CONTROL DATA
CORPORATION

CONTROL DATA®
6000 SERIES COMPUTER SYSTEMS
7600 COMPUTER SYSTEM

6000 COMPASS VERSION 2
7600 COMPASS VERSIONS 1 AND 2
REFERENCE MANUAL

CPU AND PPU INSTRUCTION INDEX

CPU INSTRUCTIONS

Mnemonic Code		Operation Code (octal)	Section Number
AXi	±jk	21ijk	8.4.25
AXi	Bj,Xk	23kjk	8.4.27
BXi	Xj	10ijj	8.4.16
BXi	Xj*Xk	11ijk	8.4.17
BXi	Xj+Xk	12ijk	8.4.18
BXi	Xj-Xk	13ijk	8.4.19
BXi	-Xk	14ikk	8.4.20
BXi	-Xk*Xj	15ijk	8.4.21
BXi	-Xk+Xj	16ijk	8.4.22
BXi	-Xk-Xj	17ijk	8.4.23
CXi	Xk	47ikk	8.4.44
DF	Xj,K	036jK	8.4.14
DXi	Xj+Xk	32ijk	8.4.33
DXi	Xj-Xk	33ijk	8.4.33
DXi	Xj*Xk	42ijk	8.4.38
EQ	Bi,Bj,K	04ijK	8.4.15
ES	K	00000	8.4.2
FXi	Xj+Xk	30ijk	8.4.32
FXi	Xj-Xk	31ijk	8.4.32
FXi	Xi*Xk	40ijk	8.4.36
FXi	Xj/Xk	44ijk	8.4.41
GE	Bi,Bj,K	06ijK	8.4.15
GE	Bi,K	06i0K	8.4.15
GT	Bj,Bi,K	07ijK	8.4.15
GT	Bj,K	070jK	8.4.15
IBj	Bk	016jk	8.4.12
ID	Xj,K	037jK	8.4.14
IR	Xj,K	034jk	8.4.14
IXi	Xj+Xk	36ijk	8.4.35
IXi	Xj-Xk	37ijk	8.4.35
IXi	Xj*Xk	42ijk	8.4.39
JP	Bj±K	02i0K	8.4.13
LE	Bj,Bi,K	06ijK	8.4.15
LT	Bi,Bj,K	07ijK	8.4.15
LXi	±jk	20ijk	8.4.24
LXi	Bj,Xk	22ijk	8.4.26
MI	Xj,K	033jK	8.4.14
MI	Bi,K	07i0K	8.4.15
MJ		01300	8.4.7
MJ	Bi±K	013jK	8.4.7
MXi	±jk	43ijk	8.4.40
NE	Bi,Bj,K	05ijK	8.4.15
NG	Bi,K	07i0K	8.4.15
NG	Xj,K	033jK	8.4.14
NO	n	46n	8.4.43
NXi	Bj,Xk	24ijk	8.4.28
NZ	Bi,K	05i0K	8.4.15
NZ	Xj,K	031jK	8.4.14
OBj	Bk	017jk	8.4.12
OR	Xj,K	035jK	8.4.14
PL	Xj,K	032jK	8.4.12
PL	Bi,K	06i0K	8.4.15
PS	K	0000K	8.4.1
PXi	Bj,Xk	27ijk	8.4.31
RE	Bj+K	011jK	8.4.4
RI	Bk	0160k	8.4.9
RJ	K	0100K	8.4.3
RL	Bj±K	011jK	8.4.5
RO	Bk	0170k	8.4.11
RXi	Xj+Xk	34ijk	8.4.34
RXi	Xj-Xk	35ijk	8.4.34
RXi	Xj*Xk	41ijk	8.4.37

CPU INSTRUCTIONS (cont'd)

Mnemonic Code		Operation Code (octal)	Section Number
RXi	Xj/Xk	45ijk	8.4.42
RXj	Xk	014jk	8.4.8
SAi	Aj±K	50ijK	8.4.45
SAi	Bj±K	51ijK	8.4.45
SAi	Xj±K	52ijK	8.4.45
SAi	Xj+Bk	53ijk	8.4.45
SAi	Aj+Bk	54ijk	8.4.45
SAi	Aj-Bk	55ijk	8.4.45
SAi	Bj+Bk	56ijk	8.4.45
SAi	Bj-Bk	57ijk	8.4.45
SBi	Aj±K	60ijK	8.4.46
SBi	Bj±K	61ijK	8.4.46
SBi	Xj±K	62ijK	8.4.46
SBi	Xj+Bk	63ijk	8.4.46
SBi	Aj+Bk	64ijk	8.4.46
SBi	Aj-Bk	65ijk	8.4.46
SBi	Bj+Bk	66ijk	8.4.46
SBi	Bj-Bk	67ijk	8.4.46
SXi	Aj±K	70ijK	8.4.47
SXi	Bj±K	71ijK	8.4.47
SXi	Xj±K	72ijK	8.4.47
SXi	Xj+Bk	73ijk	8.4.47
SXi	Aj+Bk	74ijk	8.4.47
SXi	Aj-Bk	75ijk	8.4.47
SXi	Bj+Bk	76ijk	8.4.47
SXi	Bj-Bk	77ijk	8.4.47
TBj		016j0	8.4.10
UXI	Bj,Xk	26ijk	8.4.30
WE	Bj+K	012jK	8.4.4
WL	Bj+K	012jK	8.4.5
WXj	Xk	015jK	8.4.8
XJ	Bj±K	013jK	8.4.6
ZR	Xj,K	030jK	8.4.14
ZR	Bi,K	04i0K	8.4.15
ZXi	Bj,Xk	25ijk	8.4.29

PPU INSTRUCTIONS

Name	Operation Code (octal)	Section Number
ACN d	74d	9.2.18
ADC c	21dm	9.2.4
ADD d	31d	9.2.9
ADI d	41d	9.2.10
ADM m,d	51dm	9.2.11
ADN d	16d	9.2.3
AJM m,d	64dm	9.2.13
AOD d	36d	9.2.9
AOI d	46d	9.2.10
AOM m,d	56dm	9.2.11
CRD d	60d	9.2.12
CRM m,d	61d	9.2.12
CWD d	62d	9.2.12
CWM m,d	63dm	9.2.12
DCN d	75d	9.2.18

PPU INSTRUCTIONS (cont'd)

Name	Operation Code (octal)	Section Number
EIM m,d	61dm	9.2.14
EJM m,d	67dm	9.2.13
EOM m,d	65dm	9.2.14
ERN d	270d	9.2.8
ESN d	7700	9.2.19
ETN d	260d	9.2.8
EXN d	260d	9.2.6
FAN d	76d	9.2.18
FIM m,d	60dm	9.2.14
FJM m,d	66dm	9.2.13
FNC m,d	77dm	9.2.18
FOM m,d	64dm	9.2.14
IAM m,d	71dm	9.2.16
IAN d	70d	9.2.15
IJM m,d	65dm	9.2.13
IRM m,d	62dm	9.2.14
LCN d	15d	9.2.3
LDC c	20dm	9.2.4
LDD d	30d	9.2.9
LDI d	40d	9.2.10
LDM m,d	50dm	9.2.11
LDN d	14d	9.2.3
LJM m,d	01dm	9.2.1
LMC c	23dm	9.2.4
LMD d	33d	9.2.9
LMI d	43d	9.2.10
LMM m,d	53dm	9.2.11
LMN d	11d	9.2.3
LPC c	22dm	9.2.4
LPN d	12d	9.2.3
MJN r	07d	9.2.1
MXN d	261d	9.2.6
NIM m,d	63dm	9.2.14
NJN r	05d	9.2.1
NOM m,d	67dm	9.2.14
OAM m,d	73dm	9.2.16
OAN d	72d	9.2.15
ORM m,d	66dm	9.2.14
PJN r	06d	9.2.1
PSN	2400	9.2.5
RAD d	35d	9.2.9
RAI d	45d	9.2.10
RAM m,d	55dm	9.2.11
RFN d	74d	9.2.17
RJM m,d	02dm	9.2.1
RPN d	270d	9.2.7
SBD d	32d	9.2.9
SBI d	42d	9.2.10
SBM m,d	52dm	9.2.11
SBN d	17d	9.2.3
SCN d	13d	9.2.3
SHN r	10d	9.2.2
SOD d	37d	9.2.9
SOI d	47d	9.2.10
SOM m,d	57dm	9.2.11
STD d	34d	9.2.9
STI d	44d	9.2.10
STM m,d	54dm	9.2.11
UJN r	03d	9.2.1
ZJN r	04d	9.2.1

REVISION RECORD

REVISION	DESCRIPTION
5/70	Original Printing
A	
10/70	This printing obsoletes the Preliminary version of this Reference Manual.
B	This revision corrects various typographical and technical errors and reflects changes resulting from
4/71	adding the colon to the available character set.
C	This revision notes differences for 7600 COMPASS Version 2 on pages iii, 1-1, 1-3, 3-1, 3-2, 4-28, 5-2,
2/72	5-3, 10-1, 10-2, 10-3, 10-4, 10-5, 10-6, B-1, B-13, and B-15. Other changes are corrections to technical
	content.
D	
8/72	This revision corrects various technical and typographical errors and reflects changes resulting from
	PSRs. Changes are on front cover and on pages vii, ix, 2-1, 2-6, 2-9, 2-17, 2-18, 2-26, 3-5, 3-10,
	4-2, 4-3, 4-13, 4-16, 4-25, 4-28, 4-29, 4-34, 4-43, 4-49, 4-54, 4-57, 4-58, 4-59, 4-60, 4-61, 4-63,
	4-66, 4-70, 5-7, 5-8, 5-9, 5-10, 5-16, 5-19, 6-4, 6-8, 7-1, 7-2, 7-3, 8-9, 8-10, 8-14, 8-20, 8-41, 8-42,
	8-43, 8-44, 8-45, 8-46, 8-47, 9-17, 9-20, 9-21, 9-22, 11-9, 11-10, 11-11, 11-12, 11-13, A-7, 8-8,
	A-9, A-10, B-3, B-14, D-1, D-2, D-3, Index-1, Index-2, Index-3, Index-4, Index-5, Index-7, Index-8,
	Index-9, Index-11, Index-13, Index-15, Index-16, Index-17, Index-18, Comment sheet and back cover.
	Pages 4-16.1, 4-62.1, 4-62.2, 8-14.1, and 10-10 are added.
Publication No. 60279900	

Printed in the United States of America

Address comments concerning this manual to:
Control Data Corporation
Documentation Department
215 Moffett Park Drive
Sunnyvale, California 94086

or use Comment Sheet in the back of this manual.

PREFACE

This manual is directed at programmers using COMPASS Version 1 for the CONTROL DATA® 7600 Computer System under control of the SCOPE 1, COMPASS Version 2 for the CONTROL DATA 7600 Computer System under control of SCOPE 2, and COMPASS Version 2 for the CONTROL DATA® 6000 Series Computer Systems under control of the SCOPE 3.3 Operating System.

This manual describes the principles, features, methods, rules, and techniques of producing a COMPASS language program. It does not describe instructions unique to CONTROL DATA® CYBER 70 Series Computer Systems.

The user is assumed to be familiar with either the CONTROL DATA 6000-Series Computer Systems or with the CONTROL DATA 7600 Computer System and is assumed to be familiar with assemblers in general. Familiarity with the related 6000 or 7600 Operating System is helpful.

Readers with no previous experience with 6000 COMPASS or 7600 COMPASS assemblers are encouraged to direct their initial attentions to the following sections of this 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 PPU Symbolic Machine Instructions, the chapter depending upon the machine language the user requires.
Chapter 10	Program Execution

This manual is not intended to replace the 6400/6500/6600/6700 Computer Systems Reference Manual, Pub. No. 60100000 or the 7600 Computer System Reference Manual, Pub. No. 60258200 to which the user is referred for detailed information on machine instructions. Information in Pub. No. 60100000 and Pub. No. 60258200 takes precedence over information in this manual if discrepancies should arise between these publications.

In this manual, numbers occurring in text are decimal unless otherwise noted. Lower case 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.

CONTENTS

FIGURES

TABLES

INTRODUCTION 1

The CONTROL DATA COMPASS 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 unit (PPU). The assembler executes on either a CONTROL DATA 7600 Computer System or a CONTROL DATA 6000-Series Computer System.

Subprograms to be executed on a 7600 system can be assembled on a 6000-Series system and vice versa. However, the user must use only instructions accepted by the system on which the object program is to be executed.

From CPU source language subprograms, the COMPASS assembler generates binary output acceptable for loading and execution by a 6000 or 7600 central processor unit under SCOPE control. Subprograms can be compiled independently for subsequent loading and execution as a single program.

From PPU source language programs, the COMPASS assembler generates absolute code to be loaded and executed on a peripheral processor unit of a 7600 or a 6000 Series Computer System.

Source statements consist of CPU or PPU symbolic machine instructions and pseudo instructions. The symbolic machine instructions (chapters 8 and 9) are counterparts of the binary machine instructions; they provide a means of expressing symbolically all functions of the Computer System.

The pseudo instructions are oriented towards control of the assembler itself; they control the assembler much the same as machine language instructions control the computer. The ability to control assembly places COMPASS at a level of sophistication much higher than that of the conventional assembler.

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 designate up to 255 areas to facilitate inter-program 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 small core memory (CM or SCM) with the source program. †
• Data notation	Data can be designated in integer, floating-point, and character code 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.

†COMPASS Version 2 under SCOPE 2 allows data to be loaded into LCM.

• 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. DATE and TIME are predefined by the system.
• 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 OPERATING SYSTEM INTERFACE

COMPASS Version 1 executes on a 7600 CPU under control of the SCOPE 1 Operating System; COMPASS Version 2 executes on a 6000-series computer system CPU under control of the SCOPE 3.3 Operating System or on a 7600 computer system under control of the SCOPE 2 Operating System.

1.2 CONFIGURATION

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

1.3 ASSEMBLER EXECUTION

COMPASS is called from the system library by a COMPASS control card (chapter 10) or by a compiler such as RUN or FTN upon encountering a COMPASS IDENT statement in the source input file. Parameters on the card 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.

Core requirements for tables used by the assembler are dynamically changed as requirements change during assembly. If insufficient core is available for the program, the intermediate file and cross-references are transferred to the system mass storage device and assembly continues.

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

1.4 RELOCATABLE OBJECT PROGRAM EXECUTION

When the assembler has completely processed the source deck, the programmer can use a SCOPE control card 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 may be on some programmer-specified file. Normally, this loading and execution does not take place if the COMPASS assembler detects fatal errors.

LANGUAGE STRUCTURE 2

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.

Statements are 80-to-90-column lines. 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 73-80 can be used for additional comments or sequencing. Columns 81-90 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 (-) (section 3.2.4).

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 (section 4.4.5).

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.

2.1.8 CODING CONVENTIONS

Figure 2-1 illustrates a COMPASS coding form that establishes a coding convention as follows:

Column	Contents
1	Blank, asterisk,or comma
2-9	Location field entry or plus, or minus left justified
10	Blank
11-16	Operation field entry left justified
17	Blank
18-29	Variable field entry left justified
30	Beginning of comments

All examples in this manual abide by this convention.

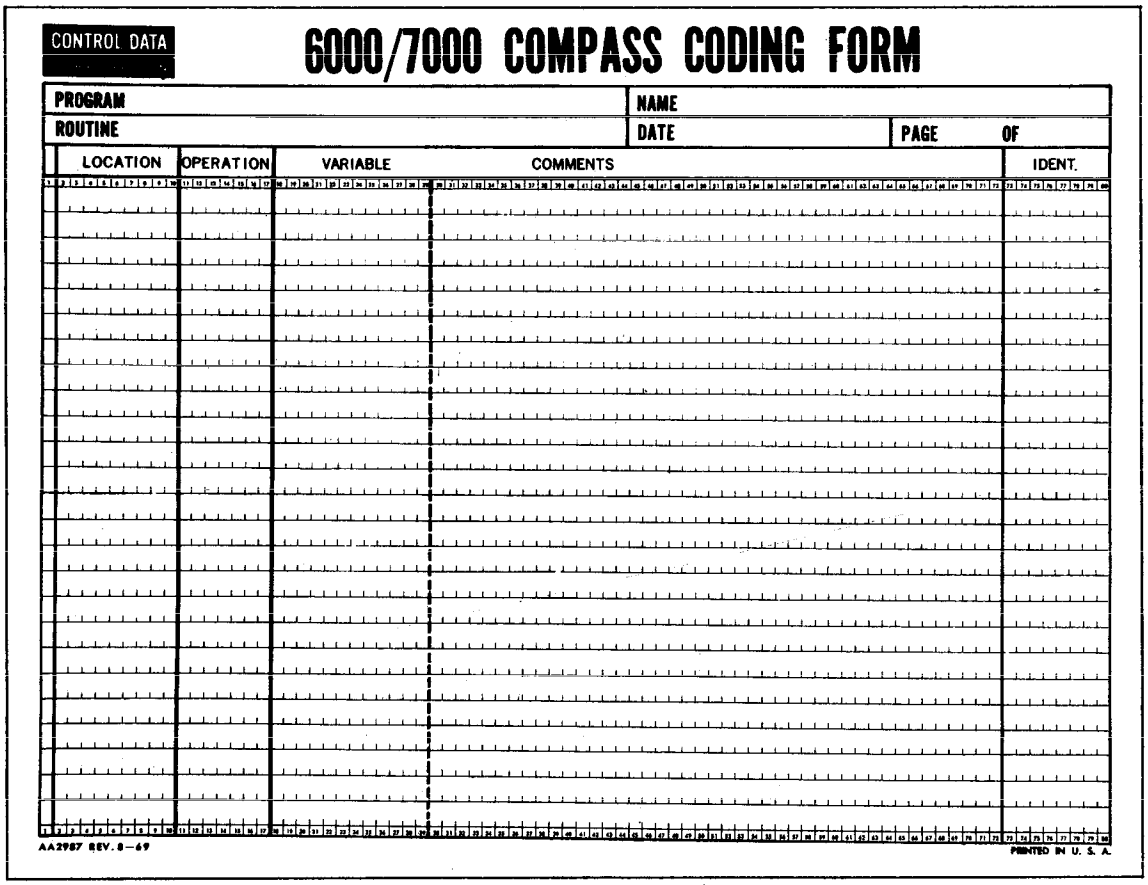


Figure 2-1. COMPASS Coding Form

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. 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 ⌐→ 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 ⌐→ is superfluous and is removed by editing before the definition is interpreted.

Each removal of ⌐→ shifts the remaining columns in the statement left one character. This could become significant when comments follow a blank variable field because the comments could 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 (≠) 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 card 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 non-fatal 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 of micro marks are removed before the name is scanned (see section 2.2 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 7600 PPU assembly, the name can be seven characters but for a 6000 Series 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 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 (section 2.4.1 Linkage Symbols).

Concatenation marks or pairs of micro marks are removed before a symbol is examined (Section 2.2 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 (Section 2.4.2). Register designators are described further in Section 2.5.

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 (section 3.2.2) and having an attribute defined as follows:

   a. Absolute for the absolute block

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

   c. Relocatable for local blocks other than absolute during pass one.

   d. Absolute for local blocks during pass two of an absolute assembly.

2. A symbol used in the location field of definition pseudo instructions (section 4.6) 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 in section 2.4.2 Default Symbols.

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

Examples:

Legal Symbols	Illegal Symbols	
P	5A	First character numeric
R3	ABCDEFGHI	Exceeds eight characters
PROGRAM	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, see section 2.4.2 Default Symbols).

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. External symbols are not qualified (section 2.4.5).

2.4.2 DEFAULT SYMBOLS

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

=Ssymbol — 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.

If the symbol is defined by a conventional method, COMPASS does not define it again but uses the programmer definition.

=Xsymbol — 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 external as though declared in an EXT pseudo instruction. A symbol prefixed by =X must conform to the requirements for external symbols.

The system does not define a default symbol and issues an error flag if a symbol is prefixed by both =S and =X, or is prefixed by =X 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 (section 4.4.3) 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. These 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 or =X (see section 2.4.2). 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.

Register Type	Designator
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.

COMPASS does not recognize registers in PPU 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 (section 4.6.2) 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
	1	11	18	30
6032001000		SB3	A2+ALPHA	

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
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 references 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 section 3.2.

Designator	Significance
* 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.

Designator	Significance
*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: 0 COMPASS was called by a COMPASS control card 1 COMPASS was called by the FORTRAN RUN compiler 2 COMPASS was called by the FORTRAN FTN compiler

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

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 (section 4.8.2) and LIT (section 4.8.4) pseudo instructions or as specifications of field lengths on VFD pseudo instructions.

2.7.2 CONSTANTS

A data constant is an expression element consisting of a value represented in octal, decimal, 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 (section 2.7.4) 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 (section 4.8.2) or a LIT (section 4.8.4) 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 section 3.1.3).

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 (section 4.11.1).

Example:

In the following example, using CPU instructions, the first statement creates a word in the literals block having the value 0000000000000000001. 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 0000000000000000002. 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

100   6120005555 +
                  6130005555 +
101   6140005556 +
                        5555
                  6120005555 +
102   6130005556 +
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		SB2	=1	
		SB3	=1RA	
		SB4	=1RB	
	L	LIT	1,2	
		SB2	L	
		SB3	L+1	

```
                CONTENT OF LITERALS BLOCK.

      005555  0000000000000000001              A
      005556  0000000000000000002              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
		1		11	18	30
		5557		LIT	1,3,1RD,2	

CONTENT OF LITERALS BLOCK.

```
005555  000000000000000000001          A
005556  000000000000000000002          B
005557  000000000000000000001          A
005560  000000000000000000003          C
005561  000000000000000000004          D
005562  000000000000000000002          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
		1		11	18	30
		5555		LIT	1,2,3,4	

CONTENT OF LITERALS BLOCK.

```
005555  000000000000000000001          A
005556  000000000000000000002          B
005557  000000000000000000003          C
005560  000000000000000000004          D
```

2.7.4 CHARACTER DATA NOTATION

Character data strings are converted to the code in use at the time the string is evaluated (section 4.4.2, CODE pseudo instruction), 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:

		Example
Data Item	sign | n | type | string	-3RABC
	or	
	sign | type | d | string | d	-R*ABC*
Constant †	n | type | string	3RABC
Literal †	= | sign | n | type | string	=-3RABC
	or	
	= | sign | type | d | string | d	=-R*ABC*

= 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 ∧
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:

Type	Significance
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, the zero bits are not guaranteed; C is the same as L.
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, the bits are not guaranteed; Z is the same as L.

d

A delimiting character used only when n is omitted. The characters between the first occurrence of d and the second occurrence of d comprise the string. d can be any character other than ⌐→ or ≠.

string

Characters from one of the COMPASS character sets (appendix D), except for those characters that act as delimiters (see n and d), the concatenation character (⌐→), and pairs of micro marks (≠).

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.

1. n is 0 and type is immediately followed by a delimiter (for example, 0L.)
2. n is omitted and the two delimiting characters are concurrent (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

```
Location        Code Generated

144   0522221722551116552O
145   042155000000000000
146   5555555555555555555
```

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

```
Location        Code Generated

1100            1725
1101            2420
1102            2524
```

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

Constants

```
Location        Code Generated

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$
      |           VFD         30/4HIOTA,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

                      100003765
                      100003770
2652    5110003772 +
                    5120003774 +
2653    5130003767 +
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	TAG1	LIT	RA+-*/(A,6L)$= ,.,0C0,0L	
		LIT	20HLITERALS	
		SA1	=0CTENCHARCTS	
		SA2	=H+LEFT JUSTIFY WITH BLANKS+	
		SA3	=0L0	

```
                CONTENT OF LITERALS BLOCK.

003765  0000000000454647505l          +-*/(
003766  5253545556570000000     )$= ,.
003767  3300000000000000000     0
003770  1411240522011423555     LITERALS
003771  5555555555555555555
003772  2405160310012203242     TENCHARCTS
003773  0000000000000000000
003774  1405062455122523241     LEFT JUSTI
003775  0631552711241055021     FY WITH BL
003776  0116132355555555555     ANKS
```

The first LIT pseudo instruction generates three words in the literals block; the 0L 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
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.

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 1.15×10^{18} (fixed point) or 7.9×10^{28} (floating point, ignoring the decimal point). Extra significant digits cause erroneous results.

If value is omitted, it is assumed to be zero.

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 — Defines a power of 10 scale factor

E±n or En or E Single precision

EE±n or EEn or EE Double precision

When the sign is plus or is omitted, the exponent (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.

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).

If EE is used with a fixed point value, the assembler produces a fixed point number in single precision.

The effect of the exponent is to multiply the value by 10 decimal raised to the n power.

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

S±n or Sn 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 Pn 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 bit.

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

For example, a value with P-6 will have a biased exponent of 2006_8; a value with P10 will have an exponent of 1765_8.

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 1.0E4000S-1200 yields a number that is approximately 5.8×10^{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   777777777777777742
5001   172350000000000000
5002   164300000000000000
5003   200000000000000012
5004   176000000000000002
5005   171546517676355442 64
5006   172003146314631463 14
5007   777777777777777777
5010   000000000000000000
```

```
|  | LOCATION | OPERATION | VARIABLE | COMMENTS
|1 |          | 11        | 18       | 30
|  | POOL     | DATA      | -29      |
|  | NUM      | DATA      | 1.0EE1   |
|  |          | DATA      | 1.0E+1P0 |
|  |          | DATA      | 3.2P1S-5E1 |
|  |          | DATA      | 0.0151E+01 |
|  |          | DATA      | 0.1P47,-E,DEES |
```

CPU Numeric Constants

```
Location        Code Generated

                          5001 +
                           555
5012
5112   20360
              43760
                   7150400000
```

```
|  | LOCATION | OPERATION | VARIABLE | COMMENTS
|1 |          | 11        | 18       | 30
|  | ALPHA    | EQU       | POOL+1   |
|  | VAL      | EQU       | 555B     |
|  |          | BSSZ      | 100B     |
|  |          | LX3       | -14B     |
|  |          | MX7       | 48       |
|  |          | SX5       | 1S17     |
```

CPU Numeric Literals

```
Location        Code Generated

5113  5150005151 +
                  5130005152 +
                        5153
                        5155
                        5156
                        5157

                CONTENT OF LITERALS BLOCK.

005151   2004675500023400000 4    PD^  B1   D
005152   1720431463146314631 5    OP☒L:L:L:M
005153   1723500000000000000 0    OS/
005154   1643000000000000000 0    N8
005155   1720031463146314631 4    OPCL:L:L:L
005156   7777777777777777754      ;;;;;;;;;=
005157   1715465176763554426 4    OM-(¬¬2=7≠
005160   7777777777777777777      ;;;;;;;;;;;
005161   0000000000000000000
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		SA5	=2004675500023400000 4B	
		SA3	=1.1	
	ABLE	LIT	1.0EE1	
		LIT	0.1P47	
		LIT	-D19	
		LIT	0.0151E+01,-E,DEES	

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

PPU Data Items

```
Location        Code Generated

300             0005
301             7766
302             0013
303             0030
304             0002
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		PPU		
		•	•	
		•	•	
		•	•	
		DATA	5,-9D,+B13,14BS1,24BE-1	

PPU Constants

Location	Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
			1	11	18	30
305	0000			CON	0,+11	
306	0011					
307	4443			CON	-3334	
		31	ABC	=	25D	
		101	NUM	SET	0101	
310	7777			CON	7777	

PPU Literals

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
311	5000 1103		LDM	=10D	
313	5100 1104		ADM	=-1	
315	5000 1105		LDM	=7777	

CONTENT OF LITERALS BLOCK.

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

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

Numeric or character constant

Special element

Register designator (CPU only)

Literal

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

^ Logical minus

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.

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 these two conditions:

1. The expression contains an even number of relocatable elements
2. 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 these two conditions, a combination of relocatable and absolute terms:

1. The expression does not contain an even number of relocatable elements
2. 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.
3. The uncanceled element can have three kinds of relocation:
    a. Positive program
    b. Negative program
    c. Positive common (negative common is not permitted by the loaders)

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.

1. The expression contains an even number of relocatable terms.

2. 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 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`	
`XYZ+ABC`	Illegal; both are external
`-ABC+*-LIMA`	Illegal; ABC is negative
`XYZ+*0`	Illegal; *O 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 (section 8.4). 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`
`LIMA+X3-10B`
`-10B+LIMA+X3`
} Produce identical results

`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, the system uses the value of a relocatable symbol relative to the block in which the symbol was defined. For pass two evaluation, the system 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:

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

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

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

4. 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 PPU 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.

PROGRAM STRUCTURE 3

This chapter describes the general structure of a program. In some cases, it repeats information described elsewhere and correlates it so that the programmer will obtain a better understanding of how the program is assembled, loaded, and executed. Some mention is made of the SCOPE loader, but for a complete description of the loader, refer to the reference manual for the operating system or loader in use.

The first topic considered in this chapter is the subprogram block and how the assembler and the programmer organize the object code into blocks. Following this is a brief description of the counters that control the blocks.

Finally, there is a summary of the differences in the structure of absolute and relocatable programs and the effect of these differences on block usage.

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 user 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 and USE pseudo instructions, 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.

All symbols are assigned absolute values, the table of block names is cleared, the list of USE and ORG 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 and ORG 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 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*.

3.1.2 ZERO BLOCK

The zero block has the block name 0 and is the nominal 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.

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.

3.1.4 USER-ESTABLISHED LOCAL BLOCKS

By using USE 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.

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 7600 SCM or 6000 CM through the USE† pseudo instruction where the name of the block is the name enclosed by slant bars i.e.,/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/ instruction, COMPASS treats the block as a local block having the slant-bar enclosed name.

†7600 COMPASS Version 2 allows presetting of data in LCM through the USELCM /name/ instruction (section 4.5.2).

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 blank common block is allocated 7600 SCM space or 6000 CM space by the SCOPE relocatable loader after all subprograms are loaded, according to the largest block area declared by any of the subprograms. The blank common block is established through a USE pseudo instruction (section 4.5.1); it has no name; a USE // indicates blank common.

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

Only a CPU program can use the USELCM pseudo instruction to establish named 7600 LCM or 6000 ECS blank common blocks. These blocks provide a means of symbolically addressing the job's LCM or ECS field from a CPU program. In pass one, COMPASS assigns addresses in each block starting with zero (RAL). At the end of assembly, COMPASS assigns an origin relative to zero to each common block, in the sequence in which they are established. No code can be assigned to the blocks; they can be used for storage reservation only. It is the responsibility of the user to assure that sufficient LCM or ECS is scheduled on the job card to accommodate the blocks when the program is executed. (For use of the LCM or ECS blocks refer to the USELCM pseudo instruction, section 4.5.2).

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 or BSS pseudo instructions to advance the origin counter; ORG also permits 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 and 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, section 4.5.5.) 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 59-00, from left to right within a 60-bit CPU word and numbered 11-00 within a 12-bit PPU word. Then, the position counter is initially 60 and 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 word 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 contains a symbolic address or + in the location field and the location field is not ignored

The next symbolic instruction to be assembled is a 6000 Series RE, WE, PS, or XJ instruction. (The programmer can negate this force upper by placing a minus sign in the location field of the instruction.)

The current pseudo instruction is END, BSS, BSSZ, DATA, DIS, CON, SEGMENT, SEG or IDENT.

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

JP
RJ
Unconditional EQ
Unconditional ZR
ES (7600 only)
MJ (7600 only)
PS (6000 Series only)
XJ (6000 Series only)

This post force upper is not done immediately, but is deferred until the next machine instruction or data generating, storage allocating, or binary control pseudo instruction in the same USE block is encountered. 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 computer run or in independent runs. The subprogram 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 SCOPE loader. A COMPASS language subprogram is composed of instructions beginning with an IDENT pseudo instruction and ending with an END pseudo instruction.

The COMPASS assembler repertoire includes pseudo instructions that facilitate relocatable subprogram linkage. Through these linkages, subprograms loaded together can transfer control to each other and can access common storage locations.

Upon completion of assembly of a relocatable subprogram, COMPASS assigns each local block an origin relative to the zero block (figure 3-1). Output is in the form of tables for the SCOPE Relocatable Loader (appendix B). Each local block thus becomes an extension of the zero block. 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 RAS (or RA).

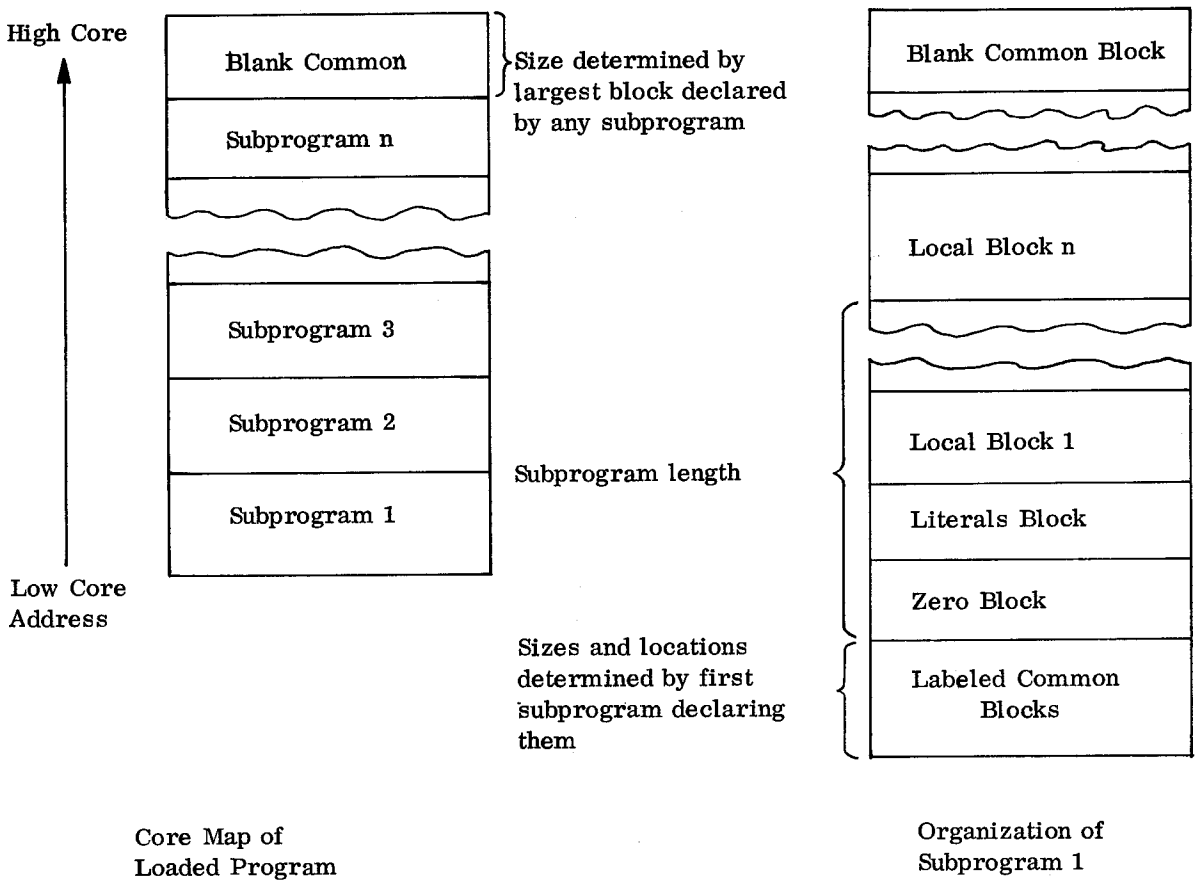


Figure 3-1. Relocatable Program Structure

3.4 ABSOLUTE PROGRAM STRUCTURE

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

The programmer has the option of constructing his 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 core storage consecutively so that total storage requirements for a program are reduced.

