
WEEKS    00002 00004A 00021     0006  0009
WORKAREA 00006 000040 00018     0005  0006  0007  0008  0009
                                0010  0011  0012  0013
WORKNO   00005 000070 00026
YEARPAY  00004 000075 00027     0008
YEARRATE 00004 00004C 00022     0007

                                                         DIAGNOSTICS                                            PAGE    1
LINE  ERROR CODE   MESSAGE

NO STATEMENTS FLAGGED IN THIS ASSEMBLY        -81/12/30  23.09-
```

- Job Control Procedure to Assemble, Link-edit, and Execute

After the errors in the source code are corrected, the following job control statements are added to assemble the code, create a load module, and execute the program.

Job Control Statements	Comments
// JOB ASSEMBLE	Assigns a unique name to the program.
// ASMLG	Assembles, link-edits, and executes the program.
/$	Signifies the start of the source statements.
. Assembler	
. source	
. code	
/*	Signifies the end of the source statements.
/&	Signifies the end of the job.
// FIN	Terminates card reader operation.

Appendix B. Character Conversion Codes

Table B—1. ASCII (American Standard Code for Information Interchange) Character Codes

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	! ①	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	∧ ①	n	~
F	SI	US	/	?	O	_	o	DEL

Columns 2–5: ② ; Columns 6–7: ③ ; Columns 2–7: ④

NOTES:

Some graphic card code and hexadecimal assignments may differ depending on the device, language, application, and installation policy.

① The following optional graphics can be substituted in the character set:

¬ for ∧

| for !

② Sixty-three printable character set.

③ Graphics available by use of the 0768—02 printer which prints a 94-character set (DEL is not a graphic)

④ Ninety-four printable character set.

Table B—2. EBCDIC (Extended Binary Coded Decimal Interchange Code) Character Codes

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS①		SP	&	–						{④	}④	\④	0
1	SOH	DC1	SOS①				/		a④	j	~④		A	J		1
2	STX	DC2	FS①	SYN					b	k	s		B	K	S	2
3	ETX	DC3							c	l	t		C	L	T	3
4		⑥							d	m	u		D	M	U	4
5	HT		LF						e	n	v		E	N	V	5
6		BS	ETB						f	o	w		F	O	W	6
7	DEL		ESC	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM						`④	i	r	z		I	R	Z	9
A					[	]⑤	¦③	:								
B	VT				.	$	,	#								
C	FF	FS⑤		DC4	<	*	%	@								
D	CR	GS⑤	ENQ	NAK	(	)	_	'								
E	SO⑤	RS⑤	ACK		+	;	>	=								
F	SI⑤	US⑤	BEL⑤	SUB	!②⑤	¬②	?	"								

NOTES:

Some graphic card code and hexadecimal assignments may differ depending on the device, language, application, and installation policy.

① DS, 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 Institute X3.4 – 1968, they must not appear in external storage media, such as American National Standard Institute 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 0768–02 printer, which prints a 94-character set.

⑤ The following substitutions are made for the UTS 400 handler:

SPROT	for SO
EPROT	for SI
SB	for FS
EB	for GS
SOE	for RS
FCC	for US
MW	for BEL
]	for !
!	for]

⑥ DC4 for the UTS 400 handler.

Table B—3. Punched Card, ASCII, and EBCDIC Codes (Part 1 of 5)

Character	Printed Symbol	Card Punches	ASCII		EBCDIC	
			Hexadecimal	Decimal	Hexadecimal	Decimal
Letters						
A	A	12–1	41	65	C1	193
B	B	12–2	42	66	C2	194
C	C	12–3	43	67	C3	195
D	D	12–4	44	68	C4	196
E	E	12–5	45	69	C5	197
F	F	12–6	46	70	C6	198
G	G	12–7	47	71	C7	199
H	H	12–8	48	72	C8	200
I	I	12–9	49	73	C9	201
J	J	11–1	4A	74	D1	209
K	K	11–2	4B	75	D2	210
L	L	11–3	4C	76	D3	211
M	M	11–4	4D	77	D4	212
N	N	11–5	4E	78	D5	213
O	O	11–6	4F	79	D6	214
P	P	11–7	50	80	D7	215
Q	Q	11–8	51	81	D8	216
R	R	11–9	52	82	D9	217
S	S	0–2	53	83	E2	226
T	T	0–3	54	84	E3	227
U	U	0–4	55	85	E4	228
V	V	0–5	56	86	E5	229
W	W	0–6	57	87	E6	230
X	X	0–7	58	88	E7	231
Y	Y	0–8	59	89	E8	232
Z	Z	0–9	5A	90	E9	233
a	a	12–0–1	61	97	81	129
b	b	12–0–2	62	98	82	130
c	c	12–0–3	63	99	83	131

Table B—3. Punched Card, ASCII, and EBCDIC Codes (Part 2 of 5)

Character	Printed Symbol	Card Punches	ASCII		EBCDIC	
			Hexadecimal	Decimal	Hexadecimal	Decimal
d	d	12–0–4	64	100	84	132
e	e	12–0–5	65	101	85	133
f	f	12–0–6	66	102	86	134
g	g	12–0–7	67	103	87	135
h	h	12–0–8	68	104	88	136
i	i	12–0–9	69	105	89	137
j	j	12–11–1	6A	106	91	145
k	k	12–11–2	6B	107	92	146
l	l	12–11–3	6C	108	93	147
m	m	12–11–4	6D	109	94	148
n	n	12–11–5	6E	110	95	149
o	o	12–11–6	6F	111	96	150
p	p	12–11–7	70	112	97	151
q	q	12–11–8	71	113	98	152
r	r	12–11–9	72	114	99	153
s	s	11–0–2	73	115	A2	162
t	t	11–0–3	74	116	A3	163
u	u	11–0–4	75	117	A4	164
v	v	11–0–5	76	118	A5	165
w	w	11–0–6	77	119	A6	166
x	x	11–0–7	78	120	A7	167
y	y	11–0–8	79	121	A8	168
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 B—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 B—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
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 B—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 C. Math Tables

C.1. HEXADECIMAL-DECIMAL INTEGER CONVERSION

Table C—1 provides for conversion of hexadecimal and decimal numbers in the range:

Hexadecimal	Decimal
000 to FFF	0000 to 4095

In the table, the decimal value appears at the intersection of the row representing the most significant hexadecimal digits (16^2 and 16^1) and the column representing the least significant hexadecimal digit (16^0).

Example:

hexadecimal C21 = decimal 3105

HEX	0	1	2
C0	3072	3073	3074
C1	3088	3089	3090
C2	3104	3105	3106
C3	3120	3121	3122

For numbers outside the range of the table, add the following values to the table figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
1000	4,096	C000	49,152
2000	8,192	D000	53,248
3000	12,288	E000	57,344
4000	16,384	F000	61,440
5000	20,480	10000	65,536
6000	24,576	20000	131,072
7000	28,672	30000	196,608
8000	32,768	40000	262,144
9000	36,864	50000	327,680
A000	40,960	60000	393,216
B000	45,056	70000	458,752

Example:

$BC21_{16} = 48{,}161_{10}$

Hexadecimal	Decimal
C21	3,105
+B000	+45,056
+BC21	48,161

Table C—1. Hexadecimal-Decimal Integer Conversion (Part 1 of 4)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
30	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

Table C—1. Hexadecimal-Decimal Integer Conversion (Part 2 of 4)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
50	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
60	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
70	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

Table C—1. Hexadecimal-Decimal Integer Conversion (Part 3 of 4)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
90	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A0	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B0	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

Table C—1. Hexadecimal-Decimal Integer Conversion (Part 4 of 4)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D0	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E0	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
F0	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

C.2. HEXADECIMAL FRACTIONS (APPROXIMATE VALUES)

Hexadecimal fractions are shown in Table C—2.

Table C—2. Hexadecimal Fractions

First Digit		Second Digit			Third Digit				Fourth Digit			
Hex.	Decimal	Hex.	Decimal		Hex.	Decimal			Hex.	Decimal		
.0	.0000	.00	.0000	0000	.000	.0000	0000	0000	.0000	.0000	0000	0000
.1	.0625	.01	.0039	0625	.001	.0002	4414	0625	.0001	.0000	1525	8789
.2	.1250	.02	.0078	1250	.002	.0004	8828	1250	.0002	.0000	3051	7578
.3	.1875	.03	.0117	1875	.003	.0007	3242	1875	.0003	.0000	4577	6367
.4	.2500	.04	.0156	2500	.004	.0009	7656	2500	.0004	.0000	6103	5156
.5	.3125	.05	.0195	3125	.005	.0012	2070	3125	.0005	.0000	7629	3945
.6	.3750	.06	.0234	3750	.006	.0014	6486	3750	.0006	.0000	9155	2734
.7	.4375	.07	.0273	4375	.007	.0017	0898	4375	.0007	.0001	0681	1523
.8	.5000	.08	.0312	5000	.008	.0019	5312	5000	.0008	.0001	2207	0313
.9	.5625	.09	.0351	5625	.009	.0021	9726	5625	.0009	.0001	3732	9102
.A	.6250	.0A	.0390	6250	.00A	.0024	4140	6250	.000A	.0001	5258	7891
.B	.6875	.0B	.0429	6875	.00B	.0026	8554	6875	.000B	.0001	6784	6680
.C	.7500	.0C	.0468	7500	.00C	.0029	2968	7500	.000C	.0001	8310	5469
.D	.8125	.0D	.0507	8125	.00D	.0031	7382	8125	.000D	.0001	9836	4258
.E	.8750	.0E	.0546	8750	.00E	.0034	1796	8750	.000E	.0002	1362	3047
.F	.9375	.0F	.0585	9375	.00F	.0036	6210	9375	.000F	.0002	2888	1836

To convert a 4-digit (2-byte) hexadecimal fraction to a decimal fraction, add the values shown in Table C—2 for each of the hexadecimal digits to be converted as the following illustrates. The hexadecimal fraction .B5A1 equals the approximate decimal fraction .70948791 from Table C—2.

.B	from the table equals	.6875
.05	from the table equals	.01953125
.00A	from the table equals	.002441406250
.0001	from the table equals	.000015258789
.B5A1	equals the sum	.709487915039

C.3. POWERS OF 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5

C.4. POWERS OF 16

16^n							n
						1	0
						16	1
						256	2
					4	096	3
					65	536	4
				1	048	576	5
				16	777	216	6
				268	435	456	7
			4	294	967	296	8
			68	719	476	736	9
		1	099	511	627	776	10
		17	592	186	044	416	11
		281	474	976	710	656	12
	4	503	599	627	370	496	13
	72	057	594	037	927	936	14
1	152	921	504	606	846	976	15

These powers of 16 are especially useful in determining the value of floating-point numbers.

Appendix D. Check-Off Table Terms

General			
OPCODE		FORMAT TYPE	OBJECT INST. LGTH. (BYTES)
MNEM.	HEX.		

Condition Codes
☐ IF RESULT = 0, SET TO 0
☐ IF RESULT < 0, SET TO 1
☐ IF RESULT > 0, SET TO 2
☐ IF OVERFLOW, SET TO 3
☐ UNCHANGED

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

The check-off table is a fast reference source for its associated instruction. The table points where to look for possible errors when writing your programs and helps you debug your program when it does not run correctly. A program interrupt occurs when the hardware detects an improper specification, use of instructions or data. Interrupt requests of this type cause the instruction currently being executed to be suppressed or terminated. When a table is used with an instruction, the checked (■) condition codes and program exceptions are the only ones that apply to that instruction. The program exceptions are explained in the following list, as well as the instruction to which it applies.

- Addressing

  A storage location outside the range of the installed storage is referenced by a program-specified address.

- Data

  — An invalid sign or digit code is detected in decimal operands.

  — Fields in decimal arithmetic overlap incorrectly.

  — The first operand of the *multiply decimal* instruction does not have a sufficient number of high order zero digits.

- Decimal Divide

  The quotient of a *divide decimal* instruction exceeds the capacity of the quotient part of the first operand field.

- Decimal Overflow

  The result of an *add decimal, subtract decimal,* or *zero and add* instruction exceeds the capacity of the first operand location.

- Execute

  The subject instruction of an *execute* instruction is an *execute* instruction.

- Exponent Overflow

  The final characteristic resulting from a floating-point arithmetic operation exceeds 127.

- Exponent Underflow

  The final characteristic resulting from a floating-point arithmetic operation is less than zero.

- Fixed-Point Divide

  The quotient of a fixed-point divide operation exceeds the capacity of the first operand (including division by zero) or the result of a *convert to binary* instruction exceeds 31 bits.

- Fixed-Point Overflow

  A fixed-point add or subtract operation exceeds the capacity of the first operand field.

- Floating-Point Divide

  The divisor fraction in a floating-point divide operation is equal to zero.

- Operation

  An illegal operation has been attempted or an operation using a noninstalled processor feature has been attempted.

- Protection

  A storage protection violation occurs on a program-generated address when the protection feature is installed.

- Significance

  The final fraction resulting from a floating-point addition or subtraction is equal to zero.

- Specification

  — The unit of information referenced is not on an appropriate boundary.

  — An invalid modifier field is specified in the STR instruction.

  — The R_1 field of an instruction which uses an even/odd pair of registers (64-bit operand) does not specify an even register.

  — A floating-point register other than 0, 2, 4, or 6 is specified.

  — A multiplicand or divisor in decimal arithmetic exceeds 15 digits and sign.

  — The first operand field is shorter than, or equal in length to, the second operand in *decimal multiply* and decimal divide instructions.

