



---

## **COMPASS VERSION 3 REFERENCE MANUAL**

---

### **CDC® OPERATING SYSTEMS:**

**NOS 1**

**NOS 2**

**NOS/BE 1**

**SCOPE 2**

# CPU AND PP INSTRUCTION INDEX

## CPU INSTRUCTIONS:

Mnemonic Code	Operation Code (octal)	Page Number
AXi	+jk	21ijk
AXi	Bj,Xk	23ijk
BXi	Xj	10ijj
BXi	Xj*Xk	11ijk
BXi	Xj+Xk	12ijk
BXi	Xj-Xk	13ijk
BXi	-Xk	14ikk
BXi	-Xk*Xj	15ijk
BXi	-Xk+Xj	16ijk
BXi	-Xk-Xj	17ijk
CR	Xj,Xk	660jk
CW	Xj,Xk	670jk
CXi	Xk	47ikk
DF	Xj,K	036jk
DXi	Xj+Xk	32ijk
DXi	Xj-Xk	33ijk
DXi	Xj*Xk	42ijk
EQ	Bi,Bj,K	04ijk
ES	K	00000
FXi	Xj+Xk	30ijk
FXi	Xj-Xk	31ijk
FXi	Xi*Xk	40ijk
FXi	Xj/Xk	44ijk
GE	Bi,Bj,K	06ijk
GE	Bi,K	06i0K
GT	Bj,Bi,K	07ijk
GT	Bj,K	070jk
IBj	Bk	016jk
ID	Xj,K	037jk
IR	Xj,K	034jk
IXi†	Xj+Xk	36ijk
IXi	Xj-Xk	37ijk
IXi	Xj*Xk	42ijk
JP	Bi+K	02iik
LE	Bj,Bi,K	06ijk
LT	Bi,Bj,K	07ijk
LXi	+jk	20ijk
LXi	Bj,Xk	22ijk
MI	Xj,K	033jk
MI	Bi,K	07i0K
MJ		01300
MJ	Bj+K	013jk
MXi	+jk	43ijk
NE	Bi,Bj,K	05ijk
NG	Bi,K	07i0K
NG	Xj,K	033jk
NO	n	46n
NXi	Bj,Xk	24ijk
NZ	Bi,K	05i0K
NZ	Xj,K	031jk
OBj	Bk	017jk
OR	Xj,K	035jk
PL	Xj,K	032jk
PL	Bi,K	06i0K
PS	K	0000K
PXi	Bj,Xk	27ijk
RE	Bj+K	011jk
RI	Bk	0160k
RJ	K	0100K
RL	Bj+K	011jk
RO	Bk	0170k
RXi	Xj+Xk	34ijk
RXi	Xj-Xk	35ijk

RXi	Xj*Xk	41ijk	8-40
RXi	Xj/Xk	45ijk	8-43
RXj	Xk	014jk	8-20
SAi	Aj+K	50ijk	8-44
SAi	Bj+K	51ijk	8-44
SAi	Xj+K	52ijk	8-44
SAi	Xj+K	53ijk	8-44
SAi	Aj+Bk	54ijk	8-44
SAi	Aj-Bk	55ijk	8-44
SAi	Bj+Bk	56ijk	8-44
SAi	Bj-Bk	57ijk	8-44
SBi	Aj+K	60ijk	8-46
SBi	Bj+K	61ijk	8-46
SBi	Xj+K	62ijk	8-46
SBi	Xj+K	63ijk	8-46
SBi	Aj+Bk	64ijk	8-46
SBi	Aj-Bk	65ijk	8-46
SBi	Bj+Bk	66ijk	8-46
SBi	Bj-Bk	67ijk	8-46
SXi	Aj+K	70ijk	8-48
SXi	Bj+K	71ijk	8-48
SXi	Xj+K	72ijk	8-48
SXi	Xj+K	73ijk	8-48
SXi	Aj+Bk	74ijk	8-48
SXi	Aj-Bk	75ijk	8-48
SXi	Bj+Bk	76ijk	8-48
SXi	Bj-Bk	77ijk	8-48
TBj		016j0	8-21
UXi	Bj,Xk	26ijk	8-35
WE	Bj+K	012jk	8-15
WL	Bj+K	012jk	8-16
WXj	Xk	015jk	8-19
XJ	Bj+K	013jk	8-17
ZR	Xj,K	030jk	8-24
ZR	Bi,K	04i0K	8-26
ZXi	Bj,Xk	25ijk	8-35

## CMU INSTRUCTIONS:

CC	$l, k_a, c_a, k_b, c_b$	8-53
CU	$l, k_a, c_a, k_b, c_b$	8-54
DM	$l, k_s, c_s, k_d, c_d$	8-52
IM	Bj+K (464jk)	8-51
MD	$l, k_s, c_s, k_d, c_d$	8-51

## PP INSTRUCTIONS:

Name	Operation Code (octal)	Page Number
ACN	d	74d
ADC	c	21dm
ADD	d	31d
ADI	d	41d
ADM	m,d	51dm
ADN	d	16d
AJM	m,d	64dm
AOD	d	36d
AOI	d	46d
AOM	m,d	56dm
CCF	m,d	654dm††
CFM	m,d	674dm††
CRD	d	60d

CRM	m,d	61dm	9-18
CWD	d	62d	9-18
CWM	m,d	63dm	9-18
DCN	d	75d	9-24
EIM	m,d	61dm	9-21
EJM	m,d	67dm	9-19
EOM	m,d	65dm	9-21
ERN	d	270d	9-14
ESN	d	7700	9-25
ETN	d	260d	9-14
EXN	d	260d	9-12
FAN	d	76d	9-24
FIM	m,d	60dm	9-21
FJM	m,d	66dm	9-19
FNC	m,d	77dm	9-24
FOM	m,d	64dm	9-21
IAM	m,d	71dm	9-22
IAN	d	70d	9-22
IJM	m,d	65dm	9-19
IRM	m,d	62dm	9-21
LCN	d	15d	9-10
LDC	c	20dm	9-11
LDD	d	30d	9-15
LDI	d	40d	9-15
LDM	m,d	50dm	9-16
LDN	d	14d	9-10
LJM	m,d	01dm	9-7
LMC	c	23dm	9-11
LMD	d	33d	9-15
LMI	d	43d	9-15
LMM	m,d	53dm	9-16
LMN	d	11d	9-10
LPC	c	22dm	9-11
LPN	d	12d	9-10
LRD	d	24d	9-12
MAN	d	262d	9-12
MJN	r	07d	9-7
MXN	d	261d	9-12
NIM	m,d	63dm	9-21
NJN	r	05d	9-7
NOM	m,d	67dm	9-21
OAM	m,d	73dm	9-22
OAN	d	72d	9-22
OKM	m,d	66dm	9-21
PJN	r	06d	9-7
PSN	d	2400	9-11
RAD	d	35d	9-15
RAI	d	45d	9-15
RAM	m,d	55dm	9-16
RFN	d	74d	9-23
RJM	m,d	02dm	9-7
RPN	d	270d	9-13
SBD	d	32d	9-15
SBI	d	42d	9-15
SBM	m,d	52dm	9-16
SBN	d	17d	9-10
SCF	m,d	644dm††	9-20
SCN	d	13d	9-10
SFM	m,d	664dm††	9-20
SHN	r	10	9-9
SOD	d	37d	9-15
SOI	d	47d	9-15
SOM	m,d	57dm	9-16
SRD	d	25d	9-12
STD	d	34d	9-15
STI	d	44d	9-15
STM	m,d	54dm	9-16
UJN	r	03d	9-7
ZJN	r	04d	9-7

† There is also an integer divide macro (IXi Xj/Xk).

†† The operation code occupies 7 bits. The d field occupies 5 bits.



---

**COMPASS VERSION 3  
REFERENCE MANUAL**

---

**CDC<sup>®</sup> OPERATING SYSTEMS:  
NOS 1  
NOS 2  
NOS/BE 1  
SCOPE 2**

# REVISION RECORD

---

<u>Revision</u>	<u>Description</u>
A (11/01/75)	Original Release.
B (03/05/76)	Manual revised to reflect a new feature and to clarify existing material. The new feature is: CPI47, LDSET pseudo instruction. See list of effective pages.
C (03/25/77)	This reflects feature F7540, Model 176 support, feature CPI54, Weak Externals, and feature CPI61, Fast Dynamic Loader, as well as miscellaneous technical corrections, at PSR level 446.
D (03/31/78)	This revision documents COMPASS Version 3.5. New features include the DEBUG preset option and 8 lines/inch density on the load map.
E (10/31/78)	This revision documents COMPASS Version 3.6, PSR level 485. New features include the PD and PS control statement parameters and common common decks.
F (06/29/79)	This revision reflects the introduction of SEGLOAD common blocks. An error list parameter is added to the COMPASS call. Numerous minor technical corrections are made to section 12.
G (07/07/80)	This revision documents support for CYBER 170 Models 720, 730, 740, 750, and 760. An example program is included. Minor technical corrections are made throughout.
H (04/26/82)	Revised at PSR level 552 to document support of the CYBER 170 models 825, 835, and 855 Computer Systems, and to incorporate minor technical corrections. This is a complete reprint.
J (09/24/82)	Revised at PSR level 577 to document support of the CYBER 170 models 865 and 875 Computer Systems and to incorporate minor technical corrections.
K (10/21/83)	Revised at PSR level 599 to document support of the CYBER 170 Model 845 Computer System, support of PD, PS, and PW listing controls, and to incorporate minor technical corrections.
L (05/25/84)	Revised at PSR level 599 to document support of the CYBER 170 Model 815 and the CYBER 180 Computer Systems.

REVISION LETTERS I, O, Q, AND X ARE NOT USED

Address comments concerning this manual to:

© COPYRIGHT CONTROL DATA CORPORATION  
1975, 1976, 1977, 1978, 1979, 1980, 1982, 1983, 1984  
All Rights Reserved  
Printed in the United States of America

CONTROL DATA CORPORATION  
Publications and Graphics Division  
P. O. BOX 3492  
SUNNYVALE, CALIFORNIA 94088-3492

or use Comment Sheet in the back of this manual

# LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

<u>Page</u>	<u>Revision</u>	<u>Page</u>	<u>Revision</u>	<u>Page</u>	<u>Revision</u>
Front Cover	-	5-7	E	9-4	K
Inside Front Cover	H	5-8	A	9-5	K
Title Page	-	5-9	C	9-6 thru 9-10	H
ii	L	5-10	E	9-11	L
iii/iv	L	5-11	D	9-12	L
v	L	5-12	A	9-13	H
vi	L	5-13	A	9-14	L
vii/viii	H	5-14	E	9-15	H
ix	L	5-15	F	9-16	L
x	L	5-16 thru 5-25	A	9-17	H
xi	J	5-26	G	9-18	H
xii	L	5-27	B	9-19	L
1-1 thru 1-4	G	5-28 thru 5-35	A	9-20	H
2-1	H	6-1	E	9-21	H
2-2	A	6-2	A	9-22	L
2-3	A	6-3	A	9-23	L
2-4	D	6-4	L	9-24	H
2-5	L	6-5	A	9-25	H
2-6	A	6-6	A	10-1 thru 10-4	H
2-7	C	6-7	B	10-5	K
2-8 thru 2-10	H	6-8	A	10-6	H
2-11	K	6-9	L	10-7 thru 10-11	G
2-12	H	6-10	A	11-1 thru 11-4	H
2-13	H	7-1	A	11-5	C
2-14	J	7-2	A	11-6	A
2-15 thru 2-18	H	7-3	G	11-7 thru 11-10	G
2-19	K	7-4	D	11-11	L
2-20	L	7-5	E	11-12	H
2-21 thru 2-27	H	7-6	F	11-13	G
3-1	G	7-7	A	11-14	D
3-2	G	8-1	L	12-1	H
3-3	H	8-2	L	12-2	F
3-4	L	8-3	A	12-3 thru 12-18	H
3-5	L	8-4	L	12-19	J
3-6 thru 3-15	G	8-5	J	12-20 thru 12-34	H
4-1	G	8-6	J	A-1 thru A-4	A
4-2	G	8-7	H	B-1	H
4-3	L	8-8	L	B-2	H
4-4	G	8-9	L	B-3	G
4-5	A	8-10 thru 8-12	H	B-4	G
4-6 thru 4-8	L	8-13	L	B-5	A
4-9	A	8-14	H	C-1	A
4-10	L	8-15	L	D-1	J
4-11	A	8-16 thru 8-22	H	D-2 thru D-8	H
4-12	L	8-23 thru 8-25	L	E-1	L
4-13 thru 4-19	A	8-26 thru 8-31	H	E-2 thru E-6	H
4-20	F	8-32	L	F-1 thru F-3	H
4-21 thru 4-53	H	8-33	L	F-4	K
4-54	J	8-34 thru 8-45	H	Index-1 thru -7	H
4-55 thru 4-60	H	8-46	L	Index-8	J
4-61	L	8-47 thru 8-49	H	Index-9 thru -13	H
4-62 thru 4-80	H	8-50	L	Comment Sheet/Mailer	L
5-1	A	8-51	G	Summary Card	L
5-2	C	8-52 thru 8-54	K	Inside Back Cover	H
5-3	C	8-55	G	Back Cover	-
5-4	A	9-1	L		
5-5	A	9-2	L		
5-6	E	9-3	H		



## PREFACE

The CONTROL DATA® COMPASS Version 3.6 Assembler provides the user with a versatile, extensive language for generation of object code to be loaded and executed on the central processor unit (CPU) or a peripheral processor (PP or PPU). The assembler executes on the following computer systems and operating systems:

NOS 1 for the CDC® CYBER 170 Computer Systems; CYBER 70 Computer System models 71, 72, 73, and 74; and 6000 Computer Systems

NOS 2 for the CDC CYBER 180 Computer Systems; CYBER 170 Computer Systems; CYBER 70 Computer System models 71, 72, 73, and 74; and 6000 Computer Systems

NOS/BE 1 for the CDC CYBER 180 Computer Systems; CYBER 170 Computer Systems; CYBER 70 Computer System models 71, 72, 73, and 74; and 6000 Computer Systems

SCOPE 2 for the CDC CYBER 170 Computer System model 176, CYBER 70 Computer System model 76, and 7600 Computer Systems

The CYBER 170 Computer Systems include the following 800 Series models: 815, 825, 835, 845, 855, 865, and 875. The CYBER 180 Computer Systems include the following 800 Series models: 810, 830, 835, 845, and 855. The CYBER 170 models 835, 845, and 855 are the same machines as the CYBER 180 models 835, 845, and 855. References in the text to 800 Series models usually do not distinguish between CYBER 170 and CYBER 180.

The CDC CYBER 170 Computer System models 720 and 730 have unified processors and use the instructions noted in this publication for computer models with a Compare/Move Unit (CMU) such as the CYBER 170 Computer System model 172. Models 825, 835, 845, and 855 also support the compare/move instructions through simulation.

The CDC CYBER 170 Computer System models 740, 750, 760, 865, and 875 have functional units and use instructions noted in this publication for computer models with functional units such as the CYBER 170 Computer System model 175.

The reader is assumed to be familiar with a Control Data computer and operating system, and with assemblers in general.

### NOTE

Avoid continued use of COMPASS in creating application programs when possible. COMPASS and other machine-dependent languages can complicate migration to future hardware and software systems. Software mobility will be restricted by continued use of COMPASS for stand-alone programs, COMPASS subroutines embedded in programs using higher-level languages, and COMPASS owncode routines used with CDC standard products.

In this manual, the acronym ECS refers to all forms of extended memory unless otherwise noted, except in the context of a multiframe environment or distributive data path (DDP) access, in which case, models 176, 810, 815, 825, 830, 835, 845, 855, 865, and 875 are excluded.

Extended memory for the CYBER 170 Model 176, the CYBER 70 Model 76, and the 7600 is large central memory (LCM) or large central memory extended (LCME). Extended memory for models 810, 815, 825, 830, 835, 845, 855, 865, and 875 is unified extended memory (UEM). Extended memory for models 865 and

875 can also include extended core storage (ECS) or extended semiconductor memory (ESM). Extended memory for all other CYBER 170, CYBER 70, and 6000 Series Computer Systems is extended core storage (ECS) or extended semiconductor memory (ESM).

The CYBER 170 Model 176 supports direct LCM and LCME transfer instructions, as described in chapter 8. LCM and LCME transfers initiate an error exit, not a half exit, as noted in ECS/UEM Instructions, chapter 8.

Hardware descriptions and further programming information for the various forms of extended memory can be found in the appropriate hardware reference manuals.

In this manual, numbers occurring in text are decimal unless otherwise noted. Lowercase letters in formats depict variables. The examples assume that assembler numeric mode is decimal and that character mode is display code unless otherwise noted. In examples, statements generated by the assembler as a result of a call or a substitution are shown in shaded print.

General explanations of COMPASS concepts have been limited to the initial pages of each chapter or section, whenever possible. Subsequent material has been presented in a concise manner to aid in rapid access to reference information. In keeping with this concept, instruction indexes have been included inside the front and back covers.

Additional information essential to programming in the COMPASS environment can be found in the publications listed in this preface. The publications are listed alphabetically within groupings that indicate their approximate importance to readers of this manual. Applicable operating systems are also indicated.

The applications programmer will need the CYBER Record Manager Basic Access Methods and Advanced Access Methods manuals for information about the macros needed to define, access, and manipulate files. Information necessary to create and manipulate program structures can be found in the appropriate Loader reference manual (CYBER Loader for the NOS and NOS/BE operating systems, and the SCOPE 2 Loader for the SCOPE 2 operating system).

In addition to the above, the systems programmer will need the appropriate operating system manual, either the NOS 1 Reference Manual or the NOS 2 Reference Set, Volume 4, Program Interface, to obtain information about system macros.

The Software Publications Release History serves as a guide to the revision level of software documentation which corresponds to the Programming System Report (PSR) level of installed site software.

The following manuals are of primary interest:

<u>Publication</u>	<u>Publication Number</u>	<u>NOS 1</u>	<u>NOS 2</u>	<u>NOS/BE 1</u>	<u>SCOPE 2</u>
COMPASS Version 3 Instant	60492800	X	X	X	X
CYBER Loader Version 1 Reference Manual	60429800	X	X		
CYBER Record Manager Advanced Access Methods Version 2 Reference Manual	60499300	X	X	X	
CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual	60495700	X	X	X	
NOS Version 1 Reference Manual, Volume 1 of 2	60435400	X			
NOS Version 1 Reference Manual, Volume 2 of 2	60445300	X			



<u>Publication</u>	<u>Publication Number</u>	<u>NOS 1</u>	<u>NOS 2</u>	<u>NOS/BE 1</u>	<u>SCOPE 2</u>
NOS Version 2 Reference Set, Volume 4 Program Interface	60459690		X		
NOS/BE 1 Reference Manual	60493800			X	
SCOPE 2 Loader Version 2 Reference Manual	60454780				X
SCOPE 2 Reference Manual	60342600				X

The following manuals are of secondary interest:

<u>Publication</u>	<u>Publication Number</u>	<u>NOS 1</u>	<u>NOS 2</u>	<u>NOS/BE 1</u>	<u>SCOPE 2</u>
CYBER Interactive Debug Version 1 Reference Manual	60481400	X	X	X	
Modify Version 1 Reference Manual	60450100	X		X	
NOS Version 1 Diagnostic Index	60455720	X			
NOS Version 2 Diagnostic Index	60459390		X		
NOS/BE Version 1 Diagnostic Index	60456490			X	
NOS Version 1 Manual Abstracts	84000420	X			
NOS Version 2 Manual Abstracts	60485500		X		
NOS/BE Version 1 Manual Abstracts	84000470			X	
Software Publications Release History	60481000	X	X	X	X
Update Version 1 Reference Manual	60449900	X		X	X

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.



# CONTENTS

1	INTRODUCTION	1-1	3.2	Block Control Counters	3-3
			3.2.1	Origin Counter	3-3
1.1	Configuration	1-3	3.2.2	Location Counter	3-4
			3.2.3	Position Counter	3-4
1.2	Assembler Execution	1-3	3.2.4	Forcing Upper	3-4
			3.3	Relocatable Program Structure	3-5
1.3	Relocatable Object Program Execution	1-4	3.4	Absolute Program Structure	3-6
1.4	Interactive Program Debugging	1-4	3.4.1	Absolute Overlays	3-8
			3.4.2	Multiple Entry Point Overlays	3-12
2	LANGUAGE STRUCTURE	2-1	3.4.3	Partial Binary	3-12
			4	PSEUDO INSTRUCTIONS	4-1
2.1	Statement Format	2-1	4.1	Introduction to Pseudo Instructions	4-1
2.1.1	First Column	2-1	4.1.1	Types of Pseudo Instructions	4-1
2.1.2	Location Field	2-1	4.1.2	Required Pseudo Instructions	4-2
2.1.3	Operation Field	2-1	4.1.3	First Statement Group	4-2
2.1.4	Variable Field	2-2	4.1.4	Permissible Anywhere Instructions	4-2
2.1.5	Comments Field	2-2			
2.1.6	Comments Statement	2-2	4.2	Subprogram Identification	4-2
2.1.7	Statement Continuation	2-2	4.2.1	IDENT - Subprogram Identification	4-2
2.1.8	Coding Conventions	2-3	4.2.2	END - End of Subprogram	4-4
			4.3	Binary Control	4-6
2.2	Statement Editing	2-4	4.3.1	ABS - Absolute CPU Program	4-6
2.2.1	Concatenation	2-4	4.3.2	MACHINE - Declare Object Processor Type	4-7
2.2.2	Micro Substitution	2-4	4.3.3	PPU - CYBER 70 Model 76 or 7600 PPU Program	4-8
			4.3.4	PERIPH - CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 72, 73, 74; or 6000 Series PPU Program	4-10
2.3	Names	2-4	4.3.5	IDENT - Identify and Generate Overlay	4-11
			4.3.6	SEGMENT - Generate Binary Segment	4-15
2.4	Symbols	2-5	4.3.7	SEG - Write Partial Binary	4-16
2.4.1	Linkage Symbols	2-6	4.3.8	STEXT - Generate System Text Record	4-17
2.4.2	Default Symbols	2-7	4.3.9	COMMENT - Prefix Table Comment	4-20
2.4.3	Previously Defined Symbols	2-7	4.3.10	NOLABEL - Delete Header Table	4-20
2.4.4	Undefined Symbols	2-8	4.3.11	LCC - Loader Directive	4-21
2.4.5	Qualified Symbols	2-8	4.3.12	LDSET - Generate LDSET Object Directives	4-21
			4.4	Mode Control	4-24
2.5	CPU Registers	2-8	4.4.1	BASE - Declare Numeric Data Mode	4-24
			4.4.2	CHAR - Define Other Character Data Code	4-26
2.6	Special Elements	2-10	4.4.3	CODE - Declare Character Data Code	4-26
			4.4.4	QUAL - Qualify Symbols	4-28
2.7	Data Notation	2-11	4.4.5	B1=1 and B7=1 - Declare that B Register Contains One	4-30
2.7.1	Data Items	2-11	4.4.6	COL - Set Comments Column	4-31
2.7.2	Constants	2-11	4.5	Block Counter Control	4-32
2.7.3	Literals	2-12	4.5.1	USE - Establish and Use Block	4-32
2.7.4	Character Data Notation	2-13	4.5.2	USELCM - Establish and Use ECS/LCM Block	4-34
2.7.5	Numeric Data Notation	2-17	4.5.3	ORG and ORGC - Set Origin Counter	4-35
2.7.6	Hexadecimal Data Notation	2-22	4.5.4	BSS - Block Storage Reservation	4-37
			4.5.5	LOC - Set Location Counter	4-38
2.8	Expressions	2-23	4.5.6	POS - Set Position Counter	4-40
2.8.1	Types of Expressions	2-24			
2.8.2	Evaluation of Expressions	2-27			
3	PROGRAM STRUCTURE	3-1			
3.1	Subprogram Blocks	3-1			
3.1.1	Absolute Block	3-2			
3.1.2	Zero Block	3-2			
3.1.3	Literals Block	3-2			
3.1.4	User-Established Local Blocks	3-2			
3.1.5	Labeled Common Blocks	3-2			
3.1.6	Blank Common Blocks	3-3			
3.1.7	Redundant Block Names	3-3			

4.6	Symbol Definition	4-40	5.4	Macros and Opdefs	5-13
4.6.1	EQU or = - Equate Symbol Value	4-41	5.4.1	ENDM - End Macro Definition	5-14
4.6.2	SET - Set or Reset Symbol Value	4-41	5.4.2	MACRO - Macro Heading	5-15
4.6.3	MAX - Set Symbol to Maximum Value	4-42	5.4.3	Macro Calls	5-18
4.6.4	MIN - Set Symbol to Minimum Value	4-43	5.4.4	MACROE - Equivalenced Macro Header	5-24
4.6.5	MICCNT - Set Symbol to Micro Size	4-44	5.4.5	Equivalenced Macro Call	5-25
4.6.6	SST - System Symbol Table	4-45	5.4.6	OPDEF - Define CPU Operation	5-27
4.7	Subprogram Linkage	4-45	5.4.7	Opdef Call	5-29
4.7.1	ENTRY and ENTRYC - Declare Entry Symbols	4-45	5.4.8	LOCAL - Local Symbols	5-31
4.7.2	EXT - Declare External Symbols	4-47	5.4.9	IRP - Indefinitely Repeated Parameter	5-33
4.8	Data Generation	4-47	5.5	System Macro and Opdef Definitions	5-35
4.8.1	BSSZ and Blank Operation Field - Reserve Zeroed Storage	4-48	6	OPERATION CODE TABLE MANAGEMENT	6-1
4.8.2	DATA - Generate Data Words	4-48	6.1	Mnemonically Identified Instructions	6-3
4.8.3	DIS - Generate Words of Character Data	4-49	6.1.1	PPOP - PPU Operation Code	6-3
4.8.4	LIT - Declare Literal Values	4-51	6.1.2	OPSYN - Synonymous Mnemonic Operation	6-5
4.8.5	VFD - Variable Field Definition	4-53	6.1.3	NIL - Do Nothing Pseudo Instruction	6-6
4.8.6	CON - Generate Constants	4-54	6.1.4	PURGMAC - Purge Macros	6-7
4.8.7	R= - Conditional Increment Instruction	4-55	6.2	Syntactically Identified Instructions	6-7
4.8.8	REP, REPC, and REPI - Generate Loader Replication Table	4-57	6.2.1	CPOP - CPU Operation Code	6-7
4.9	Conditional Assembly	4-59	6.2.2	CPSYN - Synonymous CPU Instruction	6-10
4.9.1	ENDIF - End of IF Range	4-59	6.2.3	PURGDEF - Purge CPU Operation Code	6-10
4.9.2	ELSE - Reverse Effects of IF	4-60	7	MICROS	7-1
4.9.3	IFtype - Test Object Processor Type	4-60	7.1	Micro Substitution	7-1
4.9.4	IFop - Compare Expression Values	4-62	7.2	Micro Definition	7-2
4.9.5	IFPL and IFMI - Test Sign of Expression	4-64	7.2.1	MICRO - Define Micro	7-2
4.9.6	IF - Test Symbol or Expression Attribute	4-65	7.2.2	DECMIC - Decimal Micro	7-4
4.9.7	IFC - Compare Character Strings	4-68	7.2.3	OCTMIC - Octal Micro	7-4
4.9.8	SKIP - Unconditionally Skip Code	4-70	7.3	Predefined Micro Names	7-5
4.10	Error Control	4-71	7.3.1	DATE	7-5
4.10.1	ERR - Unconditionally Set Error Flag	4-71	7.3.2	JDATE	7-6
4.10.2	ERRxx - Conditionally Set Error Flag	4-72	7.3.3	TIME	7-6
4.11	Listing Control	4-73	7.3.4	BASE	7-6
4.11.1	LIST - Select List Options	4-73	7.3.5	CODE	7-6
4.11.2	EJECT - Eject Page and Begin New Sub-Subtitle	4-76	7.3.6	QUAL	7-6
4.11.3	SPACE - Skip Lines and Begin New Sub-Subtitle	4-76	7.3.7	SEQUENCE	7-7
4.11.4	TITLE - Assemble Listing Title	4-77	7.3.8	MODLEVEL	7-7
4.11.5	TTL - New Assembly Listing Title	4-78	7.3.9	PCOMMENT	7-7
4.11.6	NOREF - Omit Symbol References	4-78	8	CPU SYMBOLIC MACHINE INSTRUCTIONS	8-1
4.11.7	CTEXT and ENDX - Disable/Enable Listing of Common Deck Text	4-79	8.1	Machine Instruction Formats	8-1
4.11.8	XREF - Reference Symbolic Address	4-80	8.2	Instruction Execution	8-2
5	DEFINITION OPERATIONS	5-1	8.2.1	6600/6700 and CYBER 70 Model 74 Execution	8-2
5.1	External Text (XTEXT)	5-2	8.2.2	CYBER 180 Computer Systems; CYBER 170 Models 171, 172, 173, 174, 720, 730, 815, 825, 835, 845, and 855; CYBER 70 Models 71, 72, and 73; and 6200, 6400, 6500 Execution	8-4
5.2	Remote Assembly	5-3	8.2.3	CYBER 170 Models 175, 176, 740, 750, 760, 865, and 875; CYBER 70 Model 76; and 7600 Execution	8-5
5.2.1	RMT - Save Remote Code	5-3	8.3	Operating Registers	8-7
5.2.2	HERE - Assemble Remote Code	5-4	8.3.1	X Registers	8-7
5.3	Code Duplication	5-6	8.3.2	A Registers	8-7
5.3.1	DUP - Simple Duplication	5-6	8.3.3	B Registers	8-7
5.3.2	ECHO - Echoed Duplication	5-7			
5.3.3	STOPDUP - Stop Duplication	5-9			
5.3.4	ENDD - End Duplication Sequence	5-10			

8.4	Symbolic Notation	8-8	8.5.3	DM - Direct Move	8-52
8.4.1	Program Stop or Exchange Jump Instruction	8-13	8.5.4	CC - Compare Collated	8-53
8.4.2	Error Exit Instruction	8-14	8.5.5	CU - Compare Uncollated	8-54
8.4.3	Return Jump Instruction	8-14			
8.4.4	ECS/UEM Instructions	8-15			
8.4.5	LCM Block Copy Instructions	8-16	9	PP SYMBOLIC MACHINE INSTRUCTIONS	9-1
8.4.6	Exchange Jump Instruction	8-17			
8.4.7	Exchange Exit Instruction	8-18	9.1	Machine Instruction Formats	9-1
8.4.8	Direct LCM Transfer Instructions	8-19			
8.4.9	Direct UEM Transfer Instructions	8-20	9.2	Symbolic Notation	9-2
8.4.10	Reset Input Channel Buffer Instruction	8-21	9.2.1	Branch Instructions	9-7
8.4.11	Set Real-Time Clock Instruction	8-21	9.2.2	Shift Instruction	9-9
8.4.12	Reset Output Channel Buffer Instruction	8-22	9.2.3	No Address Mode Instructions	9-10
8.4.13	Read Channel Status Instructions	8-22	9.2.4	Constant Mode Instructions	9-11
8.4.14	Unconditional Jump Instruction	8-23	9.2.5	No Operation Instruction	9-11
8.4.15	X-Register Conditional Branch Instructions	8-24	9.2.6	Load and Store R Register Instructions	9-12
8.4.16	B-Register Conditional Branch Instructions	8-26	9.2.7	Exchange Jump Instructions	9-12
8.4.17	Transmit Instruction	8-27	9.2.8	Read Program Address Instruction	9-13
8.4.18	Logical Product Instruction	8-28	9.2.9	6416 PP Instructions	9-14
8.4.19	Logical Sum Instruction	8-28	9.2.10	Direct Address Mode Instructions	9-15
8.4.20	Logical Difference Instruction	8-29	9.2.11	Indirect Address Mode Instructions	9-15
8.4.21	Complement Instruction	8-29	9.2.12	Indexed Direct Address Mode Instructions	9-16
8.4.22	Logical Product and Complement Instruction	8-30	9.2.13	Central Read/Write Instructions	9-17
8.4.23	Complement and Logical Sum Instruction	8-30	9.2.14	I/O Branch Instructions	9-19
8.4.24	Complement and Logical Difference Instruction	8-31	9.2.15	I/O Test and Set Channel Flag Instructions	9-20
8.4.25	Logical Left Shift jk Places Instruction	8-31	9.2.16	I/O Branch Instructions	9-21
8.4.26	Arithmetic Right Shift jk Places Instruction	8-32	9.2.17	A Register Input/Output Instructions	9-22
8.4.27	Logical Left Shift (Bj) Places Instruction	8-32	9.2.18	Block Input/Output Instructions	9-22
8.4.28	Arithmetic Right Shift (Bj) Places Instruction	8-33	9.2.19	Set Output Record Flag Instruction	9-23
8.4.29	Normalize Instruction	8-34	9.2.20	Channel Function Instructions	9-24
8.4.30	Round and Normalize Instruction	8-35	9.2.21	Error Stop Instruction	9-25
8.4.31	Unpack Instruction	8-35	10	PROGRAM EXECUTION	10-1
8.4.32	Pack Instruction	8-36	10.1	Control Statements	10-1
8.4.33	Unrounded SP Floating Point Add Instructions	8-37	10.1.1	Job Statement	10-1
8.4.34	DP Floating Point Add Instructions	8-38	10.1.2	COMPASS Control Statement	10-2
8.4.35	Rounded SP Floating Point Add Instructions	8-38	10.1.3	LGO Control Statement	10-6
8.4.36	Long Add (Fixed Point) Instructions	8-39	10.1.4	Program Call Statement	10-6
8.4.37	Unrounded SP Floating Point Multiply Instruction	8-39	10.1.5	7/8/9 Card	10-7
8.4.38	Rounded SP Floating Point Multiply Instruction	8-40	10.1.6	6/7/8/9 Card	10-7
8.4.39	DP Floating Point Multiply Instruction	8-40	10.1.7	USER Control Statement (NOS 1 Only)	10-7
8.4.40	Integer Multiply Instruction	8-41	10.2	Sample Decks	10-8
8.4.41	Mask Instruction	8-42	11	LISTING FORMAT	11-1
8.4.42	Unrounded SP Floating Point Divide Instruction	8-42	11.1	Page Heading	11-1
8.4.43	Rounded SP Floating Point Divide Instruction	8-43	11.2	Header Information	11-1
8.4.44	Pass Instruction	8-43	11.2.1	Binary Control Card Summary	11-1
8.4.45	Population Count Instruction	8-43	11.2.2	Block Usage Summary	11-3
8.4.46	Set A Register Instructions	8-44	11.2.3	Entry Point List	11-4
8.4.47	Direct Read/Write Central Memory	8-46	11.2.4	External Symbol List	11-4
8.4.48	Set B Register Instructions	8-46	11.3	Octal and Source Statement Listing	11-5
8.4.49	Set X Register Instructions	8-48	11.4	Literals	11-7
8.5	CMU Symbolic Machine Instructions	8-50	11.5	Default Symbols	11-8
8.5.1	IM - Indirect Move	8-51	11.6	Assembler Statistics	11-8
8.5.2	MD - Indirect Move Descriptor Word	8-51	11.7	Error Directory	11-9
			11.8	Symbolic Reference Table	11-12

12	COMMON COMMON DECKS	12-1	12.3.10 WRITEC	12-32
12.1	Access to the Common Common Decks	12-1	12.3.11 WRITEH	12-33
12.2	Description of the Common Common Decks	12-3	12.3.12 WRITEO	12-33
12.2.1	COMCARG - Process Arguments	12-3	12.3.13 WRITES	12-33
12.2.2	COMCCDD - Convert Integer Constant to Decimal Display Code	12-4	12.3.14 WRITEW	12-34
12.2.3	COMCCFD - Convert Constant to F10.3 Format	12-4		
12.2.4	COMCCIO - Process I/O Operation	12-5		
12.2.5	COMCCOD - Convert Constant to Octal Display Code	12-5		
12.2.6	COMCCPT - Extract Comments Field from PREFIX Table	12-6		
12.2.7	COMCDXB - Convert Display Code to Binary	12-6		
12.2.8	COMCMNS - Move Non-Overlapping Bit String	12-7		
12.2.9	COMCMOS - Move Overlapping Bit String	12-7		
12.2.10	COMCMTM - Managed Table Macros	12-8		
12.2.11	COMCMTP - Managed Table Processors	12-9		
12.2.12	COMCMVE - Move Block of Data	12-13		
12.2.13	COMCRDC - Read Coded Line, C Format	12-13		
12.2.14	COMCRDH - Read Coded Line, H Format	12-14		
12.2.15	COMCRDO - Read One Word	12-15		
12.2.16	COMCRDS - Read Coded Line to String Buffer	12-16		
12.2.17	COMCRDW - Read Words to Working Buffer	12-16		
12.2.18	COMCRSR - Restore All Registers	12-17		
12.2.19	COMCSFN - Space Fill Name	12-18		
12.2.20	COMCSRT - Set Record Type	12-18		
12.2.21	COMCSST - Sort Table Using Shell Sort	12-18		
12.2.22	COMCSTF - Set Terminal File	12-20		
12.2.23	COMCSVR - Save All Registers	12-20		
12.2.24	COMCSYS - Process System Request	12-21		
12.2.25	COMCUPC - Unpack Control Card	12-22		
12.2.26	COMCWOD - Convert Word to Octal Display Code	12-23		
12.2.27	COMCWTC - Write Coded Line, C Format	12-23		
12.2.28	COMCWTH - Write Coded Line, H Format	12-23		
12.2.29	COMCWTO - Write One Word	12-24		
12.2.30	COMCWTS - Write Coded Line from String Buffer	12-25		
12.2.31	COMCWTW - Write Words from Working Buffer	12-25		
12.2.32	COMCXJR - Restore All Registers with a System XJR Call	12-26		
12.2.33	COMCZTB - Convert All 00 Characters to Blanks	12-26		
12.3	Macros That Call the Common Common Decks	12-27		
12.3.1	MESSAGE	12-28		
12.3.2	MOVE	12-29		
12.3.3	READC	12-29		
12.3.4	READH	12-30		
12.3.5	READO	12-30		
12.3.6	READS	12-30		
12.3.7	READW	12-31		
12.3.8	RECALL	12-31		
12.3.9	SYSTEM	12-32		

<b>APPENDIXES</b>	
A	Character Sets A-1
B	Assembly-Time I/O B-1
C	Binary Card Formats C-1
D	Hints on Using COMPASS D-1
E	Dayfile Messages E-1
F	Glossary F-1

<b>INDEX</b>	
<b>FIGURES</b>	
2-1	COMPASS Coding Form 2-3
3-1	Relocatable Program Structure 3-6
3-2	Absolute Program Structure 3-7
3-3	Overlay Hierarchy 3-9
3-4	IDENT-Type Overlay Structure 3-11
3-5	SEGMENT-Type Overlay Structure 3-13
3-6	SEG Partial Binary 3-14
3-7	IDENT Partial Binary Records 3-15
8-1	CPU 15-Bit Instruction Format 8-1
8-2	CPU 30-Bit Instruction Format 8-1
8-3	Arrangements of Instructions in a 60-Bit CPU Word 8-2
9-1	PP 12-Bit Instruction Format 9-1
9-2	PP 24-Bit Instruction Format 9-2
9-3	Central Memory Access Instruction Address Relocation (Models 810, 815, 825, 830, 835, 845, 855, 865, and 875) 9-2
11-1	Format of Octal and Source Statement Listing 11-5
11-2	Format of Symbolic Reference Table 11-13

<b>TABLES</b>	
8-1	CYBER 70 Model 74 and 6000/7600 Functional Units 8-3
8-2	CYBER 170 Model 175, 176, 740, 750, 760, 865, and 875; CYBER 70 Model 76; and 7600 Functional Units 8-6
8-3	CPU Instruction/Machine Model Correspondence 8-8
8-4	CPU Instruction/Functional Unit Correspondence 8-10
9-1	Peripheral Processor Instruction Designators 9-3
9-2	PP Instruction/Machine Model Correspondence 9-4
11-1	Fatal Errors 11-9
11-2	Informative Messages 11-12
12-1	Summary of Common Common Decks 12-2
12-2	Type Codes Returned by COMCSRT 12-19
12-3	Macros That Call Common Common Decks 12-27

# INTRODUCTION

1

---

This manual describes the features of the COMPASS Version 3 assembly language processor and the principles, methods, rules, and techniques of coding a COMPASS program.

The user is assumed to be familiar with a Control Data computer and operating system, and is assumed to be familiar with assemblers in general.

Readers with no previous experience with the COMPASS assembler are encouraged to direct their initial attention to the following sections of the manual:

Chapter 1	Introduction
Chapter 2	Language Structure
Chapter 3	Program Structure, sections 3.1 through 3.3
Chapter 4	Pseudo Instructions, sections 4.1 and 4.2
Chapter 8 or 9	CPU or PP Symbolic Machine Instructions, the chapter depending upon the machine language the user requires
Chapter 10	Program Execution
Appendix D	Hints on Using COMPASS (example program)

COMPASS, like other assemblers, is machine- and operating system-dependent. The user, therefore, should be aware of restrictions imposed on COMPASS by the programming environment. Specifically, the user should note:

Differences between CPU and PP program environments

Features of COMPASS not supported by a particular operating system

Machine and operating system limitations are outlined in the preface of this manual. The applicability of instruction sets is shown in the instruction indexes (inside front and back covers), and is addressed as necessary throughout the manual.

A COMPASS program consists of one or more subprograms. From source language subprograms, the assembler generates binary output acceptable for loading and execution. The programmer can divide a subprogram, whether it is assembled as absolute or relocatable, into areas called blocks. Blocks are assembled independently. Thus, they can be loaded and executed independently or linked by the system loader preparatory to execution of the program. This capability provides much flexibility in combining, segmenting, overlaying, and ordering blocks for execution.

Subprogram blocks consist of two types of source statements:

Symbolic machine instructions

Pseudo instructions

Symbolic machine instructions are the counterparts of the binary machine instructions. They provide a means of expressing symbolically the data manipulation functions of the machine. Each symbolic instruction typically generates one machine instruction.

Pseudo instructions do not have a one-to-one relationship with binary machine instructions. They are used, instead, to control aspects of the assembly process, such as:

- Storage allocation
- Symbol definition
- Subprogram linkage
- Listing options
- Automatic generation of predefined code sequences (macros)

From CPU source language subprograms, COMPASS generates absolute or relocatable binary output acceptable for loading and execution. From PPU source language subprograms, COMPASS generates absolute binary output to be loaded and executed on a peripheral processor unit. The operating system allows only specially privileged jobs to access a peripheral processor unit.

Features inherent to COMPASS include:

Free-field source statement format	Size of source statement fields is largely controlled by user.
Control of local and common blocks	Programmer and system can designate up to 255 areas to facilitate interprogram communication. In CPU programs, common areas can be defined in small core memory (CM or SCM) or extended or large core memory (ECS or LCM).
Preloaded data	Data areas may be specified and loaded in core memory with the source program.
Data notation	Data can be designated in integer, floating-point, and character string notation. It can be introduced into the program as a data item, a constant, or a literal.
Address arithmetic	Addresses can be specified making extensive use of constants, symbolic addresses, and arithmetic expressions.
Symbol equation and redefinition	Equation and redefinition of symbols allow extensive parameterization of assembly and linkage of subprograms and subroutines.
Symbol qualification	Ability to associate a symbol qualifier with a symbol defined within a qualified sequence to render the symbol unique to the sequence. An unqualified symbol is global and can be referred to from within any sequence without qualification.
Binary control	The programmer can specify whether binary output is to be absolute or relocatable. Absolute code can be generated for any PPU or CPU. Relocatable code can be generated for any CPU. Binary can be written as overlays or as partial records.
Selective assembly of code sequences	Assembly-time tests allow the user to select or alter code sequences.
Mode control	Ability to specify the base to be used for numeric notation not explicitly defined as octal or decimal, and to specify the code conversion to be applied to character data as either display code, ASCII, internal BCD, or external BCD.



Listing control	Assembly-time control of list content.
Micro coding	Substitution of sequences of characters defined in the program whenever the micro name is referenced. Several micros are predefined by the system for user convenience.
Macro coding	Assembly of sequences of instructions defined in the program or on the system library whenever the macro name is referenced. Macro definitions can be redefined or purged from the operation code table.
Operation code table	The programmer can specify or respecify the syntax of a CPU or PPU instruction. The assembler generates an entry in the operation code table for the instruction. No macro or opdef definition is associated with the entry.
Operation code definition	Assembly of sequences of instructions defined in the program or on the system library whenever an operation code of the specified syntax is referenced.
Code repetition	Sequences of code can be repeated during assembly or at load time.
Remote assembly	Defers assembly of defined coding sequence until later in the assembly.
Library routine calls	Routines can be called from the system library.
Diagnostics	Diagnostics for source program errors are included on output listing.

## 1.1 CONFIGURATION

The hardware requirements for executing COMPASS on a CPU are the minimum required for the operating system.

## 1.2 ASSEMBLER EXECUTION

COMPASS is called from the system library by a COMPASS control statement (chapter 10) or FORTRAN compiler upon encountering a COMPASS IDENT statement in the source input file. Parameters on the control statement specify files used during the assembler run such as the file containing source statements and the files to receive listable output and load-and-go output. The COMPASS assembler executes as a CPU program.

The operating system allocates the input/output resources as needed and performs all input/output required during the assembly.

COMPASS assembles each subprogram on the source file, in turn, in two passes. During the first pass, it reads each source language instruction, expands and edits called sequences as needed, interprets the operation code, and assigns storage.

The function of the second pass is to assign block origins, locate literals, fill in all valid symbol values and produce the assembly listing and binary output. Finally, it prepares the symbolic reference table and reinitializes itself preparatory to assembling the next subprogram.

COMPASS alters its field length dynamically, thus ensuring that central memory requirements for tables used by the assembler are satisfied. The assembler requests additional central memory as needed up to a threshold field length. (The threshold value is determined by the installation.) When the threshold field length is reached, the intermediate file and cross-references are transferred to the system mass storage device. If additional core is needed, the assembler continues to request central memory up to the maximum available to the job. (COMPASS may use any ECS/LCM space assigned to the job for table space.) If core requirements are still not satisfied, COMPASS aborts and issues a diagnostic message.

All nested processing of macros and similar definitions is handled in a single recursive push-down stack. COMPASS has a maximum recursion level of 400; that is, COMPASS allows nesting to a depth of 400.

### **1.3 RELOCATABLE OBJECT PROGRAM EXECUTION**

When the assembler has completely processed the source deck, a control statement (for example, LGO) can be used to call for loading and execution of a CPU object program from the load-and-go file. The loader links the newly assembled subprogram to any previously assembled subprograms and subroutines referred to by the new program and to programs on any other files specified by the programmer. After all subprograms are loaded and linked, the operating system begins program execution at a location specified by one of the subprograms. Data for the object program can be on some programmer-specified file. Normally, this loading and execution does not take place if the COMPASS assembler detects fatal errors.

### **1.4 INTERACTIVE PROGRAM DEBUGGING**

A COMPASS program that assembles without fatal errors can be executed under control of the CYBER Interactive Debug (CID) software. CID allows the programmer to correct errors in program logic from a terminal. Using CID, the COMPASS programmer can:

- Suspend program execution at a specific location or upon occurrence of a specific trap condition, such as execution of a return jump instruction

- Alter location content during program suspension

- Resume execution at a specified location or at the location where suspension occurred

A complete description of CID features and use is given in the CYBER Interactive Debug Reference Manual listed in the preface.

---

## 2.1 STATEMENT FORMAT

A COMPASS language source program consists of a sequence of symbolic machine instructions, pseudo instructions, and comment lines. With the exception of the comment lines, each statement consists of a location field, an operation field, a variable field, and a comments field. Each field is terminated by one or more blank characters. However, a blank embedded in a character data item, parenthesized macro parameter, or comments field does not terminate a field. The size of the variable field is restricted by the maximum statement size only. Statement format is essentially free field.

When punched on cards, each card is considered a line. A single statement may be composed of as many as ten lines. Information beyond column 72 is not interpreted by COMPASS but does appear on the assembly listing. Thus, columns 73 through 80 can be used for additional comments or sequencing. Columns 81 through 90 are used for sequencing by library maintenance programs; they are normally not used by the programmer. A line that contains two or more consecutive colons may be read and printed as two lines because of operating system conventions for delimiting line images.

### 2.1.1 FIRST COLUMN

The contents of column one designate the type of line, as follows:

- ,(comma) Designates the line as a continuation of the previous line.
- \*(asterisk) Designates the line as a comments line.
- other Indicates the beginning of a new statement.

### 2.1.2 LOCATION FIELD

The location field entry begins in column one or two of a new statement line and is terminated by a blank. If columns one and two are blank, the location field has no entry. A location field entry is usually optional. It may contain a symbol or name according to the requirements of the operation field, or a plus sign (+) or a minus sign (-). (See Block Control Counters, chapter 3.)

### 2.1.3 OPERATION FIELD

If the location field is blank, the operation field can begin in column three. If the location field is nonblank, the operation field begins with the first nonblank character following the location field and is terminated by one or more blanks. The operation field is blank if there are no nonblank characters between the location field and column 30. The following are legal field entries:

Central processor unit mnemonic operation code and, optionally, the variable subfields with each variable subfield preceded by a comma.

Peripheral processor unit mnemonic operation code

Pseudo instruction mnemonic operation code

Macro name

Blank

#### 2.1.4 VARIABLE FIELD

The contents of the operation field determine if any entry is required in the variable field which consists of one or more subfields separated by commas. The variable field begins with the first nonblank character following the operation field and is terminated by one or more blanks. It is blank if there are no nonblank characters between the operation field and column 30.

A variable subfield contains one of the following:

Data item

Expression

Register designator

Name

Special element

Entry uniquely defined for the instruction

#### 2.1.5 COMMENTS FIELD

Comments are optional and begin with the first nonblank character following the variable field or, if the variable field is missing, begin no earlier than column 30. The beginning comments column can be changed through the COL pseudo instruction (chapter 4).

#### 2.1.6 COMMENTS STATEMENT

A comments statement is designated either by an asterisk in column 1 or by blanks in columns 1-29. Comments statements are listed in assembler output but have no other effect on assembly. A statement beginning with \* is not counted in line counts for IF-skipping (Section 4.9) and definition operations (chapter 5) and is not included in definitions. A statement having columns 1-29 blank is counted.

#### 2.1.7 STATEMENT CONTINUATION

Normally, column 72 terminates a source statement that has not yet terminated. However, a statement that cannot be contained in the first 72 characters can be continued on the next line by placing a comma in column one and continuing the field in column two. A maximum of nine continuation lines is permitted for a statement. The break between lines need not coincide with a field or subfield separator; even a symbol can be split between two lines. Continuation lines beyond the ninth, and continuation lines following a terminated statement are considered comment lines.



## 2.2 STATEMENT EDITING

COMPASS reads statements in sequence from the source file. It immediately edits and interprets each statement unless (1) it is a comments statement of the type indicated by an asterisk in column one, or (2) it is part of a definition, that is, it is a statement between a macro or OPDEF header and an ENDM, between a DUP or ECHO and an ENDD, or between an RMT pair. Statements within definitions are saved for editing and interpretation when the definition is referenced or expanded. ENDD and ENDM are part of the definition they terminate and are not edited. Statements within the range of a conditional (IF type) pseudo instruction are edited even when they are skipped. COMPASS performs two types of editing: concatenation, and micro substitution.

### 2.2.1 CONCATENATION

COMPASS examines the statement for the concatenation character  $\rightarrow$  and removes it from any field of the statement so that the two adjoining columns are linked. The most common use of the concatenation character is as a delimiter for a substitutable parameter name in a macro definition when there is no other type of delimiter already there to set off the parameter name. After the substitution takes place, the  $\rightarrow$  is superfluous and is removed by editing before the definition is interpreted.

Each removal of  $\rightarrow$  shifts the remaining columns in the statement left one character. This could become significant when comments follow a blank variable field because the comments would be shifted left and interpreted as a variable field entry rather than comments.

### 2.2.2 MICRO SUBSTITUTION

COMPASS examines the statement for pairs of micro marks ( $\neq$ ) that delimit references to micro definitions (chapter 7) and replaces each reference (including the micro marks) with the micro character string referenced. The string that replaces the reference in the statement can be a different number of characters than the reference so that after the substitution, remaining characters in the statement are shifted left or right, accordingly. If, as a result of micro substitution, column 72 of the last statement read is exceeded, the assembler creates up to a maximum of nine continuation cards, beyond which it discards excess without notification on the listing. No replacement takes place if the micro name is unknown or if one of the micro marks has been omitted. The micro marks and name remain in the line. In the first case, the assembler flags a nonfatal assembly error. However, a single micro mark is not illegal and does not produce an error flag.

If the micro name is null (i. e., the two micro marks are adjacent) both micro marks are deleted and no error flag is set.

The columnar displacement caused by a micro replacement could also affect the relationship of fields to the beginning comments column. For example, it could shift the operation or variable field right beyond column 30, or could shift comments left into a blank field.

A line that contains two or more consecutive colons after editing may be printed as two lines because of operating system conventions for delimiting print lines.

## 2.3 NAMES

A name is a sequence of characters that identifies one of the following:

- Subprogram or overlay
- Block

**Macro definition**

**Remote definition**

**Duplicated sequence (DUP or ECHO)**

**IF sequence**

**Micro**

A comma or a blank terminates a name. Concatenation marks and pairs or micro marks are removed before the name is scanned (see Statement Editing).

A CPU subprogram name or overlay name is used for linkage with other subprograms. It must begin with a letter (A-Z) and is limited to seven characters maximum. Conventions imposed on names by the operating system could restrict the use of certain characters in names. There is no restriction on the first character for a PPU subprogram or overlay name. For a CYBER 70 Model 76 or 7600 PPU assembly, the name can be seven characters, but for a CYBER 180 Series, a CYBER 170 Series, a CYBER 70 Model 72, 73, 74, or 6000 Series PPU assembly it is limited to three characters maximum. In all cases, the last character of a subprogram or overlay name cannot be a colon.

Any other type of name can consist of one to eight characters. A name does not have a value or attributes and cannot be used in an expression.

The different types of names do not conflict with each other. For example, a micro can have the same name as a macro, or a subprogram can have the same name as a block, etc.

## **2.4 SYMBOLS**

A symbol is a set of characters that identifies a value and its associated attributes. For an ordinary symbol, the first character cannot be a \$ or = or: or a number; a symbol can be a maximum of eight characters. A symbol cannot include the following characters:

+ - \* / blank , [ ] or ^

Other special characters must be used with care, especially in ECHO and macro definitions (chapter 5). Conventions imposed on symbols by the operating system could restrict the use of certain characters in symbols.

An external or entry point symbol is used for linkage with other subprograms and has additional restrictions (see Linkage Symbols).

Concatenation marks or pairs of micro marks are removed before a symbol is examined (see Statement Editing). In CPU assemblies, to avoid conflict with register designators, a symbol cannot normally be An, Bn, Xn, where n is a single digit from zero to seven nor can a symbol be A.x, B.x, or X.x, because x is assumed to be a data item by the assembler. However, symbols resembling register designators can be used if each use of the symbol is prefixed by =S or =X. Register designators are described further under CPU registers.

The process of associating a symbol with a value and attributes is known as symbol definition. This can occur in five major ways.

1. A symbol used in the location field of a symbolic machine instruction or certain pseudo instructions is defined as an address having the current value of the location counter (chapter 3) and having an attribute defined as follows:

Absolute for the absolute block

Common for labeled or blank common blocks (relocatable assemblies only)

Relocatable for local blocks other than absolute during pass one

Absolute for local blocks during pass two of an absolute assembly

2. A symbol used in the location field of definition pseudo instructions (see Symbol Definition, chapter 4) is defined as having the value and attributes derived from an expression in the variable subfield of the instruction. Certain of these pseudo instructions assign an attribute of redefinability to a symbol. Unless a symbol is redefinable, a second attempt to define it with a different value produces a duplicate definition fatal error flag.
3. An external symbol is defined outside the bounds of the current subprogram and is declared as external in the current subprogram or is defined in relation to a symbol declared as external. In either case it has the attribute of external. Unlike a systems symbol, the true value definition is not known to the current subprogram.
4. Definitions of systems symbols that take place outside of the current program can be carried over to the current program through the SST pseudo instruction. COMPASS uses the true definitions but assigns the additional attribute of systems symbol.
5. COMPASS defines a symbol by default if a reference to a symbol is preceded by =S and the symbol is not otherwise defined in the subprogram. This feature is further described under Default Symbols.

There is no restriction on the number of times that the symbol can be referred to in the subprogram.

Examples:

Legal Symbols

P  
R3  
PROGRAM

Illegal Symbols

5A	First character numeric
ABCDEFGHI	Exceeds eight characters
ABE+15	Contains plus sign
=.11	First character equal sign

### 2.4.1 LINKAGE SYMBOLS

A relocatable subprogram can be linked to other subprograms through linkage symbols. The two types of linkage symbols are external symbols and entry point symbols. An external or entry point symbol can be a maximum of seven characters, the first character must be a letter (A-Z), and the last character must not be a colon.

Any symbol declared as an entry point in a subprogram compiled or assembled independently of the current subprogram can be declared as an external symbol in the current subprogram. Any symbol declared as an entry point in the current subprogram can be declared as an external symbol in some other subprogram. The symbol has a zero value and an attribute of external. An external symbol can be declared either through the EXT pseudo instruction or through default (a reference to the symbol is preceded by =X or =Y; see Default Symbols).



An external symbol can be strong or weak. A strong external symbol reference causes the loader to try to find and load a subprogram having a matching entry point symbol. Failure of the loader to satisfy a strong external in this way is flagged as a non-fatal error by the loader. A weak external does not require the loader to search for a satisfying subprogram; however if one is loaded for some other reason, the loader associates the matching linkage symbols in the usual way. At the end of loading, the existence of unsatisfied weak external symbol references is not an error.

External symbols can be defined in the subprogram relative to any external symbol declared in an EXT pseudo instruction. This is possible through use of symbol definition instructions that assign the value and attributes of an expression to a symbol. If the value of the expression reduces to an external symbol + an integer, the location field symbol is defined as having an integer value and external attribute. Entry point symbols and external symbols are not qualified (see Qualified Symbols).

## 2.4.2 DEFAULT SYMBOLS

When a symbol reference is preceded by =S, =X, or =Y and the symbol is not defined in the subprogram, COMPASS defines the symbol or declares it as a strong or weak external symbol, respectively, at the end of assembly. The =X and =Y forms are defined by default in relocatable assemblies only.

- |          |  |
|----------|--|
| =Ssymbol | <p>If symbol is not defined, COMPASS assigns an address at the end of the zero block. All subsequent references to the symbol, whether preceded by =S or not, are to the location of the word. A default symbol cannot be used where a previously defined symbol is required.</p> <p>If the symbol is defined by a conventional method, COMPASS does not define it again but uses the programmer definition.</p>   |
| =Xsymbol | <p>This option permits a programmer to define his symbols in a subroutine or link to them in another subprogram. If the programmer defines the symbol, the assembler uses the programmed definition. If the programmer does not define the symbol, the assembler assumes that the symbol is a strong external as though declared in an EXT pseudo instruction. A symbol prefixed by =X must conform to the requirements for external symbols.</p>  |
| =Ysymbol | <p>This option permits a programmer to define symbols in a subroutine or to link to them in another subprogram that need not be loaded. If the programmer defines the symbol, the assembler uses the programmed definition. If the programmer does not define the symbol and if it is not referenced elsewhere with an =X or =S prefix, or declared in an EXT pseudo instruction, the assembler assumes that the symbol is a weak external. A symbol prefixed by =Y must conform to the requirements for external symbols.</p> |

The system does not define a default symbol and issues an error flag if a symbol is prefixed both by =S and =X, or is prefixed by =X or =Y, and is not defined conventionally in an absolute assembly. Default symbols are qualified by the qualifier in effect at the time of the =S reference.

## 2.4.3 PREVIOUSLY DEFINED SYMBOLS

Certain pseudo instructions require that a symbol in an expression be previously defined. This simply means that the symbol, before its use as an expression element, must be defined in a prior instruction.

## 2.4.4 UNDEFINED SYMBOLS

A reference to a symbol that is never defined (not even by default) causes a U error flag to be placed to the left of the instruction containing the erroneous reference.

## 2.4.5 QUALIFIED SYMBOLS

A symbol defined when a symbol qualifier is in effect during assembly (see QUAL pseudo instruction, chapter 4) can be referred to outside of the qualifier sequence in which it was defined through:

/qualifier/symbol

The feature permits the same symbol to be defined in different subroutines without conflict. An unqualified symbol is global and does not require a qualifier when it is referenced, unless a qualifier is in effect, and a symbol qualified by the same qualifier has been defined. In this case, the unqualified symbol can be referenced as // symbol.

The combination of qualifier and symbol permits a value to be identified by a unique 16-character identifier. Linkage symbols are not qualified.

## 2.5 CPU REGISTERS

Register designators symbolically represent the 24 CPU operating registers. The registers are described more fully in chapter 8. The designators are inherent to COMPASS and cannot be changed during assembly.

In a CPU assembly, symbols of the same form as register designators may be used if each occurrence of such a symbol is prefixed by =S, =X, or =Y (see Default Symbols). However, a warning message is issued when such symbols are defined. The prefix cannot be used in the location field of machine instructions and symbol defining, data generating, BSS pseudo instructions, in the variable field of ENTRY, EXT, and SST pseudo instructions.

<u>Register Type</u>	<u>Designator</u>
Address	An or A.n
Index	Bn or B.n
Operand	Xn or X.n

For the forms An, Bn, or Xn, n is a single digit from 0 to 7. Any other value for n, for example 8, causes An, Bn, or Xn to be interpreted as a symbol rather than a register designator.

For the forms A.n, B.n, X.n, n can be a symbol or an integer. If the value of n or the value of the symbol exceeds 7, the assembler truncates it to the least significant 3 bits and issues a warning flag.

Registers designated by A1 through A5 or A.1 through A.5 are used for addressing to obtain information from central memory. Registers designated by A6, A7, A.6, or A.7 are used for addressing to place information into central memory.

COMPASS does not recognize registers in PP assemblies; there, the designators are acceptable as ordinary symbols.

**Examples:**

- A1           Designates address register 1
- A10          Interpreted as a symbol, not a register
- A.1          Designates address register 1
- A.NUM        If the value of NUM is 6, it designates address register 6
- A.10         Designates address register 2; however, it produces a warning flag because the two was derived from the truncation of 12, the octal value for 10.

The following produce equivalent results. A SET pseudo instruction (chapter 4) defines SUM and SUB as absolute values 3 and 2, respectively. A reference to a SET-defined symbol produces the same result as if the value had been used directly. In this example, the address of ALPHA is 001000.

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
6032001000	11	SB3	A2+ALPHA	30

	LOCATION	OPERATION	VARIABLE	COMMENTS
3	SUM	SET	3	
2	SUB	SET	2	
6032001000	SB+SUM	A.SUB+ALPHA		

## 2.6 SPECIAL ELEMENTS

The following designators are reserved for use as reference to special elements and cannot be used as symbols. The use of a special element in an expression causes the assembler to replace it with a value specified by the element in the expression. The control counters are discussed further in chapter 3.

<u>Designator</u>	<u>Significance</u>
* or *L	The assembler uses the value of the location counter for the block in use. The element is relocatable unless the counter in use is for the absolute block.
*O	The assembler uses the value of the origin counter for the block in use. The element is relocatable unless the counter in use is for the absolute block.
\$	The assembler uses one less than the absolute value of the position counter for the block in use.
*P	The assembler uses the absolute value of the position counter for the block in use.
*F	The assembler uses an absolute value obtained as follows: <ul style="list-style-type: none"> <li>0 COMPASS was called by a COMPASS control statement</li> <li>1 COMPASS was called by the RUN compiler (no longer supported)</li> <li>2 COMPASS was called by the FTN4 compiler</li> <li>3 COMPASS was called by the FTN5 compiler</li> </ul>

\*F can be redefined by the COMPASS control statement F parameter (chapter 10).

These designators are inherent to COMPASS and cannot be altered by the programmer during an assembly.

Examples:

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	JD	*+1+B7	
	.		
	.		
	ZR	X3,*L-1	
	.		
	.		
	LOC	*0-RES+PPR	
	.		
	.		
	VFD	*P/	
	.		
	.		
	VFD	\$/0,1/1	
	.		
	.		
	IFEQ	*F,2	

## 2.7 DATA NOTATION

Data notation provides a means of entering values for calculation, increment counts, operand values, line counts, control counter values, text for printing out messages, characters for forming symbols, etc.

The two types of data notation are character and numeric. The assembler allows the user to introduce data in the program in three basic ways:

As a data item

As a constant in an expression

As a literal

### 2.7.1 DATA ITEMS

Character and numeric data items can be used in subfields of the DATA and LIT pseudo instructions or as specifications of field values on VFD pseudo instructions.

### 2.7.2 CONSTANTS

A data constant is an expression element consisting of a value represented in octal, decimal, hexadecimal, or character notation. It resembles a data item but is restricted by its use as an expression element in two ways:

1. The first character must be numeric, prohibiting the delimited type of character string (see Character Data Notation) and the preradix for numeric values.
2. The field size is determined by the destination field for an expression and can be a maximum of 60 bits thus prohibiting double precision floating point numbers.

### 2.7.3 LITERALS

A literal is a read-only constant. It is specified as a data item in a subfield of a LIT pseudo instruction or as an element in an expression.

The method of specifying a literal in an address expression is nearly identical to that for specifying a data item in a DATA or a LIT pseudo instruction. The primary difference is that the literal is prefixed with an equal sign, which indicates that a literal follows.

When a literal is used as an element in an expression, the expression is evaluated using the address of the literal in the literals block rather than the value of the data item. Thus, the literal is considered relocatable. (For a discussion of the literals block, see chapter 3.)

Conventionally, if a literal is used, it is the only element in an expression.

The first use of a literal causes the assembler to assemble the data specified by the literal, and store the data in the literals block using as many words as are required to hold the data. If the binary pattern of the prefixed type of literal or of all the literals in a LIT declared sequence matches the binary pattern of words previously entered in the literals block, an entry is not generated for the data. This process eliminates duplication of read-only data.

The LIT pseudo instruction permits symbols to be associated with literals block entries. Such entries can be referenced symbolically or through use of a prefixed literal. However, to preserve the integrity of the literals block, they should be used as read only locations.

The assembly listing includes a list of the literals block when the D list option is selected (see Listing Control, chapter 4).

#### Example:

In the following example, using CPU instructions, the first statement creates a word in the literals block having the value 00000000000000000001. The address of that entry (for the purpose of the example) is 5555 and is used in the address field of the two statements at address 100 and the statement at the lower part of 101.

The literal in the second statement specifies a right justified character, A, which has a display code value of 1. The SB4 creates a one-word literal block entry having the value 00000000000000000002. The address of that entry is in the address field of statements at the upper half of addresses 101 and 102. In this example, the LIT sequence duplicates a sequence of entries in the literals block and does not cause new entries to be assembled.

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
100	6120005555 +		SB2	=1	
	6130005555 +		SB3	=1RA	
101	6140005556 +		SB4	=1RD	
	5555	L	LIT	1,2	
	6120005555 +		SB2	L	
102	6130005556 +		SB3	L+1	

CONTENT OF LITERALS BLOCK.

```
005555 00000000000000000001      A
005556 00000000000000000002      B
```

Continuing the previous example, a LIT sequence as illustrated below, does not duplicate a sequence in the literals block and causes entries to be generated in the literals block:

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
	5557		LIT	1,3,1RD,2	

CONTENT OF LITERALS BLOCK.

```
005555 0000000000000000000001      A
005556 0000000000000000000002      B
005557 0000000000000000000001      A
005560 0000000000000000000003      C
005561 0000000000000000000004      D
005562 0000000000000000000002      B
```

However, if the literals sequence in the first part of the example had been followed by a LIT that duplicates, in part, the most recent entries in the literals block, only the unduplicated part is added to the block. Thus, if the following LIT sequence had been used in place of the LIT 1,3,1RD,2, the first two words of the sequence would match the last two words of the literals block so that only two additional words would be required to complete the sequence.

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
	5555		LIT	1,2,3,4	

CONTENT OF LITERALS BLOCK.

```
005555 0000000000000000000001      A
005556 0000000000000000000002      B
005557 0000000000000000000003      C
005560 0000000000000000000004      D
```

2.7.4 CHARACTER DATA NOTATION

Character data strings are converted to the code in use at the time the string is evaluated (see CODE pseudo instruction, chapter 4), and placed in a field indicated by the data type (data item, constant, or literal). When no CODE instruction has been issued, conversion is to display code representation.

**Format:**

		<u>Example</u>											
<u>Data Item</u>	<table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px;">sign</td> <td style="padding: 2px;">n</td> <td style="padding: 2px;">type</td> <td style="padding: 2px;">string</td> </tr> </table> or <table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px;">sign</td> <td style="padding: 2px;">type</td> <td style="padding: 2px;">d</td> <td style="padding: 2px;">string</td> <td style="padding: 2px;">d</td> </tr> </table>	sign	n	type	string	sign	type	d	string	d	-3RABC		
sign	n	type	string										
sign	type	d	string	d									
<u>Constant</u> †	<table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px;">n</td> <td style="padding: 2px;">type</td> <td style="padding: 2px;">string</td> </tr> </table>	n	type	string	3RABC								
n	type	string											
<u>Literal</u> †	<table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px;">=</td> <td style="padding: 2px;">sign</td> <td style="padding: 2px;">n</td> <td style="padding: 2px;">type</td> <td style="padding: 2px;">string</td> </tr> </table> or <table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px;">=</td> <td style="padding: 2px;">sign</td> <td style="padding: 2px;">type</td> <td style="padding: 2px;">d</td> <td style="padding: 2px;">string</td> <td style="padding: 2px;">d</td> </tr> </table>	=	sign	n	type	string	=	sign	type	d	string	d	=-3RABC  =-R*ABC*
=	sign	n	type	string									
=	sign	type	d	string	d								

- =** Applies to literals used as expression elements only; signifies that a literal follows.
- sign** Optional for data item or literal. A sign with a constant is interpreted as an element operator.
  - + or omitted** The value is positive
  - The complemented (negative) value is formed
- n** Signifies how the string is determined:
  - omitted** The string is delimited by d. n cannot be omitted for a constant.
  - 0** For data item or literal, the string consists of all characters following type to:
    - blank or ,
 For a constant, string consists of all characters following type to:
    - + - \* / blank , or ^
 The ^ (caret) is in the CDC character set. In the ASCII character set, use the & (ampersand).
- n** For a data item or literal, n is an integer count of the number of characters in the string not counting guaranteed zeros. It is limited only by statement size.
  - For a constant, n is an integer count of the number of characters in the string. It cannot exceed 1/6 of the number of bits in the field that will contain the expression. A truncation error is flagged for a right justified constant if the most significant bit exceeds the field. Truncated zeros do not cause an error in this case. A truncation error is flagged for a left justified constant if the least significant bit positions are truncated, even if they are zero.
  - The string consists of the n characters following type.
  - Regardless of base, COMPASS assumes that n is decimal.

---

†Expression element



type Character string justification. The characters formed by the data item or constant are right or left justified into the destination field as follows:

<u>Type</u>	<u>Significance</u>
C	Left justified with zero fill. For data item or literal, 12 zero bits are guaranteed at the end of the string even if another word must be allocated. For a constant, C is the same as L; the 12 zero bits are not guaranteed.
H	Left justified with blank fill
A	Right justified with blank fill
R	Right justified with zero fill
L	Left justified with zero fill
Z	Left justified with zero fill. For data item or literal, six zero bits are guaranteed at the end of the string even if another word must be allocated. For a constant, Z is the same as L; the six zero bits are not guaranteed.

d A delimiting character used only when n is omitted. The characters between the first occurrence of d and the second occurrence of d form the string. d can be any character other than  $\Gamma$  or  $\neq$ .

string Characters from one of the COMPASS character sets (appendix A), except for those characters that act as delimiters (see n and d), the concatenation character ( $\Gamma$ ), and pairs of micro marks ( $\neq$ ).

Concatenation marks and pairs of micro marks are removed by editing before a string is examined. A single micro mark can be used in a string.

An empty or omitted character string is defined under one of the following conditions:

n is 0 and type is immediately followed by a delimiter, for example, 0L.

n is omitted and the two delimiting characters are adjacent, for example, H+ +.

Omission of a string in a DATA pseudo instruction is legal and does not cause generation of a data word.

For a constant, an omission of the string is valid and has a zero value.

An omitted string in a LIT pseudo instruction is legal and does not cause generation of a literal for that item; however, the LIT must contain at least one non-empty data item.

An omitted string for a literal in an expression is not legal and produces an error.

It is not possible to generate empty strings using types C, Z, R, or A.

**Examples of character data:**

In these examples, characters are converted to display code representation; all lines of code generated by DATA are printed only if the D or G list option is selected.

Data Items

<u>Location</u>	<u>Code Generated</u>
144	05222217225511165520
145	04215500000000000000
146	55555555555555555555

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	DATA	L*ERROR IN PDQ *,L.,,10H	

<u>Location</u>	<u>Code Generated</u>
1100	1725
1101	2420
1102	2524

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	PPU		
	:		
	:		
	DATA	0LOUTPUT	

Constants

<u>Location</u>	<u>Code Generated</u>
4722	7130000047
4723	7140000060
	5110031117
4724	6260530000
	1117240155
4725	0155555531
	1725242025
4726	2400000001
	0700000000

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SX3	1R*	
TAG	SX4	1R.,+1	
	SA1	3RCIO	
	SB6	X0+1L\$	
	VFU	30/4HIUTA,6/1RA,24/0AX+1	
	VFD	42/0LOUTPUT,18/1	
	VFD	15/0LG,15/0L	

Note that the character constant in the expression in the second line consists of a decimal point (57 in display code) to which 01 is added before the value is stored. Similarly, in the third field of the first VFD, 1 is added to the display code representation of X right justified with blank fill (55555530) so that 55555531 is generated.

## Literals

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	100003765	1	11	18	30
	100003770				
2652	5110003772 +	TAG1	LIT	RA+--*/(A,6L)§=	.,,900,0L
	5120003774 +		LIT	20HLITERALS	
2653	5130003767 +		SA1	=9CTENCHARCTS	
			SA2	=4+LEFT JUSTIFY WITH PLANKS+	
			SA3	=9L0	

### CONTENT OF LITERALS BLOCK.

003765	000000000004546475051	+--*/(
003766	52535455565700000000	)§= ,.
003767	33000000000000000000	0
003770	14112405220114235555	LITERALS
003771	55555555555555555555	
003772	24051603100122032423	TENCHARCTS
003773	00000000000000000000	
003774	14050624551225232411	LEFT JUSTI
003775	06315527112410550214	FY WITH PL
003776	01161323555555555555	ANKS

The first LIT pseudo instruction generates three words in the literals block; the OL item is an empty string and does not produce an entry. The second LIT pseudo instruction generates one two-word entry. The expressions in the variable fields of the SA1, SA2, and SA3 instructions each consist of a literal element. The character strings in the SA1 and SA2 literals do not duplicate former literals block entries so COMPASS generates new entries. However, since SA3 references an existing entry, COMPASS places the address of the entry in the address field of the instruction.

## 2.7.5 NUMERIC DATA NOTATION

Numeric data can be specified in octal or decimal notation. The value is converted to an integer or a floating point value in single or double precision.

### Formats:

Data Item

sign	preradix	value	modifiers
------	----------	-------	-----------

Constant

value	modifiers
-------	-----------

Literal

=	sign	preradix	value	modifiers
---	------	----------	-------	-----------

=	Applies to literals only; signifies that a literal follows.
sign	Optional for data item or literal; a sign with a constant is interpreted as an element operator.
+ or omitted	The value is positive
-	The complemented (negative) value is formed
preradix	Optional for data items and literals; cannot be used for constants. The preradix indicates the notation used for the value.
omitted	Notation can be specified by a postradix modifier or can be assumed from the assembly base. See BASE pseudo instruction.
B or O	Octal notation
D	Decimal notation
value	<p>A series of octal or decimal digits optionally consisting of an integer, a decimal (or octal) point, and a fraction. An integer value (fixed point) does not contain a point. A floating point value (legal in CPU assemblies only) is noted by the occurrence of the point.</p> <p>An octal value can be a maximum of 20 significant digits (fixed point) or 32 significant digits (floating point). An octal value cannot include 8 or 9. A decimal value cannot exceed <math>1.15 \times 10^{18}</math> (fixed point) or <math>7.9 \times 10^{28}</math> (floating point, ignoring the decimal point). Extra significant digits cause erroneous results.</p> <p>If value is omitted, it is assumed to be zero.</p>
modifiers	Associated with the value are the following optional modifiers specified in any sequence. A specific type of modifier can be specified only once. A duplicate produces an error flag.
postradix	Indicates the notation used for the value. See preradix for legal values. An error is flagged if notation contains both a preradix and a postradix.
decimal exponent	<p>Defines a power of 10 scale factor</p> <p>E<sub>+</sub>n or En or E      Single precision</p> <p>EE<sub>+</sub>n or EE<sub>n</sub> or EE    Double precision</p> <p>When the sign is plus or is omitted, the exponent (n) is positive.</p> <p>When n is omitted, it is assumed to be 0. The value of n cannot exceed 32767 and is always assumed to be a decimal integer.</p> <p>A fixed point value can be single precision (one word) only but a CPU floating point value can be generated in double precision (two words).</p> <p>If EE is used with a fixed point value, the assembler produces a fixed point number in single precision.</p> <p>The effect of the exponent is to multiply the value by 10 decimal raised to the n power.</p>

binary scale

Defines a power of two scale factor and is specified as follows:

$S_{+n}$  or  $S_n$  or  $S$

When the sign is plus or is omitted, the scale factor ( $n$ ) is positive. When  $n$  is omitted, it is assumed to be 0. The value of  $n$  cannot exceed 32767 and is always assumed to be a decimal integer.

The effect of the binary scale is to multiply the value by 2 raised to the  $n$  power.

binary point position

Applies to floating point values only and is specified as follows:

$P_{+n}$  or  $P_n$  or  $P$

When the sign is + or omitted,  $n$  indicates the number of bit positions the point is to be shifted to the left of bit 0. When the sign is -,  $n$  indicates the number of bits the point is to be shifted to the right.

The effect of  $P$  is to align the value so that the binary point occurs to the right of the  $n$ th bit.

The exponent is adjusted to a value of  $-(+n)$

For example, a value with  $P-6$  will have a biased exponent of 2006<sub>8</sub>; a value with  $P10$  will have an exponent of 1765<sub>8</sub>.

If  $P$  is not specified for a floating point number or if  $n$  is omitted, the assembler generates a normalized floating point value. The  $P$  modifier permits generation of an unnormalized value.

If, as a result of  $P$ , the most significant bit of the value is shifted out of the coefficient part of the single or double precision number, the assembler generates an overflow or underflow error.

Although scale factors can exceed valid ranges, the ranges for numbers are restricted by the hardware.

Example:

The number 1.0E400S-1200 yields a number that is approximately  $5.8 \times 10^{38}$  and is in range of the floating point representation.

All calculations are performed in 144-bit precision. The values are rounded to 96 bits for double precision and to 48 bits for single precision floating point numbers and to 60 bits for integers.

The order in which the assembler acts on the modifiers, regardless of the sequence in which they are specified, is:

1. Decimal exponent (single or double)
2. Binary scaling
3. Binary point position (CPU assemblies only)

CPU Numeric Data Items

Location Code Generated

5000 77777777777777777742  
 5001 17235000000000000000  
 5002 16430000000000000000  
 5003 20100000000000000012  
 5004 17760000000000000002  
 5005 17154651767635544264  
 5006 17200314631463146314  
 5007 77777777777777777777  
 5008 00000000000000000000

LOCATION	OPERATION	VARIABLE	COMMENTS
POCL	DATA	-29	
NUM	DATA	1.CEF1	
	DATA	1.CE+1P0	
	DATA	3.2P1S-5E1	
	DATA	0.0151E+01	
	DATA	0.1P47,-E,DEES	

CPU Numeric Constants

Location Code Generated

5012  
 5112 20360  
 43760  
 7150400000  
 5001 +  
 555

LOCATION	OPERATION	VARIABLE	COMMENTS
ALPHA	EOU	POOL+1	
VAL	EOU	555B	
	BSSZ	100B	
	LX3	-14B	
	MX7	4B	
	SX5	1S17	

CPU Numeric Literals

Location Code Generated

5113 5150005151 +  
 5130005152 +  
 5153  
 5155  
 5156  
 5157

LOCATION	OPERATION	VARIABLE	COMMENTS
ABLE	SA5	=20046755000234000004B	
	SA3	=1.1	
	LIT	1.CEF1	
	LIT	0.1P47	
	LIT	-D19	
	LIT	0.0151E+01,-E,DEES	

CONTENT OF LITERALS BLOCK.

005151 20046755000234000004 PDA B1 D  
 005152 17204314631463146315 OPBL:L:L:M  
 005153 17235000000000000000 OS/  
 005154 16430000000000000000 NB  
 005155 17200314631463146314 OPCL:L:L:L  
 005156 77777777777777777754 : : : : : : : :  
 005157 17154651767635544264 OP-(772=7#  
 005160 77777777777777777777 : : : : : : : :  
 005161 00000000000000000000

Examples of numeric data (assume default radix is decimal):

**PP Data Items**

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
			PPU	.	
			.	.	
			.	.	
			.	.	
300	0005		DATA	5,-90,+813,148S1,248E-1	
301	7766				
302	0013				
303	0030				
304	0002				

**PP Constants**

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
305	0000		COM	0,+*1	
306	0011				
307	4443		COM	-3334	
			=	250	
		31	APC		
		101	NLM	0101	
310	7777		COM	7777	

**PP Literals**

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
311	2000 1103		LDC	=100	
313	2100 1104		ADC	=-1	
315	2000 1105		LDC	=7777	

**CONTENT OF LITERALS BLOCK.**

1103	0012	J
1104	7776	;;;;;;;;;;
1105	7777	;;

## 2.7.6 HEXADECIMAL DATA NOTATION

Numeric data can be specified in hexadecimal notation. The value is converted to an integer in single precision.

Formats:

<u>Data Item</u>	sign	0	preradix	value	modifiers	
<u>Constant</u>	0		preradix	value	modifiers	
<u>Literal</u>	=	sign	0	preradix	value	modifiers

- =** Applies to literals only; signifies that a literal follows.
- sign** Optional for data item or literal; a sign with a constant is interpreted as an element operator.
  - + or omitted Value is positive.
  - Complemented (negative) value is formed.
- 0** The zero is optional for data items and literals but must be present for constants, so the preradix will not be taken as the first character of a symbol.
- preradix** Must be present to indicate that a hexadecimal value follows. The preradix character is ≡ or # depending on the printer used.
- value** A series of hexadecimal digits. Each hexadecimal digit represents 4 bits and is either a decimal digit 0-9 or a letter A-F. The digits 0-9 represent values 0-9 and the letters A-F represent the decimal values 10-15.
 

The value may contain up to 26 significant hexadecimal digits. No radix point is permitted. If value is omitted, it is assumed to be zero.
- modifiers** The binary scale (S) modifier is optional and has the same form and meaning as for octal and decimal data (see Numeric Data Notation).
 

The binary point position (P) modifier is permitted but ignored, since it does not apply to integer values.

Examples of hexadecimal data:

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
0	000000000000004435274	1		18	30
1	77777777777777777777		DATA	≡123ABC,-≡,-0≡AAAAA,≡1234S12	
2	777777777777775252525				
3	000000000000110640000				
4	000000000000053012566	X	CON	0≡A01576	
5	71300000006 +	HEX	SX3	=-≡12345S2	
6	777777777777776671353	CONTENT OF LITERALS BLOCK.			
		;;;;;7AK\$			



## 2.8 EXPRESSIONS

Entries in subfields of most source statements are interpreted as expressions consisting of a combination of one or more terms. Each term consists of one or more elements joined by operators. A comma or a blank terminates the expression.

An expression element can be a:

Symbol	Register designator (CPU only)
Numeric or character constant	Literal
Special element	

Examples of elements:

ALPHA	A.7	3HABC
\$	X3	=10HOUTPUT
*P	77BS3	

A term can be a single element or two or more elements joined by the following element operators:

*	Multiplication
/	Division

An expression can be a single term or two or more terms joined by the following term operators:

+	Addition
-	Subtraction
^	Exclusive or

The exclusive or operator is printed as ^ (carat) in the CDC character set or as & (ampersand) in the ASCII character set.

Rules:

1. If the last element of a term is omitted, COMPASS provides an element of zero. For example, if ABLE is a symbol, ABLE\*+3 is interpreted as the value of ABLE times 0 plus 3.
2. Two successive elements are illegal. Note, however, that \*\* is legal because the first asterisk is interpreted as an element, the second asterisk is interpreted as an operator, and the blank is interpreted as a null element.
3. A term can contain one relocatable or external element only. Thus, \*\*ABLE, where ABLE is a relocatable address, is illegal because ABLE and \* are both relocatable.
4. The element to the left of a divisor must be absolute.
5. Division by zero results in zero with no error.
6. Two or more additive operators (+ or - or ^) in sequence are interpreted as having a term of zero value between them.
7. If an expression begins with an additive operator (+ or - or ^), COMPASS provides a term with zero value preceding the operator.
8. All arithmetic in expression is performed in integer mode, even if an element is a floating point constant such as 2.3. Results are restricted to 60 bits; that is, if a term or value exceeds 60 bits, the excess high-order bits are discarded without comment.

The operator that immediately precedes a register designator is the register operator, regardless of the placement of the designator in the expression. The register operator can be:

+ - \* or /

Examples of expressions:

ABLE	Single term
\$-29	Two terms: \$ and 29
1+=3.14159EE+6	Two terms: a constant and the address of a literal. COMPASS places the literal in the literal block and uses its address in the expression.
*+3	Two terms: value of the location counter and numeric constant 3.
ABLE*4-72/NUM	Two terms, each consisting of two elements: the value of ABLE times 4, and 72 divided by the value of NUM.
10B	Single term consisting of a numeric constant.
3+A6-NUM	The components of the expression are register A6 and 3-NUM.
1R= ^ 1R/	The character constants (= and /) are logically differenced.

## 2.8.1 TYPES OF EXPRESSIONS

Evaluation during assembly reduces an expression to:

An absolute value (absolute address or an integer value)

An external symbol + a 21-bit integer

+ relocatable value + a 21-bit integer

Register designators and one of the above (CPU assembly only)

Register designators (CPU assembly only)

### Absolute Expressions

An expression is absolute if its value is unaffected by program relocation. An expression can be absolute, even though it contains relocatable terms, under two conditions:

The expression contains an even number of relocatable elements.

The relocatable elements must cancel each other. That is, each relocatable element (or multiple thereof) in a block must be canceled by another element (or multiple thereof) in the same block. In other words, pairs of elements in the same block must have signs that oppose each other. The elements that form a pair need not be contiguous in the expression.

Examples of absolute expressions:

In the following examples, EASY and FOX are relocatable in the same block. MIKE is absolute. The control counters are for the block that contains EASY and FOX.

EASY-FOX+MIKE	EASY and FOX cancel each other.
FOX-*	FOX and the location counter cancel each other.
MIKE+16	The expression contains no relocatable elements.
*-EASY-FOX*2	EASY and the location counter cancel 2 times FOX.

Relocatable Expressions

An expression is relocatable if its value is affected by program relocation. A relocatable expression consists of a single relocatable term or, under the following two conditions, a combination of relocatable and absolute terms:

The expression does not contain an even number of relocatable elements

All the relocatable elements but one must be organized in pairs that cancel each other. That is, for all but one block, each relocatable element (or multiple thereof) in a block must be canceled by another element (or multiple thereof) in the same block. The elements that form a pair need not be contiguous in the expression.

The uncanceled relocatable element can have three kinds of relocation:

Positive program

Negative program

Positive common (Negative common relocation is not permitted by the loader.)

Examples of relocatable expressions:

In the following examples, EASY and FOX are relocatable in the same block. MIKE is absolute. LIMA is relocatable in a different block. The control counters are for the block that contains EASY and FOX.

LIMA+MIKE-16  
FOX-EASY+FOX  
3\*FOX-2\*EASY  
EASY-#+FOX  
FOX-100B/MIKE  
-MIKE\*2+LIMA  
=10HMESSAGE 33  
-\*0

The pairing of relocatable terms cancels the effect of relocation because both terms would be relocated by the same amount. The comparative value of the two terms remains the same regardless of program relocation.

## External Expressions

An expression is external if its value depends upon the value of a symbol defined outside of the current subprogram. Either an external expression consists of a single positive external term or, under the following conditions, an external expression may consist of an external term, relocatable terms, and absolute terms.

The expression contains an even number of relocatable terms.

The relocatable elements must cancel each other. That is, each relocatable element (or multiple thereof) in a block must be cancelled by another element (or multiple thereof) in the same block. In other words, pairs of elements in the same block must have signs that oppose each other. The elements that form a pair need not be contiguous in the expression.

Examples of external expressions:

In the following examples, XYZ and ABC are external symbols. EASY and FOX are in the same block. The control counters are for the block that contains LIMA. MIKE is absolute.

XYZ-**\***+FOX-EASY+LIMA                      The pairs **\*** and LIMA, and FOX and EASY cancel each other.

FOX-3**\***EASY+2**\***FOX+XYZ                      The relocatable elements all cancel.

ABC+100B+MIKE                                  MIKE and 100B are absolute; no relocatable elements.

XYZ+ABC    Illegal; both are external.

-ABC+**\***-LIMA                                  Illegal; ABC is negative.

XYZ+**\***0    Illegal; **\***0 is an unpaired relocatable element.

## Register Expressions

An expression is a register expression if, in a CPU assembly, it reduces to one or more register designators and an operand. The attributes of the operand can be that of an absolute, external, or relocatable expression. Use of register expressions is generally restricted to symbolic CPU machine instructions. If the register designator is the first element in the expression, the operator can be omitted and is assumed to be +.

Examples of register expressions:

In the following examples, XYZ is an external symbol and LIMA is a relocatable symbol.

X3+LIMA-10B	}	Produce identical results
LIMA+X3-10B		
-10B+LIMA+X3		

B1+XYZ

**\***+A.NUM

## Evaluatable Expressions

An evaluatable expression is an expression that does not contain any symbols as yet undefined. Certain pseudo instructions require that the expressions be evaluatable.

## 2.8.2 EVALUATION OF EXPRESSIONS

When evaluating an expression, COMPASS replaces each element with a 60-bit value. A character constant is first right or left adjusted in a field the size of the destination field and then extended to 60 bits. Signs are extended for 21-bit quantities, that is, for counters, addresses, and symbols. In division, the integral portion of the quotient is retained; any remainder is discarded. Thus,  $5/2*2$  results in 4.

COMPASS forms a term value by interpreting each element and operator from left to right until it reaches a + or - or ^ operator. It then notes whether or not the newly formed term contains a relocatable or external symbol or register designators. The value of the symbol is added, subtracted, or differenced from the cumulative sum of the absolute elements, relocatable elements, or external values. The assembler continues evaluating the expression until it is reduced to a symbol and/or a value. An error is flagged if the expression cannot be reduced. The expression value is truncated, if necessary, and placed in the destination field. If it is too large for the field, the system issues an error flag. The maximum field size for an expression is 60 bits.

The value of an external symbol is zero if the external symbol is defined outside of the subprogram. It is the value relative to the external used in defining the symbol if the external symbol was defined within the subprogram.

A zero value is used in place of a register designator.

For pass one evaluation, COMPASS uses the value of a relocatable symbol relative to the block in which the symbol was defined. For pass two evaluation, COMPASS uses a value relative to program or common block origin.

The field size for an expression depends upon the instruction and is determined as follows:

For a symbol definition pseudo instruction, the expression value (including character constants) is justified in a 21-bit field.

In a VFD pseudo instruction, the expression is placed in a field of the size specified.

For a CON pseudo instruction, the field size is one word (12 bits for PP assemblies, 60 bits for CPU assemblies).