During assembly of an absolute program or overlay, COMPASS creates a core 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 binary output for the program consists of a record for each overlay. Note that the record 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 record 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 record has three parts:

1. 77_8 prefix table
2. 50_8 or 51_8 overlay table, or a 6000 or 7600 PPU header table
3. Core image of the program

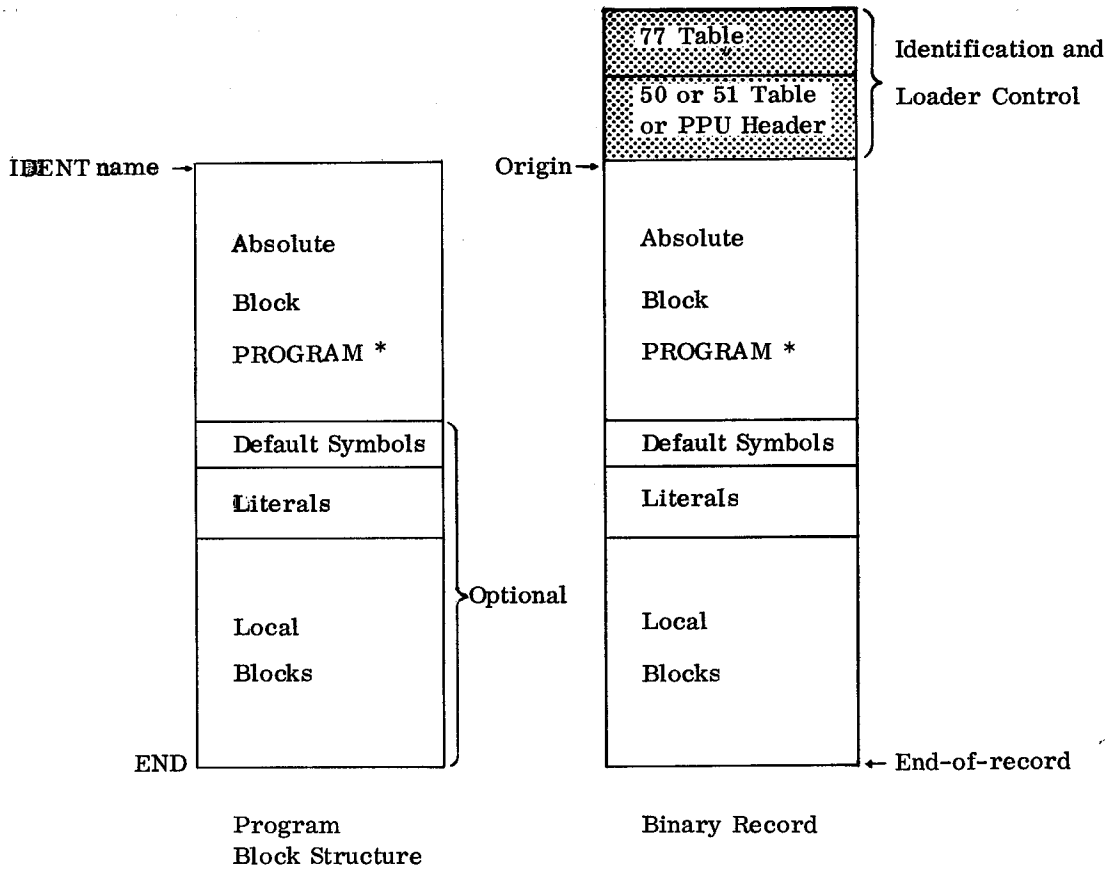
Record format is described more fully in appendix B.

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

If a complete block group is being written (everything between an IDENT and an END or between two IDENT instructions), the core 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 core image of the program or overlay ending with the value of the current origin counter.

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



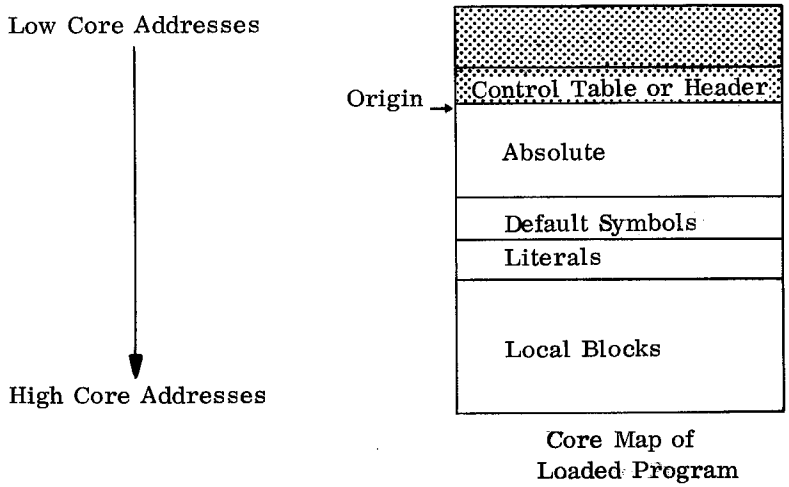


Figure 3-2. Absolute Program Structure

3.4.1 ABSOLUTE OVERLAYS

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

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

For a CPU assembly, the overlay generated is either primary or secondary as determined by the IDENT or SEGMENT pseudo instruction. The portion of the program following the first IDENT is normally the main overlay and is identified by the level numbers 0,0. Secondary overlays can be generated subsequent to the main overlay. A secondary overlay is identified by the level numbers x,y, where x is nonzero.

Conventionally, the main overlay is the first one loaded and contains calls to the operating system loader to load one or more overlays as they are required during object time execution. Any overlay can call the loader to load another overlay. Control transfers to an entry in the overlay or returns to the calling overlay according to the format of the call. (For detailed information concerning CPU loader calls, refer to the operating system reference manual.)

Because overlays are not all in core concurrently during program execution and because the sequence in which overlays are loaded and executed is beyond the scope of the assembler, it is the user's responsibility to assure that an overlay does not refer to symbols, instructions, or data that is not concurrently in core.

Although PPU overlays are not identified by level numbers, they resemble CPU overlays in all other respects.

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 appendix B.

IDENT-Type Overlays

The portions of the program from IDENT to IDENT, and IDENT to END comprise the overlays. IDENT provides the programmer with the option of specifying the overlay level numbers with each overlay, including the overlay generated by the first IDENT.

If no level number is provided for a CPU assembly, the first overlay is numbered 0,0 and any overlay after that is numbered 1,0. IDENT allows each overlay to be assigned unique numbers. Thus, the loader has a means of locating a specified overlay when several overlays are written on the same file.

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

The first IDENT causes COMPASS to generate the program or overlay identification information (appendix B) that precedes the absolute record. Upon encountering a second IDENT instruction before an END instruction, COMPASS generates output consisting of a core 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, it normally ends with the last word address. An IDENT subsequent to a SEG or SEGMENT, however, generates binary that ends at the location specified by the current origin counter. Following the core image, COMPASS writes an 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 of 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 3-3 illustrates a CPU program consisting of a main overlay and a secondary overlay. The main overlay uses the absolute block and block A. Default symbols and literals cause the assembler to generate a zero block and the literals block. Following the second nonblank IDENT instruction, the program overlay origin is set back into the block A. The overlay generates a new literals block and new blocks A, C, and D.

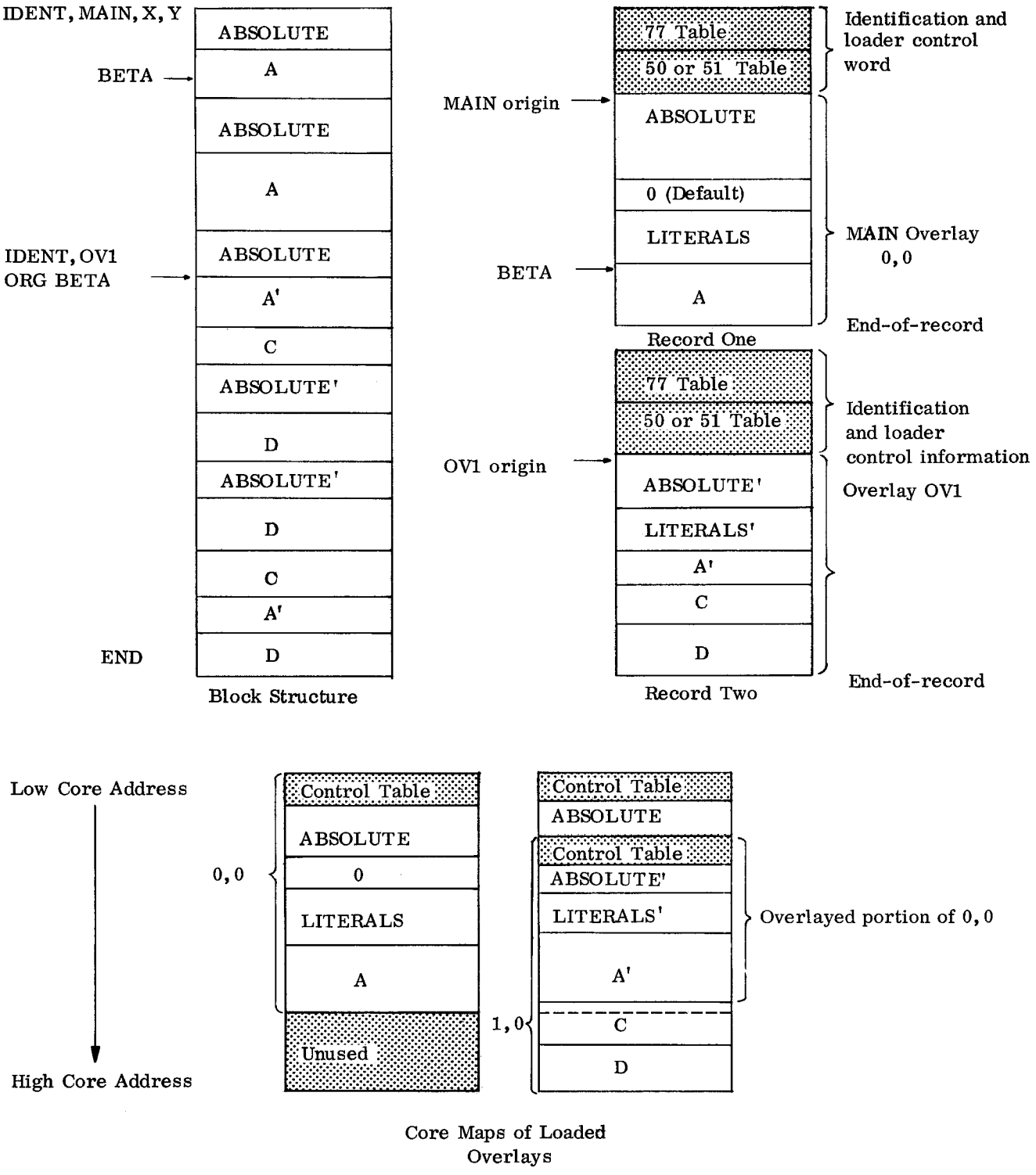


Figure 3-3. IDENT-Type Overlay Structure

SEGMENT-Type Overlays

The portions of the program from the IDENT that identifies the program to SEGMENT, from SEGMENT to SEGMENT, and from SEGMENT to END comprise the overlays. SEGMENT does not provide for unique numbering of overlays. The first overlay has the identifier 0,0. All subsequent overlays are numbered 1,0.

Upon encountering a SEGMENT instruction, COMPASS generates output consisting of a core 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-record and overlay identification information for the overlay to follow.

For SEGMENT, the last block used in the overlay is incomplete. If the overlay contains literals, it must have a user-established block as the block in use when the SEGMENT is encountered. A PPU overlay cannot contain literals. For a CPU assembly, the literals block is in the overlay that contains the end of the absolute block. It is the responsibility of the user to assure that all blocks other than the one in use are complete. The origin of the new overlay can be defined using symbols in the block in use only. SEGMENT does not clear the symbol table or reinitialize the USE table.

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 3-4 illustrates a program consisting of a main overlay and a secondary overlay. The main overlay uses the absolute block, the literals block, and block A. Default symbols cause the generation of a zero block. Following the SEGMENT, an ORG instruction sets the overlay origin back into block A, the block in use when the SEGMENT was encountered. The 1,0 overlay establishes new blocks C and D.

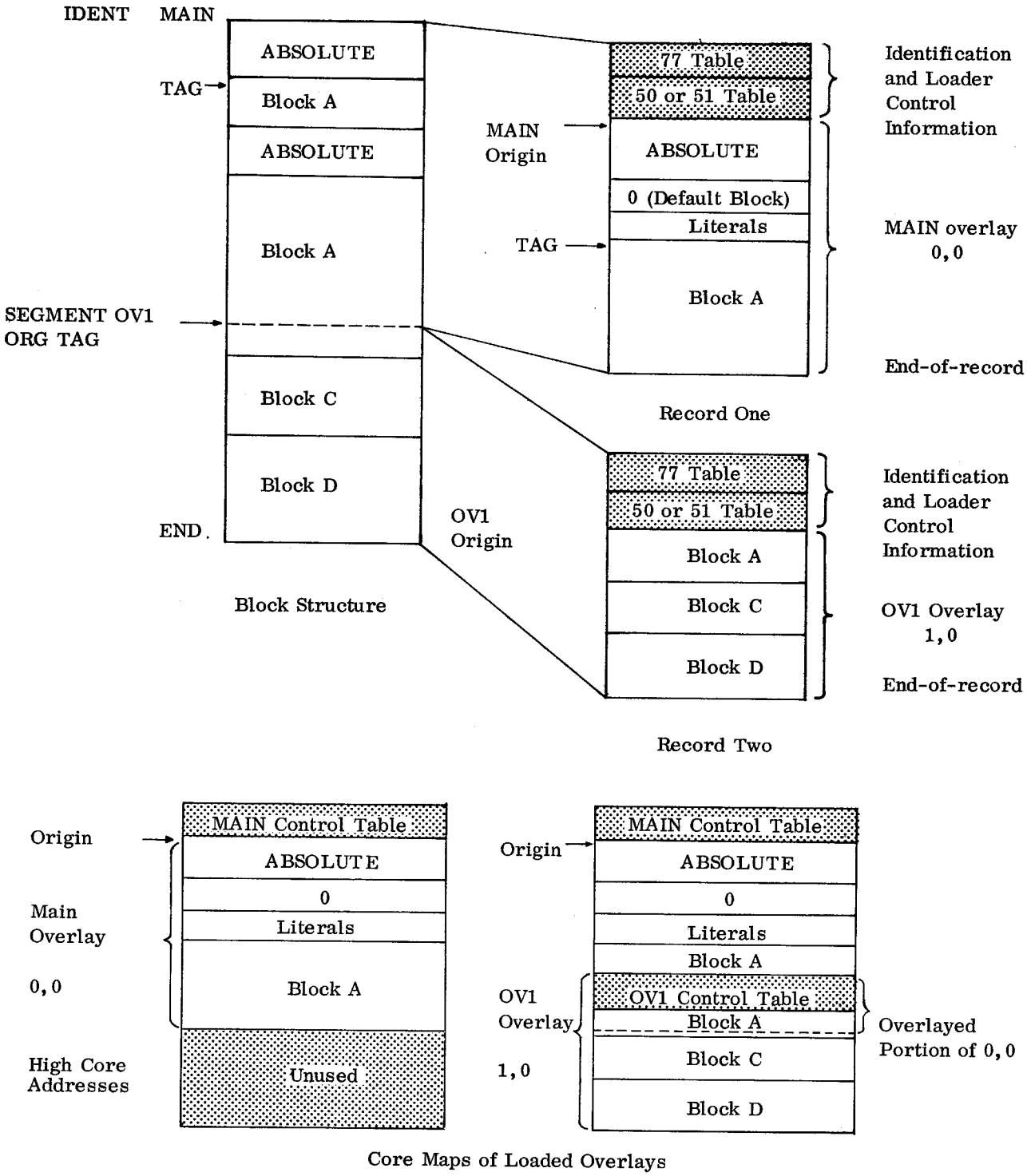


Figure 3-4. SEGMENT-Type Overlay Structure

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 51_8 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 51_8 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 51_8 table, refer to appendix B.

Overlays of this type cannot be created or loaded by the 6000 SCOPE loader.

3.4.3 PARTIAL BINARY

When a CPU absolute program or an overlay contains SEG pseudo instructions or IDENT pseudo instructions for which the parameters are omitted (blank), COMPASS writes a partial binary record consisting of the binary generated since the previous IDENT, SEGMENT, or SEG instruction. However, it does not write an end of record or a new 77_8 table. A SEGMENT, nonblank IDENT, or END instruction completes the binary record.

SEG-Type Partial Binary

By writing partial binary using SEG, the programmer can reduce the assembler storage requirements. A fatal error is issued if the user attempts to store data into a block previously written out or into a block that will be written out later.

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.

SEG does not write a complete block group. 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 illustrates how the binary for an absolute program can be written in three separate binary writes to reduce the amount of core required to assemble the program. The resulting absolute record is loaded and executed as a single program or overlay.

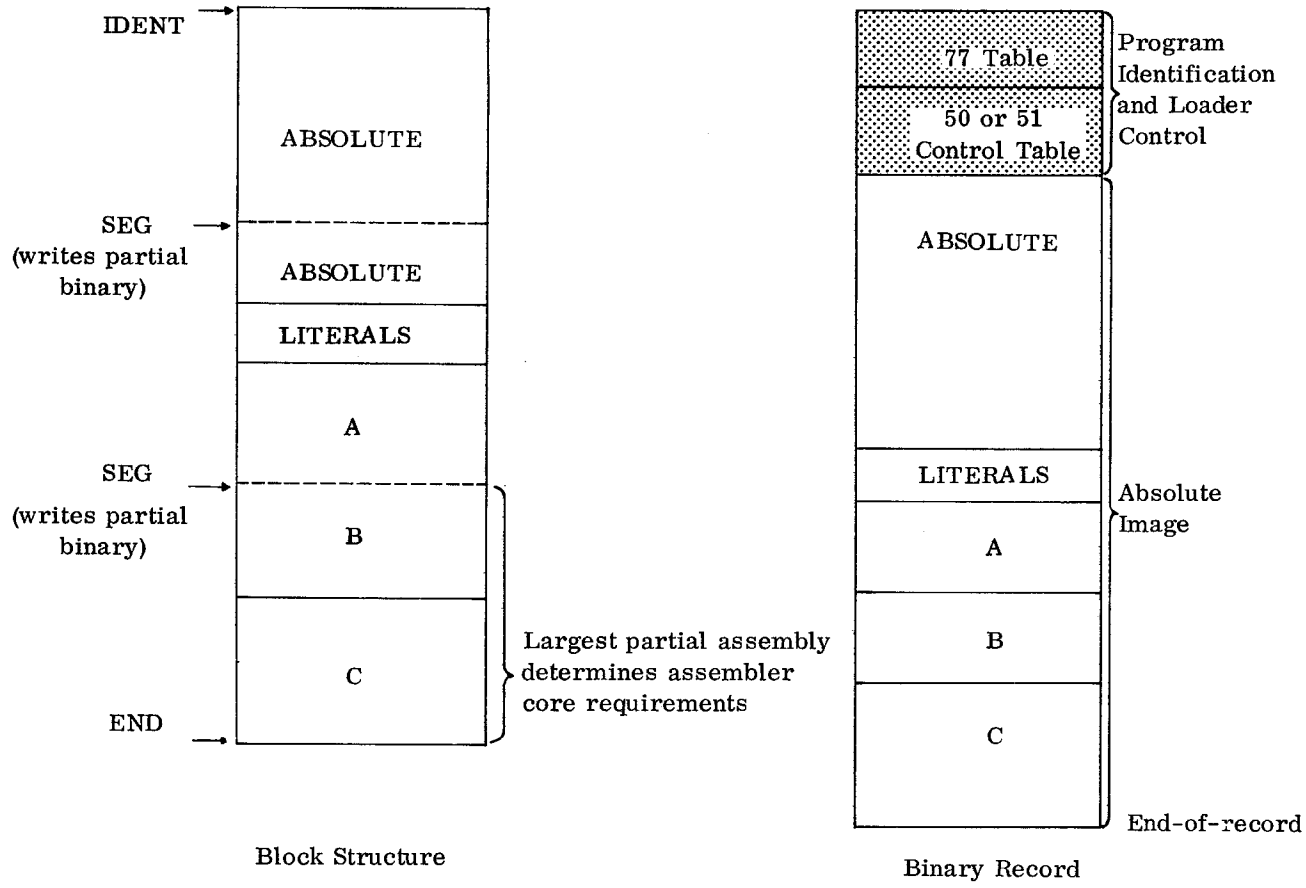


Figure 3-5. SEG-Type Partial Binary

IDENT-Type 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 record or a new 77_8 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 generated in this manner are allocated as if the 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 a 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 section. Each new section created by a blank IDENT can use the same block names as are used by the other section of IDENT-created overlays and each section 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 section written separately causes a range error.

Figure 3-6 illustrates how the binary for an overlay can be written in three discrete sections to reduce the amount of core required to assemble the program and divide the program into self-contained units. The resulting absolute record is loaded and executed as a single overlay.

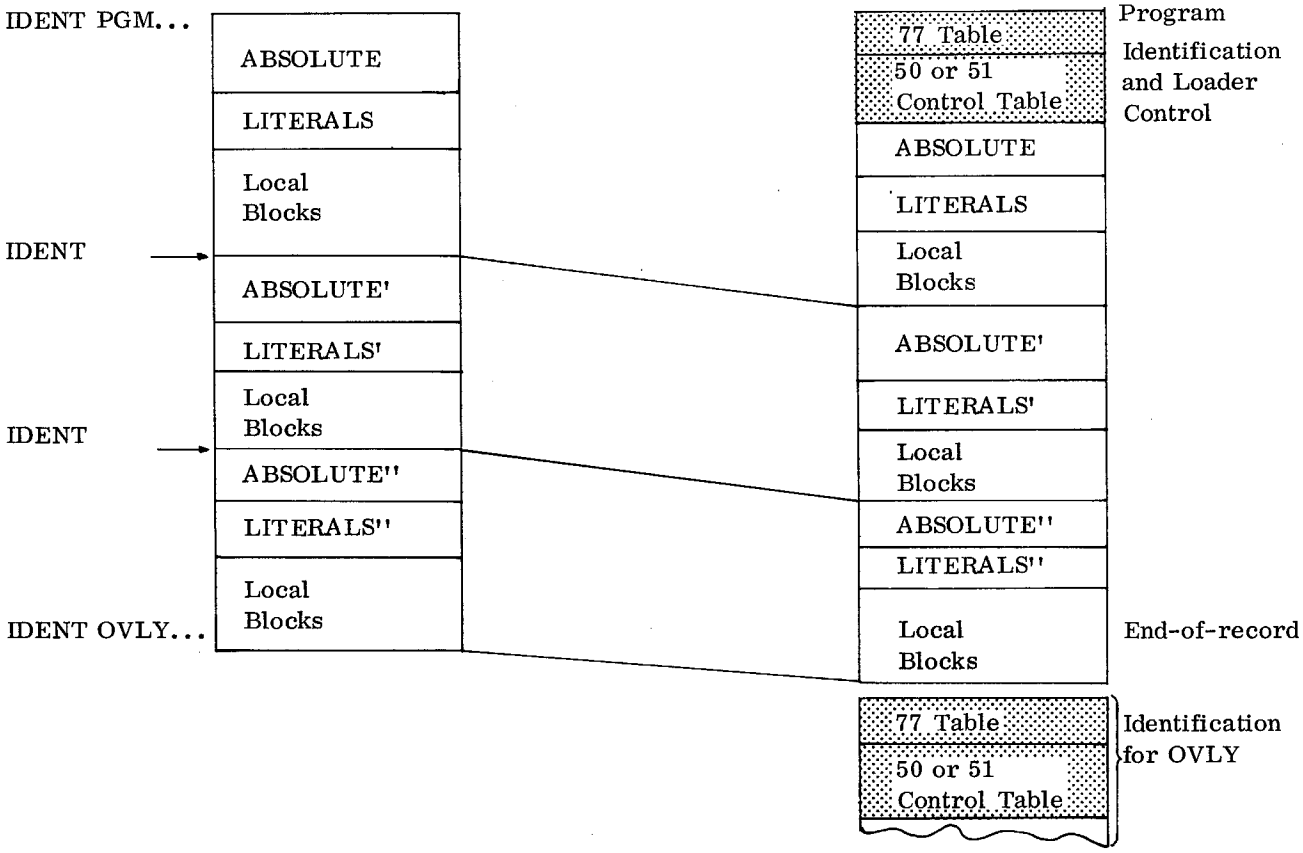


Figure 3-6. IDENT-Type Partial Binary

PSEUDO INSTRUCTIONS 4

4.1 INTRODUCTION TO PSEUDO INSTRUCTIONS

This chapter and chapters 5, 6, and 7 describe the pseudo instructions available in the COMPASS language. It is impossible to write a program in the COMPASS language without using some of the more basic pseudo instructions. The programmer who is new to the language should give special attention to these instructions:

Pseudo Instruction	Section	CPU Relocatable	CPU Absolute	PPU Absolute
IDENT	4.2.1	X	X	X
ABS	4.3.1	-	X	-
PPU or PERIPH	4.3.2 or 4.3.3	-	-	X
ORG	4.5.3	-	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, PERIPH, PPU, IDENT, SEGMENT, SEG, LCC, STEXT, COMMENT, and NOLABEL)

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

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

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

Subprogram linkage (ENTRY and EXT)

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

Assembly control (ELSE, ENDIF, IFCP and IFPP, IFop, IF, IFC, 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 comprise 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

PERIPH

PPU

STEXT

4.1.4 PERMISSIBLE ANYWHERE INSTRUCTIONS

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

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

Comments 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 COMPASS control card, the end of the source decks is indicated by an end-of-record card (7-8-9 punches in column 1).

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 1-7 characters, of which the first must be alphabetic (A-Z) and the last must not be a colon.

For a 7600 PPU assembly, name can be 1-7 characters. For a 6000-Series PPU assembly, name can be 1-3 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 comprises the overlay. For a single entry point CPU program the load address for the record is origin-1. The word at origin -1 is overlayed by the 50_8 loader control table. For a multiple entry point CPU program, the load address for the absolute record is origin-wc-1, where wc is the number of entry points in the 51_8 loader table.

For a PPU subprogram, the load address is origin-5. Five 12-bit PPU words are overlayed 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 (RAS) 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 RAS+100_8, the second subprogram starting at the first available location following the first subprogram, etc.

entry — For a 7600 PPU assembly or 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 (0-63) 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 card, IDENT must be in columns 11-15.

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     |         11         18                30
      |         IDENT      CT,CONTROL,CONTROL
      |         ABS                          ABSOLUTE CPU PROGRAM
      |         ORG        1108
      CONTROL   BSS        0                 DEFINES SYMBOL CONTROL
      |         END
```

Absolute CPU program CT will be loaded at origin address 00110_8.

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 (appendix B), 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 record 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		11	18	30
		IDENT	PROG1	
		ENTRY	BEGIN	
		•	•	
		•	•	
		•	•	
	BEGIN	SB1	1	
		•	•	
		•	•	
		•	•	
		END	BEGIN	

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
PPU	Specifies 7600 PPU binary output
PERIPH	Specifies 6000 Series PPU binary output
IDENT	Begins absolute overlay or writes partial binary record
SEGMENT	Begins absolute overlay
SEG	Writes partial binary record
STEXT	Generates systems text overlay
COMMENT	Inserts comments into the 77_8 prefix table
NOLABEL	Suppresses header information on binary output
LCC	Passes loader control information to the relocatable loader

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. Refer to appendix B for a description of the binary format.

The following instructions are illegal in an absolute program:

EXT
LCC
REP
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 (section 2.4.2).

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
           .           .
           .           .
           ORG         110B
CONTROL    BSS         0                   DEFINES SYMBOL CONTROL
           .           .
           .           .
           .           .
           END
```

4.3.2 PPU — 7600 PPU PROGRAM

A PPU instruction declares a program to be a 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 appendix B.

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

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

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 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:

Location	Code Generated
740	
760	0357

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	PPU		
	•		
	•		
TAG	BSS	20B	
	UJN	TAG-*	EXPRESSION < 37B

Location	Code Generated
740	
760	0357

LOCATION	OPERATION	VARIABLE	COMMENTS
1	11	18	30
	PPU	JUMP	
	•		
	•		
TAG	BSS	20B	
	UJN	TAG	EXPRESSION-* < 37B

4.3.3 PERIPH — 6000 SERIES PPU PROGRAM

A PERIPH instruction declares a program to be a 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, see appendix B.

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

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

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.

4.3.4 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 (see 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 record or a new 77_8 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 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 a 6000 Series PPU program, 1-3 characters; for a 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 overlayed by the 50_8 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 51_8 loader table (appendix B).

For a PPU subprogram, the load address is origin-5. Five 12-bit PPU words are overlayed 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 ($00\text{-}77_8$), ℓ_2 is the secondary level ($00\text{-}77_8$). 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 card. END dumps the last overlay or completes a partially written record. Refer to appendix B for file formats.

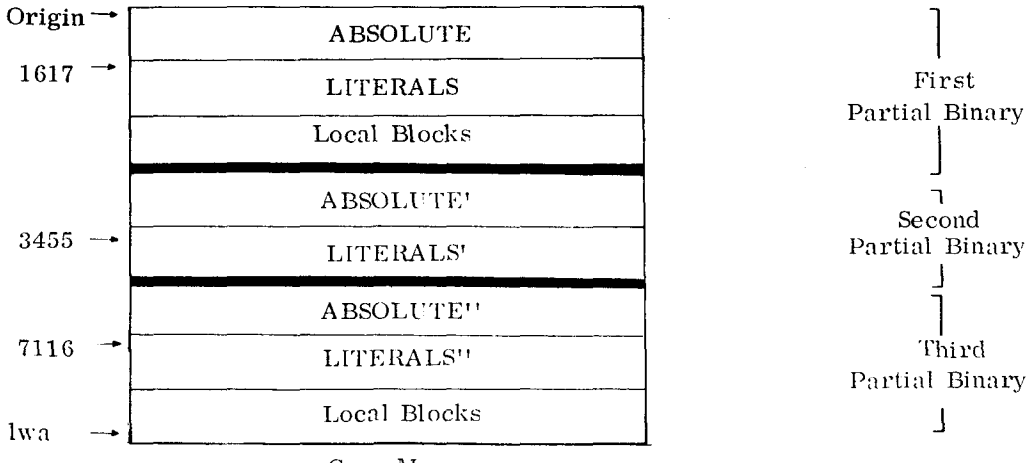
Examples:

The following program uses IDENT for overlay creation. Symbols T.OVL, 0.DMP1, etc. are defined on a system text overlay.

```
                LOCATION    OPERATION   VARIABLE            COMMENTS
                1           11          18                  30
                            IDENT       DMP.1,T.OVL,0.DMP1
                            ABS
DMP                         BASE        M
                            COMMENT     10/07/70.CONTROL CARD CALL.DMP.
                            LIST        G
                            SST                                         OVERLAY
                            ORG         T.OVL                           DMP1
                            QUAL        DMP1
                DMP         SX0         B1
                            .           .
                            .           .
                            .           .
                            QUAL        DMP2
                            IDENT       DMP2,T.OVL,0.DMP2
                            ORG         T.OVL                           OVERLAYS DMP2
                UBW2        SX0         B6+1                            THROUGH DMP8
                            .           .
                            .           .
                            .           .
                            QUAL        DMP9
                            IDENT       DMP.9,T.OVL,0.DMP9              OVERLAY
                            ORG         T.OVL                           DMP9
                            SX0         0.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         1108
                    ENT         SX0         1
                                .           .
                                .           .
                                .           .                   First
1617                            LIT         1,2,3               Partial Binary
                                .           .
                                .           .
                                .           .
                                IDENT
                                .           .
                                .           .
                                .           .                   Second
3455                            LIT         2,3                 Partial Binary
                                .           .
                                .           .
                                .           .
                                IDENT
                                .           .
                                .           .
                                .           .                   Third
7116                            LIT         1,2                 Partial Binary
                                .           .
                                .           .
                                .           .
                                END
```



Core Map

4.3.5 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 maintain compatibility with previous versions of 6000 COMPASS and to provide another way of handling literals. Use of SEGMENT is intended for CPU absolute or 6000 Series PPU assemblies. It is illegal for 7600 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 identified by the level 1, 0. END dumps the last overlay. Refer to appendix B for overlay record format. 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

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 overlayed by the 50_8 loader table.

For a PPU subprogram, the load address is origin-5. Five 12-bit PPU words are overlayed 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.

Example:

```
     LOCATION  OPERATION  VARIABLE          COMMENTS
1              11         18                30
          IDENT    SAM,ENTA
          ABS
          ORG      1108
ENTA      BSS      0                 ENTRY POINT
          .        .
          .        .
          .        .
OVLOC     BSS      0                 OVERLAY LOAD POINT
          .        .
          .        .
          .        .
SEG1      SEGMENT  STRT,ENTB
          ORG      OVLOC
          BSS      1                 LOADER TABLE
STRT      BSS      0                 FIRST WORD OF OVERLAY
          .        .
          .        .
          .        .
ENTB      BSS      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.

4.3.6 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 record or begin a new 77_8 prefix table. A SEGMENT, IDENT, or END instruction completes the binary record.

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.

See also, section 3.4.3 Partial Binary.

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		

4.3.7 STEXT — GENERATE SYSTEMS TEXT RECORD

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

The systems text record becomes available in other assemblies through use of the S or G option on the COMPASS control card (chapter 10). Through this feature, information on the systems text record need be processed only once for all COMPASS programs using the same system text record.

System text records cannot be generated and used in the same assembly batch; system text records generated by one COMPASS control card call can be used only by assemblies performed by later COMPASS control card calls.

The symbols included in the system text record 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 (that is, 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 8 characters beginning with ↓↑.

All defined micros are included in the system text record.

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 record is used as input to an assembly through the G or S option on a COMPASS control card, 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.

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. Refer to appendix B for record format.

If rname is blank, COMPASS uses the name from the IDENT instruction and generates the systems text only. Otherwise, the systems 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
               BASE       MIXED
MPRS           EQU        100
  .            .          .           SYSTEM CONSTANTS, SYMBOLS,
                                      AND COMMUNICATIONS AREAS
  .            .          .
  .            .          .
TRTS           EQU        7777
IXX/X          OPDEF      I,J,K
  .            .          .
  .            .          .           SYSTEM-DEFINED MACROS
  .            .          .           AND OPDEFS
               ENDM
SYSCOM         MACRO      N
  .            .          .
  .            .          .
  .            .          .
               ENDM
DATE           MICRO      1,10,*...*
  .            .          .
  .            .          .           SYSTEM-DEFINED MICROS
  .            .          .
               END
```

4.3.8 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 operating system reference manual.

A location field symbol, if present, is ignored.

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

4.3.9 COMMENT—PREFIX TABLE COMMENT

The COMMENT pseudo instruction inserts the character string specified in the variable field into the third through fourteenth words of the prefix table in the binary record. The prefix table, and thus the comment, is ignored by the loader but identifies the record for use by 7000 SCOPE utility programs such as LIBEDIT and CATALOG. If a subprogram contains more than one COMMENT instruction, the new comments are appended to the table for the most recent binary control card. If the subprogram contains a NOLABEL instruction, this instruction is meaningless. COMMENT, instructions following SEG and blank IDENT pseudo instructions are ignored without notification.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	COMMENT	string

string — First nonblank character following COMMENT to the last nonblank character; no more than 120 characters.

A location field symbol, if present, is ignored. Refer to section 4.3.4 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 or for writing Chippewa format CPU programs.

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 (77_8) tables, but retains the other program header tables.

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

Prefix tables (77_8)
Overlay control tables (50_8)
Multiple entry point tables (51_8)
PPU header control tables

A location field symbol, if present, is ignored. NOLABEL is not permitted in a relocatable CPU assembly.

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.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
mname	BASE	mode

mname — Optional 1-8 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 1-8 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 15_8; constant 15D is evaluated as 17_8. 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. 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 used 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 illustrates the affect of BASE on a VFD instruction that defines a 48-bit field containing 10_8.

Code Generated

```
                    D+0
0000000000000010
                    0+D
                 0000
000000000010
                    D+M
              00000000
00000010
```

```
      LOCATION  OPERATION  VARIABLE    COMMENTS
1                11         18          30
          BASE    0
          VFD     60/10
          .       .
          .       .
          BASE    D
          VFD     48/8
          .       .
          .       .
          BASE    M
          VFD     48/10
```

The following example illustrates the micro capability of BASE:

```
      LOCATION  OPERATION  VARIABLE    COMMENTS
1                11         18          30
D+M   SAVEB     BASE    M           SAVE BASE IN USE
        .         .       .
        .         .       .         CODE USING BASE M
        .         .       .
                  BASE    *SAVEB*     RESTORE SAVED BASE
M+D               BASE    D       RESTORE SAVED BASE
        .         .       .
        .         .       .
        .         .       .
```

4.4.2 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† display, enternal BCD, or internal BCD, codes. If no CODE instruction is used, COMPASS generated display code. Codes are given in appendix D.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	CODE	char

†American Standard Code for Information Interchange.

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

A	ASCII
D	Display
E	External BCD
I	Internal BCD

A location field symbol, if present, is ignored.

Example:

Code Generated

```
172524202524000000000
                    D↑A
576564606564000000000
                    A↑E
462423472423000000000
                    E↑I
466463476463000000000
                    I↑D
172524202524000000000
```

```
      LOCATION    OPERATION   VARIABLE          COMMENTS
1                 11          18                30
      OLOUTPUT    DATA
      ASCII       CODE
      OLOUTPUT    DATA
      EXTERNAL BCD CODE
      OLOUTPUT    DATA
      INTERNAL BCD CODE
      OLOUTPUT    DATA
      DISPLAY     CODE
      OLOUTPUT    DATA
```

4.4.3 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 use in a subprogram all symbols are defined as global.