Appendix E. Instruction Listings

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

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

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

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

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
CLC	Compare Logical	D5	6	$d_1,(l,b_1),d_2(b_2)$	$s_1(l),s_2$
CLCL	Compare Logical Characters Long	0F	2	r_1,r_2	r_1,r_2
CLI	Compare Logical Immediate	95	4	$d_1(b_1),i_2$	s_1,i_2
CLIS	Compare Logical Immediate and Skip	E1	6	$d_1(b_1),i_2,m_3,d_4$	s_1,i_2,m_3,s_4
CLM	Compare Logical Characters Under Mask	BD	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2
CLR	Compare Logical	15	2	r_1,r_2	r_1,r_2
CLRCH	Clear Channel	9F02	4	(Privileged)	(Privileged)
CLRDV	Clear Device	9DX2	4	(Privileged)	(Privileged)
CP	Compare Decimal	F9	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
CR	Compare Algebraic	19	2	r_1,r_2	r_1,r_2
CSM	Compare and Swap Under Mask	B9	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
CVB	Convert to Binary	4F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
CVD	Convert to Decimal	4E	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
D	Divide	5D	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
DD	Divide, Long	6D	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
DDR	Divide, Long	2D	2	r_1,r_2	r_1,r_2
DE	Divide, Short	7D	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
DER	Divide, Short	3D	2	r_1,r_2	r_1,r_2
DP	Divide Decimal	FD	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),,s_2(l_2)$
DR	Divide	1D	2	r_1,r_2	r_1,r_2
ED	Edit	DE	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
EDMK	Edit and Mark	DF	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
EIO	Enqueue I/O	E0	6	(Privileged)	(Privileged)
EX	Execute	44	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
EXD	Execute Diagnose	8300	4	(Privileged)	(Privileged)
GRB	Get IORB	0B	2	(Privileged)	(Privileged)
HDR	Halve, Long	24	2	r_1,r_2	r_1,r_2
HDV	Halt Device	9E01	4	(Privileged)	(Privileged)
HER	Halve, Short	34	2	r_1,r_2	r_1,r_2
HPR	Halt and Proceed	99	4	(Privileged)	(Privileged)
IC	Insert Character	43	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
ICM	Insert Characters Under Mask	BF	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2
IPL	Initial Program Load	8303	4	(Privileged)	(Privileged)
ISK	Insert Storage Key	09	2	(Privileged)	(Privileged)
L	Load	58	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LA	Load Address	41	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LCDR	Load Complement, Long	23	2	r_1,r_2	r_1,r_2
LCER	Load Complement, Short	33	2	r_1,r_2	r_1,r_2
LCHR	Load Channel Register	9F03	4	(Privileged)	(Privileged)
LCR	Load Complement	13	2	r_1,r_2	r_1,r_2
LCTL	Load Control	B7	4	(Privileged)	(Privileged)
LD	Load, Long	68	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LDA	Load Directive Address	51	4	(Privileged)	(Privileged)
LDR	Load, Long	28	2	r_1,r_2	r_1,r_2
LE	Load, Short	78	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LER	Load, Short	38	2	r_1,r_2	r_1,r_2
LH	Load Half Word	48	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
LIA	Load I/O Address	61	4	(Privileged)	(Privileged)

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

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
LM	Load Multiple	98	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
LNDR	Load Negative, Long	21	2	r_1,r_2	r_1,r_2
LNER	Load Negative, Short	31	2	r_1,r_2	r_1,r_2
LNR	Load Negative	11	2	r_1,r_2	r_1,r_2
LPDR	Load Positive, Long	20	2	r_1,r_2	r_1,r_2
LPER	Load Positive, Short	30	2	r_1,r_2	r_1,r_2
LPR	Load Positive	10	2	r_1,r_2	r_1,r_2
LPSW	Load Program Status Word	82	4	(Privileged)	(Privileged)
LR	Load	18	2	r_1,r_2	r_1,r_2
LRC	Longitudinal Redundancy Check	830E	4	(Privileged)	(Privileged)
LRR	Load Relocation Register	A3	4	(Privileged)	(Privileged)
LTDR	Load and Test, Long	22	2	r_1,r_2	r_1,r_2
LTER	Load and Test, Short	32	2	r_1,r_2	r_1,r_2
LTR	Load and Test	12	2	r_1,r_2	r_1,r_2
M	Multiply	5C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MD	Multiply, Long	6C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MDR	Multiply, Long	2C	2	r_1,r_2	r_1,r_2
ME	Multiply, Short	7C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MER	Multiply, Short	3C	2	r_1,r_2	r_1,r_2
MH	Multiply Half Word	4C	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
MIO	Move I/O	81	4	(Privileged)	(Privileged)
MP	Multiple Decimal	FC	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
MR	Multiply	1C	2	r_1,r_2	r_1,r_2
MVC	Move Characters	D2	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
MVCL	Move Character Long	0E	2	r_1,r_2	r_1,r_2
MVI	Move Immediate	92	4	$d_1(b_1),i_2$	s_1,i_2
MVN	Move Numerics	D1	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
MVO	Move With Offset	F1	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
MVZ	Move Zones	D3	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
N	AND Logical	54	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
NC	AND Logical	D4	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
NI	AND Logical Immediate	94	4	$d_1(b_1),i_2$	s_1,i_2
NR	AND Logical	14	2	r_1,r_2	r_1,r_2
O	OR Logical	56	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
OC	OR Logical	D6	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
OI	OR Logical Immediate	96	4	$d_1(b_1),i_2$	s_1,i_2
OR	OR Logical	16	2	r_1,r_2	r_1,r_2
PACK	Pack	F2	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
PRB	Put IORB	0C	2	(Privileged)	(Privileged)
RESET	Reset	8301	4	(Privileged)	(Privileged)
S	Subtract	5B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SD	Subtract Normalized, Long	6B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SDR	Subtract Normalized, Long	2B	2	r_1,r_2	r_1,r_2
SDV	Start Device	9C02	4	(Privileged)	(Privileged)
SE	Subtract Normalized, Short	7B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SER	Subtract Normalized, Short	3B	2	r_1,r_2	r_1,r_2
SH	Subtract Half Word	4B	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SHL	Shift Logical	9B	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2

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

Mnemonic	Instruction Name	Machine Code	Byte Length	Source Code Format	
				Explicit	Implicit
SL	Subtract Logical	5F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SLA	Shift Left Single Algebraic	8B	4	$r_1,d_2(b_2)$	r_1,s_2
SLDA	Shift Left Double Algebraic	8F	4	$r_1,d_2(b_2)$	r_1,s_2
SLDL	Shift Left Double Logical	8D	4	$r_1,d_2(b_2)$	r_1,s_2
SLL	Shift Left Single Logical	89	4	$r_1,d_2(b_2)$	r_1,s_2
SLM	Supervisor Load Multiple	B8	4	(Privileged)	(Privileged)
SLR	Subtract Logical	1F	2	r_1,r_2	r_1,r_2
SP	Subtract Decimal	FB	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
SPM	Set Program Mask	04	2	r_1	r_1
SR	Subtract	1B	2	r_1,r_2	r_1,r_2
SRA	Shift Right Single Algebraic	8A	4	$r_1,d_2(b_2)$	r_1,s_2
SRDA	Shift Right Double Algebraic	8E	4	$r_1,d_2(b_2)$	r_1,s_2
SRDL	Shift Right Double Logical	8C	4	$r_1,d_2(b_2)$	r_1,s_2
SRL	Shift Right Single Logical	88	4	$r_1,d_2(b_2)$	r_1,s_2
SRP	Shift and Round Decimal	F0	6	$d_1(l_1,b_1),d_2(b_2),i_3$	$s_1(l_1),s_2,i_3$
SSK	Set System Key	08	2	(Privileged)	(Privileged)
SSM	Set System Mask	80	4	(Privileged)	(Privileged)
SSTM	Supervisor Store Multiple	B0	4	(Privileged)	(Privileged)
ST	Store	50	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STC	Store Character	42	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STCM	Store Characters Under Mask	BE	4	$r_1,m_3,d_2(b_2)$	r_1,m_3,s_2
STCTL	Store Control	B6	4	(Privileged)	(Privileged)
STD	Store Long	60	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STE	Store Short	70	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STH	Store Half Word	40	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
STM	Store Multiple	90	4	$r_1,r_3,d_2(b_2)$	r_1,r_3,s_2
STR	Service Timer Register	03	2	(Privileged)	(Privileged)
STRR	Store Relocation Register	A2	4	(Privileged)	(Privileged)
STS	Store Status	8302	4	(Privileged)	(Privileged)
SU	Subtract Unnormalized, Short	7F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SUR	Subtract Unnormalized, Short	3F	2	r_1,r_2	r_1,r_2
SVC	Supervisor Call	0A	2	i	i
SW	Subtract Unnormalized, Long	6F	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
SWLS	Switch List Scan	830F	4	(Privileged)	(Privileged)
SWR	Subtract Unnormalized, Long	2F	2	r_1,r_2	r_1,r_2
TM	Test Under Mask	91	4	$d_1(b_1),i_2$	s_1,i_2
TMS	Test Under Mask and Skip	E2	6	$d_1(b_1),i_2,m_3,d_4$	s_1,i_2,m_3,s_4
TR	Translate	DC	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
TRT	Translate and Test	DD	6	$d_1(l,b_1),d_2(b_2)$	
TS	Test and Set	93	4	$d_2(b_2)$	s_2
UNPK	Unpack	F3	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$
X	Exclusive OR	57	4	$r_1,d_2(x_2,b_2)$	$r_1,s_2(x_2)$
XC	Exclusive OR	D7	6	$d_1(l,b_1),d_2(b_2)$	$s_1(l),s_2$
XI	Exclusive OR, Immediate	97	4	$d_1(b_1),i_2$	s_1,i_2
XR	Exclusive OR	17	2	r_1,r_2	r_1,r_2
ZAP	Zero and Add Decimal	F8	6	$d_1(l_1,b_1),d_2(l_2,b_2)$	$s_1(l_1),s_2(l_2)$

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

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

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

Instruction Name	Machine Code	Mnemonic
Clear Device — Privileged	9DX2	CLRDV
Compare	19	CR
Compare	59	C
Compare and Swap Under Mask	B9	CSM
Compare Decimal	F9	CP
Compare Half Word	49	CH
Compare Logical	15	CLR
Compare Logical	55	CL
Compare Logical	95	CLI
Compare Logical	D5	CLC
Compare Logical Characters Under Mask	BD	CLM
Compare Logical Immediate and Skip	E1	CLIS
Compare Logical Characters Long	OF	CLCL
Compare, Long	29	CDR
Compare, Long	69	CD
Compare, Short	39	CER
Compare, Short	79	CE
Convert to Binary	4F	CVB
Convert to Decimal	4E	CVD
Divide	1D	DR
Divide	5D	D
Divide Decimal	FD	DP
Divide, Long	2D	DDR
Divide, Long	6D	DD
Divide, Short	3D	DER
Divide, Short	7D	DE
Edit	DE	ED
Edit and Mark	DF	EDMK
Enqueue I/O — Privileged	EO	EIO

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

Instruction Name	Machine Code	Mnemonic
Exclusive OR	17	XR
Exclusive OR	57	X
Exclusive OR	97	XI
Exclusive OR	D7	XC
Execute	44	EX
Execute Diagnose — Privileged	8300	EXD
Get IORB — Privileged	0B	GRB
Halt and Proceed — Privileged	99	HPR
Halt Device — Privileged	9E01	HDV
Halve, Long	24	HDR
Halve, Short	34	HER
Initial Program Load — Privileged	8303	IPL
Insert Character	43	IC
Insert Characters Under Mask	BF	ICM
Insert Storage Key — Privileged	09	ISK*
Load	18	LR
Load	58	L
Load Address	41	LA
Load and Test	12	LTR
Load and Test, Long	22	LTDR
Load and Test, Short	32	LTER
Load Channel Register — Privileged	9F03	LCHR
Load Complement	13	LCR
Load Complement, Long	23	LCDR
Load Complement, Short	33	LCER
Load Control — Privileged	B7	LCTL
Load Directive Address — Privileged	51	LDA
Load Half Word	48	LH
Load I/O Address — Privileged	61	LIA
Load, Long	28	LDR
Load, Long	68	LD

*Added as a feature

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

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

→ MSS

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

Instruction Name	Machine Code	Mnemonic
OR	96	OI
OR	D6	OC
Pack	F2	PACK
Put IORB — Privileged	0C	PRB
Reset — Privileged	8301	RESET
Service Timer Register — Privileged	03	STR
Set Program Mask	04	SPM
Set Storage Key — Privileged	08	SSK*
Set System Mask — Privileged	80	SSM
Shift and Round Decimal	F0	SRP
Shift Left Double	8F	SLDA
Shift Left Double Logical	8D	SLDL
Shift Left Single	8B	SLA
Shift Left Single Logical	89	SLL
Shift Logical	9B	SHL
Shift Right Double	8E	SRDA
Shift Right Double Logical	8C	SRDL
Shift Right Single	8A	SRA
Shift Right Single Logical	88	SRL
Start Device — Privileged	9C02	SDV
Store	50	ST
Store Character	42	STC
Store Characters Under Mask	BE	STCM
Store Control — Privileged	B6	STCTL
Store Half Word	40	STH
Store, Long	60	STD
Store Multiple	90	STM
Store Relocation Register — Privileged	A2	STRR
Store, Short	70	STE
Store Status — Privileged	8302	STS

*Added as a feature.

