

TTLS (Subtitle)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2	6	7	8	14	15	16	32
	Blanks		TTLS					Subtitle in the variable field
	or an							
	integer							

The TTLS pseudo-operation is identical in function to the TTL pseudo-operation except that it causes subtitling to occur. When a TTLS pseudo-operation is encountered, the subtitle provided in columns 16-72 replaces the current subtitle; the output listing is restored to the top of the next page. The title and new subtitle are then printed.

The maximum number of subtitles that may follow a title is one.

ABS (Output Absolute Text)

LOCATION						E O	OPERATION						ADDRESS, MODIFIER						COMMENTS					
1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20				
Blanks							ABS												Column 16 must be blank					

The ABS pseudo-operation causes the Assembler to output absolute binary text.

The normal mode of the Assembler is relocatable; however, if absolute text is required for a given assembly, the ABS pseudo-operation should appear in the deck before any instructions or data. It may be preceded only by listing pseudo-operations. It may, however, appear repeatedly in an assembly interspersed with the FUL pseudo-operation. It should be noted that the pseudo-operations affecting relocation are considered errors in an absolute assembly.

Those pseudo-operations that will be in error if used in an absolute assembly are:

BLOCK
ERLK

SYMDEF
SYMREF

(Refer to the descriptions of binary punched card formats in this chapter for details of the absolute binary text.)

FUL (OUTPUT Full Binary Text)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	8	14	15	16
								32
	Blanks			FUL				Column 16 must be blank

The FUL pseudo-operation is used to specify absolute assembly and the FUL format for absolute binary text.

The FUL pseudo-operation has the same effect and restrictions on the Assembler as ABS, except for the format of the binary text output. The format of the text is of continuous information with no address identification; that is, the absolute binary cards are punched with program instructions in columns 1-78 (26 words). Such cards can be used in self-loading operations or other environments where control words are not required on the binary card.

TCD (Punch Transfer Card)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	8	14	15	16
								32
	Blanks			TCD				An expression in the variable field
	or a							
	symbol							

In an absolute assembly, the binary transfer card, produced at the end of the deck as a result of the END card, directs the loading program to cease loading and turn control over to the program at the point specified by the transfer card. Sometimes it is desirable to cause a transfer card to be produced before encountering the end of the deck. This is the purpose of the TCD pseudo-operation. Thus, a binary transfer card is produced generating a transfer address equivalent to the value of the expression in the variable field.

TCD is an error in the relocatable mode.

HEAD (Heading)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	8	14	15	16
								32
	Blanks			HEAD				From 1 to 7 subfields in the variable field,
								each containing a single, nonspecial character
								used as a heading character

In programming, it is sometimes desirable to combine two programs, or sections of the same program, that use the same symbols for different purposes. The HEAD pseudo-operation makes such a combination possible by prefixing each symbol of five or fewer characters with a heading character. This character must not be one of the special characters; that is, it must be one of the characters A-Z, 0-9, or the period(.). Using different heading characters, in different program sections later to be combined for assembly, removes any ambiguity as to the definition of a given symbol.

The effect of the HEAD pseudo-operation is to cause every symbol of five or less characters, appearing in either the location field or the variable field, to be prefixed by the current HEAD character. The current HEAD character applies to all symbols appearing after the current HEAD pseudo-operation and before the next HEAD or END pseudo-operation.

Deheading is accomplished by a zero or blanks in the variable field. To understand more thoroughly the operation of the heading function, it is necessary to know that the Assembler internally creates a six-character symbol by right-justifying the characters of the symbol and filling in leading zeros. Thus, if the Assembler is within a headed program section and encounters a symbol of five or fewer characters, it inserts the current HEAD character into the high-order, leftmost character position of the symbol. Each symbol, with its inserted HEAD character, then can be placed in the Assembler symbol table as unique entries and assigned their respective location values.

It is also possible to head a program section with more than one character. This is done by using the pseudo-operation HEAD in the operation field with from two to seven heading characters in the variable field, separated by commas. The effect of a multiple heading is to define each symbol of that section once for each heading character. Thus, for example, if the symbols SHEAR, SPEED, and PRESS are headed by

HEAD		X,Y,Z
nine unique symbols		
XSHEAR	XSPEED	XPRESS
YSHEAR	YSPEED	YPRESS
ZSHEAR	ZSPEED	ZPRESS

are generated and placed in the Assembler symbol table. This allows regions by HEADX, HEADY, or HEADZ to obtain identical values for the symbols SHEAR, SPEED, and PRESS.

Cross-referencing among differently headed sections may be accomplished by the use of six-character symbols or by the use of the dollar sign (\$). Six character symbols are immune to HEAD; therefore, they provide a convenient method of cross-referencing among differently headed regions.

To allow the programmer more flexibility in cross-referencing, the Assembler language includes the use of the dollar sign (\$) to denote references to an alien-headed region.

If the programmer wishes to reference a symbol of less than six characters in another program section, he merely prefixes the symbol by the HEAD character for that respective section, separating the HEAD character from the body of the symbol by a dollar sign (\$).

To reference from a headed region into a region that is not headed, the programmer may use either the heading character zero (0) preceding the symbol or if the symbol is the initial value of the variable field, then the appearance of the leading dollar sign will cause the zero heading to be attached to the symbol.

EXAMPLE OF HEAD PSEUDO-OPERATION

START	LDA	A	Initial instruction (no heading)

	TRA	B\$SUM	Transfer to new headed section
A	BSS	1	
	HEAD	B	
SUM	LDA	\$A	} Section headed B

	TRA	0\$START + 2	
	END		

The LDA \$A could have been written as LDA 0\$A, as they both mean the same.

DCARD (Punch BCD Card)

LOCATION		E O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1	2				
6	7	8	14	15	16
Blanks			DCARD		Two subfields in the variable field

The first subfield contains a decimal integer N (limited only by the size of available memory), and the record subfield contains a single BCD character used as a decimal data identifier. The Assembler punches the next N cards after the DCARD instruction with the specified BCD identifier in column one of each of these N cards and with the BCD information taken from the corresponding source cards on a one-for-one basis.

There are no restrictions on the BCD information that can be placed in columns 2-72 of the source cards. (One of the significant uses of DCARD is to generate Operating Supervisor (GECOS) \$ control cards.)

The DCARD has the further effect of suppressing the normal automatic generation of a \$ OBJECT and \$ DKEND card.

END (End of Assembly)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1 2	6 7 8		14 15 16		32			
	Blanks		END				Blanks or an expression in the variable field	
	or a							
	symbol							

The END pseudo-operation signals the Assembler that it has reached the end of the symbolic input deck; it must be present as the last physical card encountered by the Assembler.

If a symbol appears in the location field, it is assigned the next available location.

In a relocatable assembly, the variable field must be blank; in an absolute assembly, the variable may contain an expression. In relocatable decks, the starting location of the program will be an entry location and the location specified is given to the General Loader (GELOAD) by a special control card used with the GELOAD. (Refer to the GELOAD manual.) Absolute programs require a binary transfer card which is generated by the END pseudo-operation. The Transfer address is obtained from the expression in the variable field of the END card.

OPD (Operation Definition)

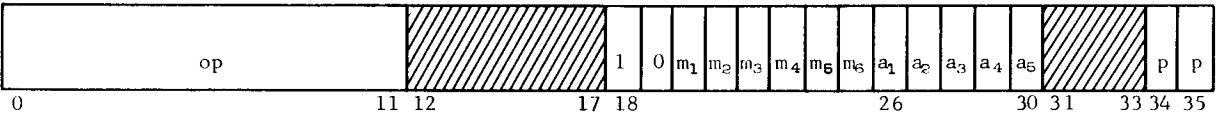
LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	New		OPD						One or more subfields, separated by commas, in the variable field. The subfields define the bit configuration of the new operation code
	operation								
	code								

The OPD pseudo-operation may be used to define or redefine machine instructions to the Assembler. This allows programmers to add operation codes to the Assembler table of operation codes during the assembly process. This is extremely useful and powerful in defining new instructions or special bit configurations, unique in a particular program, to the Assembler.

The variable field subfields are bit-oriented and have the same general form as described under the VFD pseudo-operation. In addition, the variable field, considered in its entirety, requires the use of either of two specific 36-bit formats for defining the operation.

- 1. The normal instruction format
- 2. The input/output operation format

The normal instruction-defining format and subfields are shown below:



- op--new operation code (bits 18 through 29 of instruction)
- m--modifier tag type (0-allowed; 1-not allowed)
 - m₁: register modification (R)
 - m₂: indirect addressing (*)
 - m₃: indirect and tally (T)
 - m₄: Direct Upper (DU)
 - m₅: Direct Lower (DL)
 - m₆: Sequence Character (SC) and Character from Indirect (CI)
- a--address field conditions (0 not required; 1-required)
 - a₁: address required/not required
 - a₂: address required even
 - a₃: address required absolute
 - a₄: symbolic index required
 - a₅: octal tag field required
- p--octal assembly listing format (x represents one octal digit)
 - 00: xx xxxx xxxxxx
 - 01: xxxxxxxxxxxx
 - 10: xxxxxx xxxxxx
 - 11: xxxxxx xxxx xx

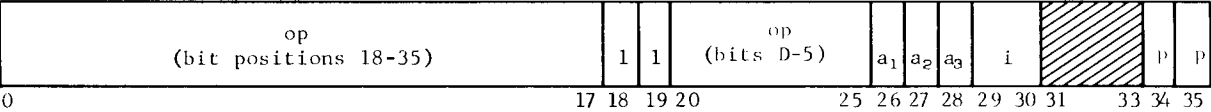
The assembly listing types 00, 01, 10, and 11 are used for input/output commands, data-generating pseudo-operations (OCT, DEC, BCI, etc.), special word-generating pseudo-operations (such as ZERO), and machine instructions.

To illustrate the use of OPD, assume one wished to define the extant machine instruction, Load A (LDA). Using the preceding format and the octal notation (as described under the VFD pseudo-operation), one could code OPD as

or	LDA	OPD	O12/2350, 6/, O2/2, 6/, O3/4, 5/, O2/3
or	LDA	OPD	O18/235000, O2/2, 6/, O3/4, 5/, O2/3
	LDA	OPD	O36/2350000401003

or in other forms, providing the bit positions of the instruction-defining format are individually specified to the Assembler.

The input/output operation-defining format and subfields are as follows:



op--new operation code for bit positions 18-35 and 0-5 (see Appendix E)

a--address field conditions (0=not required; 1=required)

a₁: address required/not required

a₂: address required even

a₃: address required absolute

i--type of input/output command (see Appendix E)

00: OP DA,CA KKDACA KKKKKK

01: OP NN,DA,CA KKDACA KKKKNN

10: OP CC,DA,CA KKDACA KKCKKK

11: OP A,C AAAAAA KKCCCC

p--see preceding normal instruction format

Input/output operation types 00, 01, and 10 are the formats for the commands; type 11 is the format for a Data Control Word (DCW).

As an example of the use of OPD to generate an input/output command (using the above format for the variable field and defining the bits according to the rules for VFD), assume one wanted to generate the extant command, Write Tape Binary (WTB--Appendix E). This could be written as

WTB OPD 18/,O2/3,O6/15,10/0

or in various other bit-oriented forms.

OPSYN (Operation Synonym)

A sym- OPSYN
bol or
opera-
tion
code

A mnemonic operation code in the
variable field.

The OPSYN pseudo-operation is used for equating either a newly defined symbol or a presently defined operation to some operation code already in the operation table of the Assembler. The operation code may have been defined by a prior OPD or OPSYN pseudo-operation; in any case, it must be in the Assembler operation table. The new symbol to be defined is entered in the location field and the operation code that must be in the Assembler operation table is entered in the variable field.

Location Counter Pseudo-Operations

USE (Use Multiple Location Counters)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	Blanks			USE					A single symbol, blanks, or the word PREVIOUS in the variable field

The Assembler provides the ability to employ multiple location counters via the USE pseudo-operation. The use of this pseudo-operation causes the Assembler to place succeeding instructions under control of the location counter represented by the symbol in the variable field. Each location counter begins with the value of zero, and its size is determined as being the highest value assumed by it (that is, occupied by some instruction assembled under it). This is not always the last instruction under the USE, as an ORG may have occurred within it. At the completion of the first pass through the symbolic program, the length of each USE will be a known value, and the order of their memory allocation will be implied by the order of their first presentation to the Assembler. Thus, the origin of each location counter may be computed based on the origin and size of the one preceding it. There is an assumed location counter, called the blank USE, implied in all assemblies, which has a natural origin of zero.

Automatic determination of a counter origin may be overridden with the BEGIN pseudo-operation. In this case, the chain of location counters will be made, completely ignoring those counters which had an associated BEGIN. In more general terms, then, the origin of a non-begin location counter is taken as one more than the highest value taken by the next prior non-begin counter. The first of these non-begin counters has an origin of zero, by definition. The location counter which is in control at the time that a USE is encountered is suspended at its current value and is preserved as the PREVIOUS counter. It may be called back into operation at any later point in the program without confusion as to its current state, and will begin counting at the address which is one higher than the last location used under it.

If the word PREVIOUS appears in the variable field, the Assembler reactivates the location counter which appeared just before the present one. It is not possible to go back more than one level via the USE PREVIOUS command, as the one in control when the USE PREVIOUS is encountered is made previous.