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 preceded by a blank QUAL and followed by a QUAL *.

Default symbols and linkage symbols are not qualified.

LOCATION	OPERATION	VARIABLE SUBFIELDS
	QUAL	qualifier

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

qualifier — 1-8 character name, the first character of which cannot be $ 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 most recent QUAL. Two or more consecutive QUAL * instructions have the same effect as a single QUAL *.

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.

A location field symbol, if present, is ignored.

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 *.

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

Examples:

```
      LOCATION  OPERATION  VARIABLE        COMMENTS
1                11         18              30
                 QUAL       PASS1
BCDE             SX6        F               BCDE QUALIFIED BY PASS1
                 .          .
                 .          .
                 EQ         LOC1
                 QUAL       PASS2
BCDE             EQU        LOC2            BCDE QUALIFIED BY PASS2
                 QUAL                       SYMBOLS GLOBAL FROM NOW ON
                 .          .
                 .          .
                 .          .
GLOB             BSS        0               GLOB IS GLOBAL
                 .          .
                 .          .
                 RJ         /PASS1/BCDE     JUMP TO PASS1 ROUTINE
                 .          .
                 .          .
                 RJ         /PASS2/BCDE     JUMP TO PASS2 ROUTINE
```

Location Code Generated

```
      LOCATION  OPERATION  VARIABLE        COMMENTS
1                11         18              30
                 TAB        MACRO     BLOCK,KWAL
                            USE       BLOCK
                            QUAL      KWAL
                 TAG1       BSS       10B
                 TAG2       VFD       60/-1
                            USE       *
                            QUAL      *
                            ENDM
                            .
                            .
                            .
                            TAB       ONE,ONE
                            USE       ONE
                            QUAL      ONE
10044
10054  7777777777777777776  TAG1      BSS       10B
                 TAG2       VFD       60/-1
                            USE       *
                            QUAL      *
                            ENDM
                            TAB       TWO,TWO
                            USE       TWO
                            QUAL      TWO
10055
10065  7777777777777777776  TAG1      BSS       10B
                 TAG2       VFD       60/-1
                            USE       *
                            QUAL      *
                            ENDM
```

4.4.4 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= described under Data Generation.

4.4.5 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 evaluatable 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.

A location field symbol, if present, is ignored.

Example:

```
      1           LOCATION    OPERATION   VARIABLE    COMMENTS
      1                       11          18          30
44                            COL         36
                              USE                     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.

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 1-8 characters, excluding blank or comma. 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, or ORG. 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.

In particular, 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 ////.

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, and ORG 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:

```
Location        Code Generated

                                    LOCATION  OPERATION  VARIABLE  COMMENTS
                                    1         11         18        30
                                              USE
13  0100000000                      GAMMA     RJ         ALPHA     BLOCK 0 IN USE
                                              USE        DATA1     BLOCK DATA1 IN USE
35  17204000000000000000            SAB       DATA       1.0
                                              USE        *         RESUME USE OF BLOCK 0
14  5130000000                                SA3        SAM
```

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

```
Location        Code Generated

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

The USELCM pseudo instruction establishes or resumes use of a block assigned to 6000 Series Extended Core Storage (ECS) or 7600 Large Core Memory (LCM). For all but COMPASS Version 2 under SCOPE 2, data generating instructions and symbolic machine instructions are illegal; only storage reservation pseudo instructions are allowed, that is, BSS and ORG. The USELCM instruction is illegal in PPU assemblies.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	USELCM	block

block — Required; identifies the ECS/LCM block as follows:

name	Common block name; 1-7 characters excluding blank or comma. The first and last characters must not be colons. This form is illegal for SCOPE 2.
//	Blank common block; allowed only by SCOPE 2. This block cannot contain data. The only storage instructions that can follow are BSS and ORG. The BSSZ instruction is illegal here because it presets the block to zeros.
/name/	Labeled common block; allowed only by SCOPE 2. A name can be 1 - 7 characters and cannot include blank or commas. The first and last characters must not be colons.
*	Block in use prior to current USE, USELCM, or ORG. Each USELCM * or USE * resumes use of the most recent entry in the USE table and removes it from the table.

A location field symbol, if present, is ignored.

The blocks provide a means of symbolically addressing (and for SCOPE 2, presetting) the job's ECS/LCM field from a CPU program. Use of a block can be resumed through use of ORG or USELCM.

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
```

†This form is illegal under SCOPE 2.

4.5.3 ORG — 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.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ORG	exp

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

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 into the literals block.

A location field symbol, if present, is ignored.

```
|  LOCATION  | OPERATION | VARIABLE        | COMMENTS
|1           |11         |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            |
|            | •         | •               |•
|            | •         | •               |•
|            | 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
|            | •         | •               |
```

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 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 connot 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
                    USE        //
          COMMON    BSS        10000B        RESERVE 512 WORDS OF BLANK COMMON
                    USE        *
                    .          .             .
                    .          .             .
                    .          .             .
                    SA6        COMMON+500B
                    .          .             .
                    .          .             .
                    .          .             .
          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, or USELCM instruction resets the location counter to the value of the origin counter.

A LOC instruction does not cause the assembler to switch from the current block to another. LOC causes the next instruction in the block 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, 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
		1	T1	EQU	1	
		0	CH	EQU	0	
7100				ORG	7100	
7100			RES	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	*O-RES+PPR	
L 215				BSS	240-*	
L 240			END	BSS		
7240				LOC	*O	

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 evaluatable 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.

CAUTION

If the POS instruction is used on a word containing relocatable or external addresses, undefined results may 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
	sym	EQU	exp
or			
	sym	=	exp

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

exp — An evaluatable expression. Any symbols in the expression must be previously defined or declared as external. The expression cannot contain symbols prefixed by =S or =X 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     EQU        *-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 evaluatable expression. The expression cannot include symbols as yet undefined and cannot contain symbols prefixed by =S or =X 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	$exp_1, exp_2, \ldots, exp_n$

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

exp_i — An evaluatable expression. Any symbols in the expression must be previously defined. The expression cannot contain symbols prefixed by =S or =X 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	$exp_1, exp_2, \ldots, exp_n$

sym — A location symbol is required (section 2.4).

exp_i — An evaluatable expression. Any symbols in the expression must be previously defined. The expression cannot contain symbols prefixed by =S or =X 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.

The symbols are in a systems overlay or on a systems text file accessed through the S or G list options on the COMPASS control card (section 10.1.2).

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	SST	$sym_1, sym_2, \ldots, sym_n$

sym_i One or more symbols on the file that are not to be defined.

A location field symbol, if present, is ignored.

Refer to section 4.3.4 for an example of use.

4.7 SUBPROGRAM LINKAGE

Pseudo instructions ENTRY 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 — 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.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	ENTRY	$sym_1, sym_2, \ldots, sym_n$

sym_i 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 be listed on an EXT pseudo instruction). Entry point symbols must be unqualified (section 2.4.5).

A location field symbol, if present, is ignored.

A list of all entry points declared in the subprogram precedes the assembly listing.

Example:

```
Location        Code Generated

                                  IDENT   CT,CONTROL,CONTROL
                                  ABS
                                  ENTRY   MODE
                                  ENTRY   ONSW
                                  ENTRY   OFFSW
                                  ENTRY   ROLLOUT
                                  ENTRY   SETPR
                                  ENTRY   SETTL
                                  ENTRY   SWITCH
110                               ORG     1100B
110                     CONTROL   BSS     0
110   5120000100        MODE      SA2     ACTR
                  73720           SX7     X2
111   5110000002                  SA1     2
                                    .       .
                                    .       .
                                    .       .
```

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.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	EXT	$sym_1, sym_2, \ldots, sym_n$

sym_i 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 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).

Only the first word appears on the listing.

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	$item_1, item_2, \ldots, item_n$

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.

$item_i$ — 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 $item_1$ appears on the listing.

Examples:

```
Location        Code Generated

552   140717000000000000000
553   400000000000000000000
554   031715201114050000000
555   172524202524000000000
556   000000000000000000000
557   172051463146314631463146
560   164031463146314631463146
```

```
      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
```

```
Location  Code Generated

                        0,0     PERIPH
                                BASE    0
                                .       .
                                .       .
1250      7070          DAT     DATA    7070,-7,0,1R1
1251      7770
1252      0000
1253      0034
1254      5501                  DATA    2C A,0LEF
1255      0000
1256      0506
1257      0123                  DATA    0123,-4
1260      7773
1261      0401                  DATA    H*DATA*
1262      2401
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30

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 evaluatable 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.

Examples:

```
Location     Code Generated

  561   0705160522012405535
  562   5503202555271722042
  563   0705160522012405535
  564   5503202555271722042
  565   0000000000000000000
```

```
      LOCATION   OPERATION   VARIABLE          COMMENTS
1                 11          18                30
      ONE        DIS         2,GENERATE 2      CPU WORDS

      TWO        DIS         ,*GENERATE 2      CPU WORDS*
```

Location	Code Generated	
		D.M
1402	0705	
1403	1605	
1404	2201	
1405	2405	
1406	5534	
1407	3355	
1410	2020	
1411	5527	
1412	1722	
1413	0423	
1414	0705	
1415	1605	
1416	2201	
1417	2405	
1420	5534	
1421	3355	
1422	2020	
1423	5527	
1424	1722	
1425	0423	
1426	0000	

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		PPU		
		BASE	M	
		DIS	10,GENERATE 10 PP WORDS	
		DIS	,*GENERATE 10 PP WORDS*	

4.8.4 LIT — DECLARE LITERAL VALUES

A LIT pseudo instruction generates data words in the literals block. This instruction and the = prefix to a data item provide the only means of generating data in the literals block. The LIT pseudo instruction assures sequential entries for a table of values.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	LIT	$item_1, item_2, \ldots, item_n$

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

$item_i$ — 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
                                  1         11         18                    30
                611               POOL      LIT        3.1,1.59265,2.7182182,57.2957795EE1
                    CONTENT OF LITERALS BLOCK.
000611  1721614631463146 3146    OQ[-Y-Y-Y-
000612  1720627557644177 6271    OP]≥.≠6;]↓
000613  1721533735113601 4426    OQ$42I3A9V
000614  1731436365144066 3121    OY8!↑L5♥YQ
000615  1651333303354057 6566    N(00C25.↑♥
```

```
Location  Code Generated

                                  LOCATION  OPERATION  VARIABLE              COMMENTS
                                  1         11         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	$item_1/exp_1, item_2/exp_2, \ldots, item_n/exp_n$

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.

$item_i$ — 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, $item_i$ is assumed to be decimal notation.

exp_i — An absolute, relocatable, or external expression, the value of which will be inserted into the field specified by $item_i$. For CPU assemblies, if exp_i is a relocatable or external value, $item_i$ must be at least 18 bits and must end at bit 30, 15, or 00, the only legal positions for addresses in 30-bit CPU instructions.

The expression is evaluated using the specified field size. Character constants are right or left adjusted in the field according to the type of justification indicated.

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:

```
Location    Code Generated

                                  31
566     2401020000023000551
567     0000005665555555555
570     777777774
                 000000000000
571     1117240155015555531
572     0000015052323010705
573     031117000000033
```

```
|  | LOCATION | OPERATION | VARIABLE | COMMENTS |
|1 |          | 11        | 18       | 30       |
|  | ALPHA    | SET       | 25       |          |
|  | TABLE    | VFD       | 36/3CTAB,6/10,18/TABLOC |
|  |          | VFD       | 30/*-1,30/5H      ,ALPHA/-0 |
|  |          |           |          |          |
|  |          | VFD       | *P/      |          |
|  |          | VFD       | 30/0HIOTA,6/1RA,24/0AX+1 |
|  |          | VFD       | 60/0RMESSAGE,30/3LCIO,15/0R0 |
```

```
Location    Code Generated

                                  0rM
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       |          |          |
|  |          | BASE      | M        |          |
|  | N4       | 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 more 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.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
sym	CON	$exp_1, exp_2, \ldots, exp_n$

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

exp_i — 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:

Location	Code Generated
1460	0000
1461	0006
1462	0003
1463	2204
1464	0024
1465	0000
1466	0006
1467	0003
1470	2172
1471	0024

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	MSG1	CON	0	
		CON	6	
		CON	3	
		CON	FAIL	
		CON	20	
	MSG2	CON	0	
		CON	6	
		CON	3	
		CON	PASS	
		CON	20	

Location		Code Generated
	574	
L	0	
L	0	0000000000000000055
L	1	0000000000000000062
L	2	0000000000000000064
L	3	0000000000000000060
L	75	0000000000000000066
L	76	0000000000000000076
L	77	0000000000000000055
	674	

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	TAD	BSS	0	
		LOC	0	
		CON	1R	00
		CON	1R]	01
		CON	1R≠	02
		CON	1R≡	03
		•	•	•
		•	•	•
		•	•	•
		CON	1R↓	75
		CON	1R¬	76
		CON	1R	77
		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 the second subfield 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
		B1=1		
		R=	B3,2	
66311		SB3	B1+B1	
		R=	B3,3	
6130000003		SB3	3	

2. R= used with B1≠1

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	TAG	R=	X5,-1	
7150777776	TAG	SX5	-1	

3. Expression is same as register designator:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	REG	MICRO	1,,*B5*	
		R=	B5,≠REG≠	
		R=	B5,B5	

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

4.8.8 REP AND REPI—GENERATE LOADER REPLICATION TABLE

The REP 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.

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 REPI, data must be in previously assembled text.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	REP 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.

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 or BSS)
3. REP 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 REPI, data must be in previously loaded text

Example:

```
Location        Code Generated

                                    10
5017    000000000000000015
5020    000000000000000020
5021    000000000000007070
5022    000000000000000001
5023    000000000000000005
5024    172163000000000000
                                    13
5251
```

```
      LOCATION  OPERATION  VARIABLE          COMMENTS
1               11         18                30
      RC        =          10
                USE        NEWP
      BA        DATA       15,20,70708,1,5,3.14
      I         EQU        *-BA+5
                USE        DBLOCK
      DA        BSS        RC*I
                USE        *
                REPI       S/BA,D/DA,B/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 (IFCP and IFPP)

Compare values of two expressions (IFop)

Compare values of two character strings (IFC)

Test the attribute of a single symbol or an expression (IF)

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

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	ℓnct

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

ℓnct — Optional absolute 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 ℓnct 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 IFCP AND IFPP — TEST ENVIRONMENT

The IFCP instruction tests for a CPU assembly; the IFPP instruction tests for a PPU assembly. When the test is satisfied, COMPASS assembles instructions in the IF range. When the test is not satisfied, COMPASS skips the instruction.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	IFCP or IFPP	ℓnct

ifname — Optional 1-8 character name.

ℓnct — Optional absolute evaluatable 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 the first ENDIF or ELSE encountered, whether named or unnamed.
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.

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	$exp_1, exp_2, ℓnct$

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, and so forth.
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_i — 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.

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

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 the first ENDIF or ELSE encountered, whether named or unnamed.
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,408
      |         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_8 words (the range of a short jump) prior to the occurrence of this code. Otherwise, the NJN and LJM are assembled.

4.9.5 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.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	IF	**att, exp, fnct**

ifname Optional 1-8 character name

att Specifies attribute test. A minus prefix to the attribute causes assembly on the false rather than the true condition.

att	Condition causing assembly
SET	The symbol given in the second subfield was defined by a SET, MAX, MIN, or MICCNT
-SET	The symbol given in the second subfield was defined other than by a SET, MAX, MIN, or MICCNT
ABS	The expression in the second subfield reduces to a value that is not relocatable or external
-ABS	The expression in the second subfield reduces to either a relocatable or an external address
REL	The expression in the second subfield reduces to a local or common relocatable address
-REL	The expression in the second subfield does not reduce to a local or common relocatable address
REG	The expression in the second subfield contains one or more register names
-REG	The expression in the second subfield does not contain a register name
COM	The expression in the second subfield reduces to a common relocatable address
-COM	The expression in the second subfield is not a common relocatable address
EXT	The expression in the second subfield contains one or more external symbols
-EXT	The expression in the second subfield does not contain an external symbol
LCM	The expression reduces to an LCM address
-LCM	The expression does not reduce to an LCM address
LOC	The expression reduces to a program relocatable address
-LOC	The expression does not reduce to a program relocatable address

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
MIC	The name in the second subfield is a micro
-MIC	The second subfield does not contain a micro name
SST	The second subfield does not contain a system symbol
-SST	The second subfield contains 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.

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

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 the first ENDIF or ELSE encountered, whether named or unnamed.
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.

Examples

```
      LOCATION  OPERATION  VARIABLE        COMMENTS
1                 11         18              30
      ABLE      BSS        20
        .       .          .
        .       .          .
        .       .          .
      TEST      IF         REL,ABLE+15
        .       .          .
        .       .          .
        .       .          .
      TEST      ENDIF
                IF         COM,DTA,2       ERRONEOUS, DTA AS YET UNDEFINED
                .          .
                .          .
                .          .
                USE        //
      DTA       BSS        1
```

4.9.6 IFC — COMPARE CHARACTER STRINGS

The IFC pseudo instruction compares two character strings according to the operator specified and assembles instructions in the IF range if the comparison is satisfied.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
ifname	IFC	op, d$string_1$d$string_2$d, ℓnct

ifname — Optional 1-8 character name

d — Delimiting character. Characters between the first and second occurrence of this character constitute the first character string; characters between the second and third occurrence constitute the second character string.

op — Specifies comparative test:

op	Condition causing assembly
EQ or -NE	$string_1$ has the same value as $string_2$
NE or -EQ	$string_1$ does not equal $string_2$
GT or -LE	$string_1$ is greater than $string_2$
GE or -LT	$string_1$ is greater than or equal to $string_2$
LT or -GE	$string_1$ is less than $string_2$
LE or -GT	$string_1$ is less than or equal to $string_2$

$string_i$ — Character string. When IFC is within a macro definition, each character string can be a formal parameter.

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

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 the first ENDIF or ELSE encountered, whether named or unnamed.
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 $string_1$ is compared with the corresponding character in $string_2$ 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             |
|          | IFC       | EQ,**P2*,1        |
| P        | ERR       |                   | FLAG ERROR
|          | .         |                   |
|          | .         |                   |
|          | .         |                   |
|          | ENDM      |                   |
```

The following example illustrates an invalidly terminated character string. The asterisk was omitted following P1 causing an error flag when the comma is interpreted.

```
| LOCATION | OPERATION | VARIABLE          | COMMENTS
|1         |11         |18                 |30
|          | IFC       | EQ,*0D*P1,2$P2
```

4.9.7 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,ℓnct
ifname	IFMI	exp,ℓnct

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.

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

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 the first ENDIF or ELSE encountered, whether named or unnamed.
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 nonzero multiple of 60, otherwise it is the address expression value modulo 60.

```
LOCATION  OPERATION  VARIABLE          COMMENTS
1         11         18                30
MXQ       OPDEF      REG,VAL
          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,6/A
          ENDM
```

Example of call:

```
Code Generated

                    LOCATION  OPERATION  VARIABLE          COMMENTS
                    1         11         18                30
                              MX6        -52
7777713             ↑+000001  SET        -52
7777713             ↑+000001  SET        ↑+000001-↑+000001/60D*60D
                              IFPL       ↑+000001,3
                              IFEQ       ↑+000001,0,3
                              IFLE       -52,0,1
                              SKIP       1
          10        ↑+000001  SET        ↑+000001+60D
43610                         VFD        6/43B,3/6,6/↑+000001
                              ENDM
```

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	ℓnct

ifname — Optional 1-8 character name

ℓnct — Optional absolute evaluatable 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 the first ENDIF or ELSE encountered, whether named or unnamed.

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.

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
1                 11         18              30
      NNN       MACRO      P1,P2,P3,P4
                IFEQ       P1,0
      A         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.

xx	Error Condition
NG	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

Location	Code Generated
7447	
	7777447
7462	

```
      LOCATION  OPERATION  VARIABLE        COMMENTS
1                11         18              30
                  PERIPH
                  .
                  .
      LASTTAG     BSS      0
      R           ERRPL    LASTTAG-7777
                  END
```

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 card (section 10.1.2) is zero, or

When the list option parameter (LO) is used on the COMPASS control card.

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 card. 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 is 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	Lists fatal and nonfatal errors and summarizes 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 in the listed statements.

Formats:

LOCATION	OPERATION	VARIABLE SUBFIELDS
	LIST or LIST	$op_1, op_2, \ldots, op_n$ *

A location field symbol, if present, is ignored.

op_i

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 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 non-referenced symbols

This option is normally selected. Cancellation of this option causes any non-system 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 assembly end, 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 non-referenced 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.

An asterisk in the variable field causes selection of the options specified by the previous LIST pseudo instruction. Two or more consecutive LIST * instructions produce the same results as one LIST *.

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 causes by an XTEXT within a system macro call is listed only when both X and S are selected. To obtain a listing showing ↦ and ≠ marks removed from external text inside a DUP range requires that A, X, and E all be selected.

Example:

```
                                    LOCATION  OPERATION  VARIABLE      COMMENTS
                                    1         11         18            30
                                              LIST       A
                                              DATA       1.3↦EE
0  17205146314631463146                       DATA       1.3EE
                                              LIST       D
                                              DATA       1.3↦EE
2  17205146314631463146                       DATA       1.3EE
3  16403146314631463146
                                              LIST       -A,-D
4  17205146314631463146                       DATA       1.3↦EE
                                              LIST       *
                                              DATA       1.3↦EE≠≠
6  17205146314631463146                       DATA       1.3EE
7  16403146314631463146
```

4.11.2 EJECT—EJECT PAGE AND BEGIN NEW SUB-TITLE

The EJECT pseudo instruction advances printer paper to a new page before printing. Then, page headings are printed and listing continues.

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-TITLE

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.

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	SPACE	scnt, 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.

scnt — 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, scnt is assumed to be decimal. If scnt 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 scnt + 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 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.

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                 11         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

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.

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.

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	$sym_1, sym_2, \ldots, sym_n$

sym_i — 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. Alternatively, sym_i can be a non-blank qualifier symbol enclosed by slant bars, /qualifier/, in which case all symbols qualified by the specified qualifier are suppressed from the sumbolic 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 only if the X list option is selected.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
name	**CTEXT**	**string**

name — If X list option is selected, name (optional) 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 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.

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.

DEFINITION OPERATIONS 5

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_8; it is not compressed. Two or more embedded spaces are replaced in the image as follows:

2 spaces replaced by 5555_8
3 spaces replaced by 0002_8
4 spaces replaced by 0003
. . .
. . .
. . .
64 spaces replaced by 0077_8
65 spaces replaced by 007755_8
66 spaces replaced by 00775555_8
67 spaces replaced by 00770002_8, 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† random-access program library file.

Format:

LOCATION	OPERATION	VARIABLE SUBFIELDS
file	XTEXT	rname

† SCOPE 2 does not support MODIFY format.

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 card (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 logical record. If rname is not blank, it is the name of the record to be read. The file must be an indexed file with named records, a random-access program library file in UPDATE format, a random-access program library file in MODIFY† format, or a random access text record.

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 a 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 record may be a COMMON deck or a text record.

COMPASS reads source statements to an end-of-record 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.

†SCOPE 2 does not support MODIFY format.

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.

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 card 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
                                 1         11         18                30
                                           MACRO      TABLE,TNAM,EQIV
                                           IFC        EQ,**EQIV*
                                 TNAM      EQU        *-ORIGINS
                                 O.TNAM    CON        BUCKET
                                           ELSE       2
                                 TNAM      EQJ        EQIV
                                 O.TNAM    EQU        O.EQIV

                                           RMT
                                 L.TNAM    EQU        TNAM+SIZES
                                           RMT
                                           .
                                           .
                                           .
                                           ENDM
                                           .
                                           .
                                           .
4727                             INTER     TABLE
                                           IFC        EQ,***                    TABLE
                      1331       INTER      EQU        *-ORIGINS                TABLE
4727  0000000000000032304        O.INTER    CON        BUCKET                   TABLE
                                           ELSE       2                        TABLE
                                                                               TABLE
                                           RMT                                 TABLE
                                 L.INTER    EQU        INTER+SIZES              TABLE
                                           RMT                                 TABLE
                                           ENDM                                TABLE
4730                             LASTAB    TABLE
                                           IFC        EQ,***                    TABLE
                      1332       LASTAB      EQU        *-ORIGINS              TABLE
4730  0000000000000032304        O.LASTAB    CON        BUCKET                 TABLE
                                           ELSE       2                        TABLE
                                                                               TABLE
                                           RMT                                 TABLE
                                 L.LASTAB    EQU        LASTAB+SIZES           TABLE
                                           RMT                                 TABLE
                                           ENDM                                TABLE
4731                             NRTAB     TABLE      LASTAB
                                           IFC        EQ,**LASTAB*              TABLE
                                           ELSE       2                        TABLE
                      1332       NRTAB      EQU        LASTAB                 TABLE
                      4730       O.NRTAB    EQU        O.LASTAB               TABLE
                                                                               TABLE
                                           RMT                                 TABLE
                                 L.NRTAB    EQU        NRTAB+SIZES            TABLE
                                           RMT                                 TABLE
                                           ENDM                                TABLE
                                           .
                                           .
                                           .
                                           HERE
                      4672       L.INTER    EQU        INTER+SIZES            *RMT*
                      4673       L.LASTAB    EQU        LASTAB+SIZES          *RMT*
                      4673       L.NRTAB    EQU        NRTAB+SIZES            *RMT*
```

In the following example, assembly of the RMT sequence is caused by the END statement.

```
                    LOCATION  OPERATION  VARIABLE                COMMENTS
                    1         11         18                      30
                              RMT
                    FLD       DECMIC     BUF+BUFL-WSA+ENDS
                    PRS       LIT        C*≠FLD≠ DECIMAL REQUIRED.*
                              RMT
                              LIST             C
                    FLD       DECMIC     BUF+BUFL-WSA+ENDS          *RMT*      1
                    PRS       LIT        C*≠FLD≠ DECIMAL REQUIRED.* *RMT*      1
100005012           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,ℓnct

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 ENDD.

rep — Absolute evaluatable 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.

ℓnct — An evaluatable expression specifying an integer count of the number of statements to be assembled repeatedly. When base mode is M, COMPASS assumes that ℓnct 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 ℓnct 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	ℓnct, 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 ENDD.

ℓnct — Optional absolute evaluatable 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 ℓnct 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 ℓnct 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.

p_i — 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 separator between p_i and (list$_i$) 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 ↦

The substitutable parameter name can occur in any field within a definition.

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_8).

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.)

($list_i$) Actual parameter list in the form $a_1, a_2, \ldots, a_n$ where a_i is substituted for p_i 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 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.

Examples:

In the following examples, the statements that result from expansion are shown faded. 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                    LOCATION  OPERATION  VARIABLE  COMMENTS
                                               1         11         18        30
                          000005                         DUP        5,1
                                                         DATA       1
0153  0000000000000000001                      DATA       1         *DUP*     1
0154  0000000000000000001                      DATA       1         *DUP*     1
0155  0000000000000000001                      DATA       1         *DUP*     1
0156  0000000000000000001                      DATA       1         *DUP*     1
0157  0000000000000000001                      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,,/ABCDEFGHIJK/
          NO    SET        1
                DUP        -1                    UNOBTAINABLE ITERATION COUNT
                GO
                ENDD
                GO                                                          *DUP*
          TAG   MICRO      NO,1,/#ALPHABET#/                                GO
          TAG   MICRO      NO,1,/ABCDEFGHIJK/                               GO
                IFC        EQ,/#TAG#/E/,1        ASSEMBLE STOPDUP WHEN TAG=E GO
                IFC        EQ,/A/E/,1        ASSEMBLE STOPDUP WHEN TAG=E    GO
                STOPDUP                                                    GO
          NO    SET        NO+1                  NO IS 6 IN LAST ITERATION GO
                ENDM                                                       GO
                ENDD                                                       *DUP*
                GO                                                         *DUP*
          TAG   MICRO      NO,1,/#ALPHABET#/                                GO
          TAG   MICRO      NO,1,/ABCDEFGHIJK/                               GO
                IFC        EQ,/#TAG#/E/,1        ASSEMBLE STOPDUP WHEN TAG=E GO
                IFC        EQ,/B/E/,1        ASSEMBLE STOPDUP WHEN TAG=E    GO
                STOPDUP                                                    GO
          NO    SET        NO+1                  NO IS 6 IN LAST ITERATION GO
                ENDM                                                       GO
                ENDD                                                       *DUP*
                .          .
                .          .
                .          .
                .          .
                GO                                                         *DUP*
          TAG   MICRO      NO,1,/#ALPHABET#/                                GO
          TAG   MICRO      NO,1,/ABCDEFGHIJK/                               GO
                IFC        EQ,/#TAG#/E/,1        ASSEMBLE STOPDUP WHEN TAG=E GO
                IFC        EQ,/E/E/,1        ASSEMBLE STOPDUP WHEN TAG=E    GO
                STOPDUP                                                    GO
          NO    SET        NO+1                  NO IS 6 IN LAST ITERATION GO
                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
                          1         11         18                30
                                    PPU
                                    .
                                    .
                                    .
                          STM       PPOP       5,5415B
                                    LIST       M,D,E
                                    ECHO       ,CM=(X,Y,Z)
                                    ECHO       2,P1=(A,B,C)
                                    LDN        CM
                                    STM        P1
                                    ENDD
                                    ECHO       2,P1=(A,B,C)    *ECHO*    1
                                    LDN        X               *ECHO*    1
                                    STM        P1              *ECHO*    1
1452      1450                      LDN        X               *ECHO*    2
1453      5415 0036                 STM        A               *ECHO*    2
1455      1450                      LDN        X               *ECHO*    2
1456      5415 0037                 STM        B               *ECHO*    2
1460      1450                      LDN        X               *ECHO*    2
1461      5415 0040                 STM        C               *ECHO*    2
                                    ENDD                       *ECHO*    1
                                    ECHO       2,P1=(A,B,C)    *ECHO*    1
                                    LDN        Y               *ECHO*    1
                                    STM        P1              *ECHO*    1
1463      1460                      LDN        Y               *ECHO*    2
1464      5415 0036                 STM        A               *ECHO*    2
1466      1460                      LDN        Y               *ECHO*    2
1467      5415 0037                 STM        B               *ECHO*    2
1471      1460                      LDN        Y               *ECHO*    2
1472      5415 0040                 STM        C               *ECHO*    2
                                    ENDD                       *ECHO*    1
                                    ECHO       2,P1=(A,B,C)    *ECHO*    1
                                    LDN        Z               *ECHO*    1
                                    STM        P1              *ECHO*    1
1474      1470                      LDN        Z               *ECHO*    2
1475      5415 0036                 STM        A               *ECHO*    2
1477      1470                      LDN        Z               *ECHO*    2
1500      5415 0037                 STM        B               *ECHO*    2
1502      1470                      LDN        Z               *ECHO*    2
1503      5415 0040                 STM        C               *ECHO*    2
                                    ENDD                       *ECHO*    1
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 listed in the heading can occur in any field within the body. A reference to a substitutable parameter 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 ⌐→ 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).

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 unnamed ENDM terminates all unterminated definitions. An ENDM outside the range of any macro sequence has no effect other than to be included in statement counts.

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 or ENDM. 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 flag and ignore the definition.

The assembler ignores a blank parameter produced by two concurrent 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
                  MACRO       DEF*LOC*ONE*TWO*TEN
      MESSAGE     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	$p_1, p_2, \ldots, p_n$

sym — Optional; depends on definition (see discussion following)

p_i — 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, and parentheses are not given the special meaning used in the variable field of a macro call line.

Example:

1. An illustration of concatenation

```
Location  Code
          Generated   |  LOCATION  |  OPERATION  |  VARIABLE  |  COMMENTS
                      |  1         |  11         |  18        |  30
                      |  MACK      |  MACRO      |  P1,P2     |
                      |            |  S.P1       |  P1+1R.P2  |
                      |            |  .          |            |
                      |            |  .          |            |
                      |            |  .          |            |
                      |            |  ENDM       |            |
                      |            |  .          |            |
                      |            |  .          |            |
                      |            |  .          |            |
                      |            |  MACK       |  A2,A      |
                      |            |  S.A2       |  A2+1R.A   |        MACK    1
7763  5822000001      |            |  SA2        |  A2+1RA    |        MACK    1
                      |            |  ENDM       |            |        MACK    1
```

2. An illustration of nested definitions and calls

```
 |  LOCATION  |  OPERATION  |  VARIABLE  |  COMMENTS
1|            |  11         |  18        |  30
 |  NAME1     |  MACRO      |            |
 |    .       |    .        |            |
 |    .       |    .        |            |
 |    .       |    .        |            |
 |  NAME2     |  MACRO      |            |
 |            |    .        |            |
 |            |    .        |            |
 |            |    .        |            |
 |  NAME2     |  ENDM       |            |
 |            |    .        |            |
 |            |    .        |            |
 |            |    .        |            |  AT THIS TIME, THIS LINE
 |            |  NAME2      |            |  IS PART OF A DEFINITION
 |            |    .        |            |  RATHER THAN BEING A CALL.
 |            |    .        |            |
 |            |    .        |            |
 |            |    .        |            |
 |  NAME1     |  ENDM       |            |
 |            |    .        |            |
 |            |    .        |            |
 |            |    .        |            |
 |            |  NAME1      |            |  NAME1 IS CALLED AND EXPANDED.
 |            |    .        |            |
 |            |    .        |            |
 |            |    .        |            |
 |            |  NAME2      |            |  CALL TO NAME2 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
                                     |            |  MACRO      |  TABLE,TABNAM,VALUE1,VALUE2
                                     |  TABNAM    |  VFD        |  60/VALUE1,60/VALUE2
                                     |            |  ENDM       |
                                     |            |    .        |
                                     |            |    .        |
                                     |            |    .        |
                                     |  SPVAL     |  TABLE      |  1.0,2.0             |  CALL ONE
4741  1720400000000000000  |  SPVAL  |  VFD        |  60/1.0,60/2.0       |          TABLE
4742  1721400000000000000  |         |             |                      |
                           |         |  ENDM       |                      |          TABLE
                           |         |    .        |                      |
                           |         |    .        |                      |
                           |         |    .        |                      |
4743                       |         |  TABLE      |  1.0                 |  CALL TWO
4743  1720400000000000000  |    VFD  |  60/1.0,60/ |                      |          TABLE
4744  0000000000000000000  |         |             |                      |
                           |         |  ENDM       |                      |          TABLE
```

4. An illustration of embedded parameters:

Definition:

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
      XAM       MACRO      A,B
                LDM        A
                LJM        B
                ENDM
```

Call:

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
                  XAM        (SUM,10B),(SAM,IND3)
```

Expansion:

```
Location    Code Generated      LOCATION  OPERATION  VARIABLE  COMMENTS
                                1                    11        18        30
[illegible] 5010 7244           [illegible]  LDM      SUM,10B
[illegible] 0117 7243                        LJM      SAM,IND3
                                             ENDM
```

5. The following example illustrates use of R= in macros:

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
      ONSW      MACRO      N
                R=         X1,N
                SX2        11B
                RJ         =XCPM=
                ENDM
      OFFSW     MACRO      N
                R=         X1,N
                SX2        12B
                RJ         =XCPM=
                ENDM
```

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        MACRO      Z,VAL,P5
          Z          SET        VAL
                     SA7        Z.ALPHA
                     .          .
                     .          .
                     .          .
                     ENDM
                     ABC        IOTA,1,3
          IOTA         SET       1
                     SA7        IOTA.ALPHA    ILLEGAL SYMBOL, TOO LONG
                     ENDM
```

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      BLOCK,KWAL
                     USE        BLOCK
                     QUAL       KWAL
          TAG1       BSS        10B
          TAG2       VFD        60/-1
                     USE        *
                     QUAL       *
                     ENDM
                     TAB        ONE,ONE
                     USE        ONE                              TAB
                     QUAL       ONE                              TAB
          TAG1       BSS        10B                              TAB
          TAG2       VFD        60/-1                            TAB
                     USE        *                                TAB
                     QUAL       *                                TAB
                     ENDM                                        TAB
                     TAB        TWO,TWO
                     USE        TWO                              TAB
                     QUAL       TWO                              TAB
          TAG1       BSS        10B                              TAB
          TAG2       VFD        60/-1                            TAB
                     USE        *                                TAB
                     QUAL       *                                TAB
                     ENDM
```

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
                                1           11        18              30
                                  CHAR      MACRO     ASCII,INTERNAL,EXTERNAL,BCD
                                            CON       ;D:C;B;A
                                            ENDM
                                            .
                                            .
                                            .
                            D+0             BASE      0
                                            CHAR      43,10,10,30 8
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      45,60,20,13 +
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 00000000000012545447                   CON       12545447            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, or ENDM. 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. The total number of unique parameter names and local symbols must not exceed 63 for any one macro definition. Also, any of these can be used to separate the mname from parameters in format two.