STEP

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

Instruction Name	Machine Code	Mnemonic
Subtract	1B	SR
Subtract	5B	S
Subtract Decimal	FB	SP
Subtract Half Word	4B	SH
Subtract Logical	1F	SLR
Subtract Logical	5F	SL
Subtract Normalized, Long	2B	SDR
Subtract Normalized, Long	6B	SD
Subtract Normalized, Short	3B	SER
Subtract Normalized, Short	7B	SE
Subtract Unnormalized, Long	2F	SWR
Subtract Unnormalized, Long	6F	SW
Subtract Unnormalized, Short	3F	SUR
Subtract Unnormalized, Short	7F	SU
Supervisor Call	0A	SVC
Supervisor Load Multiple — Privileged	B8	SLM
Supervisor Store Multiple — Privileged	B0	SSTM
Switch List Scan — Privileged	830F	SWLS
Test and Set	93	TS
Test Under Mask	91	TM
Test Under Mask and Skip	E2	TMS
Translate	DC	TR
Translate and Test	DD	TRT
Unpack	F3	UNPK
Zero and Add	F8	ZAP

Table E—3. List of Instructions by Machine Code (Part 1 of 6)

Machine Code	Mnemonic	Instruction Name
03	STR	Service Timer Register — Privileged
04	SPM	Set Program Mask
05	BALR	Branch and Link
06	BCTR	Branch on Count
07	BCR	Branch on Condition (Native and 360/20 modes)
08	SSK*	Set Storage Key — Privileged
09	ISK*	Insert Storage Key — Privileged
0A	SVC	Supervisor Call
0B	GRB	Get IORB — Privileged
0C	PRB	Put IORB — Privileged
0E	MVCL	Move Characters Long
0F	CLCL	Compare Logical Characters Long
10	LPR	Load Positive
11	LNR	Load Negative
12	LTR	Load and Test
13	LCR	Load Complement
14	NR	AND
15	CLR	Compare Logical
16	OR	OR
17	XR	Exclusive OR
18	LR	Load
19	CR	Compare
1A	AR	Add
1B	SR	Subtract
1C	MR	Multiply
1D	DR	Divide
1E	ALR	Add Logical
1F	SLR	Subtract Logical
20	LPDR	Load Positive, Long
21	LNDR	Load Negative, Long
22	LTDR	Load and Test, Long
23	LCDR	Load Complement, Long

Table E—3. List of Instructions by Machine Code (Part 2 of 6)

Machine Code	Mnemonic	Instruction Name
24	HDR	Halve, Long
28	LDR	Load, Long
29	CDR	Compare, Long
2A	ADR	Add Normalized, Long
2B	SDR	Subtract Normalized, Long
2C	MDR	Multiply, Long
2D	DDR	Divide, Long
2E	AWR	Add Unnormalized, Long
2F	SWR	Subtract Unnormalized, Long
30	LPER	Load Positive, Short
31	LNER	Load Negative, Short
32	LTER	Load And Test, Short
33	LCER	Load Complement, Short
34	HER	Halve, Short
38	LER	Load, Short
39	CER	Compare, Short
3A	AER	Add Normalized, Short
3B	SER	Subtract Normalized, Short
3C	MER	Multiply, Short
3D	DER	Divide, Short
3E	AUR	Add Unnormalized, Short
3F	SUR	Subtract Unnormalized, Short
40	STH	Store Half Word
41	LA	Load Address
42	STC	Store Character
43	IC	Insert Character
44	EX	Execute
45	BAL	Branch and Link
46	BCT	Branch on Count
47	BC	Branch on Condition

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

Machine Code	Mnemonic	Instruction Name
48	LH	Load Half Word
49	CH	Compare Half Word
4A	AH	Add Half Word
4B	SH	Subtract Half Word
4C	MH	Multiply Half Word
4E	CVD	Convert to Decimal
4F	CVB	Convert to Binary
50	ST	Store
51	LDA	Load Directive Address — Privileged
54	N	AND
55	CL	Compare Logical
56	O	OR
57	X	Exclusive OR
58	L	Load
59	C	Compare
5A	A	Add
5B	S	Subtract
5C	M	Multiply
5D	D	Divide
5E	AL	Add Logical
5F	SL	Subtract Logical
60	STD	Store, Long
61	LIA	Load I/O Address — Privileged
68	LD	Load, Long
69	CD	Compare, Long
6A	AD	Add Normalized, Long
6B	SD	Subtract Normalized, Long
6C	MD	Multiply, Long
6D	DD	Divide, Long
6E	AW	Add Unnormalized, Long

Table E—3. List of Instructions by Machine Code (Part 4 of 6)

Machine Code	Mnemonic	Instruction Name
6F	SW	Subtract Unnormalized, Long
70	STE	Store, Short
78	LE	Load, Short
79	CE	Compare, Short
7A	AE	Add Normalized, Short
7B	SE	Subtract Normalized, Short
7C	ME	Multiply, Short
7D	DE	Divide, Short
7E	AU	Add Unnormalized, Short
7F	SU	Subtract Unnormalized, Short
80	SSM	Set System Mask — Privileged
81	MIO	Move I/O — Privileged
82	LPSW	Load PSW — Privileged
8300	EXD	Execute Diagnose — Privileged
8301	RESET	Reset — Privileged
8302	STS	Store Status — Privileged
8303	IPL	Initial Program Load — Privileged
830E	LRC	Longitudinal Redundancy Check
830F	SWLS	Switch List Scan — Privileged
86	BXH	Branch on Index High
87	BXLE	Branch on Index Low or Equal
88	SRL	Shift Right Single Logical
89	SLL	Shift Left Single Logical
8A	SRA	Shift Right Single
8B	SLA	Shift Left Single
8C	SRDL	Shift Right Double Logical
8D	SLDL	Shift Left Double Logical
8E	SRDA	Shift Right Double
8F	SLDA	Shift Left Double
90	STM	Store Multiple
91	TM	Test Under Mask
92	MVI	Move Immediate

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

Machine Code	Mnemonic	Instruction Name
93	TS	Test and Set
94	NI	AND
95	CLI	Compare Logical
96	OI	OR
97	XI	Exclusive OR
98	LM	Load Multiple
99	HPR	Halt and Proceed — Privileged
9A	AI	Add Immediate
9B	SHL	Shift Logical
9C02	SDV	Start Device — Privileged
9DX2	CLRDV	Clear Device — Privileged
9E01	HDV	Halt Device — Privileged
9F02	CLRCH	Clear Channel — Privileged
9F03	LCHR	Load Channel Register — Privileged
A2	STRR	Store Relocation Register — Privileged
A3	LRR	Load Relocation Register — Privileged
B0	SSTM	Supervisor Store Multiple — Privileged
B6	STCTL	Store Control — Privileged
B7	LCTL	Load Control — Privileged
B8	SLM	Supervisor Load Multiple — Privileged
B9	CSM	Compare and Swap Under Mask
BD	CLM	Compare Logical Characters Under Mask
BE	STCM	Store Characters Under Mask
BF	ICM	Insert Characters Under Mask
D1	MVN	Move Numerics
D2	MVC	Move
D3	MVZ	Move Zones

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

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

*Added as a feature.

Appendix F. Use of PARAM Statement

This appendix describes the use of the PARAM statement and the option-specifying operands supported by the SPERRY UNIVAC Operating System/3 (OS/3) Assembler. These options permit you to identify library files, to access source or copy modules and macro definitions from these libraries, to select assembler listings, and to control object module output. Also included in this appendix is the source module correction routine.

F.1. PARAM STATEMENT

The PARAM statement specifies the assembler processing options in effect at assembly time and alters the standard default options. If you don't specify assembler options in the control stream of your job, the assembler functions as follows:

- The assembler searches only the system source library file (YSRC) for any source module or copy code referenced.
- It also searches only the system macro library file (YMAC) for any macro references.
- It stores the object module produced in the job run library file (YRUN).
- It prints the source code, object code, cross-references, and diagnostic listings.
- The value of &SYSPARM is equal to a null string.
- Columns 1 and 2 of the coding form must contain slashes, followed by at least one blank column, and then PARAM followed by at least one blank column (see following format). Multiple options are supported for each option separated by commas. The end of selected options is indicated by a blank column following the last option. All options selected are printed preceding the assembly listing.

Format:

1	10
// △ PARAM △	[COPY= {filename1 / (N) / **YSRC**} [/{filename2 / (N) / **YSRC**}]]
	[,IN=modulename [/{filename / **YSRC**}]]
	[,LIN= {filename1 / (N) / **YMAC**} [/{filename2 / (N) / **YMAC**}]]
	[,LST= {s / ([s1] [,s2] [,s3] [,s4])}]
	[,OUT= {filename / (N) / **YRUN**}]
	[,RO= {YES / **NO**}]
	[,SYSPARM= {'string' / **null-string**}]

COPY Keyword Parameter:

Enables up to two files to be identified as source code module libraries or specifies that no files are to be searched for source code modules. If this option is omitted, YSRC is assumed and is the only file searched for source code module references. Only source modules can be copied; the source code must be in the standard format and may not contain any COPY, ICTL, MACRO, PROC, or MEND directives.

COPY=filename1

Specifies that the file identified as filename1 is searched first for source code modules referenced and, if not found there, then YSRC is searched: filename is any name you specify or the system source library. If filename1 = filename2, then copy = filename1 will generate the same files to be searched as copy = /filename2, except that in the first case the order in which the files are to be searched will be filename1 and then YSRC, whereas in the second case, the order will be YSRC and then filename2.

COPY=filename1/filename2

Specifies that the file identified as filename1 is searched first. Then, the file identified as filename2 is searched for source code modules referenced. When two filenames are specified for this parameter, the YSRC file is not searched.

COPY=filename1/(N)
Specifies only the file identified as filename1 is searched for source code modules referenced. As stated previously, if filename1 = filename2, then copy = filename1/(N) is the same as copy = (N)/filename2, with only one file to be searched in either case.

COPY=(N)
Specifies no files, not even YSRC, are searched for source code modules referenced. COPY=(N)/(N) is the same as COPY=(N).

IN Keyword Parameter:

Identifies the name of the source module that is to be assembled and the file in which it resides. If this option is omitted, the source code must be in the control stream.

IN=modulename
Specifies the name of the source module and directs the assembler to search the YSRC file for the module; modulename is the name of the source module and is up to eight characters.

IN=modulename/filename
Specifies the name of the source module and the file in which it resides; filename is any name you supply or the system source library.

LIN Keyword Parameter:

Enables up to two files to be identified as macro source files or no files to be searched for macro references. If this option is omitted, YMAC is assumed and is the only file searched.

LIN=filename1
Identifies the file that is searched for macro references and, if not found there, then YMAC is searched; filename is any name or the name of the system macro library.

LIN=filename1/filename2
Identifies the two files that are searched for macro references. The file identified as filename1 is searched first, followed by the file identified as filename2. The YMAC file is not searched.

LIN=filename1/(N)
Specifies only the file identified as filename1 is searched for macro references.

LIN=(N)
Specifies no files, not even YMAC, are searched for macro references.

LST Keyword Parameter:

Indicates the type of listing desired. If this option is omitted, source, object, cross-reference, and diagnostic listings are printed.

LST=s
A single specification requiring no parentheses.

LST=([s_1]...[,s_4])
Any s in the series is one of the following:

NC
Specifies that the cross-reference listing is suppressed.

ND
Specifies that the diagnostic listing is suppressed.

NR
Specifies that the cross reference listing is to contain only those symbols that have at least one reference each.

N
Specifies that all output listings are suppressed.

DBG
Specifies a proc or macro debug mode feature within the OS/3 assembler. When the feature is selected, the output listing shows the following:

- Results of the expansion of any proc or macro called within the user program, including any conditional assembly directives processed as the result of the expansion itself. Source coding (constants, directives, and instructions) is listed twice and shows any appropriate substitutions. Any statements causing error diagnostics show the exit line in error.
- A proc or macro which produces error diagnostics at the time it is encoded is listed following the END directive; e.g., system errors. A proc or macro is encoded once, but may be called multiple times.
- If an error is detected at both expansion and encoding time, it appears two or more times. Errors detected only at encoding time appear once following the END directive.
- All lines flagged (regardless of their order of appearance) are shown in the diagnostic summary list. Lines flagged at encoding time may or may not be flagged at expansion time.

When this feature is not selected, any errors detected during proc or macro expansion may not show the exact line in error, but rather the vicinity of the item which is flagged.

OUT Keyword Parameter:

Enables you to specify the file that is to be used to store the object module output by the assembler. If this option is omitted, the object module is generated and stored in YRUN, the system-run library.

OUT=filename
Identifies the file that is used as the output file by the assembler; filename is any name or the job run library.

OUT=(N)
Specifies that no output file is used by the assembler and, thus, no object module is generated.

RO Keyword Parameter:

Permits you to optionally flag all absolute/base displacement fields of instructions that yield values less than 4096. Each statement is flagged with an 'ADDRESSABILITY' error flag. For example, if you wanted to code MVI TAG,X'40' but coded MVC TAG,X'40' by mistake, the latter instruction would be flagged, since the displacement field is less than 4096.

SYSPARM Keyword Parameter:

Specifies the equivalent of a global SETC symbol, with the value specified in this option. If this option is omitted, the value of &SYSPARM is a null string.

Operational Consideration:

The value established by SYSPARM is available within the assembly, both outside of and within macro definitions. This parameter is referenced as &SYSPARM within assembly statements. Any error in this specification directs the assembler to ignore the specification, and an appropriate error message is printed on the output printer.

SYSPARM='string'
Specifies a string of one to eight characters enclosed in apostrophes. An apostrophe within the string is represented by two apostrophes but only counts as one in determining the length of the string.

F.2. SOURCE CORRECTIONS

The OS/3 assembler supports a source module correction routine. This routine is the same as the one used in the librarian. The correction deck is interchangeable between the assembler and the librarian except that the librarian also uses the added COR control statement. The corrections made to the source module are temporary. The corrections are specified by the presence of both the source module input (//△PARAM△IN=modulename or the IN=(vol-ser-no,label) for the jproc call) and the correction records in the job control stream. These records must be with the data delimiters (/$ and /*). If there are no records between the data delimiters, no source correction is performed.

There are three control statements associated with the correction routine: sequence (SEQ), recycle (REC), and skip (SKI). To make the source module corrections, the actual source record to be inserted is used as the correction card with the same sequence number as the record to be replaced. Insertions are performed by using at least one correction card (always the first) card with a sequence number falling between the sequence numbers of the records between which the insertion is to be made. Any number of unsequenced correction cards may then follow the first sequence card. Deletions are performed by bypassing one or more original source module records in the old data set, thus eliminating them from being written on the new data set. The SKI and REC statements are used for this function.

F.2.1. SEQ Statement

Function:

Specifies the starting position and the length of the sequence field. If the sequence file is omitted, column 73 is assumed to be the first column of the sequence field and continues to the maximum of eight characters.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
unused	SEQ	,,, { column position / 73 } , { content / 00000000 }

Specifications:

column position
Is the first column position in the source record where the sequence field begins. If omitted, column 73 is assumed to be the first column of the sequence field.

content
Is one to eight characters. The length determines the length of the sequence field.

Operational Considerations:

- Card column 1 must be blank if the sequence field does not start in card column 1.
- The SEQ card is always the first card in the correction routine.

F.2.2. REC Statement

Function:

Causes the record pointer for the input module to be repositioned to the first record in the module. In conjunction with the SKI control statement, it allows the rearranging of major segments of the input module. When a REC control statement is processed, records are read from the input module up to and including the record whose sequence number matches the sequence number in the REC control statement field. Then, the second pointer for the input module is reset to the first record in the module. If the sequence field of the REC control statement is blank, repositioning of the record pointer takes place immediately.

Format:

LABEL	ΔOPERATIONΔ	OPERAND	SEQUENCE
ignored	REC	unused	[last sequence no.]

Specification:

last sequence no.
Is one to eight alphanumeric characters that identify the sequence number of the last input record to be read from the input module. If omitted, the repositioning function takes place immediately.

Operational Considerations:

- Records are replaced one at a time by writing a source statement with a sequence number matching the sequence number of the record to be replaced.

- Records are inserted by writing source correction statements with sequence numbers that fall between the sequence numbers of the input records between which insertion is to take place. Blank sequence fields cause an insertion to take place immediately.

F.2.3. SKI Statement

Function:

Allows one or more original input module records to be bypassed. Records are read from the input module until a sequence number is detected that matches the sequence number of the SKI command. The skip operation is started and continues until a sequence number that matches the operand field of the SKI command is detected. If the sequence field of the skip command is blank, the function is started immediately.

Format:

LABEL	△ OPERATION △	OPERAND	SEQUENCE
unused	SKI	last-sequence-no.	[starting-sequence-no.]

Specifications:

last-sequence-no
Is one to eight alphanumeric characters that identify the sequence number of the last input module record to be bypassed.

starting-sequence-no.
Is one to eight alphanumeric characters that identify the sequence number of the first source module record to be bypassed.

Operational Consideration:

- If omitted, the skip operation starts immediately with the input module record that follows the last record operated on.

Appendix G. System Variable Symbols

System variable symbols automatically generate values or character strings at assembly time. There are seven system variable symbols: &SYSECT, &SYSLIST, &SYSNDX, &SYSDATE, &SYSTIME, &SYSJDATE, and &SYSPARM. The following subsections contain the functions of each of the seven system variable symbols.