BEGIN (Origin of a Location Counter)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	Blanks			BEGIN					Two subfields in the variable field

The BEGIN pseudo-operation is used to arbitrarily specify the origin of a given location counter. As such, it will not be tied into the chain of location counters as described in USE. Its origin, however, may be an expression involving some symbol or symbols defined under another location counter, in which case it will be linked to the chain at the specified point. The user must beware of overlaying code with this pseudo-operation. It is primarily intended for the more sophisticated user. Under normal programming circumstances its power is not needed.

The location counter symbol is specified in the first subfield and is given the value specified by the expression found in the second subfield. Any symbol appearing in the second subfield must have been previously defined and must appear under one location counter. The BEGIN pseudo-operation may appear anywhere in the deck. It does not invoke the counter, however. A USE must be given to bring a location counter into effect.

ORG (Origin Set by Programmer)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER				COMMENTS
1	2		6	7	8	14	15	16	32
	Blanks		ORG						An expression in the variable field
	or a								
	symbol								

The ORG pseudo-operation is used by the programmer to change the next value of a counter, normally assigned by the Assembler, to a desired value. If ORG is not used by the programmer, the counter is initially set to zero.

All symbols appearing in the variable field must have been previously defined. If a symbol appears in the location field, it is assigned the value of the variable field. If the result of the evaluation of a variable field expression is absolute, the instruction counter will be reset to the specified value relative to the current location counter. If an expression result is relocatable, the current location counter will be suspended, and the counter to which the expression is relocated will be invoked with the value given by the expression.

LOC (Location of Output Text)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER				COMMENTS
1	2		6	7	8	14	15	16	32
	Blanks		LOC						An expression in the variable field

The LOC pseudo-operation functions identically to the ORG pseudo-operation, with one exception; it has no effect on the loading address when the Assembler is punching binary text. That is, the value of the location counter will be changed to that given by the variable field expression, but the

loading will continue to be consecutive. This provides a means of assembling code in one area of memory while its execution will occur at some other area of memory.

All symbols appearing in the variable field of this pseudo-operation must have been previously defined.

The sole purpose of this pseudo-operation is to allow program coding to be loaded in one section of memory and then to be subsequently moved to another section for execution.

Symbol-Defining Pseudo-Operations

Increased facility in program writing frequently can be realized by the ability to define symbols to the Assembler by means other than their appearance in the location field of an instruction or by using a generative pseudo-operation. Such a symbol definition capability is used for (1) equating symbols, or (2) defining parameters used frequently by the program but which are subject to change. The symbol-defining pseudo-operations serve these and other purposes.

It should be noted that they do not generate any machine instructions or data but are available merely for the convenience of the programmer.

EQU (Equal To)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER				COMMENTS
1	2		6	7 8	14	15	16	32	
	Symbol			EQU					An expression in the variable field

The purpose of the EQU pseudo-operation is to define the symbol in the location field to have the value of the expression appearing in the variable field. The symbol in the location field will assume the same mode as that of the expression in the variable field, that is, absolute or relocatable. (See Relocatable and Absolute Expressions.)

All symbols appearing in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot appear in the variable field.

If the asterisk (*) appears in the variable field denoting the current location counter value, it will be given the value of the next sequential location not yet assigned by the Assembler with respect to the unique location counter presently in effect.

FEQU (Special FORTRAN Equivalence)

All symbols appearing in the variable field must have been previously defined.

SET (Symbol Redefinition)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS
1	2		6	7			
			8	14	15	16	32
	Symbol		SET				An expression in the variable field

The SET pseudo-operation permits the redefinition of a symbol previously defined to the Assembler. This ability is useful in Macro expansions where it may be undesirable to use created symbols (CRSM).

All symbols entered in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot be used in the variable field.

The symbol in the location field is given the value of the expression in the variable field. The SET pseudo-operation may not be used to define or redefine a relocatable symbol. (See Relocatable and Absolute Expressions.)

When the symbol occurring in the location field has been previously defined by a means other than a previous SET, the current SET, pseudo-operation will be ignored and flagged as an error.

The last value assigned to a symbol by SET affects only subsequent in-line coding instructions using the redefined symbol.

MIN (Minimum)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS
1	2		6	7			
			8	14	15	16	32
	Symbol		MIN				A sequence of expressions, separated by commas, in the variable field -- all of the same type; that is, relocatable or absolute

The MIN pseudo-operation defines the symbol in the location field as having the minimum value among the various values of all relocatable or all absolute expressions contained in the variable field.

All symbols appearing in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot be used in the variable field.

MAX (Maximum)

The MAX pseudo-operation is coded in the same format as MIN above. It defines the symbol in the location field as having the maximum value of the various expressions contained in the variable field.

All symbols appearing in the variable field must have been previously defined and must fall under the same location counter. SYMDEF or SYMREF symbols cannot be used in the variable field.

SYMDEF (Symbol Definition)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2	6 7 8		14 15 16			
	Blanks		SYMDEF			Symbols separated by commas in the variable field

The SYMDEF pseudo-operation is used to identify symbols which appear in the location field of a subroutine when these symbols are referred to from outside the subroutine (by SYMREF). Also, the programmer must provide a unique SYMDEF for use by the Loader to denote each subprogram entry point for the loading operations. The symbols used in the variable field of a SYMDEF instruction will be called SYMDEF symbols. Multiply defined SYMDEF symbols cannot occur since the Assembler ignores the current definition if it finds the same symbol previously entered in the SYMDEF table.

The appearance of a symbol in the variable field of a SYMDEF instruction indicates that:

1. The symbol must appear in the location field of only one of the instructions within the subroutine in which SYMDEF occurs.
2. The Assembler will place each such SYMDEF symbol along with its relative address in the preface card.
3. At load time, the Loader will form a table of SYMDEF symbols to be used for linkage with SYMREF symbols.

It is possible to classify SYMDEF symbols as primary and secondary. A secondary SYMDEF symbol is denoted by a minus sign in front of the symbol. The Loader will provide linkage for a secondary SYMDEF symbol only after linkage has been required to a primary SYMDEF within the same subprogram. The use of secondary SYMDEF symbols is intended for programmers who are specifically concerned with using the system subroutine library and generating routines for accessing the library. Secondary SYMDEF symbols are normally thought of as secondary entries to subroutines contained within a subprogram library package that will be used as an entire package. (The use of primary and secondary SYMDEF symbols is further described in the General Loader--GELOAD--manual.)

SYMREF (Symbol Reference)

LOCATION		E O	OPERATION			ADDRESS, MODIFIER		COMMENTS
1	2		6	7	8	14	1516	
	Blanks		SYMREF					A sequence of symbols separated by commas entered in the variable field

The SYMREF pseudo-operation is used to denote symbols which are used in the variable field of a subroutine but are defined in a location field external to the subroutine. Symbols used in the variable field of a SYMREF instruction will be called SYMREF symbols.

When a symbol appears in the variable field of a SYMREF instruction, the following items apply:

1. The symbol should occur in the variable field of at least one instruction within the subroutine.
2. At assembly time the Assembler will enter the SYMREF symbol in the preface card of the assembled deck and place a special entry number (page III-85) in the variable fields of all instructions in the referenced subroutine which contain the symbol.
3. At load time the Loader will associate the SYMREF symbol with a corresponding SYMDEF symbol and place the appropriate address in all instructions that have been given the special entry number.

Symbols appearing in the variable field of a SYMREF instruction must not appear in the location field of any instruction within the subroutine in which SYMREF is used.

EXAMPLE OF SYMDEF AND SYMREF PSEUDO-OPERATIONS

Base Program or Subprogram			Referencing Subroutine	
ATAN2 ATAN3	SYMDEF	ATAN,ATAN2	SYMREF	ATAN,ATAN2
	STC2	INDIC	⋮	
	SAVE	0,1	⋮	
	SZN	INDIC	⋮	
	TZE	START	FLD	X
ATAN	⋮		⋮	
	STZ	INDIC	TSX1	ATAN
	TRA	ATANS	⋮	
	⋮		TSX1	ATAN2
			⋮	

NULL (Null)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	Symbol		NULL						The variable field is not interpreted.

The NULL pseudo-operation acts as an NOP machine instruction to the Assembler in that no actual words are assembled. A symbol on a NULL will be defined as current value of the location counter.

EVEN (Force Location Counter Even)

Symbol EVEN
or
blanks

The variable field is not interpreted

The EVEN pseudo-operation accomplishes the same end result as the E in column 7. If the location counter is odd, a NOP is generated, thereby making it even. If there is a symbol in the location field it will be defined at the even address.

ODD (Force Location Counter Odd)

Symbol ODD
or
blanks

The variable field is not interpreted

The ODD pseudo-operation acts as if an O has been punched in column 7. If the location counter is even, a NOP is generated, thereby making it odd. If there is a symbol in the location field it will be defined at the odd address.

EIGHT (Force Location Counter to a Multiple of 8)

Symbol EIGHT or blanks	The variable field is not interpreted
------------------------------	---------------------------------------

The EIGHT pseudo-operation behaves as an 8 punched in column 7. If the location counter is not a multiple of 8, a TRA *+n is generated, where the value of *+n is the next location which is a multiple of 8, and the location counter is bumped by n. If there is a symbol in the location field it will be defined at the mod-8 address.

NOTE: In each of the 3 pseudo-operations, (EVEN, ODD, and EIGHT) the origin of the location counter will also be forced to a related address. For EVEN and ODD, it will be forced even, and for EIGHT, it will be forced to a multiple of eight.

Data Generating Pseudo-Operations

The Assembler language provides four pseudo-operations which can be used to generate data in the program at the time of assembly. These are BCI, OCT, DEC, and VFD. The first three, BCI, OCT, and DEC, are word-oriented while VFD is bit-oriented. There exists a fifth pseudo-operation, DUP, which in itself does not generate data, but through its repeat capability causes symbolic instruction and pseudo-operations to be iterated.

OCT (Octal)

Symbol OCT or blanks	One or more subfields separated by commas appearing in the variable field, each one containing a signed or unsigned octal integer.
----------------------------	--

The OCT pseudo-operation is used to introduce data in octal integer notation into an assembled program. The OCT pseudo-operation causes the Assembler to generate n locations of OCT data where the variable field contains n subfields (n-1 commas). Consecutive commas in the variable field cause the generation of a zero data word, as does a comma followed by a terminal blank. Up to 12 octal digits plus the leading sign may make up the octal number.

The OCT configuration is considered true and will not be complemented on negatively signed numbers. The sign applies only to bit 0. All assembly program numbers are right-justified, retaining the integer form.

EXAMPLE OF OCT PSEUDO-OPERATION

OCT 1,-4,7701,+3,, -77731,04

If the current location counter were set at 506, the above would be printed out as follows (less the column headings):

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>	
000506	0000000000001	000	OCT 1, -4, 7701, +3, , -77731, 04
000507	4000000000004	000	
000510	0000000007701	000	
000511	0000000000003	000	
000512	0000000000000	000	
000513	400000077731	000	
000514	0000000000004	000	

DEC (Decimal)

Symbol DEC	One or more subfields in the variable field,
or	separated by commas, each containing a decimal
blanks	entry.

The Assembler language provides four types of decimal information which the programmer may specify for conversion to binary data to be assembled. The various types are uniquely defined by the syntax of the individual subfields of the DEC pseudo-operation. The basic types are single-precision, fixed-point numbers; single-precision, floating-point numbers; double-precision fixed-point numbers; and double-precision floating-point numbers. All fixed-point numbers are right-justified in the assembly binary words; floating-point numbers are left-justified to bit position eight with the binary point between positions 0 and 1 of the mantissa. (The rules for forming these numbers are described under Decimal Literals, page III-12.)

EXAMPLES OF SINGLE-PRECISION DEC PSEUDO-OPERATION

GAMMA DEC 3,-1,6...2E1,1B27,1.2E1B32,-4

The above would print out the following data words (without column headings), assuming that GAMMA is located at 1041.

GE(000) SERIES

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>	
001041	000000000003	000	GAMMA DEC 3,-1,6,.. 2E1,1B27, 1.2E1B32,-4
001042	777777777777	000	
001043	006600000000	000	
001044	004400000000	000	
001045	000000000400	000	
001046	000000000140	000	
001047	777777777774	000	

The presence of the decimal point and/or the E scale factor implies floating-point, while the added B (binary scale) implies fixed-point binary numbers. The absence of all of these elements implies integers. Several more examples follow (see decimal literals for further explanation):

DEC -1B17,-1.,1000

With the location counter at 1050, the above would generate:

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>	
001050	777777000000	000	DEC -1B17,-1.,1000
001051	001000000000	000	
001052	000000001750	000	

EXAMPLE OF DOUBLE-PRECISION DEC PSEUDO-OPERATION

BETA DEC .3D0,0.D0,1.2D1B68,1D-1

The location counter is at the address BETA (1060); the above subfields generate the following double-words:

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>	
001060	776463146314	000	BETA DEC .3D0,0.D0, 1.2D1B68,1D-1
001061	631463146314	000	
001062	400000000000	000	
001063	000000000000	000	
001064	000000000000	000	
001065	000000000140	000	

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>
001066	772631463146	000
001067	314631463146	000

BCI (Binary Coded Decimal Information)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	8	14	15	16
	Symbol		BCI					Two subfields in the variable field: a count
	or							subfield and a data subfield
	blanks							

The BCI pseudo-operation is used by the programmer to enter binary-coded decimal (BCD) character information into a program.