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 warning flag and ignore the definition.

The assembler ignores a blank parameter produced by two concurrent 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	$p_1=a_1,p_2=a_2,\ldots,p_n=a_n$

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.

CAUTION

After substitution, spacing between fields is the same as it was before substitution.

Example:

Location	Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
		1		11	18	30
			SAM	MACROE	A,B,C	
				CON	A	
				CON	B	
				CON	C	
				ENDM		
				•		
				•		
				•		
5007				SAM	A=1,C=5,B=0	
5007	000000000000000001			CON	1	SAM 1
5010	000000000000000000			CON	0	SAM 1
5011	000000000000000005			CON	5	SAM 1
				ENDM		SAM 1

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 CPSYN. 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 letters. The first can be any letter. The second letter 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 letter can be any other letter, in which case the operation field of the opdef call is recognized simply by a two-letter 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). Parameters can be separated by any of the characters:

+ - * / ( ) $ = , or .

A blank terminates the list.

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 and the syntax entries that would describe them:

Calling Instruction		Opdef
Operation	Variable Subfields	Syntax
JP†	K††	JPQ
JP†	Bn±K	JPBQ
JP	Bn±Bn±K	JPB+BQ
JP	Bn, K	JPB, Q
JP	Xn/Xn±K	JPX/XQ
NE†	Bn, Bn, K	NEB, B, Q
LJ	Bn-Bn, An-Xn, K	LJB-B, A-X, Q
BXn†	-Xn*Xn	BX-X*X
SBn†	Xn+Bn	SBX+B
LXn†	Bn, Xn	LXB, X
JP†	Bj+K	JPBQ
NE†	Bj, Bk, K	NEB, B, Q
BXi†	-Xk*Xj	BX-X*X
SBi†	Xj+Bk	SBX+B
SBi†	Bj+Xk	SBB+X

† Legal COMPASS CPU instructions
†† 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      P1
                 EQ         P1
                 ENDM
```

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

10002  0230007755 +
```

```
      LOCATION   OPERATION  VARIABLE     COMMENTS
1                11         18           30
                 JP         A3+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
      DXX-X      OPDEF      I,J,K
                 .
                 .
                 .
                 RJ         CKOUT
                 ENDM
```

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      P1,P2
                 AX.P1      P2
                 ENDM
```

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.

OPDEF Syntax	Call		
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.

Location	Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
				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	
10161	27404		PX.4	X.4	IX3
	27606		PX.DIV	X.DIV	IX3
	24444		NX.4	X.4,B4	IX3
	24646		NX.DIV	X.DIV,B4	IX3
10162	44346		FX.3	X.4/X.DIV	IX3
	26343		UX.3	X.3,B4	IX3
	22343		LX.3	X.3,B4	IX3
			ENDM		IX3

The following OPDEF selectively traps the SXi Xj+Bk instructions.

Definition:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	SXX+B	OPDEF	I,J,K	
		•		
		•		
		•		
		ENDM		

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 B 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 maximum number of local symbols and substitutable parameters is 63. COMPASS ignores the use of a substitutable parameter name in the local symbol list.

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 difinition in a manner similar to the way substitutable parameters are replaced.

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
           LOCAL      C
 C         BSS        10B
   .         .
   .         .                                    DEFINITION
   .         .                                    OF ABC
 XYZ       MACRO      D
           SA1        C                 DEFINITION
             .          .               OF XYZ
             .          .
           ENDM
           ABC        3,4
 ↑↑000001            BSS    10B
 XYZ       MACRO      D                           EXPANSION
           SA1        ↑↑000001          DEFINITION OF ABC      ABC
                                        OF XYZ                 ABC
           ENDM                                                ABC
                                                               ABC
```

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          |11         |18        |30
|  BCD      | MACRO     | A,B      |                    }
|           | LOCAL     | C        |                    |
|  C        | BSS       | 10B      |                    |
|  .        | .         |          |                    |
|  .        | .         |          |                    | DEFINITION
|  .        | .         |          |                    | OF BCD
|  YZA      | MACRO     |          |                    |
|           | LOCAL     | C        |                    |
|           | SA1       | C        | DEFINITION         |
|           | .         |          | OF YZA             |
|           | .         |          |                    |
|  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          |11         |18        |30
|           | BCD       | 5,6      | EXPANSION OF BCD
10175       |           | BSS  10B |                    BCD
|  ↑↑000002 | YZA       |          |                    BCD
|           | MACRO     |          |                    BCD
|           | LOCAL     | ↑↑000002 |                    BCD
|           | SA1       | ↑↑000002 |                    BCD
|  ↑↑000002 |           | BSSZ  1  |                    BCD
|           | ENDM      |          |                    BCD
```

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          |11         |18        |30
10205       |           |          |
            | YZA       |          | EXPANSION OF YZA
10205  5110010206 ↑       SA1       ↑↑000003                 YZA
10206       |           | BSSZ  1  |                    YZA
|  ↑↑000003 | ENDM      |          |                    YZA
```

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.

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 |  } DEFINITION
                              SA6        ARG     } SEQUENCE |  } OF ZAB
                              IRP                          |
                              ENDM                         }
                              .
                              .
10207                         ZAB        (J,K,L),CON
10207  5110010127 +           IRP        J,K,L
                7261010133 +  SA1        J
10210  5160010127 +           SX6        X1+CON
                5110010131 +  SA6        J
10211  7261010133 +           SA1        K
                5160010131 +  SX6        X1+CON
10212  5110010132 +           SA6        K
                7261010133 +  SA1        L
10213  5160010132 +           SX6        X1+CON
                              SA6        L
                              IRP
                              ENDM
```

2. Assign symbol at every 100_8 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
                P         BSSZ       100B                BUF
                Q         BSSZ       100B                BUF
                R         BSSZ       100B                BUF
                S         BSSZ       100B                BUF
                T         BSSZ       100B                BUF
                          IRP                            BUF
                          ENDM                           BUF
```

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 systems 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.

OPERATION CODE TABLE MANAGEMENT 6

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)

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 mnemonic operation
CPSYN	Synonymous CPU operation

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 instruction that logically removes 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, and PPU symbolic machine instructions
3. Programmer-created entries for syntactically identified instructions
4. CPU symbolic instructions

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
1                 11         18        30
      SB4       MACRO      P1,P2       DEFINE MACRO NAMED SB4
                .
                .
                .
                ENDM
                .
                .
                .
                SB4        A1+ABLE     CALL TO MACRO.  NOT CPU INSTRUCTION
                .
                .
                .
                SB3        A1+ABLE     MACHINE INSTRUCTION

      SB4       OPSYN      NIL         DISABLES MACRO BUT DOES NOT
                .                      RESTORE NORMAL USE OF SB4
                .                      AS AN OPERATION CODE.  EVEN IF
                .                      IT WERE REDEFINED WITH OPDEF
                .                      IT WOULD NOT BE RECOGNIZED.
                .                      THE MACRO FORM ALWAYS TAKES
                .                      PRECEDENCE.
                .
                PURGMAC    SB4         RESTORES NORMAL USE OF SB4
```

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

name — Mnemonic name, 1-8 characters

ctl — Control of instruction assembly

ctl	Significance
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 (for example, UJN)
3	24-bit instruction with 18-bit address (for example, LDC)
4	12-bit instruction with 6-bit address (for example, LDN)
5	24-bit instruction with 12-bit address and optional indexing (for example, LDM)
6	12-bit instruction with signed relative address (for example, SHN)
7	24-bit instruction with 12-bit address and required second (for example, IAM)

val — An evaluatable expression specifying the 4-octal 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.

Example:

```
                Code Generated                 LOCATION  OPERATION  VARIABLE      COMMENTS
                                               1         11         18            30
                                                         PERIPH
                                      0+0                BASE       0
                                                         .
                                                         .
                                                         .
                                      15       LA        EQU        15
                                      40       C         EQU        40
                                               STM       PPOP       5,5400+LA
                                                         .
                                                         .
                                                         .
7311                5415 0040                            STM        C
```

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
$name_1$	OPSYN	$name_2$

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
1|           |11         |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      |          |
 |           | .         |          |
 |           | .         |          | CODE USING LJM MACRO
 |           | .         |          |
 |  LJM      | OPSYN     | LJM.     | 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	$name_1, name_2, \ldots, name_n$

$name_i$ 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

sytx

The syntax consists of a mnemonic operator and variable field descriptors. The mnemonic operator consists of two letters. The first can be any letter. The second letter 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 letter can be any other letter, in which case the operation field of the instruction is recognized simply by a two-letter 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.

ctl	Significance
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 evaluatable 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 of two registers in variable field
0	Set field to 0

Example:

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
		11	18	30
	SAX+B	CPOP	0,5308,1328	DEFINES SAI XJ+BK
	SXXQ	CPOP	1,7208,1208	DEFINES SXI XJ+K
		•		
		•		
		•		
53731		SA7	X3+B1	
722 7231000003	TAG	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
$sytx_1$	CPSYN	$sytx_2$

$sytx_1$ Syntax of a CPU instruction (see CPOP for legal forms). If this syntax is already in the operation code table, the table entry for $sytx_2$ takes precedence over the old table entry for $sytx_1$ without notification.

$sytx_2$ Syntax of a CPU instruction for which there must be an entry in the operation code table. Following the CPSYN, an instruction in either $sytx_1$ or $sytx_2$ produces an octal instruction of the format described by the entry for $sytx_2$.

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.

MICROS 7

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. Two instructions related to micros are discussed elsewhere. They are BASE (section 4.4.1) which allows optional micro definition of the base, and MICCNT (section 4.6.5) which allows a micro size to be defined as a symbol.

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 (≠) 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 card read is exceeded as a result of micro substitution, the assembler creates up to a maximum of 9 continuation cards, 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 non-fatal 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	≠NAM≠+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

COMPASS provides four pseudo instructions that define micros, MICRO, OCTMIC, DECMIC, and BASE (section 4.4.1).

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 evaluatable expression specifying starting character in string; when the base is M, COMPASS assumes that n is decimal.

n_2 — Absolute evaluatable 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.

Examples:

1. 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,*MAJOR           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≠≠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      1,19,*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,,* LOC              SA1     ADDRESS+*
                  .
                  .
                  .
     ≠NAM≠4
      LOC        SA1        ADDRESS+4
```

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 expression to be converted

n — Optional absolute 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:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	V	DECMIC	8,6	
	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 expression to be converted

n — Optional absolute 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:

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
	V1	OCTMIC	B,6	
	S1	MICRO	1,,*≠V1≠ ADDITIONAL STORAGE NEEDED*	
	S1	MICRO	1,,*000008 ADDITIONAL STORAGE NEEDED*	

7.3 PREDEFINED MICRO NAMES

Two standard micros (DATE and TIME) are predefined by the COMPASS assembler. They are available for every assembly. The programmer simply writes the micro reference as desired.

7.3.1 DATE

The DATE micro contains the current date in 10 characters in the following form as obtained from the operating system:

Δyr/mo/dy.

The micro reference is ≠DATE≠.

7.3.2 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≠	

CPU SYMBOLIC MACHINE INSTRUCTIONS 8

6000/7000 COMPASS recognizes symbolic notation for all 7600 Central Processor Unit instructions and all 6000-Series Computer Systems Central Processor Unit instructions.

The assembler identifies each symbolic instruction according to its syntax and generates a one parcel 15-bit instruction or a two parcel 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.

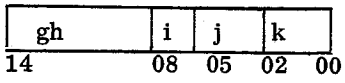


Figure 8-1. CPU 15-Bit Instruction Format

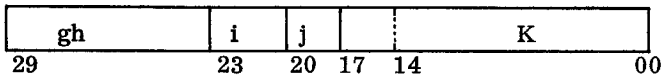


Figure 8-2. CPU 30-Bit Instruction Format

gh	6-bit instruction code
ghi	9-bit instruction code
i	3-bit code specifying one of eight designated registers (e.g., Ai)
j	3-bit code specifying one of eight designated registers (e.g., Bj)
k	3-bit code specifying one of eight designated registers (e.g., Bk)
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 parcel instructions in a 60-bit CPU instruction word. Generally, the assembler does not allow a two-parcel instruction to begin in the fourth parcel of a word. However, the assembler may generate a 30-bit instruction in a fourth parcel when all of the following are true:

1. The assember is at the fourth parcel (position counter is 15)

2. The instruction does not include K. Note that if K is included in the syntax and reduces to zero, it requires 30 bits because the evaluation of K takes place in the second pass whereas the space for the instruction is reserved in the first pass.

3. The instruction does not have a location field symbol or is not otherwise forced upper.

When a two parcel instruction begins in the last parcel of a word, the 7600 executes it as if there were a fifth parcel in the instruction word and that parcel contained all zeros. On the 6400, this condition causes an error exit. On the 6600, 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+46000B instruction
Lower 45 bits remain	They are packed with a NO instruction and an SB0 B0+46000B instruction

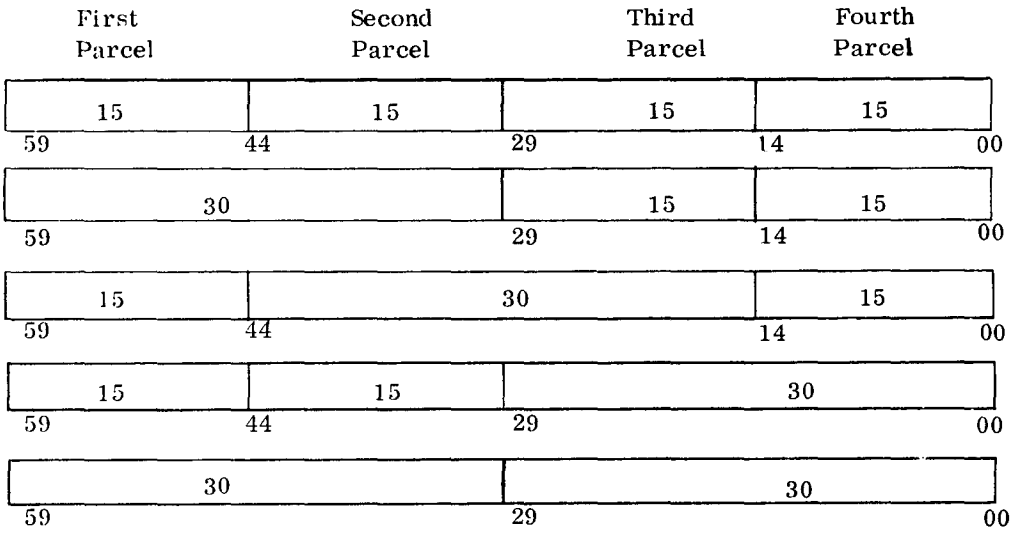


Figure 8-3. Arrangements of Instructions in a 60-bit CPU Word

8.2 INSTRUCTION EXECUTION

Execution times for all instructions are listed in Appendix A.

8.2.1 6600/6700 EXECUTION

After an exchange jump start by a PPU 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 26 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.

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 no further instructions are issued until instruction has been executed. In the execution of a branch instruction, the branch unit uses:

  Increment unit to form the instructions that branch to K + Bi and branch to K if Bi ...

  Long add unit to perform the instructions that branch to K if Xj ...

  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.

† The 6700 also includes a 6400-type central processor unit

TABLE 8-1. 6600/6700 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.

8.2.2 6400/6500 EXECUTION

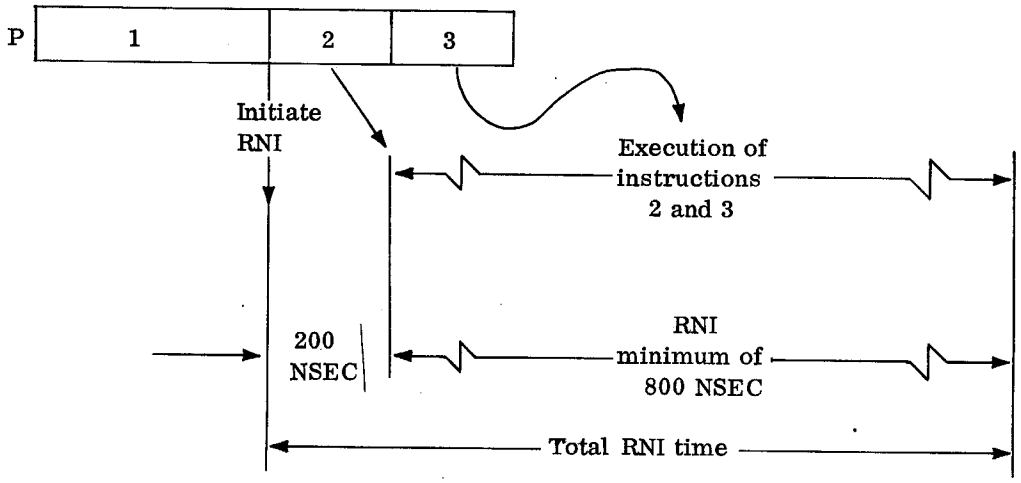
The 6400 and 6500 systems CPU has a unified arithmetic unit, rather than separate functional units as in the 6600 system. Instructions in the 6400 and 6500 CPU are executed sequentially.

For efficient coding in the 6400 and the 6500 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 (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 (Appendix A).

Example:

P	Jump to K (met)		Pass	Pass
K	Add 1	Add 2	Load	Load

Instruction	Minor Cycles Required
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 regardless of the execution times stated in Appendix A.

Example:

P	Jump to K (not met)	Pass	Pass
P + 1			

8.2.3 7600 EXECUTION

Execution of an arithmetic or logical machine instruction takes place in one of nine functional units in the computation section of 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 may be in operation at the same time.

TABLE 8-2. 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 7600 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 7600 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 7600 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 (7600 only). Data also flows from SCM (CM) and LCM (7600 only) 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 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 according to octal sequence. Instructions unique to a computer system are identified as such. These instructions can be assembled on any machine but will execute properly on the noted machine only. For details and special conditions arising during instruction execution, refer to the relevant hardware system reference manual.

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 instruction, subfields take the forms:

r -r r,r	r is a register designator
r op r -r op r	op is a register operator + - * /
±jk	jk is an absolute expression specifying a shift count or mask bit count. If the expression value is in the range -60 to -0, inclusive, COMPASS adds 60 to it. If it is less than -60 or greater than 63, COMPASS sets a warning flag and uses only the low-order 6 bits of the expression value.

For a 30-bit instruction, subfields take the forms:

K	The single subfield contains an absolute, relocatable, or external expression that does not include a register.
r op K	The single subfield contains an absolute, relocatable, or external expression that includes a register designator; op is an expression operator: + - * /
r,K	One subfield contains a register designator, the other subfield contains an absolute, relocatable, or external expression that does not include a register designator.
r,r,K	Two subfields contain register designators; a third contains an absolute, relocatable, or external expression that does not include a register.

In the formats and examples, K reduces to an 18-bit value that represents one of the following in pass two:

An absolute address or a word count

An external symbol ± an integer value

An address that is relocatable relative to the program origin or common block origin

An address of a literal

If K is negative, the assembler inserts the one's complement of the integer value in the K portion of the instruction.

In the descriptions of the formats, ±K designates that the evaluation of all nonregister elements can result in a positive or negative value for the expression (see section 2.8.2 Evaluation of Expressions). Use of ±K to represent the integer portion of the expression does not imply that the first term operator in the expression is an expression operator. If you consider that a and b are terms in expression K, then +K indicates that the sum of the values of a and b is positive and -K indicates that the sum of the values is negative. Thus, -K does not mean that a-b would become -a+b.

In the following example, the symbol XRAY has the value 407_8. The first term operator (-) forms the value 777370_8. Subtracting 1 from this results in 777367_8 or a -K (-410_8).

Code Generated

```
13  7212777367
```

```
|  | LOCATION | OPERATION | VARIABLE | COMMENTS |
|  1          |  11       |  18      |  30      |
|             |  SX1      |  X2-XRAY-1          |
```

Unless otherwise noted, subfields can be in any order. COMPASS also allows an added degree of flexibility by allowing the variable subfields of an instruction to be written in the operation field with each subfield preceded by a comma. For example:

Code Generated

```
|  | LOCATION | OPERATION | VARIABLE | COMMENTS |
|  1          |  11       |  18      |  30      |
|             |  UX1      |  B2,X3              |
```

can be written

Code Generated

```
|  | LOCATION | OPERATION | VARIABLE | COMMENTS |
|  1          |  11       |  18      |  30      |
|             |  UX1,B2   |  X3                 |
```

```
26123
```

The instructions are identical to the assembler.

Similarly, the following instructions are regarded as identical. Use of this feature is optional.

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
0423004507		EQ	B2,B3,K	
0423004507		EQ,B2	B3,K	
0423004507		EQ,B2,B3 K		
0423004507		EQ,B2,B3,K		

8.4.1 PROGRAM STOP INSTRUCTION (6000-SERIES ONLY)

This instruction stops the central processor unit at the current step in the program. An exchange jump is necessary to restart the central processor unit. 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.

Formats: 6600 Functional Unit: Branch

Operation	Variable	Description	Size	Octal Code
PS		Program stop	30 bits	00000 00000
PS	K	Program stop	30 bits	0000K

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
0000000000		PS		

8.4.2 ERROR EXIT INSTRUCTION (7600 ONLY)

This instruction is exclusive to the 7600. Its 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.

Format:

Functional Unit: None

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
```

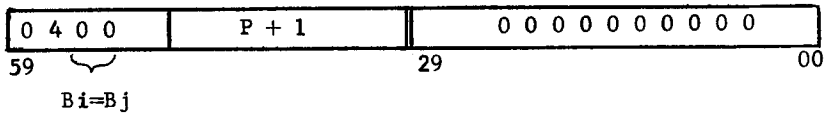
```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
      ES
```

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:

Address K



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.

6600 Functional Unit: Branch
7600 Functional Unit: None

Format:

Operation	Variable	Description	Size	Octal Code
RJ	K	Return jump to K	30 bits	0100K

Example:

Code Generated

```
0100005250 +
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RJ	HELP	

8.4.4 ECS INSTRUCTIONS (6000-SERIES ONLY)

These instructions initiate either a read or write operation to transfer (Bj) + K 60-bit words between extended core storage (ECS) and central memory (CM). The initial ECS address is (X0) + RA_{ECS}; the initial CM address is (A0) + RA_{CM}.

The assembler forces upper before assembling an RE or WE instruction.

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:

1. Parity error(s) when reading ECS. If a parity error is detected, the entire block of data is transferred before the exit is taken.
2. 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.
3. 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.

For additional information about these instructions, refer to the CONTROL DATA® 6400/6500/6600 Computer Systems Extended Core Storage Reference Manual, Publication No. 60225100.

Formats: Functional Unit: None

Operation	Variable	Description	Size	Octal Code
RE	Bj	Read extended core storage	30 bits	011j0 00000
RE	K	Read extended core storage	30 bits	0110K
RE	Bj+K	Read extended core storage	30 bits	011jK
WE	Bj	Write extended core storage	30 bits	012j0 00000
WE	K	Write extended core storage	30 bits	0120K
WE	Bj+K	Write extended core storage	30 bits	012jK

Examples:

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 (7600 ONLY)

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 1777_8. (For example, if the result of the add is 003000_8, the instruction transfers 1000_8 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 $(X0_{18-00})$ 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.

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.

Formats:

Functional Unit: None

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
1                11          18            30
           RL    1000B+B5
           RL    2000B
           WL    B4-100B
```

8.4.6 EXCHANGE JUMP INSTRUCTION (6000-SERIES ONLY)

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.

Operation is as follows:

1. 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.

2. 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 Standard Option 10104-A/B/C/D Central and Monitor Exchange Jumps for 6600 Reference Manual, Pub. No. 60203200.

The assembler forces upper before and after assembling an XJ instruction.

Formats: Functional Unit: Branch

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

Examples:

Code Generated
0130000000
0130001000
0135000600

```
      LOCATION  OPERATION  VARIABLE      COMMENTS
1                 11         18          30
                  XJ
                  XJ         1000B
                  XJ         B5+600B
```

8.4.7 EXCHANGE EXIT INSTRUCTION (7600 ONLY)

The normal termination for an exchange package execution interval is through execution of an exchange instruction (MJ). The exit mode flag in the PSD register determines the source of the exchange package.

This instruction 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 MJ instruction voids the instruction word stack. Any instructions remaining in the stack are not executed.

The system makes no protective tests on the exchange jump address.

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) $\pm$ 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.

This instruction is used for calling a system monitor program for input/output, monitor calls, etc.

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.

Formats:

Functional Unit: None

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

Examples:

Code Generated

```
01300
0134000500
0136777477
0130000600
```

```
      LOCATION  OPERATION  VARIABLE      COMMENTS
1                 11         18          30
          MJ
          MJ      B4+500B
          MJ      -300B+B6
          MJ      600B
```

8.4.8 DIRECT LCM TRANSFER INSTRUCTIONS(7600 ONLY)

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 a 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 perods 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 (Xk_{18-00}). 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.

Formats:

Functional Unit: None

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

Examples:

Code Generated

```
01465
01570
```

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
      RX6       X5
      WX7       X0
```

8.4.9 RESET INPUT CHANNEL BUFFER INSTRUCTION (7600 ONLY)

This instruction is exclusively a 7600 instruction. It 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: Functional Unit: None

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
                  PI         B7
```

8.4.10 SET REAL-TIME CLOCK INSTRUCTIONS (7600 ONLY)

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^{17} 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: Functional Unit: None

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
```

8.4.11 RESET OUTPUT CHANNEL BUFFER INSTRUCTION (7600)

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:

Functional Unit: None

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	

8.4.12 READ CHANNEL STATUS INSTRUCTIONS (7600 ONLY)

These instructions copy the contents of the input or output channel buffer address register indicated by masking (Bk_{03-00}) 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.

Formats:

Functional Unit: None

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     11        11         18        30
      IB6       B4
      OB5       B6
```

8.4.13 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 a 6000-Series system this instruction voids the stack. On a 7600, 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 0. A force upper occurs after the instruction is assembled.

6600 Functional Unit: Branch
7600 Functional Unit: None

Format:

Operation	Variable	Description	Size	Octal Code
JP	Bi+K	Jump to (Bi)+K	30 bits	02i0K
JP	Bi	Jump to (Bi)	30 bits	02i00 00000
JP	K	Jump to K	30 bits	0200K

Example:

Code Generated

```
0250005247 +
0270000000
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		JP	B5+GOTO	
		JP	B7	

8.4.14 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:

1. The ZR and NZ operations test the full 60-bit word in Xj. The words 00.....00 and 77.....77 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 to test for underflow of floating point quantities.

2. The PL and NG operations examine only the sign bit (2^{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.

3. The IR and OR operations examine the upper-order 12 bits of Xj.

   On the 7600, the following 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 a 6000-Series computer system, 3777x...x and 4000x...x are out of range; all other words are in range.

4. 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.

Formats:

6600 Functional Unit: Branch
7600 Functional Unit: None

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) $\neq$ 0	30 bits	031jK
PL	Xj,K	Branch to K if (Xj) positive	30 bits	032jK
NG	Xj,K	Branch to K if (Xj) negative	30 bits	033jK
MI	Xj,K	Branch to K if (Xj) negative	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

Examples:

Code Generated

```
0305002363 +
0313002364 +
0324002365 +
0331002366 +
0331002366 +
0340002367 +
0351002370 +
0365002371 +
0377002372 +
```

```
      LOCATION  OPERATION  VARIABLE      COMMENTS
1                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,INUEFNT
```

8.4.15 B-REGISTER CONDITIONAL BRANCH INSTRUCTIONS

These instructions test an 18-bit word from register Bi against an 18-bit word from register Bj for the condition specified. They branch to address K on a successful test. Otherwise, the program sequence continues at the next instruction. The decision is not made until both B registers are free. For the tests against zero (all zeros), the assembler sets either the i or the j designator to 0 indicating B0.

The following rules apply in the tests made by these instructions:

1. Positive zero is recognized as unequal to negative zero, and
2. Positive zero is recognized as greater than negative zero, and
3. 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 stack.

6600 Functional Unit: Branch
7600 Functional Unit: None

Formats:

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) $\neq$ 0	30 bits	05i0K
NE	Bi, Bj, K	Branch to K if (Bi) $\neq$ (Bj)	30 bits	05ijK
NZ	Bi, K	Branch to K if (Bi) $\neq$ 0	30 bits	05i0K
PL	Bi, K	Branch to K if (Bi) $\geq$ 0	30 bits	06i0K
GE	Bi, K	Branch to K if (Bi) $\geq$ 0	30 bits	06i0K
GE	Bi, Bj, K	Branch to K if (Bi) $\geq$ (Bj)	30 bits	06ijK
LE	Bj, Bi, K	Branch to K if (Bj) $\leq$ (Bi)	30 bits	06ijK
LE	Bj, K	Branch to K if (Bj) $\leq$ 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

† The assembler forces the position counter upper after assembling the instructions.

Formats (cont'd):

Operation	Variable	Description	Size	Octal Code
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

Examples:

Code Generated

```
0450005221 +
0405005222 +
0453005223 +
0400005223 +
0515005224 +
0560005225 +
0620005226 +
0645005227 +
0650005230 +
0676005231 +
0770005232 +
0730005233 +
0767005234 +
0705005235 +
0712005236 +
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		ZR	B5,BZERO	
		EQ	B0,B5,EQUAL	
		EQ	B5,B3,JUMP	
		EQ	JUMP	
		NE	B1,B5,NOTEQ	
		NZ	B6,BNOTZR	
		PL	B2,BPLUS	
		GE	B4,B5,GEQ	
		GE	B5,GEB0	
		LE	B6,B7,LTHAN	
		NG	B7,BNEG	
		MI	B3,B3LT0	
		GT	B7,B6,B7GT	
		GT	B5,B5GT0	
		LT	B1,B2,BLTB	

8.4.16 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:

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

Operation	Variable	Description	Size	Octal Code
BXi	Xj	Transmit (Xj) to Xi	15 bits	10ijj

Example:

Code Generated

```
10622
```

```
      LOCATION  OPERATION VARIABLE          COMMENTS
1     |         |11       |18               |30
      |         |BX6      |X2               |
```

8.4.17 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:

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

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            |
```

8.4.18 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.

6600 Functional Unit: Boolean
7000 Functional Unit: Boolean

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	

8.4.19 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.

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

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	

8.4.20 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.

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

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	

8.4.21 LOGICAL PRODUCT AND COMPLEMENT INSTRUCTION

This instruction forms the logical product (AND function) of the 60-bit quantity from operand register Xj 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.

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

Format:

Operation	Variable	Description	Size	Octal Code
BXi	-Xk*Xj	Logical product of (Xj) and complement of (Xk) to Xi	15 bits	15ijk

Examples:

Code Generated

15432

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		BX4	-X2*X3	

8.4.22 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 bits 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.

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

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	

8.4.23 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.

6600 Functional Unit: Boolean
7600 Functional Unit: Boolean

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
1		11	18	30
		BX7	-X1-X3	

8.4.24 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 on the value in the jk fields. If it is negative, COMPASS subtracts jk from 60 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 7-type error.

6600 Functional Unit: Shift
7600 Functional Unit: Shift

Format:

Operation	Variable	Description	Size	Octal Code
LXi	jk	Logical shift (Xk) by ± jk places	15 bits	20ijk

Example:

Code Generated

20325

20362

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		LX3	25B	
		LX3	-12B	

8.4.25 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 subtracts jk from 60 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 7-type error is generated.

Format:

6600 Functional Unit: Shift
7600 Functional Unit: Shift

Operation	Variable	Description	Size	Octal Code
AXi	jk	Arithmetic shift (Xk) by ± jk places	15 bits	21ijk

Example:

Code Generated

```
21537
```

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
          AX5     37B
```

8.4.26 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.

1. If (Bj) is positive (i.e., bit 17 of Bj = 0), the quantity from Xk is shifted left circular. The low order six bits of (Bj) specify the shift count. The higher order bits are ignored.

2. If (Bj) is negative (i.e., bit 17 of Bj = 1), the quantity from Xk is shifted right (end off with sign extension). The one's complement of the low order 12 bits of (Bj) specify shift count. The higher order bits are ignored. If the shift count is greater than 60 (decimal) the result stored in the Xi register consists of 60 copies of the operand sign bit.

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:

6600 Functional Unit: Shift
7600 Functional Unit: Shift

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

Example:

Code Generated

```
22675
22534
```

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
          LX6     X5,B7
          LX5     B3,X4
```

8.4.27 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.

1. If (Bj) is positive (i.e., bit 17 of Bj = 0), the quantity from register Xk is shifted right (end off with sign extension). The lower order 12 bits of (Bj) specify the shift count. The higher order bits are ignored. If the shift count is greater than 60 (decimal) the 7600 result stored in the Xi register consists of 60 copies of the operand sign bit. The 6000 series result consists of all zeros.
2. If (Bj) is negative (i.e., bit 17 of Bj = 1), the quantity from register Xk is shifted left circular. The complement of the lower order six bits of Bj specify the shift count. The higher order bits are ignored.

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.

6600 Functional Unit: Shift
7600 Functional Unit: Shift

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

Example:

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
23764		AX7	X4,B6	
23211		AX2	B1,X1	

8.4.28 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:

6600 Functional Unit: Shift
7600 Functional Unit: Normalize

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

Example:

Code Generated

```
24575
24505
24552
```

```
      LOCATION   OPERATION   VARIABLE   COMMENTS
1                11          18         30
          NX5     X5,B7
          NX5     X5
          NX5,B5  X2
```

8.4.29 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...x or 4000x...x) or an indefinite quantity (1777x...x or 6000x...x), no shift takes place. The contents of Xk are copied into Xi, and Bj is set to zero.

Formats:

6600 Functional Unit: Shift
7600 Functional Unit: Normalize

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

Example:

Code Generated

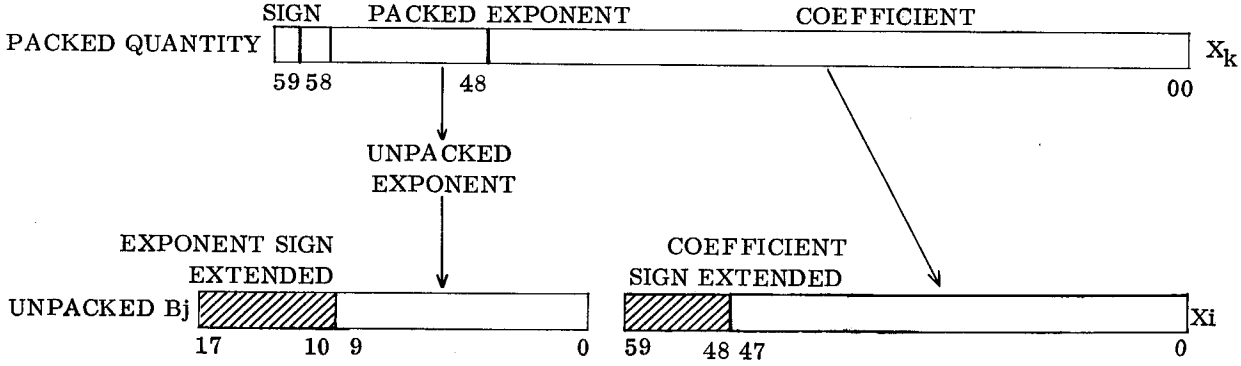
```
25474
25404
25361
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		7X4	X4,B7	
		7X4	X4	
		ZX3,B6	X1	

8.4.30 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.

6600 Functional Unit: Shift
7600 Functional Unit: Boolean

Formats:

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

Example:

Code Generated

```
26777
26342
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		UX7	X7,B7	
		UX3,X2	B4	

8.4.31 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 toggling bit 2^{10} during the pack operation. The 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 58-48 of Xk and bits 17-11 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 Xi_{58-48} is obtained from Bj_{10-00} by complementing bit 10; if (Xk) is negative, bit 10 is not complemented but bits 09-00 are complemented.

The j designator may 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).

6600 Functional Unit: Shift
7600 Functional Unit: Boolean

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)	15 bits	27ijk

Example:

Code Generated

```
27565
27671
```

```
      LOCATION   OPERATION   VARIABLE    COMMENTS
1                11          18          30
          PX5     X5,B6
          PX6,B7  X1
```

8.4.32 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.

Formats:

6600 Functional Unit: Floating Add
7600 Functional Unit: Floating Add

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

Examples:

Code Generated

```
30345
31213
```

```
      LOCATION   OPERATION   VARIABLE        COMMENTS
1     ,          11         18              30
      FX3        X4+X5
      FX2        X1-X3
```

8.4.33 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.