In a symbolic machine instruction, values of expressions are placed in address fields (18 or 6 bits for CPU assemblies; 18, 12, or 6 bits for PP assemblies).

Some relocatable program loaders may give unexpected results if relocatable or external address values are assembled into the same field of the same word more than once, as a result of ORGing backward over the word, or by having more than one subprogram preset a common block. The ORG pseudo instruction (see Block Counter Control, chapter 4) can be used to avoid such problems.



---

This chapter is designed to give the programmer a better understanding of how a program is assembled, loaded, and executed. This discussion of program structure is at the machine executable level, the level at which code is loaded into memory and executed.

A COMPASS subprogram consists of statements beginning with an IDENT pseudo instruction and ending with an END pseudo instruction. The user can designate a subprogram to be a main program by specifying a transfer address in its END pseudo instruction.

The programmer can control the assembly of COMPASS source statements so that subprograms are divided into blocks of binary code. These blocks can be controlled during the loading process. The first section of the chapter presents subprogram block concepts and how the programmer and the assembler organize object code into blocks. Following this is a brief description of the counters used to control the blocks.

A subprogram loaded into central memory can be either absolute or relocatable. An absolute subprogram is loaded at the same fixed address every time; a relocatable subprogram can be loaded into different locations, according to the available central memory at load time. Sections 3.3 and 3.4 discuss the structure of absolute and relocatable programs, respectively, and show the differences in block usage for both types.

Limited available central memory occasionally requires the use of overlays and partial binary sections in lengthy programs. Section 3.4 covers the use of these important programming tools.

### 3.1 SUBPROGRAM BLOCKS

A subprogram, whether assembled as absolute or relocatable, can be divided into subprogram areas called blocks. As assembly of a subprogram proceeds, the assembler or the programmer designates that object code be generated or that storage be reserved in specific blocks. By properly assigning code sequences, data, or reserved storage areas to blocks through use of ORG or ORGC, USE or USELCM, a programmer can intersperse instructions for the different blocks. The assembler assigns locations in a block consecutively as it encounters instructions destined for the block. A symbol defined within a block is not local to the block. That is, it is global and can be referred to from any other block in the subprogram. To render a symbol local to a sequence of code requires use of the QUAL pseudo instruction (section 4.4.3).

Blocks established between two IDENT instructions, or between an IDENT and END, form a group of blocks. COMPASS recognizes a maximum of 255 blocks in a single block group, 252 of which can be user-established. When COMPASS interprets an IDENT or END pseudo instruction, it begins pass two processing of the completed block group.

In pass two all symbols are assigned absolute values, the table of block names is cleared, the list of USE, USELCM, ORG, and ORGC instructions is cleared, and block structuring restarts. For END, the symbol table is cleared before the next subprogram is assembled. If the group does not contain a USE instruction or if object code is generated (or storage reserved) before the first USE instruction, COMPASS places the code in the nominal block (identified as PROGRAM\* on the listing). For an absolute program, the nominal block is the absolute block. For a relocatable program, the nominal block is the zero block. The user controls use of the nominal block and any user-established blocks through USE, USELCM, ORG, and ORGC pseudo instructions (section 4.5). Each occurrence of a non-redundant literal constant causes an entry in the literals block; otherwise, the user has no control of this block.

### 3.1.1 ABSOLUTE BLOCK

The absolute block is the nominal block for an absolute assembly. It is identified by the name PROGRAM\* on the listing. All code generated in the block is absolute. Each address symbol is defined during pass one as an absolute value relative to zero which is block origin. The code generated must be loaded and executed at the origin specified as the absolute block origin.

Normally, a relocatable assembly does not contain an absolute block. It may have one established, however, if the programmer issues an ORG (or ORGC) request using an absolute value. The assembler generates text tables specifying absolute block relocation. The loader loads the absolute text when it encounters the text table, without manipulating any addresses. For a relocatable assembly, an absolute block is identified on the assembly listing by the name ABSOLUTE\*. There is no ECS/LCM absolute block.

### 3.1.2 ZERO BLOCK

The zero block has the block name 0 and is the nominal CM/SCM block for a relocatable assembly. It is a local block; that is, it is not accessible to other subprograms. Upon completion of assembly, the assembler assigns any undefined default symbols at the end of the zero block. The zero block is identified by the name PROGRAM\* on the assembler listing.

An absolute program has a zero block only if the program contains default symbols. In an absolute assembly, the zero block immediately follows the absolute PROGRAM\* block. The zero block is also named PROGRAM\*.

There is no ECS/LCM zero block.

### 3.1.3 LITERALS BLOCK

COMPASS generates literal data entries in the literals block. It is local to a subprogram. The literals block is identified by the name LITERALS\* on the assembly listing. COMPASS always assigns storage to the literals block immediately following the zero block. There is no ECS/LCM literals block.

### 3.1.4 USER-ESTABLISHED LOCAL BLOCKS

By using USE and USELCM statements, a programmer can establish local blocks in addition to those previously described for an absolute or relocatable subprogram. At the end of assembly, COMPASS assigns an origin relative to the nominal block to each user-established local block, in the sequence in which they are established.

All of the CM/SCM local blocks are concatenated to form a single block, which is treated by the loader as a CM/SCM block whose name is unique to the subprogram. Similarly, all of the ECS/LCM local blocks are concatenated to form a single block which is treated by the loader as an ECS/LCM block whose name is unique to the subprogram. (SCOPE 2 does not currently allow LCM local blocks.)

The length of each ECS/LCM block, including the combined local block, is rounded up, if necessary, to an integral multiple of eight 60-bit words. The maximum size of an ECS/LCM block is 1,048,568 words.

### 3.1.5 LABELED COMMON BLOCKS

A labeled common block is a storage area that can be preset with data accessible to one or more relocatable subprograms. These blocks are designated during assembly as being in CM/SCM or ECS/LCM through the USE and USELCM pseudo instructions respectively, where the name of the block is the name enclosed by slashes; that is, /name/. The tables are designed so that the loader can allocate space in memory for the first subprogram that is loaded that declares the block. Thus, the first subprogram that names a block sets the maximum size of the block. Each subprogram, as it is loaded, can link to allocated blocks or can cause new blocks to be allocated. The contents of a labeled common block can be generated by any of the subprograms having access to it.



If an absolute subprogram attempts to establish a labeled common block by using a `USE /name/` or `USELCM /name/` pseudo instruction, COMPASS treats the block as a local block having the slash-enclosed name.

### 3.1.6 BLANK COMMON BLOCKS

A blank common block is a storage area that cannot be preset with data. That is, the loader does not load information into the area before the program is executed.

For a relocatable program, the CM/SCM and ECS/LCM blank common blocks are allocated space by the loader after all subprograms are loaded, according to the largest block area declared by any of the subprograms. A CM/SCM blank common block is established through use of the `USE` pseudo instruction (chapter 4). An ECS/LCM blank common block is established through use of the `USELCM` pseudo instruction (chapter 4). A blank common block has no name. A `USE //` indicates blank common in CM/SCM; A `USELCM //` indicates blank common in ECS/LCM.

If no relocatable program declares a blank common block, there is none. If an absolute program contains a `USE //` or `USELCM //` pseudo instruction, COMPASS treats the block as a local block named `//` and data can be stored in this block.

The `USELCM` pseudo instruction can occur only in CPU programs.

### 3.1.7 REDUNDANT BLOCK NAMES

A CPU subprogram may have two blocks with the same name and the same memory type if they have different block types (local or common). Furthermore, a CPU subprogram may have two blocks with the same name and the same block type if they have different memory types (CM/SCM or ECS/LCM). Thus, altogether, there may be up to four different blocks with the same name.

## 3.2 BLOCK CONTROL COUNTERS

For each block used in a subprogram, COMPASS maintains three counters: an origin counter, a location counter, and a position counter. When a block is first established or its use is resumed, COMPASS uses the counters for that block. During pass one, the origin and location counters are initially zero. During pass two, as the assembler constructs the program, it assigns an initial value to each local block origin counter and location counter. Thus, expressions containing relocatable symbols are not necessarily evaluated the same in pass one and pass two.

### 3.2.1 ORIGIN COUNTER

The origin counter controls the relative location of the next word to be assembled or reserved in the block. It is possible to reserve blank storage areas simply by using either the `ORG`, `ORGC`, or `BSS` pseudo instructions to advance the origin counter; `ORG` and `ORGC` also permit the programmer to reset the counter to some lower location in the block or to change blocks. `BSS` allows the programmer to decrement the counter but not to change blocks. The origin counter is incremented by one for each word assembled or skipped forward. The origin counter is decremented by one for each word skipped in the reverse direction.

When the special element `*O` is used in an expression, the assembler replaces it by the current value of the origin counter for the block in use.

### 3.2.2 LOCATION COUNTER

The location counter is normally the same value as the origin counter and is used by the assembler for defining symbolic addresses within the block. The counter is incremented whenever the origin counter is incremented. It is possible through the LOC pseudo instruction to adjust the location counter so that it differs from the origin counter. This may be desirable when the code being assembled is to be loaded at one location and subsequently moved and executed at another location. In this case, the programmer resets the location counter to reflect the actual location at which execution is to occur. As another example of its use, the programmer assembling a large table may reset the location counter to zero so that on the listing, the addresses alongside each word of the table reflect the word's position in the table rather than in the block. Note that use of this technique does not alter the placement of code in the block. (For an example of these applications, see the LOC pseudo instruction in chapter 4) When either of the special elements \* or \*L is used in an expression, the assembler replaces it by the current value of the location counter for the block in use.

### 3.2.3 POSITION COUNTER

Assume that bits are numbered 59 through 00, from left to right within a 60-bit CPU word and numbered 11 through 00 within a 12-bit PPU word. Then, the position counter is initially 60 or 12, respectively, and indicates the number of bits remaining in the word. The position counter, which is decremented by one for each completed bit of an assembled word, becomes 00 when the word is completed, and is reset to 60 or 12 when a new operation is started.

For a CPU assembly, the 15-bit and 30-bit CPU instructions cause the position counter to normally have values of 60, 45, 30, and 15 reflecting the placement in the word for the next instruction or data value to be generated. For a PPU assembly, the normal value is 12.

The normal pattern of advancement for the position counter can be altered through use of the VFD and POS pseudo instructions.

When the special element \*P is used in an expression, the assembler replaces it with the current value of the position counter.

When the special element \$ is used in an expression, the assembler replaces it with the current value minus one of the position counter for the block in use; that is, it returns the next available bit position.

### 3.2.4 FORCING UPPER

In a CPU assembly, if any of the following conditions is true, the assembler packs parcels remaining in a partially completed word with no-operation instructions (section 8.1), sets the position counter to 60, and increments the origin and location counters before it assembles code for the next instruction:

- Insufficient room remains in a partially filled word for the next instruction or data to be generated.

- The current statement is a machine instruction, or a VFD pseudo instruction, with a location symbol or + in the location field.

- The current statement is an RE, WE, PS, XJ, CC, CU, DM, or IM (or RL or WL on NOS and NOS/BE) instruction for a CYBER 180 Series, a CYBER 170 Series, CYBER 70/Model 71, 72, 73, 74, or 6000 Series. (The programmer can negate this force upper by placing a minus sign in the location field of the instruction.)

- The current statement is an END, BSS, BSSZ, DATA, DIS, CON, SEGMENT, SEG, IDENT, ORGC, LOC, ORG, or MD pseudo instruction.

The assembler forces upper after it assembles code for one of the following:

JP  
RJ  
Unconditional EQ  
Unconditional ZR  
ES (CYBER 70 Model 76 or 7600)  
MJ (CYBER 70 Model 76 or 7600)  
PS (CYBER 180 Series, CYBER 170 Series, CYBER 70 Model 71, 72, 73, 74, or 6000 Series)  
XJ (CYBER 180 Series, CYBER 170 Series, CYBER 70 Model 71, 72, 73, 74, or 6000 Series)  
IM (CYBER 70 Model 72 and 73)

This post force upper does not occur immediately, but is deferred until the assembler encounters the next machine instruction or data generating, storage allocating, or binary control pseudo instruction in the same USE block. The programmer can negate the force upper following the instruction by placing a minus sign in the location field of the next instruction. Thus, pseudo instructions following one of the above machine instructions and referencing the origin, location, or position counter will use the value before the force upper.

In a PPU assembly, no forcing upper occurs; the assembler ignores a + in the location field on any instruction other than a VFD. A plus or minus in the location field of a VFD in PPU assemblies forces the VFD data to begin at the next full word.

### 3.3 RELOCATABLE PROGRAM STRUCTURE

A CPU relocatable program consists of one or more subprograms that can be assembled separately, either in the same job run or in independent runs. The subprograms can all be written in COMPASS source language, or can be written in any other source language available in the product set of the operating system as long as the compiler or assembler produces relocatable binary output in a form acceptable to the loader. A COMPASS language subprogram is composed of instructions beginning with an IDENT pseudo instruction and ending with an END pseudo instruction. A subprogram can be either a main program or a subroutine, depending on how its END pseudo instruction has been written.

When a program is loaded into memory, its subprograms occupy contiguous blocks of words. The first word in the first block is known as the reference address (RA). The total number of words in the blocks is the job field length.

When a subprogram is relocated, each machine instruction in it that references a specific address must be adjusted. Because of this necessity, relocatable subprograms are assembled as though they begin at address zero; they are not assigned specific origins. In this way the loader can load subprograms independently, yet contiguously; their origins are relative to RA. Since all addresses within the subprogram are relative to the first word address of the subprogram, each address in the program effectively becomes a function of RA.

A nonblank IDENT pseudo instruction that does not specify a fixed load address indicates a relocatable subprogram. Upon completing assembly of a relocatable subprogram, COMPASS assigns each local block an origin relative to the zero block. Each block thus becomes an extension of the zero block (figure 3-1).

COMPASS also provides for subprogram linkage. Through pseudo instructions such as ENTRY, ENTRYC, and EXT, subprograms can transfer control to each other and access common storage locations.

The loader is thus able to load subprogram blocks independently, as required. Program execution is not affected by the relocation process.

The length of the subprogram given on the assembly listing is the sum of the final values of the origin counters for the local blocks, including the zero block and literals block, but not the absolute block. Any absolute text is simply inserted at the absolute location relative to RA.

COMPASS binary output for a relocatable subprogram consists of one section for each LCC pseudo instruction (if any) in the source program, followed by one section containing the subprogram loader tables.

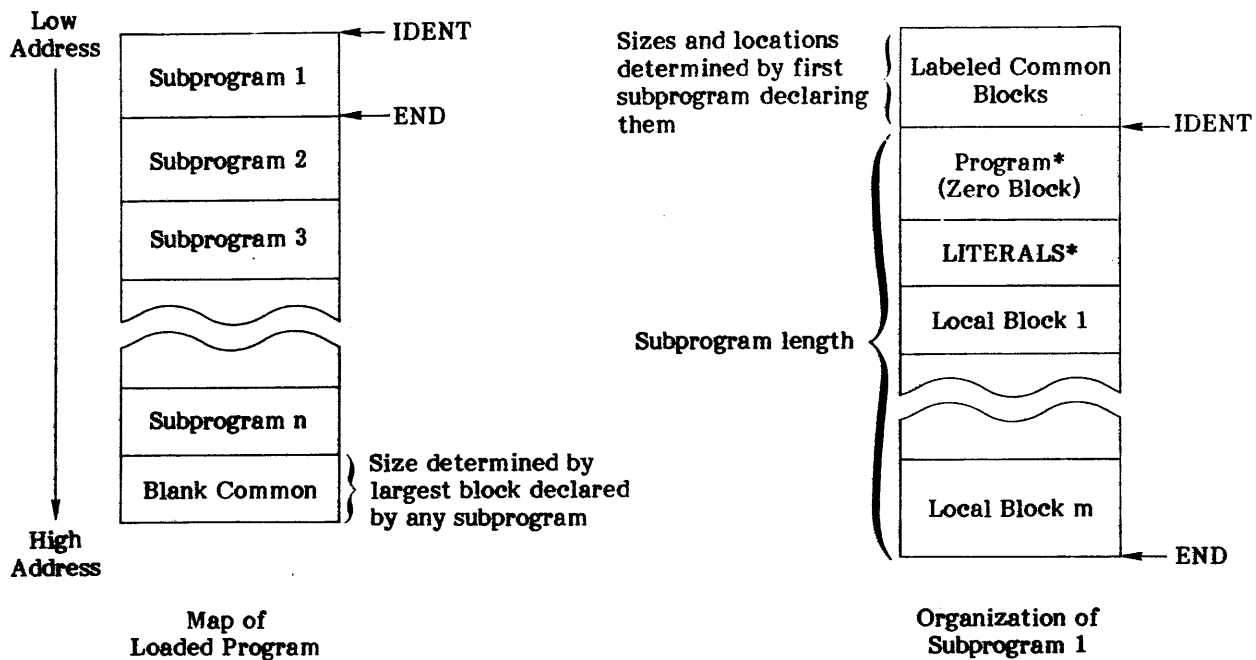


Figure 3-1. Relocatable Program Structure

### 3.4 ABSOLUTE PROGRAM STRUCTURE

An absolute program consists of code that is not relocatable and must be loaded at specific memory locations. Because the loader performs no address manipulation for absolute programs, absolute code can be loaded more rapidly than relocatable code.

A CPU program can be either relocatable or absolute. PPU programs are always absolute. PPU programs are parts of the operating system that reside in the peripheral processors; they are normally the concern of only system analysts. Any user can assemble PPU code, but cannot execute it without special system access privilege.

The programmer has the option of constructing an absolute program as a single unit, or of dividing it into overlays. Each overlay consists of data, information, or instructions that are needed at different times. Dividing a program into overlays allows several routines to occupy the same central memory storage consecutively so that total storage requirements for a program are reduced. For maximum program efficiency, the reduction of storage requirements must be weighed against an increase in execution delay while loading parts of the program.

During assembly of an absolute program or overlay, COMPASS creates a memory image of the absolute code. During pass two, it assigns each block an origin relative to the absolute block. Any relocatable symbol is reassigned an absolute address; each block effectively becomes an extension of the absolute block.

Figure 3-2 illustrates the structure of an absolute program that is not divided into overlays. The absolute block is the nominal block for the program (labeled PROGRAM\* on the listing). The use of default symbols and literals causes the generation of the zero block and the literals block, respectively. Local blocks A, B, and C follow the literals block. The transfer symbol in the END pseudo instruction indicates a main subprogram. In the binary load module the prefix (PRFX or 7700g) table and the header table precede the binary section that is the memory image of the program.

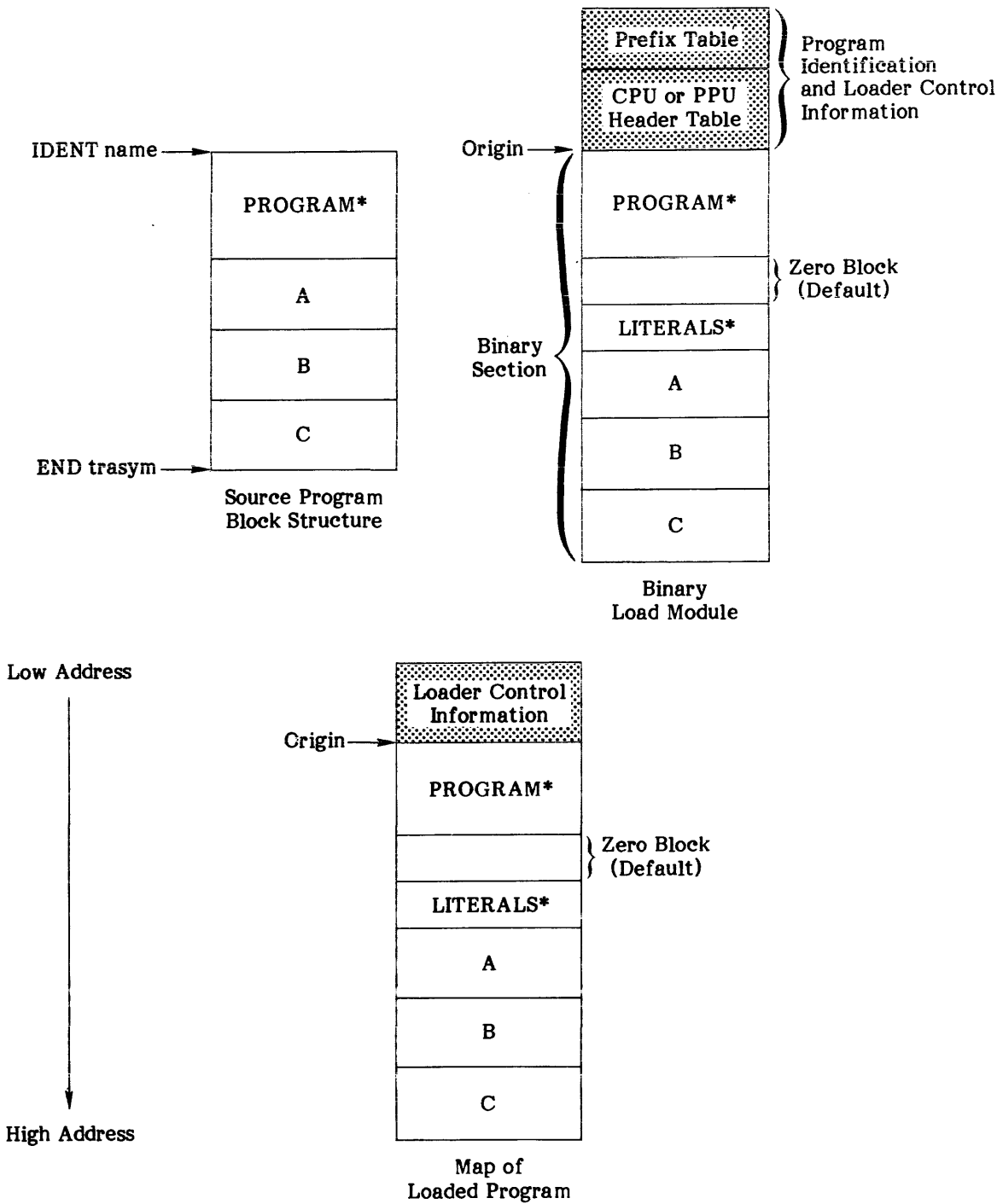


Figure 3-2. Absolute Program Structure

The binary output for the program consists of a section for each overlay. Note that the binary section for an absolute program that is not divided into overlays has the same format as the main overlay of a program divided into overlays. The user has the option of writing part of a binary section at a time by using either a SEG pseudo instruction or an IDENT (other than the first IDENT) with a blank variable field.

An absolute binary load module usually has three parts: a prefix (PRFX or 7700g) table, a header table, and the binary image of the program or overlay. A header table can be one of the following:

ASCM or 5000g.

EASCM or 5100g.

ACPM or 5300g.

EACPM or 5400g.

Tables are shown on a COMPASS listing by their octal numbers. The table formats are described in the Loader reference manual.

The amount of binary written as a result of the binary control instruction (IDENT, SEGMENT, SET, or END) is subject to whether or not an entire block group is written, as follows:

If a complete block group is being written (everything between an IDENT and an END or between two IDENT instructions), the memory image of the program or overlay ends with the maximum origin counter value for the last block established, that is, with the last word address.

If only a portion of the binary for the block group is being written, it consists of the memory image of the program or overlay ending with the value of the current origin counter.

END, SEGMENT, and a nonblank IDENT complete one overlay and write an end of section. SEGMENT and IDENT write header information for the overlay to follow.

### 3.4.1 ABSOLUTE OVERLAYS

When an absolute program contains more than the one IDENT<sup>†</sup> pseudo instruction or contains SEGMENT pseudo instructions, COMPASS does not prepare just one section of a memory image of the program as it is assembled, but, instead, generates a section for each overlay.

Dividing the program into overlays permits memory to be sequentially overlaid by different subroutines and data during program execution, reducing the maximum memory requirements for the program.

Three levels of overlays can be generated for a CPU assembly: main, primary, and secondary. Each overlay is identified by a level number specified in the IDENT or SEGMENT pseudo instruction. The level number consists of an ordered pair of octal numbers, each of which can be 0 through 77g. The first number is known as the primary level number; the second is known as the secondary level number. The level number 0,0 signifies the main overlay (normally the portion of the program following the first IDENT). A primary overlay is indicated by a nonzero primary number paired with a zero secondary level number. For a secondary overlay both the primary and the secondary level numbers are nonzero.

Conventionally, the main overlay is loaded first and remains in central memory throughout execution. Only two other overlays can remain loaded concurrently: these are usually one primary overlay and one of its associated secondary overlays.

---

<sup>†</sup>IDENT instructions described in this section are assumed to have nonblank parameters. The special case of the blank IDENT is described in section 3.4.3.

The hierarchy of overlay association is depicted by figure 3-3. The primary overlay 1,0 has three associated secondary overlays numbered 1,1; 1,2; and 1,3. A primary overlay and all of its associated secondaries have the same primary level number. The next branch of overlays (indicated by level numbers 77,y) shows that the level numbers of the overlays are not required to be consecutive nor to be indicative of the order in which they were generated.

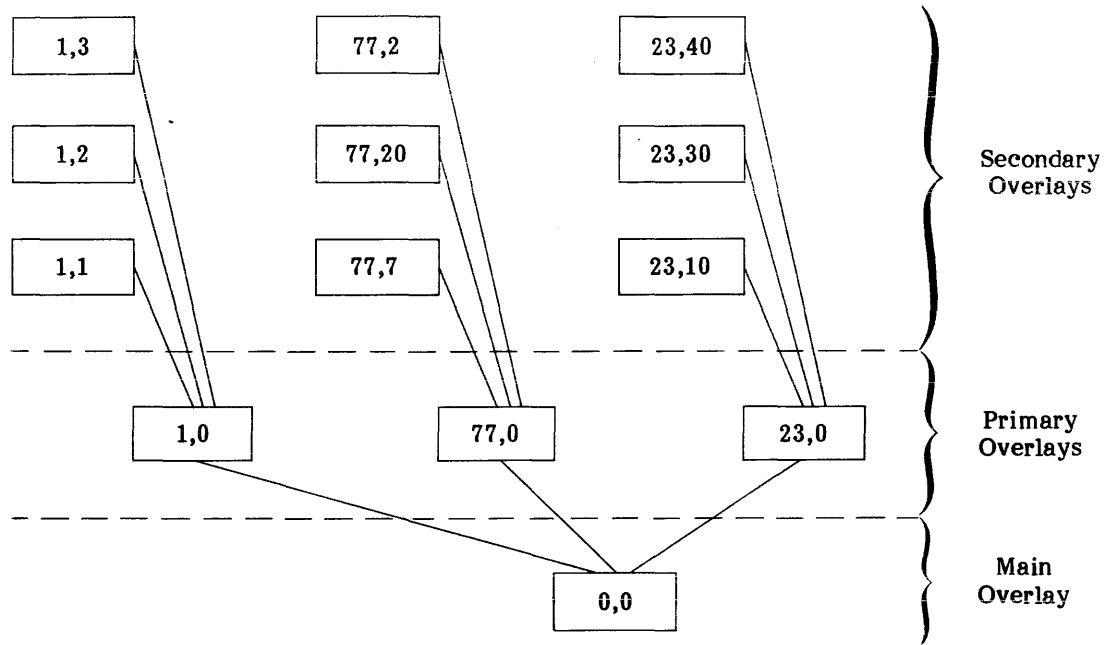


Figure 3-3. Overlay Hierarchy

The main overlay can call both primary and secondary overlays into main memory via the operating system loader. (For detailed information concerning loader calls, see the Loader reference manual.) Once a primary overlay is loaded, it can call any of its associated secondary overlays. Overlay 23,0, for example, can call overlays 23,10; 23,30; and 23,40 in any order.

The main overlay can have multiple entry points: execution can begin at any one of them. Usually, primary and secondary overlays have a single entry point which provides the transfer address. A secondary overlay can reference entry points in its primary and in the main overlay. A primary overlay can reference entry points in the main overlay. The programmer must ensure that the necessary entry points have not been overwritten.

These conventions concerning the numbering, hierarchy, loading, and execution of overlays are not enforced by COMPASS. Any overlay can call the operating system loader to load another overlay, and any overlay can reference addresses in any other overlay. However, overlays are not all in central memory during program execution and the sequence in which the overlays are loaded and executed is beyond the scope of the assembler; therefore, it is the user's responsibility to assure that an overlay does not refer to symbols, instructions, or data not concurrently in central memory.

Although PPU overlays are not identified by level numbers, they resemble CPU overlays in all other respects. However, a PPU overlay with assembled code in locations 7774g through 7777g may load incorrectly due to wraparound to location 0000.

Overlays generated by using IDENT pseudo instructions differ in certain respects from overlays generated by using SEGMENT instructions, as described below.

Binary formats for overlays are described in the Loader reference manual.

### IDENT-Type Overlays

An IDENT-type overlay consists of the portions of the program from:

One IDENT to (but not including) the next IDENT

The last IDENT in the overlay to the END

IDENT provides the programmer with the option of specifying the overlay level numbers with each overlay. The assignment of unique level numbers enables the loader to locate a specified overlay among overlays written on the same file. If the programmer does not specify level numbers for a CPU assembly, COMPASS assigns numbers 0,0 to the first overlay, and numbers 1,0 to all subsequent overlays.

The first IDENT causes COMPASS to generate the program or overlay identification information that precedes the absolute section. Upon encountering a second IDENT instruction before an END instruction, COMPASS generates output consisting of a memory image of the overlay, starting with the overlay origin specified on the previous IDENT and normally ending with the maximum origin counter value of the last block declared in the overlay; that is, the overlay normally ends with the last word address of its last block. An IDENT subsequent to a SEG or SEGMENT, however, generates binary that ends at the location specified by the current origin counter. Following the memory image, COMPASS writes an end-of-section (or end-of-record) and the overlay identification information specified by the new IDENT for the overlay to follow.

For an IDENT-type overlay, COMPASS completes all blocks, including the literals block. Block structuring starts fresh with each overlay. This means that each overlay can use the same block names used by other overlays, and each overlay can contain a literals block. The USE table and control counters are all reinitialized. The origin specified for an IDENT-type overlay can be any place in a previously generated overlay. This is possible because IDENT causes the assembler to assign an absolute address to each symbol in the symbol table. It can do this because the sizes of all the blocks are known.

Figure 3-4 illustrates a CPU program in which a second IDENT is used prior to an END pseudo instruction to generate a main overlay and a primary overlay. Between the two IDENT instructions, block usage alternates between the absolute block (labeled PROGRAM\* on the listing) and block A, as depicted in the block structure diagram. Note that in the main overlay (the first section of binary generated, labeled MAIN), the assembler has concatenated the portions of each block. Concatenation also occurs in the primary overlay, OV1, for the portions of the absolute block ABSOLUTE' and for those of blocks A', B, and C.

The occurrence of literals and default symbols causes the assembler to generate a zero block and a literals block, respectively, in both of these overlays. Following the second nonblank IDENT, the program overlay origin is set back into block A, as shown in the map of the two loaded overlays. Note that the loader control table is loaded in memory below the address specified in the ORG pseudo instruction (BETA, in the figure), as shown in the map of the loaded overlays.

The first IDENT pseudo instruction assigns the level number 0,0 to the first overlay (MAIN). COMPASS assigns level number 1,0 to overlay OV1 by default.

### SEGMENT-Type Overlays

A SEGMENT-type overlay consists of the portions of a program from:

The IDENT that identifies the program to a SEGMENT pseudo instruction

One SEGMENT to the next SEGMENT

The last SEGMENT to the END pseudo instruction



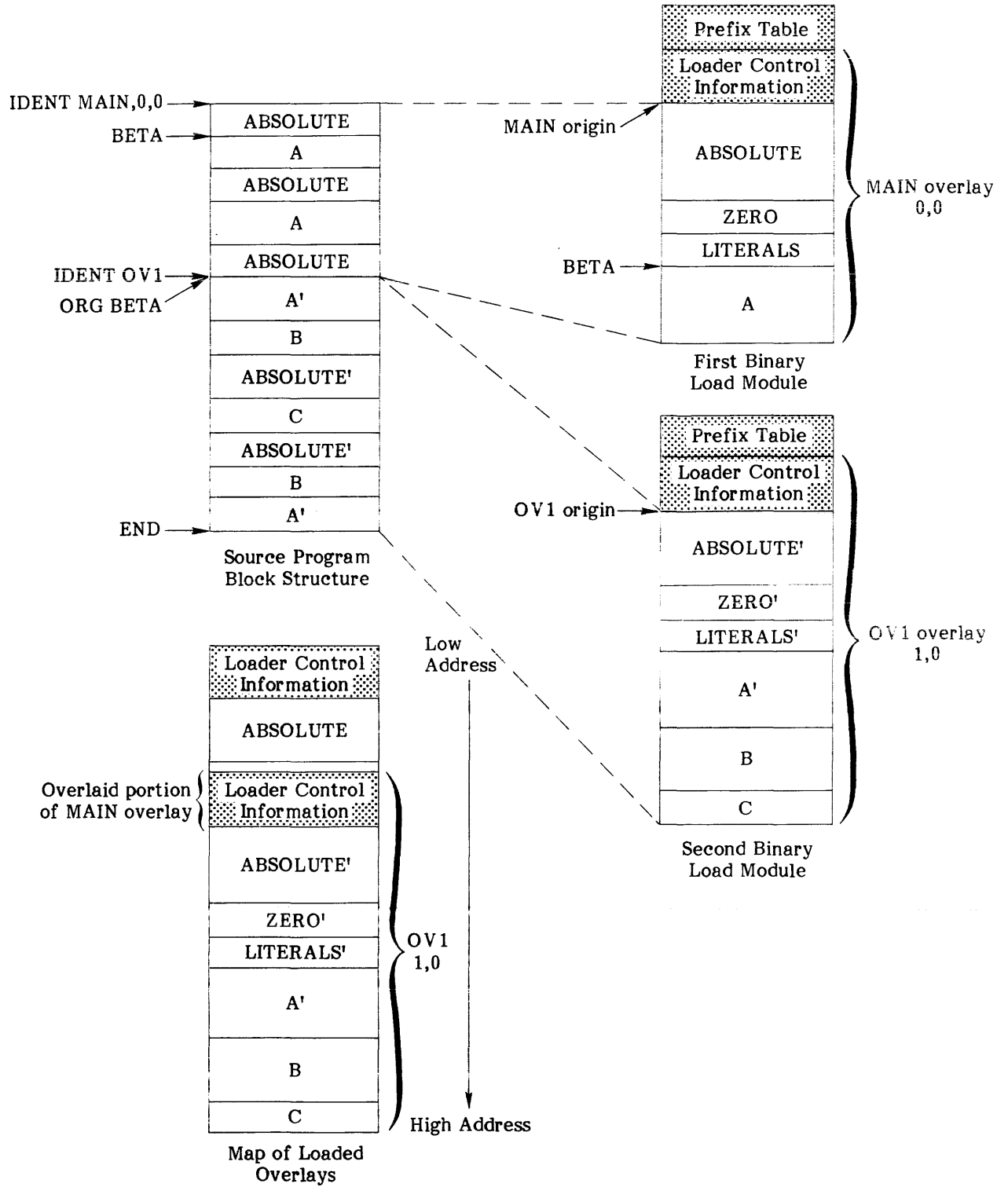


Figure 3-4. IDENT-Type Overlay Structure

SEGMENT provides the programmer with the option of specifying the overlay level numbers with each overlay. The assignment of unique level numbers enables the loader to locate a specified overlay among overlays written on the same file. If the programmer does not specify level numbers for a CPU assembly, COMPASS assigns numbers 0,0 to the first overlay, and numbers 1,0 to all subsequent overlays.

Upon encountering a SEGMENT instruction, COMPASS generates output consisting of a memory image of the overlay starting with the overlay origin specified on the previous SEGMENT (or IDENT, for the first overlay), and ending with the current origin counter value of the block in use at the time the SEGMENT was encountered. Following this, COMPASS writes an end-of-section and overlay identification information for the overlay to follow.

SEGMENT does not clear the symbol table or reinitialize the USE table. Thus, when a SEGMENT is encountered, the block in use is incomplete. It is the responsibility of the user to assure that all blocks other than the one in use are complete at that time. Also, the only symbols that can be used to define the origin of the new overlay are those valid for the block in use.

Each new SEGMENT-created overlay must use unique block names because blocks established in previous overlays cannot be resumed and because the block names remain in the USE table due to the incompleteness of the block group.

Figure 3-5 illustrates a program consisting of a main overlay, MAIN, and a primary, OV1. The use of default symbols causes generation of a zero block. The use of literals causes generation of a literals block. Both of these blocks occur in the overlay MAIN, because it contains the end of the absolute block. Block A begins in the main overlay, but is incomplete when COMPASS encounters the SEGMENT. The ORG pseudo instruction causes the origin of the primary overlay OV1, to be set at load time to TAG, at a lower address in block A. (Note that the loader control information is loaded at an address lower than the origin of the overlay.) OV1 establishes new blocks C and D.

### 3.4.2 MULTIPLE ENTRY POINT OVERLAYS

When a CPU program or overlay that calls an overlay is assembled independently of the overlay called, it may be desirable for the called overlay to identify more than one entry point. Thus, ENTRY pseudo instructions are permitted within an absolute assembly and cause the generation of a 5100g overlay table. This table consists of a control word and a list of overlay entry points. The calling program can examine the list and link to any of the entry points. The 5100g table occupies the area below the overlay origin and uses one more word than the number of entries in the table. For the format of the 5100g table, refer to the Loader reference manual.

### 3.4.3 PARTIAL BINARY

When a CPU absolute program or overlay contains SEG pseudo instructions or IDENT pseudo instructions for which the parameters are omitted (blank), COMPASS writes a partial binary section consisting of the binary generated since the previous IDENT, SEGMENT, or SEG instruction. However, it does not write an end-of-section (or end-of-record) or a new prefix table. A SEGMENT, nonblank IDENT, or END instruction completes the binary section.

#### SEG Partial Binary Record

By writing partial binary records using SEG, the programmer can reduce the assembler storage requirements. SEG does not write a complete block group. When the SEG is encountered, COMPASS writes binary beginning with the first block established in that portion of binary and ending with the final count specified by the origin count for the current block. A fatal error is issued if the user attempts to store data into a block not in the current partial binary record.

The portion of the binary that contains the end of the absolute block contains the literals block, if there is one. The symbol table and USE table are not reinitialized.

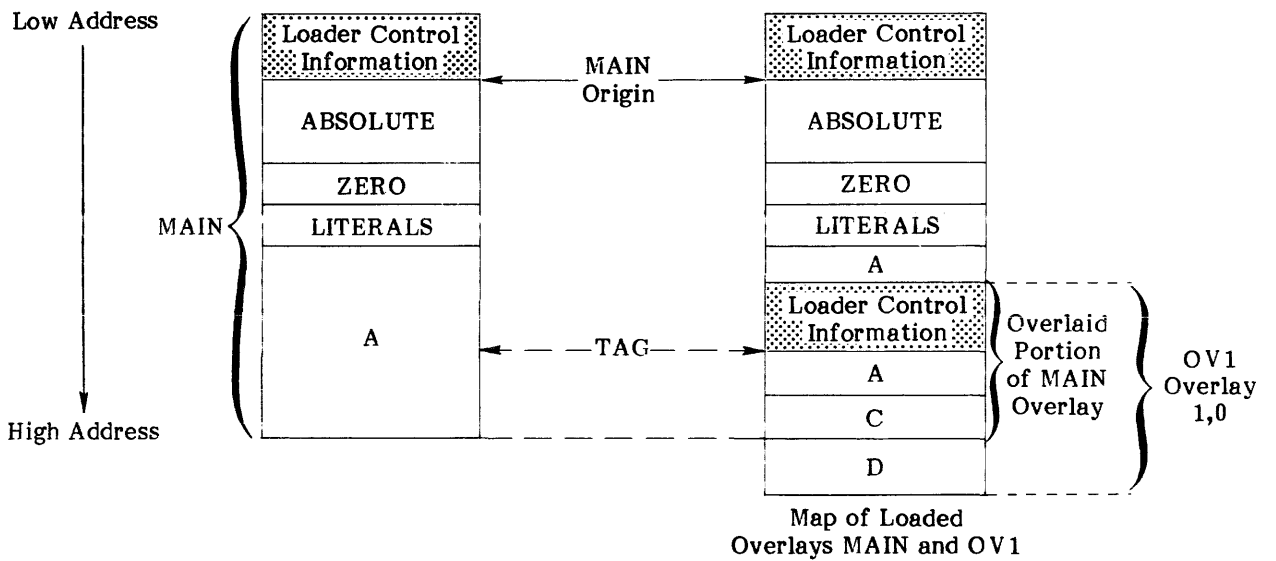
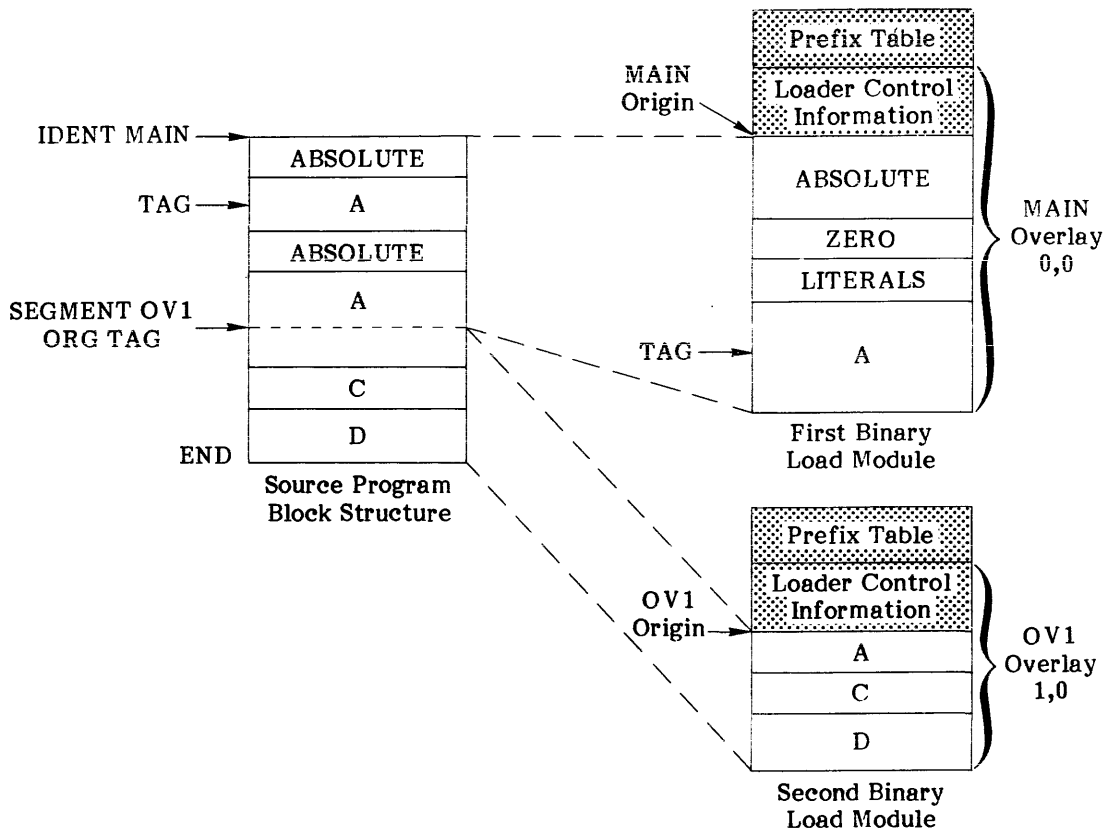


Figure 3-5. SEGMENT-Type Overlay Structure

Figure 3-6 illustrates how the binary for an absolute program can be written in three separate binary writes to reduce the amount of memory required to assemble the program. The resulting absolute section is loaded and executed as a single program or overlay.

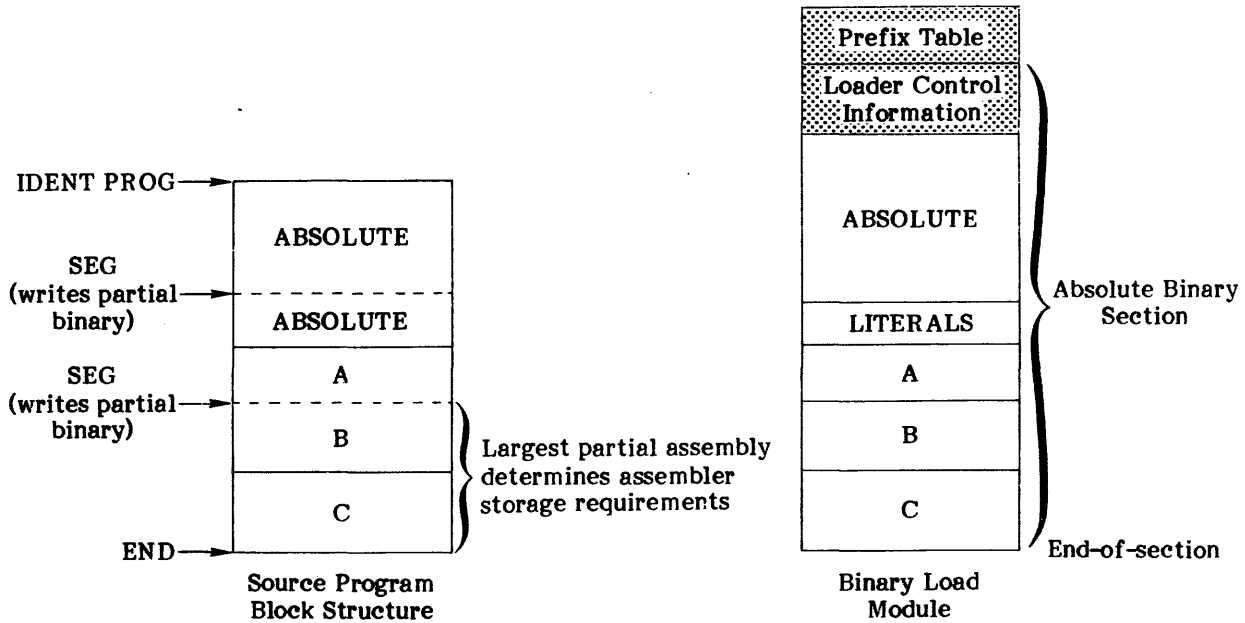


Figure 3-6. SEG Partial Binary

### IDENT Partial Binary

An IDENT with a blank variable field causes all binary accumulated since the previous IDENT, SEG, or SEGMENT to be written out without an end-of-section (or end-of-record) or a new 7700g prefix table. The USE table and the block counters are reinitialized. Each symbol in the symbol table is assigned an absolute address. The blocks in each partial binary section generated in this manner are allocated as if the partial binary section were a new subprogram with its own absolute block, literals block, and local blocks. This allows portions of a program to be self-contained units even though they are not overlays but are loaded as a single unit. The origin of an absolute block for new portion is the last word address plus one of the last block of the previous portion.

The core image written by a blank IDENT starts with the origin of the absolute block and normally ends with the maximum origin counter value of the last block declared in the block group; that is, it normally ends with the last word address. If part of the block group has already been written by a SEG or SEGMENT, however, the end of the binary is specified by the value of the origin counter for the current block.

COMPASS completes all blocks. The literals block is terminated. Block structuring starts fresh with each IDENT. Each new partial binary section created by a blank IDENT can use the same block names as are used by the other blank IDENT-created partial binary sections and non-blank IDENT-created overlays and each IDENT can contain a literals block but the blocks with the same names are independent of each other.

An attempt to write into or to reset the origin counter to a location in a partial binary section written separately causes an assembler range error.

Figure 3-7 illustrates how the binary for an overlay can be written in three discrete partial binary sections to reduce the amount of central memory required to assemble the program and divide the program into self-contained units. The resulting absolute section is loaded and executed as a single overlay.

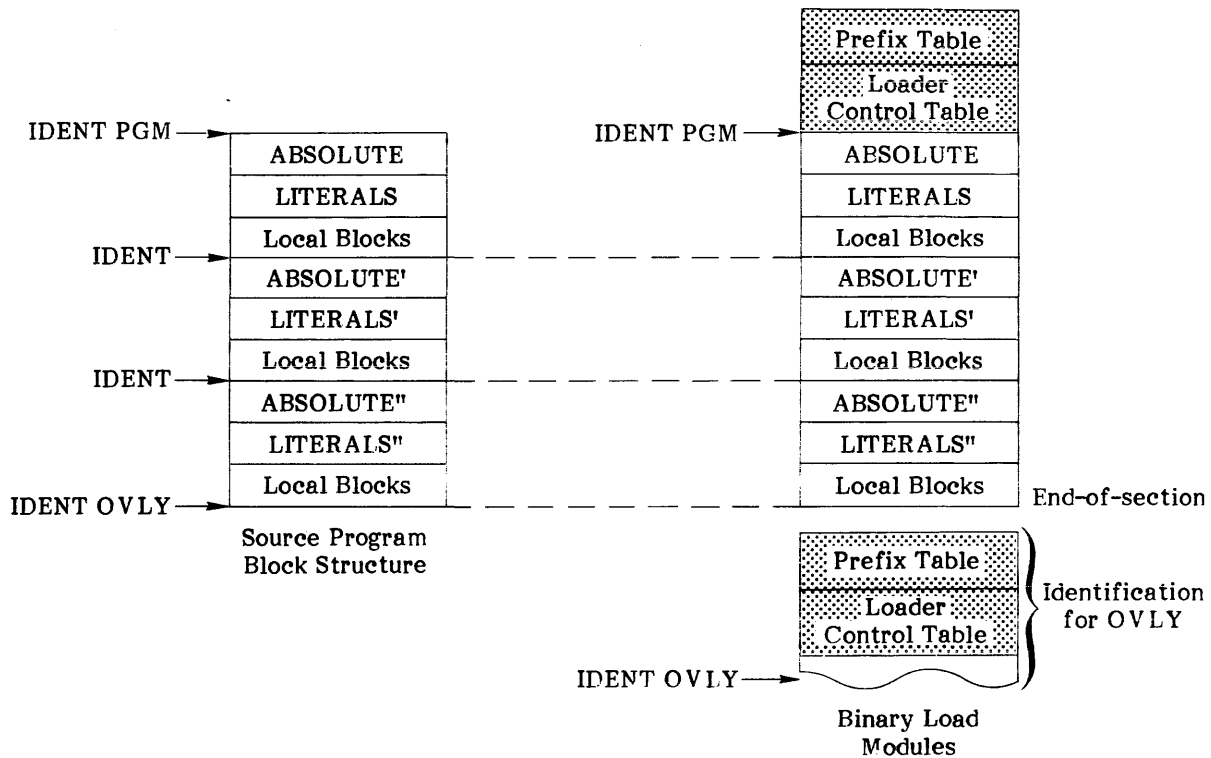


Figure 3-7. IDENT Partial Binary Records



### 4.1 INTRODUCTION TO PSEUDO INSTRUCTIONS

The format of the COMPASS pseudo instruction is the same as that of the symbolic machine instruction; it includes the location field, the operation field, the variable field, and the comments field. The pseudo instruction differs from the symbolic machine instruction in that it is used to control the actions of the assembler at assembly time, rather than those of the machine at execution time.

The pseudo instructions available in the COMPASS language are presented in this chapter and in chapters 5, 6, and 7. Programmers with little COMPASS experience should give special attention to a few important pseudo instructions, which are listed in the following table. It is not possible to write a COMPASS program without using some of them. The table indicates the type of assemblies in which the pseudo instructions can be used.

<u>Pseudo Instruction</u>	<u>Section</u>	<u>CPU Relocatable</u>	<u>CPU Absolute</u>	<u>PPU Absolute</u>
IDENT	4. 2. 1	X	X	X
ABS	4. 3. 1	-	X	-
PPU or PERIPH	4. 3. 3 or 4. 3. 4	-	-	X
ORG	4. 5. 3	X	X	X
ENTRY	4. 7. 1	X	-	-
BSS	4. 5. 4	X	X	X
CON	4. 8. 6	X	X	X
END	4. 2. 2	X	X	X

#### 4.1.1 TYPES OF PSEUDO INSTRUCTIONS

Pseudo instructions discussed in this chapter are classed according to application as follows:

Subprogram identification (IDENT and END)

Binary control (ABS, MACHINE, PERIPH, PPU, IDENT, SEGMENT, SEG, LCC, LDSET, STEXT, COMMENT, and NOLABEL)

Mode control (BASE, CHAR, CODE, COL, B1=1, B7=1, and QUAL)

Block counter control (USE, USELCM, ORG, ORGC, BSS, LOC, and POS)

Symbol definition (EQU and =, SET, MAX, MIN, MICCNT, and SST)

Subprogram linkage (ENTRY, ENTRYC, and EXT)

Data generation (BSSZ and blank operation code, DATA, DIS, LIT, VFD, CON, R=, REP, REPC, and REPI)

Assembly control (ELSE, ENDIF, IFtype, IFop, IF, IFC, IFPL, IFMI, and SKIP)

Error control (ERR and ERRxx)

Listing control (LIST, EJECT, SPACE, TITLE, TTL, NOREF, CTEXT, ENDX, and XREF)

Later chapters describe pseudo instructions that involve definition operations, alterations to the operation code table, and micros. In general, pseudo instructions can be summarized according to where they can be placed in a subprogram.

#### 4.1.2 REQUIRED PSEUDO INSTRUCTIONS

Two pseudo instructions, IDENT and END, are required for any assembly. IDENT must be the first source statement; END signals the termination of source statements for a subprogram.

#### 4.1.3 FIRST STATEMENT GROUP

Certain pseudo instructions establish basic characteristics of the assembly and provide the assembler with required information. These instructions make up the first statement group which must precede any symbol definition, storage allocation, or object code generation. The following instructions, if used, must be in the first statement group:

ABS  
MACHINE  
PERIPH  
PPU  
STEXT

#### 4.1.4 PERMISSIBLE ANYWHERE INSTRUCTIONS

The following pseudo instructions are permissible anywhere, including in the first statement group:

BASE	CPSYN	ENDM	MACROE	OPDEF	SKIP
B1=1	DECMIC	HERE	MICCNT	OPSYN	SPACE
B7=1	EJECT	IFC	MICRO	PPOP	SST
CHAR	ELSE	IRP	NIL	PURGDEF	TITLE
CODE	END	LDSET	NOLABEL	PURGMAC	TTL
COMMENT	ENDD	LIST	NOREF	QUAL	XREF
CPOP	ENDIF	MACRO	OCTMIC	RMT	

Comment lines and references to macro definitions are also permitted anywhere.

CPU or PPU symbolic machine instructions and all other pseudo instructions cannot be placed in the first statement group. The first use of one of these instructions terminates the first statement group.

## 4.2 SUBPROGRAM IDENTIFICATION

Subprogram identification pseudo instructions designate subprogram beginning and end. When two or more subprograms are assembled in a single COMPASS run called through the COMPASS control statement, the end of the source decks is indicated by an end-of-section, such as a 7/8/9 card.

### 4.2.1 IDENT — SUBPROGRAM IDENTIFICATION

An IDENT pseudo instruction of the following form is the first statement of a subprogram recognized by the assembler. Usually, any lines preceding the first IDENT or between an END and IDENT are assumed to be comments. However, when COMPASS has been called by some other language processor such as FORTRAN, the assembler returns control to the processor when the statement following END is not IDENT. For a relocatable subprogram, COMPASS flags any subsequent use of IDENT before END as an error. For an absolute subprogram, a second form of IDENT described under BINARY CONTROL is available for overlay generation.



The format of IDENT varies according to the type of assembly.

**CPU Relocatable Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name

**CPU Absolute Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name, origin, entry, $l_1, l_2$

**7600 PPU Absolute Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name, origin, entry, ppu

**6000 Series PPU Absolute Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name, origin

**name** Name of the subprogram or overlay. The parameter is required. For a CPU relocatable or absolute assembly, name can be 1 through 7 characters, of which the first must be alphabetic (A through Z) and the last must not be a colon.

For a CYBER 70/Model 76 or 7600 PPU assembly, name can be 1 through 7 characters. For CYBER 180 Series or CYBER 170 Series or CYBER 70/Model 72, 73, 74 or 6000 Series PPU assembly, name can be 1 through 3 characters. In either case, there is no restriction on the first character, but the last character must not be a colon.

**origin** An expression specifying the first word address of the absolute program or overlay. The overlay loader table and all code assembled starting at this address and ending with the next SEGMENT, nonblank IDENT, or END instruction make up the overlay. For a single entry point CPU program, the load address for the overlay is origin-1. The word at origin-1 is overlaid by the 5000<sub>g</sub> loader control table. For a multiple entry point CPU program, the load address for the absolute overlay is origin-wc-1, where wc is the number of entry points in the 5100<sub>g</sub> loader table.

For a PPU subprogram, the load address is origin-5. Five 12-bit PPU words are overlaid by the 60-bit loader table.

Data can be generated in locations starting with origin and above, but not below origin. The origin subfield does not serve the same function as ORG, nor does it replace ORG for setting the origin counter.

If the origin field is null for an absolute subprogram, the assembler uses address 000000 RA(S) as the origin for a CPU program and 0000 as the origin for a PPU program.

For a relocatable subprogram, the subfield is ignored. The loader automatically relocates the first subprogram to be loaded starting at RA(S)+100<sub>8</sub>, the second subprogram starting at the first available location following the first subprogram, and so forth.

**entry** For a CYBER 70/Model 76 or 7600 PPU assembly or for an absolute CPU assembly, this subfield contains an expression specifying the subprogram entry address, which can be symbolic.

**$l_1, l_2$**  Absolute expressions specifying the level numbers of the overlay.  $l_1$  is the primary level (0 through 63) and  $l_2$  is the secondary level (0-63). When the first IDENT identifies the main overlay,  $l_1$  and  $l_2$  can be omitted. If  $l_1$  is omitted, it is set to 00. If  $l_2$  is omitted, it is set to 00.

Because the first IDENT precedes any use of the BASE pseudo instruction, the level numbers on this IDENT are evaluated as decimal unless specifically designated as octal by a post radix.

**ppu** Absolute expression specifying the number of the PPU on which this program is to be loaded. On the first IDENT, this number is evaluated as decimal unless specifically designated as octal.

A location field symbol, if present, is ignored.

If the COMPASS assembler is called from within a FORTRAN compilation rather than by a COMPASS control statement, IDENT must be in columns 11 through 15.

When the subprogram does not include a TITLE instruction, COMPASS uses the IDENT variable field entry as the main subprogram title on the assembly listing.

**Example:**

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	IDENT	CT, CONTROL, CONTROL	
	ABS		ABSOLUTE CPU PROGRAM
	ORG	110P	
CONTROL	BSS	0	DEFINES SYMBOL CONTROL
	END		

Absolute CPU program CT will be loaded at origin address 00110<sub>8</sub>.

#### 4.2.2 END — END OF SUBPROGRAM

An END pseudo instruction must be the last instruction of each subprogram. It causes the assembler to terminate all counters, conditional assembly, macro generation, or code duplication. Before terminating assembly, COMPASS assembles any waiting remote text (see RMT).

For a relocatable subprogram, the assembler combines all local blocks into a relocatable subprogram block, generates the relocatable binary tables and produces the listing.

For an absolute assembly, the assembler assigns each block an origin relative to absolute zero, combines all blocks into an absolute subprogram or overlay, generates the absolute binary section and produces the listing.

END can also be used to signal the end of source statements from an external source (see XTEXT). In this case, it does not terminate the subprogram.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	END	trasym

sym                    Optional last word address symbol; if present, COMPASS defines it as the total subprogram length, including the literals block and all local blocks. The value is the last word address plus one.

trasym                A symbol specifying the entry point to which control transfers for a relocatable subprogram. This symbol must be declared as an entry point in a subprogram -- not necessarily the subprogram being assembled. At least one subprogram must specify a transfer address or the loader signals an error. If more than one subprogram indicates a transfer address, the loader uses the last one encountered.

For an absolute assembly, trasym is ignored.

**Example:**

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	IDENT	PROG1	
	ENTRY	REGIN	
	.	.	
	.	.	
REGIN	SR1	1	
	.	.	
	.	.	
	.	.	
	END	REGIN	

### 4.3 BINARY CONTROL

Pseudo instructions that allow the user extensive control of binary output produced by the assembler are summarized below and described fully in this section.

ABS	Specifies CPU absolute binary output
MACHINE	Specifies processor type
PPU	Specifies CYBER 70 Model 76 or 7600 PPU binary output
PERIPH	Specifies CYBER 180 Series; CYBER 170 Series; CYBER 70 Model 71, 72, 73, or 74; or 6000 Series PP binary output
IDENT	Begins absolute overlay or writes partial binary section
SEGMENT	Begins absolute overlay
SEG	Writes partial binary section
STEXT	Generates system text overlay
COMMENT	Inserts comments into the 7700g prefix table
NOLABEL	Suppresses header information on binary output
LCC	Passes loader control information to the relocatable loader
LDSET	Generates loader directive LDSET

#### 4.3.1 ABS — ABSOLUTE CPU PROGRAM

An ABS instruction declares a CPU program to be absolute. If used, it must be in the first statement group.

The following instructions are illegal in an absolute program:

EXT  
LCC  
REP  
REPC  
REPI

A symbol can be prefixed by =X if it is also defined conventionally; in this case, the =X has no significance because a conventional definition takes precedence (see Default Symbols in chapter 2).

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ABS	

Symbols in the location and variable fields, if present, are ignored. If a program contains both ABS and PERIPH (or PPU), the PERIPH (or PPU) instruction takes precedence.

**Example:**

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	IDENT	CT,CONTROL,CONTROL	
	ABS		ABSOLUTE CPU PROGRAM
	.	.	
	.	.	
CONTROL	ORG	119R	
	BSS	0	DEFINES SYMBOL CONTROL
	.	.	
	.	.	
	.	.	
	FND		

**4.3.2 MACHINE - DECLARE OBJECT PROCESSOR TYPE**

The MACHINE pseudo instruction specifies the type of computer system on which the object program can be executed successfully and optionally specifies hardware features needed by the object program. When the loader loads the object program, the required hardware features specified with MACHINE are reconciled against actual hardware features present; a missing feature causes the loader to issue a fatal diagnostic message. If used, MACHINE must be in the first statement group.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	MACHINE	type, hf <sub>1</sub> , hf <sub>2</sub> , hf <sub>3</sub> , ..., hf <sub>n</sub>

A location field symbol, if present, is ignored.

**type** Character string designating object processor type. The subfield can be any length and may contain any characters other than blank or comma. The first character identifies processor type, as follows:

- 6 The object program is restricted to the following computer systems: CYBER 180 Series; CYBER 170 Series; CYBER 70 Model 71, 72, 73, and 74; and 6000 Series. All machine instructions unique to the CYBER 70 Model 76 or 7600 Computer Systems are undefined.
- 7 The object program is restricted to a CYBER 70 Model 76 Computer System or to a 7600 Computer System. With the exception of the PS instruction (often used for subroutine entry points in CPU assemblies), all instructions unique to the following computer systems are undefined: CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series.

- 8 The object program is restricted to a model 810, 815, 825, 830, 835, 845, 855, 865, or 875 Computer System. All machine instructions unique to other computer systems are undefined. This pseudo instruction should not be used if S=AIDTEXT has been specified on the COMPASS control statement.

In a CPU assembly, if the MACHINE pseudo instruction is omitted, or the type subfield is blank, or its first character is not 6, 7, or 8, then all CPU instructions are defined, and the target and valid fields of the PRFX table in the object program are blanks. If the type subfield is present and its first character is 6, 7, or 8, the valid field contains 6X, 7X, or 8X. If the type subfield is at least two characters, the first character is 6, 7, or 8, and the second character is a digit (0-9), the target field contains those two characters.

In a PP assembly, if the MACHINE pseudo instruction is omitted, or the type subfield is blank, or its first character is not 6, 7, or 8, then: if the PERIPH pseudo instruction is present, MACHINE 6 is assumed; if the PPU pseudo instruction is present, MACHINE 7 is assumed. The target field of the PRFX table contains blanks, and the valid field contains 6P, 7P, or 8P.

hf<sub>i</sub> Optional subfield, a character string designating an optional hardware feature required for successful execution of the object program. The subfield may be any length and may contain any characters other than blank or comma. It has no effect on assembly of the program. The first character of the subfield is placed in the hardware-instruction-dependencies field in the PRFX table in the object program.

Recommended mnemonic letters are:

- C Compare/Move Unit
- D Distributive Data Path
- I Integer Multiply Instruction
- L ECS/LCM
- R Interlock Register
- X Central and Monitor Exchange Jumps

Up to nine hf<sub>i</sub> subfields are processed; any additional subfields are ignored. If the hf<sub>i</sub> subfields are omitted, the comma following type can also be omitted.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	MACHINE	6,CMU,LCM,XJ	

#### 4.3.3 PPU - CYBER 70 MODEL 76 OR 7600 PPU PROGRAM

A PPU instruction declares a program to be a CYBER 70 Model 76 or 7600 absolute PPU program rather than a CPU program. If used, PPU must be in the first statement group. For a description of binary format generated as a result of this instruction, refer to the Loader reference manual.

Floating point constants and the following instructions are illegal in a PPU assembly:

ENTRY	SEGMENT
ENTRYC	USELCM
EXT	R=
LCC	B1=1
REP	B7=1
REPC	
REPI	
SEG	

A symbol can be prefixed by = X if it is also defined conventionally.

If the program contains both a PPU and a PERIPH pseudo instruction, the PPU takes precedence. PPU programs permit symbols of the form used for CPU register designators; they are normal symbols having no special significance. The following instructions are legal but are not applicable in a PPU assembly:

OPDEF  
CPOP  
CPSYN  
PURGDEF

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	PPU	J

J                    A character string beginning with J supplied in the variable field alters the way that COMPASS assembles the variable expression on UJN, ZJN, NJN, MJN, or PJN instructions.

If J is not specified, COMPASS first tests the range of the expression against the short jump limit (+31). If the value is in range, COMPASS assembles the jump using the value of the expression. If the value is out of range, COMPASS performs a second test, this time using the expression value minus the location counter value. If the value is now in range, COMPASS assembles the instruction using the expression value minus the location counter value. However, if it is out of range, a fatal error is flagged.

Selection of the J option causes COMPASS to always subtract the value of the location counter from the value of the expression.

As a result, COMPASS is able to differentiate between an expression value that is an absolute address in the short jump range from an expression value that is a true relative address.

A symbol in the location field, if present, is ignored.

**Example:**

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
740			PPU		
760	0357	TAG	• RSS UJN	20R TAG-*	EXPRESSION < 37B

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
740			PPU	JUMP	
760	0357	TAG	• RSS UJN	20R TAG	EXPRESSION-* < 37B

**4.3.4 PERIPH - CYBER 180 SERIES; CYBER 170 SERIES; CYBER 70 MODELS 72, 73, 74; OR 6000 SERIES PPU PROGRAM**

A PERIPH instruction declares a program to be a CYBER 180 Series or a CYBER 170 Series or CYBER 70 Model 72, 73, 74, or 6000 Series absolute PPU program rather than a CPU program. If used, PERIPH must be in the first statement group. For a description of binary output produced as a result of this instruction, refer to the Loader Reference Manual.

Floating point constants and the following instructions are illegal in a PPU assembly:

ENTRY	LCC	REPI	R=
ENTRYC	REP	SEG	B1=1
EXT	REPC	USELCM	B7=1

A symbol can be prefixed by =X if it is also defined conventionally.

PPU programs permit symbols of the form used for CPU register designators; they are normal symbols having no special significance. The following instructions are legal but are not applicable to PPU assemblies:

OPDEF  
CPOP  
CPSYN  
PURGDEF

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	PERIPH	J

**J** A character string beginning with J supplied in the variable field alters the way that COMPASS assembles the variable field expression on UJN, ZJN, MJN, or PJN instructions.



If J is not specified, COMPASS first tests the range of the expression value against the short jump limit (+31). If the value is in range, COMPASS assembles the jump using the value of the expression. If the value is out of range, COMPASS performs a second test, this time using the expression value minus the location counter value. If the value is now in range, COMPASS assembles the instruction using the expression value minus the location counter value. However, if it is out of range, a fatal error is flagged.

Selection of the J option causes COMPASS to always subtract the value of the location counter from the value of the expression.

For an example illustrating how to use J, see the PPU pseudo instruction.

A symbol in the location field, if present, is ignored.

### 4.3.5 IDENT - IDENTIFY AND GENERATE OVERLAY

Two or more IDENT pseudo instructions are permitted in CPU absolute or PPU assemblies. Second and subsequent IDENT instructions having nonblank variable fields cause generation of overlays. IDENT differs from SEGMENT in the way it generates overlays. First, it allows the specification of overlay numbers. Second, the USE table and all block counters are reinitialized. The symbol table is not cleared; all symbols are reassigned absolute addresses relative to absolute zero. Thus, an ORG to a previously defined symbol restarts the absolute block at the symbolic address. The third difference is that normally the end of the overlay is determined by the last word address, the maximum origin counter value of the last block established in the overlay. A preceding SEG or SEGMENT can alter this, however (Section 3.4).

For a CPU assembly, an IDENT with a blank variable field causes a partial binary write. The output is not terminated by an end-of-section or a new 77<sub>8</sub> table. However, the USE table and the block counters are reinitialized and each symbol in the symbol table is assigned an absolute address.

Following an IDENT, COMPASS assumes that all blocks, including the literals block are complete. Block structuring starts fresh with the new overlay or portion of binary. Thus, each new overlay or partial can use the same block names as are used by other overlays or partial and each can have a literals block.

For a blank IDENT, an attempt to write into or reset the origin counter to a location in a partial section written separately causes a range error. Following the IDENT, the origin of the new absolute block is the next word after the binary written out, that is, it is lwa+1.

The format of the IDENT varies according to the type of assembly as follows:

CPU Absolute Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name, origin, entry, $l_1, l_2$

or

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	

**7600 PPU Absolute Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name, origin, entry, ppu

**6000 Series PPU Absolute Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IDENT	name, origin

- name** Name of the overlay. For a CPU program, 1-7 characters, the first of which must be alphabetic (A-Z); for CYBER 180 Series or CYBER 170 Series or a CYBER 70/Model 72, 73, or 74, or a 6000 Series PPU program, 1-3 characters; for a CYBER 70/Model 76 or 7600 PPU program, 1-7 characters. In all cases, the last character must not be a colon. A name is a loader linkage symbol required for overlays.
- origin** An expression specifying the first word address of the overlay. The overlay control word and all code assembled starting with this address and ending with the next SEGMENT, nonblank IDENT, or END instruction comprises the overlay. For a single entry point CPU program, the load address for the overlay is origin-1. The word at origin-1 is overlaid by the 50g loader table. For a multiple entry point CPU program, the load address for the overlay is origin-wc-1, where wc is the number of entry points listed in the 51g loader table.
- For a PPU subprogram, the load address is origin-5. Five 12-bit PPU words are overlaid by the 60-bit loader control table. Data can be generated in locations starting with origin and above, but not below origin. The origin subfield does not serve the same function as ORG nor does it replace ORG for setting the origin counter. The origin of an overlay can be below the origin specified on any other IDENT or SEGMENT.
- entry** An expression specifying the overlay entry address. When the overlay is called, control optionally transfers to this address.
- $l_1, l_2$**  Absolute expressions specifying the level numbers of the overlay for CPU programs only.  $l_1$  is the primary level (00-77g),  $l_2$  is the secondary level (00-77g). If base is M,  $l_1$  and  $l_2$  are assumed to be octal. If  $l_1$  and  $l_2$  are not specified,  $l_1$  is set to 01 and  $l_2$  is set to 00.

ppu An absolute expression specifying the number of the PPU in which the overlay is to be loaded. If base is M, ppu is assumed to be octal.

A location field symbol, if present, is ignored.

The binary is written on the file specified by the B parameter on the COMPASS control statement. END dumps the last overlay or completes a partially written section.

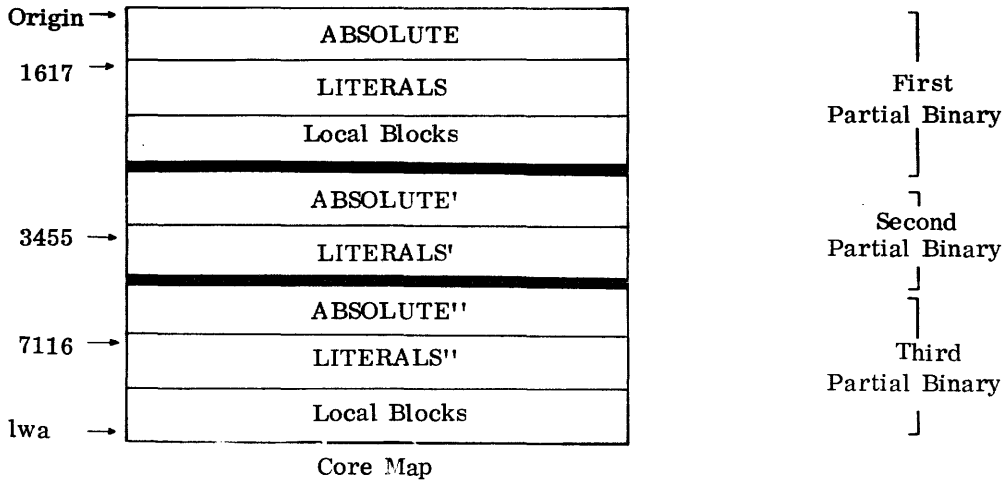
Examples:

The following program uses IDENT for overlay creation. Symbols T.OVL, O.DMP1, etc. are defined on a system text overlay.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	IDENT	DMP.1, T.OVL, O.DMP1	
	ABS		
	BASE	M	
	COMMENT	10/07/70.CONTROL CARD CALL.DMP.	
	LIST	G	
	SST		
	ORG	T.OVL	OVERLAY
	QUAL	DMP1	DMP1
DMP	SX0	B1	
	.	.	
	.	.	
	.	.	
	QUAL	DMP2	
	IDENT	DMP2, T.OVL, O.DMP2	
	ORG	T.OVL	OVERLAYS DMP2
URW2	SX0	B6+1	THROUGH DMP8
	.	.	
	.	.	
	.	.	
	QUAL	DMP9	
	IDENT	DMP.9, T.OVL, O.DMP9	OVERLAY
	ORG	T.OVL	DMP9
	SX0	O.DMP2+F.MDE	
	.	.	
	.	.	
	.	.	
	END		END OVERLAY DMP9

The following program uses IDENT instructions having blank variable fields.

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
		IDENT	VVV,110B,ENT	
		ABS		
		ORG	110R	
		SX0	1	
		.	.	
		.	.	
1617		LIT	1,2,3	First Partial Binary
		.	.	
		.	.	
		IDENT		
		.	.	
		.	.	
3455		LIT	2,3	Second Partial Binary
		.	.	
		.	.	
		IDENT		
		.	.	
		.	.	
7116		LIT	1,2	Third Partial Binary
		.	.	
		.	.	
		FND		



### 4.3.6 SEGMENT - GENERATE BINARY SEGMENT

The SEGMENT pseudo instruction produces overlays at assembly time. It has many of the features of IDENT and is included primarily to provide another way of handling literals. Use of SEGMENT is intended for 6000 Series CPU absolute or PPU assemblies. For a relocatable subprogram, a SEGMENT pseudo instruction causes BSSZ code and the FILL, REPL, and LINK relocatable tables to be written on the binary output file.

The first SEGMENT causes all binary accumulated since the IDENT to be dumped as the main (0, 0) overlay. Each subsequent SEGMENT generates a new overlay with the specified level numbers. END dumps the last overlay. When COMPASS encounters a SEGMENT pseudo instruction, it does not clear the symbol table or block declarations. All blocks other than the block in use must be complete. For a CPU assembly, the literals block must be in one overlay only but that overlay can be any overlay.

#### Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	SEGMENT	origin, entry, $l_1, l_2$

**name** Name of overlay. For a CPU program, 1-7 characters, first of which must be alphabetic (A-Z); for a PPU subprogram, 1-3 characters. In all cases, the last character must not be a colon. It is a required loader linkage symbol.

**origin** A relocatable expression specifying the first word address of the overlay. It can only be an address in the block in use. The overlay loader table and all code assembled starting at this address and ending with the next SEGMENT, nonblank IDENT, or END instruction comprises the overlay.

For a CPU program the load address for the record is origin-1. The word at origin-1 is overlaid by the 50<sub>8</sub> loader table.

For a PPU subprogram, the load address is origin-5. Five 12-bit PPU words are overlaid by the 60-bit loader table. Data can be generated in locations starting with origin and above, but not below origin. The origin subfield does not serve the same function as ORG nor does it replace ORG for setting the origin counter. The origin of an overlay can be below the origin specified on any other IDENT or SEGMENT.

**entry** An expression specifying the overlay entry address. It is used for CPU assemblies only. When the overlay is called, control optionally transfers to this address.

**$l_1, l_2$**  Absolute expressions specifying the level numbers of the overlay for CPU programs only.  $l_1$  is the primary level (00-77<sub>8</sub>),  $l_2$  is the secondary level (00-77<sub>8</sub>). If base is M,  $l_1$  and  $l_2$  are assumed to be octal. If  $l_1$  and  $l_2$  are not specified,  $l_1$  is set to 01 and  $l_2$  is set to 00.

Example:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		IDENT	SAM, ENTA	
	ENTA	ABS		
		ORG	110R	
		RSS	0	ENTRY POINT
		.	.	
		.	.	
	OVLOC	RSS	9	OVERLAY LOAD POINT
		.	.	
		.	.	
	SEG1	SFGMENT	STRT, ENTB	
		ORG	OVLOC	
		RSS	1	LOADER TABLE
	STRT	RSS	0	FIRST WORD OF OVERLAY
		.	.	
		.	.	
	ENTB	RSS	0	EXECUTION BEGINS HERE
		.	.	
		.	.	
		END		END OF OVERLAY

SEG1 is loaded as an overlay upon a call for the loader from the program. The first word of the overlay is loaded at OVLOC +1, following the loader table. The entry point to the overlay and the first executable instruction is at ENTB. The overlay, when executed occupies the area of the main program beginning at OVLOC.

#### 4.3.7 SEG - WRITE PARTIAL BINARY

The SEG pseudo instruction permits the generation of a CPU absolute subprogram or overlay in less core than would otherwise be required for assembly. It is illegal in PPU and relocatable assemblies.

SEG causes COMPASS to write on the binary output file all binary information accumulated since the previous IDENT, SEGMENT, or SEG pseudo instruction. It does not write an end-of-section or begin a new PRFX table. A SEGMENT, IDENT, or END instruction completes the binary section.

SEG does not affect the location and origin counters. The user cannot resume use of a block established prior to the SEG, except for the block in use when the SEG was encountered. An attempt to reset the origin counter so as to resume a block already written out causes an R error. Also, since the block group is incomplete and the names of the blocks already written out are still in the USE table, no new blocks can be established using the same block names as were used prior to the SEG.

The literals block is written in the portion that contains the end of the absolute block.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	SEG	

Symbols in the location field and variable field, if present, are ignored.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	IDENT	NAME,ORIGIN,ENTRY	
	ABS		
	USE	A	
	.	.	
	.	.	
	.	.	
	SEG		
	USE	B	
	.	.	
	.	.	
	.	.	
	SEG		
	.	.	
	.	.	
	.	.	
	END		

#### 4.3.8 STEXT - GENERATE SYSTEM TEXT RECORD

As a result of an STEXT pseudo instruction, binary output for the subprogram consists of all symbols, micros, and opcodes (macros, opdefs, and machine and pseudo instructions), written in overlay format at the end of pass one. The STEXT instruction must be in the first statement group.

The system text overlay becomes available in other assemblies through use of the G or S option on the COMPASS control statement (chapter 10). Through this feature, information in the system text overlay need be processed only once for all COMPASS programs using the same system text. System text overlays cannot be generated and used in the same assembly batch; system text overlays generated by one COMPASS control statement call can be used only by assemblies performed by later COMPASS control statement calls.

The symbols included in the system text overlay written are all symbols defined in the assembly except those for which at least one of the following is true:

The symbol value is relocatable or external.

The symbol is qualified.

The symbol is redefinable (i. e., defined by SET, MAX, MIN, or MICCNT).

The symbol is defined by statements read by XTEXT or occurring between CTEXT and ENDX.

The symbol is defined by SST (i. e., is a system symbol input to the present system text assembly).

The symbol is 8 characters beginning with † ↓.

All defined micros are included in the system text overlay.

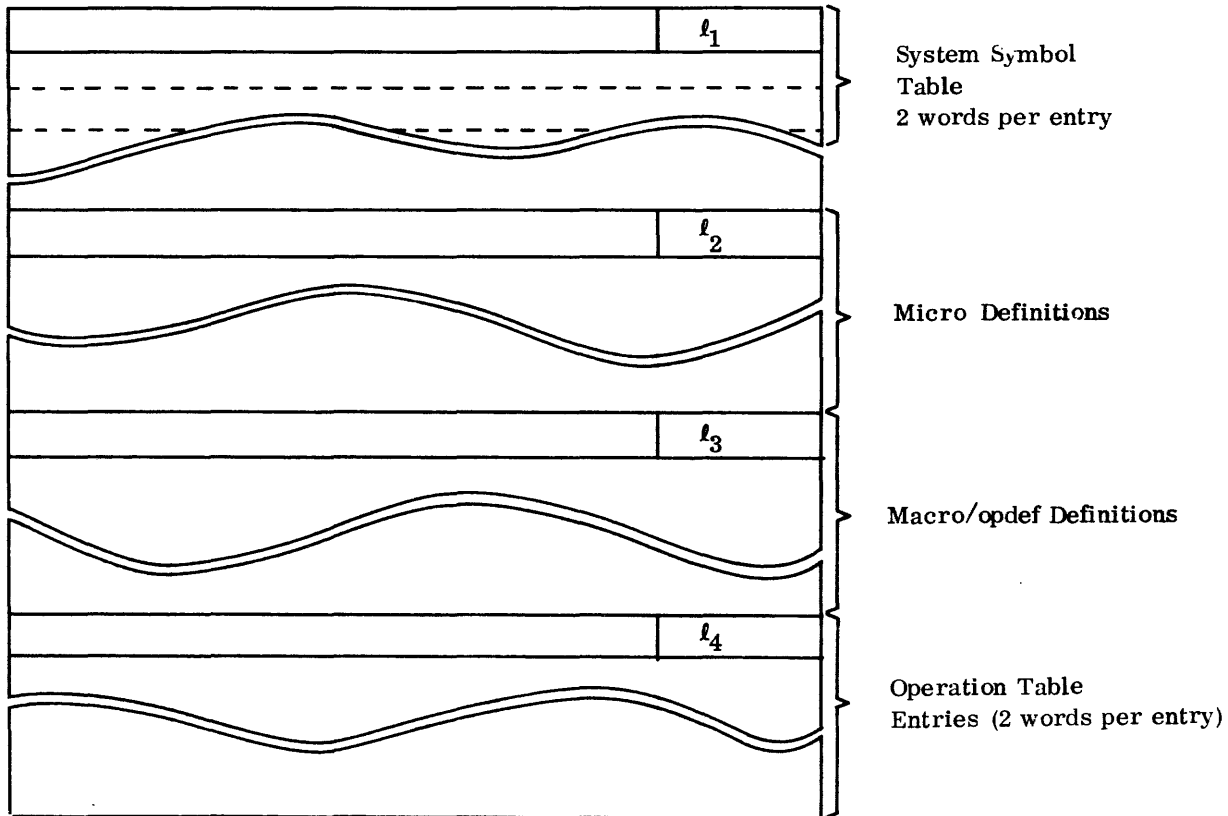
All program-defined opcodes are also included. Machine and pseudo instructions automatically defined by COMPASS, and opcodes defined by system text input (if any) to the assembly, are not included.

When a system text overlay is used as input to an assembly through the G or S option on a COMPASS control statement, all of the micros and opcodes in the system text are automatically defined at the start of each assembly; however, the symbols in the system text are defined only for those assemblies that contain the SST pseudo instruction.

A system text overlay on the library is an absolute overlay that has the following control table:

59	48	42	36	00
5000	01	01	000000000000	

Format of Text:



$l_i$  = Number of words in each part of overlay



Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
rname	STEXT	

**rname** Name assigned to overlay; 1-7 alphanumeric characters, of which the first must be a letter (A-Z) and the last must not be a colon. It is placed in the prefix table that precedes the overlay.

If rname is blank, COMPASS uses the name from the IDENT instruction and generates the system text only. Otherwise, the system text is generated in addition to the relocatable or absolute binary and precedes the binary output on the binary file.

An entry in the variable field, if present, is ignored.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	IDENT	SYSTEXT	
	STEXT		
MPRS	BASE	MIXED	] SYSTEM CONSTANTS, SYMBOLS, AND COMMUNICATIONS AREAS
.	EQU	100	
.	.	.	
.	.	.	
IRTS	EQU	7777	] SYSTEM-DEFINED MACROS AND OPDEFS
IXX/X	OPDEF	I,J,K	
.	.	.	
.	.	.	
SYSCOM	ENDM		] SYSTEM-DEFINED MICROS
.	MACRO	N	
.	.	.	
.	.	.	
DATE	ENDM		] SYSTEM-DEFINED MICROS
.	MICRO	1,10,*...*	
.	.	.	
.	.	.	
.	.	.	
	END		

### 4.3.9 COMMENT—PREFIX TABLE COMMENT

The COMMENT pseudo instruction inserts the character string specified in the variable field into the eighth through fourteenth words of the PRFX table in the object program. The prefix table, and thus the comment, is ignored by the loader but identifies the section. If a subprogram contains more than one COMMENT instruction, the new comments are appended to the table for the most recent binary control statement. If the subprogram contains a NOLABEL instruction, the COMMENT instruction is meaningless. COMMENT instructions following SEG and blank IDENT pseudo instructions are ignored without notification.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	COMMENT	string

**string** COMPASS searches the columns following the blank that terminates the operation field. If it does not find a nonblank character before the default comments column (see COL pseudo instruction), it takes the characters starting with the default comments column minus one. Otherwise, the character string begins with the first nonblank character following the operation field. In either case, the last character of the string is the last nonblank character of the statement. 1 to 10 blanks are appended on the right so that the string is followed by at least one blank and the length of the string is a multiple of 10 characters. If the variable and comment fields are all blanks, the string consists of 10 blanks. If the string length is more than 70 characters, all characters beyond the 70th are lost.

A location field symbol, if present, is ignored. Refer to section 4.3.5 for an example.

### 4.3.10 NOLABEL — DELETE HEADER TABLE

The NOLABEL instruction modifies the format of the binary output produced by COMPASS for an absolute assembly by optionally suppressing header information. It is particularly convenient for generating deadstart programs which must be loaded at location zero.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	NOLABEL	I

I Optional; if the variable field contains a character string beginning with an I, COMPASS suppresses all prefix (7700g) tables, but retains the other program header tables.

If the I option is omitted, COMPASS suppresses all of the following:

- Prefix tables (7700g)
- Overlay control tables (5000g)
- Multiple entry point tables (5100g)
- PP header control tables

A location field symbol, if present, is ignored. NOLABEL is illegal in a relocatable CPU assembly.

#### 4.3.11 LCC—LOADER DIRECTIVE

The LCC pseudo instruction provides a means of including loader directives with the tables for a relocatable program.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	LCC	directive

directive First nonblank character following LCC to the first blank. For directive formats, refer to the Loader reference manual.

A location field symbol, if present, is ignored.

COMPASS writes a directive as a section in packed display code for subsequent interpretation by the loader. COMPASS does not edit the directive; the loader recognizes illegal forms at load time.

#### 4.3.12 LDSET—GENERATE LDSET OBJECT DIRECTIVES

The LDSET pseudo instruction generates loader LDSET directives for a relocatable program. A program may contain any number of LDSET instructions. COMPASS collects all LDSET options and writes a single LDSET (7000g) table in the relocatable binary output between the PRFX (7700g) table and the PIDL (3400g) tables. The LDSET table is not written if LDSET instructions do not appear in the program. LDSET is not allowed in a PP or absolute CPU assembly.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	LDSET	options

options One or more options separated by commas. See the Loader reference manual for further information, including applicability to a particular operating system.

LIB Clear local library set.

**LIB=libname** Add the specified libraries to the local library set. More than one library can be specified by separating library names with a slash, in the form:

libname<sub>1</sub>/libname<sub>2</sub>/.../libname<sub>n</sub>

**MAP** Write load map to file OUTPUT.

**MAP=p** Write load map to file OUTPUT. Map items are selected by p:

<u>NOS and NOS/BE</u>		<u>SCOPE 2</u>	
N	No map.	0 or O	No map.
S	Statistics.	S	Statistics.
B	Block list.	B	Statistics and block list.
E	Entry point list.	E	Statistics, block list, and entry point list.
X	Cross reference map.	X	Statistics, block list, entry point list, and cross reference map.

For NOS and NOS/BE, p can be written as N or as any combination of SBEX in any order.

**MAP=p/lfn** Write load map to file named lfn. p is as above.

**MAP=/lfn** Write load map to file named lfn. Installation default determines items on the map.

**PS=p** Select page size for load map by a specification of number of lines. p can be decimal 10 through 999999. A value outside this range results in the installation default page size. This option is not supported by SCOPE 2.

**PD=p** Select print density for load map by a specification of decimal number of lines per inch. This option is not supported by SCOPE 2. p can be:

6 6 lines per inch.

8 8 lines per inch.

other Installation default.

**PRESET=p** Preset memory to the value specified by p. Under NOS/BE, p can be a 1 through 20 digit octal number with an optional + or - prefix and an optional B suffix.

p can also be one of the following key words:

NONE	No presetting for ECS (or for LCM and SCM under SCOPE 2); same as ZERO for CM				
ZERO	0000	0000	0000	0000	0000
ONES	7777	7777	7777	7777	7777
INDEF	1777	0000	0000	0000	0000
INF	3777	0000	0000	0000	0000
NGINDEF	6000	0000	0000	0000	0000
NGINF	4000	0000	0000	0000	0000
ALTZERO	2525	2525	2525	2525	2525
ALTONES	5252	5252	5252	5252	5252
DEBUG	6000	0000	0004	0040	0000

**PRESETA=p** p can be as defined for PRESET. The lower 17 bits (CM/SCM) or lower 24 bits (ECS/LCM/LCME) of each word contains its address.

<b>ERR=ALL</b>	Select loader abort for any loader errors.
<b>ERR=FATAL</b>	Select loader abort only for fatal loader errors.
<b>ERR=NONE</b>	Select loader abort only for catastrophic loader errors.
<b>REWIND</b>	Reset the default REWIND/NOREWIN option for load files to REWIND. The NR parameter on LOAD and SLOAD directives can override this default for individual files.
<b>NOREWIN</b>	Reset the default REWIND/NOREWIN option for load files to NOREWIND. The R parameter on LOAD and SLOAD directives can override this default for individual files.
<b>EPT=eptname</b>	If the symbol eptname is defined, declare it an entry point of the CAPSULE or OVCAP binary subsequently generated by the loader. This parameter can be used to specify more than one entry point; entry point names must be separated by a slash in the form eptname <sub>1</sub> /eptname <sub>2</sub> /.../eptname <sub>n</sub> .
<b>NOEPT=eptname</b>	Do not declare eptname as an entry point of the CAPSULE or OVCAP binary subsequently generated by the loader. This parameter can be used to specify more than one entry point. In this case, entry point names must be separated by a slash in the form eptname <sub>1</sub> /eptname <sub>2</sub> /.../eptname <sub>n</sub> .
<b>USEP=pname</b>	Cause the designated object modules to be loaded whether or not they are needed to satisfy external references. More than one module can be specified by separating module names by a slash in the form pname <sub>1</sub> /pname <sub>2</sub> /.../pname <sub>n</sub> .
<b>USE=eptname</b>	Cause the load of object modules containing the specified entry points whether or not they are needed to satisfy external references. More than one entry point can be specified by separating entry point names by a slash in the form eptname <sub>1</sub> /eptname <sub>2</sub> /.../eptname <sub>n</sub> .
<b>COMMON</b>	Assign all labeled blocks to a segment such that the blocks are available to all segments that reference them. Valid for segment loads only.
<b>COMMON=blkname</b>	Assign the labeled common block named blkname to a segment such that it is available to all segments that reference it. Valid for segment loads only. More than one block name can be specified by separating the individual block names with a slash in the form blkname <sub>1</sub> /blkname <sub>2</sub> /.../blkname <sub>n</sub> .
<b>SUBST=pair</b>	Treat external references to eptname <sub>1</sub> as though they were references to eptname <sub>2</sub> , where the entry point names are specified as a pair in the form eptname <sub>1</sub> -eptname <sub>2</sub> .  More than one pair of entry point names can be specified by separating the pairs with a slash in the form pair <sub>1</sub> /pair <sub>2</sub> /.../pair <sub>n</sub> .
<b>OMIT=eptname</b>	Omit satisfying external references to the specified externals. More than one entry point name can be specified by separating the names with a slash in the form eptname <sub>1</sub> /eptname <sub>2</sub> /.../eptname <sub>n</sub> .
A location field symbol, if present, is ignored.	