G.1. &SYSECT

&SYSECT is a system variable symbol used to represent the name of the control section containing a macroinstruction.

&SYSECT is assigned a value for each inner and outer macroinstruction processed by the assembler. This value is the name of the control section containing the macroinstruction. If &SYSECT is referenced in a macro definition, its substituted value is the name of the last CSECT, DSECT, or START directive that occurred prior to the macroinstruction. If a named CSECT, DSECT, or START directive did not appear prior to the macroinstruction, &SYSECT is assigned a null character value during the processing of the macro definition called by the macro call instruction.

Any CSECT or DSECT directives processed within a macro definition affect the value of &SYSECT for any subsequent inner macroinstructions in the definition and for any outer and inner macroinstructions that occur outside the current nest of macro definitions. However, the value of &SYSECT remains constant during the processing of a given macroinstruction, and is not affected by CSECT or DSECT directives or inner macroinstructions occurring in that macro definition.

G.2. &SYSLIST

Within a macro definition in macro format, each positional parameter may be referenced by a name; however, each positional parameter need not be named in the macro prototype statement and may be referenced in terms of its position within the macroinstruction operand field by wiring the system variable symbol &SYSLIST followed by an expression in parentheses. The value of the expression identifies the position of the parameter in the operand field. The expression may be a SETA symbol or a self-defining term. Therefore, if a macro definition prototype statement has the operand field:

```
&A,&B,&C
```

the first positional parameter is referenced either as &A or &SYSLIST(1), the second is referenced either as &B or &SYSLIST(2), and the third positional parameter is either &C or &SYSLIST(3), and so on. This capability, which is used to index through the positional parameters, treats each parameter in the same way.

A null character string is generated in place of &SYSLIST(m) if m is zero or greater than the number of positional parameters supplied in the macroinstruction.

The system variable &SYSLIST may not be used in a mixed-mode (positional and keyword parameters included) macro definition.

G.3. &SYSNDX

The assembler maintains a counter that is incremented by 1 each time the assembler encountered a macroinstruction. The value of this counter within the first macro is 1. The current value of this counter is supplied as the 4-digit character value of the system variable symbol &SYSNDX each time a macroinstruction is encountered. A macro definition that defines labels within the code it generates, and that may be called more than once in a single assembly, generally creates duplicate definitions of the same label. To avoid this problem, the system variable symbol &SYSNDX may be used as a suffix on the labels defined by the macro definition, so that each time the macro definition is called, it will define a different set of labels.

G.4. &SYSDATE

&SYSDATE is a system variable symbol, which you can reference in your program text or within a macro definition, to generate the date your program is assembled. The date is produced in your assembly listed as a character string representing the month, day, and year (mm/dd/yy) the program was assembled. If you

1. assemble your program,

2. store it in a library, and

3. retrieve the assembled program for execution at a later date,

any &SYSDATE reference in your program references the original assembly date, not the current date when your program is executed.

You specify &SYSDATE as either an operand in a source code statement, which defines a constant (DC), or an operand field literal.

```
LABEL    ΔOPERATIONΔ                  OPERAND
1         10        16
-----------------------------------------------------------
          .
          .
          .
ASMDATE   DC        C'&SYSDATE'
```

When this line of source code is assembled, the object code contains the current date.

You can also use the &SYSDATE system variable symbol as a literal.

```
LABEL    ΔOPERATIONΔ                    OPERAND
1         10        16
-----------------------------------------------------------
          .
          .
          .
          MVC       BUF,=C'&SYSDATE'
```

When this line of source code is executed, the assembly date is moved into a main storage area called BUF.

G.5. &SYSTIME

&SYSTIME is a system variable symbol, which you can reference either in your program text or within a macro definition, to generate the time of day your program is assembled. The time is produced in your assembly listing as a character string representing the hour, minute, and second (hh.mm.ss) the assembly was run. If you

1. assemble your program,

2. store it in a library, and

3. retrieve the assembled program for execution at another time,

any &SYSTIME reference in your program references the original assembly time, not the current time of execution.

You specify &SYSTIME as either an operand in a source code statement, which defines a constant (DC), or an operand field literal.

```
          .
          .
          .
ASMTIME   DC        C'&SYSTIME'
```

When this line of source code is assembled, the object code contains the current time.

You can also use the &SYSTIME system variable symbol as a literal.

```
          .
          .
          .
          MVC       BUF,=C'&SYSTIME'
```

When this line of source code is executed, the assembly time is moved into a main storage area called BUF.

G.6. &SYSJDATE

&SYSJDATE is a system variable symbol, which you can reference either in your program text or within a macro definition, to generate the Julian date when your program is assembled. The date is produced in your assembly listing as a character string representing the month, day, year, and Julian value — day of the year (mmddyjjj) the assembly was run. If you

1. assemble your program,
2. store it in a library, and
3. retrieve the assembled program for execution at another time,

any &SYSJDATE reference in your program references the Julian date of the original assembly.

You specify &SYSJDATE as either an operand in a source code statement, which defines a constant (DC), or an operand field literal.

```
LABEL     △OPERATION△                       OPERAND
1          10          16
--------------------------------------------------------
           .
           .
           .
JULDATE    DC          C'&SYSJDATE'
```

When this line of source code is assembled, the object code contains the Julian date.

You can also use the &SYSJDATE system variable symbol as a literal.

```
           .
           .
           .
           MVC         BUF,=C'&SYSJDATE'
```

When this line of source code is executed, the Julian date is moved into a main storage area called BUF.

G.7. &SYSPARM

&SYSPARM is a system variable symbol, which you can reference either in your program text or within a macro definition, to generate an 8-byte null character string at assembly time. The string is initially null but can be varied by using the PARAM statement as follows:

LABEL	△OPERATION△	OPERAND
//△PARAM△	SYSPARM='string'	

By using the PARAM statement, you can specify a string of up to eight characters, enclosed in apostrophes. Once you've altered the value of &SYSPARM, any reference to &SYSPARM produces the character string you specified in the PARAM statement, not a null character string.

To reference the &SYSPARM system variable symbol, you specify &SYSPARM as either an operand in a source code statement, which defines a constant (DC), or an operand field literal.

```
LABEL    △OPERATION△                       OPERAND
1         10        16

          .
          .
          .
NULSTRNG  DC        C'&SYSPARM'
          .
```

When this line of source code is assembled, the object code contains an 8-byte null character string.

You can also use the &SYSPARM system variable symbol as a literal.

```
          .
          .
          .
          MVC       BUF,=C'&SYSPARM'
```

If you don't precede this source code statement with a PARAM statement when this line of source code is executed, an 8-byte null character string is moved into a main storage area called BUF.

Index

Term	Reference	Page
A		
A instruction		
examples	10.2	10—8
formats	10.2	10—7
operational considerations	10.2	10—7
A, type constant	5.2.9	5—13
Absolute expression		
definition	4.4.1	4—16
examples	4.4.1	4—17
relocatable terms	4.4.1	4—16
Absolute term		
examples	4.4.1	4—16
expression	4.4	4—16
relocatable expression	4.4.2	4—17
ACTR statement		
format	27.4.3	27—24
function	27.4.3	27—24
AD instruction		
example	11.2	11—5
formats	11.2	11—3
operational considerations	11.2	11—4
Add (A) instruction	See A instruction.	
Add (AR) instruction	See AR instruction.	
Add decimal (AP) instruction	See AP instruction.	
Add half word (AH) instruction	See AH instruction.	
Add immediate (AI) instruction	See AI instruction.	
Add logical (AL) instruction	See AL instruction.	
Add logical (ALR) instruction	See ALR instruction.	
Add normalized (AD) instruction, long format	See AD instruction.	
Add normalized (ADR) instruction, long format	See ADR instruction.	
Add normalized (AE) instruction, short format	See AE instruction.	
Add normalized (AER) instruction, short format	See AER instruction.	
Add unnormalized (AU) instruction, short format	See AU instruction.	
Add unnormalized (AUR) instruction, short format	See AUR instruction.	
Add unnormalized (AW) instruction, long format	See AW instruction.	
Add unnormalized (AWR) instruction, long format	See AWR instruction.	

Term	Reference	Page
Address constants		
A-type	5.2.9	5—13
external	5.2.11	5—15
full-word	5.2.9	5—13
half-word	5.2.8	5—12
V-type	5.2.11	5—15
Y-type	5.2.8	5—12
Addressing		
main computer storage	3.1	3—1
register	3.2	3—3
ADR instruction		
example	11.3	11—7
formats	11.3	11—6
operational considerations	11.3	11—6
Advance listing (EJECT) direction &SYSLIST	See EJECT directive.	
AE instruction		
example	11.4	11—11
formats	11.4	11—9
operational considerations	11.4	11—10
AER instruction		
example	11.5	11—13
formats	11.5	11—12
operational considerations	11.5	11—12
AGO statement	27.2.2	27—16
AH instruction		
example	10.4	10—12
formats	10.4	10—11
operational considerations	10.4	10—11
AI instruction		
examples	10.5	10—14
formats	10.5	10—13
operational considerations	10.5	10—14
AIF statement		
examples	27.3.3	27—22
format	27.3.3	27—21
AL instruction		
example	12.2	12—2
formats	12.2	12—1
operational considerations	12.2	12—3
Alphabetic instruction listing	Table E—2	E—5
ALR instruction		
example	12.3	12—6
formats	12.3	12—5
operational considerations	12.3	12—5
American Standard Code for Information Interchange (ASCII) character codes	Table B—1	B—1
Ampersand		
character string	4.4.6	4—19
set symbols	27.1	27—2
&SYSDATE	G.4	G—2
&SYSECT	G.1	G—1
&SYSLIST		
example	G.2	G—1
mixed mode	G.2	G—2
null character string	G.2	G—2
&SYSNDX	G.3	G—2
&SYSPARM	G.7	G—5
&SYSTIME	G.5	G—3
And (N) instruction	See N instruction.	
And (NC) instruction	See NC instruction.	
And (NI) instruction	See NI instruction.	
And (NR) instruction	See NR instruction.	
ANOP statement		
example	27.2.4	27—19
format	27.2.4	27—18
AP instruction		
examples	9.3	9—9
formats	9.3	9—8
operational considerations	9.3	9—8
Apostrophe, character representation	4.4.6	4—19
Application instructions		
explicit length	7.1	7—4
formats	Fig. 7—1	7—2
implicit length	7.1	7—4
types	7.1	7—1
AR instruction		
example	10.3	10—9
format	10.3	10—9
operational considerations	10.3	10—9

Term	Reference	Page
Arithmetic operators		
absolute terms	4.4.1	4—16
description	4.3.1	4—14
ASCII	Table B—1	B—1
ASM Jproc call statement		
examples	29.3.1.1	29—5
format	29.3.1.1	29—4
function	29.3.1	29—3
ASML Jproc call statement		
format	29.3.2.1	29—10
function	29.3.2	29—10
ASMLG Jproc call statement		
format	29.3.3.1	29—11
function	29.3.3	29—11
Assemble and link-edit, ASML Jproc call statement	29.3.2.1	29—10
	29.3.2	29—10
Assemble, link-edit, and execute ASMLG Jproc call statement	29.3.3.1	29—11
Assembler application instruction notations		
assembled example	4.1.1	4—1
explicit format	4.1.1	4—2
implicit format	4.1.1	4—2
machine code	4.1.1	4—3
Assembler coding form		
additional coding rules	1.1.7	1—11
column 72	1.1.6	1—11
comment field	1.1.4	1—9
continuation field	1.1.2	1—7
description	1.1	1—1
	Fig. 1—5	1—13
label field	1.1.3	1—7
operand field	1.1.2	1—6
operation field	1.1.1	1—5
sample	Fig. 1—3	1—4
sequence field	1.1.5	1—10
Assembler control directives		
basic functions	Table 17—1	17—1
condition no operation (CNOP)	17.1	17—2
generate literal pool (LTORG)	17.3	17—5
program card (END)	17.2	17—4
program start (START)	17.5	17—8
specify location counter (ORG)	17.4	17—6
Assembling		
assemble, link-edit, and execute	29.3	29—3
coding	1.1	1—1
job control procedures	29.1	29—1
Assemble listing		
definition	4.1.2	4—6
example	29.6.1	29—14
organization	See organization of listing.	
sample	Fig. 1—8	1—16
Assign base register (USING) directive	See USING directive.	
Attribute references, conditional assembly		
applications	Table 27—3	27—26
count	27.5.5	27—31
example	27.5.6	27—32
format	27.5	27—25
integer	27.5.4	27—30
length	27.5.2	27—28
number	27.5.6	27—32
scale	27.5.3	27—30
Attributes of symbols		
length	4.2.3	4—11
	4.2.5	4—13
relocatability	4.2.3	4—11
type	Table 27—4	27—27
value	4.2.3	4—11
AU instruction		
example	11.6	11—16
formats	11.6	11—15
operational consideration	11.6	11—15
AUR instruction		
example	11.7	11—17
formats	11.7	11—17
operational consideration	11.7	11—17
AW instruction		
example	11.8	11—20
formats	11.8	11—19
operational consideration	11.8	11—19
AWR instruction		
example	11.9	11—21
formats	11.9	11—21
operational consideration	11.9	11—21

Term	Reference	Page
B		
B, type constant	5.2.3	5—9
BAL		
general description	1.1	1—1
sample program	Appendix A	
BAL instruction		
example	8.3	8—6
formats	8.3	8—5
BALR instruction		
example	8.3	8—6
format	8.3	8—5
Base and displacement constants (S), examples	5.2.10	5—13
Base register assignment directives		
assign base register (USING)	18.2	18—3
unassign base register (DROP)	18.1	18—2
Basic Assembly Language (BAL)	See BAL.	
BC instruction		
formats	8.4	8—9
operational considerations	8.4	8—11
BCR instruction		
formats	8.4	8—9
operational consideration	8.4	8—11
BCT instruction		
example	8.5	8—14
formats	8.5	8—13
BCTR instruction	8.5	8—13
Binary constants (B)		
padding	5.2.3	5—9
truncating	5.2.3	5—9
Binary representation	2.2	2—2
BLANK macro example	30.3	30—4
Braces within brackets	4.1.2	4—5
Branch and link (BAL) instruction	See BAL instruction.	
Branch and link (BALR) instruction	See BALR instruction.	
Branch on condition (BC) instruction	See BC instruction.	
Branch on condition (BCR) instruction	See BCR instruction.	
Branch on count (BCT) instruction	See BCT instruction.	
Branch on count (BCTR) instruction	See BCTR instruction.	
Branch on index high (BXH) instruction	See BXH instruction.	
Branch on index low or equal (BXLE) instruction	See BXLE instruction.	