Formats:

6600 Functional Unit: Floating Add
7600 Functional Unit: Floating Add

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

Examples:

Code Generated

```
32323
33414
```

```
      LOCATION   OPERATION   VARIABLE        COMMENTS
1                11         18              30
      DX3        X2+X3
      DX4        X1-X4
```

8.4.34 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.

6600 Functional Unit: Floating Add
7600 Functional Unit: Floating Add

Formats:

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

Examples:

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
34534		RX5	X3+X4	
35653		RX6	X5-X3	

8.4.35 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 0's), the result is a positive zero quantity. If both operands are minus zero (all 1's), the result is a negative zero quantity.

Format:

6600 Functional Unit: Long Add
7600 Functional Unit: Long Add

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:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
36545		IX5	X4+X5	
37631		IX6	X3-X1	

8.4.36 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.

Formats:

6600 Functional Unit: Floating Multiply
7600 Functional Unit: Floating Multiply

Operation	Variable	Description	Size	Octal Code
FXi	Xj*Xk	Floating point product of (Xj) and (Xk) to Xi	15 bits	40ijk

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
40011		FX0	X1*X1	

8.4.37 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:

6600 Functional Unit: Floating Multiply
7600 Functional Unit: Floating Multiply

Operation	Variable	Description	Size	Octal Code
RXi	Xj*Xk	Rounded floating point product of (Xj) and (Xk) to Xi	15 bits	41ijk

Example:

Code Generated

41232

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RX2	X3*X2	

8.4.38 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 this product (bits 47-00) 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:

6600 Functional Unit: Floating Multiply
7600 Functional Unit: Floating Multiply

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

```
42 345
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		DX3	X4 * X5	

8.4.39 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 0's or all 1's in its leftmost 12 bits, the 48-bit integer product is formed in Xi with sign extension in its leftmost 12 bits. (Exception: If each operand has bit 2^{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 the instruction is being used for integer multiplication.

Format:

6600/6700 Functional Unit: Floating Multiply
or 7600 Functional Unit: Floating Multiply

Operation	Variable	Description	Size	Octal Code
IXi	Xj*Xk	Integer product of (Xj) and (Xk) to Xi	15 bits	42ijk

Example:

Code Generated

```
42234
```

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		IX2	X3*X4	

8.4.40 MASK INSTRUCTION

This instruction clears register Xi and forms a mask in it. A positive value for expression jk defines the number of 1's 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 1's in the high order bit positions of the word and 0's 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 1's when jk is 60.

This instruction is intended for generated variable width masks for logical operations. Used with the shift instruction, this instruction creates an arbitrary field mask faster than by reading a pregenerated 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 subtracts the expression value from 60 and places the difference in the jk field of the assembled instruction.

A negative jk value less than -60 results in a 7-type assembly error.

An MXi 0 is the fastest instruction for clearing an X register.

Format:

6600 Functional Unit: Shift
7600 Functional Unit: Shift

Operation	Variable	Description	Size	Octal Code
MXi	jk	Form mask in Xi, $\pm$ jk bits	15 bits	43ijk

Example:

Code Generated

43042

43760

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		MX0	42	
		MX3	-14B	

8.4.41 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:

6600 Functional Unit: Floating Divide
7600 Functional Unit: Floating Divide

Operation	Variable	Description	Size	Octal Code
FXi	Xj/Xk	Floating point divide of (Xj) by (Xk) to Xi	15 bits	44ijk

Example:

Code Generated

44671

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		FX6	X3/X1	

8.4.42 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:

6600 Functional Unit: Floating Divide
7600 Functional Unit: Floating Divide

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	

8.4.43 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 9 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.

Format:

6600 Functional Unit: None
7600 Functional Unit: None

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		

8.4.44 POPULATION 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 59-06 are cleared.

If Xk is a word of all 1's, a count of 60 (decimal) is delivered to the Xi register. If Xk is a word of all 0's, a zero word is delivered to the Xi register.

The assembler sets the unused j designator to k.

Formats:

6000 Functional Unit: Floating Divide
7600 Functional Unit: Population Count

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
```

8.4.45 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.

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.

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)

Formats:

6600 Functional Unit: Increment
7600 Functional Unit: Increment

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

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
5010000001		SA1	A0+1	
5100777774		SA0	-3	
5121000003		SA2	3+B1	
5231777771		SA3	X1-6	
53411		SA4	X1+B1	
54541		SA5	A4+B1	
54641		SA6	A4+B1	
54540		SA5	A4	
55641		SA6	-B1+A4	
56711		SA7	B1+B1	
57721		SA7	B2-B1	

8.4.46 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.

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.

Formats:

6600 Functional Unit: Increment
7600 Functional Unit: Increment

Operation	Variable	Description	Size	Octal Code
SBi	Aj+K	Set Bi to (Aj) ± K	30 bits	60ijK
SBi	K	Set Bi to K	30 bits	61i0K
SBi	Bj+K	Set Bi to (Bj) ± K	30 bits	61ijK
SBi	Xj+K	Set Bi to (Xj) ± K	30 bits	62ijK
SBi	Xj	Set Bi to (Xj)	15 bits	63ij0
SBi	Xj+Bk	Set Bi to (Xj) + (Bk)	15 bits	63ijk
SBi	Bk+Xj	Set Bi to (Xj) + (Bk)	15 bits	63ijk
SBi	Aj	Set Bi to (Aj)	15 bits	64ij0
SBi	Aj+Bk	Set Bi to (Aj) + (Bk)	15 bits	64ijk
SBi	Bk+Aj	Set Bi to (Aj) + (Bk)	15 bits	64ijk
SBi	Aj-Bk	Set Bi to (Aj) - (Bk)	15 bits	65ijk
SBi	-Bk+Aj	Set Bi to (Aj) - (Bk)	15 bits	65ijk
SBi	Bj	Set Bi to (Bj)	15 bits	66ij0
SBi	Bj+Bk	Set Bi to (Bj) + (Bk)	15 bits	66ijk
SBi	-Bk	Set Bi to (B0) - (Bk)	15 bits	67i0k
SBi	Bj-Bk	Set Bi to (Bj) - (Bk)	15 bits	67ijk
SBi	-Bk+Bj	Set Ai to (Bj) - (Bk)	15 bits	67ijk

Examples:

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	

8.4.47 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.

Formats:

6600 Functional Unit: Increment
7600 Functional Unit: Increment

Operation	Variable	Description	Size	Octal Code
SXi	Aj+K	Set Xi to (Aj) ± K	30 bits	70ijK
SXi	K	Set Xi to K	30 bits	71i0K
SXi	Bj+K	Set Xi to (Bj) ± K	30 bits	71ijK
SXi	Xj+K	Set Xi to (Xj) ± 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

Examples:

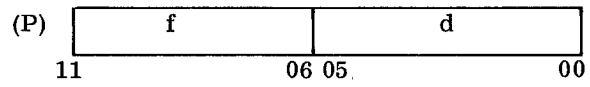
Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
7000005233 +		SX0	BNEG+A0+1	
7110775755		SX1	-2022B	
7121000005		SX2	B1+5	
7233777744		SX3	X3-33B	
73442		SX4	X4+B2	
74553		SX5	A5+B3	
74540		SX5	A4	
75641		SX6	-B1+A4	
75604		SX6	A0-B4	
76776		SX7	B7+B6	
77751		SX7	B5-B1	

PPU SYMBOLIC MACHINE INSTRUCTIONS 9

The 6000/7000 COMPASS assembler recognizes symbolic notation for peripheral processor unit instructions. 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 PPU code is absolute. Numeric data must be in integer notation. Floating point notation is illegal.

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 PPU 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.



Direct Mode:

d = memory address of operand

Indirect Mode:

d = memory address of the address of the operand

No Address Mode:

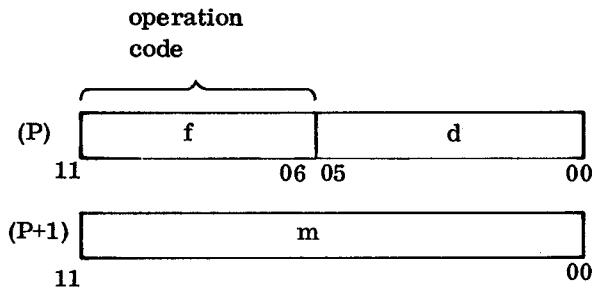
d = 6-bit operand, shift count, or relative address

Other:

d = special value; e.g., channel designator

Figure 9-1. PPU 12-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.



Indexed Mode:

d = address of the index for modifying the address of the operand

m = base address of the operand

(d) + m = address of operand

Constant Mode:

dm = 18-bit operand

Other:

dm = special values; e.g., d = channel designator and m = 12-bit address of word count on IAM and OAM instructions

Figure 9-2. PPU 24-bit Instruction Format

9.2 SYMBOLIC NOTATION

This section describes notation used for coding symbolic PPU machine instructions. Instructions are described in octal operation code sequence which generally reflects the mode of addressing. Instructions unique to either the 7600 or 6000-series computer systems are identified as such.

The location field of a symbolic PPU 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 PPU 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

	Use
	18-bit A register
	An expression that reduces to an 18-bit operand value.
	A 6-bit operand or operand address expression.
	A 12-bit expression value used with d or (d) to form an 18-bit operand or 12-bit operand address.
	12-bit Program Address register
	12-bit Q register
	An expression that reduces to a 6-bit value ($-37_8 \leq r \leq 37_8$) specifying relative address or shift count
	Contents of a register or location
	Refers to indirect addressing

tructions provide similar functions using different modes of addressing. They can be ing to function as shown below:

Description

nission

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 the upper 6 bits are set to one.

A store instruction stores the lower 12 bits of the A register contents into a memory location indicated by the instruction.

The contents of A are not altered.

Instruction	Octal Code	Section
LDN	14	9.2.3
LCN	15	9.2.3
LDC	20	9.2.4
LDD	30	9.2.9
STD	34	9.2.9
LDI	40	9.2.10
STI	44	9.2.10
LDM	50	9.2.11
STM	54	9.2.11

Function (cont'd)	Description (cont'd)

Arithmetic

A PPU 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.

Instruction	Octal Code	Section
ADN	16	9.2.3
SBN	17	9.2.3
ADC	21	9.2.4
ADD	31	9.2.6
SBD	32	9.2.6
ADI	41	9.2.7
SBI	42	9.2.7
ADM	51	9.2.8
SBM	52	9.2.8

Logical

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 for all but the LPN instruction for which the upper 12 bits are zero.

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	1100
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 one's.

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

Function (cont'd)	Description (cont'd)

Logical (cont'd)

Logical instructions include the following:

Instruction	Octal Code	Section
LMN	11	9.2.3
LPN	12	9.2.3
SCN	13	9.2.3
LPC	22	9.2.4
LMC	23	9.2.4
LMD	33	9.2.9
LMI	43	9.2.10
LMM	53	9.2.11

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.

Instruction	Octal Code	Section
RAD	35	9.2.9
AOD	36	9.2.9
SOD	37	9.2.9
RAI	45	9.2.10
AOI	46	9.2.10
SOI	47	9.2.10
RAM	55	9.2.11
AOM	56	9.2.11
SOM	57	9.2.11

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 ($01-37_8$), the jump is forward r locations. When r is negative (76_8-40_8), the jump is backward 77_8-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.

CAUTION

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 (section 4.3.2) and the PERIPH instruction (section 4.3.3) 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.

Formats:

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 or PERIPH J option has been selected, r is not valid. The contents of the variable field must be a symbolic address (tag).

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
0100 1362		LJM	START	
0271 0000		RJM	0,CTO	
0371		UJN	TAG1-*	
0404		ZJN	+4	
0525		NJN	TAG3	
0667		PJN	TAG2-*	
0726		MJN	TAG4	

In the above examples, the LJM instruction is at address 0014_8. TAG1 is address 0012_8, TAG2 has a value of 13_8, TAG3 has a value of 25_8, and TAG4 has a value of 26_8.

Code Generated

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18      30
                  PPU        J
0347              UJN        TAG1
0404              ZJN        TAG3      In this example, the UJN is at
0556              NJN        TAG2+10   address 0040. TAG1 is address
0602              PJN        -1+TAG4   0010, TAG2 is 0011, TAG3 is
0743              MJN        TAG1      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	r	Shift (A) by + (left) or - (right) r bits	12 bits	10d

Examples:

1. Shift contents of A left circular 6 places

Code Generated

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18      30
1006              SHN        6
```

2. Shift contents of A right end off 6 places

Code Generated

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18      30
      6
      SCNT        SET        6
1071              SHN        -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 core.

Formats:

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

Examples:

Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
1112			LMN	12B	
1207			LPN	7	
1321			SCN	21B	
	15	AA	SET	15B	
1415			LDN	AA	
1514			LCN	AA-1	
1601			ADN	1	
1702			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

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
2070 7070		LDC	7070708	
0	VAL	=	0	
2177 7776		ADC	VAL-1	
2207 0707		LPC	0707078	
70707	MASK	SET	0707078	
2307 0707		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.

Format:

Operation	Variable	Description	Size	Octal Code
PSN		No operation (Pass)	12 bits	2400

Example:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
2400		PSN		

Other octal operation codes (not generated by COMPASS) that act as pass instructions are:

6000 Series	7600
00	25
25	26
	27
	75
	76

9.2.6 EXCHANGE JUMP INSTRUCTIONS (6000-SERIES ONLY)

The EXN instruction transmits an 18-bit (absolute) address of which only 17 bits are used 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 PPU is not interrupted.

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 PPU A register. This address must be entered in A before the MXN instruction is executed.

In 6500 systems with dual central processors, d can be 0 or 1 and specifies which CPU the exchange jump will interrupt. In 6300, 6400 and 6600 systems, this value is not interpreted.

Formats:

Operation	Variable	Description	Size	Octal Code
EXN	d	Exchange jump to CPU d	12 bits	260d
MXN	d	Monitor exchange jump CPU d	12 bits	261d

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
2601		EXN	1	
2610		MXN	0	

9.2.7 READ PROGRAM ADDRESS INSTRUCTION (6000-SERIES ONLY)

This instruction transfers the contents of the CPU P register to the PPU A register; this allows the PPU to determine whether the CPU is in execution. In a 6500 system with dual central processors, the lowest order bit of the instruction format specifies which CPU P register is to be examined. In 6400 and 6600 systems, this bit is not interpreted. The largest value that (P) can be is 17 bits. An ECS transfer is in progress when bit 17 of this instruction is set. However, bit 17 of P is not set.

Format:

Operation	Variable	Description	Size	Octal Code
RPN	d	Read program address CPU d→A	12 bits	270d

Example:

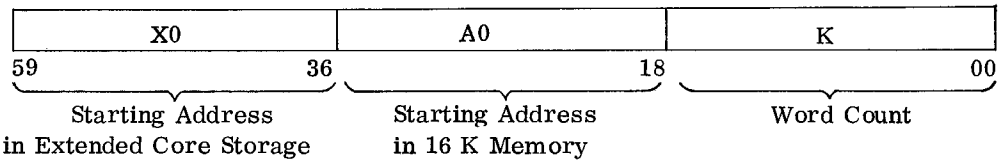
Code Generated

```
2700
```

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
      1         11         18        30
      
            RPN
```

9.2.8 6416 PPU 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 PPU 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 core storage. 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.

Formats:

Operation	Variable	Description	Size	Octal Code
ETN	d	Extended core transfer	12 bits	260d
ERN	d	Read extended core coupler status	12 bits	270d

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
2600		ETN		
2700		ERN		

9.2.9 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_8 addresses in core memory (0000 - 0077_8). During execution of LDD, ADD, and SBD, (d) is treated as a 12-bit positive 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 (d) and (A)→ 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

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
3012		LDD	TAG1	
3103		ADD	TAG2-10B	
3240		SBD	40B	
3327		LMD	TAG1+15B	
3401		STD	1	
3555		RAD	55B	
3612		AOD	TAG1	
3713		SOD	TAG2	

9.2.10 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_8 addresses in core memory (0000 - 0077_8).

On the 7600, the address formed permits referencing of all memory locations but one (0000 - 7776_8).

On a 6000-Series Computer System PPU, the address formed in indirect address mode permits referencing of all memory locations, including address 7777_8.

Formats:

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

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
4012		LDI	TAG1	
4103		ADI	TAG2-10	
4240		SBI	40B	
4327		LMI	TAG1+15B	
4401		STI	1	
4555		RAI	55B	
4612		AOI	TAG1	
4713		SOI	TAG2	

9.2.11 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_8 addresses in core memory (0000 - 0077_8). 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 (0000-776_8). Although m and/or (d) can have a value of 7777_8, the computer system does not permit m+(d) to reference address 7777_8.

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.

Formats:

Operation	Variable	Description	Size	Octal Code
LDM	m,d	Load (m+(d))→A	24 bits	50dm
ADM	m,d	Add (m+(d))→A	24 bits	51dm
SBM	m,d	Subtract (m+(d))→A	24 bits	52dm
LMM	m,d	Logical difference (A) - (m+(d))→A	24 bits	53dm
STM	m,d	Store (A)→m+(d)	24 bits	54dm
RAM	m,d	Replace add (m+(d)) + (A)→m+(d) and A	24 bits	55dm
AOM	m,d	Replace add one (m+(d)) + 1→m+(d) and A	24 bits	56dm
SOM	m,d	Replace subtract one (m+(d)) - 1→m+(d) and A	24 bits	57dm

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
5077 0203		LDM	TAG6,77B	
5106 0202		ADM	TAG5,6	
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	

9.2.12 CENTRAL READ/WRITE INSTRUCTIONS (6000-SERIES ONLY)

The CRD instruction transfers a 60-bit word from central memory to five consecutive PPU 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 (A) are not altered.

The CRM instruction reads a block of 60-bit words from central memory. The content of location d gives 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, (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 processed. The original content of P is restored at the end of the instruction.

(A) is advanced 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_8 to 0000_8. 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_8 to 0000_8. 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.

Formats:

Operation	Variable	Description	Size	Octal Code
CRD	d	Central read from (A) to d	12 bits	60d
CRM	m,d†	Central read from (d) CM words beginning at CM (A)→ PPU m	24 bits	61dm
CWD	d	Central write from d to (A)	12 bits	62d
CWM	m,d†	Central write (d) words beginning at PPU m→ CM (A)	24 bits	63dm

†Expression d is required

Examples:

Code Generated
6015
6125 0012
6232
6350 0012

```
      LOCATION  OPERATION  VARIABLE      COMMENTS
1                11         18            30
          CRD    15B
          CRM    TAG1,25B
          CWD    32B
          CWM    TAG1,50B
```

9.2.13 I/O BRANCH INSTRUCTIONS (6000-ONLY)

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. These instructions are exclusively 6000-series PPU instructions. 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 PPU accepts the word and clears the flag. An output channel remains full when a PPU 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 PPU.

Formats:

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

Examples:

Code Generated	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
6402 0012		AJM	TAG1,2	
6502 0013		IJM	TAG2,CHAN-2	
6604 0025		FJM	TAG3,4	
6704 0026		EJM	TAG4,CHAN	

9.2.14 I/O BRANCH INSTRUCTIONS (7600 ONLY)

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.

Formats:

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

Examples:

Code Generated		LOCATION	OPERATION	VARIABLE	COMMENTS
		1	11	18	30
6005 1365			FIM	TAG5,5	
6102 1365			FIM	TAG5,2	
6201 1366			IRM	TAG6,1	
	4	CHAN	SET	4	
6304 1366			NIM	TAG6,CHAN	
6415 7000			FOM	7000B,15B	
6500 1525			EOM	140B+TAG5,0	
6601 1266			ORM	-100B+TAG6,CHAN-3	
6705 1366			NOM	TAG6,CHAN+1	

9.2.15 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 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 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 d cable.

On a 6000-series machine, executing either of these instructions when the channel is inactive causes the peripheral processor unit to become inoperative until some other peripheral processor activates the channel, or the system is deadstarted

Formats:

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

Examples:

Code Generated

```
7003
7204
```

```
      LOCATION  OPERATION  VARIABLE  COMMENTS
1                 11         18        30
          IAN     3
          OAN     CHAN
```

9.2.16 BLOCK INPUT/OUTPUT INSTRUCTIONS

The following instructions tranfer a block of 12-bit words on channel d to or from a starting PPU 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 (6000 only).

On a 6000-series machine, the input operation is complete when (A) = 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 0's. The word count is not affected by this empty word. Therefore, the contents of the A register gives 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 advances by one to give the address for the next word as each word is transferred.

NOTE

If this instruction is executed on a 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 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 PPU block input storage area contains a noise word; any remaining locations are unaltered. Note that the storage location can be incremented through location 7776_8 to 0000_8 on a 7600, or location 7777 through 0000 on a 6000-series machine which could destroy existing data or a program.

On 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.

Formats:

Operation	Variable	Description	Size	Octal Code
IAM	m,d†	Input: (A) words to m from channel d	24 bits	71dm
OAM	m,d†	Output: (A) words from m to channel d	24 bits	73dm

†Expression d is required

Examples:

Code Generated
7103 1364
7304 1364

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		IAM	TAG,3	
		OAM	TAG,4	

9.2.17 SET OUTPUT RECORD FLAG INSTRUCTION (7600 ONLY)

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

7411

	LOCATION	OPERATION	VARIABLE	COMMENTS
1		11	18	30
		RFN	11B	

9.2.18 CHANNEL FUNCTION INSTRUCTIONS (6000-SERIES ONLY)

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 PPU to become inoperative until another PPU 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 PPU 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, PPUs 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 PPU to become inoperative until another PPU or an external equipment deactivates the channel, or the system is deadstarted.

Formats:

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

Examples:

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
7405		ACN	5	
7504		DCN	CHAN	
7605		FAN	CHAN+1	
7705 0020		FNC	20B,5	

9.2.19 ERROR STOP INSTRUCTION (7600 ONLY)

The ESN instruction halts execution of the 7600 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. This instruction is exclusively a 7600 PPU instruction.

Format:

Operation	Variable	Description	Size	Octal Code
ESN	d	Error Stop	12 bits	7700

Example:

Code Generated

	LOCATION	OPERATION	VARIABLE	COMMENTS
	1	11	18	30
7700		ESN		

PROGRAM EXECUTION 10

COMPASS can be called from the library and placed in execution through a COMPASS call card or through an IDENT card (section 4.2.1) in a FORTRAN source deck. When COMPASS is called through FORTRAN, parameters are ordinarily specified on the RUN or FTN card and are the same as for the FORTRAN program.

10.1 CONTROL CARDS

Normal assembly of COMPASS source programs under SCOPE and execution of CPU binary object decks requires generation of a SCOPE job file. A file is usually submitted in the form of card decks or card images. The first record † of the file must contain the SCOPE control cards described in this section. Other optional cards are described in the operating system reference manual. Following the control card record are one or more records containing source statements and data. A control card key word, i.e., name on the job card, begins in column one. A comma or a left parenthesis begins a parameter string. Parameters in the string are separated by commas. A period or right parenthesis terminates a parameter string. Comments optionally follow the terminator.

10.1.1 JOB CARD

A job card of the following format must be the first card in the deck. The parameters following name can be in any order or can be omitted. For any omitted field, SCOPE supplies a default value, which is an installation option.

Control card format:

```
name, Pp, Tt, CMscm, ECmc.
```

name — 1-7 character alphanumeric name by which the job is identified. The first character must be a letter.

Pp — Job priority

SCOPE 1

P4	Express job
P3	Preferred job; the job will be placed in execution before a job of lower class.
P2 or omitted	Normal job; the job will be placed inexecution before any deferred job but after any priority job.
P1	Deferred job; may be assigned to a job that has a high proportion of CPU time to I/O time.
Other	The system uses an installation option.

†Section for SCOPE 2.

SCOPE 2 and SCOPE 3.3

Pp — p = priority level in octal. $1 \leq p \leq 2^k$, where k is an installation option less than or equal to 8. The lowest priority is 1.

Tt — CPU time limit in octal seconds ($1\text{-}7777_8$), must be sufficient to process all control cards for the job, including assembly and execution.

CMscm — Estimate of maximum amount of 7600 SCM or 6000 CM required for execution (1 - 6 octal digits). The estimate for COMPASS is a minimum of 40000. The 7600 operating system rounds the value to the next higher multiple of 1000. The 6000 operating system round this value to the next higher multiple of 100.

ECIcm — Estimate of maximum amount of 7600 LCM or 6000 ECS in octal thousands, required for assembly or execution ($1 - 1400_8$). The estimate for COMPASS is a minimum of none for SCOPE 1 and SCOPE 3.3 and is 14 for SCOPE 2 (i.e., 14000_8).

COMPASS notes storage used on the listable output. For subsequent runs, the field lengths can be decreased accordingly.

Examples:

```
JOB1, P2, T100, CM40000, EC30.
TESTER.
```

10.1.2 COMPASS CALL CARD

The following control card causes the COMPASS assembler to be loaded from the SCOPE library and executed. Parameters specify modes and files.

Control card format:

```
COMPASS(p1, p2, ..., pn)
```

The optional parameters, p_i, 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 = filename

Mode is one or two characters as described; and filename is a 1-7 character name of a file or a character string.

Mode	Significance
A - Abort mode	
A	Abort job at end of run if any assembly errors occurred.
omitted	Do not abort job for assembly errors.
B - Binary output	
omitted or B	Binary on the load-and-go file (LGO)
B=0	No binary output
B=filename	Binary on the named file
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). 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 - Extended binary table mode (SCOPE 2 only)	
omitted or E=1	Tables are generated as described in Loader Reference Manual, Publication No. 60344200.
E or E=0	Tables are generated non-extended as described in Appendix B.
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 FTN = 2
G - Get text; takes precedence over S if both G and S options are selected.	
G	Load systems text from file SYSTEXT †
G=filename	Load systems text from named file
omitted or G=0	Load systems text from overlay named in S option
I - Source of assembler input	
omitted	Source deck is on INPUT file
I	Source deck is on COMPILE file in either compressed or expanded format.
I=0	Illegal
I=filename	Source deck is on named file

†V2TEXT for SCOPE 2.

Mode	Significance
L - Full List	
omitted or L	List output on OUTPUT file
L=filename	List output on named file. When the full list is on a different file than the short list, the listing for each subprogram is preceded by 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. Also see O option.
L=0	No full list will be generated
LO-List options; selects or deselects a maximum of seven 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=$c_1c_2\ldots c_n$	A list of up to seven 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, section 4.11.1.
N - No eject; suppresses ejects caused by normal listing control. The only page ejects are at the beginning of new subprograms.	
N	No eject
omitted	Normal ejects
O - Short list; suppressed if full list is directed to the file or if no assembly errors occur. However, if the full list and short list are on different files (e.g., the full list is written on OUTPUT and the short list is written on the named file), the short list will be augmented by the addition of any error lines originating with a macro call.	
omitted or O	List output on OUTPUT file
O=filename	List output on named file
O=0	No short list will be generated
P - Continue page	
P	Page numbering continues from subprogram to subprogram.
omitted	Page numbering begins with 1 for each END instruction.
S - Source of SYSTEXT† information	
omitted or S	Information is on SYSTEXT† overlay
S=0	No source of SYSTEXT† information
S=overlay	Information is on named library overlay

†V2TEXT for SCOPE 2.

X - Source of external text (XTEXT) when location field of XTEXT pseudo instruction is blank.

omitted	External text OLDPL file
X=filename	External text on named file
X=0	Illegal
X	External text on OPL file †

Examples:

```
COMPASS(B,D,S=OVI)
COMPASS(LO=ASGXD)
COMPASS.
```

10.1.3 LGO CONTROL CARD

An LGO control card calls for the loading and execution of CPU binary output produced by the assembler when the B option on the COMPASS card is selected. When binary output is on some file other than LGO, the card is replaced by a program call card for that file. SCOPE automatically repositions the LGO at load point before loading. The LGO file is temporary; it is released at job termination.

Format:

```
LGO. comments
```

10.1.4 PROGRAM CALL CARD

The program call card directs the operating system to search for a file or CPU program that has the name specified on the card, load it into the user's small core memory, and execute it as a CPU program.

Formats:

```
name(p1,p2,...,pn)
name.
```

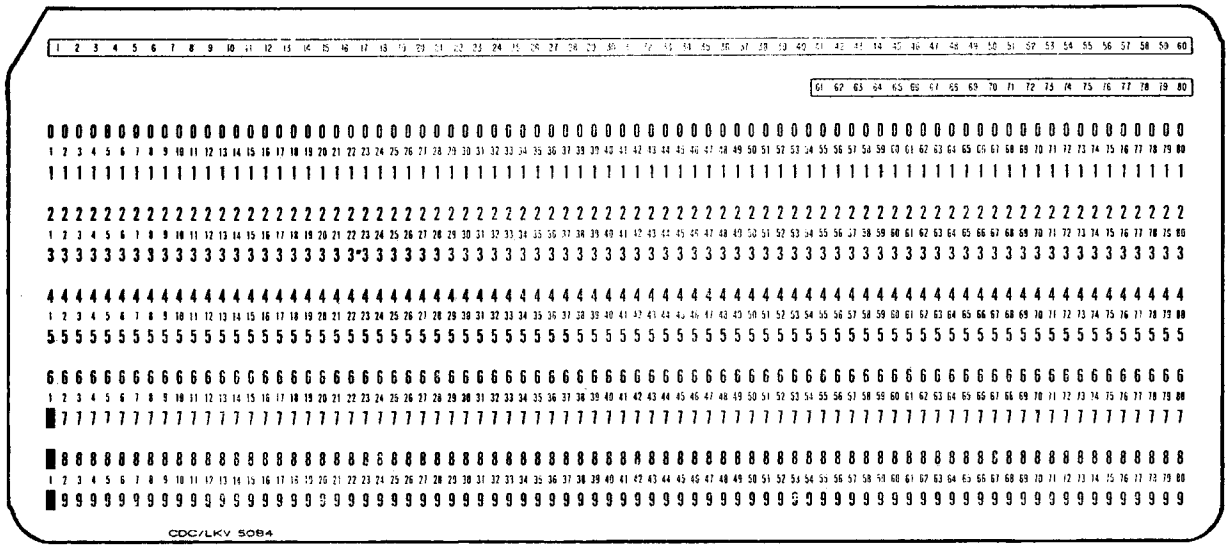
name	Program name
p_i	Parameters in a format acceptable to the program being called.

When the operating system locates the file, it repositions the file to load point, loads it, and, when loading is complete, executes the program as a CPU program.

†MODIFY is not supported by SCOPE 2.

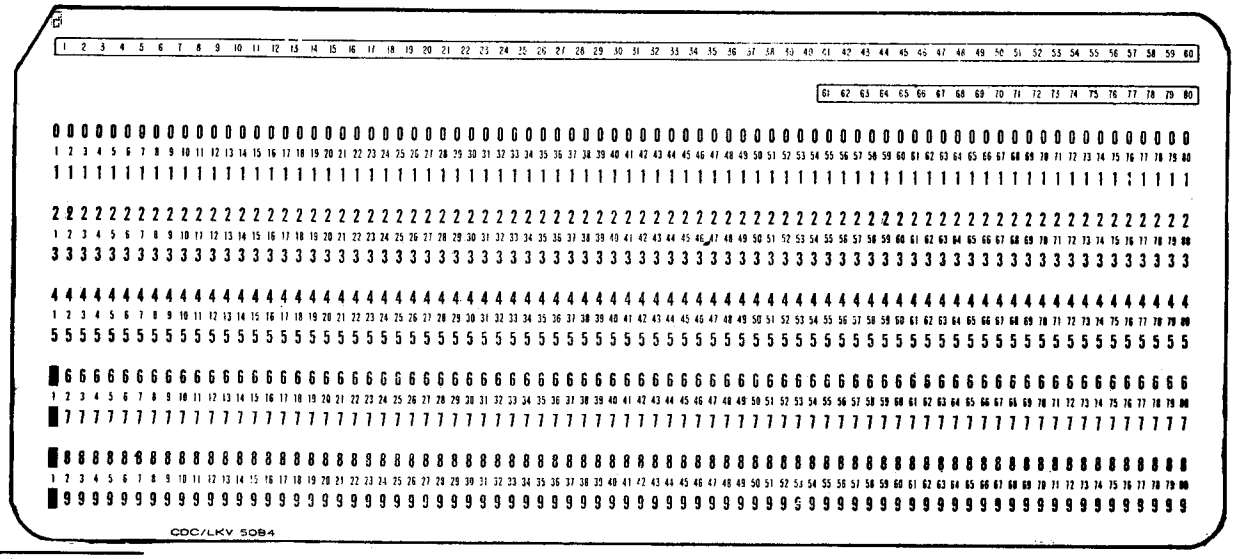
10.1.5 END-OF-RECORD CARD

An end-of-record† card is characterized by having rows 7, 8, and 9 punched in column one. Columns 2-80 optionally contain comments. The card separates records in the job deck. For example, a deck consisting of control cards, a source language deck for one or more subprograms, and a data deck, would include two end-of-record cards. The first terminates the control cards and the other terminates the source deck. A deck consisting of control cards only does not require an end-of-record card; an end-of-information card is used in its place.



10.1.6 END-OF-INFORMATION CARD

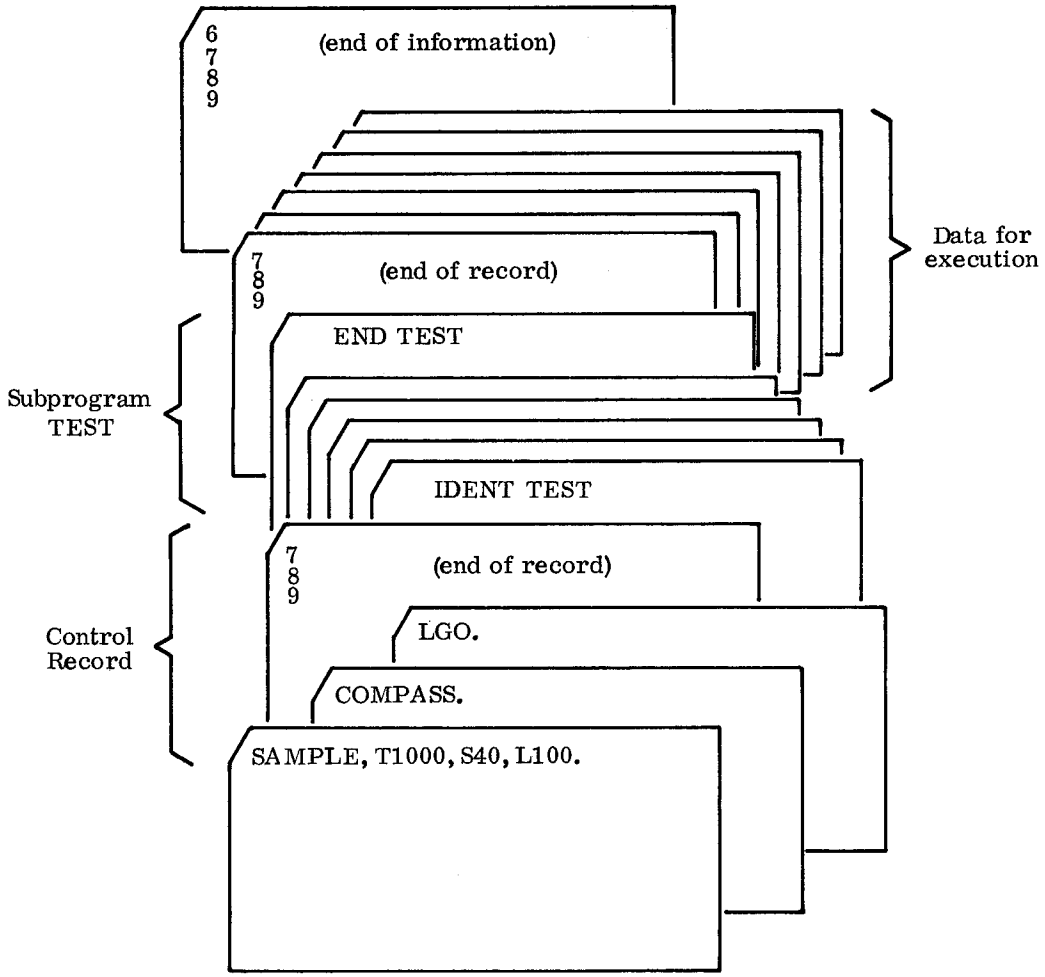
An end-of-information card is characterized by having rows 6, 7, 8, and 9 punched in column one. Its use signals the end of the job deck. Columns 2-80 optionally contain comments.