See the Loader reference manual for details of these parameters, including the operating system to which a given option applies.

#### 4.4 MODE CONTROL

Mode control pseudo instructions influence the basic operating characteristics of the assembler. Specifically, the instructions allow the programmer to alter the way in which the assembler:

Interprets binary data	BASE pseudo instruction
Generates character data	CODE pseudo instruction
Interprets the beginning of comments on statements	COL pseudo instruction
Qualifies symbols or does not qualify them	QUAL pseudo instruction
Interprets the R=instruction	B1=1 or B7=1 pseudo instruction

In each case, the assembler has a default mode which it uses if one of these instructions is never used.

##### 4.4.1 BASE — DECLARE NUMERIC DATA MODE

The BASE pseudo instruction declares the mode of interpretation for numeric data for which a base radix is not explicitly defined. Use of the BASE pseudo is optional; if BASE is not used in a subprogram, COMPASS evaluates unspecified numeric data as decimal.

An alternate application of BASE is to define the previous base as a micro.

In addition, if no program or system micro named BASE has been defined, COMPASS changes the predefined BASE micro to be a single letter D, M, or O, corresponding to the new mode established by this BASE instruction.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	BASE	mode

**mname** Optional 1 through 8 character micro name by which the previous BASE mode can be referenced in subsequent BASE instructions. If mname is present, the value of the micro named mname is (re)defined to be a single letter D, M, or O, corresponding to the BASE mode in effect prior to this BASE instruction.

**mode** Blank, in which case the base remains unchanged, or 1 through 8 characters, the first of which designates the new base as follows:

- O** Octal assembly base; any subsequent use of a data item not specifically identified by an O, D, or B prefix or suffix is evaluated as octal. For example, the constants 15 and 15B are evaluated as 15g; constant 15D is evaluated as 17g. Any item containing an 8 or 9 without a D radix is flagged as erroneous. Exceptions are scale factors, character counts, shift counts (S modifier), and binary point positions, which are always considered decimal.
- D** Decimal assembly base; any subsequent use of a data item not specifically identified by an O, D, or B prefix or suffix is evaluated as decimal.

M Mixed assembly base; any subsequent use of a data item not specifically identified by an O, D, or B is evaluated as decimal if it is one of the following. Otherwise, it is evaluated as octal.

VFD bit count

IF, ELSE, or SKIP line count

MICRO, OCTMIC, or DECMIC character count

B, C, or I subfield in REP or REPI

DUP or ECHO line count

Character count

Shift counts (S modifier)

Scale factors

Binary point position

COL column number

DIS word count

SPACE line count

\* Use base in effect prior to current base. The assembler records occurrences of BASE pseudo instructions and maintains a table of the most recent 50 occurrences. Each BASE \* resumes use of the most recent entry and removes it from the list. When the subprogram contains more BASE \* instructions than there are entries in the stack, COMPASS uses a decimal base.

other If the variable field is not blank and does not contain one of the above, COMPASS sets an error flag.

Examples:

This example shows the effect of BASE on a VFD instruction defining a 48-bit field containing 10g.

Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
		I		18	30
	D=0		BASE	0	
000000000000000010			VFD	60/10	
			.	.	
			.	.	
	O=D		BASE	0	
00000000000010	0000		VFD	48/8	
			.	.	
			.	.	
	D=M		BASE	M	
00000010	00000000		VFD	48/10	

The following example illustrates the micro capability of BASE:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
D → M	SAVER	BASE	M	SAVE BASE IN USE
	.	.	.	
	.	.	.	CODE USING BASE M
	.	.	.	
		BASE	≠SAVER≠	RESTORE SAVED BASE
M → D		BASE	D	RESTORE SAVED BASE
	.	.	.	
	.	.	.	
	.	.	.	

#### 4.4.2 CHAR-DEFINE OTHER CHARACTER DATA CODE

The CHAR pseudo instruction defines character data codes to be used when the CODE O (for Other) mode is in effect.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	CHAR	exp1, exp2

exp1      Evaluatable absolute expression whose value is 00 to 77g. The value of exp1 is the display code value of the character to be redefined.

exp2      Evaluatable absolute expression whose value is 00 to 77g. The value of exp2 is the new code other value of the character designated by exp1.

A location field symbol, if present, is ignored.

Initially, all code other values are the same as display code. CHAR need be used only for those characters whose code other values are different from display code. Characters may be redefined as many times as desired by subsequent CHAR pseudo instructions.

Example:

	LOCATION	OPERATION	VARIABLE SUBFIELDS
00 → 63		CHAR	0,63B      INTERCHANGE COLON AND
63 → 00		CHAR	63B,0      PERCENT FOR CODE OTHER

#### 4.4.3 CODE — DECLARE CHARACTER DATA CODE

The CODE pseudo instruction declares that until the next CODE pseudo instruction is encountered all constants, character strings, and character data items are to be generated in the specified code. Character data can be generated in ASCII (American Standard code for Information Interchange), display, external BCD, or internal BCD, codes. If no CODE instruction is used, COMPASS generates display code. Codes are given in appendix A.



An alternative application of CODE is to define the previous code as a micro.

In addition, if no program or system micro named CODE has been defined, COMPASS changes the predefined CODE micro to be a single letter A, D, E, I, or O, corresponding to the new mode established by this CODE instruction.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	CODE	char

**mname** Optional 1-8 character micro name by which the previous CODE mode can be referenced in subsequent CODE instructions. If mname is present, the value of the micro named mname is (re)defined to be a single letter A, D, E, I, or O, corresponding to the CODE mode in effect prior to this CODE instruction.

**char** The first character of a string indicates the code conversion:

- A** ASCII six-bit subset
- D** Display
- E** External BCD
- I** Internal BCD
- O** Other code, defined by CHAR pseudo instructions
- \*** Use code in effect prior to current code. The assembler records occurrences of CODE pseudo instructions and maintains a table of the most recent 50 occurrences. Each CODE \* resumes use of the most recent entry and removes it from the list. When the subprogram contains more CODE \* instructions than there are entries in the stack, COMPASS generates display code.

**Example:**

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
17252420252400000000		DATA	0LOUTPUT	
D→A		CODE	ASCII	
57656460656400000000		DATA	0LOUTPUT	
A→E		CODE	EXTERNAL BCD	
46242347242300000000		DATA	0LOUTPUT	
E→I		CODE	INTERNAL BCD	
46646347646300000000		DATA	0LOUTPUT	
I→D		CODE	DISPLAY	
17252420252400000000		DATA	0LOUTPUT	
O→I		CODE	*	
46646347646300000000		DATA	0LOUTPUT	

#### 4.4.4 QUAL — QUALIFY SYMBOLS

The QUAL pseudo instruction signals the beginning of a sequence of code in which all symbols defined in it are either qualified or are unqualified (global). If no QUAL is in a subprogram, all symbols are defined as global.

An alternative application of QUAL is to define the previous qualifier as a micro.

In addition, if no program or system micro named QUAL has been defined, COMPASS changes the predefined QUAL micro to be the new qualifier name established by this QUAL instruction.

Within a QUAL sequence in which a symbol is defined, a symbol reference need not be qualified. Used outside the sequence, the symbol must be referenced as/qualifier/symbol. Thus, a symbol and a qualifier become a unique identifier local to the sequence in which the symbol was defined. The same symbol used with a different qualifier is local to a different QUAL sequence. If a symbol is defined with no qualifier as well as being defined as qualified, a reference to the symbol within the QUAL sequence is assumed to be a reference to the qualified symbol rather than to the global symbol. In this case, a reference to the global symbol must be written as // symbol. However, in a NOREF statement when the unqualified symbol is previously defined and the qualified symbol is not, COMPASS assumes the reference is to the unqualified symbol.

Default symbols and linkage symbols are not qualified.

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	QUAL	qualifier

**mname** Optional 1-8 character micro name by which the previous qualifier can be referenced in subsequent QUAL instructions or symbol references. If mname is present, the value of the micro named mname is (re)defined to be the 0-8 characters comprising the qualifier in effect prior to this QUAL instructions.

**qualifier** A symbol qualifier or \* or blank, as follows:

**qualifier** 1-8 character name, the first character of which cannot be \$ or = or : or numeric. The qualifier cannot contain the characters

+ - \* / , or ^

A blank terminates the qualifier.

Any symbol defined subsequent to this QUAL up to the next QUAL must be referenced from outside the QUAL sequence as

/qualifier/symbol

The current qualifier appears as the third sub-subtitle on the assembly listing (section 11.1).

**\*** The assembler resumes using the qualifier in use prior to the current qualifier. The assembler records occurrences of QUAL pseudo instructions and maintains a table of the most recent 50 occurrences. Each QUAL \* resumes use of the most recent entry and removes it from the list. When the subprogram contains more QUAL \* instructions than there are entries in the stack, COMPASS uses the null (global) qualifier.

blank

A blank variable field causes any symbols defined up to the next QUAL to be global. A global symbol does not require a qualifier.

#### NOTE

The first attempt to redefine a global symbol from within a QUAL sequence results in A and U errors. The symbol is defined local to the QUAL sequence with a zero value. To avoid fatal errors, precede any redefinition instruction (SET, MAX, MIN, or MICCNT) within a QUAL sequence with a blank QUAL and follow it with a QUAL\*.

#### Examples:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
BCDF	QUAL SX6 . . FN	PASS1 F . . LOC1	BCDE QUALIFIED BY PASS1
RCDE	QUAL EQU QUAL . . .	PASS2 LOC2 . . .	RCDE QUALIFIED BY PASS2 SYMBOLS GLOBAL FROM NOW ON
GLOB	BSS . . RJ . . RJ	 . . /PASS1/BCDF . . /PASS2/RCDE	GLOB IS GLOBAL JUMP TO PASS1 ROUTINE JUMP TO PASS2 ROUTINE

Location      Code Generated

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	TAR	MACRO	QLOCK, KWAL
		USE	QLOCK
		QUAL	KWAL
	TAG1	BSS	10R
	TAG2	VFD	60/-1
		USE	*
		QUAL	*
		ENDM	
		.	
		.	
		.	
	TAR		ONE, ONE
		USE	ONE
		QUAL	ONE
10044	TAG1	BSS	10R
10054	TAG2	VFD	60/-1
		USE	*
		QUAL	*
		ENDM	
	TAR		TWO, TWO
		USE	TWO
		QUAL	TWO
10055	TAG1	BSS	10R
10065	TAG2	VFD	60/-1
		USE	*
		QUAL	*
		ENDM	

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
		QUAL	Z
		BSS	0
		.	.
		.	.
		.	.
		QUAL	B
		=	/Z/Z1
Z1			Z1 QUALIFIED BY 7
Z1			EQUATE SYMBOLS SO THAT Z1 IN Z CAN BE REFERRED TO AS Z1 IN B

#### 4.4.5 B1 = 1 AND B7 = 1 — DECLARE THAT B REGISTER CONTAINS ONE

The B1=1 and B7=1 pseudo instructions declare that in this CPU subprogram, the contents of the B1 register or the B7 register, respectively, are one. These instructions do not produce code; they alter the way in which code is generated by the R= instruction (section 4.8.7) and define the symbol B1=1 or B7=1. If more than one instruction is used, the assembler uses the last one encountered.

**Formats:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	B1=1 B7=1	

A symbol in the location or variable field is ignored.

Note that loading the respective B register with one is the user's responsibility.

For an example of use, refer to R= (section 4.8.7).

**4.4.6 COL — SET COMMENTS COLUMN**

The COL pseudo instruction sets the column number at which the comments field can begin when the variable field is blank. If no COL instruction is used in the subprogram, COMPASS uses 30.

LOCATION	OPERATION	VARIABLE SUBFIELDS
	COL	n

n An absolute evaluable expression designating the column number; n 12. When base is M, n is assumed to be decimal. If n is less than 12, COMPASS sets the column at 12. If n is zero or blank, COMPASS sets the column to 30, the default column.

If the current operation field extends past the current comments column, COMPASS substitutes a very large number for n in the current instruction only; that is, if n is less than or equal to the last column of the operation field, a variable field must be present if a comment is present.

A location field symbol, if present, is ignored.

**Example:**

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
44		COL USE	36	RETURN TO BLOCK 0

In this example, subsequent statements for which the variable field is blank cannot have comments beginning before column 36.

## 4.5 BLOCK COUNTER CONTROL

Counter control pseudo instructions establish local blocks, labeled common blocks, and blank common blocks in addition to the absolute, zero, and literal blocks established by the assembler; they control use of all program blocks, and provide the user with a means of changing origin, location, and position counters.

### 4.5.1 USE — ESTABLISH AND USE BLOCK

USE establishes a new block or resumes use of an already established block. The block in use is the block into which code is subsequently assembled. A user may establish up to 252 blocks in a block group.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	USE	block

**block** Identifies block to be used, as follows:

- 0 or blank** Nominal block (absolute or 0).
- //** Blank common block; for a relocatable subprogram, this block cannot contain data. The only storage allocation instructions that can follow are BSS and ORG. The BSSZ instruction is illegal because it presets the block to zeros.
- /name/** Labeled common block. A name can be a maximum of 7 characters and cannot include blank or comma. The first and last characters must not be colons. Conventions imposed by the loader or other assemblers or compilers could further restrict the use of names.
- name** Local block. A name can be 1 through 8 characters, excluding blank or comma. The first character must not be a colon. Use of this name enclosed by brackets does not cause the block to become a labeled common block. For example, USE A and USE/A/ are different blocks.
- \*** Block in use prior to current USE, USELCM, ORG, or ORGC. See discussion following.

A location field symbol, if present, is ignored.

The nominal program block contains the entire program if no USE or USELCM is encountered.

Redundancy between block names is permitted as follows.

A labeled common block designated by /0/ can coexist with the program block designated by 0. Blank common designated by // can coexist with a labeled common block designated as ///.

A CPU subprogram may have two blocks with the same name and the same memory type if they have different block types (local or common). Furthermore, a CPU subprogram may have two blocks with the same name and the same block type if they have different memory types (CM/SCM or ECS/LCM). Thus, altogether, there may be up to four different blocks with the same name.

When a block is first established, its origin and location counters are zero and its position counter is either 60 (CPU subprogram) or 12 (PPU subprogram). When a different block than that in use is indicated, COMPASS saves the values of the current origin and position counters along with an indicator as to whether the next instruction is to be forced upper. If the most recently assembled instruction under the block is one that forces the next instruction upper, the first instruction assembled upon resumption of the block is forced upper. When the designated block has been previously established, COMPASS resumes assembly in the block using the last known values for the origin and position counters. The value of the location counter is not saved. Upon resumption of the block, it is set to the value of the origin counter. If a LOC had been used previously, resetting of the location counter to produce the desired results is the responsibility of the programmer.

The assembler records occurrences of USE, USELCM, ORG, and ORGC pseudo instructions (except USE \* and USELCM \*) and maintains a USE table of the most recent 50 occurrences. Each USE \* and USELCM \* resumes use of the most recent entry and removes it from the table. When the subprogram contains more USE \* or USELCM \* instructions than there are entries in the stack, COMPASS uses the nominal block.

Examples:

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
13	0100000000	GAMMA	USE RJ	ALPHA	BLOCK 0 IN USE
35	17204000000000000000	SAR	USE DATA	DATA1 1.0	BLOCK DATA1 IN USE
14	5130000000		USE SA3	* SAM	RESUME USE OF BLOCK 0

Note that the SA3 is forced upper because the RJ causes a force upper of the next instruction in the block.

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
2615	00		USE VFD USE	TABLE 6/0 *	USE TABLE LOCAL BLOCK RESUME PREVIOUS BLOCK
			.	.	.
			.	.	.
			USE	TABLE	RESUME USING TABLE
	30002600 +		VFD USE	6/1RX,18/S *	RESUME PREVIOUS BLOCK

Note how separate blocks can be used to facilitate packing of partial-word bytes into a table residing in a block other than the one primarily being used.

## 4.5.2 USELCM - ESTABLISH AND USE ECS/LCM BLOCK

The USELCM pseudo instruction establishes or resumes use of a block assigned to extended core storage (ECS) or large core memory (LCM). For all ECS/LCM blocks in an absolute CPU assembly, and for the ECS/LCM blank common block in a relocatable assembly, data generating instructions (including BSSZ) and symbolic machine instructions are illegal; only storage reservation pseudo instructions (BSS, ORG, and ORGC) are allowed. The USELCM pseudo instruction is illegal in PPU assemblies.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	USELCM	block

**block:** Identifies block to be used, as follows:

0 or blank      Illegal.

//              Blank common block. A subprogram can have two blank common blocks if one of them is in ECS/LCM.

/name/          Labeled common block. The name can be a maximum of 7 characters and cannot include blank or comma. The first and last characters must not be colons. The loader or other assemblers or compilers could further restrict the use of names.

name            Local block. † The name can be 1-8 characters, excluding blank or comma. The first character must not be a colon. Use of this name enclosed by brackets does not cause the block to become a labeled common block. For example, A and /A/ are different blocks. All of the local ECS/LCM blocks are concatenated to form a single block, which is treated by the loader as an ECS/LCM common block whose name is unique to the subprogram.

\*                Block in use prior to current USE, USELCM, ORG, or ORGC.

A location field entry, if present, is ignored.

The length of each ECS/LCM block, including the combined local block, is rounded up, if necessary, to an integral multiple of eight 60-bit words. The maximum size of an ECS/LCM block is 1,048,568 words.

Further rules for USELCM are the same as for USE.

---

† SCOPE 2 does not currently allow local blocks in LCM.



Examples:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	BASE	0	
	USELCM	LCM	ESTABLISH AND USE LCM BLOCK
LCMC	BSS	0	DEFINE SYMBOL LCMC
BLOC1	BSS	100	RESERVE 100 WORDS
BLOC2	BSS	200	RESERVE 200 WORDS
	USE	*	RESUME PREVIOUS BLOCK
	.	.	
	.	.	
	ORG	BLOC1+1000B	
BLOC3	BSS	20	RESERVE 20 MORE WORDS
	USE	*	RESUME PREVIOUS BLOCK

#### 4.5.3 ORG AND ORGC - SET ORIGIN COUNTER

ORG indirectly indicates the block to be used for assembly of subsequent code and specifies the value to which the origin and location counters are to be set. COMPASS makes an entry in the USE table and saves the current origin and position counter values.

ORGC† indirectly indicates the block to be used for assembly of subsequent code and specifies the value to which the origin and location counters are to be set. COMPASS makes an entry in the USE table and saves the current origin and position counter values. In a PPU or absolute assembly, ORGC is the same as ORG. In a relocatable CPU assembly, ORGC is the same as ORG if the USE block specified by the address expression is not a common block; otherwise, code following an ORGC is ignored by the linking loader if that common block was first declared by a previously loaded subprogram. If two or more programs in a load sequence preset relocatable text within the same common block, the ORGC must be used; otherwise, multiple relocation of those words can occur.

Formats:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ORG	exp
	ORGC	exp

exp            Expression specifying the address to which the origin and location counters are to be set. Following ORG or ORGC, the assembly resumes at the upper position of the location specified. COMPASS determines the block as follows:

† Not supported by SCOPE 2 Loader.

1. If the expression contains a symbolic address, COMPASS uses the block in which the symbol was defined.
2. COMPASS uses the current block if the value of the expression is \*, \*L, or \*O. If the origin and location counters are the same value, and no code has been assembled in the current location, the only effect of \*, \*L, or \*O is to force the next instruction upper. If a word is partially assembled, however, the code already assembled into the location is lost.  
  
If the counter values differ, \* or \*L sets the origin counter to agree with the location counter value; \*O sets the location counter to the origin counter value.
3. An absolute expression causes use of the absolute block. In a relocatable assembly, this is the only way to establish the absolute block. All symbols defined in the absolute block are absolute.

Any symbols in the expression must be already defined in the assembly and must not result in a negative relocatable value. It is not possible to ORG or ORGC into the literals block.

A location field symbol, if present, is ignored.

Once an ORGC pseudo instruction has established the conditional loading indication for a given common block, it is in effect whenever assembly in that block is resumed by subsequent USE or USELCM pseudo instructions, and can be cleared only by an ORG pseudo instruction specifying that block.

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
	USE	ALPHA	
	.	.	.
	.	.	.
	.	.	.
ABC	DATA	20,100,1000	LOCATED IN ALPHA
	.	.	.
	.	.	.
	USE	BETA	
XYZ	BSS	0	LOCATED IN BETA
	.	.	.
	.	.	.
	.	.	.
	ORG	ABC	SETS ALPHA COUNTERS TO ABC
	.	.	AND RESUMES USE OF ALPHA
	.	.	.
	BSS	1000	
	.	.	.
	.	.	.

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
	ORG	50	SETS ABSOLUTE BLOCK COUNTER TO 50 AND BEGINS ITS USE
	.	.	.
	.	.	.
	ORG	XYZ+100	SETS BETA COUNTERS TO XYZ+100
	.	.	.
	.	.	.
	USE	*	RESUMES ABSOLUTE BLOCK
	.	.	.
	.	.	.
	USE	*	RESUMES BLOCK ALPHA
	.	.	.
	.	.	.
	USE	*	RESUMES BLOCK BETA
	.	.	.
	.	.	.
	USE	*	RESUMES BLOCK ALPHA
	.	.	.
	.	.	.
	USE	*	RESUMES NOMINAL BLOCK
	.	.	.
DATA	USE	/DATA/	
	BSS	0	
	ORGC	DATA	
	DATA	1,2,3	CONDITIONALLY PRESET DATA
	USE	ANYBLOCK	
	CON	3RXYZ	UNCONDITIONAL DATA
	USF	*	
FOUR	DATA	4	RETURN TO /DATA/ STILL
	DATA	5,6	CONDITIONALLY SKIPPING
	ORG	FOUR	
	ZR	X1,ERROR	UNCONDITIONALLY LOADED
	RJ	SUB4	INSTRUCTIONS
	.	.	.
	.	.	.
	.	.	.

#### 4.5.4 BSS—BLOCK STORAGE RESERVATION

The BSS instruction reserves core in the block in use by adjusting the origin and location counters. It does not generate data to be stored in the reserved area. A primary application is for reserving blank common storage. It can also be used to reserve an area to receive replicated code (see REP, REPC, and REPI, section 4.8.8).

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	BSS	aexp

sym            If present, sym is defined as the value of the location counter after the force upper occurs. It is the beginning symbol for the storage area.

aexp           Absolute expression specifying the number of storage words to be reserved. All symbols must be previously defined; aexp cannot contain external symbols. The value of the expression can be negative, zero, or positive and the value is added to both the origin counter and the location counter. A BSS 0 or an erroneous expression causes a force upper and symbol definition but no storage is reserved.

**Example:**

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
COMMON	USE	//	
	BSS	1000B	RESERVE 512 WORDS OF BLANK COMMON
	USE	*	
	.	.	.
	.	.	.
	SA6	COMMON+500R	.
	.	.	.
	.	.	.
TAG	BSS	0	DEFINE SYMBOL TAG
	.	.	.

**4.5.5 LOC — SET LOCATION COUNTER**

A LOC pseudo instruction sets the value of the current location counter to the value in the variable field expression. The location counter is used for assigning address values to location symbols. Changing the location counter permits code to be generated so that it can be loaded at the location controlled by the origin counter and moved and executed at the location controlled by the location counter. Thus, any addresses defined while the location counter is different from the origin counter will be correctly relocated only after the code is moved.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	LOC	exp

exp Relocatable expression specifying the address to which the location counter is to be set. Any symbols in the expression must be already defined in the assembly and must not result in negative relocation.

A location field symbol, if present, is ignored.

Following a LOC, if the value of the location counter differs from the origin counter, the location field is flagged with an L on the listing until a LOC \*O, USE, ORG, ORGC, or USELCM instruction resets the location counter to the value of the origin counter.

A LOC instruction does not affect the origin counter except that it causes the next instruction to be forced upper. The only effect of LOC \* or LOC \*L is to force upper. Because COMPASS does not save the value of the location counter when it switches blocks, a USE, ORG, ORGC, or USELCM for a different block effectively resets the location counter to the origin counter value. When use of the block is resumed, it is the responsibility of the user to reset the location counter to produce the desired results.

Example:

In the following example, the first LOC is used to generate PPU code that is to be loaded into one PPU and transmitted to a different PPU for execution. The second LOC is used so that on the listing the address field contains the table ordinal rather than a load address. At the end of the table, a LOC instruction changes the location counter to resume counting under the first LOC. At the end of the program, LOC \*O returns the location counter to the value of the origin counter.

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
		T1	EQU	1	
		CH	EQU	0	
7100			ORG	7100	
7100		RFS	BSS	0	
L 100			LOC	100	
L 100	2400	PPR	PSN	0	
L 101	2400		PSN	0	
L 102	2400		PSN	0	
L 103	6100 0100		EIM	PPR,CH	
		.	.	.	
		.	.	.	
		.	.	.	
L 205		PPRA	BSS	0	
L 0			LOC	0	
L 0	0100		CON	PPR	
L 1	0114		CON	STM	
L 2	0121		CON	DPM	
L 3	0132		CON	EXR	
L 4	0136		CON	CHS	
L 5	0147		CON	DMP	
L 6	0240		CON	END	
L 7	1000		CON	1000	
		.	.	.	
		.	.	.	
		.	.	.	
L 215			LOC	*0-RES+PPR	
L 215			BSS	240-*	
L 240		END	BSS		
7240			LOC	*0	

### 4.5.6 POS – SET POSITION COUNTER

The POS pseudo instruction sets the value of the position counter for the block in use to the value specified by the expression in the variable field.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	POS	aexp

aexp

An absolute evaluable expression having a positive value less than or equal to the assembly word size (60 for CPU, 12 for PPU). A negative value, or a value greater than 60 (or 12), causes an error. The value indicates the bit position within the current word at which the assembler is to assemble the next code generated. Use caution, because if the new position counter value is greater than the old position counter value, part of the word is reassembled. (New code is ORed with previously assembled data.) If the new position counter value is less than the old position counter value, the assembler generates zero bits to the specified bit position. If the value of aexp is zero, COMPASS assembles the next code in the following word.

A location field symbol, if present, is ignored.

#### NOTE

If the POS instruction is used on a word containing relocatable or external addresses, undefined results can occur with no diagnostics.

The POS instruction does not alter the origin and location counters. The position counter is never 0 at the beginning of an instruction. At the beginning of a new operation, if a data value has been stored into bit 0 (the rightmost bit) of a word, COMPASS increments the origin counter and the location counter and resets the position counter to 60 (or 12).

A POS \*P has no effect whereas a POS \$ subtracts one from the counter.

### 4.6 SYMBOL DEFINITION

The pseudo instructions EQU, =, SET, MAX, MIN, and MICCNT permit direct assignment of 21-bit values to symbols. The values can be absolute, relocatable, or external. Register designators are not valid in the expressions. Subsequent use of the symbol in an expression produces the same result as if the value had been used as a constant. In the listing of the symbolic reference table, a reference to an EQU, =, SET, MAX, MIN, or MICCNT instruction is flagged with a D. Symbols defined using EQU and = cannot be redefined; symbols defined using any of the other symbol definition instructions can be redefined.

#### 4.6.1 EQU OR = - EQUATE SYMBOL VALUE

An EQU or = pseudo instruction permanently defines the symbol in the location field as having the value and attributes indicated by the expression in the variable field.

Formats:

	LOCATION	OPERATION	VARIABLE SUBFIELDS
or	sym	EQU	exp
	sym	=	exp

sym                    A location symbol is required. See section 2.4 for symbol requirements.

exp                    An evaluable expression. Any symbols in the expression must be previously defined or declared as external. The expression cannot contain symbols prefixed by =S, =X, or =Y unless the symbols have also been defined conventionally. If the expression is erroneous, COMPASS does not define the location symbol but flags an error.

Examples:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
20437	OPS	=	20437B	
74	LINP	=	74B	
3	CH	EQU	3	
74	PAGESIZ	=	LINP	
64271	LGOPS	EQJ	*-OPS	

#### 4.6.2 SET - SET OR RESET SYMBOL VALUE

A SET pseudo instruction defines the symbol in the location field as having the value and attributes indicated by the expression in the variable field. A subsequent SET using the same symbol redefines the symbol to the new value and attributes. SET can be used to redefine symbols defined by SET, MAX, MIN, or MICCNT, only.

Format:

	LOCATION	OPERATION	VARIABLE SUBFIELDS
	sym	SET	exp

sym A location symbol is required. See section 2.4 for symbol requirements.

exp An evaluable expression. The expression cannot include symbols as yet undefined and cannot contain symbols prefixed by =S, =X, or =Y, unless the symbols are also defined conventionally.

If the expression is erroneous, COMPASS does not define the symbol but issues a warning flag.

The symbol in the location field cannot be referred to prior to its first definition.

Examples:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
17	A	EQU	15	A HAS VALUE OF 15
74	B	SET	*P	B HAS VALUE OF POSITION COUNTER
22	C	SET	A+3	C HAS VALUE A+3 OR 18
76	B	=	B+2	ILLEGAL, B IS DOUBLY DEFINED
24	C	SET	C+2	LEGAL, C CHANGES FROM 18 TO 20
	D	SET	F+A	ILLEGAL, F AS YET UNDEFINED
		BSS	AA	ILLEGAL, REFERENCE PRECEDES FIRST DEFINITION
20	AA	SET	16	

#### 4.6.3 MAX — SET SYMBOL TO MAXIMUM VALUE

The MAX pseudo instruction defines the symbol in the location field as having the value and attributes indicated by the largest (most positive) value of the expressions in the variable field. A subsequent SET, MAX, MIN, or MICCNT using the same symbol redefines the symbol to the new value. Conversely, MAX can be used to redefine symbols defined by these instructions.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	MAX	exp <sub>1</sub> , exp <sub>2</sub> , ..., exp <sub>n</sub>

sym A location field symbol is required. See section 2.4 for symbol requirements.

exp<sub>i</sub> An evaluable expression. Any symbols in the expression must be previously defined. The expression cannot contain symbols prefixed by =S, or =X, or =Y unless the symbols are also defined conventionally.



The expressions should have similar attributes. No test is made for attributes. The test for maximum value is made in pass one. In testing for the maximum value in pass one, COMPASS uses values for relocatable symbols relative to block origins.

NOTE

During pass two, the expression selected in pass one is used. The relocatable symbols have been reassigned values relative to program origin and these values are used for the final value of the expression selected in the first pass.

If any of the expressions are erroneous, COMPASS does not define the symbol but issues a warning flag. The symbol in the location field cannot be referred to prior to its first definition.

Example:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
5	PT3	EQU	5	
6	PT31	EQU	6	
2	PT32	EQU	2	
6	SYM	MAX	PT3,PT31,PT32	

4.6.4 MIN – SET SYMBOL TO MINIMUM VALUE

A MIN pseudo instruction defines the symbol in the location field as having the value and attributes indicated by the minimum or least positive value of the expressions in the variable field. A subsequent SET, MAX, MIN, or MICCNT using the same symbol redefines the symbol to the new value. Conversely, MIN can be used to redefine symbols defined by these instructions.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	MIN	exp <sub>1</sub> , exp <sub>2</sub> , . . . , exp <sub>n</sub>

sym                    A location symbol is required (section 2.4).

exp<sub>i</sub>                    An evaluable expression. Any symbols in the expression must be previously defined. The expression cannot contain symbols prefixed by =S, =X, or =Y, unless the symbols are also defined conventionally.

The expressions should have similar attributes; no test is made for attributes.

The test for minimum value is made in pass one. In testing for the minimum value in pass one, COMPASS uses values for relocatable symbols relative to block origins.

NOTE

During pass two, the expression selected in pass one is used. The relocatable symbols have been reassigned values relative to program origin and it is these values that are used for the final value of the expression which was selected in the first pass.

If any of the expressions are erroneous, COMPASS does not define the symbol but issues a warning flag.

The symbol in the location field cannot be referred to prior to its first definition.

#### 4.6.5 MICCNT — SET SYMBOL TO MICRO SIZE

The MICCNT pseudo instruction defines the symbol in the location field as having a value equal to the number of characters in the value of the micro named in the variable field. A subsequent SET, MAX, MIN, or MICCNT using the same symbol redefines the symbol to the new value. Conversely, MICCNT can be used to redefine symbols defined by these instructions.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	MICCNT	mname

sym                    A location symbol is required (Section 2.4).

mname                Name of a previously defined micro; it may be a system micro or may have been defined through MICRO, OCTMIC, DECMIC, or BASE. If mname has not been previously defined, the location symbol is not defined (or redefined) and a warning flag is issued.

Example:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
	MSG	MICRO	1,,*STRING*	DEFINE 6-CHARACTER MICRO
			.	.
			.	.
6	MSIZE	MICCNT	MSG	MSIZE EQUALS 6
			.	.
			.	.
	MSG	MICRO	1,,*ALPHANUMERIC #MSG#*	19 CHAR. MICRO
	MSG	MICRO	1,,*ALPHANUMERIC STRING*	19 CHAR. MICRO
23	MSIZE	MICCNT	MSG	MSIZE EQUALS 19

#### 4.6.6 SST — SYSTEM SYMBOL TABLE

An SST pseudo instruction defines system symbols, with the exception of the symbols noted, as if the symbols had been defined in the subprogram.

When a system text overlay is used as input to an assembly through the G or S option on a COMPASS control card, all micros and opcodes in the system text overlay are defined automatically at the start of each assembly; however, the symbols in the system text overlay are defined only for assemblies that contain the SST pseudo instruction.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	SST	sym <sub>1</sub> , sym <sub>2</sub> , ..., sym <sub>n</sub>

sym<sub>i</sub>            One or more symbols on the file that are not to be defined.

A location field symbol, if present, is ignored.

Refer to section 10.2 for an example of SST use.

### 4.7 SUBPROGRAM LINKAGE

Pseudo instructions ENTRY, ENTRYC, and EXT do not define symbols but either declare symbols defined within the subprogram as being available outside the subprogram or declare symbols referred to in the subprogram as being defined outside the subprogram.

#### 4.7.1 ENTRY AND ENTRYC - DECLARE ENTRY SYMBOLS

The ENTRY pseudo instruction specifies which of the symbolic addresses defined in the subprogram can be referred to by subprograms compiled or assembled independently; ENTRY lists entry points to the current subprogram. ENTRY is illegal in PPU assemblies.

The ENTRYC † pseudo instruction conditionally specifies which of the symbolic addresses defined in the subprogram can be referred to by subprograms compiled or assembled independently; ENTRYC lists conditional entry points to the current subprogram. ENTRYC is illegal in PPU assemblies and is synonymous with ENTRY in absolute CPU assemblies. In a relocatable assembly, an entry point symbol declared by ENTRYC is ignored by the linking loader if the value of the symbol is relative to a common block and that common block was first declared by a previously loaded subprogram.

†Not supported by SCOPE 2 Loader.

**Formats:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ENTRY	sym <sub>1</sub> , sym <sub>2</sub> , ..., sym <sub>n</sub>
	ENTRYC	sym <sub>1</sub> , sym <sub>2</sub> , ..., sym <sub>n</sub>

sym<sub>1</sub> Linkage symbol; 1-7 characters of which the first must be alphabetic (A-Z) and the last must not be a colon. The symbol cannot include the following characters:

+ - \* / blank , or ^

Each symbol must be defined in the subprogram as nonexternal (cannot begin with =X or =Y or be listed on an EXT pseudo instruction). Entry point symbols must be unqualified (section 2.4.5).

A location symbol, if present, is ignored.

A list of all entry points declared in the subprogram precedes the assembly listing. An asterisk appears to the right of each conditional entry point.

**Example:**

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
			IDENT	CT, CONTROL, CONTROL	
			ABS		
			ENTRY	MODE	
			ENTRY	ONSW	
			ENTRY	OFFSW	
			ENTRY	ROLLOUT	
			ENTRY	SETPR	
			ENTRY	SETTL	
			ENTRY	SWITCH	
110			ORG	110R	
110		CONTROL	RSS	0	
110	5120000100	MODE	SA2	ACTP	
	73720		SX7	X2	
111	5110000002		SA1	?	
			.	.	
			.	.	
			.	.	

## 4.7.2 EXT — DECLARE EXTERNAL SYMBOLS

The EXT pseudo instruction lists symbols that are defined as entry points in independently compiled or assembled subprograms for which references can appear in the subprogram being assembled. The EXT pseudo instruction is illegal in an absolute subprogram. In a relocatable subprogram, EXT defines symbols as strong externals (section 2.4.1).

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	EXT	sym <sub>1</sub> , sym <sub>2</sub> , ..., sym <sub>n</sub>

sym<sub>i</sub>

Linkage symbol, 1-7 characters of which the first must be alphabetic (A-Z) and the last must not be a colon. The symbol cannot include the following characters;

+ - \* / blank , or ^

These symbols must not be defined within the subprogram. External symbols are unqualified.

A location field symbol, if present, is ignored.

An external reference is flagged with an X in the address field in the listing of code generated. All external symbols are listed in the header information for the assembly listing.

## 4.8 DATA GENERATION

The instructions described in this section are the only pseudo instructions that generate data. All other program data is generated through symbolic machine instructions. An instruction that generates data cannot be used in a blank common block. The pseudo instructions that generate data are:

BSSZ	Generates zeroed words
blank operation field	Generates one zeroed word
DATA	Generates one or more words of data
DIS	Generates one or more words of data
LIT	Generates literals block entries
VFD	Places expression values in user-defined fields
CON	Places expression values in full words
R=	For use in macros; R= assumes that either (B1)=1 or (B7)=1 and generates increment instructions accordingly
REP, REPC, or REPI	Does not actually generate object code at assembly time but causes the relocatable loader to repeatedly load a sequence of code into a reserved blank storage area.

#### 4.8.1 BSSZ AND BLANK OPERATION FIELD—RESERVE ZEROED STORAGE

The BSSZ instruction reserves zeroed core in the block in use. The origin and location counters are adjusted by the requested number of words and the assembler generates data words of zero to be loaded into the reserved area. An instruction that contains a symbol in the location field but has a blank operation field has the same effect as a BSSZ of one word.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	BSSZ	aexp

**sym** If present, sym is defined as the value of the location counter after the force upper occurs. The symbol identifies the beginning of the reserved storage area.

**aexp** Absolute evaluatable expression specifying the number of zeroed words of storage to be reserved. The expression cannot contain external symbols or result in a relocatable or negative value.

A BSSZ 0 or an erroneous expression causes a force upper and symbol definition but no storage is reserved.

A BSSZ or group of BSSZ instructions of six or more words produces an REPL table in object code to reduce the physical size of the object program (appendix B).

For a blank operation field the listing shows one zero word of data; for a BSSZ instruction the listing shows the word count.

#### 4.8.2 DATA — GENERATE DATA WORDS

The DATA pseudo instruction generates one or more complete 60-bit or 12-bit data words in the current block for each item listed in the variable field.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	DATA	item <sub>1</sub> , item <sub>2</sub> , ..., item <sub>n</sub>

**sym** If present, sym is assigned the value of the current location counter after the force upper occurs. It becomes the symbolic address of the first item listed.

item<sub>1</sub> Character, octal numeric, or decimal numeric data item, according to specifications described in section 2.7. Floating point notation is illegal in PPU assemblies. Items are separated by commas and terminated by a blank. A literal cannot be used as an item.

A DATA pseudo instruction always forces upper. A blank item does not cause generation of a data word.

Unless the D list option is selected, only item<sub>1</sub> appears on the listing.

Examples:

<u>Location</u>	<u>Code Generated</u>
552	14071700000000000000
553	40000000000000000000
554	03171520111405000000
555	17252420252400000000
556	00000000000000000000
557	17205146314631463146
560	16403146314631463146

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
OPTB	DATA	0LLGO	
OPT	DATA	1BS59	
OPTT	DATA	0LCOMPILE	
OPTD	DATA	0LOUTPUT,0	
OPTY	DATA	1.3EE	

<u>Location</u>	<u>Code Generated</u>
-----------------	-----------------------

1250	7070
1251	7770
1252	0000
1253	0034
1254	5501
1255	0000
1256	0506
1257	0123
1260	7773
1261	0401
1262	2401

D=0

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	PERIPH		
	BASE	0	
	.	.	
	.	.	
DAT	DATA	7070,-7,0,1R1	
	DATA	2C A,0LEF	
	DATA	0123,-4	
	DATA	H*DATA*	

#### 4.8.3 DIS—GENERATE WORDS OF CHARACTER DATA

The DIS pseudo instruction generates words containing character data. The instruction can be used conveniently when a character data string is to be used repeatedly. Unless the D list option is selected only the first word of character data appears on the listing. The instruction has two formats:

**Format one:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	DIS	n, string

**sym** If present, sym is assigned the location counter value after the force upper occurs. It is the symbolic address of the first word containing the character string.

**n** An absolute evaluable expression specifying an integer number of words to be generated. When base is M, COMPASS assumes that n is decimal.

**string** Character string

For a CPU program, COMPASS takes 10 times n characters from the string and packs them as they occur 10 characters per word into n words. For a PPU program, COMPASS takes two times n characters from the string and packs them as they occur two characters per word into n words. If the statement ends before 10 x n (or 2 x n) characters, the remainder of the requested words are filled with blanks. If n is 0, COMPASS assumes the instruction is in format two.

**Format two:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	DIS	,dstringd

**sym** If present, sym is assigned the location counter value after the force upper occurs. It is the symbolic address of the first word containing the character string.

**d** Delimiting character

**string** Character string; any character other than delimiting character

In this form, the string must be bounded by delimiters. The comma is required. The characters between the two delimiting characters are packed into as many CPU or PPU words as are needed to contain them. Twelve zero bits are guaranteed at the end of the character string even if COMPASS must generate an additional word for them. If COMPASS detects the end of the statement before it detects a second delimiting character, it produces a fatal error.





sym If present, sym is assigned the value of the literals block location counter.

item<sub>i</sub> At least one and not more than 100 words of character, octal numeric, or decimal numeric data items. Section 2.7.3 contains specifications. Items are separated by commas and terminated by a blank. Floating point data items are illegal in PPU assemblies.

COMPASS enters data items into the literals block in the order specified.

If the converted binary values for all the data items listed with a single LIT match an existing literal block sequence, they are not duplicated. If, however, any item in the list does not match an entry in the block, the entire sequence is generated. A literal item subsequently referred to through an = prefix is not duplicated. A null item (e.g. H\*\* or 0L) does not cause a word to be generated.

Examples:

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
	611	POOL	LIT	3.1,1.59265,2.7162182,57.2957795EE1	

CONTENT OF LITERALS BLOCK.

```
000611 17216146314631463146 OQ[-Y-Y-Y-
000612 17206275576441776271 OP]≥.#6;)+
000613 17215337351136014426 OQ#4?I3A9V
000614 17314363651440663121 OY8#*L5vYQ
000615 16513333033540576566 N(00C25.ρv
```

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
	7447	N2	LIT	1R1,7070,7,0	
	7453		LIT	2C A,0LEF	
	7456		LIT	H*LITERALS*	

CONTENT OF LITERALS BLOCK.

```
7447 0034 1
7450 7070 ^^
7451 0007 G
7452 0000
7453 5501 A
7454 0000
7455 0506 EF
7456 1411 LI
7457 2405 TE
7460 2201 RA
7461 1423 LS
```

#### 4.8.5 VFD — VARIABLE FIELD DEFINITION

The VFD instruction generates data in the current block by placing the value of an expression into a field of the specified size.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	VFD	item <sub>1</sub> /exp <sub>1</sub> , item <sub>2</sub> /exp <sub>2</sub> , ..., item <sub>n</sub> /exp <sub>n</sub>

- sym** For a CPU assembly, the location field can contain sym, plus, minus, or blank, as follows:
- sym** If a symbol is provided in the location field, a force upper occurs and the value of the location counter following the force upper is assigned to the symbol. The symbol identifies the first word of data generated by the VFD.
  - +** Causes a force upper. Data generation begins in a new word.
  - COMPASS generates zero bits to the next quarter word boundary, at which point the first field begins.
  - blank** COMPASS begins the first field at the current value of the position counter.

For a PPU assembly, if the location field contains a plus, minus, or a symbol, data generation begins in a new word. If the location field is blank, the first field begins at the current value of the position counter.

- item<sub>i</sub>** An unsigned constant or previously defined symbol having a value specifying a positive integer number of bits for the field to be generated; maximum field size is 60 bits for both CPU and PPU assemblies (60 being the maximum number of significant bits for an expression value). When base is M, item<sub>i</sub> is assumed to be decimal notation.

- exp<sub>i</sub>** An absolute, relocatable, or external expression, the value of which will be inserted into the field specified by item<sub>i</sub>. The expression is evaluated using the specified field size. Character constants are right or left justified in the field according to the type of justification indicated. In a relocatable CPU assembly, no field that contains a relocatable or external address expression can cross a 60-bit word boundary, and no 60-bit word can have more than four fields that contain relocatable or external address expressions.

Each field is generated as it occurs. For a CPU assembly, if the next instruction that generates code in the block is not a VFD with a blank location field, and the last VFD field in the current VFD ends to the left of a quarter word boundary, COMPASS inserts zero bits up to the next quarter word boundary. These zero bits do not show on the assembly listing. Remaining parcels are then filled with no-operation instructions.

When a VFD instruction that does not have a location field entry immediately follows another VFD in the same block, no padding with zeros or forcing upper occurs; fields are generated sequentially as they are specified.

Following a VFD, the position counter contains the number of bits remaining to be assembled in the last word in which data was generated by the VFD.

**Examples:**

In the first example, the symbol TABLOC has been defined earlier in the program and associated with 000551.

<u>Location</u>	<u>Code Generated</u>	
		31
566	24010200000023000551	
567	00000005665555555555	
570	777777774	
		000000000000
571	11172401550155555531	
572	00000015052323010705	
573	0311170000000033	

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
ALPHA	SFT	25	
TABLE	VFD	36/3CTAB,6/19,18/TABLOC	
	VFD	30/*-1,30/5H	,ALPHA/-0
	VFD	*P/	
	VFD	30/0HIOTA,6/1RA,24/0AX+1	
	VFD	60/0RMESAGE,30/3LCIO,15/0R0	

<u>Location</u>	<u>Code Generated</u>	
		OEM
1310	3334	
1311	3536	
1312	3740	
1313	4142	
1314	4344	
1315	0010	
1316	0011	
1317	7765	
1320	0707	

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	PPU	M	
N4	BASE		
	VFD	60/10R0123456789	
A11	VFD	12/10,12/11,12/-12,12/-7070	

**4.8.6 CON – GENERATE CONSTANTS**

The CON pseudo instruction generates one or ore full words of binary data in the block in use. It differs from DATA in that it generates expression values rather than data items and differs from VFD in that the field size is fixed. For relocatable or external addresses, CON generates a 60-bit address constant that is not appropriate for routines built into capsules or overlay capsules.

**Format**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	CON	exp <sub>1</sub> , exp <sub>2</sub> , ..., exp <sub>n</sub>

sym If present, sym is assigned the value of the location counter after the force upper occurs.

exp<sub>i</sub> An absolute, relocatable, or external expression the value of which will be inserted into a field having a size of one word. For PPU assembly, floating point is not allowed; for CPU assembly, double precision is not allowed.

**Examples:**

In the first example, the symbols FAIL and PASS have been defined earlier in the program and associated with 2204 and 2172, respectively.

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
1460	0000	MSG1	CON	5	
1461	0006		CON	6	
1462	0003		CON	3	
1463	2204		CON	FAIL	
1464	0024		CON	20	
1465	0000	MSG2	CON	0	
1466	0006		CON	6	
1467	0003		CON	3	
1470	2172		CON	PASS	
1471	0024		CON	20	

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
574		TAD	PSS	0	
L 0			LOC	0	
L 0	00000000000000000000000055		CON	1R	00
L 1	00000000000000000000000062		CON	1R1	01
L 2	00000000000000000000000064		CON	1R2	02
L 3	00000000000000000000000060		CON	1R3	03
			.	.	.
			.	.	.
			.	.	.
L 75	00000000000000000000000066		CON	1Rv	75
L 76	00000000000000000000000076		CON	1Pv	76
L 77	00000000000000000000000055		CON	1P	77
674			LOC	*0	

**4.8.7 R= — CONDITIONAL INCREMENT INSTRUCTION**

The R= pseudo instruction generates a CPU increment unit instruction depending on the contents of the variable subfields and on whether or not the subprogram earlier contained a B1=1 or B7=1 pseudo instruction (section 4.4.4).

Use of R= augments macro definitions and increases optimization of object code. It is illegal in a PPU program.

The A list option controls listing of substituted instructions.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	R=	reg,exp

sym                      Optional, if present, sym is assigned the value of the location counter after the force upper occurs. This force upper occurs whether the R= generates an instruction or not.

reg A register designator (A, X, or B) and a digit (0-7) which COMPASS concatenates with S to form the instruction operation code.

exp Operand register or value expression. If exp is the same two characters as reg, no instruction is generated.

If the expression value is 0, the variable field is B0.

If the B1=1 instruction has been assembled prior to this instruction and the expression value is 1, 2, or -1, the variable field of the instruction is B1, B1+B1, or -B1, respectively.

If the B7=1 instruction has been assembled prior to this instruction and the expression value is 1, 2, or -1, the variable field for the instruction is B7, B7+B7, or -B7, respectively.

In all other cases, the variable field is the register or value indicated by the expression.

Examples:

1. R= used with B1=1

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
		P1=1		
		P=	13,2	
66311		S33	R1+R1	
		P=	13,2	
6130000003		S33	3	

2. R= used with B1≠1

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
		TAG	P=	X5,-1
7150777776		TAG	SX5	-1

3. Expression is same as register designator:

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
RFG	MICRO	1, *R5*	
	R=	R5, #REG#	
	R=	B5, B5	

No instruction is generated; SB5 B5 would be a no operation instruction.

#### 4.8.8 REP, REPC, AND REPI - GENERATE LOADER REPLICATION TABLE

The REP, REPC, and REPI instructions cause the assembler to generate an REPL loader table so that when the subprogram being assembled is loaded, the loader will load one or more copies of a data sequence. For the REPI instruction, the loader generates the copies immediately upon encountering the table; for REP, the replication takes place at the end of loading. For REPC<sup>†</sup> the loader ignores the REPL table if the destination data address is in a common block that was first declared by a previously loaded subprogram; otherwise, the loader generates the copies immediately upon encountering the tables.

Replication of object code is valid in relocatable assemblies only. It is particularly useful for setting one or more blocks of storage to a given series of values or for generating tables.

Data to be replicated must not contain any external references or common block relocatable addresses. For REPC and REPI, data must be in previously assembled text.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	REP REPC <sup>†</sup> REPI	S/saddr, D/daddr, C/rep, B/bsz, I/inc

A location field symbol, if present, is ignored.

The variable field subfields can be in any order.

S/saddr Relocatable expression specifying first word address of code to be copied. The S/saddr subfield must be provided. If it is zero, or omitted, the assembler flags the instruction as erroneous and does not generate an REPL loader table.

D/daddr Relocatable expression specifying the destination of the first word of the first copy. If D/daddr is omitted, the assembler sets daddr to zero, and, when daddr is zero, the loader uses saddr plus bsz for the destination address.

Note that room for the repeated data must be reserved in the destination block.

<sup>†</sup> Not supported by SCOPE 2 Loader.

C/rep Absolute expression specifying the number of times code is to be copied. When base is M, COMPASS assumes that rep is a decimal value. If C/rep is omitted, the assembler sets rep to zero. When rep is zero or one, the loader makes one copy.

B/bsz Absolute expression specifying the number of words to be copied (block size). When base is M, COMPASS assumes that bsz is decimal.  
 If B/bsz is omitted, the assembler sets bsz to zero. When bsz is zero or one, the loader copies one word.

I/inc Absolute expression specifying the increment size in words. When base is M, COMPASS assumes that inc is in decimal.  
 The increment size is the number of words between the first word of each copy. When inc is zero or omitted, the loader uses bsz as the increment size. The loader writes the first copy starting at daddr, the second starting at daddr+inc, the third at daddr + 2 x inc, etc. until the rep count is exhausted.

The origin and location counters for the block containing the daddr are not advanced by a value of inc x rep. Storage reservation for replicated code is the responsibility of the user.

Rules for replication:

1. The S subfield cannot be omitted
2. Room must be reserved for the copies in the destination block (for example, through ORG, ORGC, or BSS)
3. REP, REPC, and REPI can be used in relocatable assemblies only
4. Data to be replicated must not contain any external references or common block relocatable addresses
5. For REPC and REPI, data must be in previously loaded text

Example:

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	10	PC	=	10	
5017	0000000000000000000015	RA	USE	NEWP	
5020	0000000000000000000020		DATA	15,20,70700,1,5,3.14	
5021	000000000000000000007070				
5022	000000000000000000000001				
5023	000000000000000000000005				
5024	1721630000000000000000				
	13	I	EQU	*-RA+5	
			USE	ORLOCK	
5251		DA	RSS	PC*I	
			USE	*	
			REPI	S/RA,D/DA,P/I-5,C/RC,I/I	



## 4.9 CONDITIONAL ASSEMBLY

The following pseudo instructions permit optional assembly or skipping of source code. A special form, SKIP, causes unconditional skipping. COMPASS provides IF test instructions that:

- Test for assembly environment (IFtype)
- Compare values of two expressions (IFop)
- Compare values of two character strings (IFC)
- Test the attribute of a single symbol or an expression (IF)
- Test the sign of an expression (IFPL and IFMI)

Immediately following the test instruction are instructions that are assembled when the tested condition is true and skipped when the condition is false. Skipping is terminated either by a source statement count on the IF instruction, or by an ENDIF, an ELSE, or an END.

The statement count, when used, is decremented for instruction lines only; comment lines (identified by \* in column one) are not counted. Determining the IF range with a statement count produces slightly faster assembly than using the ENDIF.

The results of an IF test are determined by the values of expressions in pass one; the value of a relocatable symbol is relative to the USE block in which it was defined. The value of an external symbol is 0 if the symbol was declared as external. If the symbol was defined relative to a declared external, the value is the relative value.

### 4.9.1 ENDIF — END OF IF RANGE

An ENDIF causes skipping to terminate and assembly to resume. When the sequence containing the ENDIF is being assembled, or is controlled by a statement count, the ENDIF has no effect other than to be included in the count.

Skipped instructions such as macro references are not expanded. Thus, any ENDIF that would have resulted from an expansion is not detected.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	ENDIF	

ifname                      Name of an IF, SKIP, or ELSE sequence; or blank. ifname can be used as any other type of symbol elsewhere in the program.

Skipping of a sequence initiated by an IF, SKIP, or ELSE that is assigned a name can be terminated by an ENDIF specifying the sequence by name, or by any unnamed ENDIF. Any ENDIF terminates skipping of an unnamed sequence that is not controlled by a source line count. A named ENDIF terminates the named IF, SKIP, or ELSE and any unnamed IF, SKIP, or ELSE sequences in effect that are not under line count control.

#### 4.9.2 ELSE — REVERSE EFFECTS OF IF

Through the ELSE instruction, COMPASS provides the facility to reverse the effects of an IF test within the IF range. An ELSE detected during skipping causes assembly to resume at the instruction following the ELSE. An ELSE detected while a sequence is being assembled initiates skipping of source code following the ELSE. Skipping continues until:

1. A statement count specified on the ELSE is exhausted
2. A second ELSE is detected for the sequence
3. An ENDIF is detected for the sequence

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	ELSE	nct

**ifname** Name of an IF, SKIP, or ELSE sequence, or blank.

**nct** Optional absolute evaluable expression specifying integer number of source lines to be skipped. It has no effect if the ELSE resumes assembly. When the base is M, COMPASS assumes that nct is decimal.

An ELSE specifying the sequence by name or any unnamed ELSE terminates skipping of a sequence initiated by an IF, SKIP, or an ELSE that has an assigned name. Skipped instructions such as macro references are not expanded; any ELSE that would have resulted from the expansion is not detected.

#### 4.9.3 IFtype - TEST OBJECT PROCESSOR TYPE

IFtype pseudo instructions test for the type of processor that will execute the object program, as declared by MACHINE, and PERIPH or PPU pseudo instructions.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	IFtype	nct

**ifname** Optional 1-8 character name.

**type** Mnemonic specifying type of object processor.

<u>Type</u>	<u>Condition Causing Assembly</u>
CP	Any central processor unit
CP6	Neither PERIPH nor PPU nor MACHINE 7 has been specified. CPU code is assembled for a CYBER 180 Series, a CYBER 170 Series, CYBER 70/Model 71, 72, 73, or 74, or 6000 Series Computer System.
CP7	Neither PERIPH nor PPU nor MACHINE 6 has been specified. That is, CPU code is assembled for a CYBER 70/Model 76 or a 7600 Computer System.
PP	Any peripheral processor unit
PP6	One of the following is true: <ol style="list-style-type: none"><li>1. PERIPH has been specified but MACHINE 7 has not been specified.</li><li>2. PPU and MACHINE 6 have both been specified. PPU code is assembled for a CYBER 180 Series, a CYBER 170 Series, CYBER 70/Model 71, 72, 73, or 74, or a 6000 Series Computer System.</li></ol>
PP7	One of the following is true: <ol style="list-style-type: none"><li>1. PPU has been specified but MACHINE 6 has not been specified.</li><li>2. PERIPH and MACHINE 7 have both been specified. That is, PPU code is assembled for a CYBER 70/Model 76 or a 7600 Computer System.</li></ol>

**lnct** Optional absolute evaluable expression specifying an integer count of the number of statements to be skipped. When base is M, COMPASS assumes that **lnct** is decimal.

The **ifname** and **lnct** parameters are related as follows:

1. If a count is supplied, it takes precedence over any ENDIF but not over an ELSE. The only effect of an ENDIF in a count controlled sequence is to be included in the count. Skipping terminates when the count is exhausted or when an ELSE with a matching or blank name is encountered, whichever occurs first.
2. If neither a count nor a name is supplied, the IF range is terminated by an ENDIF, whether named or unnamed, or by a unnamed ELSE, whichever is encountered first. A named ELSE has no effect.

- If a name but no count is supplied, the IF range is terminated by an ENDIF or ELSE with a matching name or by an unnamed ENDIF or ELSE. An ENDIF or ELSE with a name that does not match has no effect.

Example:

Code Generated

```
0
173 0130000000
```

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	IDENT	XYZ	
	MACHINE	6	
	.		
	.		
	BSS	123	
	IFCP6	2	
	XJ	0	
	ELSE	1	
	MJ	0	

#### 4.9.4 IFOP - COMPARE EXPRESSION VALUES

An IFop pseudo instruction compares the values of two expressions according to the relational mnemonic specified and assembles instructions in the IF range when the comparison is satisfied.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	IFop	exp <sub>1</sub> , exp <sub>2</sub> , fct

ifname            Optional 1-8 character name

op                Specifies comparative test:

op                Condition causing assembly

EQ                Equality, the expressions are equal in all respects. That is, they not only have the same numeric value but have the same attributes as well. For example, both are names that are common relocatable, or absolute, or external, etc.

NE                Inequality, the expressions are not equal in all respects. They differ in value or in some attribute.

GT                The first expression is greater in value than the second expression. No other attributes are tested.

- GE            The first expression is greater than or equal in value to the second expression. No other attributes are tested.
  - LT            The first expression is less in value than the second expression. No other attributes are tested.
  - LE            The first expression is less than or equal in value to the second expression. No other attributes are tested.
- For these tests, positive zero and negative zero are equal.

*exp<sub>i</sub>*            An expression. When the value of *exp* is tested, *exp* can include only previously defined symbols and the result can be absolute, relocatable, or external. If an undefined symbol is used, the expression value is set to zero, the IF instruction is flagged as erroneous, and assembly continues with the next instruction.

*nct*            Optional absolute evaluable expression specifying an integer count of the number of statements to be skipped. When base is M, COMPASS assumes that *nct* is decimal. When *nct* is blank, the comma can be omitted.

The *ifname* and *nct* parameters are related as follows:

1. If a count is supplied, it takes precedence over any ENDIF but not over an ELSE. The only effect of an ENDIF in a count controlled sequence is to be included in the count. Skipping terminates when the count is exhausted or when an ELSE with a matching or blank name is encountered, whichever occurs first.
2. If neither a count nor a name is supplied, the IF range is terminated by an ENDIF, whether named or unnamed, or by an unnamed ELSE, whichever is encountered first. A named ELSE has no effect.
3. If a name but no count is supplied, the IF range is terminated by an ENDIF or ELSE with a matching name or by an unnamed ENDIF or ELSE. An ENDIF or ELSE with a name that does not match has no effect.

Example:

A demonstration of one use of IF statements in a PPU program:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	*	11	18
			30
	IF	DEF, LOOP	
	IFLT	*-LOOP, 40B	
	ZJN	LOOP	
	ELSE	2	
	NJN	*+3	
	LJM	LOOP	
	.		
	.		
	.		

This code assembles a zero jump to the symbol LOOP if LOOP has been defined within 37<sub>8</sub> words (the range of a short jump) prior to the occurrence of this code. Otherwise, the NJN and LJM are assembled.

#### 4.9.5 IFPL AND IFMI – TEST SIGN OF EXPRESSION

The IFPL and IFMI pseudo instructions test the sign of an expression and assemble instructions in the IF range according to whether the sign of the value is plus (PL) or minus (MI). The pseudo instructions allow positive zero to be distinguished from negative zero.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	IFPL	exp, <i>fnct</i>
ifname	IFMI	exp, <i>fnct</i>

**ifname**            Optional 1-8 character name

**exp**                An expression. It can include only previously defined symbols and the result can be absolute, relocatable, or external. If an undefined symbol is used, the instruction is flagged as erroneous and assembly continues with the next instruction.

***fnct***              Optional absolute expression specifying an integer count of the number of statements to be skipped. When base is M, COMPASS assumes that *fnct* is decimal. When *fnct* is blank, the comma can be omitted.

The ifname and *fnct* parameters are related as follows:

1. If a count is supplied, it takes precedence over any ENDIF but not over an ELSE. The only effect of an ENDIF in a count controlled sequence is to be included in the count. Skipping terminates when the count is exhausted or when an ELSE with a matching or blank name is encountered, whichever occurs first.
2. If neither a count nor a name is supplied, the IF range is terminated by an ENDIF, whether named or unnamed, or by an unnamed ELSE, whichever is encountered first. A named ELSE has no effect.
3. If a name but no count is supplied, the IF range is terminated by an ENDIF or ELSE with a matching name or by an unnamed ENDIF or ELSE. An ENDIF or ELSE with a name that does not match has no effect.

The condition tested for by IFPL is satisfied if the value of exp is greater than or equal to plus zero; the condition for IFMI is satisfied if the value of exp is less than or equal to minus zero.

Example:

The following opdef defines the CPU instruction MXi jk so that the address value is 60 if the expression value is negative zero or a positive non-zero multiple of 60, otherwise it is the address expression value modulo 60.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
MXQ	OPDEF	REG, VAL	
A	LOCAL	A	
A	SET	VAL	
A	SET	A-A/60D*60D	
	IFPL	A, 3	
	IFEQ	A, 0, 3	
	IFLE	VAL, 0, 1	
	SKIP	1	
A	SET	A+60D	
	VFD	6/43B, 3/REG, 5/A	
	ENDM		

Example of call:

Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
	7777713	↑↑000001	MX6	-52	
			SET	-52	
	7777713	↑↑000001	SET	↑↑000001-↑↑000001/60D*60D	
			IFPL	↑↑000001, 3	
			IFEQ	↑↑000001, 0, 3	
			IFLE	-52, 0, 1	
			SKIP	1	
43610	10	↑↑000001	SET	↑↑000001+60D	
			VFD	6/43B, 3/6, 6/↑↑000001	
			ENDM		

#### 4.9.6 IF - TEST SYMBOL OR EXPRESSION ATTRIBUTE

The IF pseudo instruction tests a symbol or an expression for a specific attribute and assembles instructions in the IF range if the test is satisfied.





DEF	All the symbols in the expression in the second subfield are defined
-DEF	One or more of the symbols in the expression in the second subfield is undefined
MAC	The name in the second subfield is an opcode name
-MAC	The name in the second subfield does not contain an opcode name
MIC	The name in the second subfield is a micro
-MIC	The second subfield does not contain a micro name
SST	The second subfield contains a system symbol
-SST	The second subfield does not contain a system symbol

**exp** For SET, SST, -SET, and -SST, exp must be a single defined symbol. For MIC and -MIC, exp must be a name. For any other test, it is an expression. The expression can include symbols as yet undefined if att is DEF, -DEF, REG, -REG, EXT, or -EXT only. If an undefined symbol is used with any other attribute, the expression value is set to zero, the instruction is flagged as erroneous, and assembly continues with the next instruction. Note that if a symbol is never defined conventionally but only by use of =S or =X prefix (see section 2.4.2), COMPASS does not define the symbol until the end of the assembly, and IF tests will consider the symbol undefined.

**∫nct** Optional absolute evaluatable expression specifying an integer count of the number of statements to be skipped. When base is M, COMPASS assumes that ∫nct is decimal. When ∫nct is blank, the comma can be omitted.

The ifname and ∫nct parameters are related as follows:

1. If a count is supplied, it takes precedence over any ENDIF but not over an ELSE. The only effect of an ENDIF in a count controlled sequence is to be included in the count. Skipping terminates when the count is exhausted or when an ELSE with a matching or blank name is encountered, whichever occurs first.
2. If neither a count nor a name is supplied, the IF range is terminated by an ENDIF, whether named or unnamed, or by an unnamed ELSE, whichever is encountered first. A named ELSE has no effect.
3. If a name but no count is supplied, the IF range is terminated by an ENDIF or ELSE with a matching name or by an unnamed ENDIF or ELSE. An ENDIF or ELSE with a name that does not match has no effect.



GE or -LT string<sub>1</sub> is greater than or equal to string<sub>2</sub>  
 LT or -GE string<sub>1</sub> is less than string<sub>2</sub>  
 LE or -GT string<sub>1</sub> is less than or equal to string<sub>2</sub>

string<sub>i</sub> Character string. When IFC is within a macro definition, each character string can be a formal parameter.

¶nct Optional absolute evaluatable expression specifying an integer count of the number of statements to be skipped. When base is M, COMPASS assumes that ¶nct is decimal. When ¶nct is blank, the comma can be omitted.

The ifname and ¶nct parameters are related as follows:

1. If a count is supplied, it takes precedence over any ENDIF but not over an ELSE. The only effect of an ENDIF in a count controlled sequence is to be included in the count. Skipping terminates when the count is exhausted or when an ELSE with a matching or blank name is encountered, whichever occurs first.
2. If neither a count nor a name is supplied, the IF range is terminated by an ENDIF, whether named or unnamed, or by an unnamed ELSE, whichever is encountered first. A named ELSE has no effect.
3. If a name but no count is supplied, the IF range is terminated by an ENDIF or ELSE with a matching name or by an unnamed ENDIF or ELSE. An ENDIF or ELSE with a name that does not match has no effect.

Each character in string<sub>1</sub> is compared with the corresponding character in string<sub>2</sub> progressing from left to right until an inequality is found or both strings are exhausted. When one string is shorter than the other, it is padded with a character that has a value less than any other character in the string.

The truth condition is based on the relative magnitudes of the characters in the strings.

Examples:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
TEST1	IFC	EQ,\$ABC\$ABC\$	ABC EQUALS ABC
TEST2	IFC	LT,*AB*ABC*	AB IS LESS THAN ABC
TEST3	IFC	GT,XAXX	A IS GREATER THAN NULL
	IFC	-GE,*Z*8*,3	Z IS LESS THAN 8

The IFC in the following example checks for an empty parameter string.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
XX	MACRO	P1,P2	
P	IFC	EQ,**P2*,1	
	ERR		FLAG EPROR
	.		
	.		
	.		
	ENDM		

The following example illustrates a character string terminated incorrectly. When COMPASS reaches end of statement without finding a third asterisk, the asterisk omitted following P1 causes an error flag.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	IFC	EQ,*00*P1,2\$P2	

#### 4.9.8 SKIP — UNCONDITIONALLY SKIP CODE

The SKIP instruction causes COMPASS to unconditionally skip the instructions in the SKIP range. It resembles an IF for which there is no true condition.

Format

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	SKIP	nct

ifname            Optional 1-8 character name

nct                Optional absolute evaluable expression specifying an integer count of the number of statements to be skipped. When base is M, COMPASS assumes that nct is decimal.

The ifname and nct parameters are related as follows:

1. If a count is supplied, it takes precedence over any ENDIF but not over an ELSE. The only effect of an ENDIF in a count controlled sequence is to be included in the count. Skipping terminates when the count is exhausted or when an ELSE with a matching or blank name is encountered, whichever occurs first.

2. If neither a count nor a name is supplied, the SKIP range is terminated by an ENDIF, whether named or unnamed, or by an unnamed ELSE, whichever is encountered first. A named ELSE has no effect.
3. If a name but no count is supplied, the SKIP range is terminated by an ENDIF or ELSE with a matching name or by an unnamed ENDIF or ELSE. An ENDIF or ELSE with a name that does not match has no effect.

## 4.10 ERROR CONTROL

The ERR and ERRxx pseudo instructions described in this section either conditionally or unconditionally set an error flag.

### 4.10.1 ERR — UNCONDITIONALLY SET ERROR FLAG

An ERR pseudo instruction produces an assembly error but does not affect other code. Usually, it is used in conjunction with a conditional assembly pseudo instruction to force an error into the assembly based on an assembly time test. One application is to use a test and ERR to detect illegal macro parameters.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
flag	ERR	

flag                      A single alphanumeric character denoting the error type. The flag is placed in the listing to the left of the line for ERR. The flag can denote a fatal or nonfatal error. A fatal error causes COMPASS to suppress generation of the binary deck unless the D mode option is selected on the COMPASS control card. If no flag is specified, or the character is not one of those given in section 11.7, COMPASS uses P.

A variable field entry, if present, is ignored.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
i	ii	iii	30
NNN	MACRO	P1,P2,P3,P4	
A	IFEQ	P1,0	
	ERR		
	.	.	
	.	.	
	.	.	
	ENDM		
	.	.	
	.	.	
	.	.	
	NNN	0,A,B,C	

## 4.10.2 ERRxx — CONDITIONALLY SET ERROR FLAG

An ERRxx pseudo instruction produces an assembly error when a condition detected during the second pass of the assembler is true.

### Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
flag	ERRxx	aexp

**flag** A single alphanumeric character denoting the error type. The flag is placed in the listing to the left of the line for ERR. The flag can denote a fatal or nonfatal error. A fatal error causes COMPASS to suppress generation of the binary deck unless the D mode option is selected on the COMPASS control card. If no flag is specified, or the character is not one of those given in section 11.7, COMPASS uses P.

**xx** Defines condition under which aexp value is erroneous.

<u>xx</u>	<u>Error Condition</u>
NG or MI	Value of expression is negative
NZ	Value of expression is nonzero
PL	Value of expression is positive
ZR	Value of expression is zero

**aexp** Absolute expression. It cannot contain external symbols or references to blank common. The test is made in pass two of the assembler. Relocatable addresses are assigned values relative to program origin rather than to the block in which they are defined.

### NOTE

ERRxx is the only conditional instruction for which the test is made in pass two. Therefore, this is the only pseudo instruction that can be used to determine PPU overflow if the PPU program has literals and USE blocks.

### Example:

Test for memory overflow in PPU assembly

<u>Location</u>	<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
7447	7777447	LASTTAG	PERIPH		
7462		R	BSS ERRPL FND	0 LASTTAG-7777	

## 4.11 LISTING CONTROL

The instructions described in this section permit extensive control of the assembly listing format.

### 4.11.1 LIST — SELECT LIST OPTIONS

The LIST pseudo instruction controls the content and format of the assembler listing. LIST instructions are disabled under either of the following conditions:

When the list parameter (L) on the COMPASS control statement (chapter 10) is zero, or

When the list option parameter (LO) on the COMPASS control statement is used and is other than LO=0.

Use of the LIST pseudo instruction is optional. If it is not used in the subprogram, COMPASS list output is according to the L and LO parameters on the COMPASS control statement. If the LO parameter is omitted or LO=0, the list options are as if L, B, N, and R only are selected and the listing contains heading information, assembly text, assembler statistics, an error directory (upon occurrence of an error only), and a symbolic reference table. Formats of this output are described in detail in chapter 11 and brief summaries are given below.

Heading Information	Program length, origin, and length of each block, entry points and external symbols.
Assembly text	Line, and assembly results of each line assembled (not skipped) from the input device (excludes code generated by RMT, DUP, ECHO, XTEXT, or a macro or opdef expansion). For data generating pseudo instructions DATA, DIS, BSSZ that produce more than one word of object code, only the first word is listed. For VFD and CON all words of object code are listed. For R=, only the pseudo instruction listed.  Each occurrence of the LIST instruction is listed.
Assembler statistics	Amount of storage used, counts of assembled statements, defined symbols, invented symbols, and references to symbols.
Error directory	List of fatal and nonfatal errors and summary of the causes of each.
Symbolic reference table	List of all symbols defined in the program according to symbol qualifier, if any, followed by an index to every reference to the symbol, whether in explicit or generated (for example, by MACRO or MICRO calls) statements.

#### Formats:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	LIST or LIST	op <sub>1</sub> , op <sub>2</sub> , . . . , op <sub>n</sub> *

A location field symbol, if present, is ignored.

op<sub>i</sub>

A list option represented by a single letter or a letter prefixed by a minus sign. The unprefixed letter selects the option; the prefixed letter cancels the option. Options are separated by commas and terminated by a blank.

**A List statements actually assembled**

When A is not selected, a line containing concatenation and micro substitution marks is listed with the marks in it exactly as presented to the assembler. When the A option is selected, however, the assembler lists the line before and after the editing takes place. Selecting A also causes the listing of lines of code resulting from the R= pseudo instruction.

**B List binary control statements**

When B is selected, the listing includes SEG, SEGMENT, IDENT, and END pseudo instructions.

**C List listing control statements**

When C is selected, the listing includes EJECT, SPACE, TTL, and TITLE pseudo instructions. A listing instruction that causes an EJECT is listed as the first line of the new page after the EJECT takes place

**D Include details**

Selection of the D option causes listing of the following items not normally listed:

- Second and subsequent lines of DATA and DIS
- Code assembled remotely when HERE or END causes its assembly
- Literals block
- Default symbols

**E Include echoed lines**

Selection of E causes listing of all iterations of code duplicated as a result of DUP and ECHO.

**F List IF-skipped lines**

When F is selected, the listing includes all lines skipped by IF, IFop, IFC, IFPP, IFCP, SKIP, and ELSE. In addition, the Symbolic Reference Table contains references to symbols in IF statements.

**G List generated code**

Selection of this option causes listing of all code generating lines regardless of list controls other than L. Instructions listed include symbolic machine instructions and BSS, BSSZ, CON, DATA, DIS, R=, and VFD.

**L Master list control**

This option is normally selected. When L is canceled, the long list contains error flagged lines, an error directory, and LIST and END pseudo instructions only, regardless of selection of any other options on LIST.

**M List macros and opdefs**

Selection of M causes all lines generated by calls to macros and opdefs other than those defined by the system to be listed.



- N List nonreferenced symbols**  
 This option is normally selected. Cancellation of this option causes any nonsystem symbol for which no reference has been accumulated (e.g., all occurrences are in IF statements with the F option deselected, or are between CTEXT or ENDX with the X option deselected) to be omitted from the symbolic reference table.
- R Accumulate and List references**  
 This option is normally selected. When R is canceled, COMPASS does not accumulate references. R should not be canceled if a complete symbolic reference table is desired. If R is canceled at the end of assembly, no symbolic reference table is produced.
- S List systems macros and opdefs**  
 Selection of S causes all lines generated by calls to systems-defined macros and opdefs to be listed.
- T List nonreferenced system symbols**  
 Selection of this option causes a symbol defined through SST to be included in the symbolic reference table even if there are no accumulated references.
- X List XTEXT lines**  
 Selection of the X option causes listing of all statements assembled as a result of an XTEXT pseudo instruction. CTEXT and ENDX provide a means of alternately turning this external designator off and on.

\$

A dollar sign in the variable field selects all options.

\*

An asterisk in the variable field causes selection of the options in effect prior to the current selection. The assembler records occurrences of LIST pseudo instructions and maintains a table of the most recent 50 occurrences. Each LIST \* resumes use of the most recent entry and removes it from the list. When the subprogram contains more LIST \* instructions than there are entries in the stack, COMPASS selects the default list options (B, L, N, and R).

For list options A, C, D, E, F, M, S, and X, all applicable options must be selected for a specific line to be listed. For example, listing of an expansion resulting from a DUP within a macro requires selection of both M and E. Similarly, an expansion caused by an XTEXT within a system macro call is listed only when both X and S are selected. To obtain a listing showing  $\bar{\Gamma}$  and  $\neq$  marks removed from external text inside a DUP range, A, X, and E must all be selected.

Example:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
		LIST	A	
0	17205146314631463146	DATA	1.3rEE	
		DATA	1.3EE	
		LIST	D	
2	17205146314631463146	DATA	1.3rEE	
3	16403146314631463146	DATA	1.3EE	
		LIST	-A,-D	
4	17205146314631463146	DATA	1.3rEE	
		LIST	*	
6	17205146314631463146	DATA	1.3rEE $\neq$	
7	16403146314631463146	DATA	1.3EE	

#### 4.11.2 EJECT—EJECT PAGE AND BEGIN NEW SUB-SUBTITLE

The EJECT pseudo instruction advances printer paper to a new page before printing. Then, page headings are printed and listing continues. EJECT has no effect, other than setting the sub-subtitle, if it is generated by DUP, ECHO, RMT, XTEXT, or a macro or opdef expansion, and the corresponding LIST options are not all selected.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	EJECT	

**name** New program sub-subtitle for the page will be printed in character positions 70-79 of the second line of the page. A blank name clears the sub-subtitle.

An entry in the variable field, if present, is ignored.

#### 4.11.3 SPACE — SKIP LINES AND BEGIN NEW SUB-SUBTITLE

The SPACE pseudo instruction spaces the assembler listing. When a page is full, an eject occurs and listing resumes on the next page. A SPACE immediately following an EJECT is ignored. SPACE has no effect, other than setting the sub-subtitle, if it is generated by a DUP, ECHO, RMT, XTEXT, or a macro or opdef expansion, and the corresponding LIST options are not all selected.

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	SPACE	scent, rcnt

**name** New subprogram sub-subtitle will be printed in characters 70-79 on the second line of the next page heading. A blank name clears the sub-subtitle.

**scent** An absolute expression specifying a positive integer number of spaces between the most recent line and the next line of printout. If base is M, scent is assumed to be decimal. If scent is omitted or zero, no line is skipped.

**rcnt** An absolute expression specifying a positive integer number of lines that must be remaining on the page following spacing. If base is M, rcnt is assumed to be decimal.

If  $scent + rcnt$  exceeds the number of lines on the page before spacing occurs, the SPACE acts like an EJECT. Note that either the eject occurs or the number of spaces are skipped but not both.

Blank cards or statements can also be used to space the listing.

#### 4.11.4 TITLE — ASSEMBLY LISTING TITLE

The first TITLE pseudo instruction establishes the title that will be printed on each page of the listing. A subsequent TITLE instruction generates a subtitle and causes a page eject. If the subprogram does not include a TITLE instruction, COMPASS prints the variable field of the first IDENT pseudo instruction as the title. A TITLE instruction without a character string produces an untitled listing. A name in the location field introduces a new subprogram sub-subtitle.

A TITLE instruction has no effect when LIST option X is deselected and the TITLE instruction is in text read by XTEXT or is between CTEXT and ENDX instructions. All other TITLE instructions (except the first which sets the main title) cause a page eject, even when generated by a macro expansion, unless LIST option L is deselected.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	TITLE	string

**name**                      New subprogram sub-subtitle to be printed in character positions 70-79 on the second line of the page. A blank name clears the sub-subtitle.

**string**                      COMPASS searches the columns following the blank that terminates the operation field. If it does not find a nonblank character before the default comments column (see COL pseudo instruction), it takes the characters starting with the default comments column minus one up to the end of the statement. Otherwise, the title or subtitle begins with the first nonblank character following TITLE and continues to the end of the statement or to 62 characters. Any characters beyond the 62nd are lost. A blank string produces an untitled listing.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	IDENT	MTD	
	LIST	C	
	TITLE	MT DRIVER	
	.		
	.		
	.		
	TITLE	I/O ROUTINES	
	.		
	.		
	.		

First page: MT DRIVER

Subsequent pages: MT DRIVER  
I/O ROUTINES

#### 4.11.5 TTL — NEW ASSEMBLY LISTING TITLE

The TTL pseudo instruction introduces a new main title to be printed on each page of the listing, and clears the subtitle.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	TTL	string

**name** New sub-subtitle to be printed in character positions 70-79 on the second line of the pages. A blank name clears the sub-subtitle.

**string** COMPASS searches the columns following the blank that terminates the operating field. If it does not find a nonblank character before the default comments column (see COL pseudo instruction), it takes the characters starting with the default comments column minus one up to the statement end. Otherwise, the title begins with the first nonblank character following TTL and continues to the end of the statement or to the 62nd character. Any characters beyond the 62nd are lost. A blank string produces an untitled listing.

TTL does not cause a page eject.

#### 4.11.6 NOREF — OMIT SYMBOL REFERENCES

The NOREF pseudo instruction causes the symbols named in the variable field to be suppressed from the symbolic reference table.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	NOREF	sym <sub>1</sub> , sym <sub>2</sub> , ..., sym <sub>n</sub>

**sym<sub>i</sub>** One or more symbols defined in the subprogram. If a symbol qualifier is in effect when the NOREF is encountered, the symbols are assumed to be qualified by the qualifier in use, unless an unqualified symbol of that name is defined before the NOREF and the qualified symbol is not defined before the NOREF. Alternatively, sym<sub>i</sub> can be a nonblank qualifier symbol enclosed by slant bars, /qualifier/, in which case all symbols qualified by the specified qualifier are suppressed from the symbolic reference table.

A location field symbol, if present, is ignored.

#### 4.11.7 CTEXT AND ENDX — DISABLE/ENABLE LISTING OF COMMON DECK TEXT

The CTEXT pseudo instruction sets the XTEXT flag for list control.

#### NOTE

When the flag is set, external text is listed and symbol references are recorded, only if the X list option is selected.

#### Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	CTEXT	string

**name** If X list option is selected, name is treated as a sub-subtitle; otherwise it is ignored.

**string** If the variable field is nonblank and the X list option is selected, the CTEXT is treated as a subtitle. The CTEXT instruction generates a subtitle and causes a page eject. If X is not selected, the CTEXT does not affect titling.

The subtitle begins with the first nonblank character following CTEXT or in the default comments column (see COL pseudo instruction) minus one, whichever comes first, and continues to the end of the statement or to 62 characters. Any characters beyond the 62nd are lost.

The ENDX pseudo instruction clears the XTEXT flag for list control and causes listing to resume, starting with the instruction after ENDX, when the X list option has not been selected.

#### Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ENDX	

Entries in the location field or variable field, if present, are ignored.

#### 4.11.8 XREF—REFERENCE SYMBOLIC ADDRESS

The XREF pseudo instruction provides the options of having the symbolic reference table contain references to symbols according to (1) location counter address, (2) page and line number, or (3) both. For the format of the symbolic reference table, refer to section 11.8.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	XREF	string

**string** An optional character string, the first character of which indicates how symbols are to be referenced.

- A The symbolic reference table lists addresses only. Flags are not included.
- B The symbolic reference table lists references to symbols according to page number, line, and address. Flags are included.
- P The symbolic reference table lists references to symbols according to page and line numbers. Flags are included.

A location field symbol, if present, is ignored.

If the string is omitted or if no XREF is issued, the symbolic reference table contains references according to page and line numbers and includes flags. The last XREF encountered in a subprogram determines the form of the listing for the entire subprogram.

This chapter describes pseudo instructions that involve definition operations. These pseudo instructions cause sequences of instructions to be saved for these reasons:

- They can be assembled from an external source (XTEXT).
- Assembly can be delayed until later in the subprogram (RMT).
- They can be assembled repeatedly (DUP and ECHO).
- They can be referred to for assembly (MACRO, MACROE or OPDEF).

Any instructions other than END, including other definitions or calls, can be in the body of a definition.

Each request for assembly of one of the saved sequences of code, such as a reference to a macro, causes an entry in the assembler recursion stack. The most recent entry in the stack points to the source of statements (the definition) to be assembled. When the definition contains an inner, nested, reference to a saved definition, the stack pointer is changed so that the source of statements is the innermost definition. The stack allows nesting of definitions to a maximum level of 400. When the end of a definition is reached, the assembler switches to the preceding entry in the stack. When the stack is empty, the assembler resumes assembly of the next statement in the input source deck. A nested definition must be wholly contained by its next outer definition.

Definitions are saved compressed but otherwise unedited (with micro and concatenation marks). Editing occurs each time the definition is processed. Compression removes blanks and replaces them with coded bytes as follows:

A single space is represented by 55<sub>8</sub>; it is not compressed. Two or more embedded spaces are replaced in the image as follows:

- 2 spaces replaced by 5555<sub>8</sub>
- 3 spaces replaced by 0002<sub>8</sub>
- 4 spaces replaced by 0003
- ·           ·
- ·           ·
- ·           ·
- 64 spaces replaced by 0077<sub>8</sub>
- 65 spaces replaced by 007755<sub>8</sub>
- 66 spaces replaced by 00775555<sub>8</sub>
- 67 spaces replaced by 00770002<sub>8</sub>, etc.

Trailing spaces are considered as embedded and are included in the image. The 00 character (colon) is represented by the 12-bit code 0001. A 12-bit zero byte marks the end of the statement.

The listing identifies the source of statements and the recursion level for all definition operations.

For XTEXT, DUP, and ECHO, assembly occurs as soon as a definition is saved. Unless the definition contains a USE, USELCM, or ORG instruction, code is assembled into the block in use when the XTEXT, DUP, or ECHO is encountered. For RMT, macros, and opdefs, however, definition and assembly take place in two steps. The block in use at definition time does not determine where code in the definition will be assembled. That is, code is assembled into the block in use when the definition is assembled if the definition does not itself contain a USE, USELCM, or ORG.

Similarly, for XTEXT, DUP, and ECHO, any qualifier in effect when the pseudo instruction is encountered applies to symbols defined in the sequence (assuming the sequence does not contain a QUAL). For RMT, macros, and opdefs, however, because definition and assembly take place in two steps, the qualifier in use at definition time does not affect symbols in the definition. The qualifier, if any, in effect when the definition is assembled is applied to the symbols defined in the sequence.

A qualifier applies to symbols only. It does not apply to block names or to the names of DUP, ECHO, RMT, or macro definitions, nor to any substitutable parameter names.

In definitions having substitutable parameters, it is possible to use a different block name, different qualifier, or different symbols with each expansion simply by declaring either the qualifier symbol, block name, or symbols to be qualified as substitutable parameters. (For an example, refer to example 7 under Macro Call.)

## 5.1 EXTERNAL TEXT (XTEXT)

The XTEXT pseudo instruction provides a means of obtaining source statements from a file other than that being used for input. COMPASS transfers the text from the external source and assembles it before taking the next statement from the interrupted source of statements. The file may be a sequential file, an indexed file with named records, or an UPDATE or MODIFY<sup>†</sup> random-access program library file.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
file	XTEXT	rname

**file** Name of a file containing source statements. If file is omitted, COMPASS assumes the file named in the X parameter on the COMPASS control statement (section 10.1.2). If no X parameter was specified, COMPASS assumes OLDPL.

**rname** If rname is blank, COMPASS assumes that the file is sequential; it rewinds the file and reads the first section. If rname is not blank, it is the name of the section to be read. The file must be a SCOPE 3 indexed file with named records, a record indexed file with named records, a random-access program library file in UPDATE format, or a random-access program library file in MODIFY format.

<sup>†</sup> MODIFY is not supported by NOS/BE 1 and SCOPE 2.



Text records may be in any of the following formats:

1. Normal text. If the first line contains rname starting in column 1, it is skipped.
2. A common deck in an UPDATE or MODIFY<sup>†</sup> random-access program library file. If the file is in UPDATE format, the first line (\*COMDECK rname) is always skipped. If the file is in MODIFY format, the identification (7700) and modification (7702) tables are skipped. COMPASS does not recognize UPDATE or MODIFY directives such as \*IF in the common deck.
3. An UPDATE or MODIFY<sup>†</sup> compressed compile file section.

COMPASS reads source statements to an end-of-section mark or an END pseudo instruction.

## 5.2 REMOTE ASSEMBLY

Definition and assembly of remote code takes place in two steps. A pair of RMT pseudo instructions delimit code that is to be saved for later assembly. Later, a HERE pseudo instruction directs COMPASS to assemble a specific sequence of remote code or to assemble all unlabeled remote code. An END instruction causes any unlabeled remote code to be assembled.

### 5.2.1 RMT — SAVE REMOTE CODE

A RMT pseudo instruction signals the beginning or the end of a sequence of code to be assembled remotely.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
rmtname	RMT	

**rmtname**            Optional 1-8 character name identifying the remote sequence. It is significant on the beginning RMT only. The field is ignored for a terminating RMT. If supplied, rmtname can be used on a subsequent labeled HERE. If the sequence is unlabeled, an unlabeled HERE or END causes its assembly.

A variable field entry, if present, is ignored.

Any instruction legal when the remote lines are called for assembly is legal between the RMT pair. If expansion of an RMT reveals a second RMT pair implicit to the saved definition, assembly of the first pair must occur through a HERE instruction so that the inner pair will be expanded by an END. Similarly, if the assembly of the second pair reveals yet a third RMT pair, the second pair must be assembled through a HERE rather than the END, etc.

Any labeled remote code present when END is processed is discarded without notice.

---

<sup>†</sup>MODIFY is not supported by NOS/BE 1 and SCOPE 2.

## 5.2.2 HERE – ASSEMBLE REMOTE CODE

A HERE pseudo instruction causes the labeled remote sequence to be assembled or unlabeled saved remote sequences to be assembled. In the absence of a USE, USELCM, IDENT, or an ORG within the saved sequence, the remote code is assembled under the block in use at the time the HERE is encountered. In the absence of a QUAL within the saved sequence, symbols are qualified under the qualifier in use at the time the HERE is encountered. RMT code is assembled only once. After it is assembled, it is no longer saved. A HERE encountered when there is no remote text saved has no effect on assembly.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
rmtname	HERE	

**rmtname**                    Optional; the name of a previously saved RMT sequence. Only the named sequence will be assembled at this time.

A variable field entry, if present, is ignored.

If unlabeled remote sequences still remain to be assembled when the END statement signaling the end of assembly is encountered, COMPASS assembles them before it terminates assembly. However, any RMT pairs that might have resulted from the assembly are lost. Also, any remaining labeled remote code is lost.

**Examples:**

The following example illustrates use of RMT within a macro definition. Following the last call to the macro, a HERE causes all saved unlabeled RMT sequences to be assembled.