The first subfield is numeric and contains a count that determines the length of the data subfield. The count specifies the number of 6-character machine words to be generated; thus, if the count field contains n, then the data subfield contains 6n characters of data. The maximum value which n can be is 9. The minimum value for n is 0. If n is 0, no words will be generated.

The second subfield contains the BCD characters, six per machine word.

EXAMPLE OF BCI PSEUDO-OPERATION

BETA BCI 3,NO ERROR CONDITION

Again assume the location counter set at 506 (location of BETA); the above would print out (less column headings):

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>	
000506	454620255151	000	BETA BCI 3, NO ERROR CONDITION
000507	465120234645	000	
000510	243163314645	000	

VFD (Variable Field Definition)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER						COMMENTS
1	2		6	7	8	14	15	16	32		
	Symbol			VFD						One or more subfields in the variable field	
	or									separated by commas.	
	blanks										

The VFD pseudo-operation is used for generation of data where it is essential to define the data word in terms of individual bits. It is used to specify by bit count certain information to be packed into words.

In considering the definition of a subfield, it is understood that the unit of information is a single bit (in contrast with the unit of information in the BCI pseudo-operation which is six bits). Each VFD subfield is one of three types: an algebraic expression, a Boolean expression, or alphanumeric. Each subfield contains a conversion type indicator and a bit count, the maximum value of which is 36. The bit count is an unsigned integer which defines the length of the subfield; it is separated from the data subfield by a slash (/). If the bit count is immediately preceded by an O or H, the variable-length data subfield is either Boolean or alphanumeric, respectively. In the absence of both the type indicators, O and H, the data subfield is an algebraic field. A Boolean subfield contains an expression that is evaluated using the Boolean operators (*,/ ,+,-).

The data subfield is evaluated according to its form: algebraic, Boolean, or alphanumeric. A 36-bit field results. The low-order n bits of the algebraic or Boolean expression determine the resultant field value; whereas for the alphanumeric subfield the high-order n bits are used.

If the required subfields cannot be contained on one card, they may be continued by the use of the ETC pseudo-operation. This is done by terminating the variable field of the VFD pseudo-operation with a comma. The next subfield is then given as the beginning expression in the variable field of an ETC card. If necessary, subsequent subfields may be continued onto following ETC cards in the same manner. The scanning of the variable field is terminated upon encountering the first blank character.

The VFD may generate more than one machine word; if the sum of the bit counts is not a multiple of a discrete machine word, the last partial string of bits will be left-justified and the word completed with zeros.

EXAMPLES OF VFD PSEUDO-OPERATION

Assume one would like to have the address ALPHA packed in the first 18 bits of a word, decimal 3 in the next 6 bits, the literal letter B in the next 6 bits, and an octal 77 in the last 6 bits. One could easily define it as follows:

```
VFD 18/ALPHA,6/3,H6/B,O6/77
```

With the location counter at 1053 and the location 731₈ assigned for ALPHA, this would print out (without column headings):

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>		
001053	000731032277	000	VFD	18/ALPHA,6/3,H6/B,O6/77

NOTE: Relocation digits 000 refer to binary code data for A, BC, and DE of the relocation scheme. (Page III- 84 and following of this chapter.)

If ALPHA had been a program relocatable element, the relocation bits would have been 010; that is, the relocation scheme would have specified the left half of the word as containing a relocatable address. The relocation is only assigned if the programmer specifies a field width of 18 bits and has it left- or right-justified; in all other cases the fields are considered absolute. The total number of bits under a VFD need not be a multiple of full words nor is the total field (sum of all subfields) restricted to one word. The total field width, however, for a single subfield is 36 bits.

Consider a program situation where one wishes to generate a three-word identifier for a table. Assume n is the word length of the table and is equal to 12. You wish to place twice the length of the table in the first 12 bits, the name of the table in the next 60 bits, the location of the table (where TABLE is a program relocatable symbol equal to 2351₈) in the next 18 bits, zero in the next 8 bits, and -1 in the next 6 bits--all in a three-word key.

With the location counter at 1054,

VFD 12/2*12,H36/PRESSU,H24/RE,18/TABLE,8/,6/-1

will generate

<u>Location</u>	<u>Contents</u>	<u>Relocation</u>		
001054	003047512562	000	VFD	12/2*12,H36/PRESSU,H24/RE, 18/TABLE,8/,6/-1
001055	626451252020	000		
001056	002351001760	010		

where 010 specifies the relocatability of TABLE.

DUP (Duplicate Cards)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	8	14	15	16
	Symbol		DUP					Two subfields in the variable field, separated
	or							by a comma
	blanks							

The DUP pseudo-operation provides the programmer with an easy means of generating tables and/or data. It causes the Assembler to duplicate a sequence (range) of instructions or pseudo-operations a specified number of times.

The first subfield in the variable field is an absolute expression which defines the count. The value of the count field specifies the number of cards, following the DUP pseudo-operation, that are included in the group to be duplicated. The value in the count field must be a decimal integer less than or equal to ten.

The second subfield of the pseudo-operation is an absolute expression which specifies the number of iterations. The value in the iteration field specifies the number of times the group of cards, following the DUP pseudo-operation, is to be duplicated. This value can be any positive integer less than $2^{18}-1$. The groups of duplicated cards appear in the assembled listing immediately behind the original group.

If either the count field or the iteration field contains 0 (zero) or is null, the DUP pseudo-operation will be ignored.

If a symbol appears in the location field of the pseudo-operation, it is given the address of the next location to be assigned by the Assembler.

If an odd/even address is specified for an instruction within the range of a DUP pseudo-operation, the instruction will be placed in odd/even address and a filler used when needed. The filler will be an NOP instruction.

All symbols appearing in the variable field of the DUP pseudo-operation must have been previously defined. Any symbols appearing in the location field of the instructions being duplicated are defined only on the first iteration, thus avoiding multiply-defined symbols. SET would of course be the exception to this rule.

The only instructions or pseudo-operations which may not appear in the range of a DUP instruction are END, MACRO, and DUP. ETC may not appear as the first card after the range of a DUP.

Storage Allocation Pseudo-Operations

These pseudo-operations are used to reserve specified core memory storage areas within the coding sequence of a program for use as storage areas or work areas.

BSS (Block Started by Symbol)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS
1 2	6 7 8		14	15 16			
Symbol			BSS				A permissible expression in the variable
or							field defines the amount of storage to be
blanks							reserved.

The BSS pseudo-operation is used by the programmer to reserve an area of memory within his assembled program for working and for data storage. The variable field contains an expression that specifies the number of locations the Assembler must reserve in the program.

If a symbol is entered in the location field, it is assigned the value of the first location in the block of reserved storage. If the expression in the variable field contains symbols, they must have been previously defined and must yield an absolute result. No binary cards are generated by this pseudo-operation.

BFS (Block Followed by Symbol)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS
1 2	6 7 8		14	15 16			
Symbol			BFS				A permissible expression in the variable
or							field defines the amount of storage to be
blanks							reserved

The BFS pseudo-operation is identical to BSS with one exception. If a symbol appears in the location field, it is assigned the value of the first location after the block of reserved storage has been assigned.

BLOCK (Block Common)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	14	15	16	32
	Blanks			BLOCK				A symbol in the variable field

The purpose of the BLOCK pseudo-operation is to specify that program data following the BLOCK entry is to be assembled in the LABELED COMMON region of the user program under the symbol appearing in the variable field. BLOCK is, in effect, another location counter external to the text of the program.

The symbol in the variable field specifies the label of the COMMON area to be assembled. If the variable field is left blank, the normal FORTRAN BLANK COMMON is specified; and data following the BLOCK pseudo-operation will be assembled relative to the unlabeled (BLANK COMMON) memory area of the user program. It is not possible to assemble data or instructions into BLANK COMMON. Storage labeling and reservation is all that is permitted.

The pseudo-operations which take the program out of BLOCK mode and into some other mode are:

- 1. BLOCK (for some other LABELED COMMON)
- 2. USE
- 3. ORG/LOC, where the value of the expression is relocatable
- 4. END

It should be noted that BLOCK does not cause the Assembler to make the current USE location counter PREVIOUS. As such, a USE PREVIOUS following a BLOCK will cause the location counter which was in effect prior to the last USE to be invoked.

LIT (Literal Pool Origin)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7	14	15	16	32
	Symbol or blanks			LIT				Column 16 must be blank

The LIT pseudo-operation causes the Assembler to punch and print out all the previously developed literals. If the LIT instruction occurs in the middle of the program, the literals up to that point are output and printed out starting with the first available location after LIT; the literal pool is reinitialized as if the assembly had just begun.

If there are literals remaining in the pool when the END card is encountered, the origin of the literal pool will be one location past the final word defined by the program. The maximum number of LIT pseudo-operations that can occur in a program is 63.

Conditional Pseudo-Operations

The pseudo-operations, INE, IFE, IFL, and IFG which follow are especially useful within MACRO prototypes to gain additional flexibility in variable-length or conditional expansion of the MACRO prototype. Their use, however, is not limited to MACROS: they can be employed elsewhere in coding a subprogram to effect conditional assembly of segments of the program.

The programmer is responsible for avoiding noncomparable elements within these pseudo-operations. In addition, symbols used in the variable field will normally have been previously defined. On the other hand, one of the primary uses of conditionals is to test whether or not a symbol has been defined at a given point in an assembly. Consequently, undefined symbols within a conditional are not flagged in the left margin of the listing. However, if the symbol is never defined within the assembly, the symbol will be listed as undefined at the end of the listing; if the symbol is defined later in the assembly, it is not listed as undefined.

INE (If Not Equal)

[illegible]

The INE pseudo-operation provides for conditional assembly of the next n instructions, depending on the relationship of the first two subfields of the variable field.

The value of the expression in the first subfield is compared to the value of the expression in the second subfield. If they are not equivalent, the next n cards are assembled, where n is specified in the third subfield; otherwise, the next n cards are bypassed, resumption beginning at the (n+1)th card. If the third subfield is not present, n is assumed to be one.

Two types of comparisons are possible in the subfields of the INE pseudo-operation. The first is a straight numeric comparison after the expression has been evaluated. The second is alphanumeric comparison and the relation is the collating sequence. Alphanumeric strings in the variable field of INE are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric string, the other will automatically be classified as such.

IFE (If Equal)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	Blanks			IFE					Two or three subfields in the variable field

The IFE pseudo-operation provides for conditional assembly of the next n cards depending on the relationship of the first two subfields of the variable field. The next n cards are assembled if and only if the expression or alphanumeric string in the first subfield is equal to the expression or alphanumeric string in the second subfield. The n is specified in the third subfield and assumed to be one if not present. If the compared subfields are not equal, the next n cards are bypassed.

Alphanumeric strings in the variable field of IFE are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric string, the other will automatically be classified as such.

IFL (If Less Than)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	Blanks			IFL					Two or three subfields in the variable field

The IFL pseudo-operation provides for conditional assembly of the next n cards depending on the value of the first two subfields of the variable field. The next n cards are assembled if the expression or alphanumeric string in the first subfield is algebraically less than the expression or alphanumeric string in the second subfield; otherwise, the next n cards are bypassed. The n is specified in the third subfield and assumed to be one if not present. Alphanumeric strings in the variable field of IFL are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric string, the other will automatically be classified as such.

IFG (If Greater Than)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1 2	6 7 8		14 15 16	32				
	Blanks		IFG				Two or three subfields in the variable field	

The IFG pseudo-operation provides for conditional assembly of the next n cards depending on the value of the first two subfields of the variable field. The next n cards are assembled if the expression or alphanumeric string in the first subfield is algebraically greater than the expression or alphanumeric string in the second subfield; otherwise, the next n cards are by-passed. Then is specified in the third subfield and assumed to be one if not present. Alphanumeric strings in the variable field of IFG are denoted by placing the subfield within apostrophe marks. If either the first or second subfield is designated as an alphanumeric string, the other will automatically be classified as such.

Special Word Formats

ARG A, M (Argument--Generate Zero Operation Code Computer Word)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1 2	6 7 8		14 15 16			32		
Symbol			ARG				Two subfields in the variable field	

The use of ARG in the operation field causes the Assembler to generate a binary word with bit configuration in the general instruction format. The operation code 000 is placed in the operation field. The variable field is interpreted in the same manner as a standard machine instruction.

NONOP (Undefined Operation)

When an undefined operation is encountered, NONOP is looked up in the operation table and used in place of the undefined operation. NONOP is initially set as an error routine, but the programmer through the use of OPD, OPSYN or MACRO may redefine NONOP to his own purpose. For example, NONOP could be redefined by the use of a MACRO to be a MME to GECHEK with a dump sequence, or it could be equivocated with the ARG pseudo-operation.

ZERO B, C (Generate One Word With Two Specified 18-bit Fields)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER	COMMENTS
1 2	6 7 8		14 15 16	32		
Symbol	ZERO				Two subfields in the variable field	
or						
blanks						

The pseudo-operation ZERO is provided primarily for the definition of values to be stored in either or both the high- or low-order 18-bit halves of a word. The Assembler will generate the binary word divided into the two 18-bit halves; bit positions 0-17 and 18-35. The equivalent binary value of the expression in the first subfield will be in bit positions 0-17. The equivalent binary value of the expression in the second subfield will be in bit positions 18-35. Literals are not allowed in the variable field of the ZERO pseudo-operation.