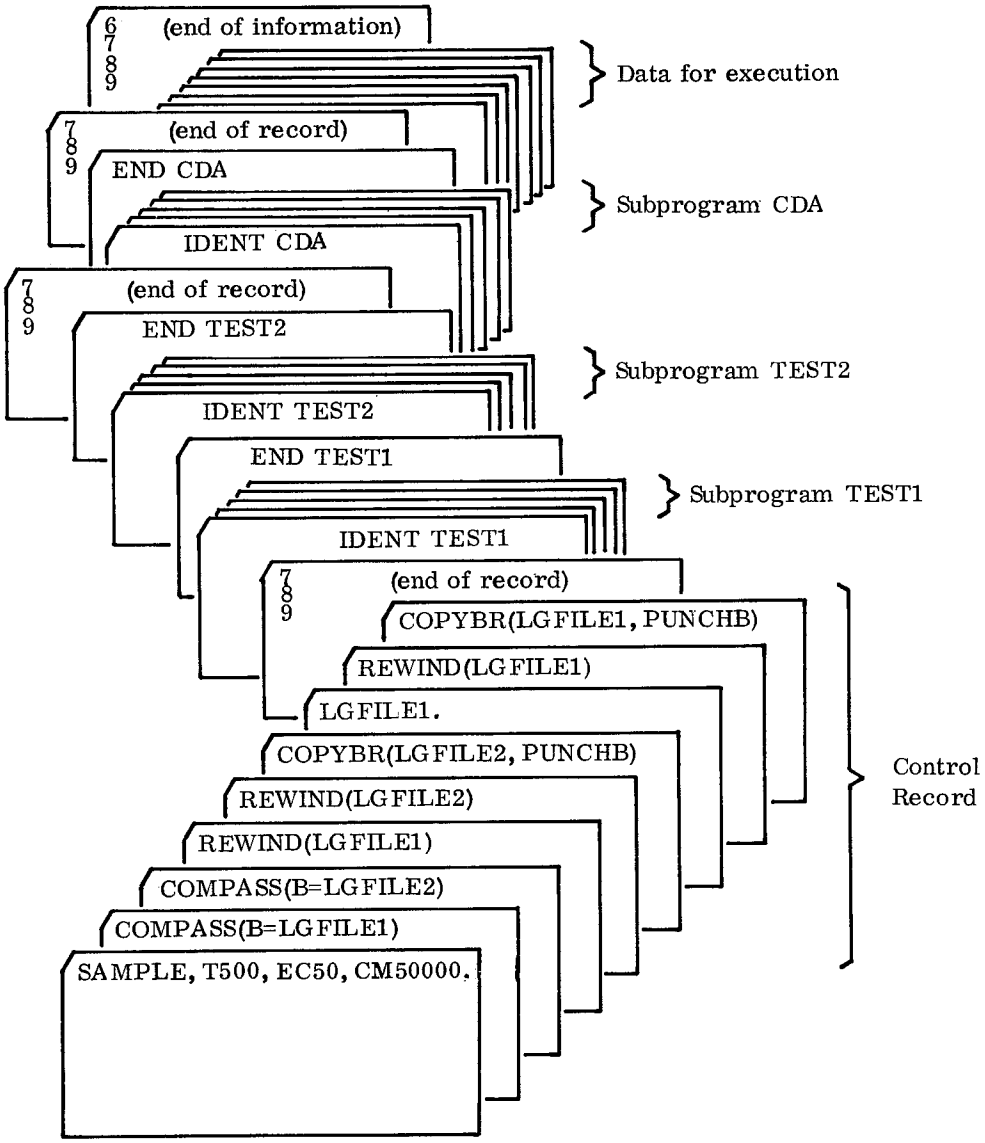
†End of section for SCOPE 2.

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. COMPASS reads source statements from file INPUT, writes the listing on OUTPUT, and writes a binary object deck on file LGO. Control card LGO calls for execution of the binary object program, which obtains its data from file input.



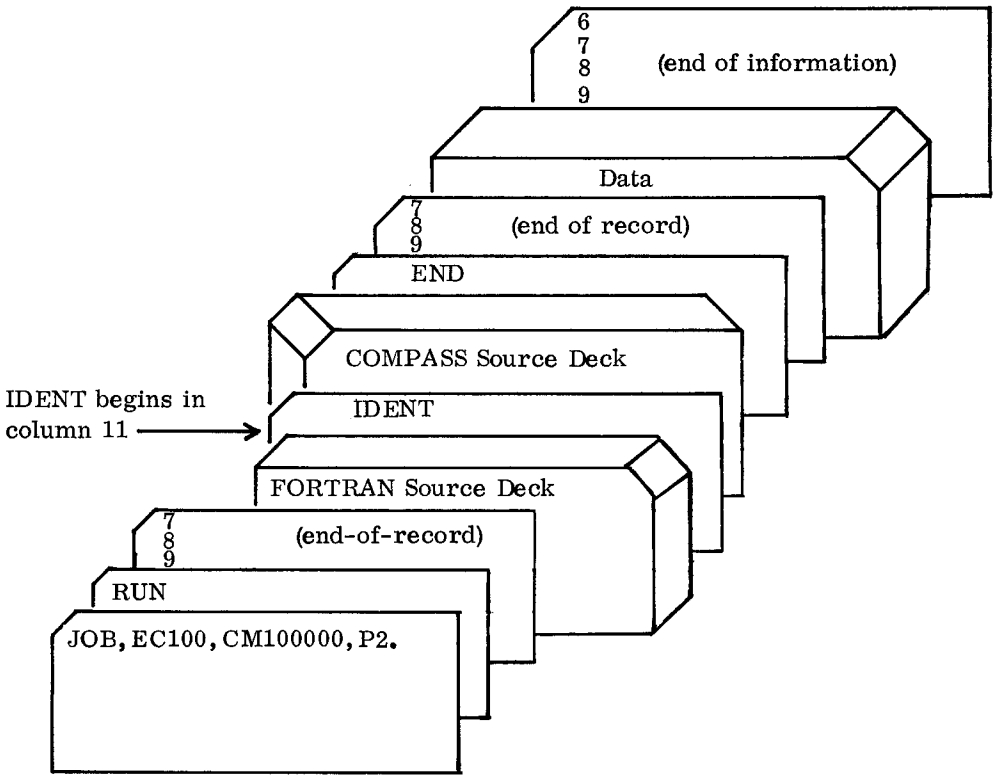
In the following job, the COMPASS assembler is called twice. During the first assembly, binary object decks for subprograms TEST1 and TEST2 are written on file LGFILE1. The source decks for these subprograms are on the second record of the INPUT file. During the second assembly, COMPASS writes a binary object deck for subprogram CDA on file LGFILE2. Each assembler run produces a full listing. Following the second assembly, both files containing binary output are 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 card 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, COMPASS is called from within a FORTRAN program. The source program follows the FORTRAN program in the same record.

No parameters on the RUN card cause:

1. Compilation and execution
2. Object program SCM and LCM fields to be set
3. Source decks on INPUT
4. Listings to be written on OUTPUT
5. Binary object programs to be written on LGO
6. No cross reference list is produced



The following sample job deck illustrates how to assemble and use a system text overlay.

```
TEXT,CM60000,T300.
COMPASS(S=0,B=TEXT)
COMPASS(G=TEXT)
7/8/9
    IDENT   TEXT
    STEXT
      .                          } Contains definitions for
      .                          } system macros, micros,
      .                          } and symbols.
    END
7/8/9
    IDENT   PROGRAM
    SST                          Defines symbols from TEXT.
     .                           } Programs using definitions
     .                           } in TEXT.
     .                           }
    END
6/7/8/9
```

LISTING FORMAT 11

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 Card.

11.1 PAGE HEADING

Each page of the assembly listing contains a title line and a subtitle line in the following format:

<table>
<tr><td>title</td><td colspan="2">COMPASS - VER n.</td><td>date</td><td>time</td><td>PAGE x</td></tr>
<tr><td>subtitle</td><td>sub-sub
title</td><td>block
name</td><td>symbol
qual</td><td colspan="2"></td></tr>
</table>

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 card
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) at beginning of page

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 card or the LIST instruction selects the B list option:

ADDRESS	LENGTH	BINARY CONTROL CARDS.
$addr_1$	ℓ_1	binary $card_1$
$addr_2$	ℓ_2	binary $card_2$
.	.	.
.	.	.
.	.	.
$addr_n$	ℓ_n	binary $card_n$

binary $card_i$	The binary card that caused generation of the binary for the overlay, partial binary, or subprogram. The list includes SEG, SEGMENT, and END instructions.
$addr_i$	The origin address for the subprogram, overlay, or partial binary written out as a result of the binary card.
ℓ_i	The length of the subprogram, overlay or partial binary.

Example:

```
ADDRESS LENGTH              BINARY  CONTROL CARDS.
    1N1    271              IDENT   COMPASS,LOVER,CMP
    372   5241              SEG
   5633   1242              SEG
   7075   4145              SEG
  13242   5175              SEG
  20437   1352              SEG
  22011                     END     COMPASS
```

11.2.2 BLOCK USAGE SUMMARY

A block usage summary of the following format is generated under control of the B list option.

BLOCKS	TYPE	ADDRESS	LENGTH
$name_1$	t_1	$baddr_1$	$b\ell_1$
$name_2$	t_2	$baddr_2$	$b\ell_2$
⋮	⋮	⋮	⋮
$name_n$	t_n	$baddr_n$	$b\ell_n$

$name_i$ 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	Identifiers 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.
COMMON	Addresses in the block are relative to the origin of the common block.
LCM	Addresses in the block are relative to the named LCM block for a relocatable program and are relative to RAL for an absolute program.

$baddr_i$ Beginning address of the block according to type.

$length_i$ 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
TABLE       COMMON           0           1
TABLE       LOCAL           36           1
LCM         LCM              0         400
ALPHA       LOCAL           37        1751
BETA        LOCAL         2010         144
//          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_1 - $addr_1$	sym_{n+1} - $addr_{n+1}$	sym_{2n+1} - $addr_{2n+1}$	sym_{3n+1} - $addr_{3n+1}$
sym_2 - $addr_2$	sym_{n+2} - $addr_{n+2}$	sym_{2n+2} - $addr_{2n+2}$	sym_{3n+2} - $addr_{3n+2}$
.	.	.	.
.	.	.	.
.	.	.	.
sum_n - $addr_n$	sym_{2n} - $addr_{2n}$	sym_{3n} - $addr_{3n}$	

If the symbol is undefined, $addr_i$ is *******.

Example:

```
ENTRY POINTS.

SNAP1  - 1345   CONOUT - 1175   GOTO   -   16   READ   -  36
SNAP2  - 1352   ADD    - 1347   IF     - 1354   RECORD -  37
SNAP3  - 1357   ADDR   - 1247   KEY    -   17   REORDER -  40
SNAP4  - 1364   BEGIN  -    0   LABEL  -   20   RPF    -  41
SNAP5  - 1510   BYTESIZ -    1   LEVEL  -   21   RPH    -  42
SNAP6  - 1632   CALL   -    2   LIMIT  -   22   SC     -  43
```

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_1	sym_{n+1}	sym_{2n+1}	sym_{3n+1}	. . .	sym_{7n+1}
sym_2	sym_{n+2}				
sym_3	sym_{n+3}				
⋮	⋮				
sym_n	sym_{2n}				

where n is 1/8th the number of external symbols.

Example:

```
EXTERNAL SYMBOLS.

ERMSG    CONEXIT  XDECBI   SYMBOL  COGOTO   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.

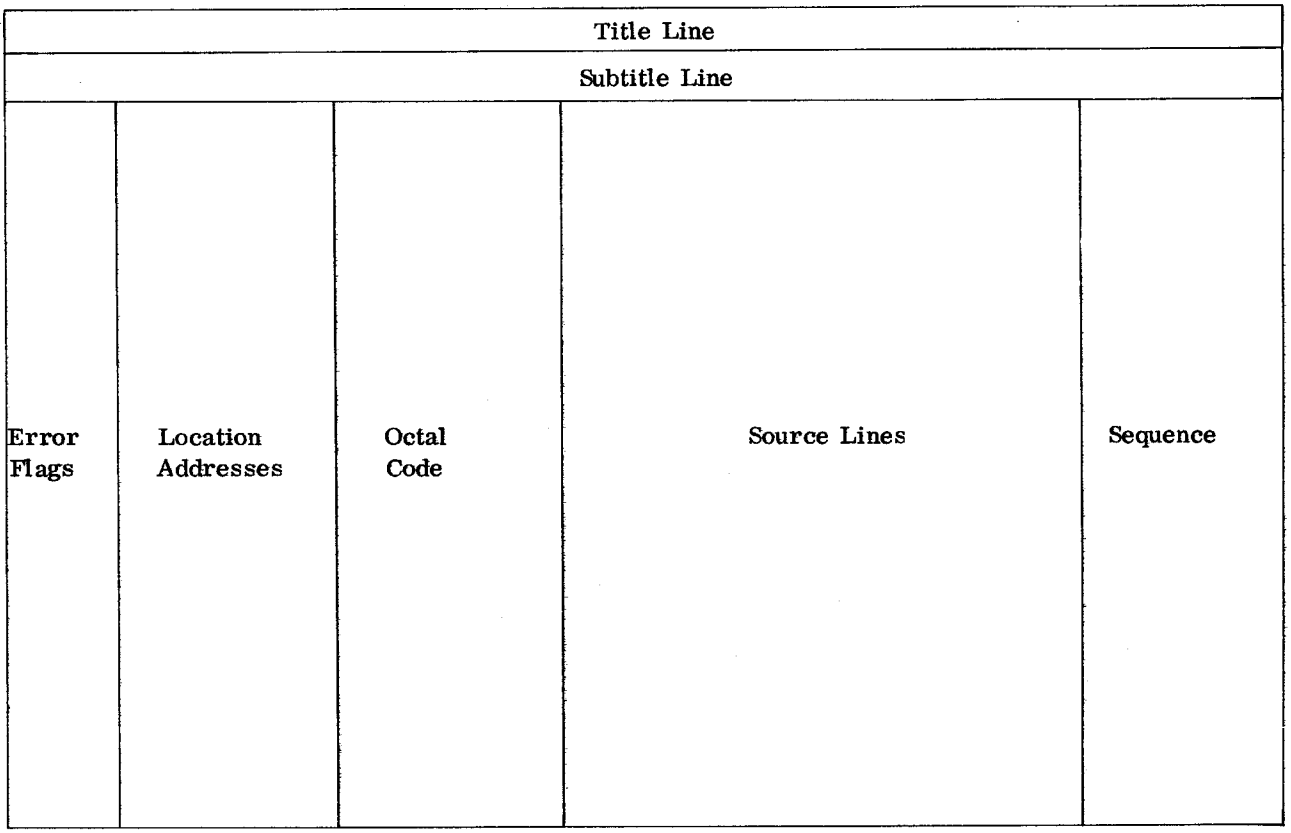


Figure 11-1. Format of Octal and Source Statement Listing

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
- `+` Postive 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 $b_1 \lceil\!\rightarrow b_2$ 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 $c_1 \lceil\!\rightarrow c_2$ 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.

Source Code — Source statement image (columns 1-72)

Sequence — Columns 73-90 of the card image or an identifier for an expansion of a definition operation as follows:

Macro	macro name
Remote code	*RMT*
Duplicated code	*DUP*
Echoed code	*ECHO*
XTEXT	file name
OPDEF	Operation field of opdef call, e.g., SB1

The recursion level is indicated in the right half of the field.

Example:

```
COMPASS - 6000/7000 ASSEMBLER - VER  .                    COMPASS - VER  .    70/05/05.  09.58.39.     PAGE    38
COMPASS MAIN BATCH CONTROL.                               COMPASS

                            **        COMPASS - MAIN CONTROL.                                          COMPASS   807
                                                                                                       COMPASS   808
                                                                                                       COMPASS   809
  4132  6110000001                    COMPASS  SB1    1           (01) = CONSTANT 1                  COMPASS   810
                  0100022354            RJ      PASS0       INITIALIZE COMPASS                       COMPASS   811
  4133  0400007334                    COMPASS4 EQ     /PASS1/PASS1 EXECUTE PASS1                     COMPASS   812
  4134  0400007571                    EXITP1   EQ     /PASS2/PASS2 EXECUTE PASS2                     COMPASS   813
  4135  0100004137                    EXITP2   RJ     TFE         TEST FOR END                       COMPASS   814
  4136  0400004137                             EQ     COMPASS4    CONTINUE                           COMPASS   815



                            **        TFE - TEST FOR END.                                              COMPASS   817
                                                                                                       COMPASS   818
                                                                                                       COMPASS   819
  4137  0000000000                    TFE      PS                 RETURN EXIT                        COMPASS   820
  4140  5110003147                             SA1    ERCNT       PROPAGATE ERROR COUNT              COMPASS   821
                  5120000062                   SA2    ERRFLG                                         COMPASS   822
  4141  36612                                  IX6    X1+X2                                          COMPASS   823
          54620                                SA6    A2                                             COMPASS   824
                  5110003075                   SA1    EOFINP      CHECK FOR EOR ON INPUT             COMPASS   825
  4142  5120000077                             SA2    BATCH                                          COMPASS   826
                  5130000042                   SA3    INBUF+1                                        COMPASS   827
  4143  5140006566                             SA4    =6LIDENT                                       COMPASS   828
                  0311004152                   NZ     X1,TFE2     IF EOR HAS BEEN FOUND              COMPASS   829
  4144  0332004150                             NG     X2,TFE1     IF NOT COMPASS ONLY                COMPASS   830
                  5120000070                   SA2    LISTFG                                         COMPASS   831
  4145  0312004137                             NZ     X2,TFE      IF LIST ON                         COMPASS   832
                  5120006567                   SA2    =10H-                                          COMPASS   833
  4146  10622                                  BX6    X2           SET DOUBLE SPACE                  COMPASS   834
          5160003171                           SA6    TITBUF                                         COMPASS   835
  4147  0400004137                             EQ     TFE         RETURN                             COMPASS   836
  4150  43044                          TFE1     MX0    36                                             COMPASS   837
                                                BX1    X3*X0       CHECK NEXT CARD FOR                ASS       838
```

11.4 LITERALS

When the D list option has been selected, the assembly listing includes a listing of the literals block following the default symbols listing. Following each literal address is the octal contents of the word and a display code conversion of the contents of the word.

Examples:

```
                    CONTENT OF LITERALS BLOCK.

010121    1745577375300000000     0+.>>X
010122    1665000000000000000     N
010123    1505232301070555636     MESSAGE 33
010124    5504050311150114552      DECIMAL R
010125    0521251122050457000     EQUIRED.
010126    5522052125112205040      REQUIRED
010127    0000000000000000000
010130    2022170722011555102     PROGRAM AB
010131    1722245700000000000     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                    TE
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                          ABC
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

Machine on 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 p_1, p_2, p_3, ... p_n

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

Error Type	Definition
A	ADDRESS FIELD BAD. Indicates any of a number of possible errors in a variable subfield entry. For example: CODE character not A, D, E, or I Symbol or name greater than 8 characters Expression does not reduce to one external term, relocatable terms do not cancel properly, instruction disallows register designators, instruction requires absolute expression, etc. Data error; 8 or 9 encountered in octal data, modifier not S, P, O, E, D, or B. No data in variable field of LIT instruction No symbol following an =S or =X prefix Relative jump out of range (-31>r >31) on PPU instruction BASE character not O, M, D, or * Register illegal in CON instruction Unable to locate synonymous instruction for OPSYN or CPSYN Micro count less than zero or greater than ten NOLABEL character not I Negative relocation on ORG POS value less than 0 or greater than word size VFD attempts to place relocatable address in CPU instruction when position counter is not 30, 15, or 0 Erroneous OPDEF reference
D	DOUBLY DEFINED SYMBOL. THE FIRST DEFINITION HOLDS. Symbol previously defined or declared external
E	ECHO, DUP, RMT, OR MACRO ILLEGALLY NESTED. Definition not wholly within next outer definition
F	NUMBER OF ENTRIES EXCEEDS PERMISSIBLE AMOUNT. LIT generates more than 100 words Data missing or erroneous on XTEXT file More than 63 formal parameters and local names in macro definition More than 255 blocks

TABLE 11-1. FATAL ERRORS (cont'd)

Error Type	Definition
L	LOCATION FIELD BAD. Required location field entry is erroneous. Format two macro definition has no substitutable parameters.
N	NEGATIVE RELOCATION ON ENTRY POINT.
O	OPERATION FIELD BAD. Instruction unrecognizable, out of sequence (e.g., ABS or PPU not in first statement group relational mnemonic on IF statement is erroneous. Location symbol begins beyond column two.
P	CONSULT LISTING FOR REASON BEHIND P-ERROR. User-generated error flag (ERR or ERRxx instruction)
R	DATA ORIGIN OUTSIDE BLOCK OR IN BLANK COMMON. Range error
U	UNDEFINED SYMBOL. VALUE ASSUMED 0. Reference to a symbol that is not defined; e.g., IF statement line count, DIS word count, unrecognizable attribute on IF statement, and undefined qualifier.
V	BIT COUNT ERROR ON VFD (MUST BE $0 \leq$ COUNT ≤ 60). VFD field size erroneous.

TABLE 11-2. INFORMATIVE ERRORS

Error Type	Description
1	LOCATION SYMBOL BAD. SYMBOL NOT DEFINED. Location field entry erroneous. The instruction does not require an entry.
2	ADDRESS ERROR ON SYMBOL DEFINITION. Erroneous variable field entry. The location field symbol is not defined.
3	DUPLICATE MACRO DEFINITION. NEW ONE OVERRIDES. Macro, opdef, or synonymous operation redefines operation code.
4	BAD FORMAL PARAMETER NAME IGNORED. Macro or ECHO formal parameter name repeated or illegal.
5	CPU OPERATION SYNTAX INCORRECTLY SPECIFIED. OPDEF, CPOP, CPSYN, or PURGDEF specifies illegal syntax.
6	LOCATION FIELD MEANINGLESS. Entry in location field is ignored.
7	ADDRESS VALUE EXCEEDS FIELD SIZE, RESULT TRUNCATED. Value of expression exceeds size of destination field. BSS address expression value is negative. MICRO starting character position or character count is negative.
8	MISSING OR EXTRA ADDRESS SUBFIELD. Variable subfield entry missing or superfluous.
9	MICRO SUBSTITUTION ERROR. NO SUBSTITUTION. Micro reference unrecognized.

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.

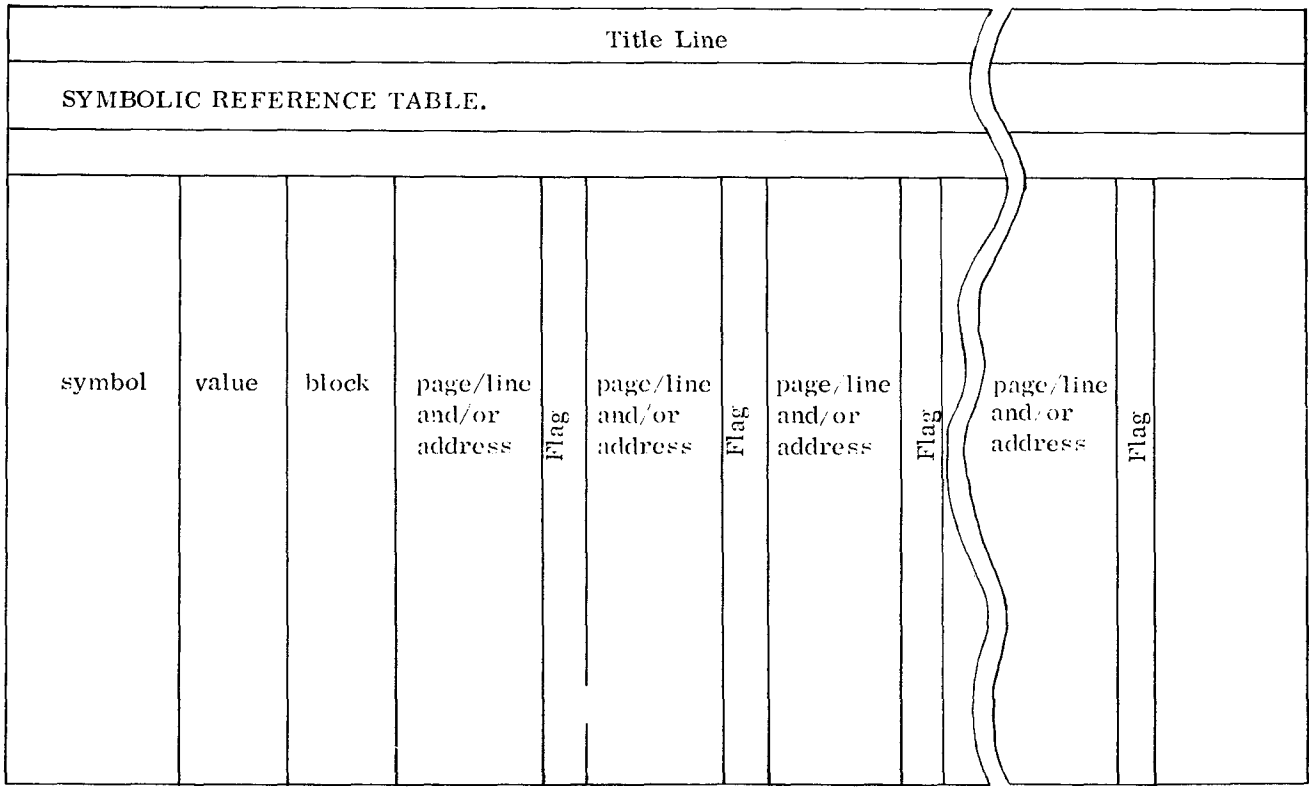


Figure 11-2. Format of Symbolic Reference Table

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	For an absolute assembly, this field is blank. For a relocatable assembly, it identifies 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.

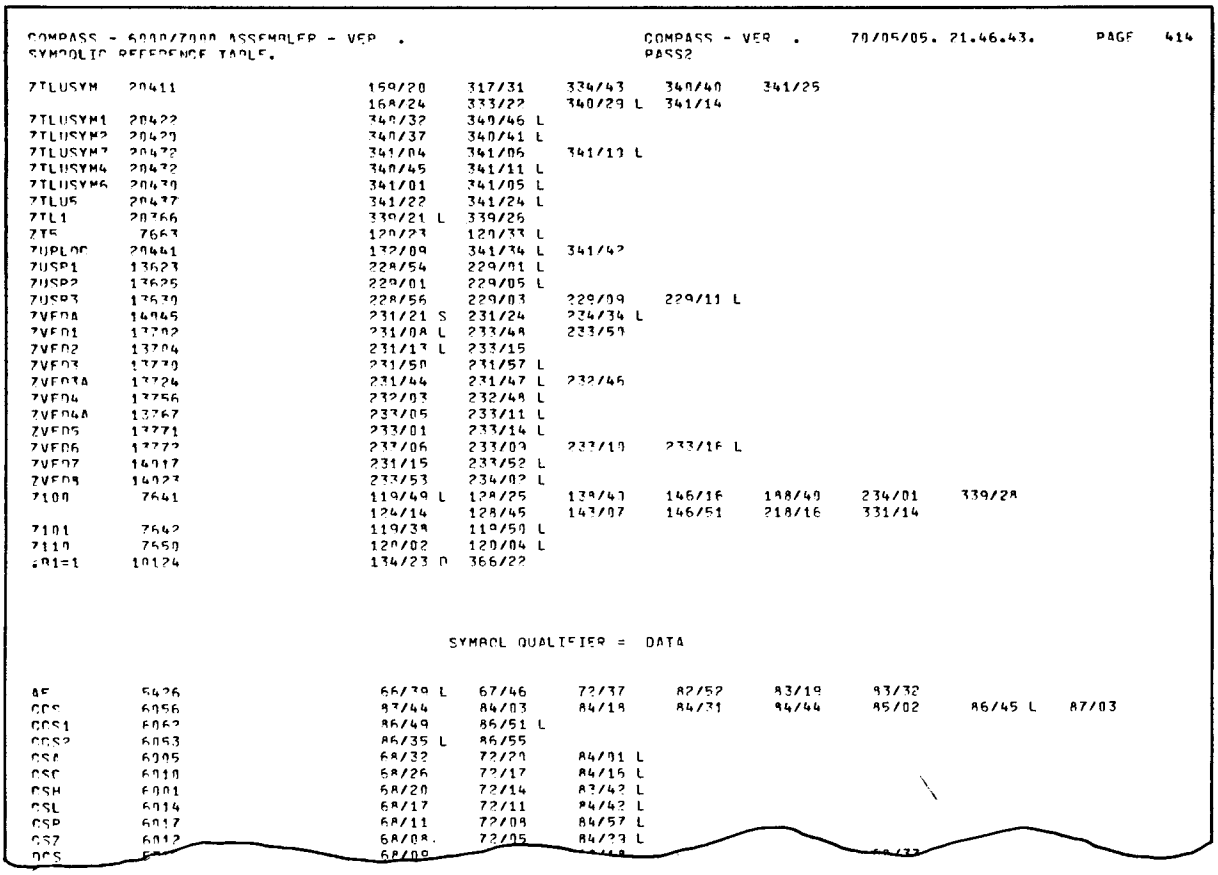
address — When the XREF pseudo (section 4.11.8) has been used, the page line field contains the location counter address of the instruction containing the reference. Page and line numbers are optionally included with the address.

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 pseudo instruction
- F Symbol used in conditional test
- 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 - 6000/7000 ASSEMBLER - VER  .                    COMPASS - VER  .     70/05/05. 21.46.43.      PAGE   414
SYMBOLIC REFERENCE TABLE.                                 PASS2

7TLUSYM   20411                 159/20    317/31    334/43    340/40    341/25
                                164/24    333/22    340/29 L  341/14
7TLUSYM1  20422                 340/32    340/46 L
7TLUSYM2  20429                 340/37    340/41 L
7TLUSYM3  20432                 341/04    341/06    341/13 L
7TLUSYM4  20432                 340/45    341/11 L
7TLUSYM5  20439                 341/01    341/05 L
7TLU5     20447                 341/22    341/24 L
7TL1      20766                 339/21 L  339/26
7T5        7663                 120/23    120/33 L
7UPLOC    20441                 132/09    341/34 L  341/42
7USP1     13623                 228/54    229/01 L
7USP2     13625                 229/01    229/05 L
7USP3     13630                 228/56    229/03    229/09    229/11 L
7VF00     14945                 231/21 S  231/24    234/34 L
7VF01     13702                 231/0A L  233/48    233/59
7VF02     13704                 231/13 L  233/15
7VF03     13720                 231/50    231/57 L
7VF03A    13724                 231/44    231/47 L  232/46
7VF04     13756                 232/03    232/48 L
7VF04A    13767                 233/05    233/11 L
7VF05     13771                 233/01    233/14 L
7VF06     13772                 233/06    233/09    233/19    233/1F L
7VF07     14017                 231/15    233/52 L
7VF08     14023                 233/53    234/02 L
7100       7641                 119/49 L  128/25    139/41    146/16    198/49    234/01    339/28
                                124/14    128/45    143/07    146/51    218/16    331/14
7101       7642                 119/39    119/50 L
7110       7650                 120/02    120/04 L
=01=1     10124                 134/23 D  366/22