Location Code Generated

LOCATION	OPERATION	VARIABLE	COMMENTS
11	11	18	30
	MACRO	TABLE, TNAM, EQU	
	IFC	EQ, **EQUIV*	
TNAM	EQU	*-ORIGINS	
O.TNAM	CON	BUCKET	
	ELSE	2	
TNAM	EQU	EQUIV	
O.TNAM	EQU	O.EQUIV	
	RMT		
L.TNAM	EQU	TNAM+SIZE	
	RMT		
	.		
	.		
	ENDM		
	.		
	.		
	.		
4727	INTER	TABLE	
	IFC	EQ, ***	TABLE .1
1331	INTER	EQU	*-ORIGINS
4727 0000000000000000032304	O.INTER	CON	BUCKET
	RMT		TABLE .1
	L.INTER	EQU	INTER+SIZE
	RMT		TABLE .1
	ENDM		TABLE .1
4730	LASTAB	TABLE	
	IFC	EQ, ***	TABLE .1
1332	LASTAB	EQU	*-ORIGINS
4730 0000000000000000032304	O.LASTAB	CON	BUCKET
	ELSE	2	TABLE .1
	RMT		
	L.LASTAB	EQU	LASTAB+SIZE
	RMT		TABLE .1
	ENDM		TABLE .1
4731	NRTAB	TABLE	LASTAB
	ELSE	2	TABLE .1
1332	NRTAB	EQU	LASTAB
4730	O.NRTAB	EQU	O.LASTAB
	RMT		TABLE .1
	L.NRTAB	EQU	NRTAB+SIZE
	RMT		TABLE .1
	ENDM		TABLE .1
	.		
	.		
	.		
	HERE		
4672	L.INTER	EQU	INTER+SIZE *RMT* .1
4673	L.LASTAB	EQU	LASTAB+SIZE *RMT* .1
4673	L.NRTAB	EQU	NRTAB+SIZE *RMT* .1

In the following example, assembly of the RMT sequence is caused by the END statement.

	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	11	18	30
	FLD	RMT		
	PRS	DECMIC	BUF+BUFL-WSA+ENDS	
		LIT	C*#FLD# DECIMAL REQUIRED.*	
		LIST	C	
100005012	FLD	DECMIC	BUF+BUFL-WSA+ENDS	*RMT* 1
	PRS	LIT	C*#FLD# DECIMAL REQUIRED.*	*RMT 1
	PRS	LIT	C*25759 DECIMAL REQUIRED.*	*RMT 1

### 5.3 CODE DUPLICATION

This section describes two pseudo instructions (DUP and ECHO) that cause a sequence of code to be assembled repeatedly. For a DUP sequence, each assembly is identical with the first, and the number of repetitions is specified or is indefinite. For an ECHO sequence, each assembly resembles a macro reference. Actual parameters supplied in a list are substituted for formal parameters on each repetition of the code sequence. The number of repetitions is determined by the number of actual parameters provided on the ECHO instruction.

Every inner DUP or ECHO sequence must lie totally within the range of the next outer DUP or ECHO, or a fatal E error is flagged.

#### 5.3.1 DUP — SIMPLE DUPLICATION

The DUP pseudo instruction specifies repeated assembly of the statements immediately following. The range of the DUP is specified either by a source statement count on the DUP instruction or by an ENDD.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
dupname	DUP	rep, (nct

**dupname** Optional name of the DUP sequence; 1-8 characters. When supplied, it can be used in an ENDD. When no name is supplied, the range of the DUP is determined by a statement count or by any unnamed ENDD.

**rep** Absolute evaluable expression specifying the integer number of times statements in the DUP range are to be assembled. If rep is null or zero, the instructions in the range are not assembled; that is, code is skipped. When base is M, COMPASS assumes that rep is decimal.

## NOTE

A very large (unobtainable) repeat count in conjunction with a STOPDUP instruction can be used for indefinite duplication of code.

*count*

An evaluable expression specifying an integer count of the number of statements to be assembled repeatedly. When base mode is M, COMPASS assumes that *count* is decimal. The count is decremented for statements only; comment lines (identified by \* in column one) are not counted. On each iteration, the assembler copies the source statements and then assembles them. Thus, any recursive statements within the sequence are counted before they are expanded.

The dupname and *count* parameters are related.

1. If a count is supplied, it takes precedence over any ENDD. The only effect of an ENDD is to be included in the count. Under count control, a name is irrelevant.
2. If neither a count nor a name is supplied, the DUP range is terminated only by an unnamed ENDD.
3. If a name but no count is supplied, the DUP range is terminated by an ENDD with a matching name or by an unnamed ENDD. An ENDD with a name that does not match does not effect the range.

### 5.3.2 ECHO — ECHOED DUPLICATION

The ECHO instruction specifies repeated assembly of the instructions immediately following. On each iteration, the assembler copies the source statements substituting an actual parameter in the list for each formal parameter until the shortest list is exhausted, and then assembles the statements. ECHO offers many of the features of macros but does not require separate definition and reference. The range of the ECHO instruction is specified either by a source statement count specified on the ECHO instruction, or by an ENDD. The statement count, when used, is decremented for instructions only; comment lines, identified by \* in column one, are not part of the definition and are not counted.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
dupname	ECHO	<i>count</i> , $p_1=(list_1)$ , $p_2=(list_2)$ , ..., $p_n=(list_n)$

dupname

Optional name of the ECHO sequence; 1-8 characters. When supplied, it can be used in an ENDD. When no name is supplied, the range of the ECHO is determined by a statement count or by any unnamed ENDD.

`!nct`

Optional absolute evaluable expression specifying an integer count of the number of source statements to be assembled repeatedly. If base mode is M, the count is assumed to be decimal. If `!nct` is zero or omitted, the comma must be present and the ECHO range is defined by an ENDD.

Any recursive statements, such as macro references, are counted before they are expanded.

If the count exceeds the range of an outer DUP or ECHO sequence, a fatal E error is flagged.

The `dupname` and `!nct` parameters are related.

1. If a count is supplied, it takes precedence over any ENDD. The only effect of an ENDD in a count-controlled sequence is for it to be included in the count. Under count control a name is irrelevant.
2. If neither a count nor a name is supplied, the ECHO range is terminated only by an unnamed ENDD.
3. If a name but no count is supplied, the ECHO range is terminated by an ENDD with a matching name or by an unnamed ENDD. An ENDD with a name that does not match does not terminate the sequence.

`pi`

Names of not more than 63 formal substitutable parameters. Each name is 1-8 characters, the first of which must be alphabetic. A name cannot be END, LOCAL, ENDD, IRP, or ENDM. A second or later occurrence of a parameter name is ignored. A name that begins with a number is ignored. The substitutable parameter name can occur in any field within a definition.

The separator between `pi` and `(listi)` is conventionally an = but can be any of the following:

+ - \* / ( ) \$ = , or .

COMPASS recognizes a substitutable parameter name within a definition when it is between any two of the following:

: + - \* / ( ) \$ = blank , . ≠ or ↪

Before the ECHO definition is stored, COMPASS replaces each use of a substitutable name. Otherwise, it saves the definition unedited, i.e., with micro and concatenation marks. Use of the semicolon is restricted in the definition because the assembler, when it expands the definition, interprets it as a substitutable parameter flag (77<sub>g</sub>).

The character ↗ flags the occurrence of a name not bounded by any other special character and, thus, not otherwise recognized. When it expands the definition, COMPASS substitutes an actual parameter value from the list for the substitutable parameter and removes the ↗ so that the adjacent items are concatenated.

Because the assembler replaces the first substitutable parameter with 7701, the second with 7702, etc. the programmer can use the display characters ;A, ;B, etc. directly in place of his substitutable parameter names in the definition and achieve the same results as if the assembler had replaced the name with the flag. (Example 8, section 5.4.3 illustrates a similar application of this technique.)

(list<sub>1</sub>)

Actual parameter list in the form  $a_1, a_2, \dots, a_n$  where  $a_1$  is substituted for  $p_1$  on the first assembly of the ECHO sequence,  $a_2$  is substituted on the second assembly, etc. until the shortest list is exhausted. Two consecutive commas are interpreted as a null parameter. An explicit zero, if desired, must be entered. An actual parameter can contain a set of embedded parameters enclosed by parentheses. However, the embedded parentheses must be properly paired. The assembler removes the outer pair of parentheses before substituting the embedded set in a line. A parenthetical item can contain blanks or commas.

If there are no parameters or any of the lists are null, COMPASS assembles the ECHO sequence zero times, effectively skipping it.

### 5.3.3 STOPDUP — STOP DUPLICATION

The STOPDUP instruction allows premature termination of a DUP duplication before the repeat count is reached or of an ECHO duplication before the shortest list is exhausted. Assembly is completed to the end of the range for the current iteration and then continues with the next source statement. Only the innermost duplication is affected.

A STOPDUP outside of a DUP or ECHO range has no effect on assembly. If a DUP or ECHO is nested, STOPDUP terminates only the innermost DUP or ECHO.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	STOPDUP	

An entry in the location or variable field is ignored.

### 5.3.4 ENDD — END DUPLICATION SEQUENCE

The ENDD pseudo instruction terminates a DUP or ECHO sequence when the statement count is unspecified on the DUP or ECHO.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
dupname	ENDD	

**dupname** Name of a DUP or ECHO sequence, or blank. A named DUP or ECHO sequence can be terminated by an ENDD specifying the sequence by name, or by any unnamed ENDD. An unnamed DUP or ECHO sequence that is not controlled by statement count is terminated only by an unnamed ENDD. An ENDD does not terminate a sequence controlled by a statement count. The ENDD is included in the count but has no other effect.

An ENDD outside the range of a DUP or ECHO has no effect on assembly.

ENDD is part of the definition it terminates; consequently, it is not edited at ECHO definition time. The following definition is in error:

```
T  ↗ 1 ECHO
      Code
T  ↗ 1 ENDD
```

In this code, the location field of the edited ECHO statement is T1, but the location field of the unedited ENDD statement remains at T↗1.

Examples:

In the following examples, the statements that result from expansion are shown shaded. They are listed only when the E list option is selected. Source statements are shown in bold characters.

1. This example illustrates use of a simple DUP instruction.

Location      Code Generated

```

                                000005
5153  0000000000000000000001
5154  0000000000000000000001
5155  0000000000000000000001
5156  0000000000000000000001
5157  0000000000000000000001
```

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	DUP	5, 1	
	DATA	1	
	DATA	1	*DUP* 1
	DATA	1	*DUP* 1
	DATA	1	*DUP* 1
	DATA	1	*DUP* 1
	DATA	1	*DUP* 1



2. This example illustrates a nested DUP instruction with one of the DUP duplications terminated by a STOPDUP.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
GO	MACRO		
TAG	MICRO	NO,1,/#ALPHABET#/ IFC EQ,/#TAG#/#E/,1	ASSEMBLE STOPDUP WHEN TAG=E
	STOPDUP		
NO	SET	NO+1	NO IS 6 IN LAST ITERATION
GO	ENDM		
ALPHABET	MICRO	1,/#AHCDEFGHIJK/ SET 1	
NO	DUP	-1	UNOBTAINABLE ITERATION COUNT
	GO		
	ENDD		
	GO		*DUP*
TAG	MICRO	NO,1,/#ALPHABET#/ TAG	GO
	MICRO	NO,1,/#ABCDEFGHIJK/ IFC EQ,/#TAG#/#E/,1	GO ASSEMBLE STOPDUP WHEN TAG=E
	IFC	EQ,/#A/#E/,1	ASSEMBLE STOPDUP WHEN TAG=E
	STOPDUP		GO
NO	SET	NO+1	GO NO IS 6 IN LAST ITERATION
	ENDM		GO
	ENDD		*DUP*
	GO		*DUP*
TAG	MICRO	NO,1,/#ALPHABET#/ TAG	GO
	MICRO	NO,1,/#ABCDEFGHIJK/ IFC EQ,/#TAG#/#E/,1	GO ASSEMBLE STOPDUP WHEN TAG=E
	IFC	EQ,/#B/#E/,1	ASSEMBLE STOPDUP WHEN TAG=E
	STOPDUP		GO
NO	SET	NO+1	GO NO IS 6 IN LAST ITERATION
	ENDM		GO
	ENDD		*DUP*
	.		
	.		
	.		
	GO		*DUP*
TAG	MICRO	NO,1,/#ALPHABET#/ TAG	GO
	MICRO	NO,1,/#ABCDEFGHIJK/ IFC EQ,/#TAG#/#E/,1	GO ASSEMBLE STOPDUP WHEN TAG=E
	IFC	EQ,/#E/#E/,1	ASSEMBLE STOPDUP WHEN TAG=E
	STOPDUP		GO
NO	SFT	NO+1	GO NO IS 6 IN LAST ITERATION
	ENDM		GO
	ENDD		*DUP*

3. This example illustrates nested ECHO instructions. A statement count terminates the second level ECHO. The ENDD terminates the first level. Notice how COMPASS assembles each copy before it begins the next iteration.

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
			PPU		
			.		
			.		
		STM	PPOP	5,5415H	
			LIST	M,D,F	
			ECHO	,CM=(X,Y,Z)	
			ECHO	2,P1=(A,B,C)	
			LDN	CM	
			STM	P1	
			ENDD		
			ECHO	2,P1=(A,B,C)	*ECHO*
			LDN	X	*ECHO
			STM	P1	*ECHO
1452	1450		LDN	X	*ECHO
1453	5415 0036		STM	A	*ECHO
1455	1450		LDN	X	*ECHO
1456	5415 0037		STM	B	*ECHO
1460	1450		LDN	X	*ECHO
1461	5415 0040		STM	C	*ECHO
			ENDD		*ECHO
			ECHO	2,P1=(A,B,C)	*ECHO
			LDN	Y	*ECHO
			STM	P1	*ECHO
1463	1460		LDN	Y	*ECHO
1464	5415 0036		STM	A	*ECHO
1466	1460		LDN	Y	*ECHO
1467	5415 0037		STM	B	*ECHO
1471	1450		LDN	Y	*ECHO
1472	5415 0040		STM	C	*ECHO
			ENDD		*ECHO
			ECHO	2,P1=(A,B,C)	*ECHO
			LDN	Z	*ECHO
			STM	P1	*ECHO
1474	1470		LDN	Z	*ECHO
1475	5415 0036		STM	A	*ECHO
1477	1470		LDN	Z	*ECHO
1500	5415 0037		STM	B	*ECHO
1502	1470		LDN	Z	*ECHO
1503	5415 0040		STM	C	*ECHO
			ENDD		*ECHO
1505	5415 1524		STM	TAG	

## 5.4 MACROS AND OPDEFS

A macro or opdef definition is a sequence of source statements that are saved and then assembled whenever needed through a macro or opdef call. A macro call consists of the occurrence of the macro name in the operation field of a statement. It usually includes parameters to be substituted for formal parameters in the macro code sequence so that code generated can vary with each assembly of the definition.

An opdef call differs from a macro call in that the assembler interprets the call by examining the format or syntax of the instruction rather than the contents of the operation field alone. The instruction comprising the opdef call usually includes parameters to be substituted for parameters in the code sequence. There are some differences in the way parameters are substituted, however, as is further described under Opdef Call.

Use of a macro or an opdef requires two steps, definition of the macro or opdef sequence, and calling of the definition.

A definition consists of three parts: heading, body, and terminator.

### Heading

A macro definition is headed by a **MACRO** or **MACROE** pseudo instruction stating the name of the macro and identifying substitutable parameters in the body of the macro.

An opdef definition is headed by an **OPDEF** pseudo instruction stating the syntax of the calling instruction and identifying substitutable parameters in the body of the macro.

The heading optionally includes one or more **LOCAL** instructions identifying symbols local to the definition.

### Body

The body begins with the first statement in a definition that is not a **LOCAL** statement or a comment line. A comment line can be either identified by **\*** in column one or can have columns 1-29 blank. (Following the first statement of the macro body, only comments identified by **\*** in column 1 are ignored.)

Use of the semicolon is restricted because when a definition is expanded a semicolon is interpreted as a substitutable parameter mark or a local symbol flag.

The body consists of a series of symbolic instructions. All instructions other than **END**, including other macro and opdef definitions and calls are legal within a definition. However, a definition within a definition is not defined until the outer definition is called. Therefore, an inner definition cannot be called before the outer definition is called.

A name of a substitutable parameter or local symbol listed in the heading can occur in any field within the body. A reference to a substitutable parameter or local symbol is recognized when it is between two of the following characters in an expression or field:

: + - \* / ( ) \$ = blank , . ≠ or ↦

The character ↦ flags the occurrence of a name not bounded by any other special

character, and, thus, not otherwise recognized. On a call, the assembler substitutes an actual parameter value for the substitutable parameter and removes the  $\rightarrow$  so that the adjacent items are concatenated.

**NOTE**

The programmer can legally use the characters . ( ) : \$ and = in symbols, but when he does, he must be careful that these characters are not interpreted as delimiters in macro definitions (example 4 under macro calls). A symbol should not begin with a colon; if it does, the colon is ignored and no error message is issued.

The macro body optionally contains IRP pseudo instructions that allow iterative assembly of a sequence within the body such that each iteration uses a different parameter value.

**Terminator** An ENDM pseudo instruction terminates a macro or opdef definition.

**Definition Processing** A macro or opdef can be defined anywhere in a subprogram before it is called. When COMPASS encounters a definition, it places the name of the macro or the syntax of the opdef along with the number of substitutable parameters and local symbols in the assembler operation code table. Before the definition is saved, COMPASS replaces each occurrence of a parameter name or local symbol with a 77xx (where xx is a number assigned to the substitutable parameter or local symbol).

On the call, each use of a substitutable parameter (each 77xx) is replaced by its actual parameter; each use of a local symbol is replaced by a unique symbol generated by the assembler. Usually, symbols replaced in this way have no meaning outside the definition. However, if the macro includes an RMT sequence which contains local symbols, the local symbols will have meaning where the remote code is assembled outside of the definition.

**5.4.1 ENDM — END MACRO DEFINITION**

An ENDM terminates a macro or opdef definition.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	ENDM	

mname Name of a macro sequence, syntax of an OPDEF sequence, or blank.

An ENDM specifying a macro by name terminates the named macro definition and any unterminated macro or opdef definitions within it. An ENDM that does not specify a macro by name terminates all unterminated definitions. An ENDM outside the range of any macro sequence has no effect other than to be included in statement counts.

ENDM is part of the definition it terminates; consequently, it is not edited at MACRO definition time. The following definition is in error:

```
T1→1  MACRO
      Code
T1→1  ENDM
```

In this code, the location field of the edited MACRO statement is T1, but the location field of the unedited ENDM statement remains at T1→1.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
JAY	MACRO	P1,P2,P3	
	.		
	.		
KAY	MACROE	PK2,PK2,PK3,PK4	
	.		
	.		
JPX/XQ	OPDEF	OP1,OP2,OP3	
	.		
	.		
KAY	ENDM		TERMINATES KAY AND THE OPDEF DEFINITION
	.		
	.		
	ENDM		TERMINATES JAY

#### 5.4.2 MACRO — MACRO HEADING

A MACRO pseudo instruction notifies the assembler to place the instructions forming the body of the macro in a table of macro definitions for assembly upon call and place the macro name in the operation code table.

The MACRO pseudo instruction has two forms:

Format one:

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	MACRO	parameters

Format two:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	MACRO	mname, parameters

The blank location field identifies the second format.

**mname**            A legal name other than END, ENDD, IRP, LOCAL, or ENDM. 1-8 characters.

A name that is identical to a PPU symbolic machine instruction, pseudo instruction, or macro already in the operation code table redefines the instruction. The most recent definition applies for the macro call. A redefinition causes an informative flag to be issued but the new definition holds.

**parameters**       Names of substitutable parameters. The order in which names are listed determines the order in which parameters must occur in the macro call. Each name is 1-8 characters, the first of which must be alphabetic. A name cannot be END, IRP, LOCAL, ENDD, ENDM, or the same as a local symbol. A name that begins with a number, or a second or later occurrence of a parameter name in the list is ignored.

Any of the following special characters separate parameters in the list:

+ - \* / ( ) \$ = , or .

These characters have no meaning other than as separators. A blank terminates the list of parameters. Also, any of these characters can be used to separate the mname from parameters in format two.

The total number of unique parameter names and local symbols must not exceed 63 for any one macro definition.

Format one does not require parameters.

Format two requires at least one substitutable parameter. This parameter is termed the location argument because the location field entry in the macro call is its substituted value. Omission of the location argument from a MACRO instruction in format two causes the assembler to issue a fatal error and ignore the definition.

The assembler ignores a blank parameter produced by two adjacent separators or by a separator at the end of the list.

For an example of definition and calls, refer to Macro Calls.

Examples of macro instructions:

1. Legal MACRO instructions:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
ABC	MACRO	P1,P2,P3	
MESSAGE	MACRO	DEF*LOC*ONE*TWO*TEN	
	MACRO	A	

2. MACRO instructions having identical parameter lists.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
SUM	MACRO	X=Y+Z+X	SECOND X PARAMETER IS IGNORED
SUM	MACRO	X(Y+Z)	
SUM	MACRO	X=Y+Z	
SUM	MACRO	X,Y,(Z+X)	NULL PARAMETER AND SECOND X ARE IGNORED
RAO	MACRO	X	
RAO	MACRO	X=X+1	SECOND X AND NUMERIC PARAMETER ARE IGNORED

3. Illegal use of format two:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	MACRO	ABC	NO SUBSTITUTABLE PARAMETER
	MACRO	ABC,,FP	NULL PARAMETER FIELD
	MACRO	ABC,16,FP	NUMERIC PARAMETER FIELD

### 5.4.3 MACRO CALLS

A macro headed by a MACRO pseudo instruction can be called by an instruction in the following format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	mname	p <sub>1</sub> , p <sub>2</sub> , ..., p <sub>n</sub>

sym                    Optional; depends on definition (see discussion following)

p<sub>i</sub>                    Parameter list composed of alphanumeric strings. Parameters are separated by commas and terminated by a blank. Two consecutive commas constitute a null parameter. An explicit zero, if desired, must be entered.

Each parameter must be in its correct relative position depending on the sequence in which its formal substitutable name is given in the MACRO pseudo instruction.

When the definition MACRO is in format one, the first parameter in the call is substituted wherever the first substitutable parameter occurs in the definition, the second parameter in the call is substituted wherever the second substitutable parameter occurs in the definition, etc. When the definition MACRO is in format two, the location field entry in the call is substituted wherever the first substitutable parameter occurs in the definition, the first parameter in the variable field of the call is substituted wherever the second substitutable parameter occurs in the definition, etc.

If null parameters are interspersed with legal parameters, the correct positions must be established with commas. When the list terminates before the last possible parameter, all remaining parameters are considered null.

When the first character of a parameter is a left parenthesis, the assembler considers all the characters between it and the matching right parenthesis as an embedded parameter or as an iterative parameter. It is an iterative parameter when the substitutable parameter has been named in an IRP pseudo instruction (section 5.4.9). Otherwise, it is an embedded parameter.

The assembler removes the outer pair of parentheses before substituting the enclosed character string in a line. Embedded parenthetical items must be properly paired. A parenthetical item can contain blanks and commas.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	MESSAGE	(=C*PROGRAM	ABORT.*)

After substitution, spacing between fields is the same as it was before substitution. One effect is that a null actual parameter replacing a formal parameter in a variable field effectively moves the comments field to the left. Then, when the line is assembled, the comments could be erroneously interpreted as a variable subfield.



Processing of a location symbol and forcing upper of the first macro instruction depend on the MACRO form used for the definition.

If the macro is defined using format one, that is, the macro name is in the location field, a location symbol on the macro call line forces the first word of generated code upper. The location field symbol is assigned the current value of the location counter. A location field (if any) on the line in the definition that generates the code is assigned the same address. If the location field of the macro call does not contain a symbol, the location and position counters are not affected by the call.

When the macro is defined using format two, that is, the macro name is in the variable field and the first parameter is a location argument, the location symbol of the call is substituted for the first parameter or location argument. The fact that this argument came from the location field rather than the variable field has no special significance in the macro expansion. In the macro call, the location field argument cannot be more than 8 characters. Parentheses are not given the special meaning used in the variable field of a macro call line.

Example:

1. An illustration of concatenation

<u>Location</u>	<u>Code</u> <u>Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
		MACK	MACRO	P1,P2	
			S→P1	P1+1R→P2	
			.		
			.		
			ENDM		
			.		
			.		
4740		MACK	AZ→A		
		S→AZ	AZ+1R→A		MACK .1
4740	5022000001	SAZ	AZ+1PA		MACK .1
			ENDM		MACK .1

2. An illustration of nested definitions and calls

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
NAME 1	MACRO		
.	.		
.	.		
NAME 2	MACRO		
.	.		
.	.		
NAME 2	ENDM		
.	.		
.	.		
.	NAME 2		AT THIS TIME, THIS LINE IS PART OF A DEFINITION RAATHER THAN BEING A CALL.
.	.		
.	.		
NAME 1	ENDM		
.	.		
.	NAME 1		NAME 1 IS CALLED AND EXPANDED.
.	.		
.	.		
NAME 2			CALL TO NAME 2 IS VALID

3. The following example illustrates two calls to a definition headed by a MACRO in format two using the location argument. The macro is named TABLE; its substitutable arguments are TABNAM, VALUE1, and VALUE2, where TABNAM is the location argument.

LOCATION	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
		TABNAM	MACRO	TABLE, TABNAM, VALUE1, VALUE2	
			VFD	60/VALUE1, 60/VALUE2	
			ENDM		
		.	.		
		.	.		
4741	17204000000000000000	SPVAL	TABLE	1.0, 2.0	CALL ONE
4742	17214000000000000000	SPVAL	VFD	60/1.0, 60/2.0	TABLE .1
			ENDM		TABLE .1
		.	.		
		.	.		
4743					
4743	17204000000000000000		TABLE	1.0	CALL TWO
4744	00000000000000000000	VFD	VFD	60/1.0, 60/	TABLE .1
			ENDM		TABLE .1

4. An illustration of embedded parameters:

Definition:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
XAM	MACRO LDM LJM ENDM	A,R A R	

Call:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
XAM	(SUM,10R), (SAM,IND3)		

Expansion:

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
7303	5010 7244		LDM	SUM,10R	
7305	0117 7243		LJM ENDM	SAM,IND3	

5. The following example illustrates use of R= in macros:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
ONSW	MACRO R= SX2 RJ ENDM	N X1,N 11R =XCPM=	
OFFSW	MACRO R= SX2 RJ ENDM	N X1,N 12R =XCPM=	

6. The following example illustrates a character in a symbol erroneously being interpreted as a delimiter for a parameter.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
ABC Z	MACRO SET SA7 . . . ENDM	Z,VAL,P5 VAL Z.ALPHA . . .	ILLEGAL SYMBOL, TOO LONG
IOTA	ABC SET SA7 ENDM	IOTA,1,3 1 IOTA.ALPHA	ILLEGAL SYMBOL, TOO LONG
			ABC .1 ABC .1 ABC .1

7. The following example illustrates changing of control blocks and symbol qualifiers through substitutable parameters in a macro. (The same call could be used by using micros to change actual parameters.)

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
TAB	MACRO USE QUAL	BLOCK,KWAL BLOCK KWAL	
TAG1 TAG2	BSS VFD USE QUAL ENDM . . .	10B 60/-1 * *	
	TAB	ONE,ONE	
	USE	ONE	TAB .1
	QUAL	ONE	TAB .1
TAG1	BSS	10B	TAB .1
TAG2	VFD	60/-1	TAB .1
	USE	*	TAB .1
	QUAL	*	TAB .1
	ENDM		TAB .1
	TAB	TWO,TWO	
	USE	TWO	TAB .1
	QUAL	TWO	TAB .1
TAG1	BSS	10B	TAB .1
TAG2	VFD	60/-1	TAB .1
	USE	*	TAB .1
	QUAL	*	TAB .1
	ENDM		TAB .1

8. The following example illustrates a technique that an experienced programmer may wish to use to save time in processing of definitions. Remember that the assembler replaces the first substitutable parameter with 7701, the second with 7702, etc. Note that 7701 is ;A in display characters, 7702 is ;B, etc. This means that the programmer can use the display characters directly in place of his substitutable parameter names in the body of the definition and achieve the same results as if the assembler had made the substitution when it saved the definition. At the time the definition is assembled, the assembler replaces each 77xx with the actual parameter whether the code was inserted by the assembler when it saved the definition or by the programmer when he coded the definition.

	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
		CHAR	MACRO ASCII,INTERNAL,EXTERNAL,BCD CON :D:C:R:A ENDM . . . BASE 0 CHAR 43,10,10,30	
7771	00000000000030101043		CON 30101043	CHAR 1
			ENDM	CHAR 1
7772			CHAR 44,11,11,31	9
7772	00000000000031111144		CON 31111144	CHAR 1
			ENDM	CHAR 1
7773			CHAR 45,50,20,13	*
7773	00000000000013206045		CON 13206045	CHAR 1
			ENDM	CHAR 1
7774			CHAR 46,40,40,15	-
7774	00000000000015404046		CON 15404046	CHAR 1
			ENDM	CHAR 1
7775			CHAR 47,54,54,12	*
7775	00000000000012345447		CON 12345447	CHAR 1
			ENDM	CHAR 1
7776			CHAR 50,21,61,17	/
7776	00000000000017612150		CON 17612150	CHAR 1
			ENDM	CHAR 1

#### 5.4.4 MACROE — EQUIVALENCED MACRO HEADER

A **MACROE** pseudo instruction can be used instead of a **MACRO** instruction to notify the assembler to place the instructions forming the body of the macro in a table of macro definitions for assembly upon call, to place the macro name in the operation code table, and to save the list of parameter names so that actual parameters supplied in the macro call can be listed by name in any sequence in the macro call.

The **MACROE** pseudo instruction has two forms:

Format one:

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	MACROE	parameters

Format two:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	MACROE	mname, parameters

The blank location field identifies the second format.

**mname** A legal name other than **END**, **ENDD**, **IRP**, **LOCAL**, or **ENDM**. It can be 1-8 characters. A name that is identical to a PPU symbolic machine instruction name, pseudo instruction, or macro instruction already in the operation code table redefines the instruction. The most recent definition is the one that applies for the macro call. A redefinition causes an informative flag to be issued but the new definition holds.

**parameters** Names of substitutable parameters. Unlike **MACRO**, the order in which names are listed does not determine the order in which parameters can occur in the macro call. Each name is 1-8 characters, the first of which must be alphabetic. A name cannot be **END**, **ENDD**, **LOCAL**, **IRP**, **ENDM**, or the same as a local symbol. A name that begins with a number, or a second or later occurrence of a parameter name in the list is ignored. Any of the following special characters separate parameters in the list:

+ - \* / ( ) \$ = , or .

These characters have no meaning other than as separators. A blank terminates the list of parameters. Also, any of these can be used to separate the **mname** from parameters in format two.

The total number of unique parameter names and local symbols must not exceed 63 for any one macro definition.

Format one does not require parameters.

Format two requires at least one substitutable parameter. This parameter is termed the location argument because the location field entry in the macro call is its substituted value. Omission of the location argument from a MACRO instruction in format two causes the assembler to issue a fatal error flag and ignore the definition.

The assembler ignores a blank parameter produced by two adjacent separators or by a separator at the end of the list.

For an example of definition and calls, refer to Equivalenced Macro Call.

### 5.4.5 EQUIVALENCED MACRO CALL

A macro definition headed by a MACROE pseudo instruction can be called by an instruction of the following format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	mname	$p_1=a_1, p_2=a_2, \dots, p_n=a_n$

mname                      Name of MACROE definition

sym                         Optional symbol. A symbol in the location field causes the location counter to be forced upper. The symbol is then assigned the value of the location counter. A location field symbol on the first line in the definition that generates code is assigned the same address. If the location field of the macro call does not contain a symbol, the manner of the force upper is a function of the first-code-generating line in the macro expansion.

$p_i=a_i$                       An equivalenced parameter. Each p is the name of a substitutable parameter. The  $a_i$  is an actual parameter to be substituted for  $p_i$ . The parameters need not be listed in the same order as they are listed on the MACROE instruction. Equivalenced parameters in the list are separated by commas and terminated by a blank.

A null value is substituted for any parameter omitted from the list.

When the first character of an actual parameter is a left parenthesis, the assembler considers all the characters between it and the matching parenthesis as an embedded parameter or as an iterative parameter. It is an iterative parameter when the substitutable parameter has been named in an IRP pseudo instruction (section 5.4.9, IRP). Otherwise, it is an embedded parameter. The assembler removes the outer pair of parentheses before substituting the enclosed character string in a line. Embedded parenthetical items must be properly paired. A parenthetical item can contain blanks and commas.

After substitution, spacing between fields is the same as it was before substitution. One effect is that a null actual parameter replacing a formal parameter in a variable field effectively moves the comments field to the left. Then, when the line is assembled, the comments could be erroneously interpreted as a variable subfield.

Processing of a location symbol and forcing upper of the first macro instruction depend on the MACROE form used for the definition.

If the macro is defined using format one, that is, the macro name is in the location field, a location symbol on the macro call line forces the first word of generated code upper. The location field symbol is assigned the current value of the location counter. A location field (if any) on the line in the definition that generates the code is assigned the same address. If the location field of the macro call does not contain a symbol, the location and position counters are not affected by the call.

When the macro is defined using format two, that is, the macro name is in the variable field and the first parameter is a location argument, the location symbol of the call is substituted for the first parameter or location argument. The fact that this argument came from the location field rather than the variable field has no special significance in the macro expansion. After substitution, spacing between fields is the same as it was before substitution.

Example, format one:

Location            Code Generated

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
SAM	MACROE	A,B,C	
	CON	A	
	CON	B	
	CON	C	
	.		
	.		
	.		
	SAM	A=1,C=5,B=0	
5007	CON	1	SAM 1
5010	CON	0	SAM 1
5011	CON	5	SAM 1
	ENDM		SAM 1

Example, format two:

Location            Code Generated

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	MACROE	SAM,XX,A,B,C	
	CON	A	
	CON	B	
	CON	C	
	ENDM		
2	SAM	A=1,B=2,C=3	



## 5.4.6 OPDEF — DEFINE CPU OPERATION

An OPDEF pseudo instruction notifies the assembler to place instructions in the body of the definition in a table of definitions for assembly upon call and place the instruction syntax in the operation code table. There is no way of removing the definition from the table. It can, however, be bypassed through redefinition, or disabled through PURGDEF. If the syntax duplicates a CPU instruction already in the table, the OPDEF definition takes precedence.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
syntax	OPDEF	parameters

syntax

The syntax consists of a mnemonic operator and variable field descriptors. The mnemonic operator consists of two characters. The first can be any character except blank. The second character can be a register designator: A, B, or X in which case the operation field of the opdef call is recognized as cAn, cXn, or cBn (c is a unique character; n is 0-7); or the second character can be any other character, in which case the operation field of the opdef call is recognized simply by a two-character mnemonic, such as EQ.

The variable field descriptors define the order of appearance of all registers, expressions, and subfield separators that comprise the variable field of the opdef call. It consists of none, one, two, or three of the following 22 subfield descriptors. Q represents an expression. An r represents a register letter (A, B, or X). A comma separates two descriptors; a blank terminates the syntax.

void	Q
r	rQ
-r	-rQ
$r_1+r_2$	$r_1+r_2Q$
$-r_1+r_2$	$-r_1+r_2Q$
$r_1*r_2$	$r_1*r_2Q$
$-r_1*r_2$	$-r_1*r_2Q$
$r_1/r_2$	$r_1/r_2Q$
$-r_1/r_2$	$-r_1/r_2Q$
$r_1-r_2$	$r_1-r_2Q$
$-r_1-r_2$	$-r_1-r_2Q$

For example,  $-r_1 * r_2$  would be written as  $-X*B$  to describe  $-X3*B1$  whereas  $rQ$  would be written as  $BQ$  to describe  $B2+ALPHA$ . The first descriptor immediately follows the mnemonic operator.

parameters

A substitutable parameter for each register designator (r) and expression designator (Q) in the syntax in the order in which they occur in the syntax (and, consequently, in the calling instruction). Each name is 1-8 characters, the first of which must be alphabetic. A name cannot be END, ENDD, ENDM, IRP, LOCAL, or the same as a local symbol. A name that begins with a number, or a second or later occurrence of a parameter name in the list is ignored. Parameters can be separated by any of the characters:

+ - \* / ( ) \$ = , or .

These characters have no meaning other than as separators. A blank terminates the list of parameters.

The total number of unique parameter names and local symbols must not exceed 63 for any one OPDEF definition.

The assembler ignores a blank parameter produced by two concurrent separators or by a separator at the end of the list. A second or later occurrence of a parameter name in the list is ignored.

Examples:

1. Listed below are some instructions that could be defined through OPDEF:

Calling Instruction		Opdef Syntax
Operation	Variable Subfields	
JP <sup>†</sup>	K <sup>††</sup>	JPQ
JP <sup>†</sup>	Bn+K	JPBQ
JP	Bn+Bn+K	JPB+BQ
JP	Bn, K	JPB, Q
JP	Xn/Xn+K	JPX/XQ
NE <sup>†</sup>	Bn, Bn, K	NEB, B, Q
LJ	Bn-Bn, An-Xn, K	LJB-B, A-X, Q
BXn <sup>†</sup>	-Xn*Xn	BX-X*X
SBn <sup>†</sup>	Xn+Bn	SBX+B
LXn <sup>†</sup>	Bn, Xn	LXB, X
JP <sup>†</sup>	Bj+K	JPBQ
NE <sup>†</sup>	Bj, Bk, K	NEB, B, Q
BXi <sup>†</sup>	-Xk*Xj	BX-X*X
SBi <sup>†</sup>	Xj+Bk	SBX+B
SBi <sup>†</sup>	Bj+Xk	SBB+X

<sup>†</sup> Legal COMPASS CPU instructions

<sup>††</sup> K represents an expression.

2. The following complete definition redefines single-address long jump JP as the EQ jump, which is faster than JP on the 6600 Computer System.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
JPQ	OPDEF EQ ENDM	P1 P1	

Each subsequent JP instruction that matches the syntax JPQ is assembled as an EQ. A JP instruction having a different syntax, such as the following, is not affected.

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
10002	0233000005 +	1	11	18	30
			JP	R3+ALPHA	

3. The following definition traps all floating point double-precision subtraction instructions (DXi Xj-Xk) and jumps to an error-check routine for debugging. I, J, and K are substitutable parameters used within the definition.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
DXY-Y	OPDEF . . . PJ ENDM	I, J, K    CKOUT	

4. The following sequence causes RXi K to be defined as AXi K. It does not affect the standard RXi instructions involving registers.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
RXQ	OPDEF AX.P1 ENDM	P1,P2 P2	

#### 5.4.7 OPDEF CALL

An opdef call resembles a CPU mnemonic machine instruction. The mnemonic code, quantity and sequence of registers, arithmetic operators, and expressions (excluding operators within the expressions) must match the syntax described in the OPDEF for the definition to be called.

NOTE

If the Q in a descriptor is combined with register letters, a plus or minus must precede an expression in the call.

<u>OPDEF Syntax</u>	<u>Call</u>	
JPQ	JP K	Not combined
JPBQ	JP Bn+K	Combined
JPB,Q	JP Bn,K	Not combined
JPX/XQ	JP Xn/Xn+K	Combined

An OPDEF call can occur any place after the definition is saved. In substituting parameters, the assembler uses only the register values given in the call. It does not substitute the register designators.

A location symbol on the opdef call line forces the first word of generated code upper. The location field symbol is assigned the current value of the current location counter after the force upper. A location field on the line in the definition that generates code is assigned the same value. If the location field of the opdef call does not contain a symbol, the manner of the force upper is a function of the first code-generating instruction in the expansion. If the call location field and the code-generating instruction field both contain symbols they are assigned the same value.

Only a line having the correct syntax calls the definition.

Examples:

The following opdef defines an instruction having the syntax IXX/X. On the call, the assembler substitutes 3, 4, and DIV (not X3, X4, and X.DIV) for P1, P2, and P3, respectively.

<u>Location</u>	<u>Code Generated</u>	<u>LOCATION</u>	<u>OPERATION</u>	<u>VARIABLE</u>	<u>COMMENTS</u>
		1	11	18	30
		IXX/X	OPDEF	P1,P2,P3	
			PX.P2	X.P2	
			PX.P3	X.P3	
			NX.P2	X.P2,B4	
			NX.P3	X.P3,B4	
			FX.P1	X.P2/X.P3	
			UX.P1	X.P1,B4	
			LX.P1	X.P1,B4	
			ENDM		
			.		
			.		
			.		
		IX3	X4/X.DIV		
	27404	PX.4	X.4		IX3 .1
	27000	PX.DIV	X.DIV		IX3 .1
27 24444		NX.4	X.4,B4		IX3 .1
	24040	NX.DIV	X.DIV,B4		IX3 .1
	44340	FX.3	X.4/X.DIV		IX3 .1
	26343	UX.3	X.3,B4		IX3 .1
30 22343		LX.3	X.3,B4		IX3 .1
			ENDM		IX3 .1

The following OPDEF selectively traps the SXi Xj+Bk instructions.

Definition:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
SXX+B	OPDEF . . . ENDM	I, J, K	

Statements that call the definition:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SX3 . . .	X1+B2	
SYM	SX.NN	X6+B.XXX	

Statements that do not call the definition:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SX5	X4	NO R DESIGNATOR OR +.
	SX6	B3+X4	REGISTERS INTERCHANGED
	SX.Y	B3	NO X DESIGNATOR OR OPERAND
	SY	X4+B4	MNEMONIC CODE NOT SX.

#### 5.4.8 LOCAL—LOCAL SYMBOLS

One or more LOCAL instructions that list symbols local to the definition optionally follows the MACRO, MACROE, or OPDEF pseudo instruction. The only lines that can separate the first header statement from LOCAL are comment lines.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	LOCAL	symbols

symbols

List of local symbols. Each symbol must begin with an alphabetic character. Symbols must be separated by and must not include the following characters:

+ - \* / ( ) \$ = , or .

A blank terminates the list. The assembler ignores a null symbol produced by two adjacent separators or by a separator at the end of the list. COMPASS ignores the use of a substitutable parameter name, another local symbol name, or a name beginning with a number in the local symbol list. A local symbol cannot be END, ENDD, ENDM, IRP, or LOCAL. The total number of unique parameter names and local symbols must not exceed 63 for any one macro or OPDEF definition.

A location field symbol, if present, is ignored.

A symbol in the list is considered local to the macro; that is, it is known only within the macro definition. On each expansion of the macro, COMPASS creates a new symbol for each local symbol and substitutes it for each occurrence of the local symbol in the definition (other than in comment lines identified by \* in column 1). Thus, invented symbols replace LOCAL-named symbols wherever they appear in a macro definition in a manner similar to the way substitutable parameters are replaced. The chief difference between substitutable parameters and local symbols is that COMPASS automatically supplies the value C (character string to be substituted for) a local symbol so that it is unique for each macro call.

A user passes a local symbol to inner macro definitions or inner macro calls when he does not declare the symbol local in any of the inner definitions saved or called. That is, a symbol declared local in a macro can be referred to in any inner macro that does not also declare it as local (see example 2).

A symbol not defined as local is accessible from outside the macro definition. An invented symbol is qualified if defined while in a QUAL block. It is not listed in the symbolic reference table. Blanks are preserved in a line containing a substituted symbol; COMPASS makes no attempt to change the structure of the line.

On the listing, each invented symbol is shown as †sym, where sym is unique for each local symbol in the subprogram. For example, if the symbol A is declared local to the macro, the subprogram can define a different symbol A elsewhere.

Examples:

1. In the following example, C is local to macro ABC and is passed to inner macro definitions. In the definition, each occurrence of formal parameter A is replaced by the parameter mark 7701; each occurrence of B by the parameter mark 7702, and each occurrence of C by the parameter mark 7703. Then, when ABC is called, COMPASS assigns invented symbol †000001 to C and replaces each occurrence of 7703 in definitions ABC and XYZ.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
ABC	MACRO	A, B	DEFINITION OF ABC
	LOCAL	C	
C	BSS	10B	
.	.	.	
XYZ	MACRO	D	DEFINITION OF XYZ
	SA1	C	
.	.	.	
	ENDM		
†000001	ABC	3,4	EXPANSION OF ABC
XYZ	MACRO	BSS 10B	
	SA1	†000001	DEFINITION OF XYZ
	ENDM		
			ABC .1
			ABC .1
			ABC .1
			ABC .1

2. In the following example, C is local to each level. Note how this example differs from the preceding one.

LOCATION	OPERATION	VARIABLE	COMMENTS
1			30
BCD	MACRO	A, B	DEFINITION OF BCD
C	LOCAL	C	
.	BSS	10B	
.	.	.	
YZA	MACRO		DEFINITION OF YZA
.	LOCAL	C	
.	SA1	C	
.	.	.	
C	BSSZ	1	
	ENDM		

On the call to BCD, the assembler replaces each occurrence of C with the invented symbol, +000002 including the use of the symbol in the LOCAL instruction for macro XYZ.

LOCATION	OPERATION	VARIABLE	COMMENTS
1			30
	BCD	5*6	EXPANSION OF BCD
+000002	BSS	10B	BCD .1
YZA	MACRO		BCD .1
	LOCAL	+000002	BCD .1
	SA1	+000002	DEFINITION BCD .1
+000002	BSSZ	1	BCD .1
	ENDM		BCD .1

Finally, on a call to YZA, +000002 is defined as local and the assembler replaces each +000002 with another invented symbol. Thus, each reference to C in the source code SA1 instruction does not result in a reference to the BSS in the outer macro.

LOCATION	OPERATION	VARIABLE	COMMENTS
1			30
	YZA		EXPANSION OF YZA YZA .1
	SA1	+000003	DEFINITION YZA .1
+000003	BSSZ	1	YZA .1
	ENDM		

#### 5.4.9 IRP — INDEFINITELY REPEATED PARAMETER

An IRP pseudo instruction in a macro definition signals the beginning or end of a sequence of code to be assembled repeatedly with one parameter varied with each repetition.

It has two formats:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	IRP	parameter
	IRP	

The first form introduces the sequence and names the substitutable parameter; the second form terminates the repeated sequence. In either form, a location field symbol, if present, is ignored.

The parameter name must be listed as a substitutable parameter on the MACRO or MACROE pseudo instruction for the definition.

On the macro call, the indefinitely repeated parameter consists of one or more subparameters enclosed by parentheses and separated by commas. The assembler assembles the sequence for each subparameter; the number of copies of the sequence depends on the number of subparameters (none at all when the actual parameter is null). When the list of subparameters is exhausted, the assembler continues with the next line in the definition. If the named substitutable parameter does not occur between the two IRP instructions, the assembler repeats the code unchanged for each subparameter provided in the call. An IRP outside of the range of a macro has no effect on assembly other than to be included in statement counts.

IF-skips of IRP sequences should be controlled by instruction bracket names rather than statement counts because IRP expansions are done even when an IF-skip is used and because the number of statements generated by IRP is variable.

Anything that can be done with an IRP pair can be done with ECHO and ENDD. IRP is faster at assembly time but ECHO is more flexible (it is not expanded during IF-skips, allows multiple arguments, and can be nested). IRP should be used when greater speed is desired and the expanded capabilities of ECHO are not needed.

Examples:

1. Repeat sequence within macro

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
	ZAB	MACRO	ARG,B	
		IRP	ARG	
		SA1	ARG	
		SX6	X1+B	REPEATED
		SA6	ARG	SEQUENCE
		IRP		
		ENDM		
		.		
		.		
		.		
??		ZAB	(J,K,L),CON	ZAB .1
		IRP	J,K,L	ZAB .1
22	5110000000	SA1	J	ZAB .1
	7261000000	SX6	X1+CON	ZAB .1
23	5160000000	SA6	J	ZAB .1
	5110000000	SA1	K	ZAB .1
24	7261000000	SX6	X1+CON	ZAB .1
	5160000000	SA6	K	ZAB .1
25	5110000000	SA1	L	ZAB .1
	7261000000	SX6	X1+CON	ZAB .1
26	5160000000	SA6	L	ZAB .1
		IRP		ZAB .1
		ENDM		



2. Assign symbol at every 100<sub>8</sub> words of zeroed storage:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	USE	STORAGE	
BUF	MACRO	P1	
	IRP	P1	
P1	BSSZ	100B	
	IRP		
	ENDM		
	BUF	(P,Q,R,S,T)	
	IRP	P,Q,R,S,T	
			BUF .1
P	BSSZ	100B	BUF .1
Q	BSSZ	100B	BUF .1
R	BSSZ	100B	BUF .1
S	BSSZ	100B	BUF .1
T	BSSZ	100B	BUF .1
	IRP		BUF .1
	ENDM		BUF .1

## 5.5 SYSTEM MACRO AND OPDEF DEFINITIONS

Definitions of such general usefulness that they should be available to any program without each program defining them can be placed on the system text file as system macros or can be placed on a file accessible through an XTEXT pseudo instruction.

System macros provide for such system functions as reading and writing files and specifying parameters for file environment tables, etc. Systems macro definitions are available to COMPASS for each assembly. The programmer can use a macro call for a system macro at any time in his program. Descriptions of system macros are given in the operating system reference manual.

Systems definitions can include any legal macro or opdef definition. An expansion of a call for a system definition is not normally included on the assembler listing. Use of the S option of the LIST pseudo instruction (Section 4.11.1) enables listing of expansions of system definitions.



---

The COMPASS operation code table contains the information that COMPASS requires for interpreting legal operation field entries for COMPASS instructions.

When assembly begins, the operation code table contains these entries:

- Pseudo instructions (except LOCAL)
- CPU symbolic instructions (chapter 8)
- CMU symbolic instructions (chapter 8)
- PPU symbolic instructions (chapter 9)
- System macro and opdef definitions

The MACRO, MACROE, and OPDEF pseudo instructions (chapter 5) cause entries to be made in this table. In addition, the programmer has the capability of creating entries through the following instructions discussed later in this chapter:

CPOP	CPU operation
PPOP	PPU operation
OPSYN	Synonymous PPU or pseudo operation or macro
CPSYN	Synonymous CPU operation or opdef

If a new entry redefines an instruction already in the table, the obsolete entry is not physically removed from the table. Instead, it is saved so that the table can be reconstructed between assemblies. COMPASS reconstructs the operation code table using all the original system macros, opdefs, pseudo instructions, and symbolic machine instructions. No programmer-created entry is preserved from assembly to assembly. The number of entries in the table is limited to 4123.

The only pseudo instructions that logically remove entries from the operation code table are PURGMAC and PURGDEF.

Entries in the operation code table are in two distinct formats permitting a logical division of the table. One type of entry permits identification of an instruction by finding a match for the contents of the operation field, thus, it provides mnemonic recognition. The other type of entry is looked at only if the search for a mnemonic operator fails to yield a match during a CPU assembly.

This type of entry provides for recognition of an instruction according to its syntax. COMPASS analyzes the statement to be interpreted, determines the syntax of the operation and variable subfields, and again searches the table.

Instructions recognized in the mnemonic search and the information provided to the assembler for each instruction are as follows:

Pseudo instructions	The entry contains addresses to routines that perform pass one and pass two operations
PPU symbolic instructions	The entry describes the format of the instructions to be assembled
Instructions described through PPOP	The entry describes the format of the instruction to be assembled
Macro instructions	The entry directs the assembler to the location of the saved definition
Instructions described through OPSYN	The entry is a copy of the synonymous entry

For a PPU assembly, a failure to find an entry for a mnemonic operator causes an operation code error. For a CPU assembly, however, if the search for the mnemonic operator does not yield a match, COMPASS searches the operation code table again for an entry with a matching syntax. Instructions recognized in the syntactical search and the information provided to the assembler for each instruction are as follows:

CPU symbolic instructions	The entry describes the format of the CPU instruction to be assembled
Instructions described through CPOP	The entry describes the format of the CPU instruction to be assembled
Instructions defined through OPDEF	The entry directs the assembler to the location of the definition
Instructions described through CPSYN	The entry is a copy of the synonymous instruction The action taken depends on the synonymous entry

If, following the second search of the operation code table, the statement still has not been identified, the assembler takes the following action:

For a PPU assembly, it generates a 24-bit instruction of which the first 12 bits are zero.

For a CPU assembly, it generates a 30-bit zero instruction.

Although OPSYN and CPSYN pseudo instructions provide a means of rendering more than one instruction synonymous, only instructions of the same type can become synonymous. The logical division of the table between the two types of entries prevents mnemonically identified instructions from being made synonymous with syntactically identified instructions.

When a MACRO, MACROE, PPOP, or OPSYN creates an entry for a mnemonic name that is already in the table for a different instruction, the new entry takes precedence over the old entry. Similarly, when a OPDEF, CPOP, or CPSYN redescribes a syntax already in the table for a different instruction, the new entry takes precedence over the old entry. As a result, the order of precedence for operation field recognition is, from highest to lowest:

1. Programmer-created entries for mnemonically identified instructions

2. System macros, pseudo instructions, PPU symbolic machine instructions, and CMU instructions other than the IM instruction.
3. Programmer-created entries for syntactically identified instructions
4. CPU symbolic instructions and the CMU IM instruction

Example:

The following example illustrates a special case in which a macro name takes precedence over one form of a machine instruction, i. e., the form using SB4 as an operation code.

	LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30	
	SB4	MACRO	P1,P2	DEFINE MACRO NAMED SB4
		.		
		.		
		ENDM		
		.		
		.		
		SR4	A1+ABLE	CALL TO MACRO. NOT CPU INSTRUCTION
		.		
		.		
		SR3	A1+ABLE	MACHINE INSTRUCTION
	SB4	OPSYN	NIL	DISABLES MACRO BUT DOES NOT RESTORE NORMAL USE OF SR4 AS AN OPERATION CODE. EVEN IF IT WERE REDEFINED WITH OPOFF IT WOULD NOT BE RECOGNIZED. THE MACRO FORM ALWAYS TAKES PRECEDENCE.
		.		
		.		
		.		
		.		
		PURGMAC	SR4	RESTORES NORMAL USE OF SR4

## 6.1 MNEMONICALLY IDENTIFIED INSTRUCTIONS

Mnemonically identified instructions include all pseudo instructions, macro instructions, and PPU symbolic instructions whether system or programmer defined. PPOP, OPSYN, NIL, and PURGMAC provide the programmer with a means of creating or removing operation code table entries that are in the mnemonically identified format.

### 6.1.1 PPOP — PPU OPERATION CODE

The PPOP pseudo instruction defines the operation and variable fields of a PPU symbolic machine instruction and creates an operation code table entry for the instruction. COMPASS generates an octal machine instruction of the defined format whenever the PPU instruction described by the PPOP instruction is used. If the operation code table already contains an entry for the name, the new definition takes precedence over the old during assembly of the subprogram or until it is redefined. No error is flagged. Any illegal parameter in PPOP causes COMPASS to ignore the PPOP and issue a 7-type error flag.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	PPOP	ctl, val, type

**name** Mnemonic name, 1 through 8 characters

**ctl** Control of instruction assembly

<u>ctl</u>	<u>Significance</u>
0	Illegal; if used, COMPASS ignores the PPOP
1	24-bit instruction with 12-bit address and no indexing
2	12-bit instruction with signed relative address or absolute address (e.g., UJN)
3	24-bit instruction with 18-bit address (e.g., LDC)
4	12-bit instruction with 6-bit address (e.g., LDN)
5	24-bit instruction with 12-bit address and optional indexing (e.g., LDM)
6	12-bit instruction with signed relative address (e.g., SHN)
7	24-bit instruction with 12-bit address and required second field (e.g., IAM)

**val** An evaluable expression specifying the octal 4-digit operation code value; usually, only the two leftmost digits are significant. If the assembly base is M, the field is assumed to be octal.

**type** An evaluable expression specifying an integer value that COMPASS interprets as follows:

- 6 Restrict the instruction being defined to the CYBER 180 Series, CYBER 170 Series, CYBER 70/Models 71, 72, 73, and 74; COMPASS sets an error flag if the instruction being defined is used in a CYBER 70/Model 76 PPU assembly.
- 7 Restrict the instruction being defined to the CYBER 70/Model 76; COMPASS sets an error flag if the instruction being defined is used in a CYBER 180 Series, CYBER 170 Series, CYBER 70/Models 71, 72, 73, and 74 PPU assembly.

**other or omitted** The instruction is not restricted to either machine type. If the base is M, type is assumed to be octal. If type is omitted, the comma preceding it can be omitted also.

Example:

Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
		I	II	18	30
	0=0		PERIPH RASE	0	
			.		
			.		
	15	LA	FOU	15	
	40	C	FOU	40	
		STM	PPOP	5,5400+LA	
			.		
			.		
			.		
7311	5415 0040		STM	0	

### 6.1.2 OPSYN – SYNONYMOUS MNEMONIC OPERATION

The OPSYN pseudo instruction makes a name in the location field of the OPSYN synonymous with the macro, pseudo instruction or PPU mnemonic name specified in the variable field. The size of the operation code table is the only limit to the number of instructions that can be made synonymous.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name <sub>1</sub>	OPSYN	name <sub>2</sub>

The name in the variable subfield must be previously defined as a standard instruction code. After an OPSYN, either name produces equivalent results. If the location field specifies a previously defined macro or operation code, the new definition takes precedence over the old without notification. Thus, a macro defined by a name that is subsequently used in an OPSYN location field is not called when the macro name is used in the operation field. The instruction actually called is the instruction named in the variable subfield of the OPSYN. On the other hand, the old macro definition is not lost and can be restored by purging the new definition with PURGMAC.

Example:

1. An operation named CALL is synonymous with RJM.

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
CALL	OPSYN	RJM	
	.		
	.		
	.		
	CALL	=XSUBR=	PRODUCES SAME RESULTS AS IF IT WERE AN RJM

2. In the following example, a programmer wishes to use a macro named LJM for part of the program and use the real LJM for the remainder of the program.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
LJM.	OPSYN	LJM	SAVE ORIGINAL DEFINITION AS LJM.
	PURGMAC	LJM	PURGE ORIGINAL DEFINITION
	.		
	.		
LJM	MACRO	XX	
	.		
	.		
LJM	ENDM		
	.		
	.		
LJM	OPSYN	LJM.	
	.		
	.		

} CODE USING LJM MACRO

} RESTORES ORIGINAL LJM

} CODE USING ORIGINAL LJM

### 6.1.3 NIL – DO NOTHING PSEUDO INSTRUCTION

The NIL pseudo instruction resembles a no-op; it produces no code and conveys no information to the assembler. It is primarily designed for disabling a macro; it cannot be used with CPSYN. The following instructions could be used in place of NIL as nil instructions:

ENDM  
 ENDD  
 ENDIF  
 IRP

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	NIL	

A location field symbol if present is ignored.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
MACK	OPSYN	NIL	
	.		
	.		
TAG	MACK	A, B, 6, 73	
	.		



The assembler interprets each call to MACK as a NIL instruction. TAG is not defined because it becomes the location field symbol for NIL when the statement is assembled.

### 6.1.4 PURGMAC—PURGE MACROS

The PURGMAC pseudo instruction provides a means of disabling operation code entries for the named instructions for the duration of the current assembly.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	PURGMAC	name <sub>1</sub> , name <sub>2</sub> , . . . , name <sub>n</sub>

name<sub>i</sub>                      Names of mnemonic operation codes for macro definitions, pseudo instructions, or PPU instructions.

A location field symbol if present is ignored.

## 6.2 SYNTACTICALLY IDENTIFIED INSTRUCTIONS

Syntactically identified instructions apply to CPU assemblies only. The CPOP and CPSYN pseudo instructions create operation code table entries for instructions that are to be identified through recognition of their syntax, rather than through the contents of the operation field only.

### 6.2.1 CPOP — CPU OPERATION CODE

The CPOP pseudo instruction describes the syntax of a new CPU symbolic machine instruction and creates an operation code table entry for the instruction. An instruction of the defined format is generated whenever the CPU instruction described by the CPOP instruction is used. If the operation code table already contains an entry for the instruction, the new definition takes precedence over the old during assembly of the subprogram. Any illegal parameter in CPOP causes COMPASS to ignore the CPOP and issue an error flag.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sytx	CPOP	ctl, val, reg, type

sytx                      The syntax consists of a mnemonic operator and variable field descriptors. The mnemonic operator consists of two characters. The first can be any character except blank. The second character can be a register designator: A, B, or X, in which case, the operation field of the instruction is recognized as cAn, cXn, or cBn, (c is a unique character; n is 0-7); or the second character can be any other character except blank, in which case the operation field of the instruction is recognized simply by a two-character mnemonic, such as EQ.

The variable field descriptors define the order of appearance of all registers, expressions, and subfield separators that comprise the variable field of the instruction being described. It consists of none, one, two, or three of the following 22 subfield descriptors. Q represents an expression. An r represents a register letter (A, B, or X). A comma separates two descriptors; a blank terminates the syntax.

void	Q
r	rQ
-r	-rQ
$r_1+r_2$	$r_1+r_2Q$
$-r_1+r_2$	$-r_1+r_2Q$
$r_1*r_2$	$r_1*r_2Q$
$-r_1*r_2$	$-r_1*r_2Q$
$r_1/r_2$	$r_1/r_2Q$
$-r_1/r_2$	$-r_1/r_2Q$
$r_1-r_2$	$r_1-r_2Q$
$-r_1-r_2$	$-r_1-r_2Q$

For example, to describe  $-X3*B1$ , the descriptor,  $-r_1*r_2$ , would be written as  $-X*B$  whereas, to describe  $B2+ALPHA$ , the descriptor  $rQ$  would be written as  $BQ$ .

ctl Control of instruction assembly.

<u>ctl</u>	<u>Significance</u>
0	15-bit instruction
1	30-bit instruction
2	15-bit instruction, force upper before assembly
3	30-bit instruction, force upper before assembly
4	15 bit instruction, force upper after assembly
5	30-bit instruction, force upper after assembly
6	15-bit instruction, force upper before and after assembly
7	30-bit instruction, force upper before and after assembly

- val** An evaluable expression specifying a 9-bit operation code; if the base is M, val is assumed to be octal.
- reg** Three octal digits specifying the order from left to right into which register numbers are to be inserted into the i, j, k portions of a 15-bit instruction, or into the i and j portions of a 30-bit instruction. If the assembly base is M, reg is assumed to be octal.
- 1 Register number obtained from operation field
  - 2 Number of second register or only register in variable field
  - 3 Number of first two registers in variable field
  - 0 Set field to 0
- type** An evaluable expression specifying an integer value that COMPASS interprets as follows:
- 6 Restrict the instruction being defined to the 6000 Series, CYBER 180 Series, CYBER 170 Series, and CYBER 70/Models 71, 72, 73, and 74; COMPASS sets an error flag if the instruction being defined is used when MACHINE 7 has been specified.
  - 7 Restrict the instruction being defined to the 7600 or the CYBER 70/Model 76; COMPASS sets an error flag if the instruction being defined is used when MACHINE 6 has been specified.
- other  
or  
omitted The instruction is not restricted to a machine type.

If base is M, type is assumed to be octal. If type is omitted, the comma preceding it can be omitted also.

**Example:**

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
	SAX+B	CPDP	0,530B,132B	DEFINES SA1 XJ+BK
	SXXG	CPDP	1,720B,120B	DEFINES SXI XJ+K
		.		
		.		
53731		SA7	X3+B1	
722 7231000033	IAG	SX3	X1+3	

## 6.2.2 CPSYN — SYNONYMOUS CPU INSTRUCTION

The CPSYN pseudo instruction renders an instruction with the syntax given in the location field synonymous with the instruction having the syntax specified in the variable field. The only limit to the number of CPU instructions that can be made synonymous is the size of the operation code table (4123 entries).

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sytx <sub>1</sub>	CPSYN	sytx <sub>2</sub>

sytx<sub>1</sub>            Syntax of a CPU instruction (see CPOP for legal forms). If this syntax is already in the operation code table, the table entry for sytx<sub>2</sub> takes precedence over the old table entry for sytx<sub>1</sub> without notification.

sytx<sub>2</sub>            Syntax of a CPU instruction for which there must be an entry in the operation code table. Following the CPSYN, an instruction in either sytx<sub>1</sub> or sytx<sub>2</sub> produces an octal instruction of the format described by the entry for sytx<sub>2</sub>.

## 6.2.3 PURGDEF — PURGE CPU OPERATION CODE

The PURGDEF pseudo instruction provides a means of disabling syntactically-identified operation code entries for the duration of the current assembly.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	PURGDEF	sytx

sytx            Syntax of a CPU instruction (see CPOP for legal forms).

A location field symbol, if present, is ignored.

The COMPASS micro capability enables the programmer to symbolically refer to a defined character string. When used in conjunction with IFC, DUP, STOPDUP, and SET pseudo instructions, micro strings provide for varied manipulation of character strings -- testing for a particular character, counting characters, concatenation of strings, etc.

Use of a micro definition requires two steps: definition of the character string, and substitution. In this discussion, substitution rather than definition is discussed first so that the reader has a better understanding of how a definition is used when it is described.

## 7.1 MICRO SUBSTITUTION

Wherever a micro name between micro marks ( $\neq$ ) occurs in a statement other than a comment line (\* in column 1), the assembler substitutes the micro before it interprets the statement. If column 72 of the last statement read is exceeded as a result of micro substitution, the assembler creates up to a maximum of 9 continuation statements, beyond which it discards excess characters without notification on the listing. No replacement takes place if the micro name is unknown or if one of the micro marks has been omitted. If the micro name is unknown, the assembler flags a nonfatal assembly error. If the micro name is null (that is, the two micro marks are adjacent), then

1. Both micro marks are deleted, and
2. No error flag is set

Example:

A micro identified as NAM is defined as the 7 characters:

ADDRESS

A reference to NAM is in the variable field of a line:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	LOC	SA1	$\neq$ NAM $\neq$ +4	

However, before the line is interpreted, COMPASS substitutes the definition for NAM producing the following line:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	LOC	SA1	ADDRESS+4	

### NOTE

Unless the A option of the LIST pseudo instruction is enabled, the listing depicts the instruction as it was before the substitution took place.

## 7.2 MICRO DEFINITION

Pseudo instructions specifically designed for the purpose of defining micros are: MICRO, OCTMIC and DECMIC. In addition, the following pseudo instructions optionally define micros: BASE, CODE, and QUAL. Also, system or built-in micros are automatically defined by COMPASS at the start of each subprogram assembly.

### 7.2.1 MICRO — DEFINE MICRO

The MICRO pseudo instruction defines a character string and assigns a name to that string.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
micname	MICRO	$n_1, n_2, dstringd$

micname	Name by which definition is called; 1-8 characters
$n_1$	Absolute evaluable expression specifying starting character in string; when the base is M, COMPASS assumes that $n_1$ is decimal.
$n_2$	Absolute evaluable expression specifying number of characters; when the base is M, COMPASS assumes that $n_2$ is decimal.
dstringd	Delimited character string. The delimiter d is a character not used in the string.

Counting the first character after d as character 1, the assembler forms the string by extracting  $n_2$  characters starting with character  $n_1$ . If the second delimiting character occurs before count  $n_2$  is exhausted, the defined string terminates at that point. If  $n_1$  is greater than zero and  $n_2$  is omitted, zero, or negative, the defined string includes all the characters from  $n_1$  to the closing delimiter (see second example).

If  $n_1$  is omitted, zero, or negative, the defined string is empty; no substitution takes place when the micro name is referred to. That is,  $n_2$  and the character string are ignored.

A previously defined micro can be a part of a micro definition; one micro can be defined as a substring of another (see third example).

A micro can combine previously defined micros or can be a subset of another. Also, a micro defined originally as one character string can be redefined subsequently with a different character string. After the redefinition, the original character string is inaccessible.

If  $n_1$  or  $n_2$  is negative, the assembler generates a 7-type error.

Examples:

- The following MICRO defines NAME as the 19 characters beginning with A and ending with G.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
NAME	MICRO	1,19,*ALPHANUMERIC STRING*	

2. This example illustrates a blank character count. The defined string begins with A and is terminated by the closing delimiter.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
MICKY	MICRO	1,,*ALPHANUMERIC STRING*	

3. One micro can be defined as a substring of another.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
NAM1	MICRO	1,25,*AJJZ	ALPHANUMERIC STRING*
.	.	.	
.	.	.	
.	.	.	
NAM2	MICRO	7,,*#NAM1#*	SAME STRING AS IN EXAMPLES 1 AND 2

4. One micro can combine others.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
NAM1	MICRO	1,12,\$ALPHANUMERIC*	
NAM2	MICRO	1,7,X STRINGX	
NAM3	MICRO	1,,+#NAM1#X#NAM2#*	COMBINES NAM1 AND NAM2

5. A micro name can be redefined.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
MSG	MICRO	1,6,*STRING*	
.	.	.	
.	.	.	
.	.	.	} CODE USING FIRST DEFINITION
MSG	MICRO	1,10,*ALPHANUMERIC #MSG#*	
.	.	.	
.	.	.	} CODE USING SECOND DEFINITION.
.	.	.	FIRST DEFINITION IS INACCESSIBLE.

6. Micro substitution takes place before a line is assembled or examined for syntax. Thus, the following is possible.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
NAM	MICRO	1,25,* LOC	SA1 ADDRESS**
.	.	.	
.	.	.	
#NAM#1			
LOC	SA1	ADDRESS+1	

## 7.2.2 DECMIC – DECIMAL MICRO

Using a decimal conversion, the DECMIC pseudo instruction converts the expression into a character string to be saved under the name specified.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
micname	DECMIC	aexp, n

**micname** Name by which definition is called; 1-8 characters

**aexp** Absolute evaluatable expression

**n** Optional absolute evaluatable expression specifying number of characters in the defined string. The defined string is a maximum of 10 characters regardless of the magnitude of n. When base is M, COMPASS assumes that n is decimal

If n is omitted or has a zero value, the micro contains the number of characters indicated by the conversion to a maximum of 10 characters. If the converted expression has more than n (or 10) digits, the most significant digits are truncated. If the value has fewer than n digits, the string is right justified and filled with leading zeros. All numbers are treated as positive.

Example:

B has the value 1024 decimal or 2000 octal before conversion.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
V	DECMIC	B,6	
SYMBL	MICRO	1,,*#V# STORAGE NEEDED*	
SYMBL	MICRO	1,,*001024 STORAGE NEEDED*	

## 7.2.3 OCTMIC – OCTAL MICRO

Using an octal conversion, the OCTMIC pseudo instruction converts the value of the expression into a character string to be saved under the name specified.



**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
micname	OCTMIC	aexp, n

micname            Name by which definition is called; 1-8 characters

aexp                Absolute evaluatable expression

n                    Optional absolute evaluatable expression specifying number of characters in the string. The defined string is a maximum of 10 characters regardless of the magnitude of n. When base is M, COMPASS assumes n as a decimal. If n is omitted or has a zero value, the micro contains the number of characters indicated by the conversion to a maximum of 10 characters.

If the converted expression has more than n (or 10) digits, the most significant digits are truncated. If the value has fewer than n digits, the string is right justified and filled with leading zeros. All numbers are treated as positive.

**Example:**

B has the value 1024 decimal or 2000 octal before conversion.

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
V1	OCTMIC	B,6	
S1	MICRO	1,,*#V1#	ADDITIONAL STORAGE NEEDED*
S1	MICRO	1,,*002000	ADDITIONAL STORAGE NEEDED*

**7.3 PREDEFINED MICRO NAMES**

Several standard micros are predefined by the COMPASS assembler. They are available for every assembly. The programmer simply writes the micro reference as desired.

These micros are automatically defined at the beginning of each assembly, and have the default values specified below until they are redefined by the programmer; thereafter, the programmer's definition holds until the start of the next assembly.

**7.3.1 DATE**

The DATE micro contains the current date in 10 characters in one of the following forms as obtained from the operating system:

$\Delta$ yr/mo/dy.    or     $\Delta$ mo/dy/yr.

The micro reference is #DATE#.

### 7.3.2 JDATE

The automatic value of the JDATE micro is five digits yyddd, where yy is the year and ddd is the day of year at the time of assembly. Thus, JDATE is the Julian date form of DATE.

The micro reference is #JDATE#.

### 7.3.3 TIME

The TIME micro contains the current time of day in 10 characters in the following form as obtained from the operating system:

Δ hr.min.sec.

The micro reference is #TIME#.

Example:

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	TITLE	PROGRAM ASSEMBLED ON #DATE# AT#TIME#	

### 7.3.4 BASE

The automatic value of the BASE micro is a single letter D, M, or O, corresponding to the number base currently in effect (specified by the most recent BASE pseudo instruction); it is initially D.

The micro reference is #BASE#.

### 7.3.5 CODE

The automatic value of the CODE micro is a single letter A, D, E, O, or I, corresponding to the character code currently in effect (specified by the most recent CODE pseudo instruction); it is initially D.

The micro reference is #CODE#.

### 7.3.6 QUAL

The automatic value of the QUAL micro is 0 to 8 characters comprising the qualifier symbol currently in effect (specified by the most recent QUAL pseudo instruction); it is null initially and whenever the blank qualifier is in effect.

The micro reference is #QUAL#.

### 7.3.7 SEQUENCE

The automatic value of the SEQUENCE micro is 18 characters comprising the sequence field (columns 73-90) of the first line of the COMPASS source statement most recently read from the main source input file. Thus, if the current statement was read from the main source input file, SEQUENCE is the sequence field of the first line of the statement. However, if the current statement is generated (i. e., part of a macro call expansion, DUP expansion, etc.) or is read from a different file via the XTEXT pseudo instruction, then SEQUENCE is the sequence field of the first line of the statement most recently read from the main source input file.

The micro reference is ~~SEQUENCE~~.

### 7.3.8 MODLEVEL

The automatic value of the MODLEVEL micro is the value (up to 9 characters) specified by the ML parameter on the COMPASS control statement. If no ML parameter is present, the automatic value of the MODLEVEL micro is equal to that of the JDATE micro. When COMPASS is called by a compiler to process embedded COMPASS subprograms, the automatic value of the MODLEVEL micro is supplied by the calling compiler. The MODLEVEL micro is intended to be used when assembling a compiler (or COMPASS itself), to provide the compiler modification level to be placed in word 6 of each PRFX table in the binary output written by the compiler.

The micro reference is ~~MODLEVEL~~.

### 7.3.9 PCOMMENT

The automatic value of the PCOMMENT micro is the value specified by the PC parameter on the COMPASS control statement, with characters truncated from the right or blanks appended to the right, as necessary, so that the micro's length is exactly 30 characters. If no PC parameter is present, the automatic value of the PCOMMENT micro is 30 blanks. When COMPASS is called by a compiler to process embedded COMPASS subprograms, the automatic value of the PCOMMENT micro is supplied by the calling compiler. The PCOMMENT micro is intended to be used in a COMMENT pseudo instruction to specify words 8 through 10 of the PRFX table in the binary output. It may also be used, in conjunction with the \*F special symbol, to determine compiler options (debug mode, rounded arithmetic, etc.) in effect at the time of assembly.

The micro reference is ~~PCOMMENT~~.



COMPASS recognizes symbolic notation for all central processor unit (CPU) instructions for the CYBER 180 Series, CYBER 170 Series, CYBER 70 Series, 7600, and 6000 Series. For COMPASS to recognize the symbolic notation for models 810, 815, 825, 830, 835, 845, 855, 865, and 875 CPU instructions, the programmer must ensure that SYSTEXT is available to the assembler.

Some instructions in existing COMPASS programs are not valid for execution on models 810, 815, 825, 830, 835, 845, 855, 865, and 875. To detect these instructions, the programmer can specify S=AIDTEXT in the COMPASS control statement. COMPASS prints a listing of the program, flagging the invalidated instructions with a type O error. S=AIDTEXT should not be specified if the 8 option is chosen for the MACHINE pseudo instruction.

The assembler identifies each symbolic instruction according to its syntax and generates a one-paragraph 15-bit instruction or a two-paragraph 30-bit instruction. The object code for an instruction is generated in the block in use when the instruction is encountered.

### 8.1 MACHINE INSTRUCTION FORMATS

Figures 8-1 and 8-2 illustrate the formats for CPU 15-bit and 30-bit instructions generated by the assembler.

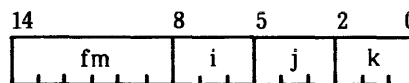


Figure 8-1. CPU 15-Bit Instruction Format

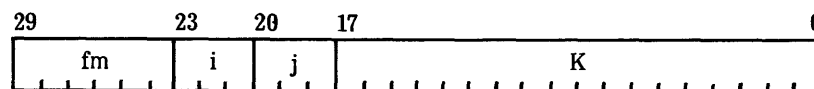


Figure 8-2. CPU 30-Bit Instruction Format

- fm      6-bit instruction code
- fmi    9-bit instruction code
- i      3-bit code (0 through 7) specifying one of eight designated registers (for example, Ai)
- j      3-bit code (0 through 7) specifying one of eight designated registers (for example, Bj)
- k      3-bit code (0 through 7) specifying one of eight designated registers (for example, Xk)
- K      18-bit integer value used as an operand, address of an operand, or branch destination address
- jk     6-bit integer value specifying a shift count or mask count

Figure 8-3 illustrates possible arrangements of one- and two-paragraph instructions in a 60-bit CPU instruction word. Generally, the assembler does not allow a two-paragraph instruction to begin in the fourth paragraph of a word.

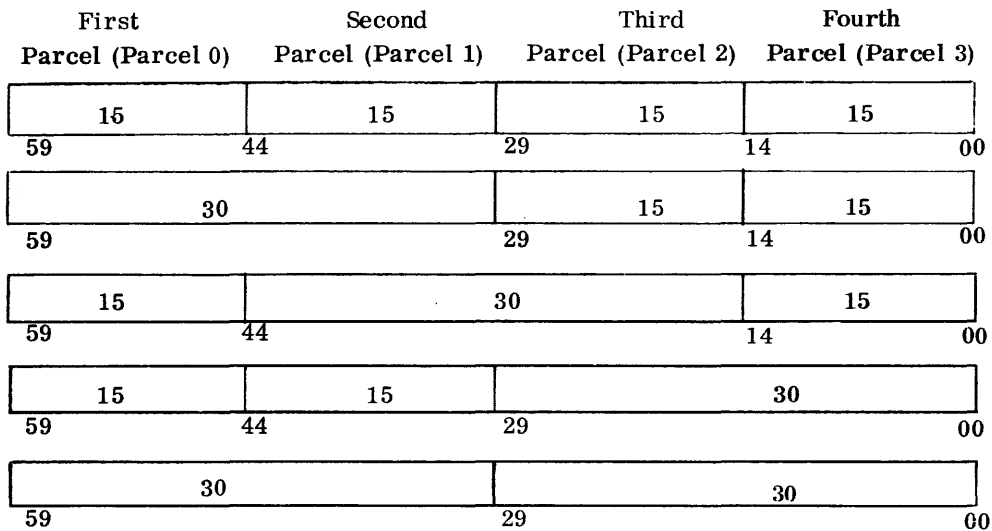


Figure 8-3. Arrangements of Instructions in a 60-bit CPU Word

When a two-parcel instruction begins in the last parcel of a word, the CYBER 170 Models 175, 176, 740, 750, 760, 865, and 875; the CYBER 70 Model 76; and the 7600 execute it as if the instruction word had a fifth parcel containing all zeros. On the CYBER 180 Computer Systems; the CYBER 170 Models 171, 172, 173, 174, 720, 730, 815, 825, 835, 845, and 855; and the CYBER 70 Models 71, 72, and 73; and the 6400, this condition causes an error exit. On the 6600 and the CYBER 70 Model 74, the CPU takes the first parcel of the current instruction.

Before it assembles an instruction that must begin in the first parcel (forced upper) and after it assembles an instruction that requires the instruction following it to be forced upper, the assembler completes a word as follows:

- Lower 15 bits remain      They are packed with a one-parcel NO (pass) instruction.
- Lower 30 bits remain      They are packed with a two-parcel SB0 B0+K instruction.
- Lower 45 bits remain      They are packed with a NO instruction and an SB0 B0+K instruction.

## 8.2 INSTRUCTION EXECUTION

### 8.2.1 6600/6700† AND CYBER 70 MODEL 74 EXECUTION

After an exchange jump start by a peripheral processor (PP) and CPU program, CPU instructions issue automatically in the original sequence, to an 8-word instruction stack. The stack can hold a program loop consisting of up to twenty-six 15-bit instructions and one 30-bit instruction.

Instructions are read from the stack, one at a time, and issued to the functional units (table 8-1) for execution. A scoreboard reservation system in CPU control keeps a current log of which units and operating registers are reserved for computation results from functional units.

†The 6700 also includes a 6400-type central processor unit

TABLE 8-1. CYBER 70 MODEL 74 AND 6000/7600 FUNCTIONAL UNITS

Unit	General Function
Branch	Handles all jumps or branches from the program.
Boolean	Handles the basic logical operations of transfer, logical product, logical sum, and logical difference.
Shift	Executes operations basic to shifting. This includes left (circular) and right (end-off sign extension) shifting, and normalize, pack, and unpack floating point operations. The unit also includes a mask generator.
Floating Add	Performs single or double precision floating point addition and subtraction on floating point operands.
Long Add	Performs addition and subtraction of two 60-bit fixed point operands
Floating Multiply	Performs single or double precision floating point multiplication on floating point operands
Floating Divide	Performs single precision floating point division of floating point operands; also counts the number of 1 bits in a 60-bit word.
Increment	Performs one's complement addition and subtraction of 18-bit operands.

Each functional unit executes several instructions, but only one at a time. Some branch instructions require two units; the second unit receives direction from the branch unit.

The rate of issuing instructions varies from the maximum of one instruction every 100 nanoseconds (one minor cycle). Sustained issuing at this rate may not be possible because of functional unit and CM conflict or because of serial rather than simultaneous operation of units. Program run time can be decreased by efficient use of the units. Instructions that are not dependent on previous steps may be arranged or nested in program areas where they may be executed concurrently with other operations to eliminate dead spots in the program and increase the instruction issue rate.

The following steps summarize instruction issuing and execution:

An instruction is issued to a function unit when:

Specified functional unit is not reserved.

Specified result register is not reserved for a previous result.

Instructions are issued to functional units at minor cycle intervals when no reservation conflicts are present.

Instruction execution starts in a functional unit when both operands are available. Execution is delayed when an operand is a result of a previous step which is not complete.

No delay occurs between the end of a first unit and the start of a second unit which is waiting for the results of the first.

After a branch instruction is initiated, no further instructions are issued until the branch has been executed. In the execution of a branch instruction, the branch unit uses:

Increment unit to form the instructions that branch to  $K + B_i$  and branch to  $K$  if  $B_i \dots$

Long add unit to perform the instructions that branch to  $K$  if  $X_j \dots$

Time spent in the long add or increment units is part of total branch time.

Read central memory access time is computed from the end of increment unit time to the time an operand is available in X operand register. Minimum time is 500 nanoseconds assuming no central memory bank conflict.

### 8.2.2 CYBER 180 COMPUTER SYSTEMS; CYBER 170 MODELS 171, 172, 173, 174, 720, 730, 815, 825, 835, 845, AND 855; CYBER 70 MODELS 71, 72, AND 73; AND 6200, 6400, 6500 EXECUTION

The CYBER 180 Computer Systems; the CYBER 170 Models 172, 173, 174, 720, 730, 815, 825, 835, 845, and 855; the CYBER 70 Models 71, 72, and 73; and the 6200, 6400, and 6500 systems CPU has a unified arithmetic unit, rather than separate functional units as in the 6600 system. Instructions in the CPU are executed sequentially.

#### NOTE

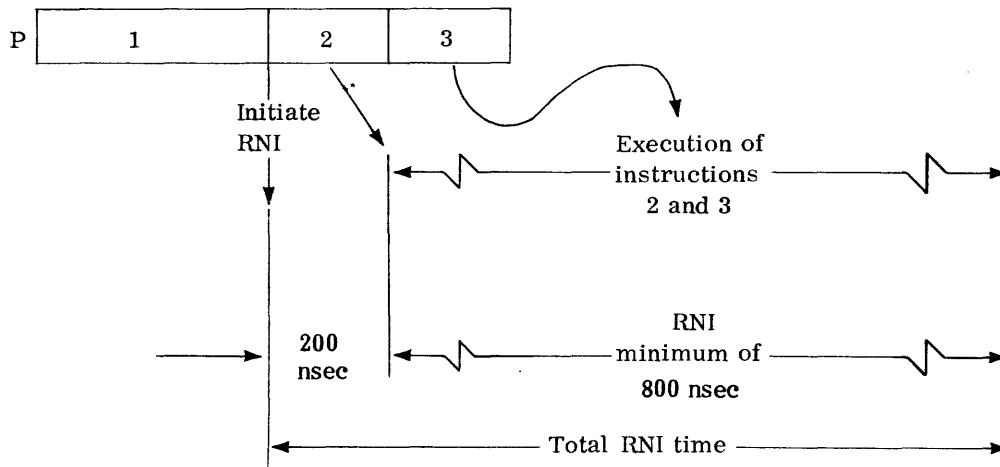
Unless otherwise stated, the remainder of this section applies to all the models listed above, except models 810, 815, 825, 830, 835, 845, and 855.

For efficient coding in the central processor unit:

Always attempt to place jump instructions in the upper portion of the instruction word to avoid both the additional time for RNI (read next instruction, 2 minor cycles) and the possibility of a memory bank conflict with  $(P + 1)$ .

Where possible, place load/store instructions in the lower two portions to avoid lengthening execution times.

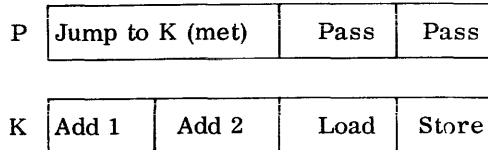
Reading the next instruction words of a program from central memory, RNI, is partially concurrent with instruction execution. RNI is initiated between execution of the first and second instructions of the word being processed. Initiating RNI operation requires two minor cycles; the remainder of the RNI is parallel in time with execution of the remaining instructions in the word:





In calculating execution times, two minor cycles are added to each instruction word in a program to cover the RNI initiation time. Exceptions are the return jump and the jump instructions (in which the jump condition is met) when they occupy the upper position of the instruction word. Since the times for these instructions already include the time required to read the new instruction word at the jump address, no additional time is consumed.

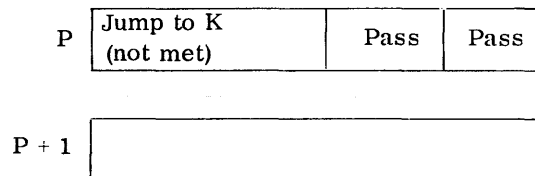
Example:



<u>Instruction</u>	<u>Minor Cycles Required</u>
Jump	13
Add 1	5
RNI Initiation	2
Add 2	5
Load	12
Store	10
Total Time	47 minor cycles

After RNI is initiated (between the first and second instructions of the word), a minimum of eight minor cycles elapses before the next instruction word is available for execution. Even if the lower order positions of the word should require less than eight minor cycles, a minimum of eight minor cycles is allowed.

Example:



### 8.2.3 CYBER 170 MODELS 175, 176, 740, 750, 760, 865, AND 875; CYBER 70 MODEL 76; AND 7600 EXECUTION

Execution of an arithmetic or logical machine instruction takes place in one of nine functional units in the computation section of the CYBER 170 Models 175, 176, 740, 750, 760, 865, and 875; the CYBER 70 Model 76; and the 7600 CPU. Each is a specialized unit with algorithms for a portion of the CPU instruction execution. Table 8-2 lists the general function of each unit. A number of functional units can be in operation at the same time.

**TABLE 8-2. CYBER 170 MODELS 175, 176, 740, 750, 760, 865, AND 875;  
CYBER 70 MODEL 76; AND 7600 FUNCTIONAL UNITS**

Unit	General Function
Boolean	Handles the basic logical operations of transfer, logical product, logical sum, and logical difference. It also performs the pack and unpack floating point operations.
Shift	Executes operations basic to shifting. This includes left (circular) and right (end-off sign extension) shifting, and mask generation.
Normalize	Performs the normalize operations.
Floating Add	Performs single or double precision floating point addition or subtraction on floating point operands.
Long Add	Performs integer addition or subtraction of two 60-bit fixed point operands.
Floating Multiply	Performs single or double precision floating point multiplication on floating point operands.
Floating Divide	Performs single precision floating point division of floating point operands.
Population Count	Counts the number of 1 bits in a 60-bit word.
Increment	Performs one's complement addition and subtraction of 18-bit operands.

A functional unit receives one or two operands from operating registers at the beginning of instruction execution and delivers the result to the operating registers after performing the function. The functional units do not retain any information for reference in subsequent instructions. The units operate in three-address mode with source and destination addressing limited to the operating registers.

Except for the floating multiply and divide units, all functional units have one clock period segmentation. This means that the information arriving at the unit, or moving within the unit, is captured and held in a new set of registers at the end of every clock period. It is therefore possible to start a new set of operands for unrelated computation into a functional unit each clock period even though the unit may require more than one clock period to complete the calculation. This process may be compared to a delay line in which data moves through the unit in segments to arrive at the destination in the proper order but at a later time. All functional units perform their algorithms in a fixed amount of time. No delays are possible once the operands have been delivered to the front of the unit.

The floating multiply unit has a two clock period segmentation. Operands may enter the multiply unit in any clock period providing there was no multiply operation initiated in the preceding clock period.

The floating divide unit is the only functional unit in which an iterative algorithm is executed. There is little segmentation possible in this unit. However, to increase execution speed, the beginning of a new divide operation can follow a previous divide operation by 18 clock periods for a gain of 2 clock periods.

Instructions involving storage references for operands or program branching are difficult to time. Program branching within the instruction stack causes no storage references and small program loops can therefore be precisely timed.

### 8.3 OPERATING REGISTERS

Twenty-four registers minimize memory references for arithmetic operands and results:

Function	Identity	Length	Number
Operand Registers	X0 - X7	60 Bits	8
Address Registers	A0 - A7	18 Bits	8
Index Registers	B0 - B7	18 Bits	8

A register is reserved if it is the destination of an instruction that has been initiated but has not been completed. A register is free in the clock period (or minor cycle) following the store into it.

#### 8.3.1 X REGISTERS

Eight 60-bit X registers in the computation section of the CPU designated X0, X1, . . . , X7 are the principal data handling registers for computation. Data flows from these registers to the SCM (CM) and the LCM (not ECS). Data also flows from SCM (CM) and LCM (not ECS) into these registers. All 60-bit operands involved in computation must originate and terminate in these registers.

Operands and results transfer between SCM (CM) and these registers as a result of placing SCM (CM) into corresponding address registers.

On the CYBER 170 Model 176, the CYBER 70 Model 76, and the 7600, the X registers also serve as address registers for referencing single words from LCM. X0 is used as the LCM relative starting address in a block copy operation.

#### 8.3.2 A REGISTERS

Eight 18-bit A registers in the computation section of the CPU, designated as A0, A1, . . . , A7, are essentially SCM (CM) operand address registers. With the exception of A0 and X0, A registers are associated one-for-one with the X registers. Placing a quantity into an address register A1 - A5 causes an immediate SCM (CM) read reference to that relative address and sends the SCM (CM) word to the corresponding operand register X1 - X5. Similarly, placing a value into address register A6 or A7 causes the word in the corresponding X6 or X7 operand register to be written into that relative address of SCM (CM).

The A0 and X0 registers operate independently of each other and have no connection with SCM (CM). A0 is used as the relative SCM (CM) starting address in a block copy operation and for scratch pad or intermediate results.

#### 8.3.3 B REGISTERS

Eight 18-bit B registers in the computation section of the CPU designated as B0, B1, . . . , B7 are primarily indexing registers for controlling program execution. Program loop counts can be incremented and decremented in these registers.

Program addresses may be modified on the way to an A register by adding or subtracting B register quantities. The B register also holds shift counts for pack and normalize operations and the channel number for channel status requests.

B0 always contains positive zero; that is, B0 is held clear. Often as a programming convention, B1 or B7 contains positive 1. See the B1=1, the B7=1, and the R= pseudo instructions.