Branching		
conditional branch (AIF)	27.2.3	27—17
define branch destination (ANOP)	27.2.4	27—18
function	27.2	27—15
macro definition exit (MEXIT)	27.2.5	27—19
sequence symbols	27.2.1	27—15
unconditional branch (AGO)	27.2.2	27—16
Branching instructions		
BAL	8.3	8—5
BALR	8.3	8—5
BC	8.4	8—9
BCR	8.4	8—9
BCT	8.5	8—13
BCTR	8.5	8—13
EX	8.8	8—20
extended mnemonic codes	8.2	8—2
	Table 8—1	8—3
function	8.1	8—1
BXH instruction		
examples	8.6	8—16
formats	8.6	8—15
BXLE instruction	8.7	8—18
byte		
definition	2.1	2—1
word structure	Fig. 4—2	4—7

Term	Reference	Page
C		
C instruction		
examples	10.6	10—16
formats	10.6	10—15
operational considerations	10.6	10—16
C, type constant	5.2.1	5—8
CCW instruction		
examples	21.6	21—10
format	21.6	21—9
operational considerations	21.6	21—10
CD instruction		
examples	11.10	11—24
formats	11.10	11—23
operational considerations	11.10	11—24
CDR instruction		
examples	11.11	11—26
formats	11.11	11—25
operational considerations	11.11	11—25
CE instruction		
examples	11.12	11—28
formats	11.12	11—27
operational considerations	11.12	11—28
CER instruction		
examples	11.13	11—30
formats	11.13	11—29
operational considerations	11.13	11—29
CH instruction		
examples	10.8	10—20
formats	10.8	10—19
operational considerations	10.8	10—20
Channel command word (CCW)	See CCW instruction.	
Character constants (C)		
padding	5.2.1	5—8
truncation	5.2.1	5—8
Character conversion codes		
American Standard Code for Information Interchange (ASCII)	Table B—1	B—1
	Table B—3	B—3
Extended Binary Coded Decimal Interchange Code (EBCDIC)	Table B—2	B—2
	Table B—3	B—3
punched card	Table B—3	B—3

Term	Reference	Page
Character expression	4.4.4	4—18
	27.1.8	27—14
Character representation		
alphabetic	2.4.1	2—5
asterisk	4.3.1	4—14
description	2.4	2—5
numeric	2.4.3	2—6
special characters	2.4.4	2—8
special letters	2.4.2	2—6
Character strings value	4.4.6	4—19
Check-off table		
explanation	Appendix D	
format	Appendix D	
terms	Appendix D	
CL instruction		
examples	12.8	12—20
formats	12.8	12—19
operational considerations	12.8	12—20
CLC instruction		
examples	12.9	12—23
formats	12.9	12—22
operational considerations	12.9	12—23
Clear channel (CLRCH) instruction	13.3.1	13—7
Clear device (CLRDV) instruction	13.3.2	13—8
CLCL instruction		
example	12.10	12—27
formats	12.10	12—25
programming considerations	12.10	12—26
CLI instruction		
examples	12.11	12—30
formats	12.11	12—29
operational considerations	12.11	12—30
CLIS instruction		
example	12.12	12—37
formats	12.12	12—34
operational considerations	12.12	12—36
CLM instruction		
example	12.13	12—41
formats	12.13	12—39
operating considerations	12.13	12—40
CLR instruction		
example	12.14	12—43
formats	12.14	12—42
operational considerations	12.14	12—43

Term	Reference	Page
CLRCH instruction	13.3.1	13—7
CLRDV instruction	13.3.2	13—8
CNOP directive		
examples	17.1	17—3
format	17.1	17—2
Code generation, repetitive	See repetitive code generation statements.	
CODEDIT section, listing content	28.3	28—2
	Table 28—1	28—2
Code an assembler program	1.1	1—1
COM directive		
examples	19.1	19—3
format	19.1	19—3
Comments field, coding form		
examples	1.1.4	1—9
forms	1.1.4	1—9
Comments statement	27.3.3	27—21
Common storage definition (COM) directive	See COM directive.	
Compare (C) instruction	See C instruction.	
Compare (CR) instruction	See CR instruction.	
Compare (CD) instruction, long format	See CD instruction.	
Compare (CDR) instruction, long format	See CDR instruction.	
Compare (CE) instruction, short format	See CE instruction.	
Compare (CER) instruction, short format	See CER instruction.	
Compare and swap under mask (CSM) instruction	See CSM instruction.	
Compare decimal (CP) instruction	See CP instruction.	
Compare half word (CH) instruction	See CH instruction.	
Compare logical (CL) instruction	See CL instruction.	
Compare logical (CLC) instruction	See CLC instruction.	
Compare logical (CLI) instruction	See CLI instruction.	
Compare logical (CLR) instruction	See CLR instruction.	
Compare logical characters long (CLCL) instruction	See CLCL instruction.	
Compare logical immediate and skip (CLIS) instruction	See CLIS instruction.	
Compare logical characters under mask (CLM) instruction	See CLM instruction.	
Complex relocatable expressions		
definition	4.4.3	4—18
examples	4.4.3	4—18
restrictions	4.4.3	4—18
Condition no operation (CNOP) directive	See CNOP directive.	
Conditional assembly		
attribute references	27.5	27—25
branching	27.2	27—15
error messages and comments	27.3	27—19
repetitive code generation	27.4	27—22
set symbols	27.1	27—2
summary	Table 27—1	27—1
Conditional assembly control counter (ACTR) statement	See ACTR statement.	

Term	Reference	Page
Conditional branch (AIF) statement	See AIF statement.	
Continuation column, coding form	1.1.4	1—9
Control section identification (CSECT) directive	See CSECT directive.	
Convert to binary (CVB) instruction	See CVB instruction.	
Convert to decimal (CVD) instruction	See CVD instruction.	
COPY directive		
example	21.5	21—8
format	21.5	21—8
Correction deck	F.2	F—5
Count attributes	27.5.5	27—31
CP instruction		
examples	9.4	9—11
formats	9.4	9—10
operational considerations	9.4	9—11
CR instruction		
examples	10.7	10—18
formats	10.7	10—17
operational considerations	10.7	10—17
Cross-reference section, listing content	28.5	28—4
	Table 28—3	28—4
CSECT directive		
examples	19.2	19—6
format	19.2	19—6
CSM instruction		
example	12.15	12—47
formats	12.15	12—44
operational considerations	12.15	12—46
CVB instruction		
examples	10.9	10—22
formats	10.9	10—21
operational considerations	10.9	10—22
CVD instruction		
example	10.10	10—25
formats	10.10	10—24
operational considerations	10.10	10—25

D

Term	Reference	Page
D instruction		
example	10.11	10—28
formats	10.11	10—27
operational considerations	10.11	10—28
D, type constant	5.2.12	5—15
Data area program exceptions	Appendix D	
DC statement		
alignment	5.1.1	5—4
constant specification	5.1.1	5—4
definition	5.1.1	5—4
example	5.1.1	5—4
DD instruction		
examples	11.14	11—32
formats	11.14	11—31
operational considerations	11.14	11—32
DDR instruction		
examples	11.15	11—35
formats	11.15	11—34
operational considerations	11.15	11—34

Term	Reference	Page
DE instruction		
examples	11.16	11—37
formats	11.16	11—36
operational considerations	11.16	11—36
Decimal divide, program exception	Appendix D	
Decimal instructions		
add decimal (AP)	9.3	9—8
compare decimal (CP)	9.4	9—10
divide decimal (DP)	9.5	9—13
move with offset (MVO)	9.12	9—75
multiply decimal (MP)	9.14	9—80
pack decimal (PACK)	9.15	9—82
packed decimal manipulation	9.1	9—1
subtract decimal (SP)	9.16	9—86
unpack decimal (UNPK)	9.18	9—95
zero and add (ZAP)	9.19	9—98
Decimal overflow, program exception	Appendix D	
Default option, example	4.1.2	4—5
Define branch destination (ANOP) statement	See ANOP statement.	
Define end of range (ENDO) statement	See ENDO statement.	
Define start of range (DO) statement	See DO statement.	
Definition-type symbol, DC statement	5.1.1	5—4
Definition types		
A character	5.2.9	5—13
B character	5.2.3	5—9
C character	5.2.1	5—8
D character	5.2.12	5—15
E character	5.2.12	5—15
F character	5.2.7	5—12
function	5.1	5—1
H character	5.2.6	5—11
P character	5.2.4	5—10
S character	5.2.10	5—13
V character	5.2.11	5—15
Y character	5.2.8	5—12
Z character	5.2.5	5—10
Delete operation code (OPSYM) directive	See OPSYM directive.	

Term	Reference	Page
DER instruction		
example	11.17	11—39
formats	11.17	11—38
operational considerations	11.17	11—38
Diagnostic section, listing content	28.6	28—5
	Table 28—4	28—5
Digit field	2.4.3.1	2—6
Digit select byte, ED instruction	9.6.1	9—17
Directives		
base register assignment	Section 18	
basic functions	Table 15—1	15—1
control	17.1	17—2
input and output control	Section 21	
listing control	Section 20	
program linking and sectioning	Section 19	
Divide (D) instruction	See D instruction.	
Divide (DR) instruction	See DR instruction.	
Divide (DD) instruction, long format	See DD instruction.	
Divide (DDR) instruction, long format	See DDR instruction.	
Divide (DE) instruction, short format	See DE instruction.	
Divide (DER) instruction, short format	See DER instruction.	
Divide decimal (DP) instruction	See DP instruction.	
DO statement	27.4.1	27—22
Double word	2.5	2—9
DP instruction		
example	9.5	9—14
formats	9.5	9—13
operational considerations	9.5	9—13
program exception	Appendix D	

Term	Reference	Page
DR instruction		
example	10.12	10—32
formats	10.12	10—31
operational considerations	10.12	10—31
DROP directive		
examples	18.1	18—2
format	18.1	18—2
DS statement		
alignment	5.1.7	5—8
constant specification	5.1.6	5—7
definition	5.1.1	5—4
definition-type symbol	5.1.4	5—6
duplication factor	5.1.3	5—5
examples	5.1.2	5—5
format	5.1	5—1
length factor (L)	5.1.5	5—6
DSECT directive		
examples	19.3	19—9
format	19.3	19—8
Dummy control section identification (DSECT) directive	See DSECT directive.	
Duplication factor		
function	5.1.3	5—5
length attribute	4.2.3	4—11
zero examples	Table 5—2	5—6

E

Term	Reference	Page
E, type constant	5.2.12	5—15
EBCDIC	Table B—2	B—2
ED instruction		
formats	9.6	9—16
operation summary	Table 9—1	9—26
operational considerations	9.6	9—17
Edit (ED) instruction	See ED instruction.	
Edit and mask (EDMK) instruction	See EDMK instruction.	
EDMK instruction		
example	9.7	9—28
formats	9.7	9—27
operational considerations	9.7	9—27
EIO instruction	13.3.3	13—9
EJECT directive	20.1	20—2
Ellipsis	4.1.2	4—5
END directive		
examples	17.2	17—4
format	17.2	17—4
End-of-data job control statement (/*)	29.5.1	29—13
ENDO statement		
examples	27.4.2	27—23
format	27.4.2	27—23
Enqueue I/O (EIO) instruction	13.3.3	13—9
ENTRY directive		
example	19.4	19—10
format	19.4	19—10
EQU directive		
examples	16.1	16—2
format	16.1	16—1

Term	Reference	Page
Equate (EQU) directive	See EQU directive.	
Error messages and comments		
MNOTE message statements	27.3.1	27—20
PNOTE message statements	27.3.2	27—21
EX instruction		
example	8.8	8—23
formats	8.8	8—20
operational considerations	8.8	8—22
program exception	Appendix D	
Exclusive or (X) instruction	See X instruction.	
Exclusive or (XC) instruction	See XC instruction.	
Exclusive or (XI) instruction	See XI instruction.	
Exclusive or (XR) instruction	See XR instruction.	
EXD instruction	13.4.1	13—16
Execute (EX) instruction	See EX instruction.	
Execute diagnose (EXD) instruction	13.4.1	13—16
Execution	1.4	1—20
Explicit format coding	7.2	7—6
Explicit source code		
examples	7.2	7—6
format	7.2	7—6
Exponent overflow, program exception	Appendix D	
Exponent underflow		
AD instruction	11.2	11—4
ADR instruction	11.3	11—7
AE instruction	11.4	11—10
AER instruction	11.5	11—13
program exception	Appendix D	
Expressions		
absolute term	4.4	4—16
character	4.4.4	4—18
complex relocatable	4.4.3	4—18
declaration	6.1	6—2
definition	4.4	4—16
evaluation	4.4	4—16
final result	4.4	4—16
intermediate results	4.4	4—16
length attribute	4.4.5	4—19
relocatable term	4.4.2	4—17
External address constants (V)		
padding	5.2.11	5—15
truncating	5.2.11	5—15
External symbol dictionary (ESD) listing	28.4	28—3
	Table 28—2	28—3
Externally referenced symbol declaration (ENTRY) directive	See ENTRY directive.	
Externally defined symbol declaration (EXTRN) directive	See EXTRN directive.	