                                        SYMBOL QUALIFIER =  DATA

BF         5436                 66/39 L   67/46     72/37     82/52     83/19     93/32
CCS        6056                 83/44     84/03     84/18     84/31     94/44     85/02     86/45 L   87/03
CCS1       6062                 86/49     86/51 L
CCS2       6063                 86/35 L   86/55
CSA        6005                 68/32     72/20     84/01 L
CSC        6010                 68/26     72/17     84/16 L
CSH        6001                 68/20     72/14     83/42 L
CSL        6014                 68/17     72/11     84/42 L
CSP        6017                 68/11     72/09     84/57 L
CSZ        6012                 68/08.    72/05     84/23 L
DCS        6[illegible]         68/09
```

6000 AND 7600 TIMING NOTES A

7600 CPU TIMING NOTES

1. Times given in Table A-1 include clock period known to occur before instruction issue, but do not consider register conflict conditions that might delay issue.

   Except for the multiply and divide units, all functional units permit new instructions to enter them every clock period. A new instruction may enter the multiply unit in any clock period, provided there was no multiply operation initiated in the preceding clock period. A new instruction can enter the divide unit two clock periods prior to completion of a previous divide operation. Once an instruction issues to a functional unit, it is executed in a fixed amount of time. No delays are possible.

   Times given for instructions 01 to 07 and 50 to 57 do not consider memory conflict conditions or SAS backup conditions caused by bank conflicts.

2. Execution of Block Copy instructions (011 and 012) will be delayed until the following conditions are satisfied:

   a. All operating registers are free.

   b. No SCM bank conflicts exist.

   c. LCM is not busy.

   d. All LCM banks have completed previously initiated read/write cycles.

3. A delay will occur during instructions 011, 012, and 013 when an I/O multiplexer request is made. A minimum delay of one clock period is required to enter the I/O word address in the address stream to the SAS. An additional delay will occur if the I/O reference causes a bank conflict in SCM.

4. A delay will occur in the execution of the Exchange Exit instruction (013) until two conditions are satisfied:

   a. All operating registers are free.

   b. No SCM bank conflicts exist.

5. The Read LCM and Write LCM instructions (014 and 015) will not issue until three conditions are satisfied:

   a. LCM is not busy.

   b. Xj register is free.

   c. Xk register is free.

6. A Read LCM instruction (014) for a word already residing in an LCM bank operand register as a result of a previous instruction will require three clock periods. For a word not currently residing in one of the LCM bank operand registers, the instruction requires 17 clock periods.

7. The Reset Buffer instructions and Read Channel Status instructions (016 and 017) will not issue and begin execution until the required B registers are free.

8. Jump instruction 02i0K will not begin execution until the Bi register is free. Instruction execution will also be delayed if an instruction fetch is in process.

9. The execution of a branch instruction (030 to 037, 04ijk, 05ijk, 06ijk, and 07ijk) may be delayed if an instruction fetch is in process.

10. Instructions 10 to 47 and 60 to 77 will not issue until the following conditions are satisfied:

    a. The required A, B, and X registers are free.

    b. X and B register input paths will be free during the required clock period.

    c. No SAS backup condition exists.

    d. The multiply unit is free (instructions 40, 41, and 42 only).

    e. The divide unit is free (instructions 44 and 45 only).

11. Instructions 50 to 57 will not issue until the following conditions are satisfied:

    a. The required A, B, and X registers are free.

    b. No SAS backup condition exists.

12. A delay may occur in the execution of the Return Jump instruction (0100K) if the instruction stack control has requested one or more instruction words that have not arrived at the instruction stack (likely to occur in straight line coding). Average execution time is 18 clock periods.

13. A register is reserved if it is the destination of an instruction that has been initiated but has not completed. A register is free in the clock period following the store into it.

6600/6700 CPU TIMING NOTES

1. The times given in Table A-1 are computational times - the time needed after the execution start until the result is computed and ready to be stored into the result register.

2. The functional units are not freed until one minor cycle after the result has been stored into the result register.

3. A result register value may be used as an operand to another instruction as soon as the result has been stored into the register (same minor cycle). This result register will not be freed to be used as a result register of another instruction until one cycle after the result has been stored into that register (no trunk priority considered).

4. An instruction is issued to a functional unit if:

   a. The word containing the instruction is in the stack and the U registers,

   b. The functional unit(s) needed are free, and

   c. The result register(s) needed are free (note Table A-2 and A-3).

   If these three conditions are not met, all further instruction issues are held until they are satisfied. Each issued 15-bit instruction requires one minor cycle before the next instruction is available for issue. Each issued 30-bit instruction requires two minor cycles before the next instruction is available for issue.

5. Execution within a functional unit does not start until the operands are available (note Table A-3). The two operands required are fetched from the registers at the same time (one operand is not loaded while the unit waits for a second operand).

6. In instructions 02-07, where more than one functional unit is used, the instruction is not issued until both functional units involved are free.

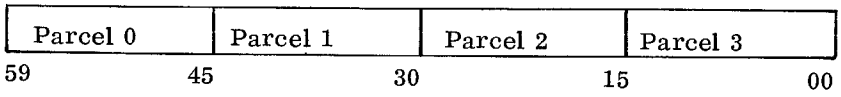
7. Times given for instructions 01-07 and 50-57 do not consider any memory conflict conditions.

8. In instructions 50-57, if i = 1,2...5 (load from memory instructions), the Xi register value is not available until 8 minor cycles after the start of the instruction execution (assuming no memory conflicts). When two load instructions begin execution one minor cycle apart, one extra minor cycle is required for execution of the later instruction. Therefore, the second executed instruction would require 9 cycles for the load, 5 cycles for the increment unit, and 4 cycles for the A register.

9. In instructions 50-57, if i = 6 or 7 (store to memory instructions), the Xi register is not available for a result register until 10 minor cycles after the instruction begins execution (assuming no memory conflicts). When two store instructions begin execution one minor cycle apart, one extra minor cycle is required for execution of the later instruction. Therefore, the second executed instruction would require 11 cycles for the store, 5 cycles for the increment unit, and 4 cycles for the A register.

† The 6700 also includes a 6400-type CPU.

10. When executing sequential instructions that are not in the stack, the minimum time is one word of instructions every 8 cycles. The time of issue of the last parcel of an instruction word to the time of issue of the frist parcel of the next instruction word (while executing sequential instructions that are not in the stack) requires a minimum of 4 cycles. If the last instruction in an instruction word is a 30-bit instruction, a minimum of 5 cycles is required from the time of issue to a functional unit of this instruction to the time of issue of the first instruction in the next word. An instruction word is parcelled as shown below:



11. When a branch is taken out of the stack, 15 minor cycles are normally required for a 03ijk instruction, and 14 minor cycles are normally required for other branch instructions (considering no memory conflict). The latter timing is from the start of branch instruction execution to the point where the instruction at the branch address is ready for issue to a functional unit.

12. Nine cycles are required for 03ijk instructions when the branch is taken within the stack. The next sequential word is recognized as being within the stack.

13. Eight cycles are required for 04ijk to 07ijk instructions when the branch is taken within the stack. The next sequential word is recognized as being within the stack.

14. Eleven cycles are required for 03ijk instructions when the branch is not taken (time from branch execution to issue of next instruction) if in the stack or if falling through to the same word. Out of the stack fall-through to the next word takes 14 cycles.

15. Ten cycles are required for 04ijk to 07ijk instructions when the branch is not taken (time from branch execution to issue of next instruction) if in the stack or if falling through to the same word. Out of the stack fall-through to the next word takes 13 cycles.

16. The B0 register is handled as any other Bi register for timing purposes (e.g., B0 delays execution of an instruction if it is a result register of a previous non-completed instruction).

17. Neither increment unit may be involved in a load operation if a store operation is to be issued, and neither increment unit may be involved in a store operation if a load operation is to be issued. The sequential loading of instruction words does not affect the load/store conditions of the increment units. Increments of A0 are considered neither loads nor stores.

18. The operand registers are available to more than one functional unit in the same minor cycles if the units are in different groups.

Group 1	Group 2	Group 3
Boolean	Shift	Increment 1
Divide	Floating Add	Increment 2
Multiply 1	Long Add	
Multiply 2		

19. The time needed for a functional unit to operate on indefinite, out-of-range, or zero values is the same as for normal, in-range values (i.e., no gain or loss in execution time due to a unit recognizing an indefinite operand and setting an indefinite result).

20. An Index Jump instruction (02) always voids the stack. If an unconditional jump back in the stack is desired, a 0400K instruction may be used (to save memory access time for instructions).

21. A Return Jump instruction (01) always voids the stack.

22. After a result has been computed by a functional unit, the result register is checked to see if it is still needed as an operand register for a previously issued instruction. This is done so that a result will not overlay an operand to a previously issued instruction. If a unit (#1) is waiting for an operand to be fetched by another unit (#2) before storing its result, for timing considerations,

    a. the result register is available to a third unit (#3) as an operand, the cycle following the fetch, and

    b. the unit (#1) is freed two cycles following the fetch.

23. In cases of bank conflict, unaccepted addresses get a chance at access every three minor cycles. If the address can then be accessed, the memory operation proceeds. If the bank is still busy, the address circulates in the hopper, while access is permitted for any other source requesting access.

6000-SERIES PPU TIMING NOTES

The execution time of peripheral and control processor instructions (Table A-4) is influenced by the following factors:

1. Number of memory references. Indirect addressing and indexed addressing require an extra memory reference. Instructions in 24-bit format require an extra reference to read m.

2. Number of words to be transferred. In I/O instructions and in references to central memory the execution times vary with the number of words to be transferred. The maximum theoretical rate of flow is one word/major cycle. I/O word rates depend upon the speed of external equipments which are normally much slower than the computer.

3. References to central memory may be delayed if there is conflict with central processor memory requests.

4. Following an exchange jump instruction, no memory references (nor other exchange jump instructions) may be made until the central processor has completed the exchange jump.

7600 PPU TIMING NOTES

1. Where more than one time is given, the shorter time is obtained when full use of bank phasing (back-to-back storage references to alternate banks) is made.

2. Conditional jump instructions list times for the jump not taken case. Add 3 or 5 clock periods for the jump taken case, depending on the value of d.

3. For the 10 (shift) instruction: Minimum time required if the shift count ≤ 3; for shift counts > 3, add 1 clock period per shift beyond 3 to the minimum time.

4. 71 Instruction:

   Case 1: Assume - a. a block input terminated by a record flag rather than by decrementing (A) to zero.
   b. a 2-clock period response time between the resume and the word flag.
   c. a 3-word block followed by a record flag.
   d. the channel d input word flag is set at instruction initiation, and
   e. the first data reference is to the alternate storage bank.

   Execution Time = 42 Clock Periods

   Case 2: Assume - a. a block input terminated by reducing (A) to zero.
   b. same response as in Item b, Case 1.
   c. a count of 2 in the A register, and
   d. items d and e in Case 1 are true.

   Execution Time = 24 Clock Periods

   Case 3: Assume - a. a block input initiated with (A) = zero.

   Execution Time = 10 Clock Periods

5. 73 Instruction:

   Case 1: Assume - a. a count of 3 in the A register.
   b. the device has a 2-clock period response time from receipt of word pulse to transmission of resume pulse.
   c. the output channel d word flag is clear, and
   d. the first word of the block is read from the alternate storage bank.

   Execution Time = 34 Clock Periods

   Case 2: Assume - a. a block output initiated with (A) = zero.

   Execution Time = 10 Clock Periods

TABLE A-1. CENTRAL PROCESSOR INSTRUCTION/TIMES

INSTRUCTION CODE	NAME	6600 FUNCTIONAL UNIT	7600 FUNCTIONAL UNIT	64/6500 EXECUTION TIME (MINOR CYCLES)	6600 EXECUTION TIME (MINOR CYCLES)	7600 EXECUTION TIME (CLOCK PERIODS)
00000	Stop	Branch	-	-	-	-
00000	Error exit to EEA	-	-	-	-	-
0100K	Return jump to K	Branch	-	21	13	Min 13†
011jK	Block copy K + (Bj) words from ECS to CM	ECS	-	†††	†††	-
011ijK	Block copy K + (Bj) words from LCM to SCM	-	-	-	-	Min = N + 15††
012jK	Block copy K + (Bj) words from CM to ECS	ECS	-	†††	†††	
012jK	Block copy K + (Bj) words from SCM to LCM	-	-	-	-	Min = N + 11††
01300	Exchange exit to NEA if exit flag clear	-	-	-	-	Min = 28
013jK	Exchange exit to (Bj) + K or MA	-	-	-	-	-
013jK	Exchange exit to K + (Bj) if exit flag set	-	-	-	-	Min = 28
014jk	Read LCM at (Xk) to Xj	-	-	-	-	3, 17†
015jk	Write (Xj) into LCM at (Xk)	-	-	-	-	3
0160k	Reset input channel (Bk) buffer if j = 0	-	-	-	-	4
016jk	Read input channel (Bk) status to Bj if j ≠ 0	-	-	-	-	3
0170k	Reset output channel (Bk) buffer if if j = 0	-	-	-	-	16
017jk	Read output channel (Bk) status to Bj if j ≠ 0	-	-	-	-	3
02i0K	Jump to K + (Bi)	Branch made in Increment Unit)	-	13	Min 14 (in stack jump)† Min 20 (out of stack jump)†	Min 3 (in stack jump)
030jK	Branch to K if (Xj) = 0	Branch (Test Made in Long Add Unit)	-	Min 5 (branch fall through) Min 13 (branch)	Min 11 (branch fall through in stack)† Min 9 (branch in stack)† Min 15 (branch out of stack)† Min 14 (branch fall through out of stack)	Min 2 (branch fall through) Min 3 (branch in stack) Min 11 (branch out of stack)
031jK	Branch to K if (Xj) ≠ 0	Branch	-			
032jK	Branch to K if (Xj) positive	Branch	-		Same as above	Same as above
033jK	Branch to K if (Xj) negative	Branch	-			
034jK	Branch to K if (Xj) in range	Branch	-			

† Refer to Timing Notes.

†† 1 ≤ N = Number of words in block. (4 clock periods if N = 0)

††† Execution times for Extended Core Storage operations depend upon several factors; refer to Extended Core Storage Reference Manual for timing information.

TABLE A-1. CENTRAL PROCESSOR INSTRUCTION/TIMES (cont'd)

INSTRUCTION CODE	NAME	6600 FUNCTIONAL UNIT	7600 FUNCTIONAL UNIT	64/6500 EXECUTION TIME (MINOR CYCLES)	6600 EXECUTION TIME (MINOR CYCLES)	7600 EXECUTION TIME (CLOCK PERIODS)
035jK	Branch to K if (Xj) out of range	Branch (Test made in Long Add Unit)	-	Min 5 (branch fall through) Min 13 (branch)	Min 11 (branch fall through in stack) † Min 9 (branch in stack) † Min 15 (branch out of stack) † Min 14 (branch fall through out of stack) †	Min 2 (branch fall through) Min 3 (branch in stack) Min 11 (branch out of stack)
036jK	Branch to K if (Xj) definite	Branch	-	Same as above	Same as above	Same as above
037jK	Branch to K if (Xj) indefinite	Branch	-			
04ijK	Branch to K if (Bi) = (Bj)	Branch	-		Min 10 (branch fall through) Min 13 (branch in stack Min 14 (branch out out of stack)	
05ijK	Branch to K if (Bi) ≠ (Bj)	Branch (Test in Increment Unit)	-		Same as above	
06ijK	Branch to K if (Bi) ≥ (Bj)	Branch	-		Same as above	
07ijk	Branch to K if (Bi) < (Bj)	Branch	-			
10ij0	Copy (Xj) to Xi	Boolean	Boolean	5	3	2
11ijk	Logical product of (Xj) and (Xk) to Xi	Boolean	Boolean	5	3	2
12ijk	Logical sum of (Xj) plus (Xk) to Xi	Boolean	Boolean	5	3	2
13ijk	Logical difference of (Xj) minus (Xk) to Xi	Boolean	Boolean	5	3	2
14i0k	Copy complement of (Xk) to Xi	Boolean	Boolean	5	3	2
15ijk	Logical product of (Xj) and comp (Xk) to Xi	Boolean	Boolean	5	3	2
16ijk	Logical sum (Xj) plus comp (Xk) to Xi	Boolean	Boolean	5	3	2
17ijk	Logical difference of (Xj) minus comp (Xk) to Xi	Boolean	Boolean	5	3	2
20ijk	Left shift (Xi) by jk	Shift	Shift	6	3	
21ijk	Right shift (Xi) by jk	Shift	Shift	6	3	2
22ijk	Left shift (Xk) by (Bj) to Xi	Shift	Shift	6	3	2

† Refer to Timing Notes.

TABLE A-1. CENTRAL PROCESSOR INSTRUCTION/TIMES (cont'd)

INSTRUCTION CODE	NAME	6600 FUNCTIONAL UNIT	7600 FUNCTIONAL UNIT	64/6500 EXECUTION TIME (MINOR CYCLES)	6600 EXECUTION TIME (MINOR CYCLES)	7600 EXECUTION TIME (CLOCK PERIODS)
23ijk	Right shift (Xk) by (Bj) to Xi	Shift	Shift	6	3	2
24ijk	Normalize (Xk) to Xi and Bj	Shift	Normalize	7	4	3
25ijk	Round and normalize (Xk) to Xi and Bj	Shift	Normalize	7	4	3
26ijk	Unpack (Xk) to Xi and Bj	Shift	Boolean	7	3	2
27ijk	Pack (Xk) and (Bj) to Xi	Shift	Boolean	7	3	2
30ijk	Floating sum of (Xj) plus (Xk) to Xi	Floating Add*	Floating Add	11	4	4
31ijk	Floating difference of (Xj) minus (Xk) to Xi	Floating Add*	Floating Add	11	4	4
32ijk	Floating DP sum of (Xj) plus (Xk) to Xi	Floating Add*	Floating Add	11	4	4
33ijk	Floating DP difference of (Xj) minus (Xk) to Xi	Floating Add*	Floating Add	11	4	4
34ijk	Round floating sum of (Xj) plus (Xk) to Xi	Floating Add*	Floating Add	11	4	4
35ijk	Round floating difference of (Xj) minus (Xk) to Xi	Floating Add*	Floating Add	11	4	4
36ijk	Integer sum of (Xj) plus (Xk) to Xi	Long Add	Long Add	6	3	2
37ijk	Integer difference of (Xj) minus (Xk) to Xi	Long Add	Long Add	6	3	2
40ijk	Floating product of (Xj) times (Xk) to Xi	Floating Multiply	Floating Multiply	57	10	5
41ijk	Round floating product of (Xj) times (Xk) to Xi	Floating Multiply	Floating Multiply	57	10	5
42ijk	Floating DP product of (Xj) times (Xk) to Xi	Floating Multiply	Floating Multiply	57	10	5
43ijk	Form mask of jk bits to Xi	Shift	Shift	6	3	2
44ijk	Floating divide (Xj) by (Xk) to Xi	Floating Divide	Floating Divide	57	29	20
45ijk	Round floating divide (Xj) by (Xk) to Xi	Floating Divide	Floating Divide	57	29	20
46000	Pass	-	-	3	1	1

* Duplexed units - instruction goes to free unit

TABLE A-1. CENTRAL PROCESSOR INSTRUCTION/TIMES (cont'd)

INSTRUCTION CODE	NAME	6600 FUNCTIONAL UNIT	7600 FUNCTIONAL UNIT	64/6500 EXECUTION TIME (MINOR CYCLES)	6600 EXECUTION TIME (MINOR CYCLES)	7600 EXECUTION TIME (CLOCK PERIODS)
47i01	Population count of (Xk) to Xi	Floating Divide	Population Count	68	8	2
50ijK	Increment (Aj) plus K to Ai	Increment	Increment	6 (i=0) 12 (i=1-5) 10 (i=6,7)	3	2 (Set Aj) Min 8 (Read to Xi) 2 (Store from Xi)
51ijK	Increment (Bj) plus K to Ai	Increment*	Increment		3	
52ijK	Increment (Xj) plus K to Ai	Increment*	Increment		3	
53ijk	Increment (Xj) plus (Bk) to Ai	Increment*	Increment		3	
54ijk	Increment (Aj) plus (Bk) to Ai	Increment*	Increment	Same as above	3	same as above
55ijk	Increment (Aj) minus (Bk) to Ai	Increment*	Increment			
56ijk	Increment (Bj) plus (Bk) to Ai	Increment*	Increment		3	
57ijk	Increment (Bj) minus (Bk) to Ai	Increment*	Increment		3	
60ijK	Increment (Aj) plus K to Bi	Increment*	Increment	5	3	2
61ijK	Increment (Bj) plus K to Bi	Increment*	Increment	5	3	2
62ijK	Increment (Xj) plus K to Bi	Increment*	Increment	5	3	2
63ijk	Increment (Xj) plus (Bk) to Bi	Increment*	Increment	5	3	2
64ijk	Increment (Aj) plus (Bk) to Bi	Increment*	Increment	5	3	2
65ijk	Increment (Aj) minus (Bk) to Bi	Increment*	Increment	5	3	2
66ijk	Increment (Bj) plus (Bk) to Bi	Increment*	Increment	5	3	2
67ijk	Increment (Bj) minus (Bk) to Bi	Increment*	Increment	5	3	2
70ijK	Increment (Aj) plus K to Xi	Increment*	Increment	6	3	2
71ijK	Increment (Bj) plus K to Xi	Increment*	Increment	6	3	2
72ijK	Increment (Xj) plus K to Xi	Increment*	Increment	6	3	2
73ijk	Increment (Xj) plus (Bk) to Xi	Increment*	Increment	6	3	2
74ijk	Increment (Aj) plus (Bk) to Xi	Increment*	Increment	6	3	2
75ijk	Increment (Aj) minus (Bk) to Xi	Increment*	Increment	6	3	2
76ijk	Increment (Bj) plus (Bk) to Xi	Increment*	Increment	6	3	2
77ijk	Increment (Bj) minus (Bk) to Xi	Increment*	Increment	6	3	2

* Duplexed units - instruction goes to free unit

TABLE A-2. FUNCTIONAL UNIT DATA TRUNK ASSIGNMENTS AND PRIORITY

FUNCTIONAL UNIT		RESULT (i)		OPERAND (j)		OPERAND (k)	
		Trunk	Priority	Trunk	Priority	Trunk	Priority
Group 1:	Shift	3 (X), 4 (B) †	1	1	2	2	2
	Add	3	2	1	1	2	1
	Long Add	3	3	1	3	2	3
Group 2:	Boolean	7	1	5	4	6	4
	Divide	7	2	5	1	6	1
	Multiply 1	7	3	5	2	6	2
	Multiply 2	7	4	5	3	6	3
Group 3:	Increment 1	10	1	8	1	9	1
	Increment 2	10	2	8	2	9	2

† The Shift Unit is sometimes required to store two results at one time: one into an X register and one into a B register.

TABLE A-3. 6600/6700 REGISTER RESERVATION CONTROL

INSTRUCTION	XBA RESULT REGISTER (ISSUE)	Q OPERAND REGISTER (EXECUTION)
Branch Unit		
02ijK	-	Bi & Bj
03ijK	-	Xi & Xj
04ijK	-	Bi & Bj
Boolean Unit		
10ijk - 17ijk	Xi	Xj & Xk
Shift Unit		
20ijk - 23ijk	Xi	Bj & Xk
24ijk - 26ijk	Xi & Bj	Bj & Xk
27ijk & 43ijk	Xi	Bj & Xk
Add Unit (Floating)		
30ijk - 35ijk	Xi	Xj & Xk
Long Add (Integer)		
36ijk - 37ijk	Xi	Xj & Xk
Multiply (2 Units)		
40ijk - 42ijk	Xi	Xj & Xk
Divide Unit		
44ijk - 47ijk	Xi	Xj & Xk
Increment (2 Units)		
50ijK	Ai & Xi†	Aj & Bk††
51ijK	Ai & Xi†	Bj & Bk††
52ijK	Ai & Xi†	Xj & Bk††
53ijk	Ai & Xi†	Xj & Bk
54ijk & 55ijk	Ai & Xi†	Aj & Bk
56ijk & 57ijk	Ai & Xi†	Bj & Bk
60ijK	Bi	Aj & Bk††
61ijK	Bi	Bj & Bk††
62ijK	Bi	Xj & Bk††
63ijk	Bi	Xj & Bk
64ijk & 65ijk	Bi	Aj & Bk
66ijk & 67ijk	Bi	Bj & Bk
70ijK	Xi	Aj & Bk††
71ijK	Xi	Bj & Bk††
72ijK	Xi	Xj & Bk††
73ijk	Xi	Xj & Bk
74ijk & 75ijk	Xi	Aj & Bk
76ijk & 77ijk	Xi	Bj & Bk

† The Xi register is considered only when i = 1, 2...7.

†† k here refers to the high order 3 bits of 18-bit address field.

TABLE A-4. PERIPHERAL PROCESSOR INSTRUCTION TIMES

INSTRUCTION CODE (OCTAL)	NAME	7600 EXECUTION TIME (CLOCK PERIODS)	6000-SERIES TIME (MAJOR CYCLES)
00	Error stop	-	-
0100	Long jump to m	10 or 15	2 or 3
01xx	Long jump to m + (d)	15, 20, 25	
0200	Return jump to m	15 or 20	3-4
02xx	Return jump to m + (d)	20, 25, 30	
03	Unconditional jump d	8, 10	1
04	Zero jump d	5	1
05	Nonzero jump d	5	1
06	Positive jump d	5	1
07	Negative jump d	5	1
10	Shift d	Min 6, Max 34	1
11	Logical difference (A) - d	5	1
12	Logical product (A) * d	5	1
13	Selective clear (A)	5	1
14	Load to A	5	1
15	Load complement d to A	5	1
16	Add d to A	5	1
17	Subtract (A) - d to A	5	1
20	Load dm to A	10	2
21	Add dm to A	10	2
22	Logical product (A) * dm to A	10	2
23	Logical difference (A)-c to A	10	2
24	Pass	5	1
25	Pass	5	1
26	Pass	5	1
260	Exchange jump	-	1†
260	6416 Extended transfer	-	
261	Monitor exchange jump CPU d	-	
27	Pass	5	1
270	Read program address of CPU d	-	
270	6416 Extended read status	-	
30	Load (d) to A	15	2
31	Add (d) to A	15	2
32	Subtract (A)-(d) to A	15	2
33	Logical difference (A)-(d) to A	15	2
34	Store (A) to d	15	2
35	Replace add (d) + (A) to d	25	3
36	Replace add one (d) + 1 to d	25	3
37	Replace subtract one (d) -1 to d	25	3
40	Load ((d)) to A	15, 25	3
41	Add ((d)) + (A) to A	15, 25	3
42	Subtract (A) - ((d)) to A	15, 25	3
43	Logical difference (A) - ((d)) to A	15, 25	3
44	Store (A) to (d)	15, 25	3
45	Replace add (A) to ((d)) to (d)	25, 35	4
46	Replace add one ((d)) + 1 to (d)	25, 35	4
47	Replace subtract one ((d)) -1 to (d)	25, 35	4
5000	Load (m) to A	20	3
50xx	Load (m + (d)) + (A) to A	20, 30	3, 4
5100	Add (m)	20	3
51xx	Add (m + (d))	20, 30	3, 4
5200	Subtract (m)	20	3
52xx	Subtract (m + (d))	20, 30	3, 4
5300	Logical difference (m)	20	3

† Though the execution time for this instruction in the Peripheral and Control Processor is only 1 major cycle, a minimum of 2 major cycles is required to complete the Exchange operation in Central Memory. Thus, Central Memory honors no requests for access for a minimum of 2 major cycles during an Exchange Jump.

TABLE A-4. PERIPHERAL PROCESSOR INSTRUCTION TIMES (cont'd)

INSTRUCTION CODE (OCTAL)	NAME	7600 EXECUTION TIME (CLOCK PERIODS)	6000 SERIES TIME (MAJOR CYCLES
53xx	Logical difference (m + (d))	20,30	3,4
5400	Store (m)	20	3
54xx	Store (m + (d))	20,30	3,4
5500	Replace add (m)	30	4
55xx	Replace add (m + (d))	30,40	4,5
5600	Replace add one (m)	30	4
56xx	Replace add one (m + (d))	30,40	4,5
5700	Replace subtract one (m)	30	4
57xx	Replace subtract one (m + (d))	30,40	4,5
60	Central read (A) to d	-	Min 6
60	Jump on input word flag	10††	-
61	Central read (d) words (A) to m	-	5 + 5/word
61	Jump if no input word flag	10	-
62	Central write (A) words	-	Min 6
62	Jump on input record flag	10	-
63	Central write (d) words to (A) from m	-	5 + 5/word
63	Jump if no input record flag	10	-
64	Jump to m if channel d active	-	2
64	Jump on output word flag	10	-
65	Jump to m if channel d inactive	-	2
65	Jump if no output word flag	10	-
66	Jump to m if channel d full	-	2
66	Jump on output record flag	10	-
67	Jump to m if channel d empty	-	2
67	Jump if no output record flag	10	-
70	Input to A from channel d	9†	2
71	Input (A) words to m from channel d	†††	4 + 1/word
72	Output from A on channel d	9††	2
73	Output (A) words from m on channel d	†††	4 + 1/word
74	Activate channel d	-	2
74	Output record flag on channel d	5	-
75	Disconnect channel d	-	2
75	Pass	5	-
76	Function (A) on channel d	-	2
76	Pass	5	-
77	Function m on channel d	-	2
77	Error stop	(restart only by a deadstart)	-

† Assume input channel d word flag is set; if not set, add the time waiting for flag to set.

†† Jump instruction times are for the jump not taken case. The jump taken execution time is identical if the jump is to an alternate bank. If the jump is taken to the same bank, add 5 clock periods.

††† Timing for these instructions are sample times only for various cases.

†††† Assumes output channel d word flag is clear; if not clear, add the time waiting for flag to clear.

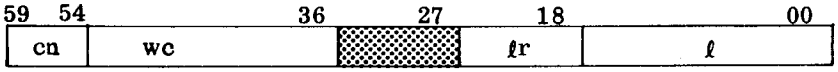
BINARY FORMATS B

This appendix describes the various binary formats that can be generated by the COMPASS assembler †. The types of binary formats are as follows:

CPU relocatable
CPU absolute
7600 PPU absolute
6000 Series PPU absolute

RELOCATABLE SUBPROGRAM

Output for a relocatable subprogram consists of a logical record composed of an indefinite number of tables. Each table is preceded by an identification word of the following form:



ID Word Format

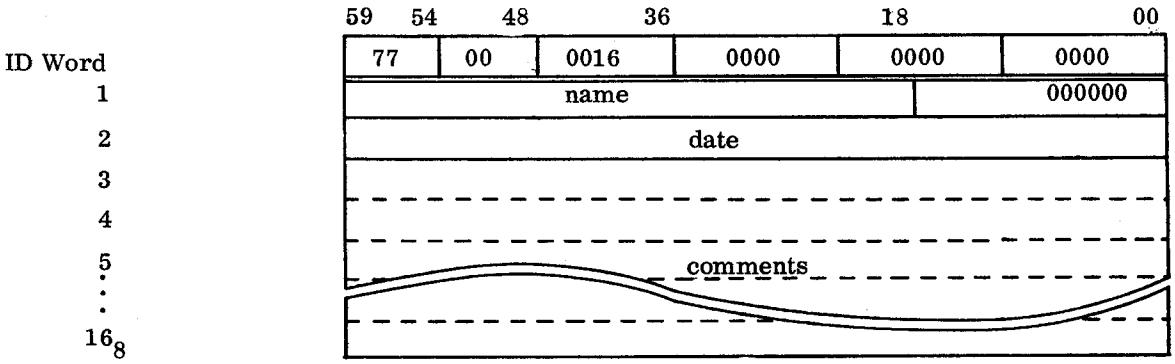
Bits	Field		Description
59-54	cn	Octal code	Number identifying table type as follows:
		77	Prefix table
		34	Program identification and length table (PIDL)
		36	Entry point table (ENTR)
		40	Text and data table (TEXT)
		42	Fill common area table (FILL)
		44	External reference table (LINK)
		43	Replication table (REPL)
		46	Transfer table (XFER)
53-36	wc	Octal count of number of 60-bit words in the table, excluding the identification word.	
35-27	none	Reserved for future system use.	
26-18	ℓr	In TEXT table, indicates method of relocation for the load address; otherwise, the field is ignored by the loader.	
17-00	ℓ	In TEXT table, indicates beginning location for data in the table; in the REPL table, this field indicates immediate or deferred replication. For all other tables, the field is zero and is ignored by the loader.	

†For SCOPE 2, 7600 COMPASS Version 2 generates the tables described here when the E or E=0 option is selected on the COMPASS control statement. Extended tables are described in the Loader Reference Manual, Publication No. 60344200.

Prefix Table (prefix)

The prefix table is described here for all binary formats. Generation of the table can be suppressed through use of the NOLABEL pseudo instruction.