## 8.4 SYMBOLIC NOTATION

This section describes notation used for coding symbolic CPU machine instructions. Instructions are listed in groups according to function. Instructions unique to a computer system are identified as such in table 8-3. These instructions can be assembled on any machine but will execute properly on the noted machine only. Table 8-4 lists the functional unit, if any, in which each instruction executes. For details and special conditions arising during instruction execution, refer to the relevant hardware system reference manual.

TABLE 8-3. CPU INSTRUCTION/MACHINE MODEL CORRESPONDENCE

Mnemonic Code	Machine Model Number			
	810, 815, 825, 830, 835, 845, 855, 865, and 875	76 and 7600	171, 172, 173, 174, 175, 720, 730, 740, 750, and 760; 71, 72, 73, and 74; and 6000 Series	176
AXi	X	X	X	X
BXi	X	X	X	X
CC†				
CR	X			
CU†				
CW	X			
CXi	X	X	X	X
DF	X	X	X	X
DM†				
DXi	X	X	X	X
EQ	X	X	X	X
ES		X		
FXi	X	X	X	X
GE	X	X	X	X
GT	X	X	X	X
IBj		X		X
ID	X	X	X	X
IM†				
IR	X	X	X	X
IXi	X	X	X	X
JP	X	X	X	X
LE	X	X	X	X
LT	X	X	X	X
LXi	X	X	X	X

TABLE 8-3. CPU INSTRUCTION/MACHINE MODEL CORRESPONDENCE (Contd)

Mnemonic Code	Machine Model Number			
	810, 815, 825, 830, 835, 845, 855, 865, and 875	76 and 7600	171, 172, 173, 174, 175, 720, 730, 740, 750, and 760; 71, 72, 73, and 74; and 6000 Series	176
MD†				
MI	X	X	X	X
MJ		X		X
MXi	X	X	X	X
NE	X	X	X	X
NG	X	X	X	X
NO	X	X	X	X
NXi	X	X	X	X
NZ	X	X	X	X
OBj		X		X
OR	X	X	X	X
PL	X	X	X	X
PS	X		X	
PXi	X	X	X	X
RE	X		X	
RI		X		X
RJ	X	X	X	X
RL		X		X
RO		X		X
RXi	X	X	X	X
RXj	X	X		X
SAi	X	X	X	X
SBi	X	X	X	X
SXi	X	X	X	X
TBj		X		X
UXi	X	X	X	X
WE	X		X	
WL		X		X
WXj	X	X		X
XJ	X		X	X
ZR	X	X	X	X
ZXi	X	X	X	X

†CMU instruction: Compare/Move Unit available on CYBER 170 Models 172, 173, 174, 720, and 730. Models 810, 815, 825, 830, 835, 845, and 855 support CMU instructions through simulation.

TABLE 8-4. CPU INSTRUCTION/FUNCTIONAL UNIT CORRESPONDENCE

Mnemonic	Operation Code	Functional Unit	
		74, 6600, and 6700	175, 176, 740, 750, and 760; 76; and 7600
AXi		Shift	Shift
BXi		Boolean	Boolean
CR		None	None
CW		None	None
CXi		Divide	Pop
DF		Branch	None
DXi	32ijk	FP Add	FP Add
DXi	33ijk	FP Add	FP Add
DXi	42ijk	Multiply	Multiply
EQ		Branch	None
ES		†	None
FXi	30ijk	FP Add	FP Add
FXi	31ijk	FP Add	FP Add
FXi	40ijk	Multiply	Multiply
FXi	44ijk	Divide	Divide
GE		Branch	None
GT		Branch	None
IBj		†	None
ID		Branch	None
IR		Branch	None
IXi	36ijk	Long Add	Long Add
IXi	37ijk	Long Add	Long Add
IXi	42ijk	Multiply	Multiply
JP		Branch	None
LE		Branch	None
LT		Branch	None
LXi		Shift	Shift
MI		Branch	None
MJ		†	None
MXi		Shift	Shift
NE		Branch	None
NG		Branch	None
NO		None	None
NXi		Shift	Normalize
NZ		Branch	None
OBj		†	None
OR		Branch	None
PL		Branch	None
PS		Branch	†
PXi		Shift	Boolean

TABLE 8-4. CPU INSTRUCTION/FUNCTIONAL UNIT CORRESPONDENCE (Contd)

Mnemonic	Operation Code	Functional Unit	
		74, 6600, and 6700	175, 176, 740, 750, and 760; 76; and 7600
RE		Branch	†
RI		†	None
RJ		Branch	None
RL		†	None
RO		†	None
RXi	34ijk	FP Add	FP Add
RXi	35ijk	FP Add	FP Add
RXi	41ijk	Multiply	Multiply
RXi	45ijk	Divide	Divide
RXj		†	None
SAi		Increment	Increment
SBi		Increment	Increment
SXi		Increment	Increment
TBj		†	None
UXi		Shift	Boolean
WE		Branch	None
WL		†	None
WXj	015jk	†	None
XJ	01300	None	None
XJ	013jk	Branch	†
ZR		Branch	None
ZXi		Shift	Normalize

†Instruction not supported for this model.

The location field of a symbolic machine instruction optionally contains a location symbol. When the symbol is present, it is assigned the value of the location counter after the force upper (if any) occurs.

The operation field of a symbolic CPU machine instruction contains a mnemonic operator, the last two characters of which are often a register designator.

The variable field contains one, two, or three subfields. For 15-bit instructions, subfields take the forms:

$$\left. \begin{array}{l} r \\ -r \\ r,r \end{array} \right\} \quad r \text{ is a register designator}$$

$$\left. \begin{array}{l} r \text{ op } r \\ -r \text{ op } r \end{array} \right\} \quad \text{op is a register operator } + - * /$$

$$jk \quad jk \text{ is an absolute expression specifying a shift count or mask bit count. If the expression value is in the range } -60 \text{ to } -0, \text{ inclusive, COMPASS adds } 60 \text{ to it. If it is less than } -60 \text{ or greater than } 63, \text{ COMPASS sets a warning flag and uses the low-order } 6 \text{ bits of the expression value.}$$





can be written

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
26123		UX1, B2	X3	

The instructions are identical to the assembler.

Similarly, the following instructions are regarded as identical. Use of this feature is optional.

	LOCATION	OPERATION	VARIABLE	COMMENTS
0423010641		EQ	B2, B3, K	
0423010641		EQ, B2	B3, K	
0423010641		EQ, B2, B3	K	
0423010641		EQ, B2, B3, K		

#### 8.4.1 PROGRAM STOP OR EXCHANGE JUMP INSTRUCTION

The CEJ/MEJ Panel Switch determines whether this instruction causes the central processor unit to halt or to execute an exchange jump. The DISABLE position disables the central exchange jump or the monitor exchange jump. In this case, the instruction is illegal for a CYBER 170 Model 175. For all other systems, PS halts the central processor unit at the current step in the program. An exchange jump is necessary to restart the central processor unit. The ENABLE position enables the jump capabilities for all systems. In this case, PS causes an exchange jump to monitor address (MA) in the exchange package. For the CYBER 180 Computer Systems and the CYBER 170 Models 176, 815, 825, 835, 845, and 855, exchange jumps are always enabled. For 6000 series systems, the CRJ/MEJ switch is ignored; PS always causes the central processor unit to halt. The job continues to hold a control point until the time limit is satisfied; at that time the job aborts.

The contents of the location field become a sub-subtitle on the assembler listing. The assembler forces upper before and after assembling a PS instruction.

Format:

Operation	Variable	Description	Size	Octal Code
PS		Program stop or exchange jump to (MA)	30 bits	00000 00000
PS	K	Program stop or exchange jump to (MA)	30 bits	0000K

Example:

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
0000000000		PS		

### 8.4.2 ERROR EXIT INSTRUCTION

ES execution is treated as an error condition and the machine sets the program range condition flag in the PSD register. The condition flag then generates an error exit request which causes an exchange jump to address (EEA). All instructions issued prior to this instruction are run to completion. Any instruction following this instruction in the current instruction word is not executed. When all operands have arrived at the operating registers as a result of previously issued instructions, an exchange jump occurs to the exchange package designated by (EEA).

The i, j, and k designators, which are ignored by the computation section, are set to zero by the assembler. The program address stored in the exchange package on the terminating exchange jump is advanced one count from the address of the current instruction word (P=P+1). This is true regardless of which parcel of the current instruction word contains the error exit instruction.

The error exit instruction is not intended for use in user program code. The program range condition flag is set in the PSD register to indicate that the program has jumped to an area of the SCM field which may be in range but is not valid program code. This should occur when an incorrectly coded program jumps into an unused area of the SCM field or into a data field. The program range condition flag is also set on the condition of a jump to address zero. These conditions can be determined on the basis of the register contents in the exchange package. The existence of an error exit condition resulting from execution of this instruction can thus be deduced.

The location field of an ES instruction becomes a sub-subtitle on the assembler listing.

A force upper occurs after the ES instruction.

Format:

Operation	Variable	Description	Size	Octal Code
ES		Error exit to EEA	15 bits	00000
ES	K	Error exit to EEA	15 bits	00000

Example:

Code Generated

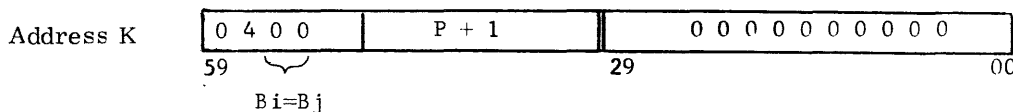
00000

LOCATION	OPERATION	VARIABLE	COMMENTS
1	ES	18	30

### 8.4.3 RETURN JUMP INSTRUCTION

When this instruction is executed, an unconditional jump to the current address plus one (P)+1 is stored in the upper half of relative address K in SCM and control then transfers to K+1 for the next instruction. The lower half of the stored word is all zeros. The instruction always branches out of the instruction stack and voids all instructions currently in the instruction stack.

After the instruction is executed the octal word at K is:



This instruction is intended for transferring control to a subroutine between execution of the current instruction word and the following instruction word. Instructions appearing after the return jump instruction in the current instruction are not executed. The called subroutine must exit at address K in CM (SCM). A jump to address K of the branch routine returns the program to the original sequence. The assembler sets the unused j designator to zero.

A force upper occurs after the instruction is assembled.

Format:

Operation	Variable	Description	Size	Octal Code
RJ	K	Return jump to K	30 bits	0100K

Example:

Code Generated

0100002374 +

	LOCATION	OPERATION	VARIABLE	COMMENTS
I		11	18	30
		RJ	HELP	

#### 8.4.4 ECS/UEM INSTRUCTIONS

These instructions initiate either a read or write operation to transfer (Bj)+K 60-bit words between extended memory (ECS or UEM) and central memory (CM). The initial extended memory address is (XO)+RA<sub>E</sub>; the initial CM address is (AO)+RA<sub>C</sub>.

##### NOTE

For the CYBER 180 Computer Systems and the CYBER 170 Models 815, 825, 835, 845, and 855, these instructions are UEM block copy instructions. For the CYBER 170 Models 865 and 875, the selection of the ECS or UEM depends on the state of the UEM enable flag. This flag is one bit in the 6-bit flag register in the exchange sequence. If the enable flag is set, transfer is between UEM and CM; if the enable flag is clear, transfer is between ECS and CM.

The assembler forces upper before assembling an RE or WE instruction.

If no error occurs, the next instruction executed is the first instruction in the current address plus one (P)+1.

Three error conditions cause an error exit to the lower-order 30 bits of the instruction word containing the RE or WE instructions. These 30 bits should always hold a jump to an error routine. The conditions are:

Parity errors when reading ECS. If a parity error is detected, the entire block of data is transferred before the exit is taken.

The ECS bank from/to which data is to be transferred is not available because the bank is in maintenance mode, or the bank has lost power. If either of these conditions exists on an attempted read or write, an immediate error exit is taken.

An attempt to reference a nonexistent address. On an attempted write operation, no data transfer occurs and an immediate error exit is taken. If the attempted operation is a read, and addresses are in range, zeros are transferred to central memory. This is a convenient high-speed method of clearing blocks of central memory.

On a CYBER 170 Model 176, action in the case of error depends on the operating system being run. Under SCOPE 2, error processing is just as for the RL and WL instructions (see LCM Block Copy Instructions). Under NOS, an error causes the job to abort. Under NOS/BE, an error exit to the lower 30 bits of the instruction word takes place. This action is provided by the operating system, not by the hardware.

For additional information about ECS instructions, refer to the 7030 Extended Core Storage Reference Manual.

Format:

Operation	Variable	Description	Size	Octal Code
RE	Bj	Read extended memory	30 bits	011j0 0000
RE	K	Read extended memory	30 bits	0110K
RE	Bj+K	Read extended memory	30 bits	011jK
WE	Bj	Write extended memory	30 bits	012j0 0000
WE	K	Write extended memory	30 bits	0120K
WE	Bj+K	Write extended memory	30 bits	012jK

Example:

Code Generated

0110002000  
 0117001000  
 0125001000

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	RE	2000B	
	RE	B7+1000B	
	WE	1000B+B5	

### 8.4.5 LCM BLOCK COPY INSTRUCTIONS

Block copy instructions move quantities of data between LCM and SCM as quickly as possible. All activity in the CPU other than I/O word requests is stopped during a block copy operation. All instructions issued prior to a block copy instruction are executed to completion and no further instructions issue until the block copy is nearly completed. As a result of these restrictions the data flow between LCM and SCM can proceed at the rate of one 60-bit word each clock period. When an I/O multiplexer word request for SCM occurs during this transfer, the data flow is interrupted for one clock period. The I/O word address is inserted in the stream of addresses to the SAS, and the addresses for the block copy are resumed with a minimum of a one clock period delay. An additional delay will occur if the I/O reference causes a bank conflict in SCM.

The length of the block is determined by adding the quantity K to the contents of register Bj. Either quantity may be used as an increment or decrement. The result is an 18-bit integer which is truncated to a 10-bit quantity. Thus, a maximum block size is 1777g. (For example, if the result of the add is 003000g, the instruction transfers 1000g words.) No error indications are given when this occurs unless the field length is exceeded causing a block range error. If the block length is zero, the instruction becomes a do-nothing instruction; the condition is not error flagged.

Relative source or destination addresses begin at (A0) in the SCM and at the relative LCM address determined from the lowest order 19 bits of (X0). If (X0) is negative, the 19 bits are treated as a positive integer. If the sum of (X0<sub>18-00</sub>) and the block count exceeds the (FLL), the copy is not executed and the

LCM block range condition flag is set in the PSD register. Similarly, if the sum of (A0) and the block exceeds (FLS), the copy is not executed and the SCM block range condition flag is set in the PSD register.

Under SCOPE 2, COMPASS will truncate a block copy instruction if it begins in the last parcel and its K field is zero. Under such conditions, a block copy is a 15-bit instruction.

Under NOS and NOS/BE, COMPASS forces upper after assembling an RL or WL instruction.

Any error condition occurring during execution of a block copy instruction causes a flag to be set in the PSD register but does not interrupt the block copy instruction. No further instructions are issued during block transfer of data. Instructions already issued are completed; all other activity, with the exception of I/O word requests, stops.

On a CYBER 170 Model 176, if no error takes place, the next instruction executed is the first instruction in the current address plus one (P)+1. Action in the case of error depends on the operating system being run. Under SCOPE 2, error processing is just as for any program running on the CYBER 70 Model 76, as described in the SCOPE 2 Reference Manual listed in the preface. Under NOS, an error causes the job to abort. Under NOS/BE, an error exit to the lower 30 bits of the instruction word takes place. This action is provided by the operating system, not by the hardware.

**Format:**

Operation	Variable	Description	Size	Octal Code
RL	Bj	Block copy (Bj) words from LCM to SCM	30 bits	011j0 00000
RL	K	Block copy (K) words from LCM to SCM	30 bits	0110K
RL	Bj±K	Block copy (Bj) ± K words from LCM to SCM	30 bits	011jK
WL	K	Block copy (K) words from SCM to LCM	30 bits	0120K
WL	Bj	Block copy (Bj) words from SCM to LCM	30 bits	012j0 00000
WL	Bj±K	Block copy (Bj) ± K words from SCM to LCM	30 bits	012jK

**Example:**

Code Generated

0115001000

0110002000

0124777677

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
	RL	1000B+B5	
	RL	2000B	
	WL	B4-100B	

**8.4.6 EXCHANGE JUMP INSTRUCTION**

This instruction unconditionally exchange jumps the central processor, regardless of the state of the monitor flag bit. Instruction action differs, however, depending on whether the monitor flag bit is set or clear.

This instruction is not legal for CYBER 170 Models 175, 740, 750, and 760 if the MEJ/CEJ switch is in the DISABLE position or if the instruction does not reside in parcel 0 of the instruction word.

Operation is as follows:

**Monitor flag bit clear:** The starting address for the exchange is taken from the 18-bit Monitor Address register. This starting address is an absolute address. During the exchange, the monitor flag bit is set.

**Monitor flag bit set:** The starting address for the exchange is the 18-bit result formed by adding K to the contents of register Bj. This starting address is an absolute address. During the exchange, the monitor flag bit is cleared.

For additional information, refer to the appropriate hardware reference manual.

The assembler forces upper before and after assembling an XJ instruction.

**Format:**

Operation	Variable	Description	Size	Octal Code
XJ		Exchange jump to MA if in program mode	30 bits	01300 00000
XJ	Bj	Exchange jump to (Bj); flag set	30 bits	013j0 00000
XJ	K	Exchange jump to K; flag set	30 bits	0130K
XJ	Bj±K	Exchange jump to (Bj) ± K; flag set	30 bits	013jK

**Example:**

Code Generated

0130000000  
 0130001000  
 0135000600

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	XJ		
	XJ	1000B	
	XJ	B5+600B	

### 8.4.7 EXCHANGE EXIT INSTRUCTION

This instruction is used for calling a system monitor program for input/output, monitor calls, etc. and has priority over all other types of exchange jump requests. If an I/O interrupt request or an error exit request occurred prior to execution of this instruction, it is denied and the exchange jump specified by the MJ is executed. The rejected interrupt request is not lost, however. The conditions that caused it are reinstated when the exchange package enters its next execution interval.

The normal termination for an exchange package execution interval is through execution of an exchange instruction (MJ). The MJ instruction voids the instruction word stack. Any instructions remaining in the stack are not executed. The exit mode flag in the PSD register determines the source of the exchange package as follows:

**Exit mode flag set:** When the exit mode flag is set, the MJ instruction causes the current program sequence to terminate with an exchange jump to a relative address in the SCM field for the current program. The exchange package is located at relative address (Bj) + K. An overflow of the lowest order 16 bits of this result causes an error condition that is not sensed in the hardware. Should a program erroneously execute an exchange exit instruction with an overflow condition, the exchange jump sequence begins at the absolute SCM address corresponding to the lowest order 16 bits of this sum. This 30-bit form of MJ is privileged to a monitor program.

Exit mode flag not set: When the exit mode flag is not set, the object program terminates the execution interval with a 15-bit form of the MJ instruction. The normal exit address (NEA) is the absolute address of the exchange package. This is an absolute address in SCM and is generally not in the SCM field for the current program. This form of the MJ instruction has a blank variable field; the assembler sets the j and k designators to zero.

The system makes no protective tests on the exchange jump address.

All operating register values, program addresses, and mode selections are preserved in the exchange package for the object program so that the object program can be continued at a later time. The program address in the object program exchange package is advanced one count from the address of the instruction word containing the exchange exit instruction. The monitor program normally resumes the object program at this address.

The assignment of (NEA) is a responsibility of the system monitor program. If (NEA) has more than 16 bits of significance, the upper bits are discarded and the lower 16 bits are used as the absolute address in SCM for the exchange jump. A force upper occurs after the instruction is assembled.

Format:

Operation	Variable	Description	Size	Octal Code
MJ		Exchange exit to NEA if exit flag clear	15 bits	01300
MJ	Bj	Exchange exit to (Bj) if exit flag set	30 bits	013j0 00000
MJ	Bj_K	Exchange exit to (Bj) + K if exit flag set	30 bits	013jK
MJ	K	Exchange exit to K if exit flag set	30 bits	0130K

Example:

Code Generated

01300

0134000500

0136777477

0130000600

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
	MJ		
	MJ	B4+500B	
	MJ	-300B+B6	
	MJ	600B	

#### 8.4.8 DIRECT LCM TRANSFER INSTRUCTIONS

A single word transfer either reads one 60-bit word from LCM and enters this word into an X register or writes one 60-bit word directly into LCM from an X register.

The execution time for transferring a word from LCM to an X register depends on whether the requested word already resides in one of the bank operand registers. A read LCM instruction for a word not currently residing in a bank operand register will require 17 clock periods for delivering a field of eight 60-bit words to the designated X register. A read LCM instruction for a word already residing in an LCM bank operand register as a result of a previous instruction will require three clock periods to deliver the requested word to the designated X register. Thus, although the first 60-bit word will require 17 clock periods, the second through eighth words in the same LCM word require three clock periods each. This means that consecutive LCM operands are available, on an average, every five clock periods as opposed to SCM operands at eight clock periods.

The LCM address is determined from the low order 19 bits of Xk. Even if (Xk) is negative, the 19 bits are treated as a positive integer. If the address exceeds the field length (FLL), the word transfer does not take place and the LCM direct range condition flag is set in the PSD register. Xj is either the source or destination register.

Instructions are buffered to the extent that each issues in one minor cycle unless a previous LCM reference is in process. When an RX instruction issues, the LCM busy flag is set and remains set until the requested word is delivered.

For a write (WX) instruction, if the word cannot be entered immediately in the proper bank operand register, it is held in the LCM write register until the bank operand register is free.

Format:

Operation	Variable	Description	Size	Octal Code
RXj	Xk	Read LCM at (Xk) and set Xj	15 bits	014jk
WXj	Xk	Write (Xj) into LCM at (Xk)	15 bits	015jk

Example:

Code Generated

01465

01570

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RX6	X5	
		WX7	X0	

### 8.4.9 DIRECT UEM TRANSFER INSTRUCTIONS

A single word transfer either reads one 60-bit word from UEM and enters that word into the specified X register, or writes one word into UEM from the specified X register.

Format:

Operation	Variable	Description	Size	Octal Code
RXj	Xk	Read UEM at (Xk) + RA <sub>e</sub> to Xj	15 bits	014jk
WXj	Xk	Write (Xj) to UEM at (Xk) + RA <sub>e</sub>	15 bits	015jk

Example:

Code Generated

01412

01512

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RX1	X2	
		WX1	X2	



### 8.4.10 RESET INPUT CHANNEL BUFFER INSTRUCTION

This instruction initiates a new record transmission from a PPU to SCM. This instruction prepares the input channel (Bk) buffer for a new record transmission from a PPU to SCM. The instruction clears the input channel buffer address and resets the input channel assembly counter to the first 12-bit position in the SCM word.

This instruction is intended to be privileged to an input routine, that is, one that terminates a record of incoming data and prepares for the next record.

The input routine removes the data in the input channel buffer and then executes this instruction to prepare the buffer for the next incoming record. This instruction is effective only if the monitor mode flag is set in the program status register. If the monitor mode flag is cleared, this instruction becomes a pass instruction. When this instruction issues, it will execute the required channel functions without regard to the current status or activity at the input channel buffer.

The lowest order four bits of (Bk) are used in this instruction. The higher order bits are ignored. If higher order bits are set in (Bk) the lowest order four bits are masked out and used to determine the channel number. If (Bk) is zero, this instruction becomes a pass instruction.

Two or more consecutive RI instructions referring to different channels will issue in consecutive clock periods with no interference resulting in the multiplexer. If two consecutive instructions refer to the same channel, they repeatedly perform the same function but do not cause interference in the multiplexer.

Format:

Operation	Variable	Description	Size	Octal Code
RI	Bk	Reset input channel (Bk) buffer	15 bits	0160k

Example:

Code Generated

01607

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RI	B7	

### 8.4.11 SET REAL-TIME CLOCK INSTRUCTION

This instruction reads the contents of the CPU clock period counter (real-time clock) and places them in Bj. The 18-bit clock counter advances one count in two's complement mode for each clock period. The 2<sup>17</sup> bit is the overflow bit. The CPU is interrupted when the overflow bit is set. When the interrupt is handled, the bit is cleared. It permits measurement of CPU execution.

Format:

Operation	Variable	Description	Size	Octal Code
TBj		Set Bj to current clock time	15 bits	016j0
TBj	K	Set Bj to current clock time; K is ignored.	15 bits	016j0

**Example:**

Code Generated

01670

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		TB7		

**8.4.12 RESET OUTPUT CHANNEL BUFFER INSTRUCTION**

This instruction initiates a new record transmission from SCM to PPU. It clears the output channel (Bk) buffer address and disassembly counter, transmits a record pulse over the output channel data path to the PPU, and initiates an SCM reference for the first word to be transmitted.

This instruction is intended for execution in an output routine to initiate a new record transmission over an output channel data path. The output channel buffer is normally inactive when this instruction is executed. The output channel buffer is loaded with the data for the next record, and this instruction is executed to initiate the transmission. The record pulse is transmitted along with the word pulse as soon as the first word of data from the SCM is entered in the output channel disassembly register.

This instruction is effective only if the monitor mode flag is set in the program status register. If the monitor mode flag is cleared, this instruction becomes a pass instruction. When this instruction issues, it will execute the required channel functions without regard to the current status or activity at the output channel.

The lowest order four bits of (Bk) are used in this instruction. The higher order bits are ignored. If higher order bits are set in (Bk), the lowest order four bits are masked out and used to determine the channel number. If (Bk) is zero, this instruction becomes a pass instruction.

Normally, the output channel buffer is inactive when this instruction is executed, the program having checked for completion of the previous record before issuing an RO. The program can detect the end of record in two ways. First, it can compare the output channel buffer address with a known record length. The alternative is to obtain a response from the peripheral unit over the corresponding input channel data path. If data is moving over the output channel data path when an RO is issued, the RO instruction takes priority, with a resulting loss of data in the previous record. Two or more consecutive RO instructions referring to different channels will issue in consecutive clock periods with no interference resulting in the multiplexer. If two consecutive instructions refer to the same channel, they transmit a record pulse over the output path and restart the buffer repeatedly. A data word may or may not be transmitted depending on the timing of the instructions and conflicts that occur.

**Format:**

Operation	Variable	Description	Size	Octal Code
RO	Bk	Reset output channel (Bk) buffer	15 bits	0170k

**Example:**

Code Generated

01705

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RO	B5	

**8.4.13 READ CHANNEL STATUS INSTRUCTIONS**

These instructions copy the contents of the input or output channel buffer address register indicated by masking the low order 4 bits of Bk and enter the value in Bj. The instructions are used for monitoring the progress of an input channel buffer or an output channel buffer.

A channel buffer area is divided into fields by the threshold testing mechanism. The first half of the buffer area constitutes one field and the last half of the buffer area the other field. An I/O multiplexer interrupt request is generated by the threshold testing mechanism whenever the channel buffer address is advanced across a field boundary. This occurs at the center of the buffer area and at the end of the buffer area.

The IBj instruction is the only vehicle for a program to determine whether an I/O multiplexer interrupt request was generated by a buffer threshold test or by a record flag. The program must retain the input channel buffer address from one interrupt period to the next. If the buffer address is in the same field as for the previous interrupt, the interrupt request was from a record flag. If the buffer address is in the opposite field from the previous interrupt, the interrupt request was from a threshold test.

The lowest order four bits of (Bk) are used in these instructions. The higher order bits are ignored. If higher order bits are set in (Bk) the lowest order four bits are masked out and used to determine the channel number. If (Bk) = 0, the IBj instruction reads the contents of the CPU clock period counter. However, the OBJ instruction places all zeros into Bj.

Two or more IBj instructions or OBJ instructions may occur in consecutive program instruction locations referencing the same or different channels. These instructions may issue in consecutive clock periods providing the Bj register reservations do not cause a delay. No interference will result in the multiplexer in these situations.

If correct results are to be obtained, an IBj instruction must not immediately follow an RI instruction nor may an OBJ instruction immediately follow an RO instruction. A delay of one clock period is sufficient.

Format:

Operation	Variable	Description	Size	Octal Code
IBj	Bk	Bj ← Read input channel (Bk) status	15 bits	016jk
OBJ	Bk	Bj ← Read output channel (Bk) status	15 bits	017jk

Example:

Code Generated

01664

01756

LOCATION	OPERATION	VARIABLE	COMMENTS
1	IB6	B4	
	OB5	B6	

#### 8.4.14 UNCONDITIONAL JUMP INSTRUCTION

This instruction adds the contents of index register Bi to K and branches to the relative CM (SCM) address specified by the sum. The remaining instructions, if any, in the current instruction word are not executed. The branch address is K when i is zero.

Addition is performed in an 18-bit one's complement mode. On the CYBER 180 Series, the CYBER 170 Series (except Model 176), the CYBER 70 Models 71, 72, 73, and 74, and 6000 Series systems, this instruction voids the stack. On the CYBER 70 Model 76, the 7600, and the CYBER 170 Model 176, the instruction word stack is not altered by execution of this instruction. The instruction is intended to allow computed branch point destinations. It is the only CPU instruction in which a computed parameter can specify a program branch destination address. All other jump instructions have preassigned destination addresses at execution time.

The assembler sets the unused j designator to the same value as the i designator. A force upper occurs after the instruction is assembled.

**Format:**

Operation	Variable	Description	Size	Octal Code
JP	Bi±K	Jump to (Bi)±K	30 bits	02iiK
JP	Bi	Jump to (Bi)	30 bits	02ii0 00000
JP	K	Jump to K	30 bits	0200K

**Example:**

Code Generated

0255002373 +  
0277000000

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	JP	B5+GOTO	
	JP	B7	

### 8.4.15 X-REGISTER CONDITIONAL BRANCH INSTRUCTIONS

These instructions cause the program sequence to branch to K or to continue with the current program sequence depending on the contents of operand register Xj. The decision is not made until the Xj register is free. These instructions do not void the stack.

The following rules apply to tests made in this instruction group:

The ZR and NZ operations test the full 60-bit word in Xj. The words 00 . . . . 00 and 77 . . . . 77<sub>8</sub> are treated as zero. All other words are non-zero. Thus, these instructions are not a valid test for floating point zero coefficients. However, they can be used for underflow of floating point quantities.

The PL and NG operations examine only the sign bit (bit 59) of Xj. If the sign bit is zero, the word is positive; if the sign bit is one, the word is negative. Thus, the sign test is valid for fixed point words or for coefficients in floating point words.

The IR and OR operations examine the upper-order 12 bits of Xj.

On the CYBER 170 Model 176, the CYBER 70 Model 76, and the 7600, the following octal quantities are detected as being out of range:

3777x . . . . x (positive overflow)  
4000x . . . . x (negative overflow)  
1777x . . . . x (positive indefinite)  
6000x . . . . x (negative indefinite)

All other words are in range. An underflow quantity is considered in range. The value of the coefficient is ignored in making this test.

On CYBER 70 Models 71, 72, 73, and 74; CYBER 180 Series; CYBER 170 Series (except Model 176); and 6000 Series computer systems, the octal quantities 3777x . . . x and 4000x . . . x are out of range; all other words are in range.

The DF and ID operations examine the upper-order 12 bits of Xj. Both positive and negative indefinite forms are detected:

1777x . . . . x and 6000x . . . . x are indefinite.

All other words are definite. The value of the coefficient is ignored in making this test.

An error exit occurs on 6000 Series; CYBER 180 Series; CYBER 170 Series; and CYBER 70 Models 71, 72, 73 and 74 systems when an indefinite or out of range value is used as an operand of an arithmetic instruction. Such error exits can be avoided by using DF, ID, IR, or OR instructions to test for such values before using them as operands.

On a 7600 or CYBER 70 Model 76 system, an error exit occurs as soon as an indefinite or out of range value is produced as the result of an arithmetic instruction. The DF, ID, IR and OR instructions are useful only when a MODE control statement is used to suppress such error exits.

**Format:**

Operation	Variable	Description	Size	Octal Code
ZR	Xj, K	Branch to K if (Xj) = 0	30 bits	030jK
NZ	Xj, K	Branch to K if (Xj) ≠ 0	30 bits	031jK
PL	Xj, K	Branch to K if (Xj) sign is plus	30 bits	032jK
NG	Xj, K	Branch to K if (Xj) sign is minus	30 bits	033jK
MI	Xj, K	Branch to K if (Xj) sign is minus	30 bits	033jK
IR	Xj, K	Branch to K if (Xj) in range	30 bits	034jK
OR	Xj, K	Branch to K if (Xj) out of range	30 bits	035jK
DF	Xj, K	Branch to K if (Xj) definite	30 bits	036jK
ID	Xj, K	Branch to K if (Xj) indefinite	30 bits	037jK

**Example:**

Code Generated

0305002363 +  
 0313002364 +  
 0324002365 +  
 0331002366 +  
 0331002366 +  
 0340002367 +  
 0351002370 +  
 0365002371 +  
 0377002372 +

LOCATION	OPERATION	VARIABLE	COMMENTS
11	11	18	30
	ZR	X5, ZERO	
	NZ	X3, NONZERO	
	PL	X4, PLUS	
	NG	X1, NEG	
	MI	X1, NEG	
	IR	X0, INRANGE	
	OR	X1, OUTRNGE	
	DF	X5, DEFINT	
	ID	X7, INDEFNT	

## 8.4.16 B-REGISTER CONDITIONAL BRANCH INSTRUCTIONS

The following rules apply in the tests made by these instructions:

Positive zero is recognized as unequal to negative zero.

Positive zero is recognized as greater than negative zero.

A positive number is recognized as greater than a negative number.

The 06 and 07 instructions are intended for branching on an index threshold test. The tests are made in a 19-bit one's complement mode. The (Bi) and the (Bj) are sign extended one bit to prevent erroneous results caused by exceeding the modulus of the comparison device. The (Bj) is then subtracted from the (Bi). The branch decision is based on the sign bit in the 19-bit result.

For these instructions, Bi and Bj must be specified in the order indicated below.

These instructions do not void the instruction stack.

Format:

Operation	Variable	Description	Size	Octal Code
ZR	K	Branch to K	30 bits	0400K
ZR	Bi, K	Branch to K if (Bi) = 0	30 bits	04i0K
EQ	K	Branch to K	30 bits	0400K
EQ	Bi, K	Branch to K if (Bi) = 0	30 bits	04i0K
EQ	Bi, Bj, K	Branch to K if (Bi) = (Bj)	30 bits	04ijK
NE	Bi, K	Branch to K if (Bi) ≠ 0	30 bits	05i0K
NE	Bi, Bj, K	Branch to K if (Bi) ≠ (Bj)	30 bits	05ijK
NZ	Bi, K	Branch to K if (Bi) ≠ 0	30 bits	05i0K
PL	Bi, K	Branch to K if (Bi) ≥ 0	30 bits	06i0K
GE	Bi, K	Branch to K if (Bi) ≥ 0	30 bits	06i0K
GE	Bi, Bj, K	Branch to K if (Bi) ≥ (Bj)	30 bits	06ijK
LE	Bj, Bi, K	Branch to K if (Bj) ≤ (Bi)	30 bits	06ijK
LE	Bj, K	Branch to K if (Bj) ≤ 0	30 bits	060jK
NG	Bi, K	Branch to K if (Bi) < 0	30 bits	07i0K
MI	Bi, K	Branch to K if (Bi) < 0	30 bits	07i0K
GT	Bj, Bi, K	Branch to K if (Bj) > (Bi)	30 bits	07ijK
GT	Bj, K	Branch to K if (Bj) > 0	30 bits	070jK
LT	Bi, K	Branch to K if (Bi) < 0	30 bits	07i0K
LT	Bi, Bj, K	Branch to K if (Bi) < (Bj)	30 bits	07ijK

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
0450005221 +		ZR	B5, BZERO	
0405005222 +		EQ	B0, B5, EQUAL	
0453005223 +		EQ	B5, B3, JUMP	
0400005223 +		EQ	JUMP	
0515005224 +		NE	B1, B5, NOTEQ	
0560005225 +		NZ	B6, BNOTZR	
0620005226 +		PL	B2, BPLUS	
0645005227 +		GE	B4, B5, GEQ	
0650005230 +		GE	B5, GEBO	
0676005231 +		LE	B6, B7, LTHAN	
0770005232 +		NG	B7, BNEG	
0730005233 +		MI	B3, B3LTO	
0767005234 +		GT	B7, B6, B7GT	
0705005235 +		GT	B5, B5GT0	
0712005236 +		LT	B1, B2, BLTB	

**8.4.17 TRANSMIT INSTRUCTION**

This instruction transfers the 60-bit word in operand register Xj to register Xi. It is essentially a copy instruction intended for moving data from X register to X register as quickly as possible. No logical function occurs. The assembler sets the k designator to the value specified for j.

**Format:**

Operation	Variable	Description	Size	Octal Code
BXi	Xj	Transmit (Xj) to Xi	15 bits	10ijj

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
10622		BX6	X2	

### 8.4.18 LOGICAL PRODUCT INSTRUCTION

This instruction forms the logical product (AND function) of 60-bit words from operand registers Xj and Xk and places the product in operand register Xi. Bits of register Xi are set to 1 when the corresponding bits of the Xj and Xk registers are 1 as in the following example:

(Xj) = 0101  
 (Xk) = 1100  
 (Xi) = 0100

This instruction is intended for extracting portions of a 60-bit word during data processing. If the j and k designators have the same value, the instruction becomes a transmit instruction.

Format:

Operation	Variable	Description	Size	Octal Code
BXi	Xj*Xk	Logical product of (Xj) and (Xk) to Xi	15 bits	11ijk

Example:

Code Generated

11553

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	BX5	X5*X3	

### 8.4.19 LOGICAL SUM INSTRUCTION

This instruction forms the logical sum (inclusive OR) of 60-bit words from operand registers Xj and Xk and places the sum in operand register Xi. A bit of register Xi is set to 1 if the corresponding bit of the Xj or Xk register is a 1, as in the following example:

(Xj) = 0101  
 (Xk) = 1100  
 (Xi) = 1101

This instruction is intended for merging portions of a 60-bit word into a composite word during data processing. If the j and k designators have the same value, the instruction degenerates into a transmit instruction.

Format:

Operation	Variable	Description	Size	Octal Code
BXi	Xj+Xk	Logical sum of (Xj) and (Xk) to Xi	15 bits	12ijk

Example:

Code Generated

12767

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	BX7	X6+X7	



### 8.4.20 LOGICAL DIFFERENCE INSTRUCTION

This instruction forms the logical difference (exclusive OR) of 60-bit words from operand registers Xj and Xk and places the difference in operand register Xi. A bit in register Xi is set to 1 if the corresponding bits in the Xj and Xk registers are unlike, as in the following example:

(Xj) = 0101  
 (Xk) = 1100  
 (Xi) = 1001

This instruction is intended for comparing bit patterns or for complementing bit patterns during data processing. If the j and k designators have the same value, the result will be a word of all zeros written into register Xi.

Format:

Operation	Variable	Description	Size	Octal Code
BXi	Xj-Xk	Logical difference of (Xj) and (Xk) to Xi	15 bits	13ijk

Example:

Code Generated

13601

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	BX6	X0-X1	

### 8.4.21 COMPLEMENT INSTRUCTION

This instruction extracts the 60-bit word from operand register Xk, complements it, and transmits this complemented quantity to operand register Xi. It is intended for changing the sign of a fixed point or floating point quantity as quickly as possible.

The assembler sets the unused j designator of the instruction to k.

Format:

Operation	Variable	Description	Size	Octal Code
BXi	-Xk	Transmit complement of (Xk) to Xi	15 bits	14ikk

Example:

Code Generated

14311

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	BX3	-X1	

## 8.4.22 LOGICAL PRODUCT AND COMPLEMENT INSTRUCTION

This instruction forms the logical product (AND function) of the 60-bit quantity from operand register X and the complement of the 60-bit quantity from operand register Xk, and places the result in operand register Xi. Thus, bits of Xi are set to 1 when the corresponding bits of the Xj register and the complement of the Xk register are 1, as in the following example:

(Xj) = 0101  
 Complemented (Xk) = 0011  
 (Xi) = 0001

This instruction is intended for extracting portions of a 60-bit word during data processing. If the j and k designators have the same value, a logical product is formed between two complementary quantities. The result will be a word of all zeros.

### Format:

Operation	Variable	Description	Size	Octal Code
BXi	-Xk*Xj	Logical product of (Xj) and complement of (Xk) to Xi	15 bits	15ijk

### Example:

#### Code Generated

15432

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		BX4	-X2*X3	

## 8.4.23 COMPLEMENT AND LOGICAL SUM INSTRUCTION

This instruction forms the logical sum (inclusive OR) of the 60-bit quantity from operand register Xj and the complement of the 60-bit word from operand register Xk, and places the result in operand register Xi. Thus, bits of Xi are set to 1 if the corresponding bit of the Xj register is one or the corresponding bit of the Xk register is a 0, as in the following example:

(Xj) = 0101  
 (Xk) = 1100  
 (Xi) = 0111

This instruction is intended for merging portions of a 60-bit word into a composite word during data processing. If the j and k designators have the same value, the result is a word of all ones.

### Format:

Operation	Variable	Description	Size	Octal Code
BXi	-Xk+Xj	Logical sum of (Xj) and complement of (Xk) to Xi	15 bits	16ijk

### Example:

#### Code Generated

16654

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		BX6	-X4+X5	

### 8.4.24 COMPLEMENT AND LOGICAL DIFFERENCE INSTRUCTION

This instruction forms the logical difference (exclusive OR) of the quantity from operand register Xj and the complement of the 60-bit word from operand register Xk, and places the result in operand register Xi. Thus, bits of Xi are set to 1 if the corresponding bits of Xj and register Xk are alike, as in the following example:

(Xj) = 0101  
 (Xk) = 1100  
 (Xi) = 0110

This instruction is intended for comparing bit patterns or for complementing bit patterns during data processing. If the j and k designators have the same value, a logical difference is formed between two complementary quantities. The result is a word of all ones.

Format:

Operation	Variable	Description	Size	Octal Code
BXi	-Xk-Xj	Logical difference of (Xj) and complement of (Xk) to Xi	15 bits	17ijk

Example:

Code Generated

17731

LOCATION	OPERATION	VARIABLE	COMMENTS
11		18	30
	BX7	-X1-X3	

### 8.4.25 LOGICAL LEFT SHIFT jk PLACES INSTRUCTION

This instruction shifts the 60-bit word in operand register Xi left circular jk places if expression jk is positive or left circular 60+jk places if jk is negative. Bits shifted off the left end of operand register Xi replace those shifted from the right end.

The 6-bit shift count jk allows a complete circular shift of (Xi).

In COMPASS notation, jk is an absolute expression. If it is positive, COMPASS places the lower 6 bits of the value in the jk fields. If it is negative, COMPASS adds 60 to jk and places the result in the jk fields. Thus, a negative value effectively designates a logical right shift. A positive value designates a left shift.

If the negative shift count is less than -60, the assembler generates a type 7 error.

Format:

Operation	Variable	Description	Size	Octal Code
LXi	jk	Logical shift (Xi) by $\pm$ jk places	15 bits	20ijk

**Example:**

Code Generated

20325

20362

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	LX3	25B	
	LX3	-12B	

### 8.4.26 ARITHMETIC RIGHT SHIFT jk PLACES INSTRUCTION

This instruction shifts the 60-bit word in operand register Xi right jk places if expression jk is positive and right 60+jk places if expression jk is negative. The rightmost bits of Xi are discarded and the sign bit is extended.

If the shift count is equal to the 60-bit register length, the result contains 60 copies of the sign bit. If the operand is positive, a positive zero results. If the operand is negative, a negative zero results.

In COMPASS notation, jk is an absolute expression. If it is positive, COMPASS places the lower 6 bits of the value in the jk fields. If it is negative, COMPASS adds 60 to jk and places the result in the jk fields. Thus, a negative value effectively designates the number of high order bits of the operand that are to be retained. If the negative shift count is less than -60, a type 7 error is generated.

**Format:**

Operation	Variable	Description	Size	Octal Code
AXi	jk	Arithmetic shift (Xi) by $\pm$ jk places	15 bits	21ijk

**Example:**

Code Generated

21537

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	AX5	37B	

### 8.4.27 LOGICAL LEFT SHIFT (Bj) PLACES INSTRUCTION

This instruction shifts the 60-bit quantity from operand register Xk the number of places specified by the quantity in index register Bj and places the result in operand register Xi. The direction of the shift operation is determined by the sign of Bj, as follows:

If (Bj) is positive (that is, bit 17 of Bj=0), the quantity from Xk is shifted left circular. The low order 6 bits of (Bj) specify the shift count. The higher order bits are ignored.

If (Bj) is negative (that is, bit 17 of Bj=1), the quantity from Xk is shifted right (end off with sign extension). For the CYBER 180 Series; the CYBER 170 Series (except Model 176); the CYBER 70 Series Models 71, 72, 73, and 74; and the 6000 Series, the one's complement of the low order 11 bits of (Bj) specify the shift count. The higher order bits are ignored. If the shift count is 59 to 63 (decimal), the result stored in the Xi register consists of 60 copies of the operand sign bit. If the shift count is 64 (decimal) or greater, the result register Xi is cleared to 60 zeros. For the CYBER 170 Model 176, CYBER 70 Model 76 and the 7600, the one's complement of the low order 12 bits of (Bj) specifies the shift count. The higher order bits are ignored. If the shift count is 59 (decimal) or greater, the result stored in the Xi register consists of 60 copies of the operand sign bit.

If -Bj is specified, the assembler converts the instruction to an arithmetic right shift. The (Bj) might be the result of an unpack instruction, in which case it is the unbiased exponent and (Xi) is the coefficient. This instruction is used for shifting a coefficient from a floating point number to the integer position after an unpack operation.

**Format:**

Operation	Variable	Description	Size	Octal Code
LXi	Xk, Bj	Logically shift (Xk) by (Bj) places to Xi	15 bits	22ijk
LXi	Bj, Xk	Logically shift (Xk) by (Bj) places to Xi	15 bits	22ijk
LXi	Xk	Transmit (Xk) to Xi	15 bits	22i0k
LXi	Bj	Logically shift (Xi) by (Bj) places to Xi	15 bits	22iji
LXi	-Bj, Xk	Arithmetic right shift (Xk) by (Bj) places to Xi	15 bits	23ijk
LXi	Xk, -Bj	Arithmetic right shift (Xk) by (Bj) places to Xi	15 bits	23ijk
LXi	-Bj	Arithmetic right shift (Xi) by (Bj) places to Xi	15 bits	23iji

**Example:**

Code Generated

22675  
22534  
22302

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	LX6	X5, B7	
	LX5	B3, X4	
	LX3	X2	

**8.4.28 ARITHMETIC RIGHT SHIFT (Bj) PLACES INSTRUCTION**

This instruction shifts the 60-bit quantity from operand register Xk the number of places specified by the quantity in index register Bj and places the result in operand register Xi. The direction of the shift operation is determined by the sign of Bj, as follows:

If (Bj) is positive (that is, bit 17 of Bj=0), the quantity from register Xk is shifted right (end off with sign extension). For the CYBER 180 Series; the CYBER 170 Series (except Model 176); the CYBER 70 Models 71, 72, 73, and 74; and the 6000 Series computer systems, the low order 11 bits of (Bj) specify the shift count. The higher order bits are ignored. If the shift count is 59 to 63 (decimal), the Xi register contains 60 copies of the (Xk) sign bit. If the shift count is 64 (decimal) or more, the Xi register is zeroed. For the CYBER 170 Model 176, CYBER 70 Model 76, or 7600 computer systems, the low order 12 bits of (Bj) specify the shift count. The higher order bits are ignored. If the shift count is 59 (decimal) or more, the Xi register contains 60 copies of the sign of the operand.

If (Bj) is negative (that is, bit 17 of Bj=1), the quantity from register Xk is shifted left circular. The complement of the lower order 6 bits of Bj specify the shift count. The higher order bits are ignored.

If -B is specified, the assembler converts the instruction to a logical left shift. This instruction is intended for use in data processing where the amount of shift is derived in the computation. This instruction is also useful for adjusting the coefficient of a floating point number while it is in its unpacked form.

Format:

Operation	Variable	Description	Size	Octal Code
AXi	Xk, Bj	Arithmetic shift of (Xk) by (Bj) places to Xi	15 bits	23ijk
AXi	Bj, Xk	Arithmetic shift of (Xk) by (Bj) places to Xi	15 bits	23ijk
AXi	Xk	Transmit (Xk) to Xi	15 bits	23i0k
AXi	Bj	Arithmetic shift of (Xi) by (Bj) places to Xi	15 bits	23iji
AXi	-Bj, Xk	Logically shift (Xk) by (Bj) places to Xi	15 bits	22ijk
AXi	Xk, -Bj	Logically shift (Xk) by (Bj) places to Xi	15 bits	22ijk
AXi	-Bj	Logically shift (Xi) by (Bj) places to Xi	15 bits	22iji

Example:

Code Generated

23764

23211

23502

23424

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	AX7	X4, B6	
	AX2	B1, X1	
	AX5	X2	
	AX4	B2	

#### 8.4.29 NORMALIZE INSTRUCTION

This instruction normalizes the floating point quantity from operand register Xk and places it in operand register Xi. Normalizing consists of shifting the coefficient the minimum number of positions required to make bit 47 different from bit 59. This places the most significant bit of the coefficient in the highest order position of the coefficient portion of the word. The exponent portion of the word is then decreased by the number of bit positions shifted. The number of shifts required to normalize the quantity is entered in index register Bj.

Format:

Operation	Variable	Description	Size	Octal Code
NXi	Xk	Normalize (Xk) to Xi	15 bits	24i0k
NXi	Bj, Xk	Normalize (Xk) to Xi; shift count to Bj	15 bits	24ijk
NXi	Xk, Bj	Normalize (Xk) to Xi; shift count to Bj	15 bits	24ijk
NXi		Normalize (Xi) to Xi	15 bits	24i0i
NXi	Bj	Normalize (Xi) to Xi; shift count to Bj	15 bits	24iji

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
24575		NX5	X5, B7	
24505		NX5	X5	
24552		NX5, B5	X2	

**8.4.30 ROUND AND NORMALIZE INSTRUCTION**

This instruction performs the same operation as the NXi instruction with the exception that the quantity from operand register Xk is rounded before it is normalized. Rounding is accomplished by placing a 1 round bit immediately to the right of the least significant coefficient bit. The resulting coefficient is increased by one-half the value of the least significant bit. Normalizing a zero coefficient places the round bit in bit 47 and reduces the exponent by 48. Note that the same rules apply for underflow, overflow, infinite, and indefinite results.

If (Xk) is an infinite quantity (3777x...xg or 4000x...xg) or an indefinite quantity (1777x...xg or 6000x...xg), no shift takes place. The contents of Xk are copied into Xi, and Bj is set to zero.

**Format:**

Operation	Variable	Description	Size	Octal Code
ZXi	Xk	Round and normalize (Xk) to Xi	15 bits	25i0k
ZXi	Bj, Xk	Round and normalize (Xk) to Xi; shift count to Bj	15 bits	25ijk
ZXi	Xk, Bj	Round and normalize (Xk) to Xi; shift count to Bj	15 bits	25ijk
ZXi	Bj	Round and normalize (Xi) to Xi; shift count to Bj	15 bits	25iji
ZXi		Round and normalize (Xi) to Xi	15 bits	25i0i

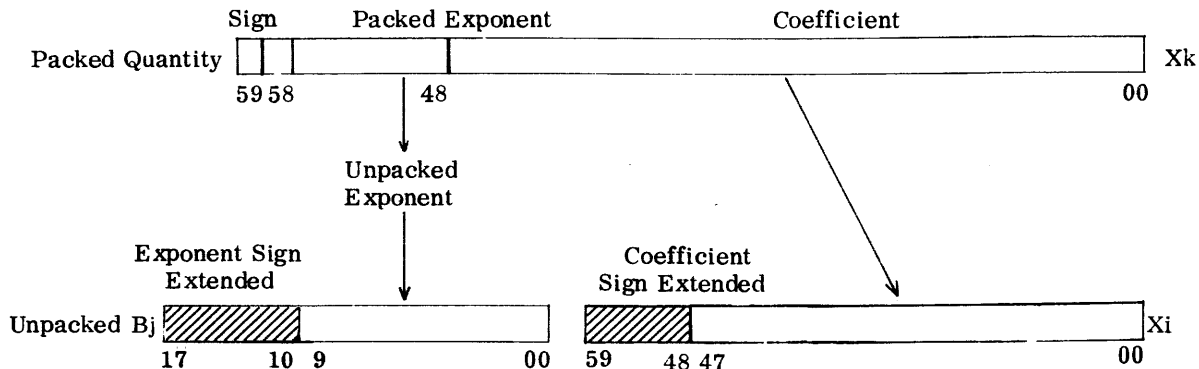
**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
25474		ZX4	X4, B7	
25404		ZX4		
25361		ZX3, B6	X1	

**8.4.31 UNPACK INSTRUCTION**

This instruction unpacks the floating point quantity from operand register Xk and sends the 48-bit coefficient to operand register Xi and the 11-bit exponent to index register Bj. The exponent packing is removed during unpack so that the quantity in Bj is the true one's complement representation of the exponent. The contents of Xk need not be normalized.

The exponent and coefficient are sent to the low-order bits of the respective registers, as shown below:



Special operand formats are treated in the same manner as normal operands.

**Format:**

Operation	Variable	Description	Size	Octal Code
UXi	Xk	Unpack (Xk) to Xi	15 bits	26i0k
UXi	Bj, Xk	Unpack (Xk) to Xi and Bj	15 bits	26ijk
UXi	Xk, Bj	Unpack (Xk) to Xi and Bj	15 bits	26ijk
UXi		Unpack (Xi) to Xi	15 bits	26i0i
UXi	Bj	Unpack (Xi) to Xi and Bj	15 bits	26iji

**Example:**

Code Generated

26777  
26342  
26707  
26777

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	UX7	X7, B7	
	UX3, X2	B4	
	UX7		
	UX7	B7	

### 8.4.32 PACK INSTRUCTION

This instruction packs a floating point number in operand register xi. The coefficient of the number is obtained from operand register Xk and the exponent is obtained from index register Bj. The exponent is packed by reversing the setting of bit 10 of the exponent during the pack operation. The pack instruction does not normalize the coefficient.

Exponent and coefficient are obtained from the proper low-order bits of the respective registers and packed in reverse order as shown in the illustration for the unpack instruction. Thus, bits 58 through 48 of Xk and bits 17 through 11 of Bj are ignored. There is no test for overflow or underflow. No flags are set in the PSD register by this instruction.



Note that if (Xk) is positive, the packed exponent occupying bits 58 through 48 of Xi is obtained from bits 10 through 00 of Bj by complementing bit 10; if (Xk) is negative, bit 10 is not complemented but bits 09 through 00 are complemented.

The j designator can be set to zero in this instruction to pack a fixed point integer into floating point format without using one of the active B registers (exponent=0).

**Format:**

Operation	Variable	Description	Size	Octal Code
PXi	Xk	Pack (Xk) to Xi	15 bits	27i0k
PXi	Xk, Bj	Pack (Xk) and (Bj) to Xi	15 bits	27ijk
PXi	Bj, Xk	Pack (Xk) and (Bj) to Xi	15 bits	27ijk
PXi		Pack (Xi) to Xi	15 bits	27i0i
PXi	Bj	Pack (Xi) and (Bj) to Xi	15 bits	27iji

**Example:**

Code Generated

27565  
27671  
27505  
27565

LOCATION	OPERATION	VARIABLE	COMMENTS
1		18	30
	PX5	X5, B6	
	PX6, B7	X1	
	PX5		
	PX5	B6	

### 8.4.33 UNROUNDED SP FLOATING POINT ADD INSTRUCTIONS

These instructions form the unrounded sum or difference of the floating point quantities from operand registers Xj and Xk and pack the result in operand register Xi. The packed result is the upper half of a double precision sum or difference.

At the start both arguments are unpacked, and the coefficient of the argument with the smaller exponent is entered into the upper half of the accumulator. The coefficient is shifted right by the difference of the exponents. The other coefficient is then added to or subtracted from the upper half of the accumulator. If overflow occurs, the result is right-shifted one place and the exponent of the result increased by one. The upper half of the accumulator holds the coefficient of the result, which is not necessarily in normalized form. The exponent and upper coefficient are then repacked in operand register Xi.

**Format:**

Operation	Variable	Description	Size	Octal Code
FXi	Xj+Xk	Floating point sum of (Xj) and (Xk) to Xi	15 bits	30ijk
FXi	Xj-Xk	Floating point difference of (Xj) minus (Xk) to Xi	15 bits	31ijk

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
30345		FX3	X4+X5	
31213		FX2	X1-X3	

### 8.4.34 DP FLOATING POINT ADD INSTRUCTIONS

These instructions form the sum or difference of two floating point numbers as in the single precision instructions, but pack the lower half of the double precision result with an exponent 48 less than the upper sum. The result is not necessarily normalized.

**Format:**

Operation	Variable	Description	Size	Octal Code
DXi	Xj+Xk	Floating DP sum of (Xj) and (Xk) to Xi	15 bits	32ijk
DXi	Xj-Xk	Floating DP difference of (Xj) and (Xk) to Xi	15 bits	33ijk

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
32323		DX3	X2+X3	
33414		DX4	X1-X4	

### 8.4.35 ROUNDED SP FLOATING POINT ADD INSTRUCTIONS

These instructions form the rounded sum or difference of the floating point quantities from operand registers Xj and Xk and pack the upper portion of the double precision result in operand register Xi. These instructions are intended for use in floating point calculations involving single precision accuracy.

**Format:**

Operation	Variable	Description	Size	Octal Code
RXi	Xj+Xk	Rounded floating sum of (Xj) and (Xk) to Xi	15 bits	34ijk
RXi	Xj-Xk	Rounded floating difference of (Xj) minus (Xk) to Xi	15 bits	35ijk

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1		18	30
34534		RX5	X3+X4	
35653		RX6	X5-X3	

### 8.4.36 LONG ADD (FIXED POINT) INSTRUCTIONS

These instructions form the 60-bit one's complement integer sum or integer difference of quantities from operand registers Xj and Xk and store the result in operand register Xi. An overflow condition is ignored.

The instructions are intended for addition or subtraction of integers too large for handling in the increment unit. They are also useful for merging and comparing data fields during data processing.

For an addition, if both operands are zero, the result is zero. If either zero operand is positive zero (all zeros), the result is a positive zero quantity. If both operands are minus zero (all ones), the result is a negative zero quantity.

**Format:**

Operation	Variable	Description	Size	Octal Code
IXi	Xj+Xk	Integer sum of (Xj) and (Xk) to Xi	15 bits	36ijk
IXi	Xj-Xk	Integer difference of (Xj) minus (Xk) to Xi	15 bits	37ijk

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1		18	30
36545		IX5	X4+X5	
37631		IX6	X3-X1	

### 8.4.37 UNROUNDED SP FLOATING POINT MULTIPLY INSTRUCTION

This instruction multiplies two floating point quantities obtained from operand registers Xj (multiplier) and Xk (multiplicand) and packs the upper product result in operand register Xi.

In this operation, the exponents of the two operands are unpacked from the floating point format and are added with a correction factor of 48 to form the exponent for the result. The coefficients are multiplied as signed integers to form a 96-bit integer product. The upper half of this product is then extracted to form the coefficient of the result. The result is a normalized quantity only when both operands are normalized; the exponent in this case is the sum of the exponents plus 47 (or 48). The result is not normalized when either or both operands are not normalized.

**Format:**

Operation	Variable	Description	Size	Octal Code
FXi	Xj*Xk	Floating point product of (Xj) and (Xk) to Xi	15 bits	40ijk

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
40011	1	11	18	30
		FX0	X1*X1	

**8.4.38 ROUNDED SP FLOATING POINT MULTIPLY INSTRUCTION**

This instruction multiplies the floating point number from operand register Xk (multiplicand), by the floating point number from operand register Xj. The upper product result is packed in operand register Xi. (No lower product is available.) The multiply operation is identical to that of the single precision instruction except that a rounding bit is added in bit position 46 of the 96-bit product. The upper half of the product is then extracted to form the coefficient for the result. An alternate output path is provided with a left shift of one bit position to normalize the result coefficient if the original operands were normalized and the double precision product has only 95 bits of significance. The exponent for the result is decremented by one count in this case.

**Format:**

Operation	Variable	Description	Size	Octal Code
RXi	Xj*Xk	Rounded floating point product of (Xj) and (Xk) to Xi	15 bits	41ijk

**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
41232	1	11	18	30
		RX2	X3*X2	

**8.4.39 DP FLOATING POINT MULTIPLY INSTRUCTION**

This instruction multiplies two floating point quantities obtained from operand registers Xj and Xk and packs the lower product in operand register Xi. The two 48-bit coefficients are multiplied together to form a 96-bit product. The lower order 48 bits of the product (bits 47 through 0) are then packed together with the resulting exponent. The result is not necessarily normalized. The exponent of this result is 48 less than the exponent resulting from an unrounded single precision instruction using the same operands.

This instruction is intended for use in multiple precision floating point calculations. It may also be used to form the product of two integers providing the resulting product does not exceed 48 bits of significance. The operands must be packed in floating point format before executing this instruction. The results must be unpacked to obtain the integer product.

Format:

Operation	Variable	Description	Size	Octal Code
DXi	Xj*Xk	Floating point DP product of (Xj) and (Xk) to Xi	15 bits	42ijk

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
42345	1	11	18	30
		DX3	X4*X5	

#### 8.4.40 INTEGER MULTIPLY INSTRUCTION

The CPU integer multiply instruction is, to COMPASS, synonymous with the double precision floating point multiply instruction. Regardless of how it is written in COMPASS, the 42ijk instruction is executed as follows: If each operand register has all zeros or all ones in its leftmost 12 bits, the 47-bit integer product is formed in Xi with sign extension in its leftmost 12 bits. (Exception: if each operand has bit 47 different from its sign bit, the result is shifted left one bit position.) Otherwise, a double precision floating point multiplication is performed. Thus, there is no need to pack exponents into the operands, and unpack the result, for an integer multiply. COMPASS provides the alternate symbolic representations IXi Xj\*Xk and DXi Xj\*Xk for the 42ijk instruction as an aid to program readability, so the programmer can indicate whether or not the instruction is being used for integer multiplication.

Format:

Operation	Variable	Description	Size	Octal Code
IXi	Xj*Xk	Integer product of (Xj) and (Xk) to Xi	15 bits	42ijk

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
42234	1	11	18	30
		IX2	X3*X4	

### 8.4.41 MASK INSTRUCTION

This instruction clears register Xi and forms a mask in it. A positive value for expression jk defines the number of ones in the mask as counted from the highest order bit in Xi. A negative value for expression jk defines the number of 0 bits (unmasked) counted from the low order bit in Xi. The completed masking word consists of ones in the high order bit positions of the word and zeros in the remainder of the word.

The contents of operand register i are zero when jk is zero. The contents of operand register i are all ones when jk is 60.

This instruction is intended for generating masks for logical operations. Used with the shift instruction, this instruction creates an arbitrary field mask faster than by reading a previously generated mask from storage.

In COMPASS notation, if the value of absolute expression jk is positive, the assembler inserts it into the jk field of the assembled instruction. If the value of absolute expression jk is negative, the assembler adds 60 to the expression value and places the sum in the jk field of the assembled instruction.

A negative jk value less than -60 results in a type 7 assembly error.

**Format:**

Operation	Variable	Description	Size	Octal Code
MXi	jk	Form mask in Xi, $\pm$ jk bits	15 bits	43ijk

**Example:**

Code Generated

43042

43360

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	MX0	42B	
	MX3	-14B	

### 8.4.42 UNROUNDED SP FLOATING POINT DIVIDE INSTRUCTION

This instruction divides two normalized floating point quantities obtained from operand registers Xj (dividend) and Xk (divisor) and packs the quotient in operand register Xi.

**Format:**

Operation	Variable	Description	Size	Octal Code
FXi	Xj/Xk	Floating point divide of (Xj) by (Xk) to Xi	15 bits	44ijk

**Example:**

Code Generated

44631

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	FX6	X7/X1	

### 8.4.43 ROUNDED SP FLOATING POINT DIVIDE INSTRUCTION

This instruction divides the floating quantity from operand register Xj (dividend) by the floating point quantity from operand register Xk (divisor) and packs the rounded quotient in operand register Xi.

Format:

Operation	Variable	Description	Size	Octal Code
RXi	Xj/Xk	Rounded floating point division of (Xj) by (Xk) to Xi	15 bits	45ijk

Example:

Code Generated

45724

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	RX7	X2/X4	

### 8.4.44 PASS INSTRUCTION

The no-operation (pass) instruction is not associated with a functional unit. This instruction is a do-nothing instruction used typically to pad the program between steps. An integer value in the variable field (optional) is inserted into the lower 8 bits of the instruction. The assembler automatically pads the remainder of a word whenever a force upper occurs; in this case, the programmer is not required to insert the NO.

On a machine with a Compare/Move Unit (CMU), a value of n greater than or equal to 400g causes the instruction to be interpreted as a CMU instruction.

On CYBER 170 Models 175, 740, 750, and 760, a value of n greater than or equal to 400g is illegal.

Format:

Operation	Variable	Description	Size	Octal Code
NO		Pass	15 bits	46000
NO	n	Pass	15 bits	46n

Example:

Code Generated

46000

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	NO		

### 8.4.45 POPULATION COUNT INSTRUCTION

This instruction counts the number of 1 bits in operand register Xk and stores the count in the lower order 6 bits of operand register Xi. Bits 59 through 06 are cleared.

If Xk is a word of all ones, a count of 60 (decimal) is delivered to the Xi register. If Xk is a word of all zeros, a zero word is delivered to the Xi register.

The assembler sets the unused j designator to k.

**Format:**

Operation	Variable	Description	Size	Octal Code
CXi	Xk	Count of number of 1's in (Xk) to Xi	15 bits	47ikk

**Example:**

Code Generated

47700

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		CX7	X0	

#### 8.4.46 SET A REGISTER INSTRUCTIONS

These instructions are intended for fetching operands from storage for computation and for delivering results back into storage. The instructions have two destination registers: the Ai register, which receives the address formed from the operands, and either the Xi register or a CM (SCM) storage location.

Operands are obtained from address (A), index (B), and operand (X) registers as well as from the instruction itself (K = 18-bit operand). Operands obtained from an Xj operand register are the truncated lower 18 bits of the 60-bit word. The highest order bits are ignored; an overflow condition is also ignored.

If the i designator is nonzero, a storage reference is made using the lower 15, 16, or 17 bits of the resulting sum or difference as the relative storage address depending on machine size. The upper bits are ignored. The type of storage reference is a function of the i designator value, as follows:

- i = 0; no storage reference
- i = 1, 2, 3, 4, or 5; contents of CM (SCM) relative address (Ai) to register Xi
- i = 6 or 7; contents of register Xi stored at CM (SCM) relative address (Ai)



**Format:**

Operation	Variable	Description	Size	Octal Code
SAi	Aj+K	Set Ai to (Aj) + K	30 bits	50ijK
SAi	K	Set Ai to K	30 bits	51i0K
SAi	Bj+K	Set Ai to (Bj) + K	30 bits	51ijK
SAi	Xj+K	Set Ai to (Xj) + K	30 bits	52ijK
SAi	Xj	Set Ai to (Xj)	15 bits	53ij0
SAi	Xj+Bk	Set Ai to (Xj) + (Bk)	15 bits	53ijk
SAi	Bk+Xj	Set Ai to (Xj) + (Bk)	15 bits	53ijk
SAi	Aj	Set Ai to (Aj)	15 bits	54ij0
SAi	Aj+Bk	Set Ai to (Aj) + (Bk)	15 bits	54ijk
SAi	Bk+Aj	Set Ai to (Aj) + (Bk)	15 bits	54ijk
SAi	Aj-Bk	Set Ai to (Aj) - (Bk)	15 bits	55ijk
SAi	-Bk+Aj	Set Ai to (Aj) - (Bk)	15 bits	55ijk
SAi	Bj	Set Ai to (Bj)	15 bits	56ij0
SAi	Bj+Bk	Set Ai to (Bj) + (Bk)	15 bits	56ijk
SAi	-Bk	Set Ai to (B0) - (Bk)	15 bits	57i0k
SAi	Bj-Bk	Set Ai to (Bj) - (Bk)	15 bits	57ijk
SAi	-Bk+Bj	Set Ai to (Bj) - (Bk)	15 bits	57ijk

**Example:**

Code Generated

5010000001  
 5100777774  
 5121000003  
 5231777771  
 53411  
 54541  
 54641  
 54540  
 55641  
 56711  
 57721

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SA1	A0+1	
	SA0	-3	
	SA2	3+B1	
	SA3	X1-6	
	SA4	X1+B1	
	SA5	A4+B1	
	SA6	A4+B1	
	SA5	A4	
	SA6	-B1+A4	
	SA7	B1+B1	
	SA7	B2-B1	

### 8.4.47 DIRECT READ/WRITE CENTRAL MEMORY

These instructions permit information to be stored into central memory from the specified X register or to be loaded from central memory into the X register. The lower 21 bits of Xk specify the central memory address relative to RA<sub>C</sub>. The other bits of Xk are unused.

Format:

Operation	Variable	Description	Size	Octal Code
CR	Xj,Xk	Read CM at (Xk) to Xj	15 bits	660jk
CW	Xj,Xk	Write Xj to CM at (Xk)	15 bits	670jk

Example:

Code Generated

66012

67012

	LOCATION	OPERATION	VARIABLE	COMMENTS
i		ii	18	30
		CR	X1 , X2	
		CW	X1 , X2	

### 8.4.48 SET B REGISTER INSTRUCTIONS

These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in index register Bi. Note the result will never be negative zero (all ones) unless negative zero is added to negative zero.

Operands are obtained from address (A), index (B), and operand (X) registers as well as from the instruction itself (K = 18-bit operand). Operands obtained from an Xj operand register are the truncated lower 18 bits of the 60-bit word. The highest order bits are ignored; an overflow condition is also ignored.

If the i designator is a zero, the instruction is a do-nothing instruction, except on the models 810, 815, 825, 830, 835, 845, 855, 865, and 875 for which two forms of the SB0 instruction (SB0 B<sub>j</sub>+B<sub>k</sub> and SB0 B<sub>j</sub>-B<sub>k</sub>) are invalid. On models 810, 815, 825, 830, 835, 845, 855, 865, and 875 the octal operation codes 660 and 670 are interpreted as the CR and CW instructions, respectively.

**Format:**

Operation	Variable	Description	Size	Octal Code
SBi	$A_j+K$	Set Bi to $(A_j) \pm K$	30 bits	60ijK
SBi	K	Set Bi to K	30 bits	61i0K
SBi	$B_j+K$	Set Bi to $(B_j) \pm K$	30 bits	61ijK
SBi	$X_j+K$	Set Bi to $(X_j) \pm K$	30 bits	62ijK
SBi	$X_j$	Set Bi to $(X_j)$	15 bits	63ij0
SBi	$X_j+B_k$	Set Bi to $(X_j) + (B_k)$	15 bits	63ijk
SBi	$B_k+X_j$	Set Bi to $(X_j) + (B_k)$	15 bits	63ijk
SBi	$A_j$	Set Bi to $(A_j)$	15 bits	64ij0
SBi	$A_j+B_k$	Set Bi to $(A_j) + (B_k)$	15 bits	64ijk
SBi	$B_k+A_j$	Set Bi to $(A_j) + (B_k)$	15 bits	64ijk
SBi	$A_j-B_k$	Set Bi to $(A_j) - (B_k)$	15 bits	65ijk
SBi	$-B_k+A_j$	Set Bi to $(A_j) - (B_k)$	15 bits	65ijk
SBi	$B_j$	Set Bi to $(B_j)$	15 bits	66ij0
SBi	$B_j+B_k$	Set Bi to $(B_j) + (B_k)$	15 bits	66ijk
SBi	$-B_k$	Set Bi to $(B_0) - (B_k)$	15 bits	67i0k
SBi	$B_j-B_k$	Set Bi to $(B_j) - (B_k)$	15 bits	67ijk
SBi	$-B_k+B_j$	Set Bi to $(B_j) - (B_k)$	15 bits	67ijk

**Example:**

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
6011777772		SB1	A1-5	
6110777772		SB1	-5	
6121000011		SB2	3+B1+6	
6231000100		SB3	X1+100B	
63427		SB4	X2+B7	
64541		SB5	A4+B1	
64540		SB5	A4	
65641		SB6	-B1+A4	
65643		SB6	A4-B3	
66711		SB7	B1+B1	
67751		SB7	B5-B1	

**8.4.49 SET X REGISTER INSTRUCTIONS**

The SXi instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result into the lower 18 bits of operand register Xi. The sign of the result is extended to the upper 42 bits of operand register Xi. An overflow condition is ignored.

Operands are obtained from address (A), index (B), and operand (X) registers as well as the instruction itself (K = 18-bit operand). Operands obtained from an Xj register are the truncated lower 18 bits of the 60-bit word. The highest order bits are ignored.

Format:

Operation	Variable	Description	Size	Octal Code
SXi	Aj-K	Set Xi to (Aj) $\pm$ K	30 bits	70ijk
SXi	K	Set Xi to K	30 bits	71i0K
SXi	Bj+K	Set Xi to (Bj) $\pm$ K	30 bits	71ijk
SXi	Xj+K	Set Xi to (Xj) $\pm$ K	30 bits	72ijk
SXi	Xj	Set Xi to (Xj)	15 bits	73ij0
SXi	Xj+Bk	Set Xi to (Xj) + (Bk)	15 bits	73ijk
SXi	Bk-Xj	Set Xi to (Xj) + (Bk)	15 bits	73ijk
SXi	Aj	Set Xi to (Aj)	15 bits	74ij0
SXi	Aj+Bk	Set Xi to (Aj) + (Bk)	15 bits	74ijk
SXi	Bk-Aj	Set Xi to (Aj) + (Bk)	15 bits	74ijk
SXi	Aj-Bk	Set Xi to (Aj) - (Bk)	15 bits	75ijk
SXi	-Bk-Aj	Set Xi to (Aj) - (Bk)	15 bits	75ijk
SXi	Bj	Set Xi to (Bj)	15 bits	76ij0
SXi	Bj-Bk	Set Xi to (Bj) - (Bk)	15 bits	76ijk
SXi	-Bk	Set Xi to (B0) - (Bk)	15 bits	77i0k
SXi	Bj-Bk	Set Xi to (Bj) - (Bk)	15 bits	77ijk
SXi	-Bk-Bj	Set Xi to (Bj) - (Bk)	15 bits	77ijk

Example:

Code Generated

7000005233 +  
 7110775755  
 7121000005  
 7233777744  
 73442  
 74553  
 74540  
 75604  
 75641  
 76776  
 77751

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SX0	BNEG+A0+1	
	SX1	-2022B	
	SX2	B1+5	
	SX3	X3-33B	
	SX4	X4+B2	
	SX5	A5+B3	
	SX5	A4	
	SX6	A0-B4	
	SX6	-B1+A4	
	SX7	B7+B6	
	SX7	B5-B1	

## 8.5 CMU SYMBOLIC MACHINE INSTRUCTIONS

The Compare/Move Unit (CMU) is a standard CPU hardware component of the CYBER 70 Models 72 and 73, and the CYBER 170 Models 172, 173, 174, 720, and 730. The models 810, 815, 825, 830, 835, 845, and 855 support compare/move instructions through simulation. These central processor instructions are used for moving and comparing data fields that consist of strings of 6-bit characters. Data fields can span word boundaries and can begin and end at any character position within a word. A data field is specified by its length in characters and the location of its leftmost character (according to word address and character position). Data fields cannot be in the operating registers nor in ECS.

Each 60-bit word of a data field contains 10 character positions numbered 0 to 9 from left to right (high order to low order).

COMPASS provides symbolic forms of the four CMU instructions plus a pseudo instruction used to generate a descriptor word to be referenced by the indirect move instruction. Of the four instructions, the indirect move (IM) instruction is the only one that syntactically resembles other CPU instructions. The other three instructions have formats dissimilar to CPU instructions and are generated through COMPASS pseudo instructions. All of these instructions must begin at the top of a 60-bit word; COMPASS automatically forces upper before each of them unless the location field contains a minus sign. All but IM are 60 bits in length. IM is 30 bits, but the hardware requires that the instruction be in the upper half of its word. The lower half of the word is not executed. COMPASS automatically forces upper following IM, unless the next instruction has a minus sign in its location field.

### 8.5.1 IM - INDIRECT MOVE

The indirect move instruction moves the contents of a data field to another location. It is a 30-bit instruction that specifies the address of a descriptor word which, in turn, contains the length and address of the data fields.

The assembler forces upper before and after the IM instruction.

The descriptor word is fetched from storage location (Bj)+K. If the data field length is zero, the instruction is executed as a pass but the execution time is longer. Otherwise, the contents of the source field are moved to the destination field. If the two fields overlap, the results are undefined. The X0 register is used for intermediate storage during execution of the instruction, and is cleared upon completion of the instruction.

Operation	Variable	Description	Octal Code
IM	K	Move data according to word at K	4640K
IM	Bj±K	Move data according to word at (Bj)±K	464jK
IM	Bj	Move data according to word at (Bj)	464j 000000

### 8.5.2 MD - INDIRECT MOVE DESCRIPTOR WORD

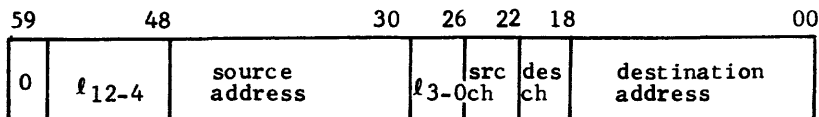
The MD pseudo instruction generates a descriptor word for use by the indirect move (IM) instruction.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	MD	$l, k_s, c_s, k_d, c_d$

- sym If present, sym is assigned the value of the location counter after the force upper occurs. It becomes the symbolic address of the descriptor word.
- $l$  Absolute address expression specifying the field length in characters (0 through 8191). The upper 9 bits ( $l$ ) are placed in bits 56 through 48 of the descriptor word; the lower 4 bits ( $l$ ) are placed in bits 29 through 26.
- $k_s$  An expression specifying the first word address of the source field in CM.
- $c_s$  An absolute expression (0 through 9) specifying the starting character position of the source field within the word at location  $k_s$ . Characters are numbered from left to right.
- $k_d$  An expression specifying the first word address of the destination field in CM.
- $c_d$  An absolute expression (0 through 9) specifying the starting character position of the destination field within the word at location  $k_d$ .

**Indirect Move Descriptor Word format:**



**Example:**

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
00760050004005007000	1	11	18	30
4640010665	1	11	18	30
	WORD	M	1001, BUFFA, 0, BUFFB, 5	
		.		
		.		
		.		
	I	1	D	D

BUFFA is at address 2560; BUFFB is at address 3584.

**8.5.3 DM - DIRECT MOVE**

The direct move (DM) symbolic instruction generates a CMU instruction that moves the contents of a data field to another data field. The machine instruction occupies one full word. The instruction includes its own data field descriptor.

The assembler forces upper before a DM instruction.

If the data field length is zero, the instruction is executed as a pass, but the execution time is longer. Otherwise, the contents of the source field are moved to the destination field. If the two fields overlap, the results are undefined. The X0 register is used for intermediate storage during execution of the instruction and is cleared upon completion of the instruction.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	DM	$l, k_s, c_s, k_d, c_d$

**sym** If present, sym is assigned the value of the location counter after the force upper occurs. It becomes the symbolic address of the instruction word.

**l** Absolute address expression specifying the field length in characters (0 through 127).

**k<sub>s</sub>** An expression specifying the first word address of the source field in CM.

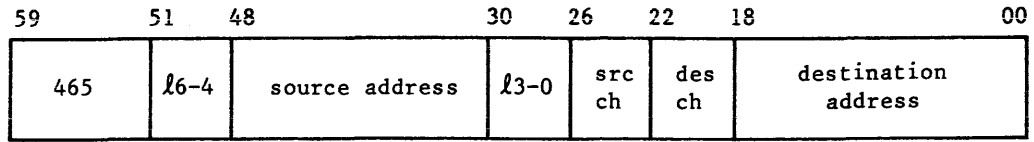
**c<sub>s</sub>** An absolute expression (0 through 9) specifying the starting character position of the source field within the word at location k<sub>s</sub>.

**k<sub>d</sub>** An expression specifying the first word address of the destination field in CM.

**c<sub>d</sub>** An absolute expression (0 through 9) specifying the starting character position of the destination field within the word at location k<sub>d</sub>. Characters are numbered from left to right.



Octal format of instruction:



Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
46570050007405007000	1	11	18	30
		DM	127,BUFFA,0,BUFFB,5	

### 8.5.4 CC - COMPARE COLLATED

The compare collated (CC) symbolic instruction generates a CMU instruction that compares the contents of two data fields, one character at a time, from left to right, until a pair of corresponding characters is found to have unequal collating values or until the data fields are exhausted. It is a 60-bit instruction that occupies one full word. It cannot be split between two words. The instruction includes its own data field descriptor. Register A0 contains the first word address of a table in storage that contains the collating values to be used in comparing characters. The result of the comparison is placed in register X0.

The first word address of the collating table is obtained from register A0. The contents of the data fields are compared from left to right, one character at a time from each field, until two unequal characters are found. The collating value of each character is obtained from the collating table. If these values are equal, the compare continues until another character pair is unequal or until all characters have been compared. If the collating values are unequal, the two data fields are unequal and the field with a larger collating value is the greater of the two fields. The collating values are treated as 6-bit unsigned integers. Note that two unequal characters could have the same collating value and would compare equal.

Upon instruction completion, register X0 contains a 60-bit signed integer as follows:

- (Field A) > (Field B)                      (X0) = ℓ - n; (X0) > 0
- (Field A) = (Field B)                      (X0) = 0
- (Field A) < (Field B)                      (X0) = n - ℓ; (X0) < 0

n is the number of pairs of characters that compared equal. If ℓ = 0, then (X0) is 0.

The format of the collating table for 6-bit characters is:

	59	53	47	41	35	29	23	17	11	0
(A0)	00	01	02	03	04	05	06	07		
(A0)+1	10	11	12	13	14	15	16	17		
⋮										
(A0)+7	70	71	72	73	74	75	76	77		

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	CC	$l, k_a, c_a, k_b, c_b$

sym If present, sym is assigned the value of the location counter after the force upper occurs. It becomes the symbolic address of the instruction.

$l$  Absolute address expression specifying the field length in characters (0 through 127).

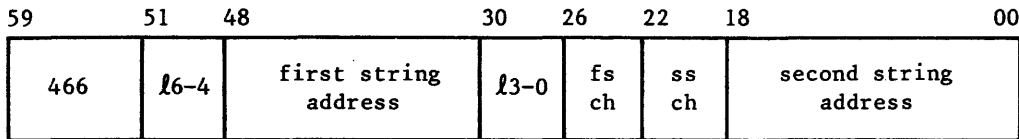
$k_a$  An expression specifying the first word address of the first data field in CM.

$c_a$  An absolute expression specifying the starting character position of the first data field within the word at location  $k_a$ . Characters are numbered from left to right.

$k_b$  An expression specifying the first word address of the second data field in CM.

$c_b$  An absolute expression (0 through 9) specifying the starting character position of the second data field within the word at location  $k_b$ .

**Octal format of instruction:**



**Example:**

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
5100003120 46670050007405007000		SAO CC	TABLE 127,BUFFA,0,BUFFB,5	

**8.5.5 CU - COMPARE UNCOLLATED**

The compare uncollated (CU) symbolic instruction generates a CMU instruction that compares the contents of two data fields, one character at a time, from left to right, until a pair of corresponding characters are found to have unequal values or until the data fields are exhausted. The machine instruction is a 60-bit instruction that occupies one full word and cannot be split between two words. It includes its own data field descriptor. The result of the comparison is placed in register X0.

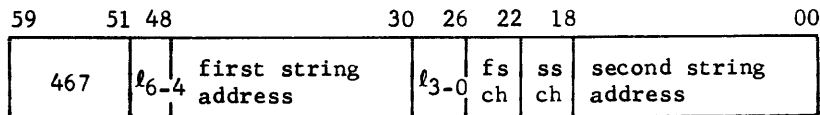
Execution resembles the CC instruction except that A0 and the collating table are not used. Instead, the characters are compared directly with each character regarded as a 6-bit unsigned binary integer. Register X0 is set in the same manner as by the CC instruction.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	CU	$l, k_a, c_a, k_b, c_b$

- sym If present, sym is assigned the value of the location counter after the force upper occurs. It becomes the symbolic address of the instruction.
- $l$  Absolute address expression (0 through 127) specifying the field length in characters.
- $k_a$  An expression specifying the first word address of the first data field in CM.
- $c_a$  An absolute expression (0 through 9) specifying the starting character position of the first data field within the word at location  $k_a$ . Characters are numbered from left to right.
- $k_b$  An expression specifying the first word address of the second data field in CM.
- $c_b$  An absolute expression (0 through 9) specifying the starting character position of the second data field within the word at location  $k_b$ .

**Octal format of instruction:**



**Example:**

Code Generated  
  
46770050007405007000

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	CU	127, BUFFA, 0, BUFFB, 5	



The COMPASS assembler recognizes symbolic notation for peripheral processor (PP or PPU) instructions. For COMPASS to recognize symbolic logic for models 810, 815, 825, 830, 835, 845, 855, 865, and 875 PP instructions, the NOS programmer must ensure that NOSTEXT is available to the assembler. When a PPU or PERIPH pseudo instruction is in the first statement group, the assembler identifies each symbolic instruction by name and generates a one word (12 bit) or two word (24 bit) object code machine instruction under control of the current origin, location, and position counters. All PP code is absolute. Numeric data must be in integer notation. Floating point notation is illegal.

NOTE

No special job validation is required to assemble peripheral processor programs, but to be executed, such programs require system origin privileges.

Some instructions in existing COMPASS programs are not valid for execution on models 810, 815, 825, 830, 835, 845, 855, 865, and 875. To detect these instructions, the programmer can specify S=AIDTEXT in the COMPASS control statement. COMPASS prints a listing of the program, flagging the invalidated instructions with a type O error. S=AIDTEXT should not be specified if the 8 option is chosen for the MACHINE pseudo instruction.

9.1 MACHINE INSTRUCTION FORMATS

An assembled instruction has a 12-bit or 24-bit format. The 12-bit format has a 6-bit operation code f and a 6-bit operand d. A PP accomplishes program indexing and manipulates operands in several modes. The 12-bit and 24-bit instruction formats provide for 6-bit, 12-bit, or 18-bit operands and 6-bit or 12-bit addresses. Figures 9-1 and 9-2 illustrate the 12-bit instruction format and the 24-bit instruction format, respectively.

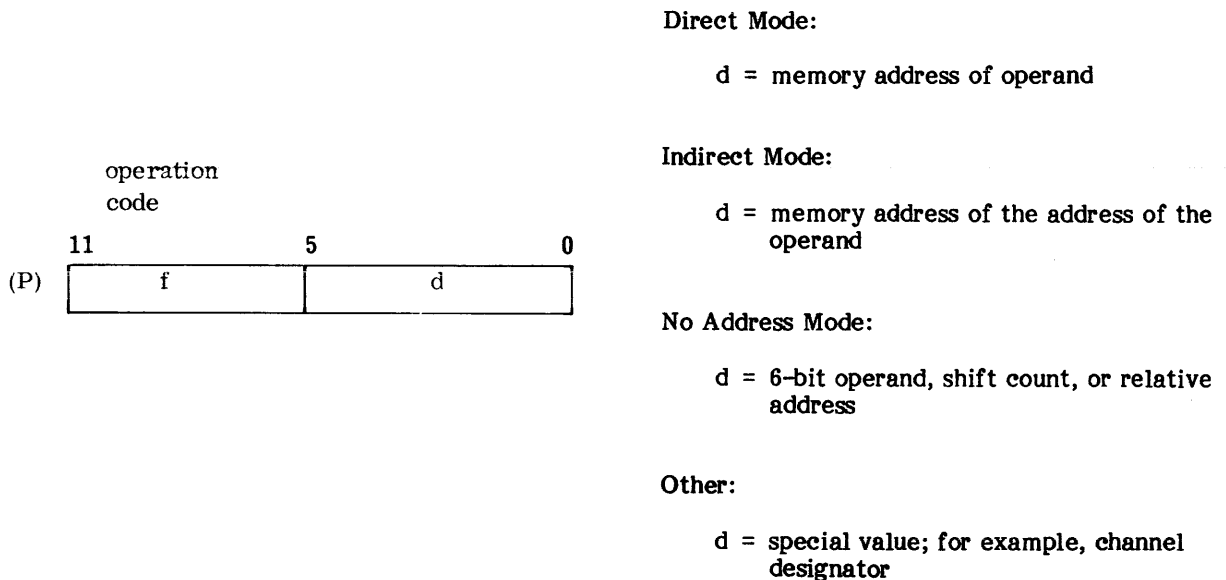


Figure 9-1. PP 12-bit Instruction Format

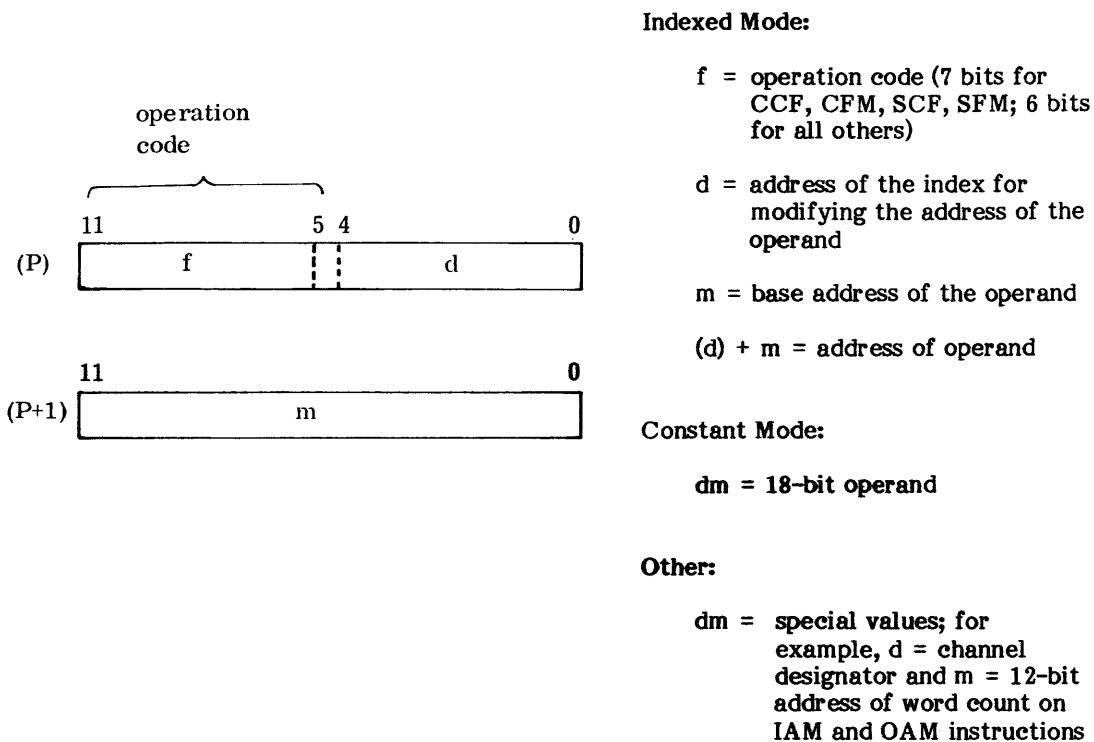


Figure 9-2. PP 24-bit Instruction Format

The 24-bit format uses the 12-bit quantity m, which is the contents of the next program address (P + 1), with d or the contents of d to form an 18-bit operand or a 12-bit operand address.

The central memory access instructions for models 810, 815, 825, 830, 835, 845, 855, 865, and 875 provide the capability of reading and writing central memory words to and from the PP memory.

The R register is a 22-bit register used to accomplish address relocation during central memory read and write instructions. This relocation occurs only if bit 17 of the A register is set to one.