EXTRN directive		
examples	19.5	19—11
format	19.5	19—11

Term	Reference	Page
F		
F, type constant	5.2.7	5—12
Field separator byte, ED instruction	9.6.1	9—17
Fill character, ED instruction	9.6.1	9—17
Fixed-point divide, program exception	Appendix D	
Fixed-point instructions		
add (A)	10.2	10—7
add (AR)	10.3	10—9
add half word (AH)	10.4	10—11
add immediate (AI)	10.5	10—13
compare (C)	10.6	10—15
compare (CR)	10.7	10—17
compare half word (CH)	10.8	10—19
convert to binary (CVB)	10.9	10—21
convert to decimal (CVD)	10.10	10—24
divide (D)	10.11	10—27
divide (DR)	10.12	10—31
general discussion	10.1	10—1
load (L)	10.13	10—33
load and test (LTR)	10.15	10—38
load complement (LCR)	10.16	10—40
load half word (LH)	10.17	10—42
load (LR)	10.14	10—35
load multiple (LM)	10.18	10—44
load negative (LNR)	10.19	10—50
load positive (LPR)	10.20	10—52
multiply (M)	10.21	10—55
multiply half word (MH)	10.23	10—61
multiply (MR)	10.22	10—59
shift left double (SLDA)	10.24	10—64
shift left single (SLA)	10.25	10—67
shift right double (SRDA)	10.26	10—70
shift right single (SRA)	10.27	10—73
store (ST)	10.28	10—76
store half word (STH)	10.29	10—78
store multiple (STM)	10.30	10—80
subtract (S)	10.31	10—83
subtract half word (SH)	10.33	10—88
subtract (SR)	10.32	10—85
Fixed-point numbers		
description	2.5	2—9
formats	Fig. 2—2	2—9
Fixed-point overflow, program exception	Appendix A	
Floating-point addition		
AD instruction	11.2	11—3
ADR instruction	11.3	11—6
AE instruction	11.4	11—9
AER instruction	11.5	11—12

Term	Reference	Page
Floating-point constants (E and D)		
example	5.2.12	5—17
formats	5.2.12	5—15
	Fig. 5—1	5—17
Floating point divide, program exception	Appendix D	
Floating-point instructions		
add normalized, long format (AD)	11.2	11—3
add normalized, long format (ADR)	11.3	11—6
add normalized, short format (AE)	11.4	11—9
add normalized, short format (AER)	11.5	11—12
add unnormalized, short format (AU)	11.6	11—15
add unnormalized, short format (AUR)	11.7	11—17
add unnormalized, long format (AW)	11.8	11—19
add unnormalized, long format (AWR)	11.9	11—21
compare, long format (CD)	11.10	11—23
compare, long format (CDR)	11.11	11—25
compare, short format (CE)	11.12	11—27
compare, short format (CER)	11.13	11—29
divide, long format (DD)	11.14	11—31
divide, long format (DDR)	11.15	11—34
divide, short format (DE)	11.16	11—36
divide, short format (DER)	11.17	11—38
general description	11.1	11—1
halve, long format (HDR)	11.18	11—40
halve, short format (HER)	11.19	11—42
load, long format (LD)	11.22	11—48
load, long format (LDR)	11.23	11—50
load, short format (LE)	11.24	11—52
load, short format (LER)	11.25	11—54
load and test, long format (LTDR)	11.30	11—64
load and test, short format (LTER)	11.31	11—66
load complement, long format (LCDR)	11.20	11—44
load complement, short format (LCER)	11.21	11—46
load negative, long format (LNDR)	11.26	11—56
load negative, short format (LNER)	11.27	11—58
load positive, long format (LPDR)	11.28	11—60
load positive, short format (LPER)	11.29	11—62
multiply, long format (MD)	11.32	11—68
multiply, long format (MDR)	11.33	11—70
multiply, short format (ME)	11.34	11—72
multiply, short format (MER)	11.35	11—74
store, long format (STD)	11.40	11—84
store, short format (STE)	11.41	11—86
subtract normalized, long format (SD)	11.36	11—76
subtract normalized, long format (SDR)	11.37	11—78
subtract normalized, short format (SE)	11.38	11—80
subtract normalized, short format (SER)	11.39	11—82
subtract unnormalized, long format (SW)	11.44	11—92
subtract unnormalized, long format (SWR)	11.45	11—94
subtract unnormalized, short format (SU)	11.42	11—88
subtract unnormalized, short format (SUR)	11.43	11—90

Term	Reference	Page
Floating-point numbers		
description	2.6	2—9
format	2.6	2—9
Free element list (FEL)	14.6	14—39
Full word	2.1	2—1
Full-word address constants (A)		
padding	5.2.9	5—13
truncating	5.2.9	5—13
Full-word fixed-point constants (F)		
padding	5.2.7	5—12
truncating	5.2.7	5—12

G

Term	Reference	Page
General register privileged instructions	13.8	13—24
General registers	3.2	3—3
Generate literal pool (LTORG) directive	See LTORG directive.	
Get IORB (GRB) instruction	13.3.8.2	13—14b
Global set symbols		
examples	27.1.2	27—6
format	27.1.2	27—5
GRB instruction	13.3.8.2	13—14b

H

Term	Reference	Page
H, type constant	5.2.6	5—11
Half word	2.1	2—1
Half-word address constants (Y)		
padding	5.2.8	5—12
truncating	5.2.8	5—12
Half-word fixed-point constants (H)		
padding	5.2.6	5—11
truncating	5.2.6	5—11
Halt device (HDV) instruction	13.3.4	13—10
Halve, long format (HDR) instruction	See HDR instruction.	
Halve, short format (HER) instruction	See HER instruction.	
HDR instruction		
example	11.18	11—41
formats	11.18	11—40
operational considerations	11.18	11—40
HDV instruction	13.3.4	13—10
HER instruction		
example	11.19	11—43
formats	11.19	11—42
operational considerations	11.19	11—42
Hexadecimal constant (X)		
padding	5.2.2	5—9
truncating	5.2.2	5—9
Hexadecimal-decimal integer conversion	C.1	C—1
Hexadecimal fractions (approximate values)	C.2	C—7
Hexadecimal representation		
description	2.3	2—3
notation	Table 2—2	2—4
HPR instruction	13.2.1	13—2

Term	Reference	Page
I		
IC instruction	See insert character instruction.	
ICM instruction	See insert characters under mask instruction.	
ICTL directive		
examples	21.1	21—3
format	21.1	21—2
Implicit format		
coding	7.3	7—6
examples	7.3	7—6
Implicit source code		
examples	7.1	7—4
	7.3	7—6
format	7.3	7—6
Include code from a library (COPY) directive	See COPY directive.	
Initial program load (IPL) instruction	13.10.1	
Inline expansion code		
BLANK macro	30.3	30—4
function	22.1	22—1
generation	23.1	23—1
variable	23.1	23—2
	30.3	30—4
Input and output control directives		
include code from a library (COPY)	21.5	21—8
input format control (ICTL)	21.1	21—2
input sequence control (ISEQ)	21.2	21—4
produce a record (PUNCH)	21.4	21—6
reproduce following record (REPRO)	21.3	21—5
Input format control (ICTL directive)	See ICTL directive.	
Input sequence control (ISEQ) directive	See ISEQ directive.	
Insert character (IC) instruction		
examples	12.20	12—62
formats	12.20	12—61
operational considerations	12.20	12—62
Insert characters under mask (ICM) instruction		
examples	12.21	12—66
formats	12.21	12—64
operational considerations	12.21	12—65
Insert storage key (ISK) instruction	13.2.2	13—3
Instruction		
aligning	7.1	7—4
application	7.1	7—1
branching	8.1	8—1
decimal	9.1	9—1
definition	7.4	7—6
fixed-point	10.1	10—1
floating-point	11.1	11—1
logical	12.1	12—1
privileged	13.1	13—1
RR	7.1	7—4
RS	7.1	7—4
RX	7.1	7—4
SI	7.1	7—4
SM	7.1	7—1
SS	7.1	7—4
status switching	13.1	13—1
Instruction listings		
alphabetic	Table E—2	E—5
machine code	Table E—3	E—11
mnemonics	Table E—1	E—1
Integer attributes		
examples	27.5.4	27—31
function	27.5.4	27—30
IPL instruction	13.10.1	13—26a
ISK instruction	13.2.2	13—3
ISEQ directive		
example	21.2	21—4
format	21.2	21—4
Italics	4.1.2	4—6

Term	Reference	Page
J		
Job control cards		
end-of-data job control statement (/*)	29.5.1	29—13
end-of-job control statement (/&)	29.5.2	29—13
terminate-the-card-reader job control statement (//FIN)	29.5.3	29—13
Job control procedures		
running an assembler program	29.1	29—1
source deck introduction	29.2	29—1
JOB control statement	29.2.1	29—2
K		
Keyword parameters		
coding	4.1.2	4—6
referencing in the call	25.3	25—4
	26.3	26—4

Term	Reference	Page
L		
L instruction		
example	10.13	10—34
formats	10.13	10—33
operational considerations	10.13	10—33
LA instruction		
examples	12.22	12—68
formats	12.22	12—67
operational considerations	12.22	12—67
Label argument, PROC format	25.6	25—12
Label field, coding form	1.1.3	1—7
LCDR instruction		
example	11.20	11—45
formats	11.20	11—44
operational considerations	11.20	11—44
LCER instruction		
example	11.21	11—47
formats	11.21	11—46
operational considerations	11.21	11—46
LCHR instruction	13.3.5	13—11
LCR instruction		
example	10.16	10—41
format	10.16	10—40
operational considerations	10.16	10—40
LCTL instruction	13.6.1	13—20
LD instruction		
example	11.22	11—49
formats	11.22	11—48
operational considerations	11.22	11—48
LDA instruction	13.3.6	13—12
LDR instruction		
example	11.23	11—50
formats	11.23	11—50
operational considerations	11.23	11—50
LE instruction		
example	11.24	11—53
formats	11.24	11—52
operational considerations	11.24	11—52
Least significant bit (LSB)	See LSB.	

Term	Reference	Page
Leave blank lines on listing (SPACE) directive	See SPACE directive.	
Length attribute		
application instruction	5.1.5	5—6
conditional assembly	27.5.2	27—28
duplication factor	4.2.2	4—11
examples	5.1.5	5—7
expressions	4.4.4	4—18
referencing	4.2.5	4—13
terms	4.2	4—8
Length factor		
boundary alignment	5.1.5	5—6
L character	5.1.5	5—7
LER instruction		
example	11.25	11—54
formats	11.25	11—54
operational consideration	11.25	11—54
Less than operator	4.3.3	4—15
LH instruction		
example	10.17	10—43
formats	10.17	10—42
operational considerations	10.17	10—42
LIA instruction	13.3.7	13—13
Linkage editor		
creating a load module	1.3	1—18
functions	29.3.3	29—11
Listing contents control (PRINT) directive	See PRINT directive.	
Listing control directives		
advance listing (EJECT)	20.1	20—2
basic functions	Table 20—1	20—1
leave blank lines on listing (SPACE)	20.3	20—5
listing content control (PRINT)	20.2	20—3
listing title declaration (TITLE)	20.4	20—6
Listing title declaration (TITLE) directive	See TITLE directive.	
Literals		
defined	4.2	4—8
	4.2.2	4—10
	5.3	5—18
examples	5.3	5—19
restrictions	5.3	5—19
source code	4.2.2	4—10
specification	4.2.2	4—11
LM instruction		
examples	10.18	10—45
formats	10.18	10—44
operational considerations	10.18	10—45
LNDR instruction		
example	11.26	11—57
formats	11.26	11—56
operational considerations	11.26	11—56
LNER instruction		
example	11.27	11—59
formats	11.27	11—58
operational considerations	11.27	11—58

Term	Reference	Page
LNR instruction		
example	10.19	10—51
format	10.19	10—50
operational considerations	10.19	10—50
Load (L) instruction	See L instruction.	
Load (LR) instruction	See LR instruction.	
Load (LD) instruction, long format	See LD instruction.	
Load (LDR) instruction, long format	See LDR instruction.	
Load (LE) instruction, short format	See LE instruction.	
Load (LER) instruction, short format	See LER instruction.	
Load address (LA) instruction	See LA instruction.	
Load and test (LTDR) instruction, long format	See LTDR instruction.	
Load and test (LTR) instruction	See LTR instruction.	
Load and test (LTER) instruction, short format	See LTER instruction.	
Load complement (LCDR) instruction, long format	See LCDR instruction.	
Load complement, short format (LCER) instruction	See LCER instruction.	
Load complement (LCR) instruction	See LCR instruction.	
Load control (LCTL) instruction	13.6.1	13—20
Load directive address (LDA) instruction	13.3.6	13—12
Load half word (LH) instruction	See LH instruction.	
Load I/O address (LIA) instruction	13.3.7	13—13
Load module, creating	1.3	1—18
Load multiple (LM) instruction	See LM instruction.	
Load negative (LNDR) instruction, long format	See LNDR instruction.	
Load negative (LNER) instruction, short format	See LNER instruction.	
Load negative (LNR) instruction	See LNR instruction.	
Load positive (LPR) instruction	See LPR instruction.	
Load positive (LPDR) instruction, long format	See LPDR instruction.	
Load positive (LPER) instruction, short format	See LPER instruction.	
Load program status word (LPSW) instruction	13.2.3	13—22
Load relocation register (LRR) instruction	13.7.1	13—22
Local set symbol		
examples	27.1.1	27—4
format	27.1.1	27—3
Location counter		
adding 1	4.2.4	4—13
asterisk	4.2.4	4—13
definition	4.2.4	4—12
restrictions	4.2.4	4—13
values	4.2	4—13

Term	Reference	Page
Logical instructions		
add logical (AL)	12.2	12—2
add logical (ALR)	12.3	12—5
and (N)	12.4	12—7
and (NC)	12.5	12—10
and (NI)	12.6	12—13
and (NR)	12.7	12—16
compare and swap under mask (CSM)	12.15	12—44
compare logical (CL)	12.8	12—19
compare logical (CLC)	12.9	12—22
compare logical (CLCL)	12.10	12—25
compare logical (CLI)	12.11	12—29
compare logical (CLR)	12.14	12—42
compare logical characters under mask (CLM)	12.13	12—39
compare logical immediate and skip (CLIS)	12.12	12—34
edit (ED)	9.6	9—16
edit and mark (EDMK)	9.7	9—27
exclusive or (X)	12.16	12—49
exclusive or (XC)	12.17	12—52
exclusive or (XI)	12.18	12—55
exclusive or (XR)	12.19	12—58
general description	12.1	12—1
insert character (IC)	12.20	12—61
insert characters under mask (ICM)	12.21	12—64
load address (LA)	12.22	12—67
move character (MVC)	9.9	9—63
move immediate (MVI)	12.23	12—69
move numeric (MVN)	9.11	9—72
move zones (MVZ)	9.13	9—78
or (O)	12.24	12—72
or (OC)	12.25	12—75
or (OI)	12.26	12—78
or (OR)	12.27	12—81
shift left double logical (SLDL)	12.28	12—84
shift left single logical (SLL)	12.29	12—87
shift logical (SHL)	12.30	12—91
shift right double logical (SRDL)	12.31	12—99
shift right single logical (SRL)	12.32	12—102
store character (STC)	12.33	12—105
subtract logical (SL)	12.35	12—111
subtract logical (SLR)	12.36	12—114
test under mask (TM)	12.37	12—117
test under mask and skip (TMS)	12.38	12—121
translate (TR)	12.39	12—126
translate and test (TRT)	12.40	12—129
Logical operators	4.3.2	4—15
Longitudinal redundancy check (LRC) instruction	13.4.2	13—17
Low order	4.1.2	4—6
Lowercase letters and terms, coding	4.1.2	4—5
LPDR instruction		
example	11.28	11—61
formats	11.28	11—60
operational considerations	11.28	11—60
LPER instruction		
example	11.29	11—63
formats	11.29	11—62
operational considerations	11.29	11—62
LPR instruction		
example	10.20	10—53
format	10.20	10—52
operational considerations	10.20	10—52
LPSW instruction	13.2.3	13—4
LR instruction		
example	10.14	10—36
format	10.14	10—35
operational considerations	10.14	10—35
LRC instruction	13.4.2	13—17
LRR instruction	13.7.1	13—22
LSB	2.1	2—1
	4.1.2	4—8
LTDR instruction		
example	11.30	11—65
formats	11.30	11—64
operational considerations	11.30	11—64
LTER instruction		
example	11.31	11—67
formats	11.31	11—66
operational considerations	11.31	11—66
LTORG directive	17.3	17—5
LTR instruction		
example	10.15	10—39
format	10.15	10—38
operational considerations	10.15	10—38

Term	Reference	Page
M		
M instruction		
example	10.21	10—56
formats	10.21	10—55
operational considerations	10.21	10—56
Machine code		
assembler format relationships	Fig. 4—1	4—4
definition	4.1.1	4—1
instruction listing	Table E—3	E—11
purpose	1.2	1—14
Macro call instruction		
call-names	23.3	23—7
format	23.3	23—6
function	22.2	22—3
keyword parameter	23.3	23—8
parameter-list	23.3	23—8
positional parameter	23.3	23—8
MACRO definition		
accessing in library	Fig. 23—2	23—5
accessing in source deck	Fig. 23—1	23—4
body	23.1	23—1
call-name	23.1	23—2
call instruction design	24.2	24—2
examples	Fig. 24—4 Section 30	24—12
general	24.1	24—1
heading	23.1 Fig. 24—1	23—1 24—1
macro instruction and definition, communication	Fig. 24—3	24—8
operation	23.1	23—1
parameter substitution	23.1	23—2
PROC and MACRO instructions compared	Fig. 24—2	24—6
prototype statement	24.2	24—3
storage	23.2	23—3
trailer	23.1	23—1
variable inline expansion code	23.1	23—2
variable symbol	24.2	24—2
Macro definition exit (MEXIT) statement	See MEXIT statement.	