MAXSZ (Maximum Size of Assembly)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS		
1	2		6	7	8	14	15	16	32
	Blank				MAXSZ				A decimal number in the variable field

The decimal number represents the programmer's estimate of the largest number of assembled instructions and data in his program or subprogram. The variable field number is evaluated, saved, and printed out at the end of the assembly listing. It can then be compared with the actual size of the assembly.

MAXSZ is provided as a programmer convenience and can be inserted anywhere in his coding.

Address Tally Pseudo-Operations

The Indirect then Tally (IT) type of address modification in several cases requires special word formats which are not instructions and do not follow the standard word format. The following pseudo-operations are for this purpose. (Refer to page III-20 and following.)

- TALLY A,T,C (Tally) Used for ID, DI, SC, and CI type of tally modification, where SC and CI are for 6 bit characters. The first subfield is the address for the indirect reference, T is the tally count, and C is the character position ($0 \leq C \leq 5$). When used with the CI modifier the contents of the tally count subfield (T) is not interpreted.
- TALLYB A,T,B (Tally Byte) Used for SC and CI type of tally modification, where 9 bit bytes (characters) are desired. A and T are the same as for TALLY and B indicates the byte position ($0 \leq B \leq 3$).
- TALLYD A,T,D, (Tally and Delta) Used for Add Delta (AD) and Subtract Delta (SD) modification. A is the address, T the tally, and D the delta of incrementing.
- TALLYC A,T,mod (Tally and Continue) Used for Address, Tally, and Continue. A is the address, T the tally count, and mod the address modification as specified under normal instructions.

Repeat Instruction Coding Formats

The machine instructions Repeat (RPT), Repeat Double (RPD), and Repeat Link (RPL) use special formats and have special tally, terminate repeat, and other conditions associated with them. (See page II-125 and following.) The Assembler coding formats for the several RPT, RPD, and RPL options follow.

- RPT N,I,k1,k2,.....,kj The command generated by the Assembler from this format will cause the instruction immediately following the command to be iterated N times and the increment value for each iteration set to I. The range for N is 0-255. If N=0, the instruction will be iterated 256 times. The fields k1,k2,.....,kj may or may not be present. They are conditions for termination. These fields may contain the allowable codes of TOV, TNC, TRC, TMI, TPL, TZE, and TNZ.

It is also possible to use an octal number rather than the special symbols to denote termination conditions. Thus if field k1 is found to be numeric, it will be interpreted as octal; the low-order seven bits will be ORed into positions 11-17 of the instruction. The variable field scan will be terminated with the octal field.

- RPTX ,I This instruction behaves just as the RPT instruction with the exception that N and the conditions of termination will be found in index register zero instead of imbedded in the instruction.

- RPD N,I,k1,k2,.....,kj The command generated by the Assembler from this format will cause the two instructions immediately following the RPD instruction to be iterated N times and the increment value for each iteration set to I. The increment I will apply to both instructions being repeated.

The variables k1,.....,kj are identical to those explained in the RPT instruction. Since the double repeat must fall in an odd location, the Assembler will force this condition and use an NOP instruction for a filler when needed.

- RPDX ,I This instruction behaves just as the RPD instruction with the exception that N and the conditions of termination will be found in index register zero instead of imbedded in the instruction.

- RPDA N,I,k1,k2,.....,kj This is the same as the RPD instruction except that only the address of the first instruction following the RPDA instruction will be incremented on each iteration by I.

- RPDB N,I,k1,k2,.....,kj This is the same as the RPD instruction except that only the address of the second instruction following the RPDB instruction will be incremented by I on each iteration.

- RPL N,k1,k2,.....,kj This format will cause the instruction immediately following it to be repeated N times or until one of the conditions specified in k1,.....,kj are satisfied. The relation of k1,.....,kj is the same as in RPT. The address effectively used by the repeated instruction is the linked address (described on page II-129 and following).

- **RPLX** This instruction behaves just as the RPL instruction except that N and conditions of termination will be found in index register zero instead of imbedded in the instruction.

MACRO OPERATIONS

Introduction

Programming applications frequently involve (1) the coding of a repeated pattern of instructions that within themselves contain variable entries at each iteration of the pattern and (2) basic coding patterns subject to conditional assembly at each occurrence. The macro operation gives the programmer a shorthand notation for handling (1) and (2) through the use of a special type of pseudo-operation referred to in the GE-625/635 Macro Assembler as a MACRO. Having once determined the iterated pattern, the programmer can, within the MACRO, designate selectable fields of any instruction of the pattern as variable. Thereafter, by coding a single MACRO instruction, he can use the entire pattern as many times as needed, substituting different parameters for the selected subfields on each use.

When he defines the iterated pattern, the programmer gives it a name, and this name then becomes the operation code of the MACRO instruction by which he subsequently uses the macro operation.

As a generative operation, the macro operation causes n card images (where n is normally greater than one) to be generated; these may have substitutable arguments. The MACRO is known as the prototype or skeleton, and the card images that may be defined are relatively unrestricted as to type.

They can be:

1. Any Processor instruction
2. Most Assembler pseudo-operations
3. Any previously defined macro operation

Card images of these types are subject to the same conditions and restrictions when generated by the macro processor as though they had been produced directly by the programmer as in-line coding.

To use the MACRO prototype, once named, the programmer enters the macro operation code in the operation field and arguments in the variable field of the MACRO instruction. (The arguments comprise variable field subfields and refer directly to the argument pointers specified in the fields of the card images of the prototype.) By suitably selecting the arguments in relation to their use in the prototype, the programmer causes the Assembler to produce in-line coding variations of the n card images defined within the prototype.

The effect of a macro operation is the same as an open subroutine in that it produces in-line code to perform a predefined function. The in-line code is inserted in the normal flow of the program so that the generated instructions are executed in-line with the rest of the program each time the macro operation is used.

An important feature in specifying a prototype is the use of macro operations within a given prototype. The Assembler processes such "nested" macro operations at expansion time only. The nesting of one macro definition within another prototype is not permitted. If macro operation codes are arguments, they must be used in the operation field for recognition. Thus, the MACRO must be defined before its appearance as an argument; that is, the prototype must be available to the Assembler before encountering a demand for its usage.

Definition of the Prototype

The definition of a MACRO prototype is made up of three parts:

- 1. Creation of a heading card that assigns the prototype a name
- 2. Generation of the prototype body of n card images with their substitutable arguments
- 3. Creation of a prototype termination card

These parts are described in the following three subparagraphs.

MACRO (MACRO Identification)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7				
	Symbol			MACRO				Blanks in the variable field

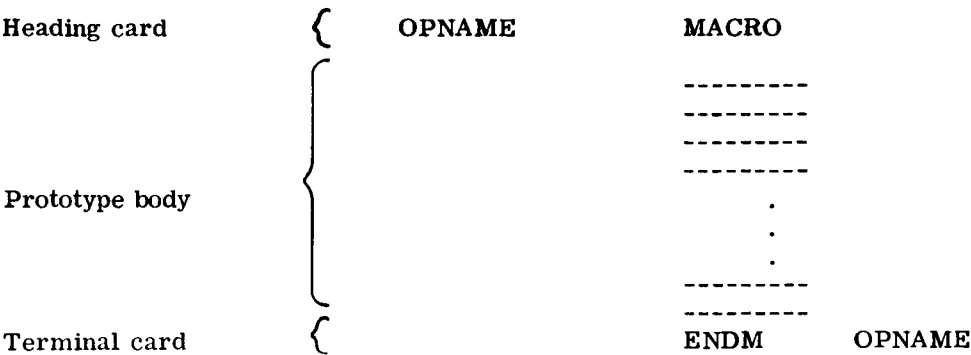
The MACRO pseudo-operation is used to define a macro operation by symbolic name. The symbol in the location field conforms to standard symbol formation rules and defines the name of a MACRO whose prototype is given on the next n lines. (The prototype definition continues until the Assembler encounters the proper ENDM pseudo-operation.) The name of the MACRO is a required entry. If the symbol is identical to an operation code already in the table, then the macro operation will be used as a new definition for that operation code. It is entered in the Assembler operation table with a pointer to its associated prototype that is entered in the MACRO skeleton table.

ENDM (End MACRO)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1	2		6	7				
	Blanks			ENDM				A symbol in the variable field

The symbol in the variable field is the symbolic name of the MACRO instruction as defined in the location field of the corresponding MACRO heading card. Every MACRO prototype must contain both the terminal ENDM pseudo-operation and the MACRO pseudo-operation.

Thus, every prototype will have the form



where OPNAME represents the prototype name that is placed in the Assembler operation table.

- Prototype Body. The prototype body contains a sequence of standard source-card images (of the types listed earlier) that otherwise would be repeated frequently in the source program. Thus, for example, if the iterated coding pattern

LOCATION		E O	OPERATION	ADDRESS, MODIFIER		COMMENTS
1	2			14	15	
6	7	8		14	15	32
			:			
			LDA	5,	DL	
			LDQ	13,	DL	
			CWL	ALPHA,	2	
			TZE	FIRST		
			:			
			:			
			:			
			LDA	U		
			LDQ	V		
			CWL	BETA,	4	
			TZE	SCND		
			:			
			:			
			:			
			LDA	W+X		
			LDQ	Y+Z		
			CWL	GAMMA		
			TZE	NEXT1		

appeared in a subprogram, it could be represented by the following prototype body (preceded by the required prototype name):

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS	
1 2	6 7 8		14 15 16			32		
CMPAR			MACRO				MACRO prototype with substitutable	
			LDA		#1		arguments in the variable field	
			LDQ		#2			
			CWL		#3			
			TZE		#4			
			ENDM		CMPAR			

Then the previous coding examples could be represented by the macro operation **CMPAR** as follows:

```
CMPAR          (5,DL),(13,DL),(ALPHA,2),FIRST
---
---
CMPAR          U,V,(BETA,4),SCND
---
---
CMPAR          W+X,Y+Z,GAMMA,NEXT1
```

The Assembler recognizes substitutable arguments by the presence of the number-sign identifier (#). Having sensed this identifier, it examines the next one or two digits. (Sixty-three is the maximum number of arguments usable in a single prototype.)

MACRO prototype arguments can appear in the location field, in the operation field, in the variable field, and coincidentally in combinations of these fields within a single card image. Substitutions that can be made in these fields are:

- 1. Location field--any permissible location symbol (see comments below)
- 2. Operation field--all machine instructions, all pseudo-operations (except the MACRO pseudo-operation) and previously defined macro operations
- 3. Variable field--any allowable expression followed by an admissible modifier tag and separated from the expression by a delimiting comma.

In general, anything appearing to the right of the first blank in the variable field will not be copied into the generated card image. For example, a substitutable argument appearing in the comments field of a card image--that is, separated from the variable field by one or more blanks--will not be interpreted by the Assembler (except in the case of the BCI, REM, TTL, and TTLS pseudo-operations). This means that only pertinent information in the location, operation, and variable fields is recognized, that internal blanks are not allowed in these fields, and that the first blank in these fields causes field termination.

When specifying a symbol in a location field of an instruction within a prototype the programmer must be aware that this MACRO can be used only once since on the second use the same symbol will be redefined, causing a multiply-defined symbol. Consequently, the use of location symbols within the prototype is discouraged. Alternatively, for cases where repeated use of a prototype is necessary, two techniques are available: (1) use of Created Symbols and (2) placement of substitutable argument in the location field and use of a unique symbol in the argument of the macro operation each time the prototype is used. These techniques are described under Using a MACRO operation, below.

The location field, operation field, and variable field may contain text and arguments which can be concatenated (linked together) by simply entering the substitutable argument (for example, AB#3) directly in the text with no blanks or special symbols preceding or following the entry. Concatenation is especially useful in the operation field and in the partial subfields of the variable field. (Refer to the discussion of BCI, REM, TTL, and TTLS immediately following.) As an example of the first use, consider a machine instruction such as LD(R) where R can assume the designators A, Q, AQ, and X0-X7.

The prototype NAME

NAME	MACRO

	LD#2

	A,#1

contains a partial operation field argument; and when the in-line coding is generated, LD#2 becomes LDA, LDQ, etc., as designated by the argument used in the macro operation.

The BCI, REM, TTL, and TTLS pseudo-operations used within the prototype are scanned in full for substitutable arguments. The variable field of these pseudo-operations can contain blanks and argument pointers. The following illustrates a typical use:

ALPHA	MACRO

NOTE#1	REM IGNORE#2ERRORS#ON#3

An asterisk (*) type comment card cannot appear in a MACRO prototype.

Using a MACRO Operation

Use of a macro operation can be divided into two basic parts; definition of the prototype and writing the macro operation. The first part has been described on the preceding pages; writing the macro operation to call upon the prototype is the process of using the MACRO and is described in the following paragraphs.

The macro operation card is made up of two basic fields; the operation field that contains the name of the prototype being referenced and the variable field that contains subfield arguments relating to the argument pointers of the prototype on a sequential, one-to-one basis. For example, the defined prototype CMPAR, mentioned earlier, could be called for expansion by the MACRO instruction

CMPAR U,V,(BETA,4),SCND

where the variable field arguments, separated by commas and taken left-to-right, correspond with the prototype pointers #1 through #4. These arguments are then substituted in their corresponding positions of the prototype to produce a sequence of instructions using these arguments in the assigned location, operation, and variable fields of the prototype body. (The above MACRO instruction expands to the coding shown on page III-65.)