When relocation is to be done, the absolute central memory address is formed by appending six zeros to the lower end of the contents of the R register and adding to the result bits 0 through 16 of the contents of the A register. Figure 9-3 illustrates this process.

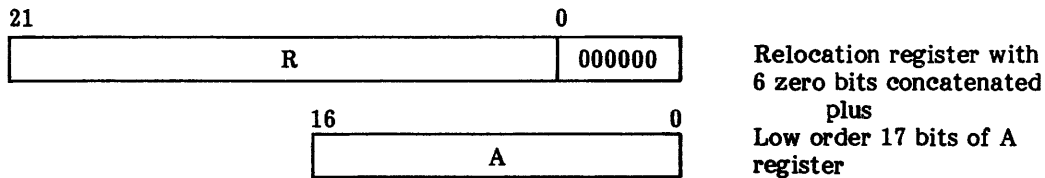


Figure 9-3. Central Memory Access Instruction Address Relocation (Models 810, 815, 825, 830, 835, 845, 855, 865, and 875)

## 9.2 SYMBOLIC NOTATION

This section describes notation used for coding symbolic PP machine instructions. Instructions are described in octal operation code sequence which generally reflects the mode of addressing. Instructions unique to a computer system are identified as such.

The location field of a symbolic PP machine instruction optionally contains a location symbol. When the symbol is present, it is assigned the value of the location counter.

The operation field of a symbolic PP machine instruction contains a three-character name.

The variable field contains one or two subfields. Each subfield contains an absolute or relocatable expression that reduces to a 6-bit, 12-bit, or 18-bit value.

Designators used in this section are listed in table 9-1.

TABLE 9-1. PERIPHERAL PROCESSOR INSTRUCTION DESIGNATORS

Designator	Use
A	18-bit A register.
c	An expression that reduces to an 18-bit operand value.
d	A 6-bit operand or operand address expression. This field is 5 bits long for the SCF, CCF, SFM, and CFM instructions.
m	A 12-bit expression value used with d or (d) to form an 18-bit operand or 12-bit operand address.
P	12-bit Program Address register.
Q	12-bit Q register.
r	An expression that reduces to a 6-bit value ( $-37_8 \leq r \leq 37_8$ ) specifying relative address or shift count.
R	22-bit R register.
()	Contents of a register or location.
(( ))	Refers to indirect addressing.

Generally, the third character of the instruction mnemonic (N, D, M, C, or I) indicates the mode of addressing:

- N No operand address reference
- D Direct operand address: d contains operand
- M Memory address m or m + (d) contains operand
- C 18-bit constant
- I Indirect; operand address is (d)

Some PP instructions can be executed only on specific machine models. Table 9-2 lists each instruction and the machine models to which it corresponds.

TABLE 9-2. PP INSTRUCTION/MACHINE MODEL CORRESPONDENCE

Mnemonic Code	Machine Model Number			
	825, 835, 845, 855, 865, and 875	76 and 7600	171, 172, 173, 174, 175, 720, 730, 740, 750, and 760; 71, 72, 73, and 74; and 6000 Series	176
ACN	X		X	X
ADC	X	X	X	X
ADD	X	X	X	X
ADI	X	X	X	X
ADM	X	X	X	X
ADN	X	X	X	X
AJM	X		X	X
AOD	X	X	X	X
AOI	X	X	X	X
AOM	X	X	X	X
CCF	X			
CFM	X			
CRD	X		X	X
CRM	X		X	X
CWD	X		X	X
CWM	X		X	X
DCN	X		X	X
EIM		X		
EJM	X		X	X
EOM		X		
ERN†				
ESN		X		
ETN†				
EXN	X		X	X
FAN	X		X	X
FIM		X		
FJM	X		X	X
FNC	X		X	X
FOM		X		
IAM	X	X	X	X
IAN	X	X	X	X
IJM	X		X	X
IRM		X		
LCN	X	X	X	X
LDC	X	X	X	X
LDD	X	X	X	X
LDI	X	X	X	X
LDM	X	X	X	X
LDN	X	X	X	X
LJM	X	X	X	X
LMC	X	X	X	X
LMD	X	X	X	X



TABLE 9-2. PP INSTRUCTION/MACHINE MODEL CORRESPONDENCE (Contd)

Mnemonic Code	Machine Model Number			
	825, 835, 845, 855, 865, and 875	76 and 7600	171, 172, 173, 174, 175, 720, 730, 740, 750, and 760; 71, 72, 73, and 74; and 6000 Series	176
LMI	X	X	X	X
LMM	X	X	X	X
LMN	X	X	X	X
LPC	X	X	X	X
LPN	X	X	X	X
LRD	X			
MAN	X		X††	X
MJN	X	X	X	X
MXN		X		
NIM		X		
NJN	X	X	X	X
NOM		X		
OAM	X	X	X	X
OAN	X	X	X	X
ORM		X		
PJN	X	X	X	X
PSN	X	X	X	X
RAD	X	X	X	X
RAI	X	X	X	X
RAM	X	X	X	X
RFN		X		
RJM	X	X	X	X
RPN			X	X
SBD	X	X	X	X
SBI	X	X	X	X
SBM	X	X	X	X
SBN	X	X	X	X
SCF	X			
SCN	X	X	X	X
SFM	X			
SOD	X	X	X	X
SHN	X	X	X	X
SOI	X	X	X	X
SOM	X	X	X	X
SRD	X			
STD	X	X	X	X
STI	X	X	X	X
STM	X	X	X	X
UJN	X	X	X	X
ZJN	X	X	X	X

†6416 only.  
 ††Not supported for 6000 Series.

Some of the instructions provide similar functions using different modes of addressing. They can be grouped according to function as shown below:

<u>Function</u>	<u>Description</u>
Data transmission	<p>The following instructions either load data into the A register or store data from it. A load instruction loads a 6-bit, 12-bit, or 18-bit value as indicated by the instruction; any remaining upper bits of A are zeroed, except for the LCN instruction, for which remaining bits are set to one.</p> <p>A store instruction stores the lower 12 bits of the A register contents into a memory location indicated by the instruction.</p> <p>The contents of A are not altered.</p>

<u>Instruction</u>	<u>Octal Code</u>
LDN	14
LCN	15
LDC	20
LDD	30
STD	34
LDI	40
STI	44
LDM	50
STM	54

Arithmetic	<p>A PP arithmetic instruction adds or subtracts a 6-bit, 12-bit, or 18-bit quantity from the contents of the A register and enters the result in A.</p>
------------	--

<u>Instruction</u>	<u>Octal Code</u>
ADN	16
SBN	17
ADC	21
ADD	31
SBD	32
ADI	41
SBI	42
ADM	51
SBM	52

Logical	<p>A logical instruction forms a logical value in A using the contents of A as one of the operands and a 6-bit, 12-bit, or 18-bit value indicated by the instruction as the second operand. When the second operand is fewer than 18 bits, the remaining upper bits of A are unaltered, except for the LPN instruction for which the upper 12 bits are zeroed.</p>
---------	--

Formation of a logical difference is equivalent to setting each bit in A that is unlike the corresponding bit in the second operand. For example:

Initial (A)	= 0101
Operand	= <u>1100</u>
Final (A)	= 1001

Formation of a logical product is equivalent to setting a bit in A when the original setting of the bit in A and the corresponding bit in the second operand are both ones.

For example:

Initial (A) = 0101  
Operand = 1100  
Final (A) = 0100

A selective clear sets a bit zero in the A register wherever a bit is set in the second operand. For example:

Initial (A) = 0101  
Operand = 1100  
Final (A) = 0001

Logical instructions include the following:

<u>Instruction</u>	<u>Octal Code</u>
LMN	11
LPN	12
SCN	13
LPC	22
LMC	23
LMD	33
LMI	43
LMM	53

Replace

A replace instruction performs an arithmetic operation and returns the results to the A register and the memory location from which one operand was obtained. The lower 12 bits of the result replaces the operand obtained from a memory location. Replace instructions include the following:

<u>Instruction</u>	<u>Octal Code</u>
RAD	35
AOD	36
SOD	37
RAI	45
AOI	46
SOI	47
RAM	55
AOM	56
SOM	57

## 9.2.1 BRANCH INSTRUCTIONS

For branch instructions, the r subfield is a numeric value that indicates the number of locations to be jumped (maximum 31). When r is positive (01 through 37<sub>g</sub>), the jump is forward r locations. When r is negative (-76<sub>g</sub> through -40<sub>g</sub>), the jump is backward 77<sub>g</sub>-r locations. In the following tests, negative zero (777777) is nonzero. For conditional instructions, when the test condition is true, the jump takes place. When the condition is not met, execution continues with the next instruction.

### NOTE

The jump count must not be 00 or 77. If it is, execution loops on the jump instruction.

The J option of the PPU instruction and the PERIPH instruction (chapter 4) cause the value of the location counter to be subtracted from the value of the symbolic address (tag) before it is placed in the d field of the object code instruction.

**Format:**

Operation	Variable	Description	Size	Octal Code
LJM	m, d	Long jump to m+(d); if d = 0, m is not modified	24 bits	01dm
RJM	m, d	Return jump to m+(d); Store P+2 at m+(d) and jump to m+(d)+1.	24 bits	02dm
UJN	r†	Unconditional jump to P+r locations	12 bits	03d
UJN	tag	Unconditional jump to tag	12 bits	03d
ZJN	r†	Zero jump; jump to P+r locations if (A) = 0	12 bits	04d
ZJN	tag	Zero jump to tag	12 bits	04d
NJN	r†	Nonzero jump; jump to P+r locations if (A) ≠ 0	12 bits	05d
NJN	tag	Nonzero jump to tag	12 bits	05d
PJN	r†	Positive jump; jump to P+r locations if (A) ≥ 0	12 bits	06d
PJN	tag	Positive jump to tag	12 bits	06d
MJN	r†	Minus jump; jump to P+r locations if (A) < 0	12 bits	07d
MJN	tag	Minus jump to tag	12 bits	07d

† If PPU J or PERIPH J option has been selected, r is not valid. The contents of the variable field must be a symbolic address (tag).

Example:

Code Generated

0100 1362  
 0271 0000  
 0371  
 0404  
 0525  
 0667  
 0726

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	LJM	START	
	PJM	0,CTO	
	UJN	TAG1-*	
	ZJN	+4	
	NJN	TAG3	
	PJN	TAG2-*	
	MJN	TAG4	

In the above example, the LJM instruction is at address 0014<sub>g</sub>. TAG1 is address 0012<sub>g</sub>, TAG2 has a value of 13<sub>g</sub>, TAG3 has a value of 25<sub>g</sub>, and TAG4 has a value of 26<sub>g</sub>.

Code Generated

0347  
 0404  
 0556  
 0602  
 0743

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	PPJ	J	
	UJN	TAG1	
	ZJN	TAG3	
	NJN	TAG2+10	
	PJN	-1+TAG4	
	MJN	TAG1	

In this example, the UJN is at address 0040. TAG1 is address 0010, TAG2 is 0011, TAG3 is address 0045, and TAG4 is address 0046.

### 9.2.2 SHIFT INSTRUCTION

The SHN instruction shifts the contents of the A register right or left *r* places. If *r* is positive +1 to +31, the shift is left circular *r* places; if *r* is negative (-31 to -1), the shift is end off *r* places to the right with no sign extension. No shift takes place when *r* is  $\pm 0$ . The assembler places the value of the *r* expression in the *d* field. If  $-31 > r > 31$ , the assembler generates an address error.

Format:

Operation	Variable	Description	Size	Octal Code
SHN	<i>r</i>	Shift (A) by + (left) or - (right) <i>r</i> bits	12 bits	10d

**Example:**

**1. Shift contents of A left circular 6 places**

Code Generated

1006

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SHN	6	

**2. Shift contents of A right end off 6 places**

Code Generated

1071

6

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	SCNT	SET SHN	6 -SCNT

**9.2.3 NO ADDRESS MODE INSTRUCTIONS**

In this mode, during instruction execution, the contents of the d field are interpreted as a 6-bit positive operand. This mode eliminates the need for storing many constants in memory.

**Format:**

Operation	Variable	Description	Size	Octal Code
LMN	d	Logical difference (A)-d → A	12 bits	11d
LPN	d	Logical product (A)*d → A	12 bits	12d
SCN	d	Selective clear (A)	12 bits	13d
LDN	d	Load d → A	12 bits	14d
LCN	d	Load complement d → A	12 bits	15d
ADN	d	Add (A)+d → A	12 bits	16d
SBN	d	Subtract (A)-d → A	12 bits	17d

**Example:**

Code Generated

1112

1207

1321

1415

1514

1601

1702

15

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	LMN	12R	
	LPN	7	
	SCN	21R	
AA	SFT	15R	
	LDN	AA	
	LCN	AA-1	
	ADN	1	
	SBN	2	

## 9.2.4 CONSTANT MODE INSTRUCTIONS

In this mode, during instruction execution, the contents of the d and m fields are taken directly as an operand. This mode also eliminates the need for storing many constants. The assembler reduces absolute or relocatable expression c to an 18-bit value and stores the upper six bits in d and the lower 12 bits in m.

Format:

Operation	Variable	Description	Size	Octal Code
LDC	c	Load c →A	24 bits	20dm
ADC	c	Add (A)+c →A	24 bits	21dm
LPC	c	Logical product (A)*c →A	24 bits	22dm
LMC	c	Logical difference (A)-c →A	24 bits	23dm

Example:

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
2070 7070		LDC	70707070	
2177 7776	VAL	=	0	
		ADC	VAL-1	
2207 0707		LPC	07070707	
2307 0707	MASK	SET	07070707	
		LMC	MASK	

## 9.2.5 NO OPERATION INSTRUCTION

The PSN instruction specifies that no operation is to be performed. It provides a means of padding a program.

For the models 810, 815, 825, 830, 835, 845, 855, 865, and 875, the variable field of the PSN instruction must be blank. Otherwise it is interpreted as an LRD instruction.

Format:

Operation	Variable	Description	Size	Octal Code
PSN		No operation (Pass)	12 bits	2400

Example:

<u>Code Generated</u>	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
2400		PSN		

Other octal operation codes (not generated by COMPASS) that act as pass instructions are:

<u>CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series</u>	<u>CYBER 70 Model 76 and 7600</u>
--	---------------------------------------

00		25
25	(2500 for Models 810, 815, 825, 830, 835, 845, 855, 865, and 875)	27
2700	(Models 810, 815, 825, 830, 835, 845, 855, 865, and 875 only)	76

### 9.2.6 LOAD AND STORE R REGISTER INSTRUCTIONS

The LRD instruction loads the R register. Bits 0 through 11 of the R register are loaded from d+1; bits 12 through 21 of R are loaded from bits 0 through 9 of d.

SRD stores the contents of the R register into d and d+1. Bits 0 through 11 of R are stored into d+1; bits 12 through 21 of R are stored into bits 0 through 9 of d.

If the variable field is set to zero, LRD and SRD execute as pass instructions.

Format:

Operation	Variable	Description	Size	Octal Code
LRD	d	Load (R) from d and d+1	12 bits	24d
SRD	d	Store (R) into d and d+1	12 bits	25d

Example:

<u>Code Generated</u>	<u>LOCATION</u>	<u>OPERATION</u>	<u>VARIABLE</u>	<u>COMMENTS</u>
	1		18	30
2400		LRD		PASS INSTRUCTION
2500		SRD		PASS INSTRUCTION
2412		LRD	12B	
2512		SRD	12B	

### 9.2.7 EXCHANGE JUMP INSTRUCTIONS

The EXN instruction transmits an 18-bit (absolute) address from the A register to the CPU with a signal notifying the CPU to execute an exchange jump. The address in A is the starting location of the 16-word exchange package which contains information about the CPU program to be executed. The 18-bit initial address must be entered in A before the EXN instruction is executed. The CPU replaces the file with similar information from the interrupted CPU program. The PP is not interrupted. The EXN instruction does not affect the monitor flag bit.



The MXN instruction conditionally exchange jumps to the CPU and initiates CPU monitor activity. If the monitor flag bit is clear, this instruction sets the flag and initiates the exchange. If the monitor flag bit is set, this instruction acts as a pass instruction. The starting address for this exchange is the 18-bit address in the PP A register. This address must be entered in A before the MXN instruction is executed.

Execution of MAN resembles MXN. However, the exchange package address is taken from the 18-bit Monitor Address (MA) register in CPU d, rather than from the PP A register.

In a system with dual central processors, d can be 0 or 1 and specifies which CPU the exchange jump will interrupt. In single processor systems, this value is not interpreted.

Format:

Operation	Variable	Description	Size	Octal Code
EXN	d	Exchange jump CPU d to (A)	12 bits	260d
MXN	d	Monitor exchange jump CPU d to (A)	12 bits	261d
MAN	d	Monitor exchange jump CPU d to (MA)	12 bits	262d

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1		18	30
2601		EXN	1	
2610		MXN	0	
2623		MAN	3	

### 9.2.8 READ PROGRAM ADDRESS INSTRUCTION

This instruction transfers the contents of the CPU P register to the PP A register; this allows the PP to determine whether the CPU is in execution. In a dual central processor system, the lowest order bit of the instruction format specifies which CPU P register is to be examined. This bit is not interpreted for a single central processor system.

Format:

Operation	Variable	Description	Size	Octal Code
RPN	d	Read program address CPU d → A	12 bits	270d

Example:

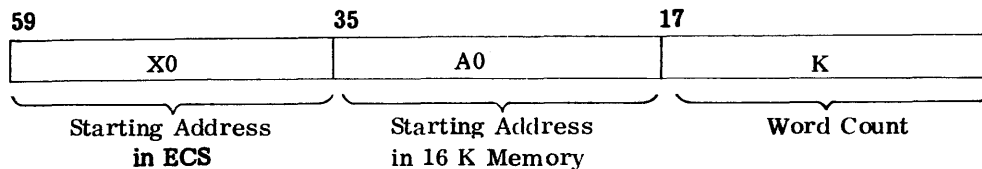
Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1		18	30
2700		RPN		

For the 6000 and CYBER 70 Series, the largest value that (P) can be is 17 bits. An ECS transfer is in progress when bit 17 of the A register is set. For the CYBER 170 series, the P register is 18 bits.

The RPN instruction is not valid for the models 810, 815, 825, 830, 835, 845, and 855. The 2700 octal code executes as a pass instruction.

### 9.2.9 6416 PP INSTRUCTIONS

COMPASS assembles the following instructions for execution on a 6416 computer system only. The ETN instruction initiates memory transfer operations by transmitting an 18-bit address from the PP A register to the 6416 16K memory. This address points to a word having the following format:



Expression d of this instruction specifies the transfer to be performed:

- If d is 0, K words are transferred from ECS to 16K memory.
- If d is 1, K words are transferred from 16K memory to ECS.

Note that addresses contained in the word are absolute addresses. Operating systems may require relocation (adding RA to an address) and field length testing, e.g., Is address + RA > FL? The Exchange Jump package contains RA and FL values for central memory and for extended memory. The 6416 has no hardware for automatic relocation and field length testing; it is therefore incumbent upon the program to perform these functions whenever required by an operating system.

The ERN instruction examines the status of the data trunk between 16K memory and the extended core coupler. If the data trunk is busy (a transfer is in progress), a 1 is placed in the most significant bit position of the A register. If the trunk is free (not busy), the A register remains cleared. The d portion of this instruction is ignored.

After execution of this instruction the program would typically test the A register for a sign before executing an instruction that initiates an ECS operation.

Format:

Operation	Variable	Description	Size	Octal Code
ETN	d	Extended core transfer	12 bits	260d
ERN	d	Read extended core coupler status	12 bits	270d

Example:

Code Generated

2600

2700

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	ETN		
	ERN		

## 9.2.10 DIRECT ADDRESS MODE INSTRUCTIONS

In this mode, during instruction execution, the contents of the d field specify the address of the operand. During assembly, the assembler reduces absolute or relocatable expression d to a 6-bit value that specifies one of the first 100<sub>8</sub> addresses in memory (0000 through 0077<sub>8</sub>). During instruction execution, (d) is treated as a positive 12-bit quantity.

Format:

Operation	Variable	Description	Size	Octal Code
LDD	d	Load (d) → A	12 bits	30d
ADD	d	Add (A) + (d) → A	12 bits	31d
SBD	d	Subtract (A) - (d) → A	12 bits	32d
LMD	d	Logical difference (A) and (d) → A	12 bits	33d
STD	d	Store (A) → d	12 bits	34d
RAD	d	Replace add (d) + (A) → d and A	12 bits	35d
AOD	d	Replace add (d) + 1 → d and A	12 bits	36d
SOD	d	Replace subtract one (d) - 1 → d and A	12 bits	37d

Example:

Code Generated

3012  
3103  
3240  
3327  
3401  
3555  
3612  
3713

LOCATION	OPERATION	VARIABLE	COMMENTS
11		18	30
	LDD	TAG1	
	ADD	TAG2-10B	
	SBD	43B	
	LMD	TAG1+15B	
	STD	1	
	RAD	55B	
	AOD	TAG1	
	SOD	TAG2	

## 9.2.11 INDIRECT ADDRESS MODE INSTRUCTIONS

In this mode, during instruction execution, d specifies an address, the contents of which specify the address of the desired operand. Thus, d specifies the operand address indirectly.

During assembly, the assembler reduces absolute or relocatable expression d to a 6-bit value that specifies one of the first 100<sub>8</sub> addresses in memory (0000 through 0077<sub>8</sub>).

On the 7600 (or CYBER 70 Model 76), the address formed permits referencing of memory locations 0000 through 7776<sub>8</sub>. Location 7777<sub>8</sub> cannot be referenced.

On a 6000 Series Computer System (as well as CYBER 180 Series or CYBER 170 Series or CYBER 70 Model 71, 72, 73, or 74) PP, the address formed in indirect address mode permits referencing of all memory locations, including address 7777<sub>g</sub>.

**Format:**

Operation	Variable	Description	Size	Octal Code
LDI	d	Load ((d)) → A	12 bits	40d
ADI	d	Add (A) + ((d)) → A	12 bits	41d
SBI	d	Subtract (A) - ((d)) → A	12 bits	42d
LMI	d	Logical difference (A) - ((d)) → A	12 bits	43d
STI	d	Store (A) → (d)	12 bits	44d
RAI	d	Replace add ((d)) + (A) → (d) and A	12 bits	45d
AOI	d	Replace add one ((d)) + 1 → (d) and A	12 bits	46d
SOI	d	Replace subtract one ((d)) - 1 → (d) and A	12 bits	47d

**Example:**

Code Generated

4012  
4103  
4240  
4327  
4401  
4555  
4612  
4713

LOCATION	OPERATION	VARIABLE	COMMENTS
1	LDI	TAG1	
	ADI	TAG2-10	
	SBI	420	
	LMI	TAG1+150	
	STI	1	
	RAI	550	
	AOI	TAG1	
	SOI	TAG2	

**9.2.12 INDEXED DIRECT ADDRESS MODE INSTRUCTIONS**

In this mode, during instruction execution, The value formed by m + (d) is used as the address of the operand. During assembly, the assembler reduces absolute or relocatable expression d to a 6-bit value that specifies one of the first 100<sub>g</sub> addresses in memory (0000 through 0077<sub>g</sub>). The value of absolute or relocatable expression m is a 12-bit base address.

**NOTE**

The address formed in indexed addressing permits referencing of all memory locations but one (0000 through 7776<sub>g</sub>). Although m and/or (d) can have a value of 7777<sub>g</sub>, the computer system does not permit m + (d) to reference address 7777<sub>g</sub>.

When in indexed direct address mode, if d is nonzero the contents of address d are added to m to produce a 12-bit operand address (indexed addressing). If d is zero, m is taken as the operand address.

Format:

Operation	Variable	Description	Size	Octal Code
LDM	m, d	Load $(m+(d)) \rightarrow A$	24 bits	50dm
ADM	m, d	Add $(A) + (m+(d)) \rightarrow A$	24 bits	51dm
SBM	m, d	Subtract $(A) - (m+(d)) \rightarrow A$	24 bits	52dm
LMM	m, d	Logical difference $(A) - (m+(d)) \rightarrow A$	24 bits	53dm
STM	m, d	Store $(A) \rightarrow m+(d)$	24 bits	54dm
RAM	m, d	Replace add $(m+(d)) + (A) \rightarrow m+(d)$ and A	24 bits	55dm
AOM	m, d	Replace add one $(m+(d)) + 1 \rightarrow m+(d)$ and A	24 bits	56dm
SOM	m, d	<b>Replace subtract one <math>(m+(d)) - 1 \rightarrow m+(d)</math> and A</b>	24 bits	57dm

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
5077 0203	1	LDM	TAG6, 77B	18
5106 0202		ADM	TAG5, 6	30
5200 0202		SBM	TAG5	
5315 7000		LMM	7000B, 15B	
5410 0272		STM	TAG5+70B, TAG1-2	
5500 0342		RAM	140B+TAG5, 0	
5600 0173		AOM	-10B+TAG6	
5712 0203		SOM	TAG6, TAG1	

### 9.2.13 CENTRAL READ/WRITE INSTRUCTIONS

The CRD instruction transfers a 60-bit word from central memory to five consecutive PP locations. The 18-bit address of the central memory location must be loaded into A prior to executing this instruction. (Note that this is an absolute address.) The 60-bit word is disassembled into five 12-bit words beginning at the left. Location d receives the first 12-bit word. The remaining 12-bit words go to successive locations. The contents of A are not altered.

The CRM instruction reads a block of 60-bit words from central memory. The contents of location d give the block length. The 18-bit address of the first central word must be loaded into A prior to executing this instruction. (Note that this is an absolute address.) During the execution of the instruction, the contents of P go to processor address 0 and P holds m. Also, the block length (from d) goes to the Q register where

it is reduced by one as each central word is processed. The original content of P is restored at the end of the instruction. The new contents of P are fetched from word 0. If the read operation overwrote the contents of word 0, the restored value of P will be different from the original contents.

The contents of A are incremented by one to provide the next central memory address after each 60-bit word is disassembled and stored. The contents of the Q register are also reduced by one. The block transfer is complete when (Q)=0. The block of central memory locations proceeds from address (A) to address (A)+(d)-1. The block of processor memory locations proceeds from address m to m+5(d)-1.

Each central word is disassembled into five 12-bit words beginning with the high-order 12 bits. The first word is stored at processor memory location m. The content of P (which is holding m) is advanced by one to provide the next address in the processor memory as each 12-bit word is stored. If P overflows, operation continues as P is advanced from 7777<sub>g</sub> to 0000<sub>g</sub>. These locations will be written into as if they were consecutive.

The CWD instruction assembles five successive 12-bit words into a 60-bit word and stores the word in central memory. The 18-bit address word designating the central memory location must be in A prior to execution of the instruction. (Note that this is an absolute address.)

Location d holds the first word to be read out of the processor memory. This word appears as the higher order 12 bits of the 60-bit word to be stored in central memory. The remaining words are taken from successive addresses.

The CWM instruction assembles a block of 60-bit words and writes them in central memory. The content of location d gives the number of 60-bit words. The content of the A register gives the beginning central memory address. (Note that this is an absolute address.) During the execution of this instruction (P) goes to processor address 0, and P holds m. Also, (d) goes to the Q register, where it is reduced by one as each central word is assembled. The original content of P is restored at the end of the instruction.

The content of P (the m portion of the instruction) gives the address of the first word to be read out of the processor memory. This word appears as the higher order 12 bits of the first 60-bit word to be stored in central memory.

The content of P is advanced by one to provide the next address in the processor memory as each 12-bit word is read. If P overflows, operation continues as P is advanced from 7777<sub>g</sub> to 0000<sub>g</sub>. These locations will be read from as if they were consecutive.

(A) is advanced by one to provide the next central memory address after each 60-bit word is assembled. Also, Q is reduced by one. The block transfer is complete when (Q)=0.

**Format:**

Operation	Variable	Description	Size	Octal Code
CRD	d	Central read from (A) to d	12 bits	60d
CRM	m,d <sup>†</sup>	Central read (d) CM words beginning at CM (A) → PP m	24 bits	61dm
CWD	d	Central write from d to (A)	12 bits	62d
CWM	m,d <sup>†</sup>	Central write (d) words beginning at PP m → CM (A)	24 bits	63dm
<sup>†</sup> Expression d is required.				

**Example:**

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
6015		CRD	158	
6125 0012		CRM	TAG1,258	
6232		CWD	328	
6350 0012		CWM	TAG1,508	

### 9.2.14 I/O BRANCH INSTRUCTIONS

The following instructions are conditional long jump instructions, each of which tests for a condition on channel d. When the condition is true, the jump to address m takes place. When the condition is not met, execution continues with the next instruction. The d expression is required.

For the FJM instruction, an input channel is full when the input equipment has sent a word to the channel register and sets the full flag. The channel remains full until the PP accepts the word and clears the flag. An output channel remains full when a PP sends a word to the channel register and sets the full flag. The channel is empty when the output equipment accepts the word and notifies the PP.

On the models 810, 815, 825, 830, 835, 845, 855, 865, and 875, d must be less than 40g.

**Format:**

Operation	Variable	Description	Size	Octal Code
AJM	m,d	Jump to m if channel d active	24 bits	64dm
IJM	m,d	Jump to m if channel d inactive	24 bits	65dm
FJM	m,d	Jump to m if channel d full	24 bits	66dm
EJM	m,d	Jump to m if channel d empty	24 bits	67dm

**Example:**

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	I	II	18	30
6402 0012		AJM	TAG1,2	
6502 0013		IJM	TAG2,CHAN-2	
6604 0025		FJM	TAG3,4	
6714 0026		FJM	TAG4,CHAN	

## 9.2.15 I/O TEST AND SET CHANNEL FLAG INSTRUCTIONS

The SCF instruction branches to the location specified by m if the channel d flag is set; otherwise, it sets the channel flag and exits. The programmer can unconditionally set the channel flag by setting m to P+2.

The CCF instruction clears the flag in the channel specified by d. The m field is required, but not used.

The SFM instruction branches to the location specified by m if the channel d error flag is set, and clears the error flag.

The CFM instruction branches to the location specified by m if the channel d error flag is clear; otherwise, it clears the error flag.

### Format:

Operation	Variable†	Description	Size	Octal Code††
SCF	m,d	Branch to m if channel d flag set	24 bits	644dm
CCF	m,d	Clear channel d flag	24 bits	654dm
SFM	m,d	Branch to m if channel d error flag set	24 bits	664dm
CFM	m,d	Branch to m if channel d error flag clear	24 bits	674dm

†The variable d is a 5-bit field containing the channel number.

††The operation code occupies 7 bits.

### Example:

#### Code Generated

6445 0100  
 6545 0100  
 6645 0100  
 6745 0100  
 6453 0100  
 6553 0100  
 6653 0100  
 6753 0100

LOCATION	OPERATION	VARIABLE	COMMENTS
I	II	18	30
	SCF	100B,5	
	CCF	100B,5	
	SFM	100B,5	
	CFM	100B,5	
	SCF	100B,13B	
	CCF	100B,13B	
	SFM	100B,13B	
	CFM	100B,13B	



### 9.2.16 I/O BRANCH INSTRUCTIONS

The following instructions are conditional long jump instructions, each of which tests a condition on channel d. When the condition is true, the jump to address m takes place. When the condition is not met, execution continues with the next instruction. These instructions are exclusively 7600 PPU instructions. The d expression is required.

Format:

Operation	Variable	Description	Size	Octal Code
FIM	m,d	Jump to m on channel d input word flag	24 bits	60dm
EIM	m,d	Jump to m if no input word flag on channel d	24 bits	61dm
IRM	m,d	Jump to m on channel d input record flag	24 bits	62dm
NIM	m,d	Jump to m if no input record flag on channel d	24 bits	63dm
FOM	m,d	Jump to m on channel d output word flag	24 bits	64dm
EOM	m,d	Jump to m if no output word flag on channel d	24 bits	65dm
ORM	m,d	Jump to m on channel d output record flag	24 bits	66dm
NOM	m,d	Jump to m if no output record flag on channel d	24 bits	67dm

Example:

Code Generated

```

6005 1365
6102 1365
6201 1366
4
6304 1366
6415 7000
6500 1525
6601 1256
6705 1366

```

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	FIM	TAG5,5	
	EIM	TAG5,2	
	IRM	TAG6,1	
4	SET	4	
	NIM	TAG6,CHAN	
	FOM	70000,15B	
	EOM	1400+TAG5,2	
	ORM	-1100+TAG6,CHAN-3	
	NOM	TAG6,CHAN+1	

### 9.2.17 A REGISTER INPUT/OUTPUT INSTRUCTIONS

The following instructions transfer a word to or from channel d and the lower 12 bits of the A register.

On the CYBER 70 Model 76 and the 7600, the IAN instruction is not executed until the input channel d word flag is set. If the flag is not set when the instruction is read, execution halts until an external signal sets the flag. The input channel d record flag does not affect the IAN execution. The IAN instruction clears the input channel d word flag and record flag and transmits a resume signal over the input cable after the word is entered in the A register.

On the CYBER 70 Model 76 and the 7600, the OAN instruction is not executed while the output channel d word flag is set. If the flag is set, execution stops until an external resume signal clears the flag. This instruction sets the output channel d word flag and transmits a work pulse over the output channel cable.

- On a CYBER 180 Series; a CYBER 170 Series; CYBER 70 Model 71, 72, 73, 74; or 6000 Series machine, executing either of these instructions when the channel is inactive causes the peripheral processor to become inoperative until some other peripheral processor activates the channel or the system is deadstarted.

Format:

Operation	Variable	Description	Size	Octal Code
IAN	d	Input: channel d to A	12 bits	70d
OAN	d	Output: (A) to channel d	12 bits	72d

Example:

Code Generated

7003  
7204

LOCATION	OPERATION	VARIABLE	COMMENTS
1	IAN	18	30
	OAN	CHAN	

### 9.2.18 BLOCK INPUT/OUTPUT INSTRUCTIONS

The following instructions transfer a block of 12-bit words on channel d to or from a starting PP memory location specified by m. The number of words transferred is specified by the contents of the A register which is reduced by one as each word is transferred. The operation is completed when (A)=0 or the channel becomes inactive (CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 only).

- On the CYBER 180 Series; the CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series machines, the input operation is complete when the contents of A equal 0 or the data channel becomes inactive. If the operation is terminated by the channel becoming inactive, the next location in the processor memory is set to all zeros. The word count is not affected by this empty word. Therefore, the contents of the A register give the block length minus the number of real data words actually read in.

During execution of either of these instructions, address 0000 temporarily holds P, while the P register holds m. The contents of P advance by one to give the address for the next word as each word is transferred.

If a read operation overwrites word 0 (address 0000), the restored value of P may be different from the contents of P before the operation.

**NOTE**

If this instruction is executed on a CYBER 180 Series; CYBER 170 Series; a CYBER 70 Model 71, 72, 73, or 74; or 6000 Series machine when the data channel is inactive, no operation is accomplished and the program continues at P + 2. However, the location specified by m is set to all zeros for the IAM instruction.

On a CYBER 70 Model 76 or a 7600, the IAM instruction is not executed until the input channel d word flag is set. If the flag is not set when the instruction is read, execution halts until an external signal sets the flag. The presence of an input channel d record flag is ignored for the first word of the block but terminates the block input at any word after the first. In this case, the next location in the PP block input storage area contains a noise word; any remaining locations are unaltered. Note that the storage location can be incremented through location 7776<sub>8</sub> to 000<sub>8</sub> on a 7600 (or CYBER 70 Model 76), or location 7777<sub>8</sub> through 0000 on a 6000 Series machine (or a CYBER 180 Series; a CYBER 170 Series; CYBER 70 Model 71, 72, 73, or 74), which could destroy existing data or a program.

On a CYBER 70 Model 76 or a 7600, the OAM instruction is not executed until the output channel d word flag is cleared. If the flag is set when the instruction is read, execution halts until a resume pulse clears the flag. An output channel d record flag does not affect OAM execution.

**Format:**

Operation	Variable	Description	Size	Octal Code
IAM	m, d <sup>†</sup>	Input: (A) words to m from channel d	24 bits	71dm
OAM	m, d <sup>†</sup>	Output: (A) words to channel d from m	24 bits	73dm
†Expression d is required.				

**Example:**

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
7103 1364	11	IAM	TAG, 3	30
7304 1364		OAM	TAG, 4	

**9.2.19 SET OUTPUT RECORD FLAG INSTRUCTION**

The RFN instruction sets the output channel d record flag and transmits a record pulse over the cable. The instruction ignores the previous status of the channel d flags; the instruction is executed even if the output channel d record flag is set.

**Format:**

Operation	Variable	Description	Size	Octal Code
RFN	d	Set output record flag on channel d	12 bits	74d

**Example:**

Code Generated

7406

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
		PFN	6	

**9.2.20 CHANNEL FUNCTION INSTRUCTIONS**

The ACN instruction activates the channel specified by d. This instruction must precede the IAN, IAM, OAM, or OAN instructions. Activating a channel alerts the input/output equipment for the exchange of data. Activating an already active channel causes the PP to become inoperative until another PP or an external equipment deactivates the channel, or the system is deadstarted.

The DCN instruction deactivates the channel specified by expression d. It stops the input/output equipment and terminates the buffer. Deactivating an already inactive channel causes the PP to become inoperative until deadstart or until the channel is activated. Avoid disconnecting the channel before first sensing for channel empty, deactivating a channel before stopping the associated processor, or deactivating a channel before placing a useful program into the associated processor. After deadstart, PPs wait on an input channel. Deactivating a channel after deadstart causes an exit to address 0001 and execution of the program.

The FAN instruction sends the external function code from the lower 12 bits of the A register on channel d.

The FNC instruction sends the external function code specified by m on channel d. For this instruction, expression d is required.

Execution of a FAN or FNC instruction when the channel is active causes the PP to become inoperative until another PP or an external equipment deactivates the channel, or the system is deadstarted.

**Format:**

Operation	Variable	Description	Size	Octal Code
ACN	d	Activate channel d	12 bits	74d
DCN	d	Disconnect channel d	12 bits	75d
FAN	d	Function (A) on channel d	12 bits	76d
FNC	c, d	Function c on channel d	24 bits	77dm

**Example:**

7405

7504

7605

7705 0020

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
		ACN	5	
		DCN	CHAN	
		FAN	CHAN+1	
		FNC	20F,5	

### 9.2.21 ERROR STOP INSTRUCTION

The ESN instruction halts execution of the peripheral processor program and indicates a program error condition to the monitor control unit. The PPU must be restarted by a deadstart sequence from the MCU, only.

Format:

Operation	Variable	Description	Size	Octal Code
ESN	d	Error Stop	12 bits	7700

Example:

Code Generated

7700

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		ESN		





**Examples:**

```
JOB1, T100, EC30.  
TESTER.
```

**10.1.2 COMPASS CONTROL STATEMENT**

The following statement causes the COMPASS assembler to be loaded from the library and executed. Parameters specify modes and files.

**Format:**

```
COMPASS(p1, p2, ..., pn)
```

The optional parameters, p, may be in any order within the parentheses. A parameter can be omitted or can be in one of the following forms:

- mode
- mode=0
- mode=lfm

Mode is one or two characters as described below; lfm is a 1 through 7 character name of a file or a character string.

<u>Mode</u>	<u>Significance</u>
A - Abort mode.	
A	Abort job step at end of run if any assembly errors occurred.
omitted	Do not abort job step for assembly errors.
B - Binary output.	
omitted or B	Binary on the load-and-go file (LGO).
B=0	No binary output.
B=lfm	Binary on the named file.
BL - Burstable listing.	
BL	Generates output listing easily separable into components: <ul style="list-style-type: none"><li>● Issues page ejects between listing segments (storage allocation map, source code, and cross reference table).</li><li>● Assures an even number of pages (page parity) for each program unit listing, issuing a blank page at end if necessary.</li></ul>
omitted or BL=0	Generates listings in compact format. Page parity and listing segment page ejects are suppressed.



D - Debug mode.

D Binary is generated on the file indicated by B parameter in spite of assembly errors and regardless of the abort mode (A parameter). The A parameter is ignored when the D parameter is selected.

D is ignored if B=0.

omitted Assembly errors inhibit binary output. In abort mode (A parameter present), no binary output is written at all for a subprogram containing assembly errors. Otherwise (A parameter omitted), the message ERRORS IN ASSEMBLY is written to the file indicated by the B parameter for each subprogram containing assembly errors.

E - Error list. Suppressed if full list is directed to the same file or if no assembly errors occur. However, if the full list and error list are on different files (for example, the full list is written to OUTPUT and the error list is written on the named file), the error list will contain all statements having error flags. If an error line was generated by a macro call, the macro call can also appear in the error list. Specification of both the E and the O parameter results in a control statement error.

omitted Error list on file OUTPUT.

E Error list on file ERRS.

E=ifn Error list on named file.

E=0 No error list is generated (equivalent to directing error list to the same file as full list).

F - FORTRAN mode. Establishes value of special element \*F.

omitted or F \*F is 0.

F=number \*F is number (one decimal digit).

F=name \*F is a number corresponding to name as follows:

COMPASS = 0  
RUN = 1 (The RUN compiler is no longer supported.)  
FTN4 = 2  
FTN5 = 3

G - Get system text.

omitted or G=0 Load no system text from a sequential binary file.

G Load the first system text overlay, if any, from file named SYSTEXT.

G=ifn Load the first system text overlay, if any, in the specified sequential binary file.

G=ifn/ovl Search the specified sequential binary file for a system text overlay whose name is ovl and load the first such overlay.

I - Source of assembler input.

omitted Source deck is on INPUT file.

I Source deck is on COMPILE file in either compressed (see the UPDATE control statement X option, in the Update reference manual) or expanded format.

- I=0**                    **Illegal.**
- I=ifn**                **Source deck is on named file.**
- L - Full list.**
- omitted or **L**            **List output on OUTPUT file.**
- L=ifn**                **List output on named file. When the full list is on a different file than the short list (see O option) and the P option is not specified, the listing for each subprogram is a separate section beginning with a one-word header consisting of an asterisk and the first six characters of the subprogram name. This header identifies the subprogram as a convenience for sorting and cataloging.**
- L=0**                    **No full list will be generated.**
- LO - List options. Selects or deselects a maximum of nine of the list options A, B, C, D, E, F, G, L, M, N, R, S, T, or X.**
- omitted or **LO=0**        **Same as selecting B, L, N, and R only.**
- LO**                    **Selects list options C, F, G, and X, and deselects R.**
- LO=c<sub>1</sub>c<sub>2</sub>...c<sub>n</sub>**        **A list of up to nine characters. Inclusion of B, L, N, or R deselects the corresponding option. Otherwise, inclusion of a character selects the option. For options, refer to LIST pseudo instruction, chapter 4.**
- LO=\$\$\$\$**                **Selects all list options.**
- ML - Initial Value of MODLEVEL Micro.**
- omitted or **ML**        **MODLEVEL is defined equal to JDATE (chapter 7) at the start of each assembly.**
- ML=string**            **MODLEVEL is defined as string (nine characters maximum) at the start of each assembly.**
- N - No eject.**
- omitted                **Explicit ejects are honored.**
- N**                    **Explicit ejects (from TITLE or EJECT pseudo instruction) are suppressed.**
- O - Short list. Suppressed if full list is directed to the same file or if no assembly errors occur. However, if the full list and short list are on different files (for example, the full list is written on OUTPUT and the short list is written on the named file), the short list will contain all statements having error flags. If an error line was generated by a macro call, the macro call may also be in the short list. Specification of both the O parameter and the E parameter results in a control statement error.**
- omitted or **O**            **List output on OUTPUT file.**
- O=ifn**                **List output on named file.**
- O=0**                    **No short list will be generated (equivalent to directing short list to the same file as full list).**

- P - Continue page.**
- P** Page numbering continues from subprogram to subprogram, creating a single continuous listing file. End-of-record is also suppressed between routines on the L file.
- omitted** Page numbering begins with 1 at the start of each subprogram.
- PC - Initial Value of PCOMMENT Micro.**
- omitted or PC** PCOMMENT is defined as 30 blanks at the start of each assembly.
- PC=string** PCOMMENT is defined as string at the start of each assembly. Characters are truncated from the right or blanks are appended to the right, as necessary, so that the length of the micro value is exactly 30 characters.
- PD - Print Density. Job default print density is assumed upon entry. This option affects only the listing files.**
- PD=6** Print density is six lines per inch.
- PD=8 or PD** Print density is eight lines per inch.
- PD=other or omitted** Print density defaults to job default (an installation parameter, user changeable) lines per inch.
- PS - Page Size.**
- PS=x** Page size is x lines per page. Acceptable values of x are  $4 \leq x \leq 99$ .
- PS=other or omitted** If PD is not specified, page size defaults to job default lines per page. If PD is specified, page size defaults to  $PS = (PD * \text{job default page size}) / \text{job default print density}$ .
- S - System Text Name.**
- omitted** If there are no G parameters other than G=0, load the overlay named SYSTEXT from the job's current global library set.
- S=0** Load no system text from a library.
- S** Load system text overlay named SYSTEXT from job's current global library set.
- S=ovl** Load the system text overlay named ovl from the job's current global library set.
- S=lib/ovl** Load the system text overlay named ovl from the library named lib, which may be a user library file or a system library.
- X - Source of external text (XTEXT) when location field of XTEXT pseudo instruction is blank.**
- omitted** External text OLDPL file.
- X=lfm** External text on named file.
- X=0** Illegal.
- X** External text on OPL file.

**Example:**

<code>COMPASS(B,D,S=OVI)</code>	Reads source from INPUT, writes the binary output to LGO, and the listing to OUTPUT. Assemble in debug mode with system text from overlay OVI in the global library set.
<code>COMPASS(LO=ASGXD)</code>	Disables LIST pseudo instruction and sets LIST options A, S, G, X, and D.
<code>COMPASS.</code>	Uses the standard default options.

**MULTIPLE SYSTEM TEXT OVERLAYS**

COMPASS allows up to seven system text overlays to be used for an assembler run. They are specified by G and S parameters on the COMPASS control statement. Each G parameter (except G=0) specifies loading of a system text overlay from a sequential binary file, and each S parameter (except S=0) specifies loading of a system text overlay from a user library file or a system library. The G and S parameters can be used in any combination and in any order, and can be intermixed freely with other parameters, provided the total number of system text overlays specified does not exceed seven. COMPASS loads the system text overlays in the order in which the G and S parameters occur on the COMPASS statement. If a system macro, micro, or symbol is defined by more than one system text, only the last definition is used. S=0 has no effect if there are any other S or G parameters.

**Example:**

<code>COMPASS(I,S,S=PFMTEXT,G=MYTEXT)</code>	Reads source from file COMPILE and gets system text from overlays SYSTEXT and PFMTEXT in the global library set, and from the local file MYTEXT.
<code>COMPASS(G=FILE/SCPTEXT,S=MYLIB/TEXT)</code>	Get system text from overlay SCPTEXT on the file FILE, and from overlay TEXT in library MYLIB.

**10.1.3 LGO CONTROL STATEMENT**

An LGO control statement calls for the loading and execution of CPU binary output produced by the assembler unless the B option on the COMPASS control statement is set to 0 or to some other file name. When binary output is on some file other than LGO, the statement is replaced by a program call statement for that file. The file is automatically rewound before loading. The LGO file is temporary; it is released at job termination.

**NOTE**

A peripheral processor program can be executed only by the operating system. This type of program execution requires system origin privileges.

**Format:**

<code>LGO(p<sub>1</sub>,p<sub>2</sub>,p<sub>3</sub>,...,p<sub>n</sub>)</code>	or	<code>LGO.</code>
---	----	-------------------

**10.1.4 PROGRAM CALL STATEMENT**

The program call statement directs the operating system to search for a file or CPU program that has the specified name, load it into central memory (CM or SCM), and execute it as a CPU program.

**Formats:**

```
name(p1,p2,...,pn)  
name.
```

name                    Program name.

P<sub>i</sub>                    Parameters in a format acceptable to the program being called.

When the operating system locates the file, it rewinds and loads the file. When loading is complete, it executes the program as a CPU program.

**10.1.5 7/8/9 CARD**

A card with rows 7, 8, and 9 punched in column one separates sections in the job deck. The level is assumed zero unless columns 2 and 3 contain an octal level number punched in Hollerith code. The remaining columns optionally contain comments.

As an example, a deck consisting of a control statement section and a COMPASS source input section would include two 7/8/9 cards. The first terminates the control statements and the second terminates COMPASS input. A 7/8/9 card of level 17 is interpreted by the operating system as a 6/7/8/9 card.

**10.1.6 6/7/8/9 CARD**

A card with rows 6, 7, 8, and 9 punched in column one signals the end of the job deck. Columns 2 through 80 optionally contain comments.

**10.1.7 USER CONTROL STATEMENT (NOS 1 ONLY)**

The user control statement format is:

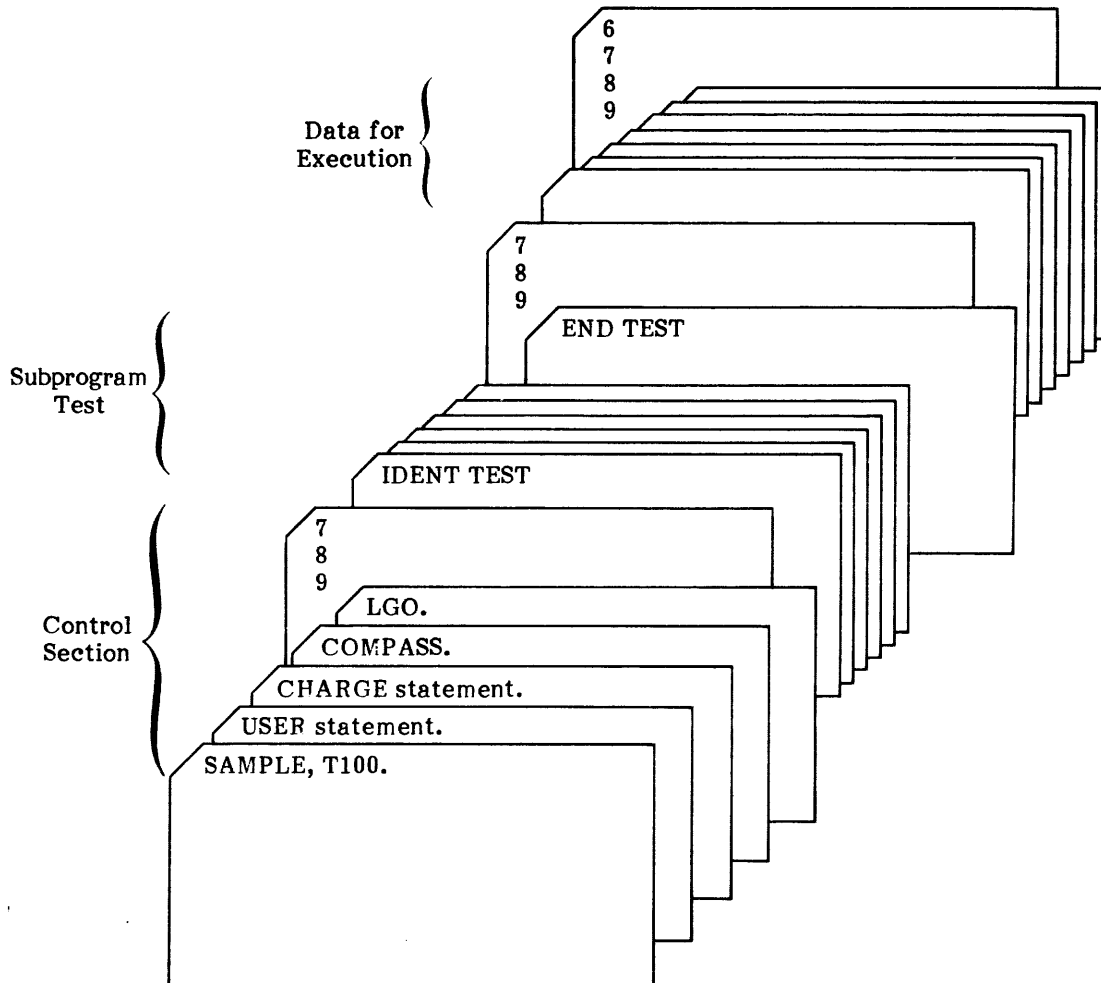
```
USER, usernam, passwd, famname.
```

usern	User number or name
passwd	User password
famname	Name of user permanent file device family name

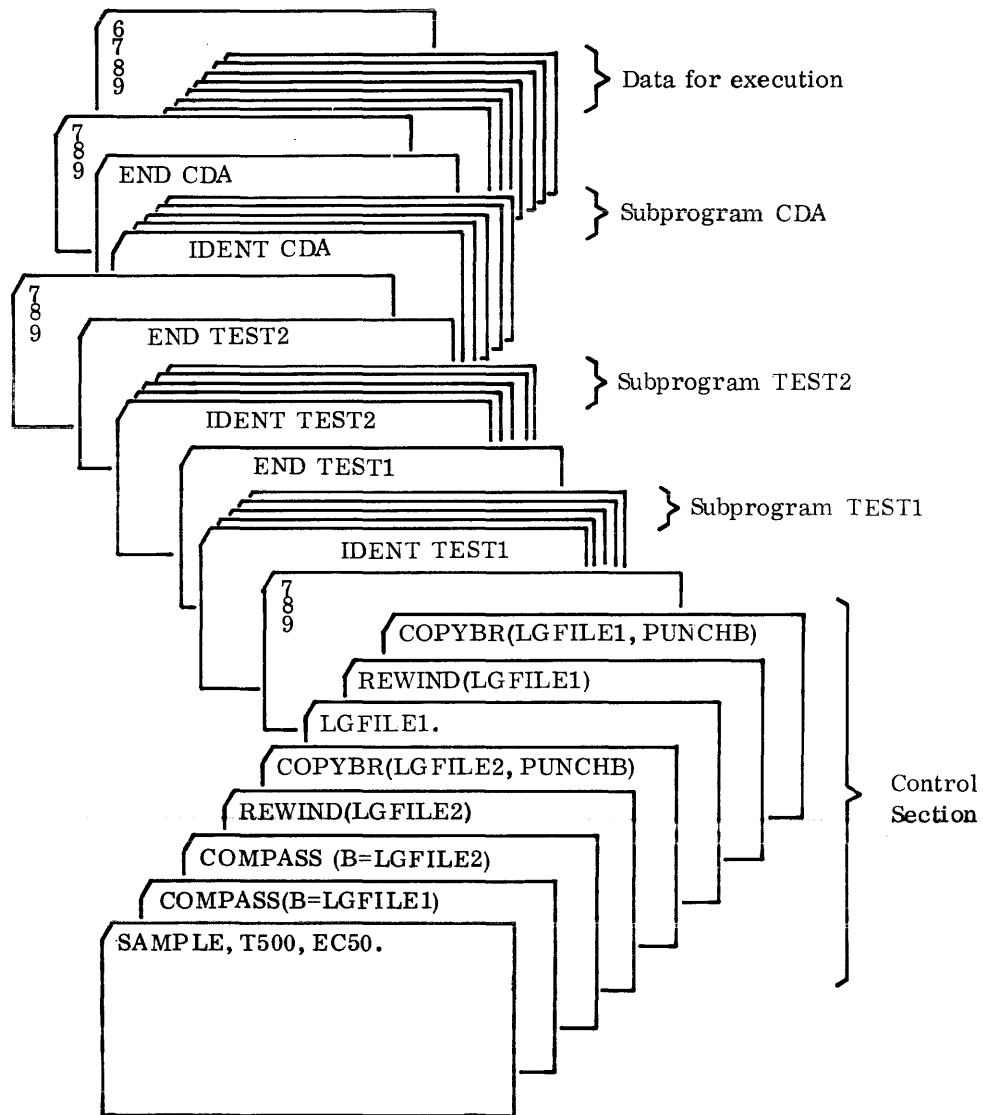
The USER statement, required by NOS 1, follows the job control statement and specifies user access information. The user name is used in system bookkeeping and defines the user's file catalog area. The user can specify a different permanent file catalog during job processing by issuing another USER control statement.

## 10.2 SAMPLE DECKS

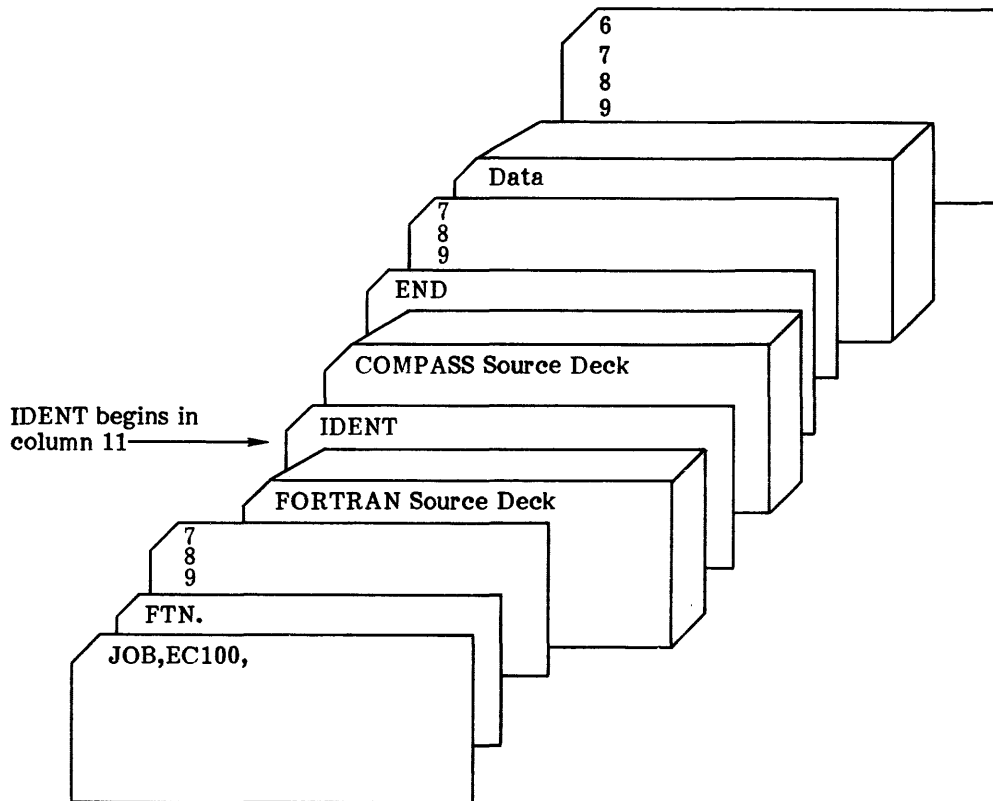
The following job calls for assembly of the source program and execution of the binary object program produced by the assembly. The USER control statement (for NOS 1 only) provides required user access information. COMPASS reads source statements from file INPUT, writes the listing on OUTPUT, and writes a binary object deck on file LGO. Control statement LGO calls for execution of the binary object program, which obtains its data from file INPUT.



In the following job, the COMPASS assembler is called twice. During the first assembly, binary object decks for subprograms TEST1 and TEST2 are written on file LGFILE1. The source decks for these subprograms are in the second section of the INPUT file. During the second assembly, COMPASS writes a binary object deck for subprogram CDA on file LGFILE2. Each assembler run produces a full listing. Following the second assembly, LGFILE2 is repositioned to the beginning of the file. Then, the COPYBR program is called to copy the contents of LGFILE2 to a punch file (PUNCHB). The LGFILE1 statement then calls for the loading and execution of subprograms TEST1 and TEST2 from LGFILE1. Following successful execution of the subprograms, the file is rewound and copied to the punch file, after which the job terminates.



In the following example, the IDENT statement causes FTN to call COMPASS to process the COMPASS source deck. If the COMPASS END statement is not followed by another IDENT statement, then COMPASS returns control to the compiler that called it.



The following sample programs illustrate how to assemble and use a system text overlay.

```

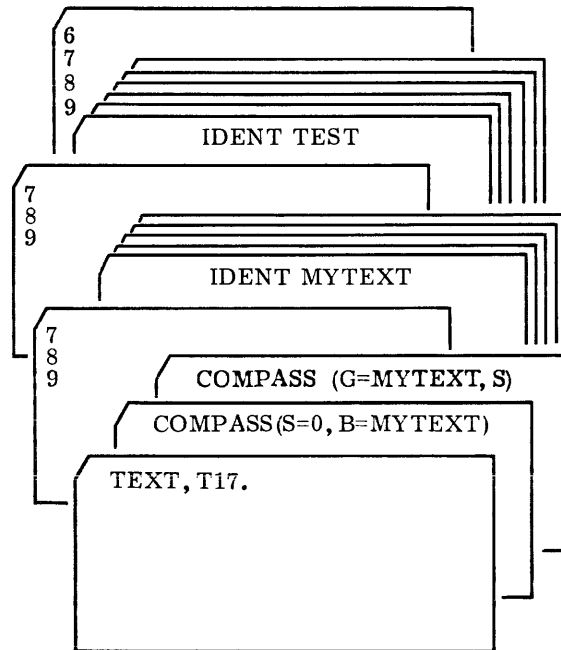
IDENT MYTEXT
STEXT
1 ONE EQU 1 CONSTANT ONE
36 HALF EQU 30 POS CONSTANT
SHIFT MACRO ALPHA,BETA POSITIONING MACRO
IFC NE,$ALPHA$X2$,1
SA2 ALPHA
IFC NE,$BETA$B2$,1
SB2 BETA
LX6 X2,B2
ENDM
END

```



		IDENT	TEST	
		ENTRY	TEST	
		SST		
6110000001	TEST	SB1	ONE	CONSTANT ONE FROM TEXT
5120000004 +		SA2	INBUF	PICK UP VALUE FROM STORAGE
6120000036		SHIFT	X2, HALF	POSITION . WORD IN X6
5160000006 +		SA6	OUTBUF	RETURN NEW WORD TO STORAGE
7160247021		ENDRUN		
	2	INBUF	BSS	2
	1	OUTBUF	BSS	1
		END	TEST	

The deck for this job could be set up as follows:





This section describes assembly listing format. Control of the contents of the listing is described in section 4.11 Listing Control, and in section 10.1.2 COMPASS Control Statement.

## 11.1 PAGE HEADING

Each page of the assembly listing contains a title line and a subtitle line in the following format:

title	COMPASS	Version	date	time	PAGE x
subtitle	sub-sub title	block name	symbol qual		

- title                   Up to 62 characters taken from the first TITLE pseudo instruction or from a TTL pseudo instruction or, in lieu of these, from the IDENT instruction
- date                    Date of assembly
- time                    Time of assembly in hours, minutes and seconds
- PAGE x                 Page number of listing.  Pagination begins with 1 for each END instruction unless the P option is selected on the COMPASS control statement
- subtitle                Up to 62 characters taken from second and subsequent TITLE pseudo instructions or a CTEXT pseudo instruction
- sub-subtitle          Up to 10 characters taken from the most recent EJECT, SPACE, TITLE, or TTL pseudo instruction or the location field of an ES or PS machine instruction.  If the instruction that introduces the new sub-subtitle also causes a page eject, the instruction immediately follows the heading (assuming the C list option is also selected).
- block name             Name of the block in use at beginning of page
- symbol qual            Qualifier in use (see QUAL pseudo instruction)

## 11.2 HEADER INFORMATION

The first page of the assembly listing for each subprogram contains a summary of binary control cards (optional), a list of all the blocks established for the subprogram, and lists of entry points and external symbols.

### 11.2.1 BINARY CONTROL CARD SUMMARY

A binary control card summary in the following format is generated for each IDENT instruction when the

COMPASS control statement or the LIST instruction selects the B list option:

ADDRESS	LENGTH	BINARY CONTROL CARDS
addr <sub>1</sub>	l <sub>1</sub>	binary card <sub>1</sub>
addr <sub>2</sub>	l <sub>2</sub>	binary card <sub>2</sub>
.	.	.
.	.	.
addr <sub>n</sub>	l <sub>n</sub>	binary card <sub>n</sub>
eop	(leop)	END card or blank

binary card<sub>i</sub>      The binary card that caused generation of the binary for the overlay, partial binary, or subprogram. The list includes SEG, SEGMENT, and IDENT instructions.

addr<sub>i</sub>              The central memory or peripheral processor memory origin address for the subprogram, overlay, or partial binary written out as a result of the binary card.

l<sub>i</sub>                  The octal length of the subprogram, overlay or partial binary, in central memory words for a central processor assembly, or in peripheral processor words for a peripheral processor assembly.

eop                  The octal central memory or peripheral processor address for the end of the program unit begun by the previous IDENT.

leop                 The octal length in central memory words of a peripheral assembly; not present in a listing of a central processor assembly.

Examples:

ADDRESS	LENGTH	BINARY CONTROL CARDS.
101	271	IDENT COMPASS, LOVER, CMP
372	5241	SEG
5633	1242	SEG
7075	4145	SEG
13242	5175	SEG
20437	1352	SEG
22011		END COMPASS

ADDRESS	LENGTH	BINARY CONTROL CARDS.
0	7761	IDENT DSD,0
7761	(1462)	

## 11.2.2 BLOCK USAGE SUMMARY

A block usage summary of the following format is generated in the assembly listing under control of the B list option:

BLOCKS	TYPE	ADDRESS	LENGTH
name <sub>1</sub>	t <sub>1</sub>	baddr <sub>1</sub>	bl <sub>1</sub>
name <sub>2</sub>	t <sub>2</sub>	baddr <sub>2</sub>	bl <sub>2</sub>
.	.	.	.
.	.	.	.
name <sub>n</sub>	t <sub>n</sub>	baddr <sub>n</sub>	bl <sub>n</sub>

name<sub>i</sub> Name of the block used in the subprogram, as follows:

- PROGRAM\* For a relocatable assembly, indicates the zero block. For an absolute assembly, the first PROGRAM\* indicates the absolute block, the second indicates the default symbols block.
- ABSOLUTE\* Appears in a relocatable assembly only and indicates the use of an absolute block.
- LITERALS\* Identifies the literals block.
- other Identifies a local, labeled common, or blank common block.

type The type of the block as follows:

- ABSOLUTE All addresses in the block are relative to absolute zero. For an absolute assembly, all blocks are ABSOLUTE.
- +LOCAL Addresses in the block are relative to the origin assigned to block zero. The + is present for an ECS/LCM block.
- +COMMON Addresses in the block are relative to the origin of the common block. The + is present for an ECS/LCM block.

baddr<sub>i</sub> Beginning address of the block according to type.

length<sub>i</sub> Number of words in the block.

Examples:

BLOCKS	TYPE	ADDRESS	LENGTH
PROGRAM*	ABSOLUTE	0	5416
LITERALS*	ABSOLUTE	5416	215
CONTROL	ABSOLUTE	5633	1242
PSEUDO	ABSOLUTE	7075	4145
SUBS	ABSOLUTE	13242	5175
BUFFERS	ABSOLUTE	20437	11140

BLOCKS	TYPE	ADDRESS	LENGTH
ABSOLUTE*	ABSOLUTE	0	62
PROGRAM*	LOCAL	0	35
DATA1	LOCAL	35	1
LCM	+LOCAL	0	5
TABLE	+LOCAL	5	5
TABLE	+COMMON	0	123
TABLE	LOCAL	36	1
TABLE	COMMON	0	1
//	COMMON	0	1000

### 11.2.3 ENTRY POINT LIST

If the subprogram declares entry points, a list of entry point symbols in the following format follows the block usage summary.

ENTRY POINTS.

sym <sub>1</sub> *+addr <sub>1</sub> +block <sub>1</sub>	sym <sub>n+1</sub> *+addr <sub>n+1</sub> +block <sub>n+1</sub>	sym <sub>2n+1</sub> *+addr <sub>2n+1</sub> +block <sub>2n+1</sub>
sym <sub>2</sub> *+addr <sub>2</sub> +block <sub>2</sub>	sym <sub>n+2</sub> *+addr <sub>n+2</sub> +block <sub>n+2</sub>	sym <sub>2n+2</sub> *+addr <sub>2n+2</sub> +block <sub>2n+2</sub>
⋮	⋮	⋮
sym <sub>n</sub> *+addr <sub>n</sub> +block <sub>n</sub>	sym <sub>2n</sub> *+addr <sub>2n</sub> +block <sub>2n</sub>	sym <sub>3n</sub> *+addr <sub>3n</sub> +block <sub>3n</sub>

Where n is one-third the number of entry points. The asterisk to the right of sym<sub>i</sub> is present if sym<sub>i</sub> is a conditional entry point (declared by ENTRYC). The + to the left of addr<sub>i</sub> is present if block<sub>i</sub> is an ECS/LCM block. The + to the right of addr<sub>i</sub> is present if addr is relocatable. Block<sub>i</sub> is blank or a common block name surrounded by slashes.

If the symbol is undefined, addr<sub>i</sub> is \*\*\*\*\*.

Example:

ENTRY POINTS.

SNAP1	1345+	CALL	72+	REFORDER	2375+
SNAP2	1352+	GOTO	156+	RPF	2461+
SNAP3	1357+	IF	224+	RPH	2463+
JUMPVEC *	0+/JUMPVEC/	LABEL	372+	LCM	0+
BEGIN	0+	READ	435+	LCMR	** 100+/LCMR/
BYTESIZ	6	RECORD	24+/DATA/		

### 11.2.4 EXTERNAL SYMBOL LIST

If external symbol references are declared in the subprogram, a list of the following format follows the list of entry point symbols:

EXTERNAL SYMBOLS.

sym <sub>1</sub>	sym <sub>n+1</sub>	sym <sub>2n+1</sub>	sym <sub>3n+1</sub>	⋯	sym <sub>7n+1</sub>
sym <sub>2</sub>	sym <sub>n+2</sub>				

```

sym3      symn+3
  ⋮        ⋮
symn      sym2n

```

Where n is one-eighth the number of external symbols. If a symbol is a weak external it is followed by an asterisk.

Example.

**EXTERNAL SYMBOLS.**

```
FRMSG  CONEXIT  XDFCPI  SYMBOL  CCGOTO  CPC
```

### 11.3 OCTAL AND SOURCE STATEMENT LISTING

The contents of the octal and source statement listing depends on the options selected.

The list is 130 characters wide with fields assigned as shown in figure 11-1.

Title Line				
Subtitle Line				
Error Flags	Location Addresses	Octal Code	Source Lines	Sequence

Figure 11-1. Format of Octal and Source Statement Listing

**Error Flags** Error flags indicating that errors of the type indicated have been detected on the source line or in a subsequent statement that is not listed. These flags are described more fully under Error Directory. Lines containing errors are always listed.

**Location Addresses** The value of the location counter with leading zeros suppressed. If no code is generated or no location symbol is defined by the statement, this field is blank. If at the time the value is assigned, the value of the location counter differs from the value of the origin counter, an L precedes the address.

**Octal Code** The actual code generated by this statement. Depending on options selected, the listing shows just the first word or all words generated for data generation instructions. The field does not include NO instructions ( $46000_8$ ) packed for a force upper or zeros packed for a completed parcel on a VFD. A 24-bit PPU instruction is shown two words of data per line.

If the word contains an address, the octal code is flagged as follows:

- Negative relocatable address
- + Positive relocatable address
- C Common relocatable address
- X External address

For a statement that does not generate code, this field is normally blank. Exceptions are as follows:

For a LIT instruction the field contains the address of the first word of the literals generated.

For a COL instruction, the field contains the new beginning-of-comments column number.

For a symbol defined through SET, MAX, MIN, EQU, =, or MICCNT, this field contains the octal value of the symbol right justified with leading zeros suppressed.

For an instruction resulting in a change of base, the notation  $b_1 \rightarrow b_2$  is right justified in the field.  $b_1$  indicates the old base and  $b_2$  indicates the new base.

For an instruction resulting in a change of code conversion, the notation  $c_1 \rightarrow c_2$  is right justified in the field.  $c_1$  indicates the old code and  $c_2$  indicates the new code.

For a DUP instruction, the field contains the repeat count.

For a BSS or BSSZ instruction, the field contains the octal value of the word count right justified with leading zeros suppressed. If the word count is zero the field is blank.

For a DECMIC or OCTMIC instruction, the field contains the octal value of the expression right justified with leading zeros suppressed.





Examples:

**CONTENT OF LITERALS BLOCK.**

010121	17455773753000000000	0+.>>X
010122	16650000000000000000	N+
010123	15052323010705553636	MESSAGE 33
010124	55040503111501145522	DECIMAL R
010125	05212511220504570000	EQUIRFD.
010126	55220521251122050400	REQUIRFD
010127	00000000000000000000	
010130	20221707220115550102	PROGRAM AB
010131	17222457000000000000	ORT.

**CONTENT OF LITERALS BLOCK.**

7315	0034	1
7316	7070	↑↑
7317	0007	G
7320	0000	
7321	5501	A
7322	0000	
7323	0506	FF
7324	1411	LI
7325	2405	TF
7326	2201	RA
7327	1423	LS

**11.5 DEFAULT SYMBOLS**

When the D list option is selected, a list of default symbols immediately precedes the literals block.

Example:

**DEFAULT SYMBOLS DEFINED BY COMPASS**

000000 X	MSG=
005461	TAG1
005462	TAG2
005463	ARC
005464	SYM

**11.6 ASSEMBLER STATISTICS**

Assembler statistics are printed at the end of the octal and source statement listing or, if the D list option is selected, following the default symbols. Information includes the following:

- Amount of storage used (octal)
- Number of source statements
- Number of symbols defined
- Number of invented symbols
- Number of symbol references
- CPU type in which COMPASS executed and assembly time
- Number of errors encountered during assembly
- Number of lost references, that is, references to symbols that have been omitted from the symbolic reference table

## 11.7 ERROR DIRECTORY

The assembly listing includes an error directory if any errors are detected during assembly. The directory begins a new page identified with the subtitle ERROR DIRECTORY. Each type of error that occurred is called out with a two-line message of the following format:

```
x  TYPE ERROR          description
      OCCURRED ON PAGES      P1, P2, P3, ... Pn
```

Types and descriptions are given in Tables 11-1 and 11-2. Errors flagged with an alphabetic character are fatal. A fatal error causes suppression of binary output. Nonfatal warning flags are numeric; they are informative only.

TABLE 11-1. FATAL ERRORS

Type	Message	Significance	Action
A	ADDRESS FIELD BAD.	<p>An error exists in a variable subfield entry. The following is a list of possible errors:</p> <p>The CODE character is not A, D, E, I, O, or *.</p> <p>The symbol or name is greater than 8 characters.</p> <p>The expression does not reduce to one external term.</p> <p>The relocatable terms do not cancel properly.</p> <p>The instruction requires an absolute expression.</p> <p>The instruction disallows register designators.</p> <p>A data error; 8 or 9 is encountered in octal data and the modifier is not S, P, O, E, D, or B.</p> <p>No data is found in the variable field of a LIT instruction.</p> <p>No symbol is following an =S, =X, or =Y prefix.</p> <p>The relative jump is out of range (-31&gt;r&gt;31) on a PPU instruction.</p> <p>The BASE character is not O, M, D, or *.</p>	<p>Refer to the manual for the correct address field format for the operation code specified.</p>

TABLE 11-1. FATAL ERRORS (Contd)

Type	Message	Significance	Action
A	ADDRESS FIELD BAD. (Contd)	<p>A register is illegal in a CON instruction.</p> <p>A synonymous instruction for OPSYN or CPSYN cannot be located.</p> <p>The micro count is less than zero or greater than ten.</p> <p>The NOLABEL character is not I.</p> <p>A negative relocation is specified on ORG or ORGC.</p> <p>The POS value is less than 0 or greater than word size.</p> <p>The OPDEF reference is erroneous.</p> <p>No comma is following the DIS word count.</p> <p>An illegal entry is in the variable field of IDENT.</p>	
D	DOUBLY DEFINED SYMBOL. THE FIRST DEFINITION HOLDS.	A symbol has been previously defined or declared external.	Rename the duplicate symbol in the program.
E	ECHO, DUP, RMT, OR MACRO ILLEGALLY NESTED.	The definition of ECHO, DUP, RMT, or MACRO is not entirely within the next outer definition.	Correct the program.
F	NUMBER OF ENTRIES EXCEEDS PERMISSIBLE AMOUNT.	<p>One of the following error conditions exists:</p> <p>LIT generates more than 100 words.</p> <p>Data is missing or erroneous on XTEXT file.</p> <p>More than 63 formal parameters and local names are in a macro definition.</p> <p>There are more than 255 blocks.</p> <p>There are more than 511 external symbols.</p>	Correct error condition and rerun the job.

TABLE 11-1. FATAL ERRORS (Contd)

Type	Message	Significance	Action
L	LOCATION FIELD BAD.	The required location field entry is erroneous. The format two macro definition has no substitutable parameters.	Correct the location field entry.
N	NEGATIVE RELOCATION ON ENTRY POINT.	An entry point may not be negatively relocated.	Change to use positive or absolute relocation for entry points. Rerun job.
O	OPERATION FIELD BAD.	One of the following error conditions exists in the operation field:  The instruction is unrecognizable.  The instruction is out of sequence, such as ABS or PPU not in the first statement group.  The instruction is illegal for binary mode.  The relational mnemonic on the IF statement is erroneous.  AIDTEXT has determined that the instruction has changed or is not valid for the models 810, 815, 825, 830, 835, 845, and 855.	Correct the operation field.  Replace instruction.
P	CONSULT LISTING FOR REASON BEHIND P-ERROR	A user-generated error flag from an ERR or ERRxx instruction has been encountered.	Action to be taken depends upon source of error.
R	DATA ORIGIN OUTSIDE BLOCK OR IN BLANK COMMON.	An attempt was made to set data into blank common or beyond block limits.	Use labeled common or increase block size and rerun job.
U	UNDEFINED SYMBOL. VALUE ASSUMED 0.	There is a reference to a symbol that is not defined; for example, an IF statement line count, a DIS word count, an unrecognizable attribute on an IF statement, or an undefined qualifier.	Define the symbol.
V	BIT COUNT ERROR ON VFD (MUST BE 0 COUNT 60).	The VFD field size is erroneous.	Correct the size of the VFD field.

TABLE 11-2. INFORMATIVE MESSAGES

Type	Message	Significance	Action
1	LOCATION SYMBOL BAD. SYMBOL NOT DEFINED.	The location field is erroneous. The instruction does not require an entry.	Define or eliminate the symbol in the location field.
2	ADDRESS ERROR ON SYMBOL DEFINITION.	The variable field entry is erroneous. The location field symbol is not defined.	Correct the symbol definition.
3	DUPLICATE MACRO DEFINITION. NEW ONE OVERRIDES.	The macro, opdef, or synonymous operation redefines the operation code.	Rename the duplicate macro name.
4	BAD FORMAL PARAMETER NAME IGNORED.	The macro or ECHO formal parameter name is repeated or illegal.	Correct the formal parameter name.
5	CPU OPERATION SYNTAX INCORRECTLY SPECIFIED.	The OPDEF, CPOP, CPSYN, or PURGDEF specifies an illegal syntax.	Correct the syntax of the pseudo instruction.
6	LOCATION FIELD MEANINGLESS.	The entry in the location field is erroneous; it is ignored.	Correct the location field.
7	ADDRESS VALUE EXCEEDS FIELD SIZE, RESULT TRUNCATED.	The value of the address is erroneous; one of the following conditions exists:  The value of the expression exceeds the size of the destination field.  The BSS address expression value is negative.  The MICRO starting character position or character count is negative.	Check the possible values of the variable subfield.
8	MISSING OR EXTRA ADDRESS SUBFIELD.	The variable subfield entry is missing or superfluous.	Correct the variable subfield.
9	MICRO SUBSTITUTION ERROR. NO SUBSTITUTION.	The micro reference is unrecognizable.	Correct the micro reference.

## 11.8 SYMBOLIC REFERENCE TABLE

The assembler generates a symbolic reference table (figure 11-2) if the L list option is on at the end of assembly. The table is not complete if the option was turned off at any time during the assembly. The table lists symbols according to the qualifier, if any, under which they were defined. The global symbols are listed first. A new heading of the following form introduces each new list of qualified symbols.

SYMBOL QUALIFIER = qualifier

The qualifiers are in the order declared in the subprogram. Symbols are listed alphabetically.

When symbol references are lost because table space has been exceeded, the subtitle line includes notification in the form n LOST REFERENCES.

Format 1 reflects the XREF P effect; P is the default for the XREF pseudo instruction. Formats 2 and 3 reflect the effects of XREF B and XREF A, respectively.

Title Line												
SYMBOLIC REFERENCE TABLE.												
Format 1 (XREF P):												
symbol	value	block	page/line	flag	page/line	flag	page/line	flag	page/line	flag	page/line	flag
Format 2 (XREF B):												
symbol	value	block	page/line	flag	address,	page/line	flag	address,	page/line	flag		
Format 3 (XREF A):												
symbol	value	block	address,	address,	address,	address,	address,	address,	address,			

Figure 11-2. Format of Symbolic Reference Table

- symbol            Alphabetical list of symbols defined under the qualifier.
- value            Absolute value of the symbol or the address assigned to this symbol relative to the block named.
- block            If the symbol was defined by the SST pseudo instruction, block is the system text file or overlay name. Otherwise, this field is blank in an absolute assembly or, in a relocatable assembly, it contains the name of the block containing the symbol.
- page/line        From left to right and from top to bottom, a list of indices sequenced according to page number. Each index points to a statement containing references to the symbol or defining the symbol. Present when XREF B or P is in effect.
- address          The location counter address of the instruction containing the reference. Present when XREF A or B is in effect.

flag

Identifies page/line index to a statement that defines the symbol or uses it in an IF statement as follows:

- D Definition statement; EQU, =, SET, MAX, MIN, or MICCNT
- E ENTRY or ENTRYC pseudo instruction
- F Symbol used in conditional test
- I Symbol used for indirect storage (applies only to PPU or PERIPH assemblies)
- L Symbol used in location field of the statement
- S Symbol used for storage
- X EXT pseudo instruction

When XREF A is in effect, the table does not include the flags.

Example:

COMPASS 3.71210 - CYBER 70/ COMPREHENSIVE ASSEMBLER. SYMBOLIC REFERENCE TABLE.		COMPASS 3.71210 DERIV		8/26/72	16.25.44.	PAGE	551
SNTMP	5115	72/12 L	74/51 S	74/53	76/77 S	76/74	
SNUNB	5421	73/40	74/63	74/12	74/	74/42	75/44 75/53 78/14 L
SNUNB1	5416	74/48 L	77/53	78/56			
SNMLIN	5423	73/28	77/41	74/65	74/92	76/23	79/18 L 79/41
SNMLIN1	5425	79/14 L	79/16				
SNMLIN2	5427	79/13	79/17 L				
SNX	5134	72/16 L	77/39 S	74/10	77/16	77/74	
		72/32 S	72/42 S	74/16	77/35	77/34	
SYMBOL QUALIFIER = DATA							
AF	6675	115/39 L	115/46	121/37	131/52	132/19	132/37
CCS	7326	132/44	137/37	133/18	133/31	133/44	134/22 135/44 L 136/6
CCS1	7332	135/52	135/54 L				
CCS2	7323	135/38 L	136/21				
CSA	7254	117/32	121/20	137/21 L			
CSC	7297	117/26	121/17	137/16 L			
CSH	7250	117/20	121/14	137/42 L			
CSL	7261	117/17	121/11	137/42 L			
CSR	7266	117/11	121/6	137/57 L			
CSZ	7251	117/4	121/65	137/20 L			
CCS	7222	117/19	117/12	117/14	117/21	117/27	117/32 131/33 L
CCS1	7225	131/42 L	131/46				
DL	6674	115/38 L	125/35	134/22			
DO	6673	115/37 L	115/46 S	115/35	126/35	134/19	
DV	6653	115/16 L	125/25 S	122/41	127/36	124/27	132/25
EF	6651	115/21 L	122/21	125/11	126/33	127/25 S	
ERR	6715	116/35 L	121/35	122/24	123/13	126/25	127/27 131/51 132/16
		116/53	121/51	122/17	123/39	126/11	128/19 132/10
		118/57	122/11	123/16	125/50	126/44	126/41 132/11
		115/22 L					
ES	6662	122/22	128/04 L				
ESC	7141						
EV	6663	115/23 L	122/43	123/27 S	123/42		
FC	6660	115/19 L	127/35 S	122/49			
FW	6676	115/40 L	135/63	135/17			
GCS	7271	132/49	137/09	133/21	133/36	137/47	134/15 134/19 L
GCS1	7275	134/34 L	134/37				
GCS2	7277	134/32	134/39 L				
GCS3	7304	134/41 L	134/44				
GCS4	7303	134/40	134/45 L				
GCS5	7304	134/48 L	134/51				
GCS6	7306	134/46	134/53 L				
GCS7	7307	134/53	134/55 L				
GCS8	7316	135/62	135/11	135/15 L			
INT	7135	125/48	126/55 L				
LRS	6740	117/15	117/24	117/30	119/06 L		
NCS	7233	121/06	121/09	121/12	121/15	121/18	121/21 132/05 L
			132/2				



---

The common common decks are a set of COMPASS subroutines which are powerful tools for use by COMPASS programmers. The common common decks perform functions such as:

Data conversion

Dynamic table management

Saving/restoring registers

Providing an input/output interface at the CIO and FET level

All of the common common decks run under NOS and NOS/BE; a subset of them run under SCOPE 2. Table 12-1 shows each deck name, relocatable program name, entry point names, and the decks supported under SCOPE 2.

## 12.1 ACCESS TO THE COMMON COMMON DECKS

The common common decks are available in two forms:

As relocatable subroutines

In source code form as a set of common decks

Both methods of accessing the common common decks are illustrated in the sample program in appendix D.

All the common common decks except the table management decks COMCMTM and COMCMTP are available as relocatable subroutines that reside on the system library SYSLIB. In this form the common common decks are easy to use; relocatable COMPASS programs need only include external references to entry point names in the common common decks. These external references are satisfied from SYSLIB at load time. (The CYBER loader searches SYSLIB by default when satisfying external references, but the SCOPE 2 Loader does not; under SCOPE 2, SYSLIB must be explicitly included in the library set.)

Occasionally, the programmer may need to access the source code of the common common decks. That source code resides on the COMCPL old program library as a set of common decks (see the Update reference manual). The source code of these common decks can be made available to a COMPASS program in three ways:

Update-based procedures can use the COMCPL old program library as a secondary old program library (see the Update reference manual). The decks are called just as one would call a common deck from one's own old program library.

Modify-based products can convert the COMCPL old program library to an OPL via the UPMOD statement (see the NOS reference manual); the OPL is then used as the source for the common common decks.

The programmer can use the COMPASS XTEXT pseudo-instruction in the program to obtain the source code from either an old program library or an OPL (see the X file option of the COMPASS control statement).

The system texts required to assemble the common common decks residing on the COMCPL old program library are IPTEXT and CPUTEXT. These texts can be made available to the program via the S parameter on the COMPASS control statement.

TABLE 12-1. SUMMARY OF COMMON COMMON DECKS

Common Common Deck Name	Relocatable Program Name	Entry Points	Available Under SCOPE 2
COMCARG	CPU.ARG	ARG=	Yes
COMCCDD	CPU.CDD	CDD=	Yes
COMCCFD	CPU.CFD	CFD=	Yes
COMCCIO	CPU.CIO	CIO=	No
COMCCOD	CPU.COD	COD=	Yes
COMCCPT	CPU.CPT	CPT=	Yes
COMCDXB	CPU.DXB	DXB=	Yes
COMCMNS	CPU.MNS	MNS=	Yes
COMCMOS	CPU.MOS	MOS=	Yes
COMCMTM			Yes
COMCMTP			Yes
COMCMVE	CPU.MVE	MVE=	Yes
COMCRDC	CPU.RDC	RDC=	No
COMCRDH	CPU.RDH	RDH=	No
COMCRDO	CPU.RDO	RDO=	No
COMCRDS	CPU.RDS	RDS=	No
COMCRDW	CPU.RDW	RDW= RDX= LCB=	No
COMCRSR	CPU.RSR	RSR=	Yes
COMCSFN	CPU.SFN	SFN=	Yes
COMCSRT	CPU.SRT	SRT=	Yes
COMCSST	CPU.SST	SST=	Yes
COMCSTF	CPU.STF	STF=	No
COMCSVR	CPU.SVR	SVR=	Yes
COMCSYS	CPU.SYS	SYS= RCL= WNB= MSG=	No
COMCUPC	CPU.UPC	UPC=	Yes
COMCWOD	CPU.WOD	WOD=	Yes
COMCWTC	CPU.WTC	WTC=	No
COMCWTH	CPU.WTH	WTH=	No
COMCWTO	CPU.WTO	WTO=	No
COMCWTS	CPU.WTS	WTS=	No
COMCWTW	CPU.WTW	WTW= WTX= DCB=	No
COMCXJR	CPU.XJR	XJR=	No
COMCZTB	CPU.ZTB	ZTB=	Yes

## 12.2 DESCRIPTION OF THE COMMON COMMON DECKS

A detailed external reference description of each common common deck follows. The decks are described in alphabetical order. Each description lists entry and exit conditions, registers used, and routines explicitly called.

The following rules apply to the use of all common common decks:

Any input/output buffers, string buffers, exchange package save areas, and so forth, to be used by any of the common common decks should not be located with the last 10<sub>8</sub> words of the field length. Some fetch loops, move loops, and so forth, may encounter a hardware fault (out of range address) if the above restriction is not adhered to.

Registers that are not used by the common common decks are not modified.

Entry and exit conditions are only those listed in the descriptions below.

### 12.2.1 COMCARG – PROCESS ARGUMENTS

COMCARG processes a list of arguments (in the format generated by COMCUPC) by the use of an equivalence table. The equivalence table must be terminated by a word of all zeros and must be in the following format:

12/op,18/asv,12/st,18/addr

op	One or two character keywords (left justified, zero filled)
asv	Address of assumed value
st	Status
addr	Address where argument is placed

This format is generated by the COMPASS VFD pseudo instruction. ARG= is the only entry point for COMCARG.

Entry conditions:

(B1)	1
(B4)	Argument count
(A4)	Address of first argument
(X4)	First argument
(B5)	Address of equivalence table

Exit conditions:

(X1) ≠ 0

1	Option not found in table
2	Single argument equivalenced
3	Illegal re-entry of argument

Registers used:

A2, A3, A4, A7  
B2, B3, B4  
X0, X1, X2, X3, X4, X6, X7

The following conditions apply to the use of COMCARG:

If a keyword=value form is found in the argument list, addr is set to the upper 42 bits of the argument value (in bits 59-18) and the lower 18 bits of asv (in bits 17-0).

If only a keyword is found in the argument list, addr is set to the full 60 bits of asv.

If asv = 0, the argument cannot be equivalenced.

If status=4000<sub>g</sub>, a zero value is retained as a display code zero. Otherwise, a value of zero (full word) is stored at addr.

If asv=addr, only one entry of that argument is allowed and op is set to -0.

### 12.2.2 COMCCDD - CONVERT INTEGER CONSTANT TO DECIMAL DISPLAY CODE

COMCCDD converts an integer constant to decimal display code. Up to ten digits are converted with leading zero suppression. The converted integer contains space fill. One register contains the display code right justified; another register contains it left justified. CDD= is the only entry point for COMCCDD.

Entry conditions:

(B1) 1  
(X1) Number to be converted

Exit conditions:

(B2) 6\*(count of digits converted)  
(X4) Conversion left justified  
(X6) Conversion right justified

Registers used:

A2, A3, A4  
B2, B3, B4  
X1, X2, X3, X4, X6, X7

### 12.2.3 COMCCFD - CONVERT CONSTANT TO F10.3 FORMAT

COMCCFD converts a 30 bit integer to display code in FORTRAN F10.3 format. The value returned is equal to the input value divided by 1000. The result is returned in two forms: left justified and right justified. Leading zeros in the integer portion of the result are suppressed. If the 30-bit input value exceeds 999999.999(7346544777<sub>g</sub>), the result is◆◆◆◆◆◆◆◆◆◆. An input value greater than 30 bits is truncated to the lower 30 bits. CFD= is the only entry point for COMCCFD.