Macro design		
macro call instruction	23.3	23—6
macro definition	23.1	23—1
macro definition storage	23.2	23—3
Macro facility		
processor	22.1	22—1
source code	22.2	22—2
MACRO format		
basic design	26.1	26—1
label argument	26.5	26—6
referencing keyword parameters in the call	26.3	26—4
referencing positional parameters in the call	26.2	26—2
referencing subparameters in the call	26.4	26—5
Macro (MACRO) definition	See MACRO definition.	
Macro processor		
inline macro expansion	Fig. 22—1	22—3
YMAC	22.1	22—1
Macro source code		
example	30.3	30—4
macro facility	22.2	22—3
types	22.2	22—3
YMAC	22.2	22—3
Main computer storage addressing		
data field	3.1.2	3—2
instruction	3.1.1	3—1
symbolic	3.1.1	3—1
Math tables		
hexadecimal-decimal integer conversion	C.1	C—1
hexadecimal fractions (approximate values)	C.2	C—7
powers of 2	C.3	C—8
powers of 16	C.4	C—9
MD instruction		
example	11.32	11—69
formats	11.32	11—68
operational considerations	11.32	11—68
MDR instruction		
example	11.33	11—71
formats	11.33	11—70
operational considerations	11.33	11—70
ME instruction		
example	11.34	11—73
formats	11.34	11—72
operational considerations	11.34	11—72
MER instruction		
example	11.35	11—75
formats	11.35	11—74
operational considerations	11.35	11—74

Term	Reference	Page
Message character, ED instruction	9.6.1	9—20
MEXIT statement	27.2.5	27—19
MH instruction		
example	10.23	10—63
formats	10.23	10—61
operational considerations	10.23	10—62
MIO instruction	13.3.8	13—14
Mnemonic		
definition	1.1	1—1
instruction listing	Table E—1	E—1
operation code	1.1.1	1—5
MNOTE message statement		
example	27.3.1	27—20
format	27.3.1	27—20
Most significant bit (MSB)	See MSB.	
Move character (MVC) instruction	See MVC instruction.	
Move character long (MVCL) instruction	See MVCL instruction.	
Move immediate (MVI) instruction	See MVI instruction.	
Move I/O (MIO) instruction	13.3.8	13—14
Move numeric (MVN) instruction	See MVN instruction.	
Move with offset (MVO) instruction	See MVO instruction.	
Move zones (MVZ) instruction	See MVZ instruction.	
MP instruction		
example	9.13	9—81
formats	9.13	9—80
operational considerations	9.13	9—81
MR instruction		
example	10.22	10—60
format	10.22	10—59
operational considerations	10.22	10—59

Term	Reference	Page
MSB	2.1 4.1.2	2—1 4—8
Multiple options, coding	4.1.2	4—5
Multiply (M) instruction	See M instruction.	
Multiply (MR) instruction	See MR instruction.	
Multiply, long format (MD) instruction	See MD instruction.	
Multiply, long format (MDR) instruction	See MDR instruction.	
Multiply, short format (ME) instruction	See ME instruction.	
Multiply, short format (MER) instruction	See MER instruction.	
Multiply decimal (MP) instruction	See MP instruction.	
Multiply half word (MH)	See MH instruction.	
MVC instruction		
examples	9.9	9—64
formats	9.9	9—63
operational considerations	9.9	9—63
MVCL instruction		
examples	9.10	9—69
formats	9.10	9—67
operational considerations	9.10	9—68
MVI instruction		
examples	12.23	12—70
formats	12.23	12—69
operational considerations	12.23	12—70

Term	Reference	Page
MVN instruction		
examples	9.11	9—73
formats	9.11	9—72
operational considerations	9.11	9—72
MVO instruction		
examples	9.12	9—76
formats	9.12	9—75
operational considerations	9.12	9—75
MVZ instruction		
examples	9.13	9—79
formats	9.13	9—78
operational considerations	9.13	9—78

N

Term	Reference	Page
N instruction		
examples	12.4	12—8
formats	12.4	12—7
operational considerations	12.4	12—8
NC instruction		
example	12.5	12—11
formats	12.5	12—10
operational considerations	12.5	12—11
NI instruction		
examples	12.6	12—14
formats	12.6	12—13
operational considerations	12.6	12—14
Normalization		
AD instruction	11.2	11—3
ADR instruction	11.3	11—6
AE instruction	11.4	11—9
AER instruction	11.5	11—12
NR instruction		
example	12.7	12—17
formats	12.7	12—16
operational considerations	12.7	12—17
Number attributes		
example	27.5.6	27—32
function	27.5.6	27—32
Numeric data, comparison	Table 2—1	2—2
Numeric representation		
packed format	2.4.3.2	2—7
unpacked format	2.4.3.1	2—6

Term	Reference	Page
O		
O instruction		
example	12.24	12—73
formats	12.24	12—72
operational considerations	12.24	12—73
Object code		
example	1.2	1—14
format	1.2	1—14
Object module format	Fig. 1—9	1—18
Object program		
definition	4.1.2	4—5
general	1.2	1—14
OC instruction		
example	12.25	12—76
formats	12.25	12—75
operational considerations	12.25	12—76
OI instruction		
examples	12.26	12—80
formats	12.26	12—78
operational considerations	12.26	12—79
Operand field, coding form	1.1	1—4
Operand length, ED instruction	9.6	9—16
Operation, program exception	Appendix D	
Operation field, coding form	1.1.1	1—4
Operators		
arithmetic	4.3.1	4—14
description	4.3	4—13
logical	4.3.2	4—15
priority	Table 27—2	27—9
relational	4.3.3	4—15
summary	Table 4—2	4—14
OPSYM directive		
examples	16.2	16—3
format	16.2	16—3
OPTION job control statement	29.2.2	29—2
Option-specifying operands	F.1	F—1
Optional information, coding	4.1.2	4—5
OR, bit comparison	4.3.2	4—15
OR instruction		
example	12.27	12—82
formats	12.27	12—81
operational considerations	12.27	12—82
Or (O) instruction	See O instruction.	
Or (OC) instruction	See OC instruction.	
Or (OI) instruction	See OI instruction.	
ORG directive		
examples	17.4	17—7
format	17.4	17—6
function	17.4	17—6
Organization of listing		
CODEDIT	28.3	28—2
cross-reference	28.5	28—4
diagnostic	28.6	28—5
example	29.6	29—14
external symbol dictionary (ESD)	28.4	28—3
preface	28.2	28—1
OS/3 Assembler	1.1	1—1
Output, assembly listing	Section 28	

Term	Reference	Page
P		
P type constant	5.2.4	5—10
Pack decimal (PACK) instruction	See PACK instruction.	
PACK instruction		
examples	2.4.3.2	2—7
	9.15	9—82
formats	9.15	9—82
operational considerations	9.15	9—83
packed format conversion	2.4.3.2	2—7
Packed decimal constant (P)		
padding	5.2.4	5—10
truncation	5.2.4	5—10
PARAM statement		
format	F.1	F—2
function	F.1	F—1
operational considerations	F.2	F—5
PNOTE message statement	27.3.2	27—21
Positional parameter 0		
description	25.5	25—9
example	30.1	30—1
Positional parameters		
coding	4.1.2	4—6
comma	4.1.2	4—6
referencing in the call	25.2	25—3
	26.2	26—2
Powers		
of 2	C.3	C—8
of 16	C.4	C—9
Preface section of listing	28.2	28—1
PRINT directive		
format	20.2	20—3
examples	20.2	20—4
Privileged instructions, status	13.1	13—1
PRB instruction	13.3.8.1	13—14a
PROC definition		
call instruction design	24.2	24—2
examples	Fig. 24—4	24—12
	30.2	30—2
general	24.1	24—1
MACRO and PROC instruction compared	Fig. 24—2	24—6
macro instruction and definition, communication	Fig. 24—3	24—8
variable symbol	24.2	24—2
PROC format		
basic design	25.1	25—1
label argument	25.6	25—11
multiple PROC names and positional parameter 0	25.5	25—9
referencing keyword parameters in the call	25.3	25—4
referencing positional parameters in the call	25.2	25—3
referencing subparameters in the call	25.4	25—7
Procedure (PROC) definition	See PROC definition.	
Produce a record (PUNCH) directive	See PUNCH directive.	
Program end (END) directive	See END directive.	
Program exceptions		
addressing	Appendix D	
data	Appendix D	
decimal divide	Appendix D	
decimal overflow	Appendix D	
execute	Appendix D	
exponent overflow	Appendix D	
exponent underflow	Appendix D	
fixed-point divide	Appendix D	
fixed-point overflow	Appendix D	
floating-point divide	Appendix D	
operation	Appendix D	
protection	Appendix D	
significance	Appendix D	
specification	Appendix D	

Term	Reference	Page
Program linking and sectioning directives		
common storage definition (COM)	19.1	19—3
control section identification (CSECT)	19.2	19—6
dummy control section identification (DSECT)	19.3	19—8
externally defined symbol declaration (ENTRY)	19.4	19—10
externally referenced symbol declaration (EXTRN)	19.5	19—11
Program start (START) directive	See START directive.	
Program status word (PSW)	See PSW.	
Protection program exception	Appendix D	
PSW	8.1	8—1
PUNCH directive		
example	21.4	21—7
format	21.4	21—6
Punched card codes	Table B—3	B—3
Punctuation marks, coding	4.1.2	4—5
Put IORB (PRB) instruction	13.3.8.1	13—14a

R

Term	Reference	Page
Reading instruction notation		
assembler application instruction	4.1.1	4—1
rules and meanings	4.1.2	4—5
REC statement		
format	F.2.2	F—7
operational consideration	F.2.2	F—7
Register instruction equate instruction	6.1	6—2
Register notations, example	4.1.2	4—6
Relational operators	4.3.3	4—15
Relative addressing, location counter	4.2.4	4—12
Relocatability attribute	4.2.3	4—11
Relocatable expression		
absolute terms	4.4.2	4—17
definition	4.4.2	4—17
examples	4.4.2	4—18
relocatable term	4.4.2	4—17
requirements	4.4.2	4—17
Relocatable term		
absolute expression	4.4.1	4—16
division	4.4	4—16
expression	4.4	4—16
multiplication	4.4	4—16
Repetitive code generation statements		
conditional assembly control counter (ACTR)	27.4.3	27—24
define end of range (ENDO)	27.4.2	27—23
define start of range (DO)	27.4.1	27—22
REPRO directive		
example	21.3	21—5
format	21.3	21—5
Reproduce following record (REPRO) directive	See REPRO directive.	
Reset (RESET) instruction	13.4.3	13—18
RR instruction	7.1	7—1
Running an assembler program		
examples	29.6.1	29—14
using job control	29.1	29—1
RS instruction	7.1	7—1

Term	Reference	Page
S		
S instruction		
example	10.31	10—84
formats	10.31	10—83
operational considerations	10.31	10—83
S switch, ED instruction	9.6	9—17
S, type constant	5.2.10	5—13
Sample program	Appendix A	
Scale attribute	27.5.3	27—30
SD instruction		
example	11.36	11—77
formats	11.36	11—76
operational considerations	11.36	11—77
SDR instruction		
example	11.37	11—79
formats	11.37	11—78
operational considerations	11.37	11—78
SDT		
binary conversion	4.2.1	4—9
character	4.2.1	4—9
decimal	4.2.1	4—9
hexadecimal	4.2.1	4—9
meaning	4.2	4—9
	4.2.1	4—9
negative term	4.2.1	4—10
use	4.2.1	4—9
SDV instruction	13.3.9	13—14
SE instruction		
example	11.38	11—81
formats	11.38	11—80
operational considerations	11.38	11—81
Self-defining terms (SDT)	See SDT.	
SEQ statement		
format	F.2.1	F—6
operational considerations	F.2.1	F—6
specifications	F.2.1	F—6
Sequence field, coding form	1.1.3	1—9
Sequence symbols	27.2.1	27—15
SER instruction		
example	11.39	11—83
formats	11.39	11—82
operational considerations	11.39	11—82
Service timer register (STR) instruction	13.5.1	13—19
Set program mask (SPM) instruction	See SPM instruction.	
SET statement		
examples	27.1.4	27—9
format	27.1.4	27—7
function	27.1.3	27—6
	27.1.4	27—6
operator priority	Table 27—2	27—9
Set storage key (SSK) instruction	13.2.4	13—5
SET symbols		
character expressions	27.1.8	27—14
function	27.1	27—2
global	27.1.2	27—5
local	27.1.1	27—3
SET statement	27.1.4	27—7
SETA statement	27.1.5	27—9
SETB statement	27.1.6	27—10
SETC statement	27.1.7	27—13
subscripted	27.1.9	27—14
value assignment	27.1.3	27—6
Set system mask (SSM) instruction	13.2.5	13—6
SETA statement		
examples	27.1.5	27—10
format	27.1.5	27—9
SETB statement		
examples	27.1.6	27—13
format	27.1.6	27—10
SETC statement		
examples	27.1.7	27—14
format	27.1.7	27—13
Setting of UPSI byte	29.5.4	29—14
SH instruction		
example	10.33	10—89
formats	10.33	10—88
operational considerations	10.33	10—89
Shaded option	4.1.2	4—5
Shift and round decimal (SRP) instruction	See SRP instruction.	

Term	Reference	Page
Shift left double (SLDA) instruction	See SLDA instruction.	
Shift left double logical (SLDL) instruction	See SLDL instruction.	
Shift left single (SLA) instruction	See SLA instruction.	
Shift left single logical (SLL) instruction	See SLL instruction.	
Shift logical (SHL) instruction	See SHL instruction.	
Shift right double (SRDA) instruction	See SRDA instruction.	