The maximum number of MACRO call arguments is 63; arguments greater than 63 are treated modulo 64. For example, the 70th argument is the same as the 6th argument and would be so recognized by the Assembler. Each such argument can be a literal, a symbol, or an expression (delimited by commas) that conforms to the restrictions imposed upon the field of the machine instruction or pseudo-operation within the prototype where the argument will be inserted.

The following conditions and restrictions apply to the expansion of MACROs:

1. Anything appearing in the location field of a prototype card image, whether text or a substitutable argument, causes generation to begin in column 1 for that text or argument.
2. Location field text generated from an argument pointer (in a prototype location field) so as to produce a resultant field extending beyond column 8 causes the operation field to begin in the next position after the generated text. Normally, the operation field will begin in column 8.
3. Operation field text generated from an argument pointer (in a prototype operation field) so as to produce a resultant field extending beyond column 16 causes the variable field to start in the next position after the generated text. Normally, the variable field will begin in column 16.
4. The variable field may begin after the first blank that terminates the operation field but not later than column 16 in the absence of the condition in 3 above.
5. No generated card image can have more than 72 characters recorded; that is, the capacity of one card image cannot be exceeded (columns 73-80 are not part of the card image).
6. No argument string of alphanumeric characters can exceed 57 characters.
7. Up to 63 levels of MACRO nesting are permitted.

An argument can also be declared null by the programmer when writing the MACRO instruction; however, it must be declared explicitly null. Explicitly null arguments of the MACRO instruction argument list can be specified in either of two ways; by writing the delimiting commas in succession with no spaces between the delimiters or by terminating the argument list with a comma with the next normal argument of the list omitted. (Refer to the CRSM description, following.) A null argument means that no characters will be inserted in the generated card image wherever the argument is referenced. When a macro operation argument relates to an argument pointer and the pointer requires the argument to have multiple entries or contains blanks, the corresponding argument must be enclosed within parentheses with the parenthetical argument set off by the normal comma delimiters. The parenthetical argument can contain commas as separators.

Examples of prototype card images that require the use of parentheses in the MACRO call are pseudo-operations such as IDRP, VFD, BCI, and REM, as well as the variable field of an instruction where the address and tag may be one argument.

It is also possible to enclose an argument within brackets, making them subarguments, in which case blanks are ignored as part of the argument. For example the MACRO call of the MACRO named ABC can be written as

```
ABC  [A,  
ETC  24,  
ETC  2*D]
```

and is equivalent to

```
ABC  (A, 24, 2*D)
```

even though numerous blanks occur after the arguments A, and 24,. Thus, the Assembler packs everything it finds within brackets and suppresses all blanks therein. The above manner of writing the MACRO call permits the programmer additional flexibility in placing one subargument per card by means of using ETC, the blanks no longer being significant.

It can happen that the argument list of a macro operation extends beyond the capacity of one card. In this case, the ETC pseudo-operation is used to extend the list on to the next card. In using ETC, the last argument entry of the macro operation is delimited by a following comma, and the first entry of the ETC card is the next argument in the list. Within the prototype, as many ETC cards as required can be used for internal MACROs or VFD pseudo-operations.

Pseudo-Operations Used Within Prototypes

- Need for Prototype Created Symbols. In case of a MACRO prototype in which an argument pointer is used in the location field, the programmer must specify a new symbol each time the prototype is called. In addition, for those cases where a nonsubstitutable symbol is used in a prototype location field, the programmer can use the macro operation only once without incurring an Assembler error flag on the second and all subsequent calls to the prototype (multiply-defined symbol). Primarily to avoid the former task (having to repeatedly define new symbols on using the macro operation) and to enable repeated use of a prototype with a location field symbol (nonsubstitutable), the created symbol concept is provided.

- Use of Created Symbols. Created symbols are of the type .xxx, where xxx runs from 001 through 999, thus making possible up to 999 created symbols for an assembly. The periods are part of the symbol. The Assembler will generate a created symbol only if an argument in the macro operation is implicitly null; that is, only if the macro operation defines fewer arguments than given in the related MACRO prototype or if the designator # is used as an argument. Explicitly null arguments will not cause created symbols to be generated. The example given clarifies these ideas.

Assume a MACRO prototype of the form

NAME	MACRO

	#1, #2
#4	-----
	X
#5	-----
	ALPHA, #3

	#4
	TMI
	#5
	ENDM
	NAME

with five arguments, 1 through 5. The macro operation NAME in the form

NAME A,7,,B

specifies the third and fourth arguments as explicitly null; consequently, no created symbols would be provided. The expansion of the operation would be

	-----	A,7
	-----	X
B	-----	ALPHA,

	TMI	B

The macro operation card

NAME A,7,

indicates the third argument is explicitly null, while arguments four and five are implicitly null. Consequently, created symbols would be provided for arguments four and five but not for three. This is shown in the expansion of the macro operation as follows:

	-----	A,7
.011.	-----	X
.012.	-----	ALPHA,
	-----	.011.
	TMI	.012.

A created symbol could be requested for argument three simply by omitting the last comma. The programmer can conveniently change an explicitly null argument to an implicitly null one by inserting the # designator in an explicitly null position. Thus, for the preceding example

NAME A,7, #,B

the fourth argument becomes implicitly null and a created symbol will be generated.

CRSM ON/OFF (Created Symbols)

LOCATION		E O	OPERATION			ADDRESS, MODIFIER		COMMENTS
1	2		6	7	8	14	15 16	
	Blanks			CRSM			ON	Normal mode

Created symbols are generated only within MACRO prototypes. They can be generated for argument pointers in the location, operation, and variable fields of instructions or pseudo-operations that use symbols. Accordingly, the created symbols pseudo-operation affects only such coding as is produced by the expansion of MACROs. CRSM ON causes the Assembler to initiate or resume the creation of symbols; CRSM OFF terminates the symbol creation if CRSM ON was previously in effect. If the Assembler is already in the specified mode, the pseudo-operation is ignored.

ORGCSM (Origin Created Symbols)

Blanks ORGCSM One expression in the variable field.

The variable field is evaluated and becomes the new starting value between the decimal points of the created symbols.

IDRP (Indefinite Repeat)

LOCATION		E O	OPERATION			ADDRESS, MODIFIER		COMMENTS
1	2		6	7	8	14	15 16	
	Blanks			IDRP			#3	An argument number or blanks in the variable field, depending on the IDRP of the IDRP pair

The purpose of the IDRP is to provide an iteration capability within the range of the MACRO prototype by letting the number of grouped variables in an argument pointer determine the iteration count.

The IDRP pseudo-operation must occur in pairs, thus delimiting the range of the iteration within the MACRO prototype. The variable field of the first IDRP must contain the argument number that points to the particular argument used to determine the iteration count and the variables to be affected. The variable field of the second IDRP must be blank.

At expansion time, the programmer denotes the grouping of the variables (subarguments) of the iteration by placing them, contained in parentheses, as the nth argument where n was the argument value contained in the initial IDRP variable field entry.

IDRP is limited to use within the MACRO prototype, and nesting is not permitted. However, as many disjoint IDRP pairs may occur in one MACRO as the programmer wishes.

For example, given the MACRO skeleton

```
NAME          MACRO
      .
      .
      IDRP          #2
      ADA          #2
      IDRP
      .
      .
      ENDM          NAME
```

the MACRO call (with variables X1, X2, and X3)

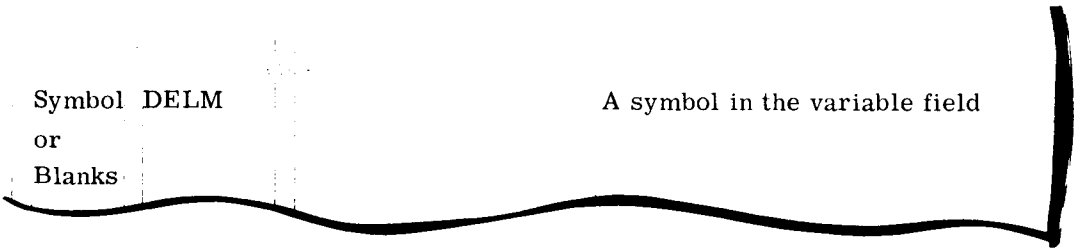
```
A          NAME  Q+2, (X1, X2, X3), B
```

would generate

```
A          .
          .
          .
          ADA      X1
          ADA      X2
          ADA      X3
          .
          .
```

In the example, arguments #1 and #3, Q+2, and B respectively, are used in the skeleton ahead of and after the appearance of the IDRP, range-iteration pair.

DELM (Delete MACRO)



The function of this pseudo-operation is to delete the MACRO named in the variable field from the MACRO prototype area, and disable its corresponding operation table entry. Through the use of this pseudo-operation, systems which require many, or large MACRO prototypes, or which have minimal storage allocation at assembly time, can re-use storage in the prototype area for redefining or defining new MACROs. Redefinition of a deleted MACRO will not produce an M multiple defined flag on the assembly listing.

Implementation of System MACROs

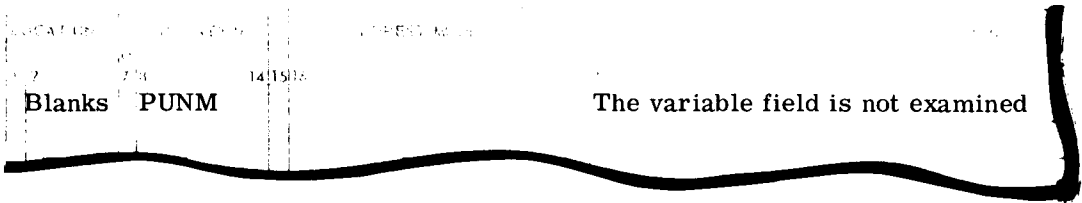
GMAP has been implemented with the facility for loading a unique set (or sets) of MACROs, under control of a pseudo-operation. This permits the various language processors to uniquely identify those standard system MACROs that are required for the assembly of their generated code.

GMAP itself has a set of system MACROs which it loads as part of its initialization procedures. This includes FILCB, the GEFRC File Control Block MACRO, SORT, DUAL, and MERGE (cf. GE-600 Series SORT/MERGE Reference Manual CPB-1005) and the DEBUG Symbol Table MACROs VTAB and LTAB. This action is dependent on the elected option on the \$ GMAP control card. The option GMAC/NGMAC instructs GMAP to load or not load its own system MACRO's in initializing for assembly. The absence of either option is equivalent to having elected GMAC, hence the normal user of GMAP does not need to be aware of the fact that GMAPS MACROs are optionally loaded.

System MACROs are, by definition, located on the System File on the high speed drum. They are put there by the System Editor, in System Loadable Format, as a free-standing system program. Their catalog name is that which is to be used by GMAP in the loading operation. For proper implementation, the MASTER option of the System Editor parameters card must be elected. It may be in absolute or relocatable System Loadable Format.

This implementation technique permits any unit, or functionally related group of users of GMAP to define and implement a unique set of System MACROs; or on a larger scale, it allows various GE-600 installations to install local standard sets of MACROs, without changing the Assembler.

PUNM (Punch MACRO Prototypes and Controls



This pseudo-operation causes the Assembler, in pass one, to scan the operation table for all MACROs defined. It then appends their definitions to the end of the prototype table and constructs a control word specifying the length of this area and the number of MACROs defined therein.

At the beginning of pass two, this information is punched onto relocatable binary instruction cards, along with \$ OBJECT, preface, and \$ DKEND cards. The primary SYMDEF of this deck will arbitrarily be .MACR. .

In the normal preparation of System MACROs, it would not be desirable to include the GMAP System MACROs. For this reason, the assembly of a set of System MACROs should have NGMAC elected on its \$ GMAP card.

LODM (Load System MACROs)

LOCATION	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32			
Blanks	LODM		A symbol in the variable field

This pseudo-operation causes the Assembler to issue an MME GECALL for a set of System MACROs. The name used in the GECALL sequence is the symbol taken from the variable field of the LODM pseudo-operation. MACROs thus loaded will be appended to (not overlay) the MACRO prototype table. They will be defined and made available for immediate use. If a MACRO is redefined by this operation the LODM instruction will be flagged with an M.

Notes and Examples on Defining a Prototype

The examples following show some of the ways in which MACROs can be used.

● **Field Substitution**

Prototype definition:

ADDTO	MACRO	
	LDA	#1
	ADA	#2
	STA	#3
	ENDM	ADDTO

Use:

ADDTO	A,(1,DL),B+5
-------	--------------

● **Concatenation of Text and Arguments**

Prototype definition:

INCX	MACRO	
	ADLX#2	#3,DU
	INE	#1,'*+1'
	TRA	#1
	ENDM	INCX

Use:

INCX	LOCA,4,1
------	----------

or

INCX	*+1,4,1
------	---------

● **Argument in a BCI Pseudo-Operation**

Prototype definition:

ERROR	MACRO	
	TSX1	DIAG
	ARG	#1
	BCI	5,ERROR,b#1,bCONDITION,bIGNORED
	ENDM	ERROR

● MACRO Operation in a Prototype

TEST	MACRO	
	LDA	#1
	CMPLA	#2
	#3	#4
	ERROR	#5
	ENDM	TEST

- Indefinite Repeat

SYMGEN	MACRO	
#1	IDRP	#1
	BCI	1,#1
	IDRP	
	ENDM	SYMGEN

- Subroutine Call MACRO

DOO	MACRO	
K	SET	0
	IDRP	#2
K	SET	K+1
	IDRP	
	TSX1	#1
	TRA	*+1+K
	IDRP	#2
	ARG	#2
	IDRP	
	ENDM	DOO

GE-600 SERIES

PROGRAM LINKAGE PSEUDO-OPERATIONS

CALL (Call--Subroutines)

LOCATION		E O	OPERATION		ADDRESS, MODIFIER		COMMENTS
1 2	6 7		8	14 15	16	32	
Symbol			CALL				Subfields in the variable field with contents and delimiters as described below
or							
blanks							

The CALL pseudo-operation is used to generate the standard subroutine calling sequence.