Entry conditions:

(B1) 1  
(X1) Integer to be converted

Exit conditions:

(B3) - (number of blank fill bits in result)  
(X4) Conversion left justified  
(X6) Conversion right justified

Registers used:

A1, A2, A3, A4  
B2, B3, B4, B5  
X1, X2, X3, X4, X6, X7

## 12.2.4 COMCCIO - PROCESS I/O OPERATION

COMCCIO performs input/output operations via the peripheral processor program CIO. An operation is performed when the buffer is not busy. If the file-status-word is zero, the operation is not processed and IN and OUT are set to FIRST. CIO= is the only entry point for COMCCIO.

### Entry conditions:

- (X2) 24/unused, 18/skip count to CIO, 18/FET address for file
- (X7) Function code; if < 0, X7 is the complement of the request and auto recall is requested

### Exit conditions:

- (X2) FET address
- (X7) 0

### If ERP\$ is defined:

- (X2) FET address
- (X7) FET error code:
  - 0 No error, operation performed, normal exit
  - other Error code from FET; operation not performed, exit to ERP\$

### If ERP1\$ is defined:

- (X2) FET address
- (X7) FET error code:
  - 0 No error, operation performed, normal exit
  - other Error code from FET; operation not performed, normal exit

### Registers used:

- A1, A6, A7
- X1, X2, X6, X7

## 12.2.5 COMCCOD – CONVERT CONSTANT TO OCTAL DISPLAY CODE

COMCCOD converts an integer constant to octal display code with leading zero suppression. Up to ten digits can be converted. The converted integer contains space fill. One register contains the display code right justified, another register contains it left justified. COD= is the only entry point for COMCCOD.

### Entry conditions:

- (B1) 1
- (X1) Number to be converted

### Exit conditions:

- (B2) 6\*(count of digits converted)
- (X4) Conversion left justified
- (X6) Conversion right justified

### Registers used:

- A4
- B2, B3, B4
- X1, X2, X3, X4, X6, X7

## 12.2.6 COMCCPT – EXTRACT COMMENTS FIELD FROM PREFIX TABLE

COMCCPT copies the comments field of a prefix (7700g) table to a working storage area. Either the old or new forms of the prefix table can be used. COMCCPT differentiates between the forms by checking word FWA+3 of the table to see if it looks like a time-of-day word. The copy terminates on end-of-table, zero byte, or COPYRIGHT. The working storage area is terminated by a zero word. CPT= is the only entry point for COMCCPT.

### Entry conditions:

(A1) Prefix table address  
(A6) Address of working storage - 1  
(B1) 1  
(X1) Control word

### Registers used:

A2, A3, A4, A6  
B3, B4  
X1, X2, X3, X4, X6

## 12.2.7 COMCDXB – CONVERT DISPLAY CODE TO BINARY

COMCDXB converts a string of display code digits up to one word in length (left-justified and zero-filled) into internal integer format. Either a base 10 or a base 8 string of digits can be converted as specified in the call. This specification, however, is overridden if an explicit B (octal) or D (decimal) is the last character of the value to be converted. DXB= is the only entry point for COMCDXB.

The assembly option DXB1\$ controls the processing of an 8 or 9 when octal is specified for the display code value and no explicit B or D appears in the value. If DXB1\$ is not defined, an error occurs. If DXB1\$ is defined, the value is considered to be decimal.

### Entry conditions:

(B1) 1  
(B7) Base; if >0, decimal base; if 0, octal base.  
(X5) Word to be converted (left justified, zero filled)

### Exit conditions:

(X6) Converted digits  
(X4) Error code:  
0 No error  
other Error in assembly

### Registers used:

B2, B3, B4, B5  
X0, X1, X2, X3, X4, X5, X6, X7

The presence of one or more of the following always causes an error:

- A non-digit in the word to be converted
- A character after the post radix
- An 8 or 9 with the post radix equal to B

## 12.2.8 COMCMNS – MOVE NON-OVERLAPPING BIT STRING

COMCMNS moves a specified source string from one location to another in central memory. The only bits disturbed in the destination field are those extracted to accept the source string. The destination field must not overlap the source field in any way; results are undefined if overlapping occurs; COMCMOS can be used for overlapping moves. MNS= is the only entry point for COMCMNS.

### Entry conditions:

(B1) 1  
(B2) Source first bit (0, 1, . . . , 59)  
(B4) Destination first bit (0, 1, . . . , 59)  
(X0) Number of bits to move  
(X2) Source first word address  
(X4) Destination first word address

### Exit conditions:

(B1) 1  
(B2) Source next bit (0, 1, . . . , 59)  
(B4) Destination next bit (0, 1, . . . , 59)  
(X2) Source next word address  
(X4) Destination next word address

### Registers used:

A1, A2, A3, A5, A6  
B1, B2, B3, B4, B5, B6  
X0, X1, X2, X3, X4, X5, X6, X7

## 12.2.9 COMCMOS – MOVE OVERLAPPING BIT STRING

COMCMOS moves a specified source string from one location to another in central memory. The only bits disturbed in the destination field are those extracted to accept the source string. COMCMOS allows the user to move strings where the destination field overlaps (lies partly or completely within) the source field. If the move is not an overlap move, COMCMOS calls the faster common common deck COMCMNS to do the move. For this reason, COMCMNS should always be called whenever COMCMOS is. MOS= is the only entry point for COMCMOS.

### Entry conditions:

(B1) 1  
(B2) Source first bit (0, 1, . . . , 59)  
(B4) Destination first bit (0, 1, . . . , 59)  
(X0) Number of bits to move  
(X2) Source first word address  
(X4) Destination first word address

### Exit conditions:

(B1) 1  
(B2) Source next bit (0, 1, . . . , 59)  
(B4) Destination next bit (0, 1, . . . , 59)  
(X2) Source next word address  
(X4) Destination next word address

**Registers used:**

A1, A2, A3, A5, A6, A7  
B1, B2, B3, B4, B5, B6  
X0, X1, X2, X3, X4, X5, X6, X7

**Calls:**

MNS=

**12.2.10 COMCMTM – MANAGED TABLE MACROS**

COMCMTM contains four macros, ADDWRD, ALLOC, SEARCH, and TABLE, for generation, allocation, and processing of managed tables. COMCMTM is intended to be used with COMCMTP.

**ADDWRD - ADD WORD TO TABLE**

ADDWRD adds a word to a managed table. ADDWRD calls ADW and uses A0 and X1.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ADDWRD	table, reg

**table** Table number

**reg** Register name or expression for word to be added

**ALLOC - ALLOCATE TABLE SPACE**

ALLOC allocates table space. ALLOC calls ATS and uses A0 and X1.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ALLOC	table, words

**table** Table number

**words** Word count (+ or -) to be added

**SEARCH - SEARCH MANAGED TABLE**

SEARCH searches for a specified entry. SEARCH calls EQS or MES and uses A0, B7, and X6.

**Format:**

LOCATION	OPERATION	VARIABLE SUBFIELDS
	SEARCH	tname, entry, mask

**tname** Table name

**entry** Entry to be searched for

**mask** Search mask in X0; if not present, defaults to all bits.



## TABLE - GENERATED MANAGED TABLE

TABLE generates a managed table.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	TABLE	tname, count, equiv

tname Table name

count Word count per entry (1 if not specified)

equiv Equivalent table name; allows certain tables to be used by different processors

After the table is generated:

F.tname is the name of the word containing the table FWA.

L.tname is the name of the word containing the table length.

C.tname is the word count per entry.

### 12.2.11 COMCMTP — MANAGED TABLE PROCESSORS

COMCMTP contains the following routines for processing managed tables:

ADW Adds a word to the table.

AMU Returns the total memory used by the tables.

ATS Allocates table space.

EQS Searches table for equal entries.

MES Searches a table for equal entries using a mask.

MTD Moves the table down.

MTU Moves the table up.

Macros for calling these routines and for table generation are contained in COMCMTM.

The managed table processors allow the partitioning of central memory into variable regions called tables. These tables are referenced by pointers that indicate the first word address of the table and the table length. Memory is allocated to each table as it is required; the user can delete space from the tables. Each table is allowed at least one word of expansion space to allow a dummy word between each table, thus ensuring efficient search methods.

The caller of the table processors is expected to provide certain constants for use by the processors. Other data is provided by COMCMTM.

Data provided by the caller:

MEML Lowest address of managed memory

TOV Address of the table overflow processor

**Data provided by COMCMTM:**

**NTAB**    Number of managed tables  
**FTAB**    Start of table addresses  
**LTAB**    Start of table lengths  
**F.tnam**   Address pointer for table tnam  
**L.tnam**   Length pointer for table tnam

**Data dynamically changeable:**

**TN**        Number of managed tables. Set to **NTAB** by **COMCMTM**. **TN** must be less than **NTAB** during use.  
**TO**        Table overflow processor. Set to **TOV** by **COMCMTM**.  
**LM**        Low memory limit. Set to **MEML** by **COMCMTM**. If this value is increased, **MTU** should be called to allow room for change.  
**F.TEND**   High memory limit. **F.TEND** must be initialized by the user. If this value is decreased, **MTD** should be called to allow room for change.  
**TOVT**     **TOV** threshold. If the word is defined, it should contain the threshold for calling **TOV**; **ATS** calls **TOV** when the tables must be moved and less than **TOVT** free words remain. If **TOVT** is not defined, an effective value of zero is used.

**ADW - ADD WORD TO TABLE**

**ADW** adds a word to a managed table.

**Entry conditions:**

**(A0)**     Table number  
**(X1)**     Word to be added

**Exit conditions:**

**(A6)**     Address of added word  
**(X1)**     Added word  
**(X2)**     **FWA** of table  
**(X3)**     Length of table  
**(X6)**     Added word

**Registers used:**

**A1, A2, A3, A4, A6, A7**  
**X1, X2, X3, X4, X6, X7**

**Calls:**

**ATS**

**AMU - ACCUMULATE MEMORY USED**

**AMU** returns the amount of memory used by the managed tables or the current length, whichever is the largest. The variable **MU** is set to this value.

Exit conditions:

MU MAX(memory used, current assigned length)

Registers used:

A1, A2, A6  
B2  
X1, X2, X3, X6

ATS - ALLOCATE TABLE SPACE

ATS allocates table space. The table length can be increased or decreased as specified.

Entry conditions:

(A0) Table number  
(X1) Change (+ or -) to the table size

Exit conditions:

(X1) Change made to the table size  
(X2) FWA of table  
(X3) New length of table  
(X7) Less than 0 if tables moved

Registers used if tables are not moved.

A2, A3, A4, A6  
X2, X3, X4, X6, X7

Registers used if tables are moved:

A1, A2, A3, A4, A6, A7  
B2, B3, B4, B5, B6, B7  
X0, X1, X2, X3, X4, X5, X6, X7

Registers restored:

B2, B3, B4, B5, B6, B7 (except -0 restored as +0)  
X0, X1, X5

Calls:

AMU, MVE=, TOV

TOV, the user provided table overflow processor, is described below.

Entry conditions:

(B1) 1  
(B5) Complement of number of words required  
(B6) Return address to continue processing

The location TOV must contain executable code. TOV is entered via a JP instruction.

Exit from TOV via a JP B6 instruction.

**Exit conditions:**

Only B1 must be preserved.

A pointer word must be incremented by the number of words newly available. If TN has not been altered during execution, the address of the pointer word is F.TEND. If TN has changed, the address of the pointer word is FTAB-1 plus the contents of TN.

**EQS - EQUALITY SEARCH TABLE**

EQS searches for a specified entry.

**Entry conditions:**

(A0) Table number  
(B7) Word count per entry  
(X6) Entry for search

**Exit conditions:**

(X2) = 0 if entry not found  
(X2) = entry, if found  
(A2) = address of entry found

**Registers used:**

A1, A2, A6  
X1, X2, X3, X7

**MES - MASKED EQUALITY SEARCH TABLE**

MES searches for a specified entry using a mask.

**Entry conditions:**

(A0) Table number  
(B7) Word count per entry  
(X0) Mask  
(X6) Entry for search

**Exit conditions:**

(X2) = 0 if entry not found  
(X2) = entry, if found  
(A2) = address of entry found

**Registers used:**

A1, A2, A6  
X1, X2, X3, X4, X7

**MTD - MOVE TABLES DOWN**

MTD moves the tables down (away from RA) to eliminate unused memory.

**Exit conditions:**

(B2) Number of tables

**Registers used:**

A1, A2, A3, A7  
B2, B3  
X0, X1, X2, X3, X4, X7

**Calls:**

MVE=

MTU - MOVE TABLES UP

MTU moves the tables up (toward RA) to eliminate unused memory.

**Registers used:**

A1, A2, A7  
B3  
X0, X1, X2, X3, X7

**Calls:**

MVE=

### 12.2.12 COMCMVE – MOVE BLOCK OF DATA

COMCMVE moves a block of data to a specified location. COMCMVE moves the data from the source address through the source address plus the word count minus one to the destination address through the destination address plus the word count minus one. The move can be in either direction. MVE= is the only entry point for COMCMVE.

**Entry conditions:**

(B1) 1  
(X1) Word count  
(X2) Source address  
(X3) Destination address

**Registers used:**

A2, A4, A6, A7  
B7  
X1, X2, X3, X4, X6, X7

### 12.2.13 COMCRDC – READ CODED LINE, C FORMAT

Before a data transfer routine is called, a CIO read function macro (READ, READW, and so forth) must be used to establish the type of read and initialize the type of transfer required.

If any other type of CIO function is used (REWIND, SKIP, and so forth) on the file, it may be necessary to reestablish the type of read/transfer before another data transfer routine is called.

When an EOR or EOF is sensed while performing the data transfer, the CIO read function macro must be called again to continue reading the file. Refer to the NOS or NOS/BE reference manual for a description of the CIO macros.

COMCRDC reads a coded line terminated by a zero byte from a CIO buffer to a working buffer. RDC= is the only entry point for COMCRDC.

**Entry conditions:**

- (B6) FWA of working buffer
- (B7) Word count of working buffer
- (X2) Address of FET for file

If B7 is less than zero, then the complement of B7 is the word count of the working buffer; COMCRDC will not read and discard words until an end-of-line for lines longer than the working buffer.

**Exit conditions:**

- (B1) 1
- (B6) Address of last word transferred to working buffer plus one
- (X1) Status of transfer:
  - 0 Transfer completed
  - 1 EOF detected on file
  - 2 EOI detected on file
  - B6 EOR detected on file before transfer completed
- (X2) Address of FET for file
- (X4) Contents of last data word transferred before EOL guaranteed
- (X7) Level number of EOR

**Registers used:**

- A1, A2, A3, A4, A6, A7
- B1, B2, B3, B4, B5, B6, B7
- X1, X2, X3, X4, X6, X7

**Calls:**

LCB=, RDX=

### 12.2.14 COMCRDH — READ CODED LINE, H FORMAT

Before a data transfer routine is called, a CIO read function macro (READ, READW, and so forth) must be used to establish the type of read and initialize the type of transfer required.

If any other type of CIO function is used (REWIND, SKIP, and so forth) on the file, it may be necessary to reestablish the type of read/transfer before another data transfer routine is called.

When an EOR or EOF is sensed while performing the data transfer, the CIO read function macro must be called again to continue reading the file. Refer to the NOS or NOS/BE reference manual for a description of the CIO macros.

COMCRDH reads a coded line terminated by a zero byte from a CIO buffer to a working buffer with trailing space fill. RDH= is the only entry point for COMCRDH.

**Entry conditions:**

- (B6) FWA of working buffer
- (B7) Word count of working buffer
- (X2) Address of FET for file

**Exit conditions:**

- (B1) 1
- (B6) Address of last word transferred to working buffer plus one

- (X1) Status of transfer:
  - 0 Transfer completed
  - 1 EOF detected on file
  - 2 EOI detected on file
  - B6 EOR detected on file before transfer completed
- (X2) Address of FET for file
- (X7) Level number of EOR

**Registers used:**

A1, A2, A3, A4, A6  
 B1, B2, B3, B4, B5, B6, B7  
 X1, X2, X3, X4, X6, X7

**Calls:**

LCB=, RDX=

**12.2.15 COMCRDO – READ ONE WORD**

Before a data transfer routine is called, a CIO read function macro (READ, READW, and so forth) must be used to establish the type of read and initialize the type of transfer required.

If any other type of CIO function is used (REWIND, SKIP, and so forth) on the file, it may be necessary to reestablish the type of read/transfer before another data transfer routine is called.

When an EOR or EOF is sensed while performing the data transfer, the CIO read function macro must be called again to continue reading the file. Refer to the NOS or NOS/BE reference manual for a description of the CIO macros.

COMCRDO reads one word from a CIO buffer into X6. RDO= is the only entry point for COMCRDO.

**Entry conditions:**

- (A1) Address of IN pointer
- (X1) IN

**Exit conditions:**

- (B1) 1
- (X1) Status of transfer:
  - 0 Transfer completed
  - 1 EOR detected on file
  - 1 EOF detected on file
  - 2 EOI detected on file
- (X2) Address of FET for file
- (X6) Word read

**Registers used:**

A1, A2, A3, A4, A6, A7  
 B1  
 X1, X2, X3, X4, X6, X7

**Calls:**

CIO=

## 12.2.16 COMCRDS – READ CODED LINE TO STRING BUFFER

Before a data transfer routine is called, a CIO read function macro (READ, READW, and so forth) must be used to establish the type of read and initialize the type of transfer required.

If any other type of CIO function is used (REWIND, SKIP, and so forth) on the file, it may be necessary to reestablish the type of read/transfer before another data transfer routine is called.

When an EOR or EOF is sensed while performing the data transfer, the CIO read function macro must be called again to continue reading the file. Refer to the NOS or NOS/BE reference manual for a description of the CIO macros.

COMCRDS reads a coded line from a CIO buffer to a working buffer. Words in the circular buffer are unpacked and stored one character per word in the working buffer. This process is continued until the end-of-line byte is detected. If the coded line terminates before the working buffer is filled, the working buffer is padded with spaces; the buffer is not padded if the complement of the word count of the buffer is used. If the coded line exceeds the size of the working buffer, the excess characters are ignored. RDS= is the only entry point for COMCRDS.

Entry conditions:

(B6) FWA of working buffer  
(B7) Word count of working buffer  
(X2) Address of FET for file

If B7 is less than 0, B7 is the complement of the buffer length and the string buffer will not be space filled.

Exit conditions:

(B1) 1  
(B6) Address of the last character from the coded line in the working buffer plus one  
(X1) Status of transfer:  
    0 Transfer completed  
    -1 EOF detected on file  
    -2 EOI detected on file  
    B6 EOR detected on file before transfer completed  
(X2) Address of FET for file  
(X7) Level number of EOR

Registers used:

A1, A2, A3, A4, A6, A7  
B1, B2, B3, B4, B5, B6, B7  
X1, X2, X3, X4, X6, X7

Calls:

LCB=, RDX=

## 12.2.17 COMCRDW – READ WORDS TO WORKING BUFFER

Before a data transfer routine is called, a CIO read function macro (READ, READW, and so forth) must be used to establish the type of read and initialize the type of transfer required.

If any other type of CIO function is used (REWIND, SKIP, and so forth) on the file, it may be necessary to reestablish the type of read/transfer before another data transfer routine is called.



When an EOR or EOF is sensed while performing the data transfer, the CIO read function macro must be called again to continue reading the file. Refer to the NOS or NOS/BE reference manual for a description of the CIO macros.

COMCRDW reads a specified number of words from a CIO buffer to a working buffer. COMCRDW also contains the load CIO buffer and read exit routines required by COMCRDC, COMCRDH, and COMCRDS. RDW=, LCB=, and RDX= are the entry points for COMCRDW. The RDX\$ assembly option controls read-ahead. The programmer can prevent read-ahead by defining the symbol RDX\$.

**Entry conditions:**

- (B6) FWA of working buffer
- (B7) Word count of working buffer
- (X2) Address of FET for file

**Exit conditions:**

- (B1) 1
- (B6) Address of last word transferred to the working buffer plus one
- (B7) Word count remaining to be transferred
- (X1) Status of transfer:
  - 0 Transfer completed
  - 1 EOF detected on file
  - 2 EOI detected on file
  - 3 CIO= was called to read more data and returned an error status
  - B6 EOR was detected on file before transfer was completed
- (X2) Address of FET for file
- (X7) Error status if X1 is -3, otherwise level number of EOR

**Registers used:**

- A1, A2, A3, A4, A6, A7
- B1, B2, B3, B4, B5, B6, B7
- X1, X2, X3, X4, X6, X7

**Calls:**

- CIO=

### 12.2.18 COMCRSR — RESTORE ALL REGISTERS

COMCRSR restores the B, A, and X registers from a specified register save area. The format of the registers in the save area is B0, B1, . . . , B7, A0, A1, . . . , A7, X0, X1, . . . , X7. Each register occupies a full word with the B and A register values in bits 17 through 0. RSR= is the only entry point for COMCRSR.

**Entry conditions:**

- (X1) Address of register save area

**Exit conditions:**

- All registers are set to the content of the register save area.

**Registers used:**

- A0, A1, A2, A3, A4, A5, A6, A7
- B1, B2, B3, B4, B5, B6, B7
- X0, X1, X2, X3, X4, X5, X6, X7

### 12.2.19 COMCSFN – SPACE FILL NAME

COMCSFN converts trailing 00 characters in a word to blanks. SFN= is the only entry point for COMCSFN.

Entry conditions:

(X1) Name left justified, zero fill  
(B1) 1

Exit conditions:

(X6) Name space filled  
(X7) Final character mask

Registers used:

A3  
B2  
X3, X6, X7

### 12.2.20 COMCSRT – SET RECORD TYPE

COMCSRT attempts to identify the format of a record, given the initial part of that record (64 words are usually sufficient) in a working buffer. The type codes returned are listed in table 12-2. SRT= is the only entry point for COMCSRT.

Entry conditions:

(B1) 1  
(X1) LWA+1 of block  
(X2) FWA of current record

Exit conditions:

(X6) 42/0L → name, 12/0, 6/type number  
(X7) Record name in L format

If type number and record name are zero, the record is zero length.

Registers used:

A1, A2, A3, A7  
B2, B3  
X0, X1, X2, X3, X4, X6, X7

### 12.2.21 COMCSST – SORT TABLE USING SHELL SORT

COMCSST sorts a table of one word entries into ascending order using a shell sort. All of the entries should be of the same sign. SST= is the only entry point for COMCSST.

Entry conditions:

(B1) 1  
(B7) Address of table to be sorted  
(X1) Number of elements in the table

Exit conditions:

The table is sorted.

TABLE 12-2. TYPE CODES RETURNED BY COMCSRT

Type	Number	Format	Determined by
TEXT	0	Text record	No 7700 <sub>g</sub> table and first word with all zeros in bits 0 through 17
6PP	1	6000-series peripheral processor overlay	7700 <sub>g</sub> table with three-character name in header word
OVCAP	2	Overlay Capsule	7700 <sub>g</sub> table followed by 6000 <sub>g</sub> table with bit 18=1
REL	3	Relocatable subprogram	3400 <sub>g</sub> table
OVL	4	Central processor overlay	5000 <sub>g</sub> table, 5300 <sub>g</sub> table with bit 17=0, or 5400 <sub>g</sub> table with non-(0,0) overlays
ULIB	5	NOS user library	7600 <sub>g</sub> table
OPL	6	Modify program library deck	7001 <sub>g</sub> table with 0 word count
OPLC	7	Modify program library common deck	7002 <sub>g</sub> table with 0 word count
OPLD	8	Modify program library directory	7000 <sub>g</sub> table with 0 word count
ABS	9	Multiple entry point overlay	5100 <sub>g</sub> table, 5300 <sub>g</sub> table with bit 17=1, or 5400 <sub>g</sub> table with (0,0) overlays
7PP	10	7000-series peripheral processor overlay	5200 <sub>g</sub> table
UPLX	11	Update sequential program library with X master control character	No 7700 <sub>g</sub> table and characters CHECK in bits 30 through 59 (control character obtained from bits 0 through 5)
UCF	12	Update compressed compile file	7700 <sub>g</sub> table with 0 word count
ACF	13	Modify compressed compile file	Bits 0 through 17 in second word of 7700 <sub>g</sub> table are non-zero
CAP	14	Fast dynamic load capsule	7700 <sub>g</sub> table followed by 6000 <sub>g</sub> table
DATA	15	Arbitrary data	Unrecognizable by criteria defined in these tables
PROC	16	Procedure record	PROC followed by delimiter
	17	CDC reserved	
SDR	18	Special deadstart record	NOS/BE 1 deadstart tape position
UPLR	19	Update random program library	7700 <sub>g</sub> table followed by 6000 <sub>g</sub> table followed by COMDECK, YANK, or DECK

**Registers used:**

A1, A2, A6, A7  
B2, B3, B4, B5  
X1, X2, X3, X4, X6, X7

**12.2.22 COMCSTF – SET TERMINAL FILE**

COMCSTF detects if a file is assigned to an interactive terminal. STF= is the only entry point for COMCSTF.

**Entry conditions:**

(B1) 1  
(X2) Address of FET

The FET must be greater than five words in length.

**Exit conditions:**

(X2) Address of FET  
(X6) 0 if file is assigned to a terminal

**Registers used:**

A1, A4, A6  
X1, X3, X4, X6

**Calls:**

CIO=

**12.2.23 COMCSVR – SAVE ALL REGISTERS**

COMCSVR saves the B, A, and X registers in a specified register save area. The registers are saved in the following order:

B0, B1, ..., B7, A0, A1, ..., A7, X0, X1, ..., X7

Each register occupies a full word with the B and A register values in bits 17 through 0. B and A registers are sign extended. SVR= is the only entry point for COMCSVR.

**Entry conditions:**

Bits 17 through 0 of the word from which SVR= was called contain the address of the register save area.

**Exit conditions:**

(save thru save+7)	B registers
(save+8 thru save+15)	A registers
(save+16 thru save+23)	X registers

**Registers used:**

A0, A1, A2, A3, A4, A5, A6, A7  
B1, B2, B3, B4, B5, B6, B7  
X0, X1, X2, X3, X4, X5, X6, X7

## 12.2.24 COMCSYS – PROCESS SYSTEM REQUEST

COMCSYS issues a system monitor request through RA+1. SYS=, RCL=, WNB=, and MSG= are the entry points for COMCSYS.

### SYS= - PROCESS SYSTEM REQUEST

SYS= waits for RA+1 to clear before issuing the desired request. Central exchange jump hardware is used if it is available. If the hardware is not available and the auto-recall bit is set, SYS= waits for the monitor to process the call before returning.

Entry conditions:

(X6) System request

Exit conditions:

Request accepted by monitor

Registers used:

A1, A6  
X6 (Contents restored upon exit)

### RCL= - PLACE PROGRAM ON RECALL

RCL= issues a single system request for periodic recall. If RA+1 is busy, no request is issued.

Exit conditions:

Request processed.

Registers used:

A1  
X1, X6

### WNB= - WAIT NOT BUSY

WNB= waits for a specified status word, bit 0, to be set. If the word is initially 0, WNB= returns.

Entry conditions:

(X2) Address of status word

Exit conditions:

Returns when bit 0 of status word is set.

Registers used:

A1  
X1, X6

### MSG= - SEND MESSAGE

MSG= formats and issues a system request to send a message (80 characters or less) to the job dayfile. The message appears in the dayfile as two lines (if necessary) of 40 characters each. Messages exceeding 80 characters are truncated.

**Entry conditions:**

- (X1) Address of first word of data (data must be packed in sequential locations, and should not exceed 80 characters)
- (X6) Message options:
  - bit 16 - Auto recall if on
  - bits 11 through 0 - Message option code (see MESSAGE macro in operating system reference manual)

**Exit conditions:**

Returns when operation is complete.

**Registers used:**

A1, A6  
X1, X6

### 12.2.25 COMCUPC – UNPACK CONTROL CARD

COMCUPC unpacks a control statement into the keyword and individual parameters. The following conditions apply to the use of COMCUPC:

If B7 is negative on entry, a blank after the keyword is considered to be a separator; otherwise, blanks are ignored.

The characters ) and . are considered as the termination of the control statement.

Characters with display code values 0 or 60<sub>g</sub> through 77<sub>g</sub> are illegal before the terminator.

The parameter must contain 7 or fewer characters.

The parameters are stored left-justified with zero fill.

The separator character is placed in the lower 18 bits of the parameter unless it is a \*,\* in which case the lower 18 bits are zero.

Two successive separators or a separator followed by a terminator results in a parameter of all zeros.

UPC= is the only entry point for COMCUPC.

**Entry conditions:**

- (A5) Address of first word of control statement
- (B1) 1
- (B7) First word address of buffer containing parameter information
- (X5) First word of control statement

If B7 is negative, B7 contains the complement of the first word address of the parameter buffer.

**Exit conditions:**

- (B6) Parameter count
- (X6) 0 if no error during unpacking

**Registers used:**

A1, A2, A5, A6, A7  
B2, B3, B4, B5, B6  
X0, X1, X2, X3, X4, X5, X6, X7

### 12.2.26 COMCWOD – CONVERT WORD TO OCTAL DISPLAY CODE

COMCWOD converts a word into octal display code. WOD= is the only entry point for COMCWOD.

Entry conditions:

(X1) Word to be converted

Exit conditions:

(B1) 1  
(X6, X7) Conversion

Registers used:

A2, A3, A4, A5  
X0, X1, X2, X3, X4, X5, X6, X7

### 12.2.27 COMCWTC – WRITE CODED LINE, C FORMAT

COMCWTC writes a zero byte delimited line from a working buffer to a CIO buffer. If the CIO buffer becomes sufficiently full to require writing or if the device type indicates a NOS/BE terminal, COMCWTC performs a WRITE function unless the symbol WRIF\$ is defined. In this case, the CIO function that is in the FET is reissued. WTC= is the only entry point for COMCWTC.

When the data transfer is completed, a call to the WRITER or WRITEF CIO function is necessary to obtain the final contents of (to flush) the working buffer.

Entry conditions:

(B6) FWA of working buffer  
(X2) Address of FET for file

Exit conditions:

(B1) 1  
(X2) Address of FET for file

Registers used:

A1, A2, A3, A4, A6, A7  
B1, B2, B3, B4, B5, B6  
X1, X2, X3, X4, X6, X7

Calls:

DCB=, WTX=

### 12.2.28 COMCWTH – WRITE CODED LINE, H FORMAT

COMCWTH writes a coded line in H format from a working buffer to a CIO buffer. Trailing spaces are deleted. If the buffer becomes sufficiently full to require writing, or the device type indicates a NOS/BE terminal, COMCWTH performs a WRITE function unless the symbol WRIF\$ is defined. In this case, the CIO function that is in the FET is reissued. If the line to be written terminates with 6 bits of zero, a word containing a blank byte is appended to preserve the 00 character as a colon. If the line terminates on an end-of-line, it is written as is. WTH= is the only entry point for COMCWTH.

When the data transfer is completed, a call to the WRITER or WRITEF CIO function is necessary to obtain the final contents of (to flush) the working buffer.

**Entry conditions:**

- (B6) FWA of working buffer
- (B7) Word count of working buffer
- (X2) Address of FET for file

If B7 is 0, no transfer is performed.

**Exit conditions:**

- (B1) 1
- (X2) Address of FET for file

**Registers used:**

- A1, A2, A3, A4, A6, A7
- B1, B2, B3, B4, B5, B6, B7
- X1, X2, X3, X4, X6, X7

**Calls:**

DCB=, WTX=

### 12.2.29 COMCWTO – WRITE ONE WORD

COMCWTO writes one word to a CIO buffer from X6. If the buffer becomes sufficiently full to require writing, COMCWTO performs a WRITE function unless the symbol WRIF\$ is defined. In this case, the CIO function that is in the FET is reissued. WTO= is the only entry point for COMCWTO.

When the data transfer is completed, a call to the WRITER or WRITEF CIO function is necessary to obtain the final contents of (to flush) the working buffer.

**Entry conditions:**

- (A1) Address of IN pointer
- (X1) IN
- (X6) Word to write

**Exit conditions:**

- (B1) 1
- (X2) Address of FET for file

**Registers used:**

- A1, A2, A3, A4, A6, A7
- B1
- X1, X2, X3, X4, X6, X7



### 12.2.30 COMCWTS – WRITE CODED LINE FROM STRING BUFFER

COMCWTS writes a coded line from a working buffer to a CIO buffer with trailing space suppression. Characters in the working buffer are packed and stored in the circular buffer. If the buffer becomes sufficiently full to require writing or if the device type indicates a NOS/BE terminal, COMCWTS performs a WRITE function unless the symbol WRIF\$ is defined. In this case, the CIO function that is in the FET is reissued. WTS= is the only entry point for COMCWTS.

When the data transfer is completed, a call to the WRITER or WRITEF CIO function is necessary to obtain the final contents of (to flush) the working buffer.

Entry conditions:

(B6) FWA of working buffer  
(B7) Word count of working buffer  
(X2) Address of FET for file

If B7 is 0, no transfer is performed.

Exit conditions:

(B1) 1  
(B6) Word count of data written  
(X2) Address of FET for file

Registers used:

A1, A2, A3, A4, A6, A7  
B1, B2, B3, B4, B5, B6, B7  
X1, X2, X3, X4, X6, X7

Calls:

DCB=, WTX=

### 12.2.31 COMCWTW – WRITE WORDS FROM WORKING BUFFER

COMCWTW writes data from a working buffer to a CIO buffer. If the buffer becomes sufficiently full to require writing or if the device type indicates a NOS/BE terminal, COMCWTW performs a WRITE function unless the symbol WRIF\$ is defined. The WTX\$ assembly option controls write-behind. The programmer can prevent write-behind by defining the symbol WTX\$. In this case, the CIO function that is in the FET is reissued. WTW=, DCB=, and WTX= are the entry points for COMCWTW.

When the data transfer is completed, a call to the WRITER or WRITEF CIO function is necessary to obtain the final contents of (to flush) the working buffer.

Entry conditions:

(B6) FWA working buffer  
(B7) Word count of working buffer  
(X2) Address of FET for file

If B7 is 0, no transfer is performed.

Exit conditions:

(B1) 1  
(B6) Address of next word to be transferred from working buffer  
(B7) Status of transfer:  
    0 Transfer completed  
    other Remaining word count if CIO= was called to write data and returned an error status  
(X2) Address of FET for file  
(X7) Error status if B7 is 0

Registers used:

A1, A2, A3, A4, A6, A7  
B1, B2, B3, B4, B5, B6, B7  
X1, X2, X3, X4, X6, X7

Calls:

CIO=

### 12.2.32 COMCXJR – RESTORE ALL REGISTERS WITH A SYSTEM XJR CALL

COMCXJR restores all registers from a register save area with a system XJR call. The format of the registers in the save area is B0, B1, . . . , B7, A0, A1, . . . , A7, X0, X1, . . . , X7. Each register occupies a full word with the B and A register values in bits 17-0. XJR= is the only entry point for COMCXJR.

Entry conditions:

(X1) Address of the register save area.

Exit conditions:

All registers are set to the contents of the register save area.

Registers used:

A0, A1, A2, A3, A4, A5, A6, A7  
B0, B1, B2, B3, B4, B5, B6, B7  
X0, X1, X2, X3, X4, X5, X6, X7

### 12.2.33 COMCZTB – CONVERT ALL 00 CHARACTERS TO BLANKS

COMCZTB converts all 00 characters in a word to blanks. ZTB= is the only entry point for COMCZTB.

Entry conditions:

(B1) 1  
(X1) Word to be converted

Exit conditions:

(X6) Converted word  
(X7) Final character mask

Registers used:

A3  
X3, X6, X7

## 12.3 MACROS THAT CALL THE COMMON COMMON DECKS

Entry points in the common common decks can be called by using system macros. Table 12-3 shows which macros call entry points in the common common decks. All of the macros are supported under NOS and NOS/BE. Only the MOVE macro is supported under SCOPE 2. All macros applicable to a given operating system exist in the system text CPUTEXT. Each macro is described in detail in the following paragraphs.

TABLE 12-3. MACROS THAT CALL COMMON COMMON DECKS

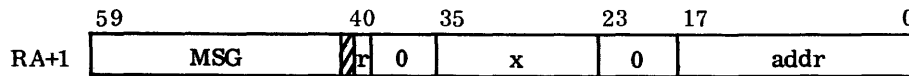
Macro	Entry Points Called	Description
MESSAGE	MSG=	Displays a message on the system console and enters it in a dayfile.
MOVE	MVE=	Moves a block of data from one address to another.
READC	RDC=	Reads one coded line from the input/output buffer to the working buffer.
READH	RDH=	Reads one coded line with space fill from the input/output buffer to the working buffer.
READO	RDO=	Reads one word from the input/output buffer to X6.
READS	RDS=	Reads a line image to a character buffer.
READW	RDW=	Fills the working buffer from an input/output buffer.
RECALL	RCL= WNB=	Relinquishes the CPU until a function is completed or the CPU recall time has elapsed.
SYSTEM	SYS=	Requests the system to process any three-character request.
WRITEC	WTC=	Writes a coded line image from the working buffer to the input/output buffer.
WRITEH	WTH=	Writes a coded line, deleting all trailing spaces, from the working buffer to the input/output buffer.
WRITEO	WTO=	Writes one word from X6 to the input/output buffer.
WRITES	WTS=	Writes a line image from the character buffer.
WRITEW	WTW=	Writes data from the working buffer to the input/output buffer.

### 12.3.1 MESSAGE

MESSAGE displays a message on the system console display and enters it into a dayfile. If the job is of system origin, the message can be flashed on the B display by including a dollar sign as the first character of the message. MESSAGE requires the common common deck COMCSYS.

The maximum length that a message can be is 80 characters; up to 40 characters per line are displayed. The message ends with either the first word containing 12 bits of zeros in any byte or at the eightieth character. The user must pack the display code message in sequential locations before calling MESSAGE.

The format of the RA+1 call for this macro is:



Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	MESSAGE	addr, x, r

**addr** Beginning address of the message. If the upper 12 bits of the location specified by this address are zero, then the next 18 bits (47 thru 30) of this location are assumed to contain the beginning address of the message.

**x** Message routing option:

- 0 Message is placed in the system dayfile, the user dayfile, and is displayed at line 1 of the control point.
- 1 Message is displayed at line 1 of the control point.
- 2 Message is displayed at line 2 of the control point.
- 3 Message is placed in the user dayfile and displayed at line 1 of the control point.
- 4 Message is placed in the error log dayfile if the job is a special system job (that is, has an SSJ=entry point) or is of system origin; otherwise, the message is placed in the user dayfile.
- 5 Message is placed in the account dayfile if the job is a special system job or is of system origin; otherwise, the message is placed in the user dayfile.
- 6 Message is placed in the system dayfile, the user dayfile, and is displayed at line 1 of the control point.
- 7 Message is placed in the user dayfile and displayed at line 1 of the control point.

If x is not specified or is an illegal value, x=0 is assumed. If x is not defined, x=1 is assumed. If x is the character string LOCAL, x=3 is used.

**r** If r is specified, control is not returned until the operation is complete.

The control point message areas (lines 1 and 2) provide the user with the ability to display concurrently messages that enter the dayfile and those that require operator action. Line 2 is normally used to display information about the current status of the executing program.

Only messages that do not refer to the job, such as the control statements processed and compilers used, should be placed in the system dayfile (x=0). All messages that refer to the job, such as the path taken by the programs and the number of records copied, should be placed only in the user dayfile (x=3). All messages placed in the user dayfile (x=0 and x=6) are counted by the system. If the number of messages issued by the job exceeds the limit for which the user is validated, the error message MESSAGE LIMIT; is issued to the user dayfile and the job is aborted.

### 12.3.2 MOVE

MOVE moves a block of data from one address to another. MOVE requires the common common deck COMCMVE for absolute assemblies.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	MOVE	count, addr1, addr2

**count** Number of words in the block to be moved

**addr1** Address of the first word of the block to be moved

**addr2** Address of the first word of the destination

MOVE allows overlap in data moves (addr2 can be less than addr1 plus count).

### 12.3.3 READC

READC reads one coded line from the input/output buffer to the working buffer. Data is transferred until the end of the line (0000 in bits 11 through 0) is sensed or until the specified number of words are transferred. READC requires the common common deck COMCRDC.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	READC	addr, buf, n

**addr** FET address

**buf** Working buffer address

**n** Working buffer word count

### 12.3.4 READH

READH reads a coded line with space fill from the input/output buffer to the working buffer. Data is transferred until the end of the line (0000 in bits 11 through 0) is sensed or until the specified number of words are transferred. READH requires the common common deck COMCRDH.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	READH	addr, buf, n

addr FET address

buf Working buffer address

n Working buffer word count

### 12.3.5 READO

READO reads one word from the input/output buffer to X6. READO requires the common common deck COMCRDO.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	READO	addr

addr FET address

### 12.3.6 READS

READS reads a line image to a character buffer. The words are unpacked and stored in the working buffer right justified, one character per word, until the end-of-byte (000) is detected. If the coded line terminates before the specified number of characters are stored, the working buffer is blank filled. READS requires the common common deck COMCRDS.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	READS	addr, buf, n

addr FET address

buf Working buffer address

n Working buffer word count

### 12.3.7 READW

READW fills the working buffer from an input/output circular buffer. READW reads ahead in the input/output buffer. This could cause the program to abort if the last word address of the input/output buffer is within four words of the FL. If the word count is greater than the length of the working buffer, READW writes beyond the end of the working buffer. READW requires the common common deck COMCRDW.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	READW	addr, buf, n

addr FET address

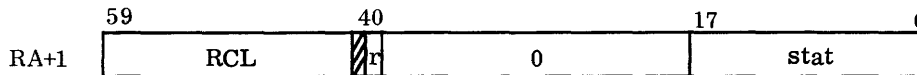
buf Working buffer address

n Working buffer word count

### 12.3.8 RECALL

RECALL enables the user to relinquish the CPU until a function is completed or the CPU recall time has elapsed (delay time depends on the operating system and the site). If the stat parameter is included in the call, control is not returned to the program until bit 0 of the word specified by stat is set. If stat is not included in the macro call, the program relinquishes the CPU only until the next pass through the recall loop. RECALL requires the common common deck COMCSYS.

The format of the RA+1 call for this macro is:



Macro format:

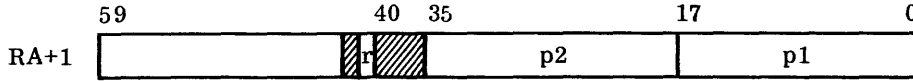
LOCATION	OPERATION	VARIABLE SUBFIELDS
	RECALL	stat

stat If this parameter is present, control is returned to the program when bit 0 of the word specified by the address stat is set.

### 12.3.9 SYSTEM

SYSTEM processes a three-letter request. The request can be either the functions that MTR performs or a PP program. A PP program can be called from a CPU program if the first character of the name is alphabetic. SYSTEM requires the common common deck COMCSYS.

The format of the RA+1 call for this macro is:



Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	SYSTEM	req, r, p1, p2

- req Three-character system request
- r If specified, control is returned only after the request is completed
- p1 Bits 17 through 0 of the request
- p2 Bits 35 through 18 of the request

### 12.3.10 WRITEC

WRITEC writes a coded line image from the working buffer to the input/output buffer. Data is transferred until the end of the line (0000 in bits 11 through 0) is sensed. WRITEC requires the common common deck COMCWTC.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	WRITEC	addr, buf

- addr FET address
- buf Working buffer address



### 12.3.11 WRITEH

WRITEH writes a coded line, deleting all trailing spaces, from the working buffer to the input/output buffer. WRITEH requires the common common deck COMCWTH.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	WRITEH	addr, buf, n

addr FET address

buf Working buffer address

n Working buffer word count

### 12.3.12 WRITEO

WRITEO writes one word from X6 to the input/output buffer. WRITEO requires the common common deck COMCWTO.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	WRITEO	addr

addr FET address

### 12.3.13 WRITES

WRITES writes a line image from the working buffer. Characters are packed ten characters per word. Trailing spaces are deleted before the characters are packed. WRITES requires the common common deck COMCWTS.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	WRITES	addr, buf, n

addr FET address

buf Working buffer address

n Working buffer word count

### 12.3.14 WRITEW

WRITEW writes data from the working buffer to the input/output circular buffer. WRITEW writes ahead in the input/output buffer. This could cause the program to abort if the last word address of the input/output buffer is within four words of the FL. If the word count is greater than the length of the working buffer, WRITEW reads beyond the end of the working buffer. WRITEW requires the common common deck COMCWTW.

Macro format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	WRITEW	addr, buf, n

**addr**    FET address

**buf**     Working buffer address

**n**        Working buffer word count

# CHARACTER SETS

A

---

## NOTES

1. The terms upper case and lower case apply only to the case conversions, and do not necessarily reflect any true case.
2. When translating from display code to ASCII/EBCDIC the upper case equivalent character is taken.
3. When translating from ASCII/EBCDIC to display code, the upper case and lower case characters fold together to a single display code equivalent character.
4. All ASCII and EBCDIC codes not listed are translated to display code 55 (space).
5. Where two display code graphics are shown for a single octal code, the leftmost graphic corresponds to the CDC 64-character set (system assembled with IP CSET set to C64.1), and the rightmost graphic corresponds to the CDC 64-character ASCII subset (system assembled with IP CSET set to C64.2).
6. In a 63-character set system, the display code for the : graphic is 63. The % character does not exist, and translations from ASCII/EBCDIC % or ENQ yield blank (55<sub>g</sub>). The display code value 00 is undefined in 63-character set systems.
7. Twelve or more zero bits at the end of a 60-bit word are interpreted as an end-of-line mark rather than two colons. An end-of-line mark is converted to external BCD 1632 and internal BCD 1672 by operating systems when writing 7-track magnetic tape in even parity (coded) mode, and converted back to 0000 when reading.
8. This code is changed to 12 when written on a 7-track magnetic tape in even parity (coded) mode.
9. 11-0 and 11-8-2 are equivalent on input. The character will be punched as 11-0 on output.
10. 12-0 and 12-8-2 are equivalent on input. The character will be punched as 12-0 on output.
11. 12-8-7 and 11-0 are equivalent on input. The character will be punched as 12-8-7 on output.
12. 12-8-4 and 12-0 are equivalent on input. The character will be punched as 12-8-4 on output.
13. CODE pseudo selects 6-bit octal code as follows:

A	ASCII
D	Display Code (default)
E	External BCD
I	Internal BCD

Display Code		Hollerith Punch (026)	BCD		ASCII						EBCDIC				
Octal (13)	Char. (7)		Ext. (13)	Int. (13)	Upper Case			Lower Case			Upper		Lower		
					6-Bit Octal (13)	Hex.	Char.	Punch (029)	Hex.	Char.	Punch	Hex.	Char.	Hex.	Char.
00	:	8-2	00	12	32	3A	:	8-2	1A	SUB	9-8-7	7A	:	3F	SUB
01	A	12-1	61	21	41	41	A	12-1	61	a	12-0-1	C1	A	81	a
02	B	12-2	62	22	42	42	B	12-2	62	b	12-0-2	C2	B	82	b
03	C	12-3	63	23	43	43	C	12-3	63	c	12-0-3	C3	C	83	c
04	D	12-4	64	24	44	44	D	12-4	64	d	12-0-4	C4	D	84	d
05	E	12-5	65	25	45	45	E	12-5	65	e	12-0-5	C5	E	85	e
06	F	12-6	66	26	46	46	F	12-6	66	f	12-0-6	C6	F	86	f
07	G	12-7	67	27	47	47	G	12-7	67	g	12-0-7	C7	G	87	g
10	H	12-8	70	30	50	48	H	12-8	68	h	12-0-8	C8	H	88	h
11	I	12-9	71	31	51	49	I	12-9	69	i	12-0-9	C9	I	89	i
12	J	11-1	41	41	52	4A	J	11-1	6A	j	12-11-1	D1	J	91	j
13	K	11-2	42	42	53	4B	K	11-2	6B	k	12-11-2	D2	K	92	k
14	L	11-3	43	43	54	4C	L	11-3	6C	l	12-11-3	D3	L	93	l
15	M	11-4	44	44	55	4D	M	11-4	6D	m	12-11-4	D4	M	94	m
16	N	11-5	45	45	56	4E	N	11-5	6E	n	12-11-5	D5	N	95	n
17	O	11-6	46	46	57	4F	O	11-6	6F	o	12-11-6	D6	O	96	o
20	P	11-7	47	47	60	50	P	11-7	70	p	12-11-7	D7	P	97	p
21	Q	11-8	50	50	61	51	Q	11-8	71	q	12-11-8	D8	Q	98	q
22	R	11-9	51	51	62	52	R	11-9	72	r	12-11-9	D9	R	99	r
23	S	0-2	22	62	63	53	S	0-2	73	s	11-0-2	E2	S	A2	s
24	T	0-3	23	63	64	54	T	0-3	74	t	11-0-3	E3	T	A3	t
25	U	0-4	24	64	65	55	U	0-4	75	u	11-0-4	E4	U	A4	u
26	V	0-5	25	65	66	56	V	0-5	76	v	11-0-5	E5	V	A5	v
27	W	0-6	26	66	67	57	W	0-6	77	w	11-0-6	E6	W	A6	w
30	X	0-7	27	67	70	58	X	0-7	78	x	11-0-7	E7	X	A7	x
31	Y	0-8	30	70	71	59	Y	0-8	79	y	11-0-8	E8	Y	A8	y
32	Z	0-9	31	71	72	5A	Z	0-9	7A	z	11-0-9	E9	Z	A9	z
33	0	0	12	00	20	30	0	0	10	DLE	12-11-9-8-1	F0	0	10	DLE
34	1	1	01	01	21	31	1	1	11	DC1	11-9-1	F1	1	11	DC1
35	2	2	02	02	22	32	2	2	12	DC2	11-9-2	F2	2	12	DC2
36	3	3	03	03	23	33	3	3	13	DC3	11-9-3	F3	3	13	TM
37	4	4	04	04	24	34	4	4	14	DC4	11-9-4	F4	4	3C	DC4

CODE D (default)      CODE E      CODE I      CODE A

↓            ↓            ↓

Display Code		Hollerith Punch (026)	BCD		ASCII						EBCDIC				
Octal (13)	Char.		Ext. (13)	Int. (13)	Upper Case		Lower Case				Upper		Lower		
Octal (13)	Char.			6-Bit Octal (13)	Hex.	Char.	Punch (029)	Hex.	Char.	Punch	Hex.	Char.	Hex.	Char.	
40	5	5	05	05	25	35	5	5	15	NAK	9-8-5	F5	5	3D	NAK
41	6	6	06	06	26	36	6	6	16	SYN	9-2	F6	6	32	SYN
42	7	7	07	07	27	37	7	7	17	ETB	0-9-6	F7	7	26	ETB
43	8	8	10	10	30	38	8	8	18	CAN	11-9-8	F8	8	18	CAN
44	9	9	11	11	31	39	9	9	19	EM	11-9-8-1	F9	9	19	EM
45	+	12	60	20	13	2B	+	12-8-6	0B	VT	12-9-8-3	4E	+	0B	VT
46	-	11	40	40	15	2D	-	11	0D	CR	12-9-8-5	60	-	0D	CR
47	*	11-8-4	54	54	12	2A	*	11-8-4	0A	LF	0-9-5	5C	*	25	LF
50	/	0-1	21	61	17	2F	/	0-1	0F	SI	12-9-8-7	61	/	0F	SI
51	(	0-8-4	34	74	10	28	(	12-8-5	08	BS	11-9-6	4D	(	16	BS
52	)	12-8-4	74	34	11	29	)	11-8-5	09	HT	12-9-5	5D	)	05	HT
53	\$	11-8-3	53	53	04	24	\$	11-8-3	04	EOT	9-7	5B	\$	37	EOT
54	=	8-3	13	13	35	3D	=	8-6	1D	GS	11-9-8-5	7E	=	1D	IGS
55	space	space	20	60	00	20	space	space	00	NUL	12-0-9-8-1	40	space	00	NUL
56	,	0-8-3	33	73	14	2C	,	0-8-3	0C	FF	12-9-8-4	6B	,	0C	FF
57	.	12-8-3	73	33	16	2E	.	12-8-3	0E	SO	12-9-8-6	4B	.	0E	SO
60	#	0-8-6	36	76	03	23	#	8-3	03	ETX	12-9-3	7B	#	03	ETX
61	[	8-7	17	17	73	5B	[	12-8-2	1C	FS	11-9-8-4	4A	¢	1C	IFS
62	]	0-8-2	32	72	75	5D	]	11-8-2	01	SOH	12-9-1	5A	!	01	SOH
63	%	8-6	16	16	05	25	%	0-8-4	05	ENQ	0-9-8-5	6C	%	2D	ENQ
64	"	8-4	14	14	02	22	"	8-7	02	STX	12-9-2	7F	"	02	STX
65	-	0-8-5	35	75	77	5F	-	0-8-5	7F	DEL	12-9-7	6D	-	07	DEL
66	!	11-0 <sup>9</sup>	52	52	01	21	!	12-8-7 <sup>11</sup>	7D	}	11-0	4F		D0	}
67	&	0-8-7	37	77	06	26	&	12	06	ACK	0-9-8-6	50	&	2E	ACK
70	'	11-8-5	55	55	07	27	'	8-5	07	BEL	0-9-8-7	7D	'	2F	BEL
71	?	11-8-6	56	56	37	3F	?	0-8-7	1F	US	11-9-8-7	6F	?	1F	IUS
72	<	12-0 <sup>10</sup>	72	32	34	3C	<	12-8-4 <sup>12</sup>	7B	{	12-0	4C	<	C0	{
73	>	11-8-7	57	57	36	3E	>	0-8-6	1E	RS	11-9-8-6	6E	>	1E	IRS
74	@	8-5	15	15	40	40	@	8-4	60	\	8-1	7C	@	79	\
75	≥ \	12-8-5	75	35	74	5C	≥ \	0-8-2	7C		12-11	E0	≥ \	6A	
76	^	12-8-6	76	36	76	5E	^	11-8-7	7E	~	11-0-1	5F	^	A1	~
77	;	12-8-7	77	37	33	3B	;	11-8-6	1B	ESC	0-9-7	5E	;	27	ESC

## HEXADECIMAL-OCTAL CONVERSION TABLE

		First Hexadecimal Digit															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Second Hexadecimal Digit	0	000	020	040	060	100	120	140	160	200	220	240	260	300	320	340	360
	1	001	021	041	061	101	121	141	161	201	221	241	261	301	321	341	361
	2	002	022	042	062	102	122	142	162	202	222	242	262	302	322	342	362
	3	003	023	043	063	103	123	143	163	203	223	243	263	303	323	343	363
	4	004	024	044	064	104	124	144	164	204	224	244	264	304	324	344	364
	5	005	025	045	065	105	125	145	165	205	225	245	265	305	325	345	365
	6	006	026	046	066	106	126	146	166	206	226	246	266	306	326	346	366
	7	007	027	047	067	107	127	147	167	207	227	247	267	307	327	347	367
	8	010	030	050	070	110	130	150	170	210	230	250	270	310	330	350	370
	9	011	031	051	071	111	131	151	171	211	231	251	271	311	331	351	371
	A	012	032	052	072	112	132	152	172	212	232	252	272	312	332	352	372
	B	013	033	053	073	113	133	153	173	213	233	253	273	313	333	353	373
	C	014	034	054	074	114	134	154	174	214	234	254	274	314	334	354	374
	D	015	035	055	075	115	135	155	175	215	235	255	275	315	335	355	375
	E	016	036	056	076	116	136	156	176	216	236	256	276	316	336	356	376
	F	017	037	057	077	117	137	157	177	217	237	257	277	317	337	357	377
Octal		000 – 037	040 – 077	100 – 137	140 – 177	200 – 237	240 – 277	300 – 337	340 – 377								

---

## SCOPE 2

COMPASS 3 under SCOPE 2 uses the Record Manager for all of its I/O operations. Thus, COMPASS 3 can read and write files with a variety of external formats. For each of the files used by COMPASS, the default format, and the combinations of file format description parameters that may be specified in FILE control statements to override the defaults, are given below.

### Main Source Input File

The main source input file may be a normal source input file or a compressed compile file; COMPASS determines which it is by inspecting the data in the file. A normal source input file under SCOPE 2 comprises the following:

File Organization (FO)	sequential (SQ)
Block Type (BT)	unblocked
Maximum Block Length (MBL)	none
Record Type (RT)	control word (W)
Maximum Record Length (MRL)	160 characters
Conversion Mode (CM)	NO
Label Type (LT)	unlabeled (UL)

The only other formats that may be specified by FILE control statements are as follows (X means allowed):

Block Type	Record Type		
	F	W	Z
unblocked	X	X	
C	X	X	X
I		X	

File Organization (FO) must be sequential (SQ).

Maximum Record Length (MRL) must not exceed 160 characters.

Label Type (LT) may be any value supported by the operating system.

Although the maximum record length may be as large as 160 characters, only the first 90 characters of each record are reproduced in the listing output files.





<u>File Characteristics</u>	<u>SCOPE 2</u>	<u>NOS and NOS/BE 1</u>
File Organization (FO)	sequential (SQ)	sequential (SQ)
Block Type (BT)	unblocked	character count (C)
Maximum Block Length (MBL)	none	5120 chars.
Record Type (RT)	control word (W)	system-logical-record (S)
Maximum Record Length (MRL)	1,310,710 chars.	none
Conversion Mode (CM)	NO	NO
Label Type (LT)	Unlabeled (UL)	ANY

No other formats are allowed, except that the label type (LT) can be any value supported by the operating system used for assembly. The format shown above under SCOPE 2 is the default binary output file format under that system.

### Scratch Files

COMPASS uses two scratch files named ZZZZZRL and ZZZZZRM, when table storage space overflows. Regardless of what is specified by FILE control statements, COMPASS sets the file format description parameters for these files under SCOPE 2 as follows:

File Organization (FO) = sequential (SQ).

Conversion Mode (CM) = NO.

For file ZZZZZRL:

Block Type (BT) = unblocked.

Maximum Block Length = 5120 characters.

Record Type (RT) = undefined (U) Maximum Record Length = 2550 characters.

For file ZZZZZRM:

Block Type (BT) = character count (C), Maximum Block Length = 5120 characters.

Record Type (RT) = SCOPE logical (S), no Maximum Record Length.

## ALL OPERATING SYSTEMS

### System Text Input Files

A user library file designated by an S parameter on the COMPASS control statement must have the standard library file format for the system on which COMPASS is being used.<sup>†</sup> COMPASS uses the operating system overlay loader to access these files.

For a sequential binary (non-library) file designated by a G parameter on the COMPASS control statement, the default and permitted formats are the same as those given above for the COMPASS binary output file.

---

<sup>†</sup>Overlay residence in user libraries is not currently supported by NOS.

## XTEXT Input Files

A file read by COMPASS when processing an XTEXT pseudo instruction can have any of several formats. COMPASS determines the file format (a) by whether the XTEXT pseudo instruction variable field is empty and (b) by inspecting the data in the file.

If the variable field is empty, the File Organization (FO) must be sequential (SQ). COMPASS rewinds the file and reads until end of section or a COMPASS END statement is encountered, whichever comes first. The default and permitted formats under SCOPE 2 are the same as those given above for the main source input file.

If the XTEXT variable field is non-empty, the file organization can be any of three non-standard types:

Record indexed with name index (under SCOPE 2 only).

SCOPE 3.3 style random file with name index (not supported under SCOPE 2).

Update or Modify<sup>†</sup> random program library file.

In each case, COMPASS sets the file format description parameters to the appropriate values; no FILE control statement is needed.

The record indexed file organization is actually the word addressable (WA) file organization with a set of format conventions superimposed on it. Such a file can be created by a FORTRAN program by using the library subroutines OPENMS, STINDEX, WRITMS, and CLOSMS with a name index, or by a COBOL program specifying ORGANIZATION IS WORD-ADDRESS, WORD-ADDRESS IS data-name. When COMPASS detects such a file under SCOPE 2, it sets the file format description parameters as follows (no FILE card is needed):

File Organization (FO) = word addressable (WA).

Block Type (BT) = unblocked.

Record Type (RT) = control word (W); Maximum Record Length (MRL) = 160 characters.

Conversion Mode (CM) = NO.

COMPASS positions the file at the record pointed to by the index entry containing the name given in the XTEXT statement variable field, and then reads records sequentially until end of section or a COMPASS END statement is encountered, whichever comes first.

The SCOPE 3.3 style random file with name index is permitted for compatibility with previous versions of COMPASS. When COMPASS detects such a file, it searches the file index and positions the file at the beginning of the specified section, and then reads sequentially until end of section or a COMPASS END statement is encountered, whichever comes first. Such files cannot be used with SCOPE 2.

An Update or Modify<sup>†</sup> random program library file is processed similarly. The name in the variable field of the XTEXT statement must be the name of a common deck. When COMPASS detects such a file under SCOPE 2, it sets the file format description parameters as follows (no FILE control statement is needed):

---

<sup>†</sup>Modify is not available under SCOPE 2 or NOS/BE 1.

File Organization (FO) = word addressable (WA).

Block Type (BT) = unblocked

Record Type (RT) = control word (W), Maximum Record Length (MRL) = 5120 characters

Conversion Mode (CM) = NO

COMPASS positions the file at the first card image of the designated section (common deck). For an UPDATE program library, the first active card image (the \*COMDECK card) is skipped. COMPASS then reads card images sequentially, ignoring inactive card images, until end of section or a COMPASS END statement is encountered, whichever comes first.

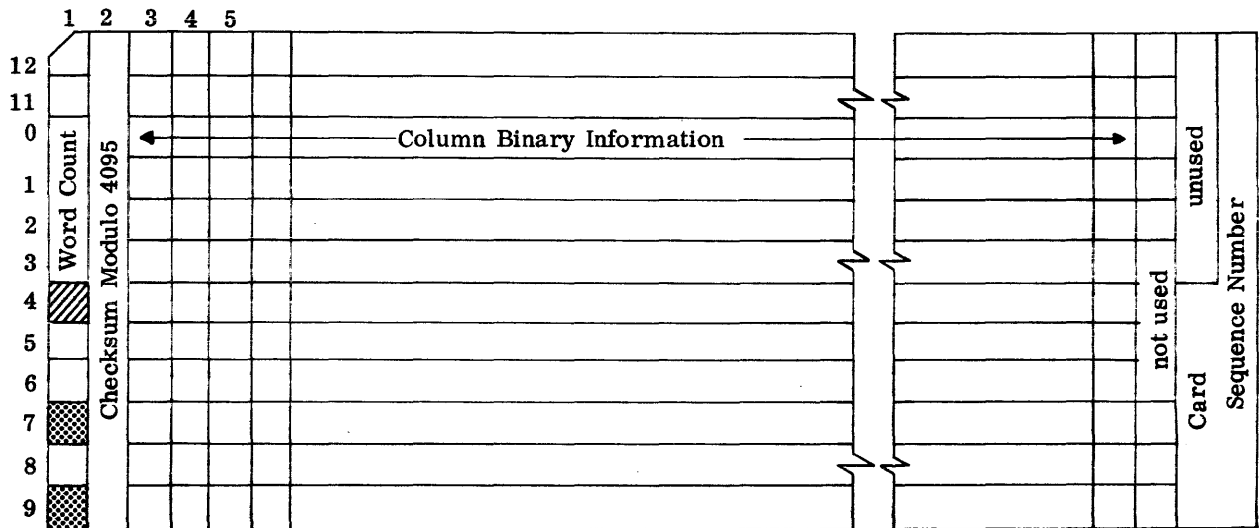


# BINARY CARD FORMATS

C

## Column 1

7, 8, 9 levels 0 to 16	End-of-section
6, 7, 9	End-of-partition (NOS only)
6, 7, 8, 9 or 7, 8, 9 level 17	End-of-information
7, 9	Binary card
7 and 9 not both in column 1	Coded card



A binary card can contain up to 15 60-bit CPU words starting at column 3. Column 1 also contains a count of 60-bit words in rows 0, 1, 2, and 3 plus a check indicator in row 4. If row 4 of column 1 is zero, column 2 is used as a checksum for the card on input; if row 4 is one, no check is performed on input.

Column 78 of a binary card is not used, and columns 79 and 80 contain a binary serial number. If a section is punched, each card has a checksum in column 2 and a serial number in columns 79 and 80, which sequences it within the logical record.



## HINTS ON USING COMPASS

D

---

1. Within a macro definition:

Use comment statements having \* in column one. These are not saved, whereas other types of comments are saved.

Whenever possible, minimize the number of lines of code.

IRP is faster than either ECHO or DUP.

Use the substitute parameter flags ;A, ;B, and so forth, for macros, to avoid a second line.

Within macros, use symbols such as .1, .2, and so forth, instead of local symbols.

If possible, avoid recursive macro structure to increase assembly speed.

If a macro call is the cause of an error, direct full list output to a file other than OUTPUT (L=filename) to obtain a list of the erroneous macro call with the error listing.

2. In IF sequences:

Use line counts rather than ENDIF to terminate sequences.

Use SKIP rather than IFPP to skip code.

3. Micros:

Micro replacement is time-consuming.

Avoid using local symbols for micros.

Use ## for a null substitution.

4. Minimize SYSTEXT size.

5. To reduce core requirements, use SEG statements in absolute programs.

6. Use NOREF for symbols for which listing is not required.

7. Use QUAL for all overlays.

The program EXAMPLE (figures D-1 and D-2) presents fundamental program organization. It also demonstrates some COMPASS coding conventions and illustrates efficient coding practice. The program obtains numbers from six successive locations, adding the numbers one at a time to the running sum. The total is then printed with a label.

1				IDENT	EXAMPLE	
2				ENTRY	BEGIN	
3	0	301		OBUF	BSS 3018	OUTPUT BUFFER
4	301	17252420252400000001		OUTPUT	FILEC OBUF,3018	
5	306	6110000001		BEGIN	SB1 1	
6		66200			SB2 80	INITIALIZE ADDRESS COUNTER TO ZERO
7	307	6130000006			SB3 6	SET FOR USE AS A LOOP LIMIT
8		43100			MX1 0	INITIALIZE RUNNING SUM TO ZERO
				*		
9	310	5122000321 +		LOOP	SA2 TABLE+B2	GET NEXT MEMORY ADDRESS
10		66221			SB2 B2+B1	INCREMENT THE ADDRESS COUNTER
11		36121			IX1 X2+X1	ADD NEW NUMBER TO RUNNING SUM
12	311	0523000310 +			NE B2,B3,LOOP	LOOP IF ADDRESS CNTR # (B3)
				*		
13		0100000443 +			RJ =XCDD	CONVERT BINARY NUMBER TO DISPLAY CODE
14	312	5160000331 +			SA6 ANS	STORE THE DISPLAY CODED NUMBER IN ANS
15		6160000327 +			WRITEH OUTPUT,WORDS,LEN	WRITES TO THE OUTPUT BUFFER
16	315	7120000301 +			WRITER OUTPUT	PRINTS CONTENTS OF OUTPUT BUFFER
18	317	7160247021			ENDRUN	END OF EXECUTABLE CODE
				*		
19	321	00000000000000000001		TABLE	DATA 1	-----
20	322	00000000000000000002			DATA 2	THESE ARE
21	323	00000000000000000003			DATA 3	THE NUMBERS
22	324	00000000000000000004			DATA 4	TO TOTAL.
23	325	00000000000000000005			DATA 5	
24	326	00000000000000000006			DATA 6	-----
				*		
25	327	55241005550116232705		WORDS	DATA H* THE ANSWER IS *	
26	331	1		ANS	BSS 1	
27		3		LEN	EQU *-WORDS	
				*		
				*	ACCESS TO EXTERNAL TEXT.	
				*		
28					SST	TO DEFINE SYSTEM SYMBOLS
				*		
29	332			RNCPL	XTEXT COMCSYS	
30	372			RNCPL	XTEXT COMCWTH	
31	425			RNCPL	XTEXT COMCCIO	
32	440			RNCPL	XTEXT COMCCOD	
33	452			END	BEGIN	END OF PROGRAM
	Octal Location	51200B CM	STORAGE USED	443 STATEMENTS	78 SYMBOLS	
	Addresses		7600-TYPE CPU ASSEMBLY	0.216 SECONDS	25 REFERENCES	
34		THE ANSWER IS	21			

Octal Code  
Assembled

Figure D-1. Example COMPASS Program (NOS and NOS/BE)





One of the main considerations in assembly language programming is the reduction of execution time. The instruction repertoire of COMPASS often allows an operation to be coded in several ways. The programmer, therefore, should give careful consideration to the instructions used in the program to perform specific functions.

Line 1. The IDENT pseudo instruction is always the first instruction in a program. It specifies a program name (EXAMPLE, in this case) to identify the program to the assembler.

Line 2. The ENTRY pseudo instruction declares the point in the program at which execution is to begin. The main entry point in a program is the control transfer address.

Line 3. NOS and NOS/BE - figure D-1. The BSS instruction establishes the output buffer OBUF. The programmer has allocated 301<sub>8</sub> words of storage for the buffer, as shown in the assembled octal code listed to the left of the source code. Note that the octal code format for the pseudo instructions will differ from the format for the symbolic machine instructions because pseudo instructions do not have single machine instruction equivalents.

Line 4. NOS and NOS/BE - figure D-1. The operating system macro FILEC is called to create a file environment table (FET) for the output buffer. Only the first word of the FET is shown in the octal code, but examination of the location addresses reveals that the table is actually five words in length (the minimum length of a FET). For more information about FETs, see the appropriate operating system reference manual.

SCOPE 2 - figure D-2. The FILE macro is used to establish a file information table (FIT) for the output buffer.

Line 5. The first executable line of code has been designated the main entry point for the program. Incrementing by one occurs so often within a program that it has become a COMPASS coding convention for register B1 to always be initialized to one, and to remain one throughout the entire program. This is particularly important during the use of the common common decks (chapter 12), and can be a factor in execution time (see B1=1 pseudo instruction) as well as in assembly time.

Line 6. A counter is initialized to zero by setting the contents of a B register (chapter 8) equal to the contents of the B0 register. B0 is hard-wired to zero, thereby avoiding the need for repeated processing of the literal or constant zero.

Line 7. Comparing the octal code for lines 6 and 7, the programmer can see the difference between two forms of register-setting instructions. The 15-bit form of the instruction is used in line 6, where only three bits are required to represent the B0 register as the source of an operand. The 30-bit form of set B register instruction is required for line 7, where the constant 6 is represented by the lower 18 bits of the instruction.

Line 8. The mask instruction is normally used to extract fields from a register. Here, it is used instead of the slower set X register instruction to initialize an X register.

Another important feature of COMPASS is illustrated here. The octal code seems to indicate that the lower 15 bits of the current word in memory have been left blank. This is the result of a force upper. The next instruction is too large to fit in the remaining 15-bit parcel, so COMPASS packs that parcel with a no-operation instruction. The next instruction is placed at the beginning of the next word (see section 8.1).

Line 9. The use of the set A register instruction to obtain a word of data is demonstrated here. As seen in the octal code, the address of the word (321<sub>8</sub>) is placed in the specified A register. The data itself is placed in the corresponding X register (X2 in this instance). (See Set A Register Instructions, chapter 8.)

The plus sign (+) after the octal code indicates that the address or K portion of the instruction (the lower 18 bits in this case) is relocatable.

Line 10. The 15-bit format of the set B instruction is illustrated here. The first six bits contain the operation code for the instruction (66g in this instance). The next three bits designate the destination register (B2) for the results of the instruction. The next three bits indicate the register containing the first source operand (B2). The final three bits indicate the source register for the second source operand (B1).

Line 11. The number obtained in the previous instruction is added to the running sum kept in X1. This is a 60-bit add instruction, as opposed to the SXi instruction, which adds only 18-bit operands.

Line 12. The NE instruction shows another use of the B registers in testing for a conditional branch. In each iteration of the loop, the source operands are compared. While they are unequal, control is transferred from this instruction back to LOOP. When the operands become equal, control passes to the next instruction.

Line 13. The return jump (RJ) instruction is used here to access a common common deck, COMCCDD, as a relocatable subroutine. The programmer has taken advantage of the COMPASS default method of defining external symbols. The =X indicates to the assembler that CDD, the entry point to the subroutine, is external to EXAMPLE.

The use of common common decks is important to the programmer. Note that the decks require certain entry conditions. Specific arguments are expected to be in certain registers, for example, upon entry to the routines. An efficient program will establish these conditions with a minimum of data transfers by using the registers judiciously prior to the call. COMCCDD, for example, converts an octal word to decimal display code; that word is expected to be in register X1. For this reason, the running total has been kept in X1, avoiding the need for extra data transfers.

Line 14. The method of storing an operand in memory is illustrated here. Setting register A6 or A7 to a valid address causes the contents of X6 or X7, respectively, to be stored in the address specified. When COMCCDD has converted the word, it places the result in register X6, ready for storage upon return to the calling routine.

Line 15. NOS and NOS/BE - figure D-1. Another method of accessing a common common deck is shown here. A call is made to a system macro, WRITEH, which utilizes the common common deck COMCWITH to write a line from a working buffer to an output buffer.

SCOPE 2 - figure D-2. The Record Manager macro OPENM is used to open the output buffer in preparation for processing.

Line 16. NOS and NOS/BE - figure D-1. A call is made to the operating system macro WRITER to write the contents of the buffer OBUF (with which the system communicates through the FET OUTPUT) to the system default output file, also named OUTPUT. (For more information about operating system macros, see the appropriate operating system reference manual.)

SCOPE 2 - figure D-2. The Record Manager macro PUTW is used to transfer data into the output buffer.

Line 17. SCOPE 2 - figure D-2. The Record Manager macro CLOSEM is used to close the output buffer and to print its contents.

Line 18. The operating system macro ENDRUN is called to terminate program execution.

Lines 19 through 24. DATA pseudo instructions are used here to establish a table comprising six consecutive words in memory, starting at location TABLE. The default base mode is base 10 in COMPASS (see Mode Control, chapter 4).

Line 25. DATA is used here to set in memory a display-coded image of the characters specified, for use in the output line. Ten 6-bit characters can be stored per word in this fashion. Therefore, more than one word is required here, as seen from the location address on the next line.

Line 26. One word of memory is reserved for the final sum. This word is labeled ANS. Note that this word is not initialized by the BSS instruction.

Line 27. The symbol LEN is equated with the value of the origin counter minus the address of WORDS. This yields the length of the output line specified in line 15.

Line 28. The SST instruction ensures that symbols from the system texts used by the program are defined.

Lines 29 through 32. These XTEXT pseudo instructions tell COMPASS to search the system-defined program library OPL for the common common decks named. Declarations of this type are normally grouped together after the end of the executable code for easy reference.

Line 33. The END instruction signifies the end of the program. Control is released through the transfer address at BEGIN.

The dayfile for the program EXAMPLE as run on NOS is shown in figure D-3.

The dayfile for the program EXAMPLE as run on NOS/BE is shown in figure D-4.

The dayfile for the program EXAMPLE as run on SCOPE 2 is shown in figure D-5.

```
15.18.00.EXAMPLE.  
15.18.00.UCCR, 7641,      0.048KCDS.  
15.18.00. USER statement.  
15.18.01. CHARGE statement.  
15.18.01.ATTACH,COMCPL/UN=xxx.  
15.18.02.COMPASS(S,S=IPTTEXT,S=CPUTEXT,X=COMCPL)  
15.18.09. ASSEMBLY COMPLETE. 523008 CM USED.  
15.18.09. 0.244 CPU SECONDS ASSEMBLY TIME.  
15.18.09.LGO.  
15.18.10.UEAD,      0.002KUNS.  
15.18.10.UEPF,      0.014KUNS.  
15.18.10.UEMS,      0.784KUNS.  
15.18.10.UECP,      0.252SECS.  
15.18.10.AESR,      2.622UNTS.  
15.22.18.UCLP, 7645,      0.256KLNS.
```

The parameter xxx is the site-defined NOS user name.

Figure D-3. Dayfile of EXAMPLE under NOS

```

09.17.20.EXAMP2P FROM
09.17.20.IP C0C00320 WORDS - FILE INPUT , DC 04
09.17.20.EXAMPLE.
09.17.20.ACCOUNT statement.
09.17.23.ATTACH,COMCPL,ID= yyy.
09.17.23.PFN IS
09.17.23.COMCPL
09.17.24.AT CY= 001 SN=PFQST
09.18.34.COMPASS(S,S=IPTXT,S=CPUTXT,X=COMCPL)
09.18.51. ASSEMBLY COMPLETE. 57600B CM USED.
09.18.51. 3.492 CPU SECONDS ASSEMBLY TIME.
09.18.51.LGD.
09.18.51.OP CCCC1024 WORDS - FILE OUTPUT , DC 40
09.18.51.MS 3584 WORDS ( 7168 MAX USED)
09.18.52.CPA 1.497 SEC. 1.497 ADJ.
09.18.52.CPB 2.261 SEC. 2.261 ADJ.
09.18.52.ID .495 SEC. .495 ADJ.
09.18.52.CM 121.287 KWS. 7.402 ADJ.
09.18.52.SS 11.656
09.18.52.PP 4.849 SEC. DATE 02/06/81
09.18.52.EJ END OF JOB, **

```

The value yyy is the site-defined ID under which COMCPL has been catalogued.

Figure D-4. Dayfile of EXAMPLE under NOS/BE



# DAYFILE MESSAGES

E

The dayfile messages that can be issued by COMPASS are listed in table E-1.

The following message, with xxxxxxx denoting the name of the subprogram being assembled, is displayed at the system operator's console only; it is not written to the dayfile. COMPASS updates the display whenever it processes an IDENT statement with a non-blank variable field.

ASSEMBLING xxxxxxx

TABLE E-1. DAYFILE MESSAGES

Message	Significance	Action
ASSEMBLY ABORTED - ECS READ ERROR.	This message can occur only when the job has an ECS field length and is used on a CYBER 180 or a CYBER 170 or CYBER 70 Model 71, 72, 73, or 74. COMPASS may store some of its internal tables in ECS. When an ECS error persists through four attempts to read the data, the message is issued, and the job is aborted. For the CYBER 70 Model 76, LCM errors are handled by the operating system.	Rerun job. If condition persists, contact a system analyst.
ASSEMBLY ABORTED - ECS WRITE ERROR.	This message can occur only when the job has an ECS field length and is used on a CYBER 180 or a CYBER 170 or CYBER 70 Model 71, 72, 73, 74. COMPASS may store some of its internal tables in ECS. When an error occurs in writing data to ECS, no retry attempt is made. The message is issued, and the job is aborted. For the CYBER 70 Model 76, LCM errors are handled by the operating system.	Rerun job. If condition persists, contact a system analyst.





TABLE E-1. DAYFILE MESSAGES (Contd)

Message	Significance	Action
<p>ASSEMBLY ERRORS. nnnnnnB { CM } USED.            xxxx.xxx CPU { SECONDS ASSEMBLY TIME. }                              { SEC. nnnnnnB { ECS } USED. }</p>	<p>If COMPASS detected at least one fatal error during assembly, this message is issued at the completion of processing of all source programs on the input file. If the A option was specified on the COMPASS control statement, the job is aborted after this message is issued. The minimum field length needed to perform the assemblies successfully is the octal number of SCM words, nnnnnn. The second line of the message can be suppressed by an installation parameter; xxxx.xxx represents the total central processor time, in seconds, used by COMPASS. If any ECS/LCM space was assigned to the job, nnnnnn is the octal number of words used.</p>	<p>Correct the fatal errors and reassemble.</p>
<p>BAD CONTROL STATEMENT ARGUMENT - xx</p>	<p>The COMPASS control statement contains an unrecognized or invalid argument. The offending argument is named in the message.</p>	<p>Refer to chapter 10 of this manual to correct the COMPASS control statement.</p>
<p>CANT LOAD COMP3\$</p>	<p>The operating system loader reported a fatal error when COMPASS attempted to load its primary overlay. This message should be preceded by an explanatory message from the loader.</p>	<p>Refer to the loader diagnostics in the loader reference manual for information about the specific loader error.</p>
<p>COMPASS NEEDS AT LEAST nnnnnB SCM.</p>	<p>The SCM field length for the job is too small for COMPASS. The number of octal words needed by COMPASS before it can begin processing is nnnnnn. This number varies depending on the version of COMPASS used and the listing and binary output options specified on the control statement. It is an absolute minimum number of words; it does not include whatever space may be required for system text, local macro and micro definitions, and so forth.</p>	<p>Rerun job inserting an RFL statement specifying sufficient field length.</p>

TABLE E-1. DAYFILE MESSAGES (Contd)

Message	Significance	Action
nnnnnnnnn ERRORS IN xxxxxxxx	COMPASS issues this message for each source program in which fatal errors are detected; nnnnnnnnn is the number of errors and xxxxxxxx is the sub-program name.	Correct the fatal errors and reassemble.
FILE USE CONTRADICTION.	Control statement specifies the same file name for two or more of the following:  Source input List output (full or short list) Binary output XTEXT source	Correct contradiction.
IDENT STATEMENT MISSING.	COMPASS issues this message for each source program in which an END statement is encountered before an IDENT statement is found. This is a fatal error.	Correct the source program to include an IDENT and END statement for each subprogram.
IMPROPER SYSTEM TEXT FORMAT. BAD SYSTEM TEXT - x=yyyyyy/zzzzzz	A system text overlay does not have the internal format required by this version of COMPASS. This may be caused by a system error. COMPASS ignores the bad overlay but does not abort the job. The expression, x=yyyyyy/zzzzzz, identifies the offending overlay in the same form in which it is specified in the COMPASS control statement; it may be any of the following:  G=filename G=filename/overlay S=overlay S=library/overlay	Correct the internal format of the system text overlay.
INPUT FILE EMPTY OR MISPOSITIONED.	When attempting to read the first line from the source input file, COMPASS encountered end of data and aborted.	Correct the name of the source input file or reposition the file.
INPUT FILE RECORD TYPE NOT ALLOWED.	The record type of the source input file is not allowed. COMPASS aborts the job step.	Convert source input to acceptable record type.

TABLE E-1. DAYFILE MESSAGES (Contd)

Message	Significance	Action
<p>INSUFFICIENT STORAGE FOR SYSTEM TEXT. BAD SYSTEM TEXT - x=yyyyyy/zzzzzz</p>	<p>When an irrecoverable table overflow occurs, COMPASS issues this message before the first assembly is begun. It does not abort the job step. The expression, x=yyyyyy/zzzzzz, identifies the system text being loaded at the time.</p>	<p>Increase the SCM field length for the job.</p>
<p>nnnnnB LCM NEEDED TO CONTINUE.</p>	<p>The specified amount of memory (nnnnnB) is required for the job to complete. The job step is aborted.</p>	<p>Increase the LCM field length for the job.</p>
<p>nnnnnnnnn LOST REFERENCES IN xxxxxxx</p>	<p>The symbolic cross-reference table is sorted before it is printed. If the table does not fit in the job's SCM field length for sorting, COMPASS discards some of the references. A message is issued; nnnnnnnn is the number of references discarded, and xxxxxxx is the subprogram name. The job step is not aborted. The ASSEMBLY COMPLETE message gives the field length needed to avoid lost references.</p>	<p>Increase the SCM field length for the job.</p>
<p>MORE THAN 7 SYSTEM TEXTS SPECIFIED.</p>	<p>COMPASS issues this message and aborts the job step, when the G and S parameters on the COMPASS control statement specify a total of more than seven system text overlays.</p>	<p>Restructure the job to reduce the number of system text overlays required.</p>
<p>NO CONTROL STATEMENT TERMINATOR.</p>	<p>Before finding a parenthesis or period not in a \$-delimited string, COMPASS read continuation control statements and encountered an end-of-section. This is not a fatal error.</p>	<p>Correct the control statement.</p>

TABLE E-1. DAYFILE MESSAGES (Contd)

Message	Significance	Action
<p>RECURSION DEPTH EXCEEDED 400.</p>	<p>COMPASS maintains a pushdown stack for source input control. This stack has one entry for each active DUP, ECHO, HERE, XTEXT, or macro call. The maximum depth of the stack is set by an installation parameter; it is 400 in the released system. When this limit is exceeded, COMPASS sets a fatal error and clears the stack. The next statement can then be read from the source input file. The job step is not aborted. This error is usually caused by a source program in which a macro calls itself indefinitely.</p>	<p>Correct the macro call program error.</p>
<p>SYSTEM TEXT NOT FOUND. BAD SYSTEM TEXT - x=yyyyyy/zzzzzz</p>	<p>When it cannot load the system text overlay identified by x=yyyyyy/zzzzzz, COMPASS issues this message. It does not abort the job step. For an overlay loaded from a library file (S parameter), this message should be preceded by an explanatory message from the operating system loader. For an overlay loaded from a non-library file (G parameter), COMPASS could not find the overlay on the file.</p>	<p>For an overlay loaded from a library file, refer to the diagnostics in the loader reference manual. For an overlay loaded from a non-library file, check that the overlay name is specified correctly and that the overlay is located on the file.</p>
<p>nnnnnnnn WARNING MESSAGES IN xxxxxxx</p>	<p>COMPASS issues this message for each source program in which nonfatal errors are detected; nnnnnnnn is the number of errors, and xxxxxxx is the subprogram name.</p>	<p>Correct the non-fatal errors and reassemble.</p>

# GLOSSARY

F

---

## Absolute Block -

A block of object code generated in an absolute assembly. The ABS pseudo instruction is used to declare a program absolute.

## Assembler -

A computer language that prepares an executable program from a source language program by substituting machine operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

## Blank Common Block -

A common block into which no data is stored at load time. The first declaration of a blank common block need not be the largest declaration for the common block.

## Block -

A grouping of words of object code or storage within a subprogram for a specific purpose.

## Capsule -

A relocatable collection of one or more programs bound together in a special format that allows the programs to be loaded and unloaded dynamically to form an executing program by the Fast Dynamic Loading facility.

## Central Processor Unit (CPU) -

The high-speed arithmetic unit that performs the addition, subtraction, multiplication, division, incrementing, logical operations, and branching instructions needed to execute programs.

## Comment Line -

A statement providing documentary information for a section of code. Comment lines are indicated by either an asterisk in column 1 or blanks in columns 1 through 29, and are listed but not otherwise processed by the assembler.

## Comments Field -

The field in a COMPASS statement providing documentary information for the statement. It is listed but not otherwise processed by the assembler. This field begins with the first nonblank character following the variable field, or in column 30 if the variable field is blank.

## Common Block -

An area of memory that can be declared by more than one subprogram and used for storage of shared data.

## Constant -

An expression element consisting of a value represented in octal, decimal, hexadecimal, or character notation.

## Data Item -

A type of character or numeric value that can be used in subfields of the DATA and LIT instructions, and as specifications of field lengths on VFD pseudo instructions.

## Entry Point -

A location within a subprogram that can be referenced from other subprograms. Each entry point has a name with which it is associated.

**External Reference -**

A reference in one subprogram to an entry point in another subprogram.

**Force Upper -**

To guarantee that an instruction begins on a word boundary by packing the parcels remaining in a partially completed word with no-op instructions and beginning to assemble the specified instruction in the next word. The assembler automatically forces upper in some cases, and the user program can specify that a given instruction be forced upper.

**Labeled Common Block -**

A common block into which data can be stored at load time. The first program declaring a labeled common block determines the amount of memory allocated.

**Linking -**

The process of matching external references to entry points of the same names and inserting the addresses of the entry points into the external references.

**Literal -**

A read-only constant. Conventionally, it is the only element in an expression. Literals are stored in the program's literals block to avoid duplication of read-only data.

**Literals Block -**

A block of literal data entries local to a subprogram.

**Load Sequence -**

One or more consecutive control statements processed by the loader as a unit. A load sequence can be a single name call statement, or it can consist of loader statements (such as LOAD and LDSET) that are terminated by NOGO, EXECUTE, or a name call statement.

**Local Block -**

A storage area defined by a USE or USELCM pseudo instruction.

**Location Counter -**

Normally the same as the origin counter. Can be reset by the programmer to relocate code or data without affecting relative positions within the block.

**Location Field -**

The first field in a COMPASS statement, usually providing a name for the address of the instruction or for the entity defined by the statement. The location field begins in column 1 or 2.

**Machine Instruction -**

A string of bits capable of being interpreted directly by a central processor or peripheral processor as an instruction to perform some operation.

**Macro -**

A sequence of source statements that are saved and then assembled whenever needed through a macro call.

**Micro -**

A character string identified by a symbolic name. Wherever the name is encountered in the program, the character string is substituted.

**OPDEF -**

A sequence of source statements that are saved and then assembled whenever needed through an opdef call. Differs from a macro in that the assembler interprets the call by examining the format or syntax of the instruction rather than the contents of the operation field alone.

- Operation Code -**  
A mnemonic operator, used in the operator field of a COMPASS statement, to indicate a specific machine instruction.
- Operation Field -**  
The field in a COMPASS statement indicating the operation to be performed. It begins with the first nonblank character following the location field; or, if the location field is blank, it begins with the first nonblank character after column 2.
- Origin Counter -**  
A pointer indicating the relative location of the next word to be assembled or reserved in a given block.
- Overlay -**  
One or more relocatable programs that were relocated and linked together into a single absolute program.
- Parcel -**  
One of the 15-bit sections of a central memory word. A CPU machine instruction occupies one, two, or four parcels.
- Peripheral Processor Unit (PP or PPU) -**  
An individual computer with its own memory, used for high-speed transfer of information (input and output) between peripheral devices and central memory.
- Position Counter -**  
A pointer indicating the bit position within the word of the next item to be assembled in a given block.
- Program -**  
One or more subprograms capable of being executed as a unit.
- Pseudo Instruction -**  
An assembler-defined instruction appearing in the operation field of a statement. It normally does not specify the assembly of a single machine instruction, but instead specifies some other assembly process (such as symbol definition, listing control, and so forth).
- Qualified Symbol -**  
A symbol defined when a qualifier is in effect during assembly. Through qualification, the same symbol can be referred to in different subprograms without conflict.
- Reference Address ( $RA_c$ ) and ( $RA_e$ ) -**  
 $RA_c$  is the absolute central memory address that is the starting or zero relative address assigned to a program. Addresses within the program are relative to  $RA_c$ .  $RA_e$  is the absolute extended memory starting address assigned to a program.
- Register -**  
A unit within the central processor used to hold operands. The A registers contain the addresses of words within central memory; the X registers contain operands used in calculations; the B registers are used for incrementing and indexing.
- Relocation -**  
Placement of object code into central memory in locations that are not predetermined, and adjusting the addresses accordingly.
- Remote Assembly -**  
An operation in which code is assembled, saved, and then inserted into the object code when specified.
- Strong External -**  
An external reference whose satisfaction is obligatory for program loading.

**Subprogram -**

A group of COMPASS statements beginning with an IDENT pseudo instruction and ending with an END pseudo instruction.

**Symbol -**

A set of characters that identifies a value and its associated attributes.

**Symbolic Instruction -**

An assembler-defined instruction appearing in the operation field of a statement. It provides a means of expressing symbolically the data manipulation functions of the machine. Each symbolic instruction typically generates one machine instruction.

**System Text -**

A set of tables containing symbol, micro, macro, and opdef definitions that can be saved on a file to be accessed by other programs.

**Transfer Address -**

The address of the entry point to which the loader jumps to begin program execution.

**Variable Field -**

The field in a COMPASS statement identifying operands for the statement. It consists of one or more subfields, and begins with the first nonblank character after the operation field.

**Weak External -**

An external reference that is ignored by the loader during library searching and cannot cause any other program to be loaded. A weak external is linked, however, if the corresponding entry point is loaded for any other reason.

**Zero Block -**

The nominal central memory block for a relocatable assembly. It is local to a sub-program. Also, a zero block is created for an absolute assembly if default symbols are used.