Shift right double logical (SRDL) instruction	See SRDL instruction.	
Shift right single logical (SRL) instruction	See SRL instruction.	
Shift right single (SRA) instruction	See SRA instruction.	
SHL instruction		
example	12.30	12—95
formats	12.30	12—91
operational considerations	12.30	12—94
SI instruction	7.1	7—1
Sign consideration for operand 2, ED instruction	9.6.1	9—17
Signed unpacked number	See zoned decimal constants.	
Significance, program exception	Appendix D	
Significance start byte, ED instruction	9.6	9—17
SKI statement		
format	F.2.3	F—8
function	F.2.3	F—7
operational considerations	F.2.3	F—8
specifications	F.2.3	F—8

Term	Reference	Page
SL instruction		
example	12.35	12—112
formats	12.35	12—111
operational considerations	12.35	12—112
SLA instruction		
example	10.25	10—69
formats	10.25	10—67
operational considerations	10.25	10—68
SLDA instruction		
example	10.24	10—65
formats	10.24	10—64
operational considerations	10.24	10—64
SLDL instruction		
example	12.28	12—85
formats	12.28	12—84
operational considerations	12.28	12—85
SLL instruction		
example	12.29	12—88
formats	12.29	12—87
operational considerations	12.29	12—88
SLM instruction		
example	13.8.1	13—24
formats	13.8.1	13—24
SLR instruction		
example	12.36	12—115
formats	12.36	12—114
operational considerations	12.36	12—114
SM instruction	7.1	7—1
Source card images		
definition	4.1.2	4—5
general	1.1	1—1
Source code		
literals	4.2.2	4—10
PROC (DO loop)	30.2	30—2
PROC (positional parameter 0)	30.1	30—1
Source deck		
definition	4.1.2	4—6
job control cards	29.5	29—12
requesting an assembly	29.6	29—14
Source deck introduction		
JOB control statement	29.2.1	29—2
OPTION job control statement	29.2.2	29—2
Source module correction routine		
control statements	F.2	F—5
correction deck	F.2	F—5
REC statement	F.2.2	F—7
SEQ statement	F.2.1	F—6
SKI statement	F.2.3	F—7

Term	Reference	Page
Source program		
general	1.2	1—14
meaning	4.1.2	4—6
SP instruction		
examples	9.16	9—87
formats	9.16	9—86
operational considerations	9.16	9—87
SPACE directive		
examples	20.3	20—5
format	20.3	20—5
Special characters	2.4.4	2—8
Special letters	2.4.2	2—6
Specification, program exceptions	Appendix D	
Specify location counter (ORG) directive	See ORG directive.	
SPM instruction		
example	13.12	13—28
formats	13.12	13—27
operational considerations	13.12	13—27
SR instruction		
examples	10.32	10—86
format	10.32	10—85
operational considerations	10.32	10—85
SRA instruction		
examples	10.27	10—75
formats	10.27	10—73
operational considerations	10.27	10—74
SRDA instruction		
examples	10.26	10—71
formats	10.26	10—70
operational considerations	10.26	10—71
SRDL instruction		
example	12.31	12—100
formats	12.31	12—99
operational considerations	12.31	12—100
SRL instruction		
example	12.32	12—103
formats	12.32	12—102
operational considerations	12.32	12—103
SRP instruction		
example	9.17	9—91
formats	9.17	9—89
operational considerations	9.17	9—90
SS instruction	7.1	7—1
SSK instruction	13.2.4	13—5
SSM instruction	13.2.5	13—6
SSTM instruction		
example	13.8.2	13—25
formats	13.8.2	13—25
ST instruction		
example	10.28	10—77
formats	10.28	10—76
operational considerations	10.28	10—76
Start device (SDV) instruction	13.3.9	13—15
START directive		
examples	17.5	17—9
format	17.5	17—8
Start-of-data job control statement(/$)	29.4	29—12
Status switching instructions		
general discussion	13.1	13—1
set program mask (SPM)	13.12	13—27
supervisor call (SVC)	13.13	13—29
test and set (TS)	13.14	13—31
STC instruction		
example	12.33	12—106
formats	12.33	12—105
operational considerations	12.33	12—106
STCM instruction		
example	12.34	12—109
formats	12.34	12—108
operational considerations	12.34	12—109
STCTL instruction	13.6.2	13—21
STD instruction		
example	11.40	11—84
formats	11.40	11—85
operational considerations	11.40	11—84
STE instruction		
example	11.41	11—87
formats	11.41	11—86
operational consideration	11.41	11—86

Term	Reference	Page
STH instruction		
example	10.29	10—79
formats	10.29	10—78
operational considerations	10.29	10—78
STM instruction		
examples	10.30	10—81
formats	10.30	10—80
operational considerations	10.30	10—81
Storage, type characteristics	Table 5—1	5—2
Store (ST) instruction	See ST instruction.	
Store (STD) instruction, long format	See STD instruction.	
Store (STE) instruction, short format	See STE instruction.	
Store character (STC) instruction	See STC instruction.	
Store characters under mask (STCM) instruction	See STCM instruction.	
Store control (STCTL) instruction	13.6.2	13—21
Store half word (STH) instruction	See STH instruction.	
Store multiple (STM) instruction	See STM instruction.	
Store relocation register (STRR) instruction	13.7.2	13—23
Store status (STS) instruction	13.4.3	13—18
STR instruction	13.5.1	13—19
STRR instruction	13.7.2	13—23
STS instruction	13.4.3	13—18
SU instruction		
example	11.42	11—89
formats	11.42	11—88
operational considerations	11.42	11—89

Term	Reference	Page
Subparameters		
referencing in the call (macro)	26.4	26—5
referencing in the call (proc)	25.4	25—7
Subroutine linkage	19.6	19—12
Subscripted SET symbols	27.1.9	27—14
Subtract (S) instruction	See S instruction.	
Subtract decimal (SP) instruction	See SP instruction.	
Subtract half word (SH) instruction	See SH instruction.	
Subtract logical (SL) instruction	See SL instruction.	
Subtract logical (SLR) instruction	See SLR instruction.	
Subtract normalized (SD), long format	See SD instruction.	
Subtract normalized (SDR) instruction, long format	See SDR instruction.	
Subtract normalized (SE) instruction, short format	See SE instruction.	
Subtract normalized (SER) instruction, short format	See SER instruction.	
Subtract (SR) instruction	See SR instruction.	
Subtract unnormalized (SW) instruction, long format	See SW instruction.	
Subtract unnormalized (SWR) instruction, long format	See SWR instruction.	
Subtract unnormalized (SU) instruction, short format	See SU instruction.	

Term	Reference	Page
Subtract unnormalized (SUR) instruction, short format	See SUR instruction.	
Supervisor call (SVC) instruction	See SVC instruction.	
Supervisor load multiple (SLM) instruction	13.8.1	13—24
Supervisor store multiple (SSTM) instruction	13.8.2	13—25
SUR instruction		
example	11.43	11—91
formats	11.43	11—90
operational considerations	11.43	11—90
SVC instruction		
example	13.13	13—30
formats	13.13	13—29
operational considerations	13.13	13—29
SW instruction		
example	11.44	11—93
formats	11.44	11—92
operational considerations	11.44	11—93
Switch list scan (SWLS) instruction	13.11.1	13—26b
SWLS instruction	13.11.1	13—26b
SWR instruction		
example	11.45	11—95
formats	11.45	11—94
operational considerations	11.45	11—94

Term	Reference	Page
Symbol		
applications	6.2	6—3
definition	4.2.3	4—11
	Section 6	
equivalent	6.1	6—2
invalid examples	6.1	6—2
valid examples	4.2.3	4—11
	6.1	6—1
values	4.2.3	4—11
Symbol attributes		
length	4.2.3	4—12
relocatability	4.2.3	4—12
value	4.2.3	4—12
System variable symbols		
&SYSDATE	G.4	G—2
&SYSECT	G.1	G—1
&SYSJDATE	G.6	G—4
&SYSLIST	G.2	G—1
&SYSNDX	G.3	G—2
&SYSPARM	G.7	G—5
&SYSTIME	G.5	G—3

Term	Reference	Page
T		
Terminate-the-card-reader job control statement (//FIN)	29.5.3	29—13
Terms		
classes	4.2	4—8
comparison	Table 4—1	4—9
Test and set (TS) instruction	See TS instruction.	
Test under mask (TM) instruction	See TM instruction.	
Test under mask and skip (TMS) instruction	See TMS instruction.	
TITLE directive		
examples	20.4	20—6
format	20.4	20—6
TM instruction		
examples	12.37	12—119
formats	12.37	12—117
operational considerations	12.37	12—118
TMS instruction		
example	12.38	12—123
format	12.38	12—121
operational considerations	12.38	12—123
TR instruction		
example	12.39	12—127
formats	12.39	12—126
operational considerations	12.39	12—127
Translate (TR) instruction	See TR instruction.	
Translate and test (TRT) instruction	See TRT instruction.	
TRT instruction		
example	12.40	12—131
formats	12.40	12—129
operational considerations	12.40	12—130
TS instruction		
examples	13.14	13—32
formats	13.14	13—31
operational considerations	13.14	13—32
U		
Unassign base register (DROP) directive	See DROP directive.	
Unconditional branch (AGO) statement	See AGO statement.	
Unpack decimal (UNPK) instruction	See UNPK instruction.	
Unpacked format, numeric representation	2.4.3.1	2—6
UNPK instruction		
examples	9.18	9—96
formats	9.18	9—95
operational considerations	9.18	9—96
Uppercase letters and terms, coding	4.1.2	4—5
UPSI byte, setting	29.5.4	29—14
USING directive		
examples	18.2	18—4
format	18.2	18—3

Term	Reference	Page
V		
V, type constant	5.2.11	5—15
Value attribute	4.2.3	4—11
W		
Word structure, example	Fig. 4—2	4—7
Writing conventions	1.1	1—1
X		
X instruction		
example	12.16	12—51
formats	12.16	12—49
operational considerations	12.16	12—50
X, type constant	5.2.2	5—9
XC instruction		
example	12.17	12—53
formats	12.17	12—52
operational considerations	12.17	12—53
XI instruction		
examples	12.18	12—57
formats	12.18	12—55
operational considerations	12.18	12—56
XOR, bit comparison	4.3.2	4—15
XR instruction		
example	12.19	12—59
formats	12.19	12—58
operational considerations	12.19	12—59

Term	Reference	Page
Y		
Y, type constant	5.2.8	5—12
Z		
Z, constant	5.2.5	5—10
ZAP instruction		
examples	9.19	9—99
formats	9.19	9—98
operational considerations	9.19	9—99
Zero and add (ZAP) instruction	See ZAP instruction.	
Zero result		
AD instruction	11.2	11—4
ADR instruction	11.3	11—7
AE instruction	11.4	11—10
AER instruction	11.5	11—13
Zone field	2.4.3.1	2—6
Zoned decimal constants (Z)		
padding	5.2.5	5—10
truncating	5.2.5	5—10

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

(Document Title)

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

(Name of User)

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

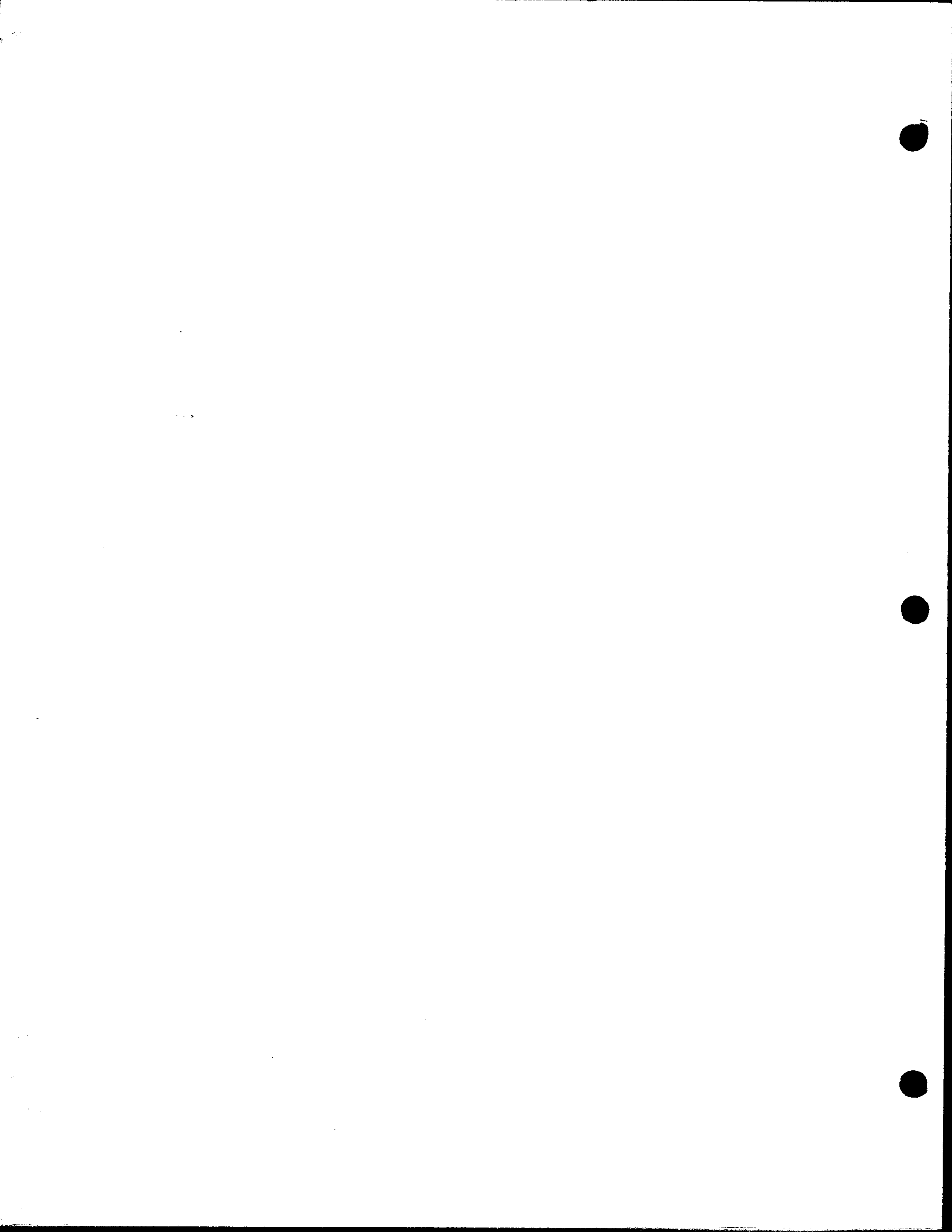
SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

CUT

FOLD



SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

(Document Title)

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

From:

(Name of User)

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

Cut along line.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

FOLD

CUT

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

(Document Title)

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

(Name of User)

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

FOLD

CUT