The first subfield in the variable field of the instruction is separated from the next n subfields by a left parenthesis. This subfield contains the symbol which identifies the subroutine being called. It is possible to modify this symbol by separating the symbol and the modifier with a comma. (In a Relocatable Assembly the symbol entered in this subfield is treated as if it were entered in the variable field of a SYMREF instruction.)

The next n subfields are separated from the first subfield by a left parenthesis and from subfield n+1 by a right parenthesis. Thus the next n subfields are contained in parentheses and are separated from each other by commas. The contents of these subfields are arguments which will be used in the subroutine being called.

The next m subfields are separated from the previous subfields by a right parenthesis and from each other by commas. These subfields are used to define locations for error returns from the subroutine. If no error returns are needed, then m=0.

The last subfield is used to contain an identifier for the instruction. This identifier is used when a trace of the path of the program is made. The identifier may be an expression contained in apostrophes. Thus the last subfield is separated from the previous subfields by an apostrophe. If the last subfield is omitted, the assembly program will provide an identifier.

In the examples following, the calling sequences generated by the pseudo-operation are listed below the CALL pseudo-operation. For clarification AAAAA defines the location the CALL instruction; SUB is the name of the subroutine called; MOD is an address modifier; A1 through An are arguments; E1 through Em define error returns; E.I. is an identifier; and .E.L.. defines a location where error linkage information is stored. The number sequences 1,2,...,n and 1,2,...,m designate argument positions only.

AAAAA	CALL	SUB,MOD(A1,A2,...,An)E1,E2,.....Em'E.I.'
AAAAA	TSX1	SUB,MOD
	TRA	*+2+n+m
	ZERO	.E.L.,E.I.
	ARG	A1
	ARG	A2
	.	
	.	
	ARG	An
	TRA	Em
	.	
	.	
	TRA	E2
	TRA	E1

The preceding example of instructions generated by the CALL pseudo-operation was in the relocatable mode. The following example is in the absolute mode.

AAAAA	CALL	SUB,MOD(A1,A2,.....,An)E1,E2,.....,Em'E.I.'
AAAAA	TSX1	SUB,MOD
	TRA	*+2+n+m
	ZERO	0,E.I.
	ARG	A1
	ARG	A2
	.	
	.	
	ARG	An
	TRA	Em
	.	
	.	
	TRA	E2
	TRA	E1

If the variable field of the CALL cannot be contained on a single line of the coding sheet, it may be continued onto succeeding lines by use of the ETC pseudo-operation. (See page III-54 or III-68.) This is done by terminating the variable field of the CALL instruction with a comma (,). The next subfield is then placed as the first subfield of the ETC pseudo-operation. Subsequent subfields may be continued onto following lines in the same manner.

SAVE (Save--Return Linkage Data)

LOCATION	E	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2	6 7 8	14 15 16	32	
Symbol	SAVE			Blanks or subfields separated by commas in the variable field-- as described below

The SAVE pseudo-operation is used to produce instructions necessary to save specified index registers and the contents of the error linkage index register.

The symbol in the location field of the SAVE instruction is used for referencing by the RETURN instruction. (This symbol is treated by the Assembler as if it had been coded in the variable field of a SYMDEF instruction when the Assembler is in the relocatable mode.)

The subfields in the variable field, if present, will each contain an integer 0-7. Thus each subfield specifies one index register to be saved.

When the SAVE variable field is blank, the following coding is generated:

NAME	TRA	*+2
	RET	.E.L. .
	STI	.E.L. .
	STX1	.E.L. .

The instructions generated by the SAVE pseudo-operation are listed on the following page. The symbols i₁ through i_n are integers 0-7. .E.L. defines the location provided for the contents of the error linkage register.

BBBBB is a symbol that must be present; it is always a primary SYMDEF.

Example one is in the relocatable mode, and example two is in the absolute mode.

EXAMPLE ONE			EXAMPLE TWO		
BBBBB	SAVE	i ₁ , i ₂ , ...i _n	BBBBB	SAVE	i ₁ , i ₂ , ...i _n
BBBBB	TRA	*+2+n	BBBBB	TRA	*+3+n
	LDX(i ₁)	** ,DU		ZERO	
	.			LDX(i ₁)	** ,DU
	.			LDX(i ₂)	** ,DU
	.			.	
	LDX(i _n)	** ,DU		.	
	RET	.E.L..		.	
	STI	.E.L..		.	
	STX1	.E.L..		LDX(i _n)	** ,DU
	STX(i ₁)	BBBBB+1		RET	BBBBB+1
	STX(i ₂)	BBBBB+2		STI	BBBBB+1
	.			STX1	BBBBB+1
	.			STX(i ₁)	BBBBB+2
	STX(i _n)	BBBBB+n		STX(i ₂)	BBBBB+3
				.	
				.	
				STX(i _n)	BBBBB+n+1

RETURN (Return--From Subroutines)

LOCATION		E O	OPERATION	ADDRESS, MODIFIER	COMMENTS
1 2	6 7				
Symbol	8		RETURN	14 15 16	32
or					One or two subfields in the
blanks					variable field

The RETURN pseudo-operation is used for exit from a subroutine. The instructions generated by a RETURN pseudo-operation must make reference to a SAVE instruction within the same subroutine. This is done by the first subfield of RETURN. The first subfield in the variable field must always be present. This subfield must contain a symbol which is defined by its presence in the location field of a SAVE pseudo-operation.

The second subfield is optional and, if present, specifies the particular error return to be made; that is, if the second subfield contains the value k, then the return is made to the kth error return.

In the examples following, the assembled instructions generated by RETURN are listed below the RETURN instruction. For both examples the group of instructions on the left are generated when the Assembler is in the relocatable mode, and the instructions on the right when the Assembler is in the absolute mode.

EXAMPLE ONE

RETURN		BBBBB	
TRA	BBBBB+1	TRA	BBBBB+2
} Generated Instruction		} Generated Instruction	

EXAMPLE TWO

RETURN		BBBBB,k	
LDX1	.E.L.,*	LDX1	BBBBB+1,*
SBX1	k,DU	SBX1	k,DU
STX1	.E.L..	STX1	BBBBB+1
TRA	BBBBB+1	TRA	BBBBB+2
Generated Instructions		Generated Instructions	

ERLK (Error Linkage--to Subroutines)

[illegible]

The normal operation of the Assembler is to assign a location for error linkage information, as referenced by .E.L.. in the examples of the CALL, SAVE, and RETURN pseudo-operations. However, if the programmer wishes to specify the location for error linkage information, he can do so by using ERLK. The appearance of ERLK causes the Assembler to generate two words of the following form:

```
.E.L..      ZERO
          BCI      1,NAME
```

These words will be placed in the assembly at the point the Assembler encountered ERLK. Note that if the programmer has placed all program data under the BLOCK pseudo-operation, he must use ERLK since in this case automatic error linkage is suppressed.

NAME, as selected by the Assembler, will be the first SYMDEF defined in the routine. This may have been accomplished explicitly through use of the SYMDEF pseudo-operation, or implicitly through SAVE.

Error linkage will be generated for all relocatable assemblies, except in the case mentioned above, where all assembling has been relative to BLOCK counters.

NOTE: The symbol .E.L.. may not appear to the right of an EQU pseudo-operation.

SYSTEM (BUILT-IN) SYMBOLS

It is possible to include additional permanently defined system symbols in the Assembler. This is accomplished by a reassembly of the Macro Assembler and by placing the proper information in the required tables.

SOURCE PROGRAM INPUT

Activity Definition

The input job stream managed by the Comprehensive Operating Supervisor (GECOS, GEFLOW module) can comprise assembled object programs, Macro Assembler language source programs, and FORTRAN or COBOL compiler-language source programs. Such programs of a job are referred to as activities. A source program input to the Assembler written in the GE-625/635 machine language is an Assembler language input activity. Comments to follow in this section pertain to this type of activity, as opposed to the others noted above.

The Assembler language activity is composed of the following parts, in order:

- 1. \$ GMAP control card (calls the Assembler into Memory from external storage and provides Assembler output options; refer to the paragraph following)
- 2. Text of the subprogram (one instruction per card)
- 3. END pseudo-operation card (terminates the input subprogram)

The \$ GMAP control card is prepared as shown below:

Card Column	1	8	16
Symbolic Example	\$	GMAP	Option 1, Option 2, ...
Actual Example	\$	GMAP	NDECK,LSTOU,NCOMDK

The operand field specifies the system options listed in any random order. When an option, or its converse, does not appear in the operand field, there is a standard entry which is assumed. (The standard entries are asterisked below.)

The options available with GMAP are as follows:

- LSTOU--A listing of the output will be prepared.*
- NLSTOU--No listing of the output will be prepared.
- DECK--A program deck will be prepared as part of the output of this processor.*
- NDECK--No program deck will be prepared.
- COMDK--A compressed version of the source program will be prepared.
- NCOMDK--No compressed deck will be prepared.*
- GMAC--The GMAP System Macros are required for assembly.*
- NGMAC--The GMAP System Macros must not be used for this assembly.

The content of columns 73-80 is used as an identifier to uniquely identify the binary object programs resulting from the assembly.

Compressed Decks

The Assembler program contains routines and tables for compressing source subprogram cards from a one-instruction-per-card input to a multiple-instruction-per-card input. This Assembler feature is provided primarily for reducing the size of input source decks as concerns handling and correcting (altering) the input subprogram. (For details of the compression and the compressed deck card format, refer to the next paragraph and the GE-625/635 File and Record Control Reference manual CPB-1003.)

The compressed deck (COMDK) option is specified in the operand field of the \$ GMAP control card. The normal mode of Assembler operation is NCOMDK; that is, no compressed deck is produced. To use the Assembler COMDK feature, the \$ GMAP control card would appear as

\$ GMAP COMDK

and be placed as the first card of the deck. When combined with the standard output options, the above control card would cause the Assembler to produce:

1. An output listing containing in its format a complete listing of the source card images (See the listing and symbolic reference table formats, page III-91.)
2. A compressed deck of the source card images, column-binary, alphanumeric.

The COMDEK format is produced by a procedure which compresses any Hollerith-coded card image by removing sequences of 3 or more blanks and packing the information in standard column binary form.

To accomplish the compression, the Hollerith card is considered as being made up of a series of fields and strings. A field is defined as a segment of the card containing no sequences of more than 2 blanks except at the beginning. A string is that portion of a field obtained by deleting any leading blanks.

Each field specification starts with the octal value of A(0-A₆₇) followed by the octal value of B(0-A₆₇) followed by the B characters constituting the string. (A=the number of characters in the field; B=the number of characters in the string.)

The size of A and B is limited, as indicated above, in order to reserve a set of codes to serve as flags when found in a position in which a count had been expected. If a given length exceeds the maximum length, it is segmented into separate fields. For example, given 70 (decimal) consecutive nonblank characters, it is necessary to treat this as two fields with:

Field 1 A = 67,	B = 67 (octal values)
Field 2 A = 17,	B = 17 (octal values)

The field specifications (A,B,string) are packed sequentially on a binary card in the format indicated below. A field specification may be started on a COMDEK card (X) and may be completed on the following card (X+1).

The following codes for A are used to designate specific conditions. The B character is not present in such cases.

A = 0	End of a compressed card; continue decoding on the next card
A = 77 ₈	End of encoded string for a given Hollerith card image
A = 76 ₈	End of the compressed deck segment
A = 70 ₈	Available for extension

The COMDEK card layout consists of:

Word 1:	0-2	Column binary card type 5
	3-8	Zeros
	9-11	101 (7-9 punches)
	12-35	Binary sequence number
Word 2:		Checksum of word 1 or words 3-24
Words 3- 24:		Compressed card image
Words 25- 27:		Hollerith-coded label or zeros

The binary sequence number is maintained when a COMDEK output is produced and is checked when the deck is used as input. When a sequence error is found in an input COMDEK file, the activity will be terminated.

The label words of the card are supplied in uncompressed form by the I/O Editor and give identification data from columns 73-80 of the standard binary deck cards.

Source Deck Corrections

Corrections to an Assembler language source deck are made by the use of \$ ALTER control cards. A source program correction deck consists of the following parts in order:

1. \$ GMAP control card
2. Text of the subprogram in either of two forms:
 - a. Standard one-instruction-per-card deck
 - b. Compressed deck
3. \$ UPDATE control card (notifies the Comprehensive Operating Supervisor that the cards to follow are to be placed on the A* (alter) file for use by the Assembler)
4. ALTER Information
 - a. ALTER cards (contain the updating delimiting information)
 - b. New source cards which are to be inserted into the source deck as additions or replacement instructions

The operand field of the ALTER card uses alter numbers that are obtained from the previous assembly listing of the deck now being processed. (See page III-90.) The format of the ALTER card is:

Card Column	1	8	16
Symbolic Example	\$	ALTER	n, m
Actual Example	\$	ALTER	07364,07464

The entries define whether the cards following are to be added or to replace cards in the primary input file. These numbers are simply consecutive card numbers starting with 00001 and increasing by one for each source input card.