# INDEX

- A abort mode 10-2
- A code option 4-27
- A error 11-9
- A list option 4-74
- A reference table option 4-80
- A register
  - Description 8-7
  - Designators 2-8
  - Setting 8-44
  - Used for CM relocation 9-2
- ABS attribute 4-66
- ABS pseudo
  - Description 4-6
  - Example 4-4, 4-7, 4-13, 4-14, 4-16, 4-17
  - First statement group 4-2
- Absolute block
  - Absolute program 3-6
  - Description 3-2
  - Establishment 4-32
  - Relocatable program 3-5
  - Using 4-32, 4-33
- Absolute program
  - Declaration 4-6
  - Structure 3-6
- Absolute text 3-5
- ACN instruction 9-24
- ADC instruction
  - Arithmetic function 9-6
  - Description 9-11
  - Example 2-20, 9-11
- ADD instruction
  - Arithmetic function 9-6
  - Description 9-15
- Add unit
  - Floating point 8-3, 8-6
  - Long 8-3
- Address
  - Absolute 4-4
  - Direct 9-15
  - Entry point 4-4, 4-5, 4-45
  - External 4-6, 4-9, 4-47
  - Indexed direct 9-16
  - Indirect 9-15
- Address modes, PP 9-1
- ADI instruction
  - Arithmetic function 9-6
  - Description 9-15
- ADM instruction
  - Arithmetic function 9-6
  - Description 9-16
- ADN instruction
  - Arithmetic function 9-6
  - Description 9-10
- AIDTEXT 8-1, 9-1, 11-11
- AJM instruction 9-19
- AOD instruction
  - Description 9-15
  - Replace function 9-7
- AOI instruction
  - Description 9-15
  - Replace function 9-7
- AOM instruction
  - Description 9-16
  - Replace function 9-7
- Arithmetic functions, PP 9-6
- Arithmetic shift 8-33
- Arrow
  - Parameter separator 5-8, 5-13
  - Special character 2-4
- ASCII code
  - Character set A-1
  - Option 4-27
- Assembler 1-1
  - Central memory requirements 1-3, 10-1
  - Statistics 4-73, 11-8
- Assembly environment test 4-60
- Assembly listing
  - Detailed description 11-1
  - General description 4-73
  - Generation 1-3
- Assembly, remote code 5-3
- Assembly time 11-8
- Asterisk
  - BASE instruction 4-25
  - Element operator 2-22
  - First column 2-1, 2-2
  - Local symbol separator 5-31
  - Location counter 2-9, 3-4
  - Parameter separator 5-8, 5-13, 5-16, 5-24, 5-28
  - Special element 2-9, 2-32, 3-4
  - USE instruction 4-32
  - USELCM instruction 4-34
- Attribute, symbol 2-5
- Attribute test 4-66
- AXI instruction 8-32, 8-34
- B base 2-17, 2-18, 4-22
- B binary mode 10-2
- B list option 4-74
- B reference table option 4-80
- B register
  - Conditional jumps 8-26
  - Contents of 4-30
  - Description 8-7
  - Designators 2-8
  - Setting 8-46
- Base, assembly 4-23
  - COL column count 4-31
  - DIS word count 4-49
  - DUP count 5-6
  - ECHO count 5-7
  - Line count 4-60, 4-61, 4-63, 4-64, 4-67, 4-69, 4-70
  - Micro count 7-2, 7-4
  - Numeric value 2-16
  - Overlay level numbers 4-4
  - PP number 4-4
  - REP counts 4-57
  - Setting through BASE 4-24
  - SPACE line count 4-76
  - String count 2-13
  - VFD count 4-53
- BASE micro 7-6
- BASE pseudo
  - Description 4-24
  - Example 4-13, 4-19, 4-26, 4-49, 4-51
  - Permissible anywhere 4-2
- Binary card formats C-1
- Binary control statements 4-1, 4-74, 11-1
- Binary load module 3-8
- Binary mode 10-2

Binary output generation 1-3, 3-7, 3-9, 3-11,  
   3-13, 10-2  
 Binary write 3-8  
 Blank  
   Compressed 5-1  
   Embedded 2-1  
   Expression terminator 2-1  
   Name terminator 2-5  
   Operation field 2-1  
   Parameter separator 5-8, 5-13  
   Statement terminator 2-1  
   String terminator 2-14  
   Use in character data 2-14  
   Variable field 2-2, 2-3, 3-8  
 Blank card 4-76  
 Blank common  
   CM 4-32  
   Description 3-3  
   ECS 4-34  
   Establishment 4-32, 4-34  
   Example 4-38  
   LCM 4-34  
   SCM 4-32  
 Blank fill 2-14  
   DIS 4-49  
 Blank operation field 4-47  
 Block  
   Absolute 3-1, 4-34, 4-38  
   Blank common 3-3, 4-34, 4-36  
   Labeled common 3-2, 4-32  
   Literals 2-11, 3-2, 3-5 thru 3-15  
   Local 3-2, 4-32  
   Maximum number 3-1, 4-32  
   Origin assigned 1-2, 3-5, 3-7  
   Subprogram 3-1  
   Used for definition operation 5-2  
   User established 3-2, 4-32, 4-34  
   Zero 3-2, 4-32, 4-34  
 Block copy instruction 8-16  
 Block group 3-1, 3-12, 3-14  
 Block group listing 11-2  
 Block name 3-3, 4-32, 4-34  
 Block name listed 11-1  
 Block origin 1-2, 3-5  
 Block usage summary 11-2  
 Boolean unit  
   Description 8-3, 8-6  
   Instructions 8-27 thru 8-31, 8-35, 8-36  
 Branch instructions  
   CPU 8-13, 8-14, 8-17, 8-23, 8-24, 8-26  
   PP 9-7  
 Branch unit  
   Description 8-3  
   Instructions 8-10, 8-14, 8-17, 8-23, 8-24, 8-26  
 BSS Pseudo  
   Description 4-37  
   Effect on origin counter 3-3  
   Example 4-4, 4-7, 4-10, 4-16, 4-30, 4-35,  
   4-38, 4-39, 4-42, 4-46,  
   5-22, 5-32  
   Force upper 3-4  
 BSSZ pseudo  
   Description 4-48  
   Dumped by SEGMENT 4-16  
   Example 2-19, 5-33, 5-35  
   Force upper 3-4  
 BXi instruction 8-27 thru 8-31  
 Byte, guaranteed zero 2-14, 4-50  
 B1=1 or B7=1 pseudo instruction  
   Description 4-30  
   Effect on R= 4-55  
   Example 4-56  
   Illegal for PP 4-9, 4-10  
 C hardware feature code 4-8  
 C list option 4-74  
 C on octal listing 11-6  
 Call  
   Equivalenced macro 5-25  
   Macro 5-18  
   Opdef 5-29  
 CC instruction 8-53  
 CCF instruction 9-20  
 Central memory  
   Read instruction 8-46  
   Requirements 1-3, 10-1  
   Write instruction 8-46  
   Access instructions, PP 9-2  
 Central processor unit  
   Functional units 8-3, 8-6, 8-8  
   Instructions 8-1  
   Registers 8-7  
 CFM instruction 9-20  
 Channel buffer instruction  
   Read status 8-22  
   Reset input 8-21  
   Reset output 8-22  
 Channel flag instructions 9-20  
 CHAR  
   Define other character 4-26  
 Character sets A-1  
 Character data 2-13  
   Code conversion 4-26  
   Evaluation 2-27  
   Examples 2-12, 2-15  
 CMU 8-50  
 Code  
   CPU operation 6-7, 8-1  
   Duplication 5-6  
   Code other 4-26  
   PP operation 6-3, 9-1  
   Remote assembly 5-3  
   Replication 4-57  
 CODE micro 7-6  
 CODE pseudo  
   Description 4-26  
   Effect on character data 2-13, 4-49  
   Example 4-27  
   Permissible anywhere 4-2  
 Coding form 2-3  
 COL pseudo  
   Description 4-9  
   Octal listing 11-6  
 Column one 2-1  
 COM attribute 4-66  
 Comma  
   Character string 2-13  
   Column one 2-1  
   Continuation 2-1  
   Expression terminator 2-21  
   Local symbol separator 5-31  
   Name terminator 2-5  
   Parameter separator 5-8, 5-13, 5-16, 5-24, 5-28  
   String terminator 2-13  
   Subfield delimiter 2-1  
 COMMENT pseudo  
   Description 4-20  
   Example 4-13  
   First statement group 4-2  
 Comments column control 4-31  
 Comments field 2-2, 2-3, 4-31  
 Comments, prefix table 4-20  
 Comments statement 2-2  
   Heading of definition 5-13  
   Micros not substituted 7-1  
   Not counted 4-59, 5-7, 5-8  
   Permissible anywhere 4-2

Common common decks  
 COMCARG 12-3  
 COMCCDD 12-4  
 COMCCFD 12-4  
 COMCCIO 12-5  
 COMCCOD 12-5  
 COMCCPT 12-6  
 COMCDXB 12-6  
 COMCMNS 12-7  
 COMCMOS 12-7  
 COMCMTM 12-8  
 COMCMTP 12-9  
 COMCMVE 12-13  
 COMCRDC 12-13  
 COMCRDH 12-14  
 COMCRDO 12-15  
 COMCRDS 12-16  
 COMCRDW 12-16  
 COMCRSR 12-17  
 COMCSFN 12-18  
 COMCSRT 12-18  
 COMCSST 12-18  
 COMCSTF 12-20  
 COMCSVR 12-20  
 COMCSYS 12-21  
 COMCUPC 12-22  
 COMCWOD 12-23  
 COMCWTC 12-23  
 COMCWTH 12-23  
 COMCWTO 12-24  
 COMCWTS 12-25  
 COMCWTW 12-25  
 COMCXJR 12-25  
 COMCZTB 12-25

Compare character strings 4-68  
 Compare expression values 4-62  
 Compare/Move unit 8-50  
 COMPASS control statement  
 Description 10-2  
 Effect on LIST 4-79  
 Compile file 10-4  
 Complement and logical difference instruction 8-31  
 Complement and logical sum instruction 8-30  
 Complement instruction 8-29  
 Compressed code 5-1  
 CON pseudo  
 Description 4-54  
 Example 2-22, 4-55, 5-5, 5-23, 5-26  
 Force upper 3-4  
 Concatenation 2-4  
 Concatenation mark 2-4  
 Example of use 5-19  
 In definition 5-1  
 Conditional assembly 4-59  
 Conditional jump  
 B register 8-26  
 PP 9-7  
 X register 8-24  
 Configuration 1-3  
 Constant  
 Character 2-14  
 Description 2-11  
 Expression element 2-21, 2-26  
 Field size 2-12  
 Generated by pseudo 4-54  
 Numeric 2-16  
 Read only 2-11  
 Continuation, statement 2-2  
 Generation of lines 2-4, 7-1  
 Control statements  
 COMPASS 10-2  
 Job statement 10-1

Counter control  
 BSS 4-37  
 Forcing upper 3-4  
 LOC 4-38  
 ORG 4-35  
 ORGC 4-35  
 POS 4-40  
 USE 4-32  
 USELCM 4-34  
 Counters, block control 3-3, 3-10, 3-12  
 CPOP pseudo 6-7  
 CPSYN pseudo  
 Description 6-10  
 Permissible anywhere 4-2  
 CPU instructions  
 Block copy 8-16  
 Boolean 8-27 thru 8-31, 8-35, 8-36  
 Branching 8-10, 8-14, 8-17, 8-23, 8-24, 8-26  
 Channel buffer 8-21, 8-22  
 Channel status 8-22  
 Complement 8-29, 8-31  
 Conditional 8-24, 8-26  
 Direct LCM transfer 8-19  
 Divide 8-42  
 Double precision 8-38, 8-40  
 ECS 8-15  
 Error exit 8-14  
 Exchange exit 8-18  
 Exchange jump 8-17  
 Fixed point 8-39  
 Floating point 8-34 thru 8-40  
 Increment 8-44, 8-46, 8-48  
 Left shift 8-31, 8-32  
 Logical 8-28 thru 8-32  
 Long add 8-39  
 Mask 8-42  
 Multiply 8-39, 8-40, 8-41  
 No operation 8-43  
 Normalize 8-34  
 Pack 8-36  
 Pass 8-43  
 Population 8-43  
 Program stop 8-13  
 Real-time clock 8-21  
 Return jump 8-14  
 Right shift 8-32, 8-33  
 Set register 8-44, 8-46, 8-48  
 Set time 8-21  
 Shift 8-31 thru 8-33  
 Single precision 8-37 thru 8-40, 8-42, 8-43  
 Transmit 8-27  
 Unconditional jump 8-23  
 Unpack 8-35  
 CPU program execution 1-3, 10-1  
 CPU register designators 2-8, 8-11  
 CRD instruction 9-17  
 Created symbol 5-31, 11-8  
 CRM instruction 9-18  
 Cross reference table  
 (see symbolic reference table)  
 CTEXT pseudo 4-79  
 CR Instruction 8-46  
 CU instruction 8-54  
 CW instruction 8-46  
 CWD instruction 9-18  
 CWM instruction 9-18  
 CXi instruction 8-43

D base 2-17, 4-24  
 D code option 4-26  
 D debug mode 10-3

- D definition flag 11-14
- D error 11-10
- D hardware feature code 4-7
- D list option 4-74
- Data generation 4-47
- Data item
  - Character format 2-13
  - DATA pseudo 4-49
  - General description 2-10
  - LIT pseudo 4-51
  - Numeric format 2-17
  - VFD pseudo 4-53
- Data notation
  - Character 2-13
  - Constant 2-11, 2-13, 2-16
  - Decimal 2-17
  - Element 2-10, 2-21
  - Fixed point 2-17
  - Floating point 2-17
  - Hexadecimal 2-22
  - Item 2-11, 2-13, 2-16
  - Literal 2-12, 2-13, 2-16
  - Numeric 2-17
  - Octal 2-17
- DATA pseudo
  - Description 4-48
  - Example 2-15, 2-19, 2-20, 4-27, 4-33, 4-37, 4-49
  - Force upper 3-4
- Data transmission, PP 9-6
- DATE micro 7-5
- Date of listing 11-1
- Dayfile messages E-1
- DCN instruction 9-24
- Debug, interactive 1-4
- Debug mode 10-3
- Decimal exponent 2-17
- Decimal notation 2-17
- DECMIC pseudo
  - Description 7-4
  - Example 5-6, 7-4
  - Permissible anywhere 4-2
- DEF attribute 4-67
- Default symbols
  - Definition 2-7
  - Listing 11-9
  - Unqualified 4-27
  - Zero block 3-2
- Deferred symbols  
(see default symbols)
- Definition
  - Equivalenced macro 5-24
  - Macro 5-13, 5-15, 5-24
  - Micro 7-2
  - Opdef 5-13, 5-27
  - Processing 5-13
  - Purging 6-9
  - Reference 5-18, 5-25, 5-30
  - Symbol 2-6, 4-44
  - System 5-35
- Definition operation
  - Duplicated code 5-6
  - Equivalenced macro 5-13
  - External text 5-2
  - Macro 5-13
  - Operation code 5-13
  - Processing 5-14
  - Recursion level 5-1
  - Remote text 5-3
- Delimiter
  - Actual parameter 5-18, 5-26
  - Data item 2-15, 2-16
  - Expression element 2-21
- Delimiter (Contd)
  - Field 2-1, 2-2
  - Substitutable parameter 5-8, 5-13, 5-16
  - Term 2-22
- Descriptor, variable field 5-27
- Destination field 2-26
- Detailed listing 4-74
- DF instruction 8-24
- Direct address mode 9-15
- Directives, loader 4-21
- Directory, error 11-9
- DIS pseudo
  - Description 4-47
  - Example 4-49, 4-51
  - Force upper 3-4
- Display code option
  - Character set A-1
  - Default mode 2-13
  - Option 4-27
- Divide instructions 8-42, 8-43
- DM instruction 8-52
- Dollar sign
  - Local symbol separator 5-31
  - Parameter separator 5-8, 5-13, 5-16, 5-24, 5-28
  - Special element 2-5
- Double precision instructions 8-38, 8-40
- DUP pseudo
  - Description 5-6
  - Example 5-10, 5-11
  - Listing of count 11-6
- Duplication
  - Code 5-6
  - Echoed 5-7
  - Indefinite 5-7, 5-9
- DXi instructions
  - Add 8-38
  - Multiply 8-40
- E code option 4-27
- E entry point flag 11-14
- E error 11-10
- E list option 4-74
- E numeric data modifier 2-17
- ECHO pseudo
  - Description 5-7
  - Example 5-12
- ECS blocks 4-34
- Editing 2-4
- EE numeric data modifier 2-17
- EIM instruction 9-21
- EJECT pseudo 4-76
  - Permissible anywhere 4-2
- Eject suppression 10-4
- EJM instruction 9-19
- Element
  - Absolute 2-24
  - Data 2-11
  - Expression 2-23, 2-26
  - External 2-26
  - Operator 2-23
  - Register 2-26
  - Relocatable 2-9, 2-25
  - Special 2-9, 2-23
- ELSE pseudo
  - Description 4-60
  - Example 5-5
  - Permissible anywhere 4-2
- END pseudo
  - Assembly of remote code 5-3
  - Binary generation 3-6
  - Description 4-4
  - Effect on blocks 3-1, 3-6, 3-8, 3-10, 3-12

END pseudo (Contd)  
   Example 4-4, 5-7, 5-13, 5-14, 5-16  
   External text use 5-3  
   Force upper 3-4  
   Illegal definitions 5-1  
   Permissible anywhere 4-2  
 ENDD pseudo  
   Acting as nil 6-6  
   Description 5-10  
   Example 5-11  
   Permissible anywhere 4-2  
   Used with DUP 5-7  
   Used with ECHO 5-8  
 ENDIF pseudo  
   Acting as nil 6-6  
   Description 4-59  
   Permissible anywhere 4-2  
 ENDM pseudo  
   Acting as nil 6-6  
   Description 5-14  
   Example 4-31, 5-11, 5-15, 5-19, 5-20, 5-21  
   Permissible anywhere 4-2  
 End-of-line mark 5-1  
 ENDX pseudo 4-79  
 Entry address  
   Absolute 4-3  
   Declaration 4-45  
   Multiple 3-12  
   Relocatable 4-4  
 ENTRY pseudo  
   Description 4-45  
   Example 4-5, 4-46  
 Entry point list 11-4  
 ENTRYC pseudo 4-45  
 Environment test 4-60  
 EOM instruction 9-21  
 EQ IF operator 4-62  
   IFC operator 4-68  
 EQ instruction  
   Description 8-26  
   Example 8-27  
   Force upper 3-4  
 EQU pseudo  
   Description 4-41  
   Example 2-19, 2-21, 4-19, 4-39, 4-41, 4-64, 5-6  
   Listing 11-6  
 Equal sign  
   Default symbol prefix 2-7  
   Instruction 4-41  
   Literals prefix 2-11, 2-13, 2-17  
   Local symbol separator 5-31  
   Parameter separator 5-8, 5-13, 5-16, 5-25, 5-28  
 ERN instruction 9-14  
 ERR pseudo  
   Description 4-71  
 Error, assembly  
   Fatal 11-9  
   Informative 11-12  
   Programmer controller 4-71, 4-72  
 Error directory  
   Detailed description 11-9  
   General description 4-73  
 Error exit instruction 8-14  
 Error flags  
   Conditionally set 4-71  
   Fatal 11-9  
   Informative 11-12  
   Unconditionally set 4-72  
   Where on listing 11-6  
 ERRxx pseudo 4-72  
 ES instruction 8-14  
 ESN instruction 9-25  
 ETN instruction 9-14  
 Evaluation of expression 2-26  
 Exchange exit instruction 8-18  
 Exchange jump instruction 8-17  
 Execution, CPU program 1-3  
 EXN instruction 9-12  
 Exponent 2-17  
 Expression  
   Absolute 2-24  
   Attribute 4-66  
   Comparison 4-62  
   CON use 4-54  
   Description 2-23  
   Evaluatable 2-26  
   Evaluation 2-21, 2-27, 3-3  
   Examples 2-24, 2-55  
   External 2-26  
   Maximum size 2-27  
   Operators 2-23  
   Pass one value 2-27, 3-3  
   Pass two value 2-27, 3-3  
   Register 2-26, 8-9  
   Relocatable 2-25  
   Rules 2-22  
   Size 2-26  
   Types 2-24  
   Value 2-23, 2-26, 3-3, 8-5  
   VFD 4-53  
 EXT attribute 4-66  
 EXT pseudo  
   Description 4-47  
   Illegal in absolute code 4-6, 4-9, 4-10  
 External BCD  
   Character set A-1  
   Option 4-27  
 External symbol  
   Declaration 4-47  
   Description 2-5  
   Strong 2-7  
   Weak 2-7  
 External symbol list 11-4  
 External text  
   Assembly 5-2  
   File declaration 10-3  
   Listing 4-79  
 F conditional flag 11-14  
 F error 11-10  
 F FORTRAN mode 10-3  
 F list option 4-74  
 FAN instruction 9-24  
 Fatal error flag 11-9  
 Features of COMPASS 1-2  
 Field  
   Comments 2-2, 4-31  
   Conventional 2-3  
   Delimiter 2-1, 2-2  
   Destination 2-25, 4-53  
   Free 2-1  
   Length, threshold 1-3  
   Location 2-1  
   Operation 2-1  
   Size 2-1  
   Subfield 2-2  
   Terminator 2-1  
   Variable 2-2  
 File  
   COMPILE 10-3  
   INPUT 10-3  
   LGO 10-2  
   List output 10-3  
   Load and go 10-2  
   OLDPL 10-5  
   OPL 10-5  
   OUTPUT 10-3

## File (Contd)

- Source 10-3
- SYSTEXT 4-17, 10-3, 10-4, 10-5
- System text overlay 10-5
- Fill
  - Blank 2-14
  - Zero 2-14
- FIM instruction 9-21
- First column 2-1
- First statement group 4-2
- Fixed point data notation 2-17
- Fixed point instructions 8-39, 8-41
- FJM instruction 9-19
- Flag, error
  - Listing 11-6
  - Setting 4-71
  - Type 11-14
- Floating point data notation 2-16
- Floating point unit 8-3, 8-6
  - Add 8-37, 8-38
  - Divide 8-43
  - Multiply 8-39, 8-40
- FNC instruction 9-24
- FOM instruction 9-21
- Forcing upper 3-4
  - BSS 4-37
  - CPU instructions 8-2
  - LOC 4-38
  - Macro call 5-18, 5-25
  - Opdef call 5-27
  - ORG 4-35
  - ORGC 4-35
  - R= 4-55
  - USE 4-32
  - USELCM 4-34
  - VFD 4-53
- Form, COMPASS coding 2-3
- Format
  - Control statement 10-1
  - CPU instruction 8-1
  - Line 2-1
  - Listing 11-1
  - PP instruction 9-1
- FORTTRAN 4-4, 10-3
- Full list 10-3
- Functional units 8-3, 8-6, 8-10
- Functions, PP
  - Arithmetic 9-6
  - Data transmission 9-6
  - Logical 9-6
  - Replace 9-7
- FXI instruction
  - Add 8-37
  - Divide 8-42
  - Multiply 8-39
- G assembly mode 10-3
- G list option 4-74
- GE IF operator 4-62
  - IFC operator 4-68
- GE instructions 8-26
- Generated code listing 4-74
- Generation, data 4-48
- Get text mode 10-3
- Glossary F-1
- GT IF operator 4-69
  - IFC operator 4-74
- GT instruction 8-26
- Guaranteed zero 2-14, 4-50
- Hardware configuration 1-3
- Hardware feature dependency 4-7

## Heading

- Information 11-1
- Listing 4-73, 11-1
- Macro 5-13
- Opdef 5-13
- Page 11-1
- HERE pseudo
  - Description 5-4
  - Permissible anywhere 4-2
- Hexadecimal data 2-22
- I code option 4-21
- I hardware feature code 4-8
- I input mode 10-3
- I NOLABEL option 4-21
- IAM instruction 9-22
- IAN instruction 9-22
- IBj instruction 8-22
- ID instruction 8-24
- IDENT pseudo
  - Binary generation 3-8 thru 3-10
  - Blank variable field 3-14, 4-11
  - Description 4-2, 4-11
  - Example 4-4, 4-7, 4-13, 4-14, 4-16, 4-17, 4-19
  - Force upper 3-4
  - Overlay generation 3-8, 3-9, 3-10
  - Program identification 4-2
- IF pseudo 4-65
- IF skipped lines listed 4-74
- IFC pseudo
  - Description 4-68
  - Example 5-5, 5-11
  - Permissible anywhere 4-2
- IFCP pseudos 4-61
- IFOP pseudo 4-62
- IFPP pseudo 4-61
- IFtype pseudo 4-61
- IJM instruction 9-19
- IM instruction 8-51
- Increment unit 8-3, 8-6, 8-44, 8-46, 8-48
- Index register 8-7
- Indexed address, PP 9-16
- Indirect address, PP 9-15
- Input, assembler 10-3
- Instructions
  - Coding of 2-1
  - CMU 8-50
  - CPU 8-1
  - Mnemonically identified 6-3
  - Nil 6-6
  - No-operation 8-43, 9-11
  - PP 9-1
  - Pseudo 4-1
  - Redefinition 5-16, 5-25
  - Synonymous 6-5, 6-10
  - Syntactically identified 6-7
- Integer add 8-39
- Integer subtract 8-39
- Integer multiply 8-41
- Integer value 2-17
- Interactive debugging 1-4
- Internal BCD
  - Character set D-1
  - Option 4-27
- Invented symbol 5-32, 11-8
- IR instruction 8-24
- IRM instruction 9-21
- IRP pseudo
  - Acting as nil 6-6
  - Description 5-33
  - Example 5-34, 5-35
  - Permissible anywhere 4-2
- IXI instructions 8-39, 8-41

J option 4-9, 4-10, 9-8  
 JDATE micro 7-6  
 Job statement 10-1  
 JP instruction  
     Description 8-23  
     Force upper 3-5

L control statement option  
     Description 10-3  
     Related to LIST 4-74  
 L error 11-11  
 L hardware feature code 4-8  
 L list option 4-74  
 L location flag 4-38, 11-14  
 Labeled common  
     Description 3-2  
     Establishment 4-32, 4-34  
 LCC pseudo  
     Description 4-23  
     Illegal if absolute 4-6, 4-9, 4-10  
 LCM attribute 4-66  
 LCM blocks 3-2, 4-34  
 LCM transfer instructions 8-16, 8-19  
 LCN instruction  
     Data transmission 9-6  
     Description 9-10  
 LDC instruction  
     Data transmission 9-6  
     Description 9-11  
     Example 2-20  
 LDD instruction  
     Data transmission 9-6  
     Description 9-15  
 LDI instruction  
     Data transmission 9-6  
     Description 9-15  
 LDM instruction  
     Data transmission 9-6  
     Description 9-16  
     Example 5-21  
 LDN instruction  
     Data transmission 9-6  
     Description 9-10  
     Example 5-12, 9-10  
 LDSET pseudo  
     Description 4-21  
     Permissible anywhere 4-2  
 LE IF operator 4-62  
     IFC operator 4-68  
 LE instruction 8-26  
 Left shift instruction 8-31, 8-32  
 LGO control statement 10-6  
 Library maintenance programs 2-1  
 Linkage symbols 2-6, 4-45  
 List, full 10-3  
 List, parameter  
     ECHO 5-8  
     Equivalenced macro 5-25  
     Macro 5-18  
 LIST pseudo  
     Description 4-73  
     Example 4-13, 5-6, 5-12  
     Permissible anywhere 4-2  
 List, short 10-4  
 Listable output  
     Assembled code 11-5  
     Assembler statistics 11-8  
     Binary control cards 11-1  
     Block usage 11-2  
     Control statement 10-3  
     Default symbols 11-8  
     Entry point symbols 11-4

Listable output (Contd)  
     Error directory 11-9  
     Error flags 11-9 thru 11-12  
     External symbols 11-4  
     Header information 11-1  
     Literals 11-7  
     Octal 11-5  
     Source statements 11-5  
     Statistics 11-8  
     Subtitles 11-1  
     Symbolic reference table 11-12  
     Titles 11-1  
     User control 4-79, 10-3, 10-4  
 Listing control  
     Control statement 10-3, 10-4  
     Pseudo 4-73  
 LIT pseudo  
     Description 4-51  
     Example 2-12, 2-17, 2-21, 4-15, 4-58, 5-6  
     Listing 11-6, 11-7  
 Literals  
     Absolute program 3-6, 3-7, 3-10, 3-11  
     Description of block 3-1, 3-2  
     IDENT 3-10, 3-14  
     Listing 11-7  
     Location 1-3, 3-1, 3-2  
     Notation 2-12  
     Protection 4-35  
     SEGMENT overlay 3-10  
     SEG partial binary 3-12  
     Symbol (default) 2-7  
 LJM instruction  
     Description 9-6  
     Example 5-21  
 LMC instruction  
     Description 9-11  
     Logical function 9-6  
 LMD instruction  
     Description 9-15  
     Logical function 9-6  
 LMI instruction  
     Description 9-15  
     Logical function 9-6  
 LMM instruction  
     Description 9-16  
     Logical function 9-6  
 LMN instruction  
     Description 9-10  
     Logical function 9-6  
 LO control statement option 10-4  
 Load address 4-3  
 Load-and-go file 1-3, 10-2  
 Loader control statement 4-21  
 LOC attribute 4-66  
 LOC pseudo  
     Description 4-38  
     Example 4-39, 4-55  
     Location counter changed 3-4  
 Local blocks 3-2  
     Absolute program 3-6  
     Description 3-2  
     Establishment 4-32, 4-34  
     Relocatable program 3-5  
 LOCAL statement  
     Description 5-31  
     Example 5-32  
     Heading 5-13  
 Local symbol  
     Macro body 5-13  
     Subprogram 3-1, 4-29  
 Location counter  
     BSS 4-37  
     Control 4-38

Location counter (Contd)  
   Description 3-4  
   Forced upper 3-4  
   ORG 4-35  
   ORGC 4-35  
   Special element 2-9, 3-4  
   USE 4-32  
   USELCM 4-34  
 Location field  
   Listing 11-6  
   Statement 2-1  
 LO control card option  
   Description 10-4  
   Related to LIST 4-73  
 Logical difference instruction 8-29  
 Logical functions, PP 9-6  
 Logical minus 2-22  
 Logical product and complement instruction 8-30  
 Logical product instruction 8-28  
 Logical shift instruction 8-31, 8-32  
 Logical sum instruction 8-28  
 Long add unit  
   Description 8-3, 8-4, 8-6  
   Instructions 8-39  
 LPC instruction  
   Description 9-11  
   Logical function 9-6  
 LPN instruction  
   Description 9-10  
   Logical function 9-6  
 LRD instruction 9-12  
 LT IF operator 4-62  
   IFC operator 4-66  
 LT instruction 8-26  
 LXi instruction  
   Description 8-31, 8-32  
   Example 2-19  
  
 M base option 4-25  
 M list option 4-74  
 MACHINE pseudo 4-7  
 Machine test 4-60  
 Macro  
   Body 5-13  
   Call 5-18, 5-25  
   Equivalenced 5-24  
   Definition 5-13  
   Header 5-14  
   List control 4-74  
   Name 2-2, 5-15, 5-18, 5-25, 6-1  
   Permissible anywhere 4-2  
   Processing 5-1, 5-14  
   System defined 4-75, 5-35  
   Terminator 5-14  
 MACRO pseudo  
   Description 5-15  
   Example 4-31, 4-76, 5-5, 5-19 thru 5-22,  
     5-32 thru 5-34  
   IRP related 5-33  
   Operation code table entry 6-1  
   Permissible anywhere 4-2  
 MACROE pseudo  
   Description 5-24  
   Example 5-26  
   IRP related 5-33  
   Operation code table entry 6-1  
   Permissible anywhere 4-2  
 MAN instruction 9-12  
 Mask instruction 8-42  
 Mass storage, system 1-3  
 Master list control 4-73  
  
 MAX pseudo  
   Description 4-42  
   Listing 11-6  
 MD instruction 8-51  
 MESSAGE macro 12-25  
 MI instruction 8-24, 8-26  
 MIC attribute 4-67  
 MICCNT pseudo  
   Description 4-44  
   Example 4-44  
   Listing 11-6  
   Permissible anywhere 4-2  
 Micro  
   Decimal 7-4  
   Definition 4-24, 4-27, 4-28, 7-2  
   Editing 2-4  
   Mark 2-4, 5-1  
   Octal 7-4  
   Predefined names 7-5  
   Reference 7-1  
   Size 4-44, 7-2  
   Substitution 7-1  
   System defined 4-17, 7-2, 7-5  
   Test for 4-67  
 MICRO pseudo  
   Description 7-2  
   Example 4-44, 5-11, 7-2, 7-3  
   Permissible anywhere 4-2  
 MIN pseudo  
   Description 4-43  
   Listing 11-6  
 Minus as local symbol separator 5-31  
 Minus as parameter separator 5-8, 5-13, 5-16,  
   5-24, 5-28  
 Minus on listing 11-6  
 Minus operator 2-21, 2-22, 8-11, 8-12  
 Minus sign in location field  
   CPU instruction 3-4, 3-5, 4-53  
   PP instruction 3-4, 4-53  
   VFD instruction 4-53  
 MJ instruction 8-18  
   Force upper 3-4  
 MJN instruction  
   Description 9-6  
   Effect of J 4-9, 4-11  
 ML control statement option 10-4  
 Mnemonic operation code  
   Legal operation field entry 2-1  
   OPDEF defined 5-27  
   Search for 6-1  
 Modifiers, numeric data 2-17  
 MODIFY common decks 5-2  
 MODLEVEL micro 7-7  
 MOVE macro 12-28  
 Multiple entry point table  
   Suppression 4-20  
   Used for overlays 3-12  
 MXi instruction  
   Description 8-42  
   Example 2-19, 8-42  
 MXN instruction 9-12  
  
 N eject mode 10-4  
 N error 11-11  
 N list option 4-75  
 Name  
   Block 4-32, 4-34  
   Different types 2-4  
   Duplicate code 5-7, 5-8  
   General description 2-4  
   IF sequence 4-59



Name (Contd)

- Macro 5-16
- Micro 4-24, 4-27, 4-28, 7-2, 7-4, 7-5
- Mnemonic operation 6-1
- Overlay 4-11, 4-15
- Parameter 5-8
- Remote code 5-3
- NE IF operator 4-62
- IFC operator 4-68
- NE instruction 8-26
- Nesting, level of 1-3
- NG instruction 8-24, 8-26
- NIL pseudo 6-6
- Permissible anywhere 4-2
- NIM instruction 9-21
- NJN instruction
  - Description 9-7
  - Effect of J 4-9, 4-10
- NO eject option 10-4
- NO instruction 8-43
- NOLABEL pseudo
  - Description 4-20
  - Permissible anywhere 4-2
- NOM instruction 9-21
- NOREF pseudo 4-78
- Permissible anywhere 4-2
- Normalize instruction 8-34
- Normalize unit
  - Description 8-6
  - Instructions 8-34, 8-35
- Not equal sign
  - Parameter separator 5-8, 5-13
  - Special character 2-4
- Numeric data 2-17
- NXi instruction 8-34
- NZ instruction 8-24, 8-26

O base 2-18, 4-24

O error

- Description 11-11
- With AIDTEXT 8-1, 9-1

O mode 10-4

OAM instruction 9-22

OAN instruction 9-22

OBj instruction 8-22

Octal listing 11-5

Octal notation 2-16

OCTMIC pseudo 7-4

- Permissible anywhere 4-2

OLDPL file 10-3

Opdef

- Body 5-13
- Call 5-29
- Definition 5-13
- Heading 5-13
- List control 4-72, 4-73
- Processing 5-14
- System defined 4-17, 4-33

OPDEF pseudo

- Description 5-27
- Example 5-29 thru 5-32
- Operation code table entry 6-1
- Permissible anywhere 4-2

Operand register 8-7

Operation code table 6-1

Operation code value

- CPU 6-7, 8-1
- PP 6-3, 9-1

Operation, definition

- Compressed 5-1
- Duplicated text 5-6
- External text 5-2
- General description 5-1

Operation, definition (Contd)

- Macro definition 5-13
- Opdef definition 5-13
- Remote text 5-3
- System 5-35

Operation field

- Blank 4-48
- Description 2-1
- Search 6-1

Operator

- Element 2-22
- Mnemonic 5-27, 6-3
- Register 2-21, 5-28, 6-7
- Term 2-22

Operator with constant 2-13, 2-16

OPL file 5-2, 10-3, 12-1

OPSYN pseudo

- Description 6-5
- Permissible anywhere 4-2

OR instruction 8-24

ORG pseudo

- Description 4-35
- Determine blocks 3-1
- Establish absolute blocks 3-2, 4-35
- Example 4-4, 4-7, 4-13, 4-14, 4-16
- Location counter changed 4-35
- Origin counter changed 3-3, 4-35

ORGC pseudo 4-35

Origin

- Multiply entry point 4-3
- Overlay 4-12, 4-15
- Program 4-3

Origin counter

- BSS 4-37
- Control 3-3, 4-35, 4-37
- Description 3-3
- Final value, absolute 3-6
- Final value, relocatable 3-5
- Forced upper 3-4
- ORG 4-35
- ORGC 4-35
- Special element 2-9, 3-3
- USE 4-32

ORM instruction 9-21

Overflow error 2-17

Overlay

- Absolute 3-8
- Control tables 4-21
- Entry point 4-12, 4-15
- General description 3-6, 3-8
- Level numbers 4-4, 4-12, 4-15
- Multiple entry point 3-12
- Name 4-12, 4-15
- Origin 4-12, 4-15
- PP 3-7, 3-9
- Primary 3-8, 3-9, 3-11, 3-13, 4-12, 4-15
- Secondary 3-6, 3-8, 3-9, 4-12, 4-15

P error 11-11

P numeric data modifier 2-17

P pagination mode 10-4

Pack instruction 8-36

Padding of CPU word 3-4, 4-53, 8-2

Page heading 11-1

Page number 11-1

Pagination control 10-4

Parameter

- Actual 5-7, 5-18, 5-25
- Embedded 5-18, 5-25
- Formal 5-8, 5-13
- Indefinitely repeated 5-34
- Iterative 5-18, 5-25, 5-34
- Substitutable 5-8, 5-13, 5-16, 5-25, 5-28, 5-34

Parameter mark 5-9, 5-13  
 Parameter, null 5-9, 5-18, 5-25  
 Parameter separator  
   Actual 5-18, 5-25  
   Formal 5-8, 5-13, 5-16  
 Parcel 8-1  
 Parentheses  
   Local symbol separator 5-31  
   Nested 5-9  
   Parameter separator 5-8, 5-13, 5-16, 5-25, 5-28  
 Partial binary  
   IDENT type 3-14  
   SEG type 3-12  
 Pass instruction  
   CPU 8-43  
   PP 9-9  
 Pass one  
   Expression evaluation 2-23, 2-26, 2-28, 3-3  
   General description 1-3  
   Maximum test 4-42  
   Minimum test 4-43  
   Symbol definition 2-6  
 Pass two  
   Expression evaluation 2-22, 2-26, 3-3  
   General description 1-3  
   Symbol definition 2-5  
   Value for MAX 4-42  
   Value for MIN 4-43  
 PC control statement option 10-4  
 PCOMMENT micro 7-7  
 PD control statement option 10-4  
 PERIPH pseudo  
   Description 4-10  
   Effect on branch instructions 9-8  
   Example 4-49, 6-5  
   First statement group 4-2  
 PJN instruction  
   Description 9-7  
   Effect of J 4-9, 4-10  
 PL instruction 8-24, 8-26  
 Plus as local name separator 5-31  
 Plus as parameter separator 5-8, 5-13, 5-16, 5-25, 5-28  
 Plus in location field  
   CPU instruction 3-4  
   PP instruction 3-5  
   VFD instruction 4-53  
 Plus on listing 11-6, D-2, D-3  
 Plus operator 2-21, 2-23, 8-11  
 Point  
   Binary 2-18, 2-19  
   Decimal 2-18, 2-19  
   Octal 2-18, 2-19  
   Parameter separator 5-8, 5-13, 5-16, 5-25, 5-28  
   Register designator 2-8  
 Population unit 8-43  
 POS pseudo 4-40  
 Position counter  
   Control 4-40, 4-53  
   Description 3-4  
   Special element 2-9, 3-4  
 Post radix 2-17  
 PP instructions 9-1  
   A-register I/O 9-22  
   Block I/O 9-22  
   Branch I/O 9-19, 9-21  
   Branch 9-7  
   Central read/write 9-18  
   Channel function 9-24  
   Constant mode 9-11  
   Designators 9-3  
   Direct address 9-15  
   Error stop 9-25  
   Exchange jump 9-12  
   Format 9-1  
   Functions 9-6  
   Indexed direct address 9-16  
   Indirect address 9-15  
   No address 9-10  
   No operation 9-11  
   Output record flag 9-23  
   Shift 9-9  
 PPOP  
   Description 6-3  
   Example 5-12, 6-5  
   Permissible anywhere 4-2  
 PPU pseudo  
   Description 4-8  
   Effect on branch 9-8  
   Example 4-10, 4-54  
   First statement group 4-2  
 Prefix table  
   Comments 4-20  
   Generation 3-6 thru 3-8  
   Suppression 4-21  
 Preradix 2-17  
 Program, absolute 3-6, 4-6  
 Program example D-1 thru D-8  
 Program execution 10-5  
 Program identification 4-2  
 Program origin 4-3  
 Program, relocatable 3-5  
 Program stop instruction 8-13  
 Program structure 3-1  
 PS control statement option 10-4  
 PS instruction  
   Description 8-13  
   Force upper 3-4  
 Pseudo instructions  
   Binary control 4-6  
   Block counter control 4-32  
   Conditional assembly 4-59  
   Data generation 4-47  
   Definition operation 5-1  
   Error control 4-71  
   First statement group 4-2  
   Introduction 4-1  
   Listing control 4-73  
   Micro 7-1  
   Mode control 4-24  
   Operation code table management 6-1  
   Operation field entry 2-2  
   Permissible anywhere 4-2  
   Required 4-2  
   Subprogram identification 4-2  
   Subprogram linkage 4-45  
   Symbol definition 4-40  
   Types 4-1  
 PSN instruction 9-11  
 PURGDEF pseudo  
   Description 6-10  
   Permissible anywhere 4-2  
 PURGMAC pseudo  
   Description 6-7  
   Example 6-6  
   Permissible anywhere 4-2  
 Push down stack 1-3  
 PXi instruction 8-36  
  
 Q to represent expression 5-27, 6-8  
 QUAL micro 7-6  
 QUAL pseudo  
   Description 4-28  
   Example 4-13, 4-30, 5-22  
   Permissible anywhere 4-2  
 Qualifier, symbol 4-28  
   Used for definition operations 5-2

R error 11-11  
R hardware feature code 4-8  
R list option 4-75  
R register 9-2, 9-3  
RAD instruction  
  Description 9-15  
  Replace function 9-7  
Radix 2-17  
RAI instruction  
  Description 9-15  
  Replace function 9-7  
RAM instruction  
  Description 9-16  
  Replace function 9-7  
RE instruction  
  Description 8-15  
  Force upper 3-4  
Read central memory instruction 8-46, 9-17  
Real-time clock set instruction 8-21  
Record name, external text 5-3  
Recursion level 1-4, 5-1  
Recursion stack 1-4, 5-1  
Reference  
  Macro 5-18  
  Macroe 5-24  
  Nested 5-1  
  Opdef 5-27  
Reference table, symbolic 11-13  
Register designators  
  CPOP 6-7  
  Description 2-8, 8-7  
  Not symbols 2-5  
  OPDEF 5-27  
  OPSYN 6-5  
  PURGDEF 6-10  
Registers, CPU 2-8, 8-7  
READC macro 12-28  
READH macro 12-28  
READO macro 12-29  
READS macro 12-29  
READW macro 12-29  
RECALL macro 12-30  
REL attribute 4-66  
Relocatable program structure 3-5  
Relocatable test 4-66  
Relocation, CM access 9-2  
Relocation register  
  Description 9-2, 9-3  
  Load and store instructions 9-12  
Remote assembly 5-3  
REP pseudo 4-57  
REPC pseudo 4-57  
Repeat count  
  DUP 5-7  
  Replication 4-57  
REPI pseudo  
  Example 4-57  
  Description 4-57  
  Illegal if absolute 4-6, 4-9, 4-10  
REPL table  
  Result of BSSZ 4-48  
  Result of REP, REPC, or REPI 4-57  
  Written by SEGMENT 4-15  
Replace functions, PP 9-7  
Replication of code 4-57  
Return jump, CPU 8-14  
RFN instruction 9-23  
RI instruction 8-21  
Right shift 8-32, 8-33  
RJ instruction  
  Description 8-14  
  Example 4-33, 5-21, 8-15  
  Force upper 3-5  
RJM instruction 9-7  
RL instruction 8-16  
RMT pseudo  
  Description 5-3  
  Example 5-5, 5-6  
  Permissible anywhere 4-2  
RO instruction 8-22  
Round and normalize instruction 8-35  
RPN instructions 9-13  
RXi instructions  
  Add 8-38  
  Divide 8-43  
  Multiply 8-40  
RXj instruction 8-20  
R= pseudo  
  Description 4-55  
  Example 4-56, 5-21  
  Illegal in PP program 4-9, 4-10  
  
S list option 4-75  
S numeric data modifier 2-18  
S storage flag 11-14  
S system text mode 10-5  
SAi instructions  
  Description 8-44  
  Example 2-15, 2-16, 2-19, 4-33, 4-38, 5-22, 33, 8-45  
SBD instruction  
  Arithmetic function 9-6  
  Description 9-15  
SBI instruction  
  Arithmetic function 9-6  
  Description 9-15  
Sbi instructions  
  Description 8-46  
  Example 2-9, 2-12, 8-47  
SBM instruction  
  Arithmetic function 9-6  
  Description 9-16  
SBN instruction  
  Arithmetic function 9-6  
  Description 9-10  
Scale, binary 2-18  
SCF instruction 9-20  
SCM blank common 3-3  
SCM labeled common 3-2  
SCN instruction  
  Description 9-10  
  Logical function 9-6  
SEG pseudo  
  Binary generation 3-12  
  Description 4-15  
  Example 4-16  
  Force upper 3-4  
  Illegal in PP program 4-9, 4-10  
SEGMENT pseudo  
  Binary generation 3-8 thru 3-10, 3-12  
  Description 4-16  
  Example 4-17  
  Force upper 3-4  
  Illegal in PP program 4-9, 4-10  
  Overlay structure 3-10, 3-12  
Semicolon in definition 5-8, 5-13  
SEQUENCE micro 7-7  
Sequencing  
  Listing 11-7  
  Statement 2-1  
SET attribute 4-66  
Set register instructions 8-44 thru 8-49  
SET pseudo  
  Description 4-41  
  Example 2-9, 2-20, 5-11, 5-22  
  Listing 11-6  
SFM instruction 9-20

- Shift
  - Description of unit 8-3, 8-6
  - CPU instructions 8-31 thru 8-33
  - PP instruction 9-9
- SHN instruction 9-9
- Short jump limit 4-9, 4-11
- Short list 10-4
- Single precision instructions
  - Add rounded 8-38
  - Add unrounded 8-37
  - Divide rounded 8-43
  - Divide unrounded 8-42
  - Multiply rounded 8-40
  - Multiply unrounded 8-39
- SKIP pseudo
  - Description 4-70
  - Permissible anywhere 4-2
- Slant bar
  - Local symbol separator 5-31
  - Operator 2-22
  - Parameter separator 5-8, 5-13, 5-16, 5-24, 5-28
- SOD instruction
  - Description 9-15
  - Replace function 9-7
- SOI instruction
  - Description 9-15
  - Replace function 9-7
- SOM instruction
  - Description 9-16
  - Replace function 9-7
- Space, embedded (see blank)
- SPACE pseudo
  - Description 4-76
  - Permissible anywhere 4-2
- Special elements
  - FORTTRAN call 2-9
  - General description 2-9
  - In variable field 2-2
  - Location counter 3-4
  - Origin counter 3-3
  - Position counter 3-4
- SRD instruction 9-12
- SST attribute 4-67
- SST pseudo 4-45
  - Example 4-13
  - Permissible anywhere 4-2
- Stack, recursion 1-4, 5-1
- Statement
  - Coding conventions 2-3
  - Comments 2-2
  - Compressed 5-1
  - Continuation 2-2
  - External source 5-2
  - First column 2-1
  - First group 4-1
  - Format 2-1
  - Listing 11-5
  - Number assembled 11-8
  - Size 2-1
  - Source of 5-1, 10-3
- Statistics, assembler 11-8
- STD instruction
  - Data transmission function 9-6
  - Description 9-15
- STEXT pseudo
  - Description 4-17
  - Example 4-19
  - First statement group 4-2
- STI instruction
  - Data transmission function 9-6
  - Description 9-15
- STM instruction
  - Data transmission function 9-6
  - Description 9-16
- STOPDUP pseudo
  - Description 5-9
  - Example 5-11
- Storage reservation 4-37, 4-48
- String, character
  - Comparison 4-68
  - Data generation 4-49
  - Delimited 2-11, 2-14
  - Empty 2-14
  - Micro 2-4
  - Notation 2-13
- Strong external 2-7
- Subprogram length 3-5
- Substitution, micro 7-1
- Subsubtitle
  - CTEXT 4-79
  - EJECT 4-76
  - Listing of 11-1
  - QUAL 4-28
  - SPACE 4-76
  - TITLE 4-77
  - TTL 4-78
- Subtitle
  - CTEXT 4-79
  - Listing of 11-1
  - TITLE 4-77
- SXi instruction
  - Description 8-48
  - Example 2-15, 2-19, 5-21, 5-31, 8-49
- Symbol
  - Attribute 2-6, 4-40, 4-66
  - Created 5-32
  - Default 2-7
  - Definition 2-5, 4-40
  - Duplicate 2-6
  - Entry point 2-6
  - External 2-7
  - Invented 5-32, 11-8
  - Literals 2-6
  - Local to macro 5-13, 5-31
  - Local to QUAL 3-1
  - Location field 2-6
  - Lost 11-8, 11-13
  - Number defined 11-8
  - Number referenced 11-8
  - Previously defined 2-7
  - Qualified 2-7, 4-27
  - Redefinition 4-29, 4-41
  - System-defined 2-6, 4-45
  - Undefined 2-7
  - Value 2-6, 4-39
- Symbol qualifier listed 11-1
- Symbol table
  - Clearing 3-10, 3-12
  - System text 4-17
- Symbolic notation 8-1, 9-1
- Symbolic reference table
  - Address reference 4-80
  - Detailed description 11-12
  - General description 4-73
  - Generation 1-3
  - List control 4-73, 10-3
  - Omit symbol 4-78
- Synonymous operation
  - CPU 6-10
  - Mnemonic 6-5
  - PP 6-5
  - Syntactic 6-7
- Syntax definition 5-27, 6-7, 6-10
- Syntax search 6-1
- SYSTEM macro 12-30
- System text 4-19
- SYSTEXT option 10-4
  - Related to G mode 10-4
  - Related to STEXT 4-17

T list option 4-73  
 Table  
     Operation code 6-1  
     Symbolic reference 11-12  
     USE 4-32  
 Tbj instruction 8-21  
 Term 2-22  
 Term operator 2-22  
 Terminator, macro 5-13  
 Test symbol attribute 4-66  
 Time limit 10-1  
 TIME micro 7-6  
 Time of assembly 11-1  
 Title  
     ES 8-14  
     IDENT 4-3  
     Listing of 11-1  
     PS 8-13  
     TITLE 4-77  
 TITLE pseudo 4-77  
     Permissible anywhere 4-2  
 Transfer symbol 4-4  
 Transmit instruction 8-27  
 Truncation, character data 2-13  
     Expression value 2-26  
 TTL pseudo 4-78  
     Permissible anywhere 4-2  
  
 U error 11-11  
 UEM  
     Block copy instructions 8-15  
     Direct transfer instructions 8-20  
 UJN instruction  
     Effect of J 4-9, 4-10  
     Description 9-7  
 Unconditional jump  
     CPU 8-23  
     PP 9-7  
 Underflow error 2-18  
 Unpack instruction 8-35  
 USE pseudo  
     Change blocks 3-1 thru 3-3, 3-5, 4-32  
     Description 4-32  
     Establish common blocks 3-2, 3-3, 4-32  
     Establish local blocks 3-2, 4-32  
     Example 4-17, 4-30, 4-31, 4-33, 4-36, 4-38  
 USE table  
     Entry 4-32, 4-34, 4-35  
     Reinitialization 3-10, 3-12, 4-11  
 USELCM pseudo  
     Description 4-34  
     Establish common blocks 3-2, 3-3  
     Example 4-35  
     Illegal in PP program 4-9, 4-10  
 USER control statement 10-7  
 UXi instruction 8-35  
  
 V error 11-11  
 Value, numeric 2-17  
  
 Variable field 2-2  
 Variable field definition 4-53  
 VFD pseudo  
     Description 4-53  
     Example 2-15, 4-25, 4-30, 4-33, 4-54, 5-22  
  
 WE instruction  
     Description 8-15  
     Force upper 3-4  
 Weak external 2-7  
 WL instruction 8-16  
 Write central memory instruction 8-46  
 WRITEC macro 12-31  
 WRITEH macro 12-31  
 WRITEO macro 12-31  
 WRITES macro 12-32  
 WRITEW macro 12-32  
 WXj instruction 8-20  
  
 X external flag 4-47, 11-6  
 X external text mode 10-5  
 X file option  
     Description 10-5  
     XTEXT default 5-3  
 X hardware feature code 4-8  
 X list option 4-75  
 X register  
     Conditional instructions 8-24  
     Description 8-7  
     Designator 2-8  
     Setting 8-48  
 XJ instruction  
     Description 8-17  
     Force upper 3-4  
 XREF pseudo  
     Description 4-80  
     Permissible anywhere 4-2  
 XTEXT pseudo 5-1  
     Related to CTEXT/ENDX 4-79  
 XTEXT source 10-5  
  
 Zero block  
     Absolute program 3-2, 3-6, 3-7  
     Description 3-2  
     Relocatable program 3-5  
 Zero fill 2-14, 4-53  
 Zero guaranteed  
     Data item 2-14  
     DIS item 4-50  
 Zeroed words 4-48  
 ZJN instruction  
     Description 9-8  
     Effect of J 4-9, 4-10  
 ZR instruction  
     Description 8-24, 8-26  
     Force upper 3-4  
 ZXi Instruction 8-35



COMMENT SHEET

MANUAL TITLE: COMPASS Version 3 Reference Manual

PUBLICATION NO.: 60492600

REVISION: L

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments on the back (please include page number references).

\_\_\_\_\_ Please reply

\_\_\_\_\_ No reply necessary

FOLD

FOLD



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS      PERMIT NO. 8241      MINNEAPOLIS, MINN.

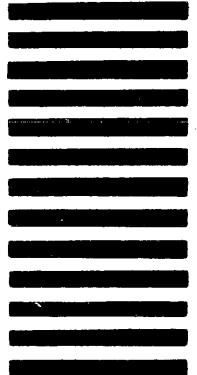
POSTAGE WILL BE PAID BY

**CONTROL DATA CORPORATION**

Publications and Graphics Division

P.O. BOX 3492

Sunnyvale, California 94088-3492



CUT ALONG LINE

FOLD

FOLD

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.  
FOLD ON DOTTED LINES AND TAPE

NAME:

COMPANY:

STREET ADDRESS:

CITY/STATE/ZIP:

TAPE

TAPE

# COMPASS VERSION 3 SUMMARY



Pub. No. 60492600 M 1984, 1986

CDC® OPERATING SYSTEMS: NOS 2, NOS/BE 1, SCOPE 2

## CONTROL STATEMENT

COMPASS (p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>) or COMPASS.

<b>A</b> omitted A	Do not abort Abort on assembly errors	<b>ML</b> omitted or ML ML=string	MODLEVEL returns JDATE MODLEVEL returns 9-character string
<b>B</b> omitted or B B=0 B=ifn	Binary on LGO No binary Binary on ifn	<b>N</b> omitted or O N	Normal ejects No ejects
<b>BL</b> omitted or BL=0 BL	Compact listing format Burstable listing format	<b>O</b> omitted or O O=ifn O=0	Short list on OUTPUT Short list on ifn No short list
<b>D</b> omitted D	No debug mode Debug mode	<b>P</b> omitted P	New pagination on END Continue pagination
<b>E</b> omitted E E=ifn E=0	Error list on OUTPUT Error list on ERRS Error list on ifn No error list	<b>PC</b> omitted or PC PC=string	PCOMMENT is 30 blanks PCOMMENT is 30-character string
<b>F</b> omitted or F F=n F=name	*F returns 0 *F returns n (decimal) *F returns as follows: COMPASS = 0 RUN = 1 FTN4 = 2 FTN5 = 3	<b>PD</b> omitted or other PD=6 PD=8	Print density default Print density 6 lines/inch Print density 8 lines/inch
<b>G†</b> omitted or G=0 G G=ifn G=ifn/ovl	No system text System text on SYSTEXT Overlay on ifn Named overlay on ifn	<b>S†</b> omitted S S=0 S=ovl S=lib/ovl	SYSTEXT overlay SYSTEXT on global library No system text Named overlay on library Named overlay on named library
<b>I</b> omitted I I=ifn	Source on INPUT Source on COMPILER Source on ifn	<b>X</b> omitted X=ifn X	XTEXT on OLDPL XTEXT on ifn XTEXT on OPL
<b>LO</b> omitted or LO=0 LO LO=\$\$\$\$ LO=c <sub>1</sub> c <sub>2</sub> ...c <sub>n</sub>	Selects B, L, N, and R Selects C, F, G, X Selects all list options Deselects if c <sub>i</sub> is B, L, N, or R; selects if c <sub>i</sub> is other	<b>†</b> Seven G and S parameters allowed	

## MODEL 72, 73, 74 EXIT MODES

Bit 50	Bit 49	Bit 48
INDEFINITE OPERAND	OPERAND OUT OF RANGE	ADDRESS OUT OF RANGE

## MODEL 76 PSD REGISTER

MODE						CONDITION											
Exit	Mon	Step	Ind	Ovf	Undf	LPar	SPar	LBlk	SBlk	LDir	SDir	Prog	Bkp	Step	Ind	Ovf	Undf
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

## FATAL ERRORS

<b>A</b> ADDRESS FIELD BAD
<b>B</b> DOUBLY DEFINED SYMBOL. THE FIRST DEFINITION HOLDS.
<b>E</b> ECHO, DUP, RMT, OR MACRO ILLEGALLY NESTED.
<b>F</b> NUMBER OF ENTRIES EXCEEDS PERMISSIBLE AMOUNT.
<b>L</b> LOCATION FIELD BAD.
<b>N</b> NEGATIVE RELOCATION ON ENTRY POINT.
<b>O</b> OPERATION FIELD BAD.
<b>P</b> CONSULT LISTING FOR REASON BEHIND P-ERROR.
<b>R</b> DATA ORIGIN OUTSIDE BLOCK OR IN BLANK COMMON BLOCK.
<b>U</b> UNDEFINED SYMBOL. VALUE ASSUMED 0.
<b>V</b> BIT COUNT ERROR ON VFD (MUST BE 0 ≤ COUNT ≤ 60).

## INFORMATIVE ERRORS

<b>1</b> LOCATION SYMBOL BAD. SYMBOL NOT DEFINED.
<b>2</b> ADDRESS ERROR ON SYMBOL DEFINITION.
<b>3</b> DUPLICATE MACRO DEFINITION. NEW ONE OVERRIDES.
<b>4</b> BAD FORMAL PARAMETER NAME IGNORED.
<b>5</b> CPU OPERATION SYNTAX INCORRECTLY SPECIFIED.
<b>6</b> LOCATION FIELD MEANINGLESS.
<b>7</b> ADDRESS VALUE EXCEEDS FIELD SIZE. RESULT TRUNCATED.
<b>8</b> MISSING OR EXTRA ADDRESS SUBFIELD.
<b>9</b> MICRO SUBSTITUTION ERROR. NO SUBSTITUTION.

## STANDARD CHARACTER SETS

CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code
†	:	00†	8-2	00	8-2	3A
A	A	01	12-1	61	12-1	41
B	B	02	12-2	62	12-2	42
C	C	03	12-3	63	12-3	43
D	D	04	12-4	64	12-4	44
E	E	05	12-5	65	12-5	45
F	F	06	12-6	66	12-6	46
G	G	07	12-7	67	12-7	47
H	H	08	12-8	68	12-8	48
I	I	09	12-9	69	12-9	49
J	J	10	11-1	41	11-1	4A
K	K	11	11-2	42	11-2	4B
L	L	12	11-3	43	11-3	4C
M	M	13	11-4	44	11-4	4D
N	N	14	11-5	45	11-5	4E
O	O	15	11-6	46	11-6	4F
P	P	16	11-7	47	11-7	50
Q	Q	17	11-8	50	11-8	51
R	R	18	11-9	51	11-9	52
S	S	19	0-2	22	0-2	53
T	T	20	0-3	23	0-3	54
U	U	21	0-4	24	0-4	55
V	V	22	0-5	25	0-5	56
W	W	23	0-6	26	0-6	57
X	X	24	0-7	27	0-7	58
Y	Y	25	0-8	30	0-8	59
Z	Z	26	0-9	31	0-9	5A
0	0	27	0	12	0	30
1	1	28	1	01	1	31
2	2	29	2	02	2	32
3	3	30	3	03	3	33
4	4	31	4	04	4	34
5	5	32	5	05	5	35
6	6	33	6	06	6	36
7	7	34	7	07	7	37
8	8	35	8	08	8	38
9	9	36	9	09	9	39
+	+	37	12	60	12-8-6	2B
-	-	38	11	40	11	2D
*	*	39	11-8-4	54	11-8-4	2A
/	/	40	0-1	21	0-1	2F
(	(	41	0-8-4	34	12-8-5	28
)	)	42	12-8-4	74	11-8-5	29
\$	\$	43	11-8-3	53	11-8-3	24
=	=	44	8-3	13	8-6	3D
blank	blank	45	no punch	20	no punch	20
(comma)	(comma)	46	0-8-3	33	0-8-3	2C
(period)	(period)	47	12-8-3	73	12-8-3	2E
≡	#	48	0-8-6	36	8-3	23
{	{	49	8-7	17	12-8-2	5B
}	}	50	0-8-2	32	11-8-2	5D
%	%	51	8-6	16	0-8-4	25
"	"(quote)	52	8-4	14	8-7	22
~	~(underline)	53	0-8-5	35	0-8-5	5F
!	!	54	11-0	52	12-8-7	21
&	&	55	0-8-7	37	12	26
^	^ (apostrophe)	56	11-8-5	55	8-5	27
?	?	57	11-8-6	56	0-8-7	3F
<	<	58	12-0	72	12-8-4	3C
>	>	59	11-8-7	57	0-8-6	3E
≤	≤	60	8-5	15	8-4	40
≥	≥	61	12-8-5	75	0-8-2	5C
∩	∩ (circumflex)	62	12-8-6	76	11-8-7	5E
;	;(semicolon)	63	12-8-7	77	11-8-6	3B

†Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons. End-of-line mark is converted to external BCD 1632.

††In installations using the CDC 63-graphic set, display code 00 has no associated graphic or Hollerith code; display code 63 is the colon (8-2 punch).

## LANGUAGE ELEMENTS

### SPECIAL CHARACTERS

**Any Field**  
≠ Micro substitution  
⇒ Concatenation

**First Column**  
\* Comments line  
+ Continuation

**Location Field**  
+ Force Upper  
- Negate Force Upper

**Variable Field**  
=item Literal  
=Ssym Deferred Symbol  
=Xsym Strong External Symbol  
=Ysym Weak External Symbol  
\*O Origin Counter  
\*L Location Counter  
\*P Position Counter  
\*S Caller: COMPASS 0  
RUN 1  
FTN4 2  
FTN5 3

/name/ Subfield Delimiter  
Symbol Qualifier

### CHARACTER NOTATION

**Data Item**  
sign n t string  
or  
sign t d string d

**Constant**  
n t string

**Literal**  
= sign n t string  
or  
= sign t d string d

t = C Left, 12 zero bits  
H Left, blank fill  
A Right, blank fill  
R Right, zero fill  
L Left, zero fill  
Z Left, 6 zero bits

d = delimiter character  
n = no. of characters  
string = Code characters

### NUMERIC NOTATION

**Data Item**  
sign prerx value mods

**Constant**  
value mods

**Literal**  
= sign prerx value mods

**Mods Spec Default**  
Sign + or - +  
Radix O or B BASE  
(pre/post) or D Pseudo  
Integer n 0  
Fraction .n or . none  
Pwr 10 sgl E or En none  
or E±n none  
Pwr 10 dbl EE or EE± none  
or EE±n none  
Pwr 2 S or Sn none  
or S±n none  
Binary pt P or Pn none  
or P±n none

### SYSTEM MICROS

DATE JDATE  
TIME TIME  
BASE BASE  
CODE CODE

QUAL SEQUENCE  
MODLEVEL  
PCOMMENT

## PSEUDO INSTRUCTIONS

### PROGRAM DEFINITION

IDENT name, fwa, eptsym  
sym END trasymp

### BINARY CONTROL

ABS  
MEMSEL  
MACHINE type, hf<sub>1</sub>, hf<sub>2</sub>, ..., hf<sub>n</sub>  
type = 6, 7, or 8 hf<sub>i</sub> = C, D, I, L, X  
PPU J  
PERIPH J  
CIPPU  
IDENT name, org, entry, v<sub>1</sub>, v<sub>2</sub>  
IDENT name, org, entry, ppu  
name SEGMENT fwa, eptsym  
SET  
record STEXT  
LDOSET p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>  
LCC directive  
COMMENT string  
NOLABEL I

### MODE CONTROL

mname BASE O or D or M or \*  
CODE A or D or E or I or \*  
QUAL name or \*  
B1=1  
B7=1  
COL n

### COUNTER CONTROL

USE \* or name or // or /name/  
USELCM \* or name or // or /name/  
ORG exp  
ORGC exp  
sym BSS exp  
LOC exp  
POS aexp

### SYMBOL DEFINITION

sym = exp  
sym EQU exp  
sym SET exp  
sym MAX exp<sub>1</sub>, exp<sub>2</sub>, ..., exp<sub>n</sub>  
sym MIN exp<sub>1</sub>, exp<sub>2</sub>, ..., exp<sub>n</sub>  
sym MICCNT mname  
sym SST sym...

### LINKAGE CONTROL

ENTRY sym<sub>1</sub>, sym<sub>2</sub>, ..., sym<sub>n</sub>  
ENTRYC sym<sub>1</sub>, sym<sub>2</sub>, ..., sym<sub>n</sub>  
EXT sym<sub>1</sub>, sym<sub>2</sub>, ..., sym<sub>n</sub>

### DATA GENERATION

sym BSSZ exp  
sym DATA item<sub>1</sub>, ..., item<sub>n</sub>  
sym DIS n, string  
sym DIS d, stringd  
sym LIT item<sub>1</sub>, ..., item<sub>n</sub>  
sym VFD item<sub>1</sub>/exp<sub>1</sub>, ..., exp<sub>n</sub>  
sym CON item<sub>1</sub>/exp<sub>1</sub>, ..., exp<sub>n</sub>  
sym R= reg, exp  
REP S/addr, D/addr, C/rep, B/bsz, V/inc  
REPC S/addr, D/addr, C/rep, B/bsz, V/inc  
REPI S/addr, D/addr, C/rep, B/bsz, V/inc

### ERROR FLAG

t ERR  
t ERRop aexp  
op = ZR, NZ, PL, NG, MI

### CONDITIONAL ASSEMBLY

name IFop	exp <sub>1</sub> , exp <sub>2</sub> , &nct	op	alt
name IFM1	exp, &nct	EQ	ABS
name IFPL	exp, &nct	GE	COM
name IFCP	&nct	GT	DEF
name IFCP6	&nct	LE	EXT
name IFCP7	&nct	LT	LCM
name IFPP	&nct	NE	LOC
name IFPP6	&nct		MIC
name IFPP7	&nct		REG
name IF	att, exp, &nct		REL
name IF	-att, exp, &nct		SET
name IFC	op, dstring <sub>1</sub> , dstring <sub>2</sub> , d, &nct		SST
name IFC	-op, dstring <sub>1</sub> , dstring <sub>2</sub> , d, &nct		
name ENDIF	&nct		
name ELSE	&nct		
name SKIP	&nct		

### LIST CONTROL

LIST p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub> or \*  
p = A Assembly  
B Binary control  
C Control of statements  
D Detail  
E Echoed lines  
F IF - Skipped lines  
G Code generation  
L Reference table only  
M User macros  
N Referenced symbols only  
R No references  
S System macros  
T SST symbols  
X XTEXT lines

name EJECT  
name SPACE  
name TITLE  
name NOREF  
name TTL  
name CTEXT  
name ENDX  
name XREF

aexp<sub>1</sub>, aexp<sub>2</sub>  
string  
\* or name or // or /name/  
sym<sub>1</sub>, ..., sym<sub>n</sub>  
string  
string  
A or B

### DEFINITION OPERATIONS

file XTEXT record  
name DUP rep, &nct  
name ECHO &nct, p<sub>1</sub>=f(list<sub>1</sub>), p<sub>2</sub>=f(list<sub>2</sub>), ...  
name ENDD  
name STOPDUP  
name RMT  
name HERE  
name MACRO p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>  
name MACRO name, p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>  
name MACROE p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>  
name MACROE name, p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>  
sytx OPDEF p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>  
name ENDM  
name LOCAL sym<sub>1</sub>, ..., sym<sub>n</sub>  
name IRP p

### OP CODE MANAGEMENT

name PPOP ctl, val, type  
name OPSYN name<sub>2</sub>  
name NIL  
name PURGMAC name<sub>1</sub>, name<sub>2</sub>, ..., name<sub>n</sub>  
name PURGDEF sytx  
sytx CPPOP ctl, val, reg, type  
sytx CPSYN sytx<sub>2</sub>

### MICRO

mname MICRO n<sub>1</sub>, n<sub>2</sub>, dstringd  
mname MICRO n<sub>1</sub>, dstringd  
mname DECMIC aexp, n  
mname OCTMIC aexp, n



CPU INSTRUCTIONS		① CYBER 70 Model 76, and 7600	② CYBER 170 Models 171, 172, 173, 174, 175, 720, 730, 740, 750, and 760; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series	③ CYBER 170 Model 176	④ CYBER 170 Models 815, 825, 835, 845, and 855; and CYBER 180 Series	⑤ Privileged to monitor
Instruction	Operation	Variable	Description			
00000	ES ①⑤	K	Error exit to EEA			
0000K	PS ②④		Program stop			
0100K	RJ	K	Return jump to K			
011jK	RL ①③	Bj±K	Block-copy K plus (Bj) words from LCM to SCM			
011jK	RE ②④	Bj±K	Read extended core storage			
012jK	WL ①③	Bj±K	Block-copy K plus (Bj) words from SCM to LCM			
012jK	WE ②④	Bj±K	Write extended core storage			
01300	MJ ①③		Exchange-exit to NEA if exit flag clear			
013jK	MJ ①③⑤	Bj±K	Exchange-exit to K + (Bj) if exit flag set			
013jK	XJ ②③④⑤	Bj±K	Central exchange jump to Bj+K			
014jK	RXj ①③④	Xk	Read LCM at (Xk) to Xj			
015jK	WXj ①③④	Xk	Write (Xj) into LCM at (Xk)			
0160k	RI ①③⑤	Bk	Reset channel (Bk) input buffer			
016jK	IBj ①③	Bk	Read channel (Bk) input status to Bj if j ≠ 0; otherwise, same as RI			
016j0	TBj ①③		Set Bj to current clock time			
0170k	RO ①③⑤	Bk	Reset channel (Bk) output buffer			
017jK	OBj ①③⑤	Bk	Read channel (Bk) output status to Bj if j ≠ 0; otherwise, same as RO			
02i0K	JP	Bi+K	Jump to K plus (Bi)			
030jK	ZR	Xj, K	Branch to K if (Xj) = 0			
031jK	NZ	Xj, K	Branch to K if (Xj) ≠ 0			
032jK	PL	Xj, K	Branch to K if (Xj) sign is plus			
033jK	MI	Xj, K	Branch to K if (Xj) sign is minus			
033jK	NG	Xj, K	Branch to K if (Xj) sign is minus			
034jK	IR	Xj, K	Branch to K if (Xj) in range			
035jK	OR	Xj, K	Branch to K if (Xj) not in range			
036jK	DF	Xj, K	Branch to K if (Xj) definite			
037jK	ID	Xj, K	Branch to K if (Xj) indefinite			
0400K	EQ	K	Branch to K			
0400K	ZR	K	Branch to K			
04jK	EO	Bi, Bj, K	Branch to K if (Bi) = (Bj)			
04i0K	ZR	Bi, K	Branch to K if (Bi) = 0			
05jK	NE	Bi, Bj, K	Branch to K if (Bi) ≠ (Bj)			
05i0K	NZ	Bi, K	Branch to K if (Bi) ≠ 0			
06jK	GE	Bi, Bj, K	Branch to K if (Bi) ≥ (Bj)			
06i0K	GE	Bi, K	Branch to K if (Bi) ≥ 0			
06jK	LE	Bi, Bj, K	Branch to K if (Bi) ≤ (Bj)			
06i0K	LE	Bi, K	Branch to K if (Bi) ≤ 0			
06i0K	PL	Bi, K	Branch to K if (Bi) > 0			
07i0K	LT	Bi, K	Branch to K if (Bi) < 0			
07jK	LT	Bi, Bj, K	Branch to K if (Bi) < (Bj)			
07i0K	NG	Bi, K	Branch to K if (Bi) < 0			
07jK	GT	Bi, Bj, K	Branch to K if (Bi) > (Bj)			
07i0K	GT	Bi, K	Branch to K if (Bi) > 0			
07i0K	MI	Bi, K	Branch to K if (Bi) < 0			
10ijj	BXi	Xj	Copy (Xj) to Xi			
11jK	BXi	Xj * Xk	Logical product of (Xj) and (Xk) to Xi			
12jK	BXi	Xj + Xk	Logical sum of (Xj) plus (Xk) to Xi			
13jK	BXi	Xj - Xk	Logical difference of (Xj) minus (Xk) to Xi			
14kk	BXi	-Xk	Copy complement of (Xk) to Xi			
15jK	BXi	-Xk * Xj	Logical product of (Xj) and complement of (Xk) to Xi			
16jK	BXi	-Xk + Xj	Logical sum of (Xj) plus complement of (Xk) to Xi			
17jK	BXi	-Xk - Xj	Logical difference of (Xj) minus complement of (Xk) to Xi			
20jK	LXi	jk	Logical-shift (Xi) by ±jk			
21jK	AXi	jk	Arithmetic-shift (Xi) by ±jk			
22jK	LXi	Bj, Xk	Logical-shift (Xk) by (Bj) to Xi			
22ij	LXi	Bj	Logical-shift (Xi) by (Bj) to Xi			
22i0K	LXi	Xk	Transmit (Xk) to Xi			
22jK	LXi	Xk, Bj	Logical-shift (Xk) by (Bj) to Xi			
23jK	AXi	Bj, Xk	Arithmetic-shift (Xk) by (Bj) to Xi			
23ij	AXi	Bj	Arithmetic-shift (Xi) by (Bj) to Xi			
23i0K	AXi	Xk	Transmit (Xk) to Xi			
23jK	AXi	Xk, Bj	Arithmetic-shift (Xk) by (Bj) to Xi			
24jK	NXi, Bj	Xk	Normalize (Xk) to Xi and Bj			
24i0i	NXi		Normalize (Xi) to Xi			
24ij	NXi, Bj		Normalize (Xi) to Xi and Bj			
24i0K	NXi	Xk	Normalize (Xk) to Xi			
24jK	NXi, Xk	Bj, Xk	Normalize (Xk) to Xi and Bj			
24ijK	NXi	Xk, Bj	Normalize (Xk) to Xi and Bj			
25jK	XXi, Bj	Xk	Round and normalize (Xk) to Xi and Bj			
25i0i	ZXi		Round and normalize (Xi) to Xi			
25ij	ZXi, Bj		Round and normalize (Xi) to Xi and Bj			
25i0K	ZXi	Xk	Round and normalize (Xk) to Xi			
25jK	ZXi, Xk	Bj, Xk	Round and normalize (Xk) to Xi and Bj			
25ijK	ZXi	Xk, Bj	Round and normalize (Xk) to Xi and Bj			
26jK	UXi, Bj	Xk	Unpack (Xk) to Xi and Bj			
26i0i	UXi		Unpack (Xi) to Xi			
26ij	UXi, Bj		Unpack (Xi) to Xi and Bj			
26i0K	UXi	Xk	Unpack (Xk) to Xi			
26jK	UXi, Xk	Bj, Xk	Unpack (Xk) to Xi and Bj			
26ijK	UXi	Xk, Bj	Unpack (Xk) to Xi and Bj			
27jK	PXi	Bj, Xk	Pack (Xk) and (Bj) to Xi			
27i0i	PXi		Pack (Xi) to Xi			
27ij	PXi	Bj	Pack (Xi) and (Bj) to Xi			
27i0K	PXi	Xk	Pack (Xk) to Xi			
27jK	PXi	Xk, Bj	Pack (Xk) and (Bj) to Xi			
30jK	FXi	Xj + Xk	Sum of (Xj) plus (Xk) to Xi			
31jK	FXi	Xj - Xk	Difference of (Xj) minus (Xk) to Xi			
32jK	DXi	Xj + Xk	Double-precision sum of (Xj) plus (Xk) to Xi			
33jK	DXi	Xj - Xk	Double-precision difference of (Xj) minus (Xk) to Xi			
34jK	RXi	Xj + Xk	Rounded sum of (Xj) plus (Xk) to Xi			
35jK	RXi	Xj - Xk	Rounded difference of (Xj) minus (Xk) to Xi			
36jK	IXi	Xj + Xk	Integer sum of (Xj) plus (Xk) to Xi			
37jK	IXi	Xj - Xk	Integer difference of (Xj) minus (Xk) to Xi			
40jK	FXi	Xj * Xk	Product of (Xj) times (Xk) to Xi			
41jK	RXi	Xj * Xk	Rounded product of (Xj) times (Xk) to Xi			
42jK	IXi	Xj * Xk	Integer product of (Xj) times (Xk) to Xi			

Logical Operators	
*	0100 1101 0100
+	0100 1101 1101
-	0100 1101 1001

Instruction	Operation	Variable	Description
42jK	DXi	Xj * Xk	Double-precision product of (Xj) times (Xk) to Xi
43jK	MXi	±jk	Form mask of ±jk bits in Xi
44jK	FXi	Xj/Xk	Divide (Xj) by (Xk) to Xi
45jK	RXi	Xj/Xk	Rounded divide (Xj) by (Xk) to Xi
46000	NO		Pass (do-nothing)
47kk	CXi	Xk	Population count of (Xk) to Xi
50jK	SAi	Aj±K	(Aj) plus K to Ai
51jK	SAi	Bj±K	(Bj) plus K to Ai
51i0K	SAi	K	K plus 0 to Ai
52jK	SAi	Xj±K	(Xj) plus K to Ai
53jK	SAi	Xj+Bk	(Xj) plus (Bk) to Ai
53jK	SAi	Bk+Xj	(Bk) plus (Xj) to Ai
53i0	SAi	Xj	(Xj) plus 0 to Ai
54jK	SAi	Aj+Bk	(Aj) plus (Bk) to Ai
54jK	SAi	Bk+Aj	(Bk) plus (Aj) to Ai
54i0	SAi	Aj	(Aj) plus 0 to Ai
55jK	SAi	Aj-Bk	(Aj) minus (Bk) to Ai
55jK	SAi	-Bk+Aj	(Aj) minus (Bk) to Ai
56jK	SAi	Bj+Bk	(Bj) plus (Bk) to Ai
56i0	SAi	Bj	(Bj) plus 0 to Ai
57jK	SAi	Bj-Bk	(Bj) minus (Bk) to Ai
57i0K	SAi	-Bk	0 minus (Bk) to Ai
57jK	SAi	-Bk+Bj	(Bj) minus (Bk) to Ai
60jK	SBi	Aj±K	(Aj) plus K to Bi
61jK	SBi	Bj±K	(Bj) plus K to Bi
61i0K	SBi	K	K plus 0 to Bi
62jK	SBi	Xj±K	(Xj) plus K to Bi
63jK	SBi	Bk+Xj	(Bk) plus (Xj) to Bi
63i0K	SBi	Xj	(Xj) plus 0 to Bi
64jK	SBi	Aj+Bk	(Aj) plus (Bk) to Bi
64jK	SBi	Bk+Aj	(Bk) plus (Aj) to Bi
64i0	SBi	Aj	(Aj) plus 0 to Bi
65jK	SBi	Aj-Bk	(Aj) minus (Bk) to Bi
65jK	SBi	-Bk+Aj	(Aj) minus (Bk) to Bi
660jK	CR ④	Xj, Xk	Read central memory
66jK	SBi	Bj+Bk	(Bj) plus (Bk) to Bi
66i0	SBi	Bj	(Bj) plus 0 to Bi
670jK	CW ④	Xj, Xk	Write central memory
67jK	SBi	Bj-Bk	(Bj) minus (Bk) to Bi
67i0K	SBi	-Bk	0 minus (Bk) to Bi
67jK	SBi	-Bk+Bj	(Bj) minus (Bk) to Bi
70jK	SXi	Aj±K	(Aj) plus K to Xi
71jK	SXi	Bj±K	(Bj) plus K to Xi
71i0K	SXi	K	K plus 0 to Xi
72jK	SXi	Xj±K	(Xj) plus K to Xi
73jK	SXi	Xj+Bk	(Xj) plus (Bk) to Xi
73jK	SXi	Bk+Xj	(Bk) plus (Xj) to Xi
73i0K	SXi	Xj	(Xj) plus 0 to Xi
74jK	SXi	Aj+Bk	(Aj) plus (Bk) to Xi
74jK	SXi	Bk+Aj	(Bk) plus (Aj) to Xi
74i0	SXi	Aj	(Aj) plus 0 to Xi
75jK	SXi	Aj-Bk	(Aj) minus (Bk) to Xi
75jK	SXi	-Bk+Aj	(Aj) minus (Bk) to Xi
76jK	SXi	Bj+Bk	(Bj) plus (Bk) to Xi
76i0	SXi	Bj	(Bj) plus 0 to Xi
77jK	SXi	Bj-Bk	(Bj) minus (Bk) to Xi
77i0K	SXi	-Bk	0 minus (Bk) to Xi
77jK	SXi	-Bk+Bj	(Bj) minus (Bk) to Xi

FUNCTIONAL UNITS	
Model 74 Octal Codes	Model 76 Octal Codes
Branch 00-07	Boolean 10-17
Boolean 00-07	26-27
Shift 10-17	Shift 20-23
20-27	43
43	Normalize
FP Add 30-35	24, 25
30-35	FP Add
Long Add 36, 37	30-35
36, 37	Long Add
FP Multiply 40-42	36, 37
40-42	FP Multiply
44, 45, 47	FP Divide
Increment	44, 45
50-77	Population
	47
	Increment
	50-77

CMU INSTRUCTIONS					
Instruction	Operation	Variable	Description		
464 0 K	IM	K	Move data according to word at K		
464 j K	IM	Bj+K	Move data according to word at Bj+K		
464 j 000000	IM	Bj	Move data according to word at Bj		
0 0 0 0 0 0 0 0	MD	0, k <sub>s</sub> , c <sub>s</sub> , k <sub>d</sub> , c <sub>d</sub>	Indirect move descriptor word		
465 0 0 0 0 0 0 0	DM	0, k <sub>s</sub> , d <sub>s</sub> , k <sub>d</sub> , c <sub>d</sub>	Direct move		
466 0 0 0 0 0 0 0	CC	0, k <sub>a</sub> , c <sub>a</sub> , k <sub>b</sub> , c <sub>b</sub>	Compare collated		
467 0 0 0 0 0 0 0	CU	0, k <sub>a</sub> , c <sub>a</sub> , k <sub>b</sub> , c <sub>b</sub>	Compare uncollated		

① CYBER 70 Model 76, and 7600  
 ② CYBER 170 Models 171, 172, 173, 174, 175, 720, 730, 740, 750, and 760;  
 CYBER 70 Models 71, 72, 73, and 74; and 6000 Series  
 ③ CYBER 170 Model 176  
 ④ CYBER 170 Models 825, 835, and 855  
 ⑤ d is required

Instruction	Operation	Variable	Description
01d m	LJM	m, d	Long jump to m + (d)
02d m	RJM	m, d	Return jump to m + (d)
03d	UJN	r	Unconditional jump to p + r
04d	ZJN	r	Zero jump to p + r
05d	NJN	r	Nonzero jump to p + r
06d	PJN	r	Positive jump to p + r
07d	MJN	r	Negative jump to p + r
10r	SHN	r	Shift (A) left-circular (+r) or right-end off (-r)
11d	LMN	d	Logical difference, (A) - d → A
12d	LPN	d	Logical product, (A) * d → A
13d	SCN	d	Selective clear, (A) at each d bit set
14d	LDN	d	Load d → A
15d	LCN	d	Load complement d → A
16d	ADN	d	Add d + (A) → A

Instruction	Operation	Variable	Description
17d	SBN	d	Subtract (A) - d → A
20d m	LDC	c	Load c → A
21d m	ADC	c	Add (A) + c → A
22d m	LPC	c	Logical product; (A) * c → A
23d m	LMC	c	Logical difference; (A) - c → A
2400	PSN		Pass
24d	LRD	d ④	Load (R) from d and d+1
25d	SRD	d ④	Store (R) into d and d+1
260d	EXN	d ②③④	Exchange jump CPU d unconditionally to (A)
260d	ETN	d ②	6416 Extended transfer
261d	MXN	d ①	Monitor exchange jump CPU d to (A)
262d	MAN	d ②③④	Monitor exchange jump CPU d to (MA)
270d	RPN	d ②③	Read program address of CPU d to A
270d	ERN	d ②	6416 Extended read status
30d	LDD	d	Load (d) → A
31d	ADD	d	Add (A) + (d) → A
32d	SBD	d	Subtract (A) - (d) → A
33d	LMC	d	Logical difference; (A) and (d) → A
34d	STD	d	Store (A) → d
35d	RAD	d	Replace add; (d) + (A) → d and A
36d	AOD	d	Replace add one; (d) + 1 → d and A
37d	SOD	d	Replace subtract one; (d) - 1 → d and A
40d	LDI	d	Load ((d)) → A
41d	ADI	d	Add (A) + ((d)) → A
42d	SBI	d	Subtract (A) - ((d)) → A
43d	LMI	d	Logical difference; (A) - ((d)) → A
44d	STI	d	Store (A) → (d)
45d	RAI	d	Replace add; (A) + ((d)) → (d) and A
46d	AOI	d	Replace add one; ((d)) + 1 → (d) and A
47d	SOI	d	Replace subtract one; ((d)) - 1 → (d)
50d m	LDM	m, d	Load (m + (d)) → A
51d m	ADM	m, d	Add (m + (d)) + (A) → A
52d m	SBM	m, d	Subtract (A) - (m + (d)) → A
53d m	LMM	m, d	Logical difference; (A) - (m + (d)) → A
54d m	STM	m, d	Store (A) → m + (d)
55d m	RAM	m, d	Replace add; (A) + (m + (d)) → m + (d) and A

# PSEUDO INSTRUCTION INDEX

Name	Placement	Usage	Page Number	Name	Placement	Usage	Page Number
ABS	first group	CPA	4-6	LOC	normal	CP, PP	4-38
BASE	anywhere	CP, PP	4-24	LOCAL	macro or opdef	CP, PP	5-33
BSS	normal	CP, PP	4-37	MACHINE	first group	CP, PP	4-7
BSSZ	normal	CP, PP	4-48	MACRO	anywhere	CP, PP	5-15
B1=1	anywhere	CP	4-30	MACROE	anywhere	CP, PP	5-24
B7=1	anywhere	CP	4-30	MAX	normal	CP, PP	4-42
CHAR	anywhere	CP, PP	4-26	MICCNT	anywhere	CP, PP	4-44
CODE	anywhere	CP, PP	4-26	MICRO	anywhere	CP, PP	7-2
COL	normal	CP, PP	4-31	MIN	normal	CP, PP	4-43
COMMENT	anywhere	CP, PP	4-20	NIL	anywhere	CP, PP	6-6
CON	normal	CP, PP	4-54	NOLABEL	anywhere	CPA, PP	4-20
CPOP	anywhere	CP	6-7	NOREF	anywhere	CP, PP	4-78
CPSYN	anywhere	CP	6-10	OCTMIC	anywhere	CP, PP	7-4
CTEXT	normal	CP	4-79	OPDEF	anywhere	CP	5-27
DATA	normal	CP, PP	4-48	OPSYN	anywhere	CP, PP	6-5
DECMIC	anywhere	CP, PP	7-4	ORG	normal	CP, PP	4-35
DIS	normal	CP, PP	4-49	ORGC	normal	CP, PP	4-35
DUP	normal	CP, PP	5-6	PERIPH	first group	PP	4-10
ECHO	normal	CP, PP	5-7	POS	normal	CP, PP	4-40
EJECT	anywhere	CP, PP	4-76	PPOP	anywhere	PP	6-3
ELSE†	anywhere	CP, PP	4-60	PPU	first group	PP	4-8
END†	required last	CP, PP	4-4	PURGDEF	anywhere	CP	6-10
ENDD	anywhere	CP, PP	5-10	PURGMAC	anywhere	PP, CP	6-7
ENDIF†	anywhere	CP, PP	4-59	QUAL	anywhere	CP, PP	4-28
ENDM	anywhere	CP, PP	5-14	REP	normal	CPR	4-57
ENDX	normal	CP, PP	4-79	REPC	normal	CPR	4-57
ENTRY	normal	CP, PP	4-45	REPI	normal	CPR	4-57
ENTRYC	normal	CP, PP	4-45	RMT	anywhere	CP, PP	5-3
EQU	normal	CP, PP	4-41	R=	normal	CP	4-55
ERR	normal	CP, PP	4-71	SEG	normal	CPA, PP	4-16
ERRMI	normal	CP, PP	4-72	SEGMENT	normal	CPA, PP	4-15
ERRNG	normal	CP, PP	4-72	SET	normal	CP, PP	4-41
ERRNZ	normal	CP, PP	4-72	SKIP	anywhere	CP, PP	4-70
ERRPL	normal	CP, PP	4-72	SPACE	anywhere	CP, PP	4-76
ERRZR	normal	CP, PP	4-72	SST	anywhere	CP, PP	4-45
EXT	normal	CP, PP	4-47	STEXT	first group	CP, PP	4-17
HERE	anywhere	CP, PP	5-4	STOPDUP	normal	CP, PP	5-9
IDENT	required first	CP, PP	4-2	TITLE	anywhere	CP, PP	4-77
			and 4-11	TTL	anywhere	CP, PP	4-78
IF	normal	CP, PP	4-65	USE	normal	CP, PP	4-32
IFC	anywhere	CP, PP	4-68	USELCM	normal	CP	4-34
IFCP	normal	CP, PP	4-60	VFD	normal	CP, PP	4-53
IFCP6	normal	CP, PP	4-60	XREF	anywhere	CP, PP	4-80
IFCP7	normal	CP, PP	4-60	XTEXT	normal	CP, PP	5-2
IFEQ	normal	CP, PP	4-62	(blank)	normal	CP, PP	4-48
IFGE	normal	CP, PP	4-62	=	normal	CP, PP	4-41
IFGT	normal	CP, PP	4-62				
IFLE	normal	CP, PP	4-62				
IFLT	normal	CP, PP	4-62				
IFMI	normal	CP, PP	4-64				
IFNE	normal	CP, PP	4-62				
IFPL	normal	CP, PP	4-64				
IFPP	normal	CP, PP	4-60				
IFPP6	normal	CP, PP	4-60				
IFPP7	normal	CP, PP	4-60				
IRP	anywhere	CP, PP	5-35				
LCC	normal	CPR	4-21				
LDSET	anywhere	CPR	4-21				
LIST	anywhere	CP, PP	4-73				
LIT	normal	CP, PP	4-51				

†Looked for during IF skipping.

### Legend

CP Absolute or relocatable CPU program  
 CPA Absolute CPU program  
 CPR Relocatable CPU program  
 PP Absolute PPU program

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN 55440  
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

 CONTROL DATA