Prefix Table:

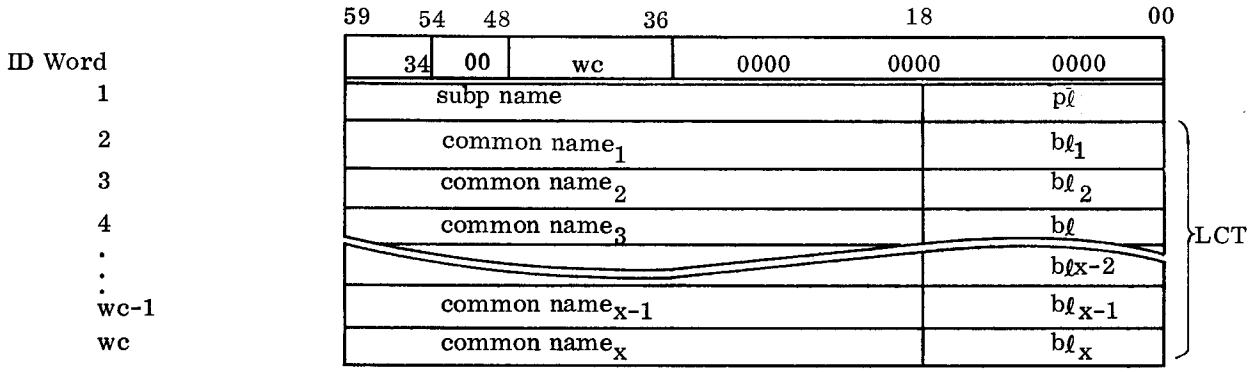


Word	Bits	Field	Description
1	59-18	name	Name of record in display code from IDENT or SEGMENT pseudo instruction left justified with zero fill.
	17-00		Reserved for future system use.
2	59-00	date	Date of assembly in the form Δ yr/mo/dy. where month (mo), day (dy), and year (yr) are each two display code digits.
$3\text{-}16_8$		comments	Display code text from COMMENT pseudo instruction, left justified with zero fill.

Program Identification and Length Table (PIDL)

The PIDL table contains the name and length of the subprogram block and each of the common blocks. The only table that can precede it is the PREFIX table. The entries starting with the third word comprise a table within the PIDL, called the Local Common Table (LCT). This embedded table lists labeled common and blank common blocks in the order in which they are established in the subprogram. Relocation of addresses in subsequent loader tables is relative to common blocks according to the position of the block name in the LCT. The first word in the LCT is position 1.

PIDL Table:

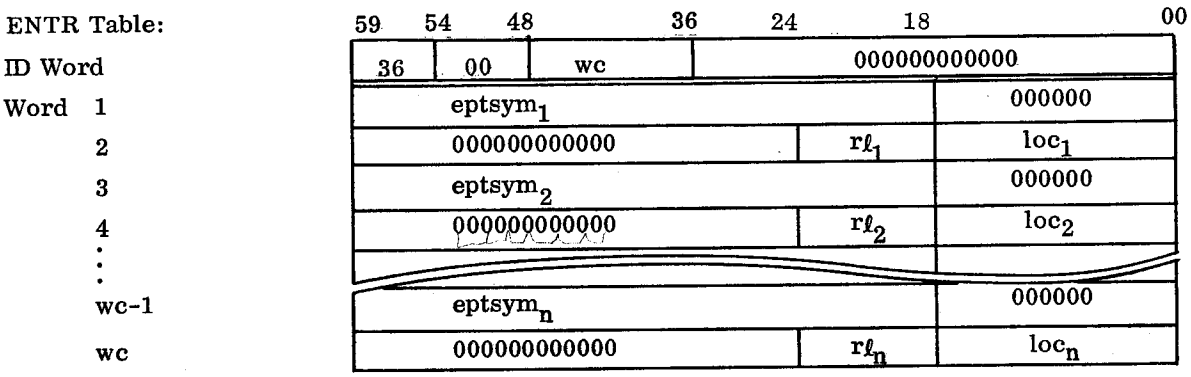


Word	Bits	Field	Description
ID	53-36	wc	Length of LCT plus 1, in octal
1	59-18	subp name	Name of subprogram in display code as taken from IDENT pseudo instruction, left justified with zero fill.
	17-00	pℓ	Length of subprogram including all local blocks with exception of absolute block if there is one.
2-wc	59-18	common name$_i$	Name of common block as declared on USE pseudo instruction. For blank common, name is 7 display code blank characters.
	17-00	bℓ$_i$	Length of the common block, in octal. For COMPASS 2 under SCOPE 2, blocks declared by USELCM have bit 17 set to one and bits 16-00 contain the integer part of (block length + 7) / 8. For all other systems, blocks declared by USELCM are not recorded in the PIDL table.

Entry Point Table (ENTR)

An entry point table lists each entry point symbol to the subprogram as declared on an ENTRY pseudo instruction, and the labeled common block containing the entry point symbol. ENTR table must immediately follow the PIDL table. Each entry is two words.

ENTR Table:



Word	Bits	Field	Description
1,3, ..., wc-1	59-18	$eptsym_i$	Name of entry point symbol in display code specified on ENTRY pseudo instruction, left justified with zero fill.
	17-00	none	Reserved for future use.
2,4 ..., wc	59-27	none	Reserved for future use.
	26-18	$r\ell_i$	Relocation for address loc: 000 Absolute relative to RAS (no relocation) 001 Program relocation 003-077 Relative to the common block named in position rℓ-2 in the LCT. It cannot be the number for the blank common block.
	17-00	loc_i	Address of entry point relative to block origin.

Text and Data Table (TEXT)

A TEXT table contains an origin for data, indicators for relocating the data in the table, and data. The subprogram can contain any number of TEXT tables which can be in any order.

TEXT Table:

ID Word

Word 1

2

3

4

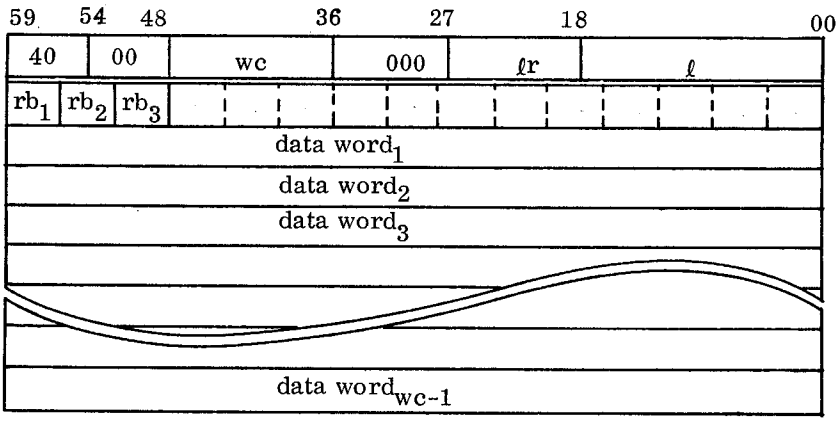
.

.

.

wc-1

wc



Word	Bits	Field	Description
ID	53-36	wc	Number of data words in the table plus one; wc must be in the range $2\text{-}20_8$.
	26-18	ℓr	Relocation of load address 1.
			000 Absolute relative to RAS (no relocation)
			001 Relative to subprogram origin
			003-077 Relative to common block named in position rℓ-2 in the LCT. It cannot be the number for the blank common block.
	17-00	ℓ	Beginning load address relative to block origin for data beginning in $Word_2$. The load address is relocated according to rℓ.
1	59-56 55-52, ..., 03-00	rb_i	Up to 15 4-bit relocation bytes describing address relocation according to the three possible positions of addresses in a 60-bit word. The first byte (bits 59-56) describes the relocation for the first data word (Word 2). The second byte describes the relocation for the second data word (Word 3), etc. The bytes permit independent and simultaneous relocation of both upper and lower addresses. Relocation is relative to program origin (positive) or to the complement of program origin (negative) as follows:

Byte	Significance
000x	No address relocation
10xx	Upper address, program relocation
11xx	Upper address, negative relocation
010x	Middle address, program relocation
011x	Middle address, negative relocation
1x10	Lower address, program relocation
1x11	Lower address, negative relocation
0010	Lower address, program relocation
0011	Lower address, negative relocation

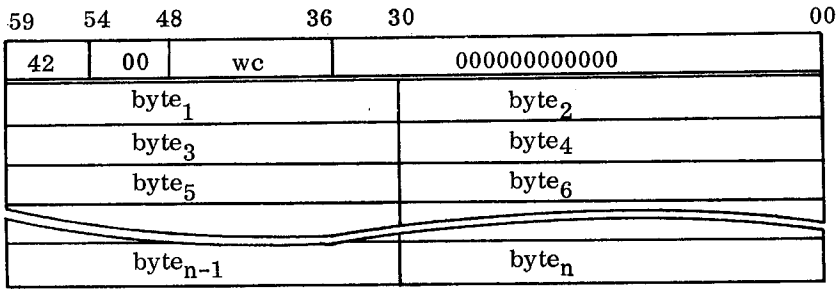
Note that an address is either not relocated (absolute) or it is relocated relative to program origin. It cannot be relocated relative to a labeled common block. Thus, TEXT tables cannot be used to relocate addresses relative to labeled common. This must be accomplished through FILL tables.

2-wc 59-00 data word$_i$ Data words to be stored beginning at absolute address or relocated address indicated in the ID word.

Fill Common Area Table (FILL)

The FILL table specifies relocation of addresses in words already loaded through TEXT tables. References to common blocks are relocated through this table. Although program relocation is also possible through use of the FILL table, the usual method requiring fewer words is through the TEXT table.

FILL Table:



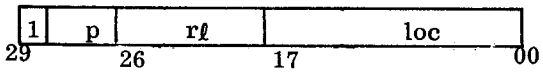
Words 1 through wc are divided into two types of 30-bit bytes, control bytes and data bytes.

Control Bytes - Each control byte (indicated by a 0 in bit 29) specifies the relocation of addresses in the data bytes following it until the next control byte. A control byte has the following format:

ar Relocation of addresses in words already loaded.

000	Absolute relative to RAS (no relocation)
001	Positive program relocation relative to program origin.
002	Negative program relocation relative to program origin.
003-077	Relative to common block named in position ar-2 in the LCT.

Data Bytes - Each data byte (indicated by a 1 in bit 29) identifies a word containing a reference address that requires relocation and specifies where the word containing the reference is located. The format of a data byte is as follows:

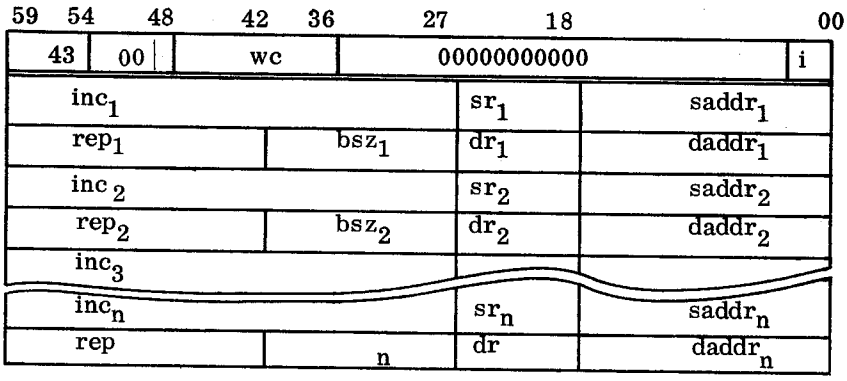


Bits	Field	Description
28,27	p	Position within word of address to be relocated 10_2 Upper 01_2 Middle 00 Lower
26-18	rℓ	Relocation of address loc: 000 Absolute relative to RAS 001 Program relocation relative to program origin 003-077 Relative to block named at position rℓ-2 in the LCT.
17-00	loc	Address of word containing the address to be modified. The address is modified by adding the relocated origin for the block specified by ar. A word that contains two addresses requires two data bytes.

Replication Table (REPL)

The REPL table directs the loader to generate one or more copies of data immediately or at the end of loading so that fewer TEXT tables are required. Each entry contains two words of information obtained from a REP or a REPI pseudo instruction or resulting from 5 or more BSSZ instructions.

REPL Table:

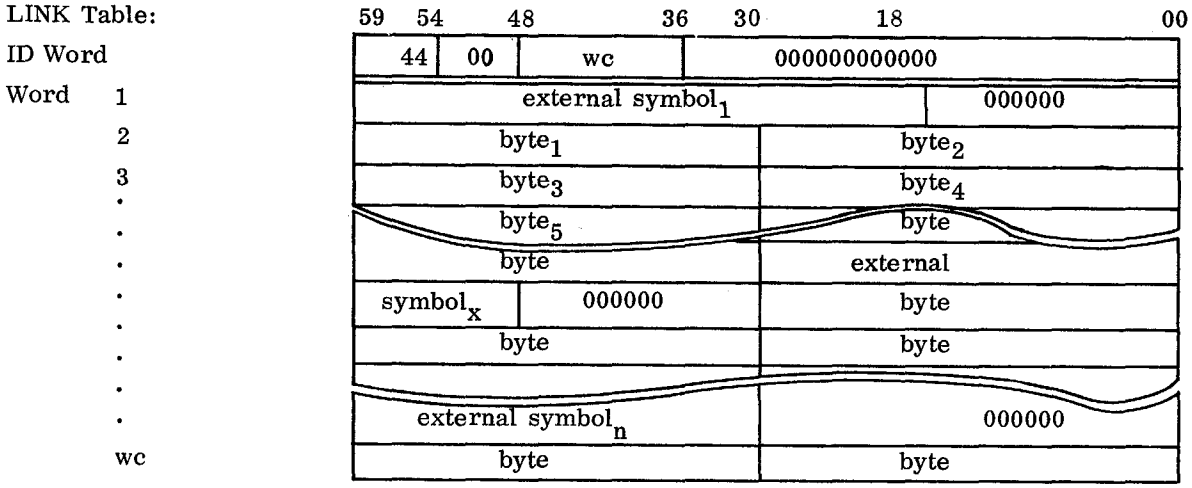


Word	Bits	Field	Description
ID	00	i	Indicates whether text is to be duplicated immediately (1) or is to be deferred until all text is loaded (0).
1,3,5,...,wc-1	59-27	inc_i	Number of words in each copy of the text. If inc_i is 0, the loader uses bsz_i as the increment size. The loader writes the first copy starting at $daddr_i$, the second starting at $daddr_i + inc_i$, the third at $bsz_i + 2\ inc_i$, etc. until the rep_i count is exhausted.
	26-18	sr_i	Relocation of $saddr_i$ 000 Absolute relative to RAS 001 Relative to program origin 003-077 Relative to block named in position sr-2 of the LCT.
	17-00	$saddr_i$	First word address of source data (text to be copied); must be nonzero.
2,4,6,...,wc	59-42	rep_i	Number of times text is to be copied. When rep_i is 0, the loader makes one copy.
	41-27	bsz_i	Number of words to be copied (block size). When bsz_i is 0, the loader copies one word.
	26-18	dr_i	Relocation of $daddr_i$; range of values same as sr.
	17-00	$daddr_i$	Destination address of first word of first copy. If $daddr_i$ is 0, the loader uses $saddr_i + bsz_i$.

External Reference Table (LINK)

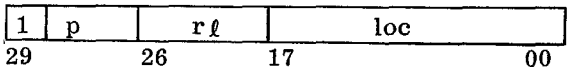
The LINK table contains an entry for each external symbol reference declared in the subprogram. Each entry consists of a 60-bit field for the symbol followed by an indefinite number of 30-bit contiguous data bytes.

LINK Table:



It is possible for a symbol to be split between two words. However, if the upper half of a word is a data byte and the lower half is all zeros, the loader takes the zeros as filler rather than the first half of an external symbol; this may be used to avoid having an external symbol split between two words. Each symbol must begin with a character for which the display code representation is not greater than 37 (has a high order bit of 0). The 1-7 character symbol is placed in the 60-bit field left justified with zero fill.

Each byte (indicated by a 1 in bit 29) identifies a word containing a reference to the external symbol and specifies the position at which the address for the symbol is to be inserted. The format of a data byte is as follows:

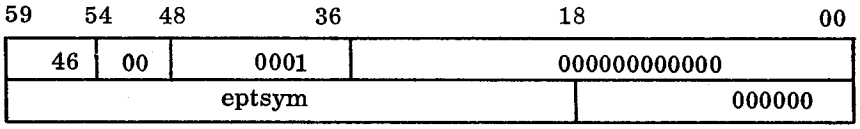


Bits	Field	Description
28,27	p	Position within word of external symbol reference
		10_2 Upper
		01_2 Middle
		00 Lower
26-18	rℓ	Relocation of address loc:
		000 Absolute relative to RAS
		001 Relative to subprogram origin
		003-077 Relative to block named at position rℓ-2 in the LCT.
17-00	loc	Address of word containing the external reference. The address of the external symbol is inserted at the position indicated.

Transfer Table (XFER)

The last table in a relocatable subprogram is the XFER table.

XFER Table:



eptsym — symbol to which control transfers when loading is complete. It is a 1-7 character entry point symbol in display code, left justified with zero fill. It need not be in the same subprogram as the XFER table. At least one subprogram of a program must name a transfer point; otherwise, SCOPE aborts the job with the comment NO TRANSFER ADDRESS. If more than one of the subprograms has a transfer point, the loader gives control to the last one encountered.

The location of the entry point is returned following a loader request.

If the first character of eptsym is blank (55_8) the XFER table is ignored.

CPU ABSOLUTE SUBPROGRAM OR OVERLAY

The binary output for an absolute CPU subprogram or overlay consists of a logical record that may contain:

A prefix table (optionally suppressed through NOLABEL)

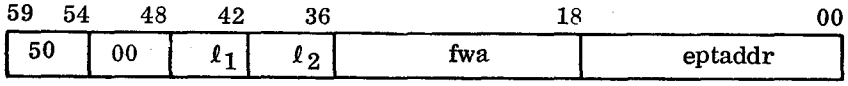
A 50_8 or 51_8 table (optionally suppressed through NOLABEL)

An absolute image of text.

For a description of the prefix table, refer to the Relocatable Subprogram description.

Single Entry Point Table (50)

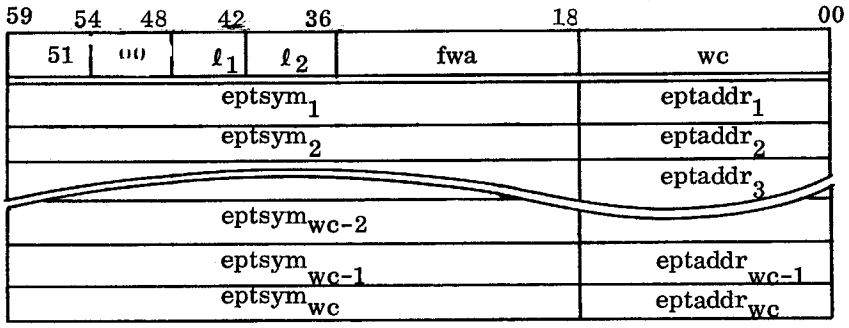
Following the prefix table but preceding the absolute text for a subprogram or overlay containing a single entry point is a control table of the following format:



Bits	Field	Description
47-42	ℓ_1	Indicates primary level of the subprogram or overlay. It is determined by an IDENT or SEGMENT pseudo instruction as follows: 00 First IDENT 01 SEGMENT, or IDENT other than first that does not specify a primary level. n 2 octal digit specified on IDENT
41-36	ℓ_2	Indicates secondary level of the subprogram or overlay. It is determined by an IDENT or SEGMENT pseudo instruction as follows: 00 First IDENT, or IDENT that does not specify a secondary level, or any SEGMENT. n 2-octal digit specified on IDENT
35-18	fwa	Origin -1 - address where 50-table is loaded as specified on the IDENT or SEGMENT pseudo instruction.
17-00	eptaddr	Absolute address of entry point specified on the IDENT or SEGMENT pseudo instruction.

Multiple Entry Point Table (51)

A CPU overlay that has multiple entry points has a 51 table in place of a 50 table. The table has the following format:



Word	Bits	Field	Description
ID	47-36	ℓ_1, ℓ_2	Same as ℓ_1 and ℓ_2 in 50 control word.
	35-18	fwa	Origin -wc-1-address where 51-table is loaded as specified on the IDENT pseudo instruction
	17-00	wc	Length of table in octal, excluding ID word.
1,2,3, ..., wc	59-18	$eptsym_i$	Entry point symbol in display code, left justified with zero fill.
	17-00	$eptaddr_i$	Absolute address of $eptsym_i$

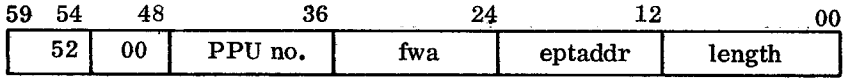
7600 PPU ABSOLUTE PROGRAM OR OVERLAY

Binary output for a 7600 PPU program or overlay is a logical record that may contain the following:

A prefix table (optionally suppressed through NOLABEL)
A 52_8 binary control table (optionally suppressed through NOLABEL)
Absolute image of all text generated since previous IDENT pseudo instruction.

PPU text is generated 5 PPU words per 60-bit CPU word.

The format of the control table is as follows:



Bits	Field	Description
47-36	PPU no.	Number of PPUs in which program or overlay is to be executed.
35-24	fwa	Origin -5 as specified on the IDENT pseudo instruction; address at which 52-table is loaded
23-12	eptaddr	Absolute address of the entry point specified on the IDENT pseudo instruction.
11-00	length	Number of CPU words in the program or overlay image including the 52 table (1/5 the number of PPU words).

6000 SERIES PPU ABSOLUTE PROGRAM OR OVERLAY

Binary output for a 6000 Series PPU program or overlay is a logical record that may contain the following:

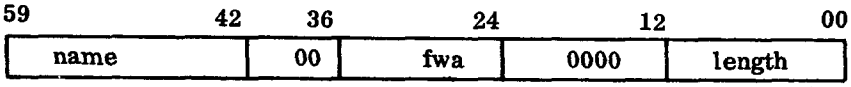
A prefix table (Optionally suppressed through NOLABEL)

A 6000 Series PPU program control table (Optionally suppressed through NOLABEL)

An absolute image of all text generated since previous IDENT or SEGMENT pseudo instruction.

PPU text is generated 5 PPU words per 60-bit CPU word.

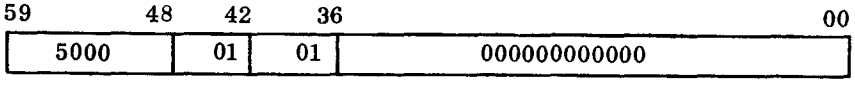
The format of the control table is as follows:



Bits	Field	Description
59-42	name	Program name, 1-3 display code characters, left justified with zero fill.
41-36	none	Reserved for future system use.
35-24	fwa	Origin -5 as specified on IDENT or SEGMENT pseudo instruction; at which table is loaded
23-12	none	Reserved for future system use.
11-00	length	Number of CPU words in program image (1/5 the number of PPU words)

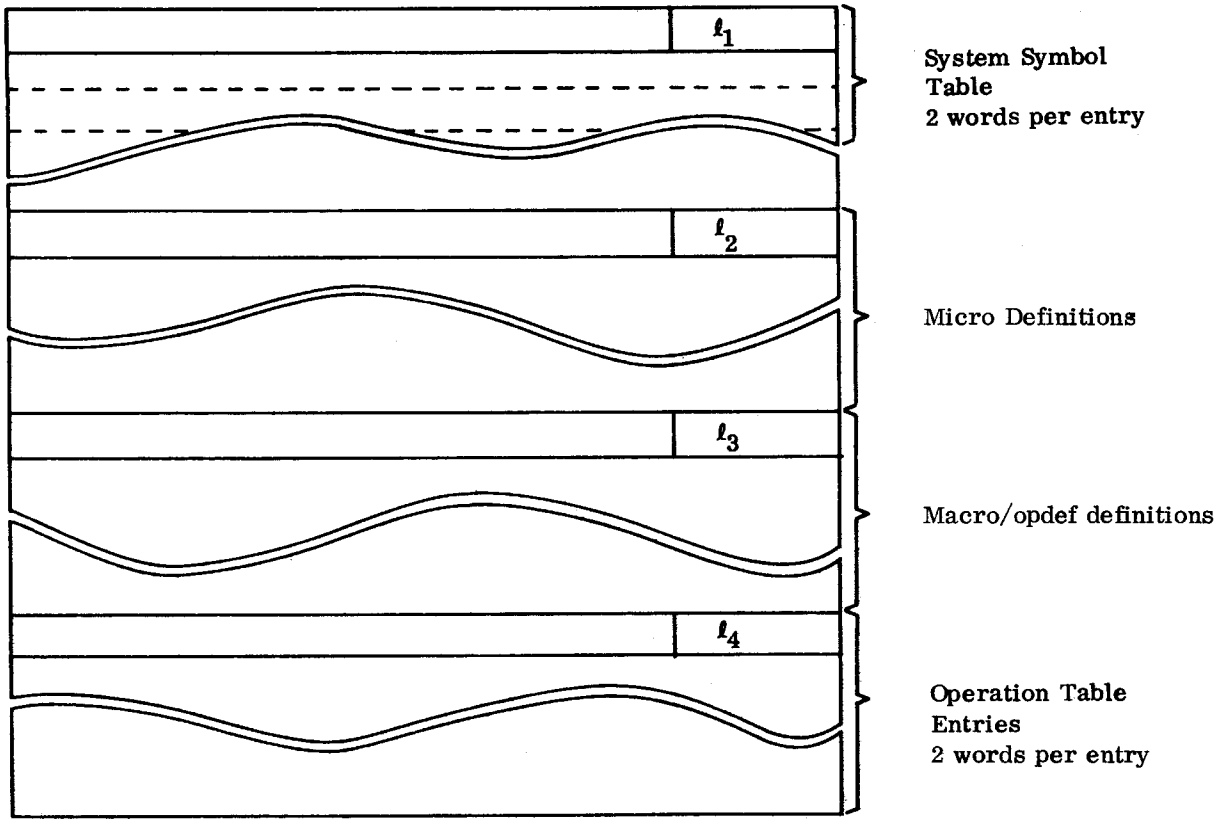
SYSTEM TEXT (SYSTEXT)

Normally, system text is derived from the library overlay named SYSTEXT† and is assembled prior to assembly of the source program. However, the source of system text can be changed through the S option on the COMPASS card (section 10.1.2). A system text overlay on the library is an absolute overlay that has the following control table:



†V2TEXT for SCOPE 2.

Format of Text:



ℓ_i= Number of words in each part of record.

COMPASS produces a systems text record as a result of encountering an STEXT pseudo instruction during assembly. If the rname is blank no binary is generated. If rname is nonblank, both the binary and the systems text are generated.

The System Symbol Table does not include the following symbols:

Local symbols

Qualified symbols

SET-defined symbols

Relocatable symbols (not in the absolute block)

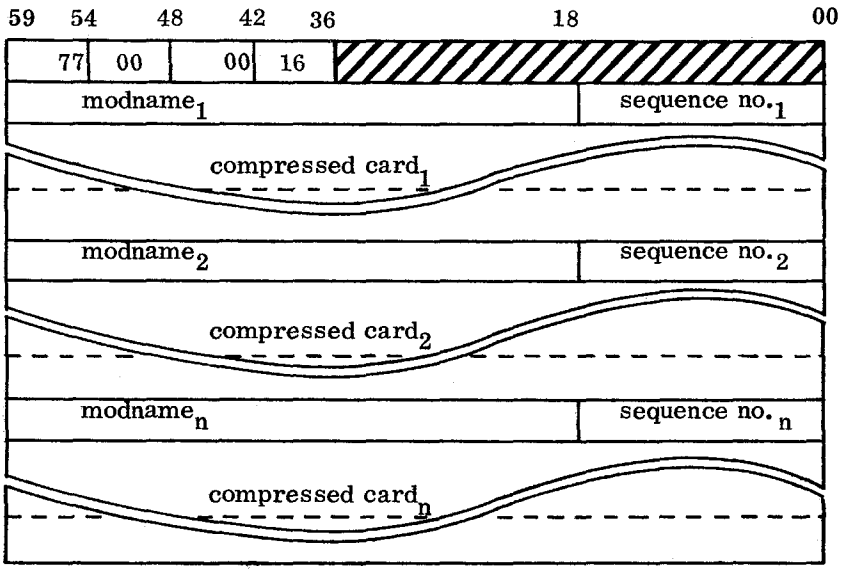
External text symbols, that is, library input symbols and symbols read from XTEXT or occurring between CTEXT and ENDX

SST-defined symbols

COMPRESSED COMPILE FILE

Source statement input for COMPASS assembly can be in the form of a 6000/7000 UPDATE or a 7600 MODIFY† created compressed compile file.

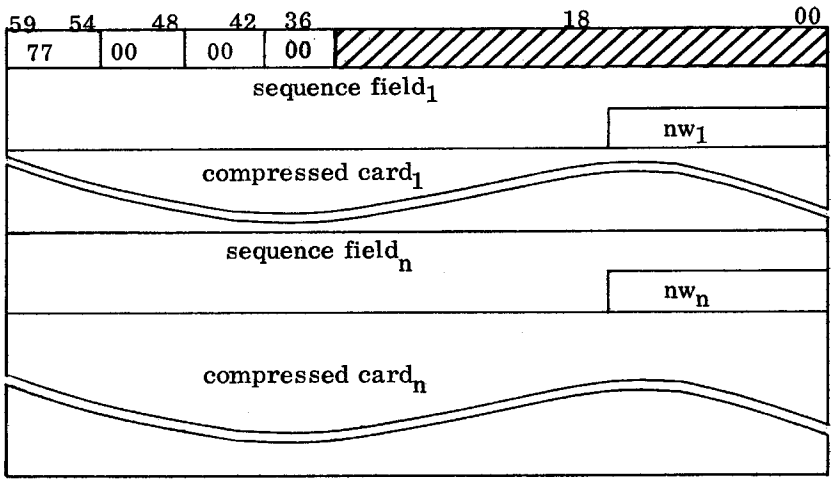
A compressed compile file record written by MODIFY† in A mode has the following format:



$modname_i$	1-7 character name identifying latest modification for the card.
sequence $no._i$	Sequence number of the card relative to the modification set identified by modname.
compressed card	A COMPASS source card in compressed form. That is, two or more consecutive blanks (to a maximum of 64) are replaced by a byte of the form 0001 through 0077_8. A single blank is represented in display code (55_8). If the source card contains 65 consecutive blanks, the 65^{th} is represented by a display code blank (55_8). 66 or more successive blanks are represented by 0001, etc. A zero byte (0000) signifies an end of line (EOL).

†MODIFY is not supported by SCOPE 2.

A compressed compile file record written by UPDATE in X mode is in the following format:

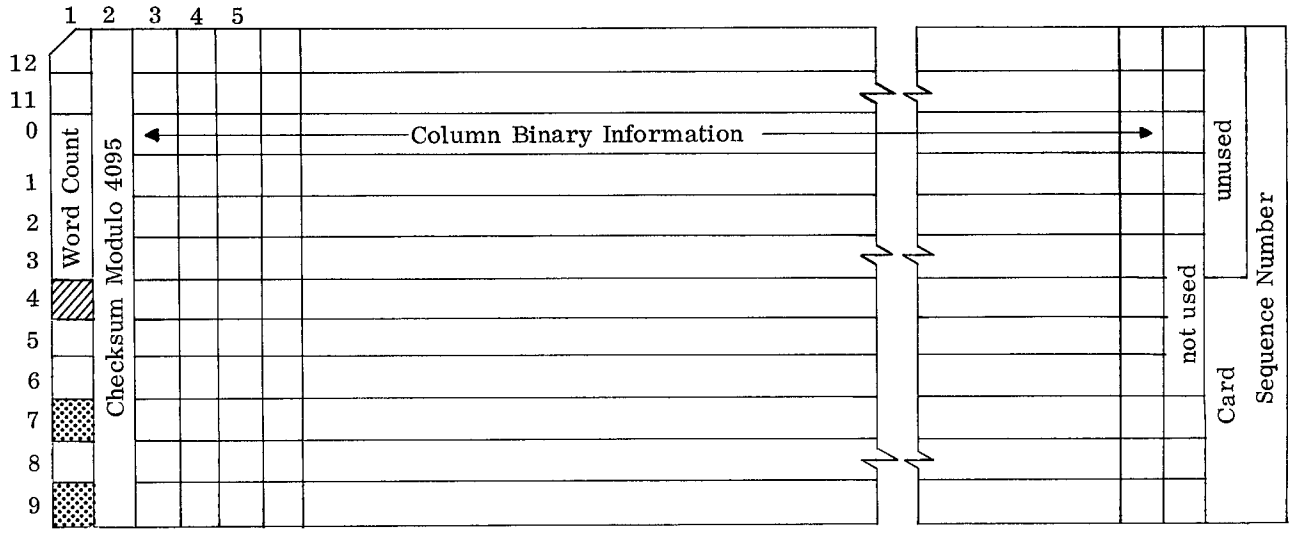


sequence field$_i$	17 characters comprising card columns 74-90 (column 73 is always blank).
nw$_i$	Binary number of words in compressed card$_i$.
compressed card$_i$	Columns 1-72 of a COMPASS source card in compressed form. That is, each 00 character is replaced by the 12-bit value 0001, and three or more consecutive blanks (to a maximum of 64) are replaced by a 12-bit value 0002 through 0077_8. A single blank is represented in display code (55_8); two consecutive blanks are represented by the 12-bit value 5555_8. If the last word is not full, it is padded on the right with binary zeros. Because word count nw$_i$ is present, an extra all-zero is not required to guarantee 12 zero bits.

BINARY CARD FORMATS C

Column 1

7,8,9	End of logical record
6,7,9	End of file
6,7,8,9	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 logical record 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.

Coded cards are translated on input from Hollerith to display code, and packed 10 columns per CPU word. A CPU word with a lowest byte of zero marks the end of a coded card (it is a coded record), and the full length of the card is not stored if it has trailing blanks. A compact form is thereby produced if coded cards are transferred to another device.

CHARACTER SETS D

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 C64.2), 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_8). 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

CODE D (default) | CODE E | CODE I | CODE A

Display Code		Hollerith Punch (026)	BCD			ASCII						EBCDIC			
						Upper Case			Lower Case			Upper		Lower	
Octal (13)	Char.		Ext. (13)	Int. (13)	6-Bit Octal (13)	Hex.	Char.	Punch (029)	Hex.	Char.	Punch	Hex.	Char.	Hex.	Char.
00	:(7)	8-2	00(8)	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			
						Upper Case			Lower Case			Upper		Lower	
Octal (13)	Char.		Ext. 13	Int. (13)	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	=	ID	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	≡ # (5)	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	% (6)	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 (9)	52	52	01	21	!	12-8-7 (11)	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 (10)	72	32	34	3C	<	12-8-4 (12)	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	EO	\	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

HINTS ON USING COMPASS E

1. Within a macro definition:

   a. Use comment cards having * in column one. These are not saved whereas other types of comments are saved.

   b. Whenever possible minimize the number of lines of code.

   c. IRP is faster than either ECHO or DUP.

   d. Use the substitutable parameter flags ;A, ;B, etc., for macros to avoid a second line.

   e. Within macros, use symbols such as .1, .2, etc. instead of local symbols.

   f. If possible, avoid recursive macro structure to increase assembly speed.

   g. 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:

   a. Use line counts rather than ENDIF to terminate sequences.

   b. Use SKIP rather than IFPP to skip code.

3. Micros:

   a. Micro replacement is time consuming.

   b. Avoid using local symbols for micros.

   c. Use $\neq \neq$ for a null substitution.

4. Minimize SYSTEXT size.

5. To reduce core requirements, use SEG cards in absolute programs.

6. Use NOREF for symbols for which listing is not required.

7. Use QUAL for all overlays.

DAYFILE MESSAGES F

nERRORS IN name

COMPASS issues this message for each source program in which fatal errors are detected.

nnnnnnB L C M NEEDED TO CONTINUE.

During initialization, the 7600 COMPASS Version 2 assembler estimates the number of LCM words required to begin processing. If the estimated number (in octal) is less than the job's LCM field length and if the job is in user controlled FLL mode, this message is issued and the job aborted.

nLOST REFERENCES IN name

COMPASS issues this message for each source program whose symbolic cross-reference table does not fit in the job's CM† field length for sorting just before it is printed. Rather than aborting the job, COMPASS discards some of the references. The ASSEMBLY COMPLETE message gives the field length needed to avoid lost references.

nnnnnnB S C M NEEDED TO CONTINUE.

During initialization, the 7600 COMPASS Version 2 assembler estimates the number of SCM words required to begin processing. If the estimated number (in octal) is less than the job's SCM field length and if the job is in user controlled FLS mode, this message is issued and the job aborted.

nWARNING MESSAGES IN name

COMPASS issues this message for each source program in which non-fatal errors are detected.

ASSEMBLING name

This message is displayed at the system operator's console only. It is not written in the dayfile. COMPASS updates the display whenever it processes an IDENT statement with a non-blank variable field.

ASSEMBLY COMPLETE. nB SCM††USED.

COMPASS issues this message when it has completed processing of all source programs on the input file without having detected any fatal errors. n is the octal number of SCM words needed. For instance, the minimum field length needed to perform the assemblies successfully. It may be larger than the actual field length, in which case, it is the minimum field length needed to avoid lost references.

†LCM for the 7600 COMPASS Version 2 assembler.

††For the 7600 COMPASS Version 2 assembler, SCM is replaced by LCM and is the octal number of LCM words required for the internal tables maintained in LCM.

ASSEMBLY ERRORS. nB SCM† USED.

COMPASS issues this message when it has completed processing of all source programs on the input file and detected at least one fatal error. If the A option was specified on the COMPASS control card, COMPASS aborts the job after issuing this message. n is the same as in the ASSEMBLY COMPLETE message.

BAD SYSTEMS TEXT.

The 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 overlay but does not abort the job.

CANT LOAD COMP1$

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.

ERROR IN COMPASS ARGUMENTS.

The COMPASS control card contains an unrecognized or invalid argument. The job is aborted.

IDENT CARD MISSING.

COMPASS issues this message for each source program in which an END statement is encountered before an IDENT statement. This is a fatal error.

INPUT FILE EMPTY OR MISPOSITIONED.

COMPASS encountered end of data when it attempted to read the first line from the source input file. After issuing this message, COMPASS generates an END card which causes the IDENT CARD MISSING message and a fatal error. This message is issued by 7600 COMPASS Version 2 only.

INSUFFICIENT STORAGE FOR SYSTEMS TEXT.

COMPASS issues this message and aborts the job when an irrecoverable table overflow occurs during system text loading, before the first assembly is begun. For 6000 COMPASS Version 2 and 7600 COMPASS Version 1, a substantial increase in the job's CM/SCM field length may be needed. The message is also issued if system text specified by the G option is of such length that it cannot be read into the job's field length. In this case, the job is not aborted but the system text is ignored. For the 7600 COMPASS Version 2 assembler, an increase may be needed for either LCM or SCM if the job is in user controlled FL mode. This message will be followed by MORE S C M NEEDED, or MORE L C M NEEDED.

†For the 7600 COMPASS Version 2 assembler, SCM is replaced by LCM and n is the octal number of LCM words required for the internal tables maintained in LCM.

INSUFFICIENT STORAGE. JOB ABORTED.

The job's CM field length is too small for COMPASS to begin processing. This message is issued not issued by 7600 COMPASS Version 2.

MORE L C M NEEDED.

This message is issued by the 7600 COMPASS Version 2 assembler after a message concerning table overflow has been issued in order to emphasize that the internal tables require LCM.

NO SYSTEM TEXT FOUND.

COMPASS issues this message, but does not abort the job, when it cannot load the system text specified on the COMPASS control card. For an overlay loaded from a library (S parameter), this message should be preceded by an explanatory message from the operating system loader. For an overlay loaded from non-library file (G parameter), COMPASS cannot find the overlay on the file.

RECURSION DEPTH EXCEEDED 400.

COMPASS maintains a push-down stack for source input control, with one entry for each active DUP, ECHO, HERE, XTEXT, OPDEF, or macro call. The maximum depth of this stack is set by an installation parameter, 400 in the released system. When this limit is exceeded, COMPASS sets a fatal error and clears the stack so that the next statement will be read from the source input file, but does not abort the job. This error is usually caused by a source program error in which a macro calls itself indefinitely.

TABLE OVERFLOW IN PASS n.

An irrecoverable table overflow condition has occurred in assembly pass 1 or 2 while processing a source program. COMPASS allocates memory space dynamically to all of its internal tables, so that when one overflows, all do. When the tables do not all fit in the available CM† space, COMPASS stores some of them on mass storage scratch files.

COMPASS issues the above message, and aborts the job, when CM† is insufficient after all such files have been written to mass storage.

†LCM for the 7600 COMPASS Version 2 assembler.

INDEX

COMMENT SHEET

CONTROL DATA
CORPORATION

TITLE: 6000 COMPASS Version 2
7000 COMPASS Versions 1 and 2 Reference Manual

PUBLICATION NO. 60279900 REVISION D

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Documentation Department
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE

PSEUDO INSTRUCTION INDEX

Name	Placement	Usage	Section Number
ABS	first group	CPA	4.3.1
BASE	anywhere	CP, PP	4.4.1
BSS	normal	CP, PP	4.5.4
BSSZ	normal	CP, PP	4.8.1
B1_1	anywhere	CP	4.4.4
B7_1	anywhere	CP	4.4.4
CODE	anywhere	CP, PP	4.4.2
COL	normal	CP, PP	4.4.5
COMMENT	anywhere	CP, PP	4.3.9
CON	normal	CP, PP	4.8.6
CPOP	anywhere	CP	6.2.1
CPSYN	anywhere	CP	6.2.2
CTEXT	normal	CP	4.11.7
DATA	normal	CP, PP	4.8.2
DECMIC	anywhere	CP, PP	7.2.2
DIS	normal	CP, PP	4.8.3
DUP	normal	CP, PP	5.3.1
ECHO	normal	CP, PP	5.3.2
EJECT	anywhere	CP, PP	4.11.2
ELSE†	anywhere	CP, PP	4.9.2
END†	required last	CP, PP	4.2.2
ENDD	anywhere	CP, PP	5.3.4
ENDIF†	anywhere	CP, PP	4.9.1
ENDM	anywhere	CP, PP	5.4.1
ENDX	normal	CP, PP	4.11.7
ENTRY	normal	CP, PP	4.7.1
EQU	normal	CP, PP	4.6.1
ERR	normal	CP, PP	4.10.1
ERRNG	normal	CP, PP	4.10.2
ERRNZ	normal	CP, PP	4.10.2
ERRPL	normal	CP, PP	4.10.2
ERRZR	normal	CP, PP	4.10.2
EXT	normal	CP, PP	4.7.2
HERE	anywhere	CP, PP	5.2.2
IDENT	required first	CP, PP	4.2.1 and 4.3.4
IF	normal	CP, PP	4.9.5
IFC	anywhere	CP, PP	4.9.6
IFCP	normal	CP, PP	4.9.3
IFGE	normal	CP, PP	4.9.4
IFGT	normal	CP, PP	4.9.4
IFLE	normal	CP, PP	4.9.4
IFLT	normal	CP, PP	4.9.4
IFMI	normal	CP, PP	4.9.7
IFNE	normal	CP, PP	4.9.4
IFPL	normal	CP, PP	4.9.7
IFPP	normal	CP, PP	4.9.3
IFEQ	normal	CP, PP	4.9.4
IRP	anywhere	CP, PP	5.4.9
LCC	normal	CPR	4.3.8
LIST	anywhere	CP, PP	4.11.1
LIT	normal	CP, PP	4.8.4
LOC	normal	CP, PP	4.5.5
LOCAL	macro or opdef	CP, PP	5.4.8
MACRO	anywhere	CP, PP	5.4.2
MACROE	anywhere	CP, PP	5.4.4
MAX	normal	CP, PP	4.6.3
MICCNT	anywhere	CP, PP	4.6.5
MICRO	anywhere	CP, PP	7.2.1
MIN	normal	CP, PP	4.6.4
NIL	anywhere	CP, PP	6.1.3
NOLABEL	anywhere	CP, PP	4.3.10
NOREF	anywhere	CP, PP	4.11.6
OCTMIC	anywhere	CP, PP	7.2.3

Name	Placement	Usage	Section Number
OPDEF	anywhere	CP	5.4.6
OPSYN	anywhere	CP, PP	6.1.2
ORG	normal	CP, PP	4.5.3
PERIPH	first group	PP	4.3.3
POS	normal	CP, PP	4.5.6
PPOP	anywhere	PP	6.1.2
PPU	first group	PP	4.3.2
PURGDEF	anywhere	CP	6.2.3
PURGMAC	anywhere	PP	6.1.4
QUAL	anywhere	CP, PP	4.4.3
REP	normal	CPR	4.8.8
REPI	normal	CPR	4.8.8
RMT	anywhere	CP, PP	5.2.1
R=	normal	CP	4.8.7
SEG	normal	CPA, PP	4.3.6
SEGMENT	normal	CPA, PP	4.3.5
SET	normal	CP, PP	4.6.2
SKIP	anywhere	CP, PP	4.9.8
SPACE	anywhere	CP, PP	4.11.3
SST	anywhere	CP, PP	4.6.6
STEXT	first group	CP, PP	4.3.7
STOPDUP	normal	CP, PP	5.3.3
TITLE	anywhere	CP, PP	4.11.4
TTL	anywhere	CP, PP	4.11.5
USE	normal	CP, PP	4.5.1
USELCM	normal	CP	4.5.2
VFD	normal	CP, PP	4.8.5
XREF	anywhere	CP, PP	4.11.8
XTEXT	normal	CP, PP	5.1
(blank)	normal	CP, PP	4.8.1
=	normal	CP, PP	4.6.1

Legend

CP	Absolute or relocatable CPU program
CPA	Absolute CPU program
CPR	Relocatable CPU program
PP	Absolute PPU program

† Looked for during IF skipping.