When it is desired to insert cards into a deck the m subfield is not used. In this case, the cards following this ALTER card, up to but not including the next ALTER card will be inserted just prior to the card corresponding to alter number n.

When it is desired to delete and/or replace one or more cards from a deck, the m subfield is given as shown above. When n and m are equal card n will be deleted. When m identifies a card following n all cards n through m will be deleted. In addition, any cards following this ALTER card up to but not including the next ALTER card will be inserted in place of the deleted cards.

The end of an alter file is designated by the normal end-of-file convention appropriate to the media containing the file.

The \$ UPDATE control card is prepared as indicated below.

Card Column	1	8	16
Symbolic Example	\$	UPDATE	List Option
Actual Example	\$	UPDATE	

The UPDATE control card is used when supplying alter input to a compiler or the Assembler. In the input sequence for a job the \$ UPDATE control card and associated ALTER card with its alter statements must follow and be contiguous to the source program to which the alter statements apply.

The operation field contains the word UPDATE. The variable field may contain the word LIST, in which case a listing of the Alter input will be included with the output.

ASSEMBLY OUTPUTS

Binary Decks

When the \$ GMAP control card specifies the DECK option, the Assembler punches a binary assembly output deck. Since the normal mode of the Assembler is relocatable or is implied as a standard option, all addresses punched in the output cards are relative to zero. Alternatively, still considering the DECK option the Assembler can operate in the absolute mode and punch only absolute addresses in the output cards.

The first card generated by GMAP for every subprogram object deck is a \$ OBJECT card. The format of the \$ OBJECT card is as follows:

Card Column	1	8	16	57	67	72	73	80
Symbolic Example	\$	OBJECT	OPTIONAL COMMENT		{ Date of Ass'y }		{ Optional LABEL }	

The LBL pseudo-operation provides the optional comment and label. The date of the assembly, as determined by a MME GETIME, is inserted in columns 67-72.

This binary information may be represented on four types of binary cards. These cards and their uses are summarized below. GE-625/635 Loader functions performed by using the information from these cards are described in the Loader Manual. In addition, that manual describes the memory map layouts applicable to each user subprogram. The user subprogram memory map blocks are (1) the subprogram region (2) the LABELED COMMON region and (3) the BLANK COMMON region.

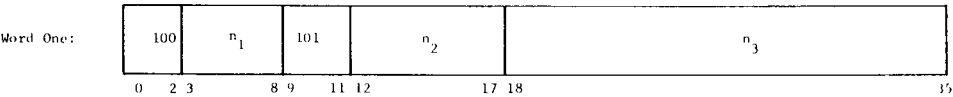
CARD TYPE	USE
Preface	Provides the Loader with (1) the length of the subprogram text region; (2) the length of the BLANK COMMON region; (3) the total number of SYMDEF, SYMREF, and LABELED COMMON symbols; (4) the type identification of each symbol in (3); and (5) the relative entry value or the region length for each symbol in (3).
Relocatable Binary Text	Supplies the Loader with relocatable binary text by using preface card information and relocation identifiers where the relocation identifiers specify whether the 18-bit field refers to a subprogram, LABELED COMMON, or BLANK COMMON regions (of the assembly core-storage area) and will allow the loader to relocate these fields by an appropriate value.
Absolute Binary Text	Provides the Loader with absolute binary text and the absolute starting-location value for Loader use in assigning core-storage addresses to all words on the card.
Transfer	Can be generated only in an absolute assembly and causes the Loader to transfer control to the routine at the location given on the card. (The transfer card is generated automatically as the last card of an absolute subprogram assembly by the END pseudo-operation; however, use of the TCD pseudo-operation can cause the card to appear anywhere in the assembly.)

The formats in which the Assembler punches the above cards are described in the paragraphs to follow.

Preface Card Format

Preface card symbolic entries are primary SYMDEF symbols secondary SYMDEF symbols, SYMREF symbols, LABELED COMMON symbols (from the BLOCK pseudo-operation), and the .SYMT, LABELED COMMON symbol. These symbols appear on the card in a precise order. All SYMDEF symbols appear before any other symbol. Following the SYMDEF symbols are any LABELED COMMON symbols. The SYMREF symbols are then recorded.

The format and content of the preface card are summarized as follows:

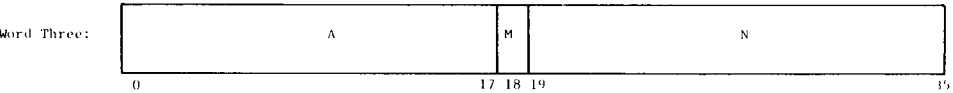


n_1 --V is a value within the range $5 < V \leq 17$ and represents the size of the field within a special relocation entry needed to point to the specific preface card entry. Thus, $V = \log_2 N + 1$, where N is the number of LABELED COMMON and SYMREF entries.

n_2 --Word count of the preface card text

n_3 --Length of the subprogram

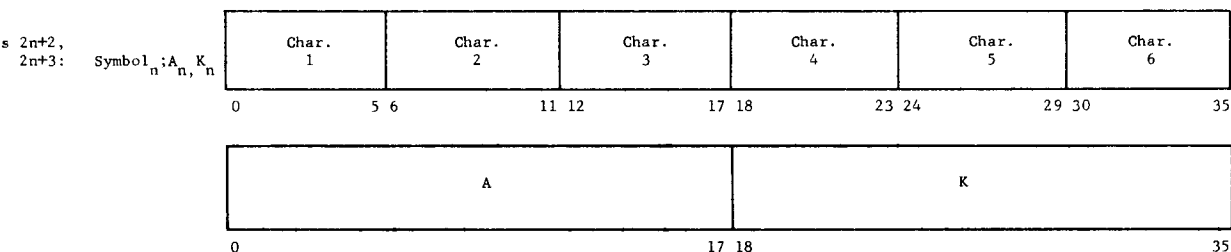
Word Two: Checksum of columns 1-3 and 7-72



The value A is the length of BLANK COMMON; and N is two times the total number of SYMDEFs, SYMREFs, and LABELED COMMONs. The M bit indicates, when set to 1, that the subprogram must be loaded beginning at a location which is a multiple of eight.

Words Four,
Five: Symbol₁; A₁, K₁

Words Six,
Seven: Symbol₂; A₂, K₂



The even-numbered word contains the symbol in BCD. The value K defines the type symbol in the even-numbered word; A is a value associated with K as explained in the following list.

If K equals zero, then the symbol is a primary SYMDEF symbol; A is the entry value relative to the subprogram region origin.

If K equals one, then the symbol is a secondary SYMDEF symbol; A is the entry value relative to the subprogram region origin.

If K equals five, then the symbol is a SYMREF symbol; A is zero.

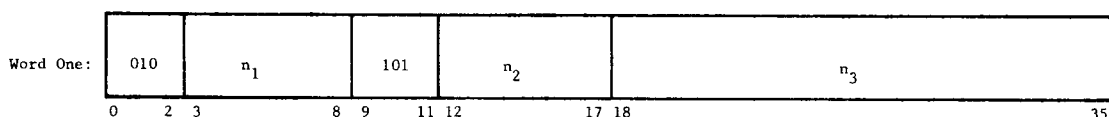
If K equals six, then the symbol is a LABELED COMMON symbol; A is the length of the region.

If K equals seven, then the symbol is a .SYMT. LABELED COMMON symbol; A is the length of the region reserved for debug information.

NOTE: If preface continuation cards are necessary, word three will be repeated unchanged on all continuation cards.

Relocatable Card Format

A relocatable assembly card has the format and contents summarized in the following comments.



n₁--0 indicates that loading is within the subprogram region of the user subprogram core-storage area

n₂--Word count of the data words to be loaded using the origin and relative address in this control word

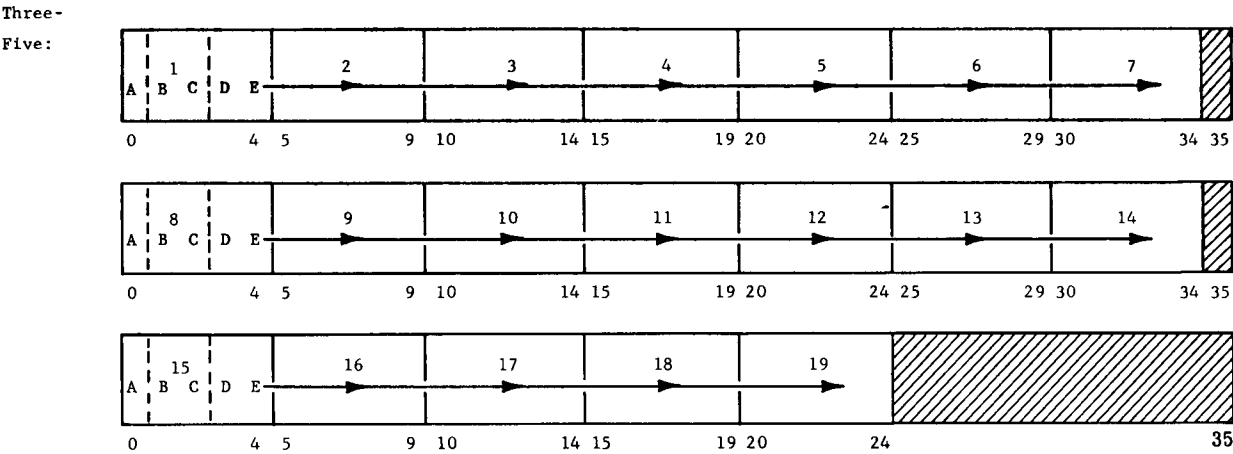
n₃--Loading address, relative to the subprogram region origin.

or for the alternative cases:

n_1--i , where $i \neq 0$ indicates that the i th entry (beginning with the first LABELED COMMON entry in the preface card text has been used and that n_3 is relative to the origin of that entry.

Word Two: Checksum of columns 1-3 and 7-72

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35



Relocation data--words three and four comprise seven 5-bit relocation identifiers, while word five holds 5 such identifiers. The five bits of each identifier carry relocation scheme data for each of the card words ($7+7+5=19$, or fewer). The identifiers are placed in bit positions 0-34 of words three and four and in 0-24 of word five. (Refer to the Relocation Scheme description in the paragraph following.)

Words Six-
Twenty-Four:

Instructions and data (up to 19 words per card). If the card is not complete and at least two words are left vacant, then after the last word entered, word one may be repeated with a new word count and loading address. The loading is then continued with the new address, and the relocation bits are continuously retrieved from words three through five. This process may be repeated as often as necessary to fill a card.

Relocation Scheme

For each binary text word in a relocatable card, the five bits--A, BC, and DE--of each relocation scheme identifier are interpreted by the Loader as follows:

Bit A--0 (reserved for future use)

Bits BC--Left half-word

Bits DE--Right half-word

To every 18-bit half-word one of four code values apply; these are:

<u>CODE VALUE</u>	<u>MEANING</u>
xx = 00	Absolute value that is not to be modified by the Loader.
= 01	Relocatable value that is to be added to the origin of the sub-program region by the Loader.
= 10	BLANK COMMON, relative value that is to be added to the origin of the BLANK COMMON region by the Loader.
= 11	Special entry value (to be interpreted as described in the next paragraph)

apply where xx stands for BC or DE.

If special entry is required, the Loader decodes and processes the text and bits of the 18-bit field (left/right half of each relocatable card word) as follows:

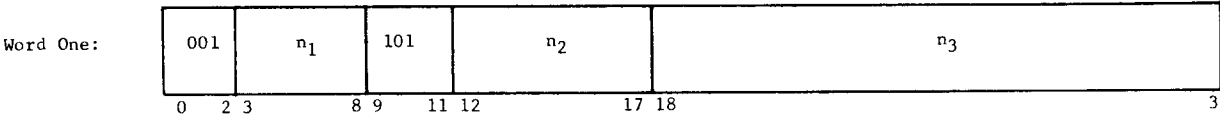
Bit 1	--This is the sign of the addend; 0 implies a plus (+) and 1 implies a minus (-).
Bits 2→V+1	--The value V that was specified in word 1 of the preface card dictates the length of the field. The contents of the field is a relative number which points to a LABELED COMMON region or a SYMREF that appeared in the preface card. The value one in this field would point to the first symbol entry after the last SYMDEF.
Bits V+2→18	--The value in this field is the addend value that appeared in the expression. If the field is all bits then the corresponding 18 bits of the next data word are interpreted as the addend. In this special case there will be no relocation bits for the addend word.

All references to each undefined special symbol are chained together. When the symbol is defined, the Loader can rapidly insert the proper value of the symbol in all relocatable fields that were specified in the chain.

Absolute Card Format

The absolute binary text card appears as shown below.

Word One:



- n₁--0
- n₂--Word count of the card text
- n₃--Loading address relative to the absolute core-storage origin zero (of allocated memory).

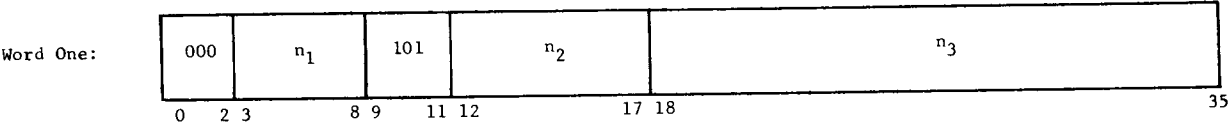
Word Two: Checksum of columns 1-3 and 7-72

Words Three-
Twenty-Four: Instructions and text (22 words per card, maximum). If the card is not complete and at least two words are left vacant, then after the last word entered, word one may be repeated with a new word count and loading address.

Transfer Card Format

The transfer card is generated by the Assembler only in an absolute assembly deck. Its format and contents are:

Word One:



n_1 --0
 n_2 --0
 n_3 --Transfer address (in absolute only).

Words Two-
Twenty-Four: Not used

Assembly Listings

Each Assembler subprogram listing is made up of the following parts:

1. The contents of all preface cards (primary SYMDEF symbols, secondary SYMDEF symbols, SYMREF symbols, LABELED COMMON symbols--from the BLOCK pseudo-operation-- and the .SYMT. LABELED COMMON symbol). This section is omitted from an absolute assembly.
2. The sequence of instructions in order of input to the Assembler.
3. The symbolic reference table.

Full Listing Format

Each instruction word produced by the Assembler is individually printed on a 120-character line. The line contains the following items for each such word of all symbolic cards:

1. Error flags--one character for each error type (see "Error Codes" page III-90).
2. Octal location of the assembled word.

- 3. Octal representation of the assembled word
- 4. Relocation bits for the assembled word (see the topic, Relocation Scheme, Loader manual)
- 5. Reproduction of the symbolic card, including the comments and identification fields, exactly as coded

The exact format of the full listing is shown below.

Fields	A	B	C	D	E	F	G	H
Print line columns	1-6	7-12	15-20	22-25	27, 28	31-33	35-39	41-120
			Machine Instruction					Source Card Image
A--Error flags					E--Tag field modifier			
B--Relative/absolute location					F--Relocation bits			
C--Operand address					G--Alter statement number			
D--Operation code					H--Card image			

Several variations appear for bit positions 15 through 28. (The six, four, two subfield groups C, D, and E shown above is the octal configuration for machine instructions.) These are summarized in the table below in which the X represents one octal digit.

Type of Machine Word	Listing Format	Source Program Instruction
Processor instruction and indirect address	XXXXXX XXXX XX	Processor instruction and indirect address word
Data	XXXXXXXXXXXXX	Data generating pseudo-operations (OCT, DEC, BCI, etc.)
Data Control	XXXXXX XX XXXX	Data Control Word (DCW)
Special 18-bit field data	XXXXXX XXXXXX	ZERO pseudo-operation
Input/output command	XX XXXX XXXXXX	Input/output pseudo-operation (See Appendix D.)

Error flags are summarized at the end of this section. The interpretation of the relocation bits is described in the Loader manual. That field (F) will be blank in an absolute assembly.

Preface Card Listing

The listing of the preface information is in a self-explanatory format, with each major subdivision of preface symbols preceded by a heading. The order is the same as that of the card(s) produced.

Primary SYMDEFs, Secondary SYMDEFs, LABELED COMMON, and SYMREFs. The LABELED COMMONs and SYMREFs are numbered sequentially 1 through n, where this number represents the special relocation entry number employed in referencing these special symbols.

BLANK COMMON Entry

Prior to the listing of the special symbols, the Assembler enters a statement of the amount of BLANK COMMON storage requested by the subprogram. The statement format is self-explanatory.

Symbolic Reference Table

The symbol table listing contains all symbols used, their octal values (normally, the location value), and the alter numbers of all instructions that referenced the symbol. The table format is as follows:

<u>Definition</u>	<u>Symbol</u>	<u>Alter Numbers</u>
00364	BETA	00103,00103,01027,01761,03767,07954

The above sample indicates that the symbol BETA has been assigned the value 364₈ and is referenced in five places: namely, at alter number positions 00103,01027, 01761, 03767, and 07954 in the listing of instructions. The first alter number is the point in the instruction listing where the symbol was defined. If an instruction contains a symbol twice, the alter number for that point in the instruction listing is given twice. The alter numbers are assigned sequentially in the subprogram listing, one per instruction. Because of this fact, it is easy for the programmer to locate in the listing those card images that referenced any particular symbol as well as locate the card image that caused the symbol to be defined.

Error Codes

The following list comprises the error flags for individual instructions and pseudo-operations.

<u>ERROR</u>	<u>FLAG</u>	<u>CAUSE</u>
Undefined	U	Undefined symbol(s) appear in the variable field.
Multidefined	M	Multiple-defined symbol(s) appear in the location field and/or the variable field.
Address	A	Illegal value or symbol appears in the variable field. Also used to denote lack of a required field.
Index	X	Illegal index or address modification.
Relocation	R	Relocation error; expression in the variable field will produce a relocatable error upon loading.
Phase	P	Phase error; this implies undetected machine error or symbols becoming defined in Pass two with a different value from Pass one.
Even	E	Inappropriate character in column 7.
Conversion	C	Error in conversion of either a literal constant or a subfield of a data-generative pseudo-operation. Illegal character.
Location	L	Error in the location field.
Operation	O	Illegal operation.
Table	T	An assembly table overflowed not permitting proper processing of this card completely. Table overflow error information will appear at the end of testing.

IV. CODING EXAMPLES

PRELIMINARY

This chapter contains examples of coding techniques for performing typical program functions. These examples:

1. Indicate how certain very efficient Processor instructions can be used
2. Illustrate the use of address modification variations for indexing, indirection and automatic tallying
3. Demonstrate operations performed on characters
4. Show operations on fixed- and floating-point numbers
5. Present the use of the BCD instruction

The list of examples is by no means complete in that it does not present all of the processor capabilities; however, the examples provided can serve as convenient references for programmers newly acquainted with the GE-635.

Each example is self-contained and self-explanatory. In most cases, questions that may be raised can be answered by referring to the descriptions of particular instructions or pseudo-operations. Convenient references are contained in Appendices A through D.

EXAMPLES

Fixed Point to Floating Point (Integer)

The following example illustrates the conversion of a fixed-point integer to floating point (float an integer). The integer to be converted is in the location M.

Step 01 resets the Overflow Indicator.

Step 02 places the binary integer to be converted in the accumulator.

Step 03 places zeros in the quotient register.

Step 04 sets the exponent register to 35_{10} .

Step 05 converts the number in the accumulator to floating point.

For example, if the contents of M equal 000000000002_8 , then the contents of the floating-point register will be $E = 2_{10}$, and $AQ = 2000000000000000000000_8$ at the completion of step 05.

```

01      TOV      1,IC
02      LDA      M          .FLOAT AN INTEGER M
03      LDO      ,DL        C(A0) = M AT B35.
04      LDE      =35B25,DU  C(E) = 35.
05      FNO      NORMALIZE M

```

Floating Point to Fixed Point (Integer)

The following example illustrates the conversion of a double-precision, floating-point number to a fixed-point number, binary point 71. The result will be only the integral part of the number. The number to be converted must lie between -2^{71} and $2^{71}-1$ inclusive.

Step 01 loads the floating-point number to be converted into the floating-point register.

Step 02, an unnormalized floating add of zero (exponent of 71), causes the contents of AQ to be shifted right a number of places equal to the difference between 71 and the exponent of the number to be converted. This will leave in AQ the binary integer (binary point 71) equal to the integral part of the floating-point number in X and X+1.

For example, if prior to executing step 02, the floating-point register contained -2, that is, if the exponent register contained 2_{10} and AQ contained 40000000000000000000_8 , then the result in AQ after the addition of zero (exponent 71) would be 77777777777777777776_8 .

```

      .
      .
      .
01      DFLD      X      COMPUTE THE INTEGER PART OF
                        A FLOATING-POINT NUMBER CON-
                        TAINED IN X AND X+1.
02      UFA      =71B25,DU  FIX THE RESULT IN AO, BINARY
                        POINT 71.

```

Real Logarithm

Purpose:

Compute $\log X$ for $\text{ALOG}(X)$ or $\text{ALOG10}(X)$ in an expression.

Method:

1. $\log_2 X = \log_2 (2^I * F) = I + \log_2 F$, where $X = 2^I * F$.
2. $\log_e X = \log_e 2^{(\log_2 X)} = (\log_2 X) * (\log_e 2)$, and similarly $\log_{10} X = (\log_2 X) * (\log_{10} 2)$.
3. $\log_2 X = Z * \left(A + \frac{B}{Z^2 - C} \right) - \frac{1}{2}$, where $Z = \frac{F - \sqrt{\frac{2}{2}}}{F + \sqrt{\frac{2}{2}}}$ and

 $A = 1.2920070987$
 $B = -2.6398577031$
 $C = 1.6567626301$
4. X and $\log X$ are real numbers, with values of X from 2^{-129} to $2^{127} - 2^{100}$ inclusive.
5. $\log X$ is accurate to 8 decimal places.

Use:

Calling Sequence -- `CALL ALOG(X)` for $\log_e X$
`CALL ALOG10(X)` for $\log_{10} X$

LOGS	SYMDEF	ALOG10,ALOG	REAL LOGARITHM FUNCTIONS
	SAVE		$X = (2^{**I}) * F = \text{ARGUMENT}$
	FLD	2,1*	
	FNO		
	TZE	ERR1	ERROR IF X=0
	TMI	ERR2	ERROR IF X NEGATIVE
BEGIN	FCMP	=1.0,DU	
	TZE	UNITY	
	STE	I	LOG(1) = 0
	LDE	0,DU	STORE I AT BINARY POINT 7
	DFAD	SRHLF	OBTAIN F
	DFST	Z	
	DFSB	SRTWO	
	DFDV	Z	
	DFST	Z	$Z = (F - \text{SQRT}(1/2)) / (F + \text{SQRT}(1/2))$
	DFMP	Z	Z^2
	DFSB	C	$Z^2 - C$
	DFDI	B	$B / (Z^2 - C)$
	DFAD	A	$A + B / (Z^2 - C)$
	DFMP	Z	$Z(A + B / (Z^2 - C))$
	DFST	Z	$Z = Z(A + B / (Z^{**2} - C)) = \text{LOG2}(F) + 1/2$
I	LDA	*,*,DU	
	LDQ	0,DU	
	LDE	=7B25,DU	
	FSB	=0.5,DU	
	DFAD	Z	Float I
INDIC	DFMP	*	$\text{LOG2}(X) = I + \text{LOG2}(F)$
	RETURN	LOGS	CONVERT TO BASE 10 OR E
ERR1	CALL	.FXEM.(EALN1)	ERROR EXIT NUMBER 1 (X=0)
UNITY	FLD	=0.0,DU	
	RETURN	LOGS	
ERR2	CALL	.FXEM.(EALN2)	ERROR EXIT NUMBER 2 (X IS NEGATIVE)
	FNEG		
	TRA	BEGIN	
ALOG10	ESTC2	INDIC	REAL COMMON LOGARITHM
	TRA	LOGS	
	DEC	.301029996D0	
ALOG	ESTC2	INDIC	REAL NATURAL LOGARITHM
	TRA	LOGS	
	DEC	6.93147180559D-1	
EALN1	DEC	9	
EALN2	DEC	10	
A	DEC	.12920070987D1	
B	DEC	-.26398577031D1	
C	DEC	.16567626301D1	
SRHLF	DEC	.707106781187D0	SQUARE ROOT OF TWO DIVIDED BY TWO
SRTWO	DEC	.1414213562374D1	SQUAPE ROOT OF TWO
Z	BSS	2	
	END		

BCD Addition

The following example illustrates the addition of two words containing BCD integers. The example limits the result to 999999.

Step 01 places the number in A into the accumulator.

Step 02 adds the number in B to the accumulator. Column V in the table, following, shows the possible results for any digit. It should be noted that there are 19 possible results, indicated by lines 0-18.

Step 03 forces any carries into the units position of the next digit. Lines 10-18 of Column V contain the sums that will carry into the next digit. Column W contains the 20 possible results for each digit position. The additional possibility (line 19) arises from the fact that there can be a carry of one into a digit.

Step 04 stores the intermediate result in C.

Step 05 extracts an octal 60 from each non-carry digit. The results are indicated in column X. The digits that did not force a carry (lines 0-9) result in an octal 60, the digits that had a carry into the next digit (lines 10-18) result in 00.

Step 06 performs an exclusive OR of the contents of the accumulator with the contents of C. This in effect subtracts octal 60 from each digit that did not have a carry (lines 0-9). The results are indicated in column Y.

Step 07 shifts the octal 60s to the right three places.

Step 08 negates the contents of the accumulator.

Step 09 is an add to storage the contents of the accumulator to the contents of C. This in effect subtracts a 06 from each digit that did not have a carry, the results of which are indicated in Column Z.

01	LDA	A	{	TO ADD C = A+B IN BCD. COMPUTE A+B
02	ADLA	B		
03	ADLA	=066666666666	{	ADD OCTAL 66 TO EACH DIGIT TO FORCE CARRIES
04	STA	C		
05	ANA	=0606060606060	{	EXTRACT OCTAL 60 FROM EACH NON-CARRY SUBTRACT OCTAL 60 FROM EACH NON-CARRY
06	ERSA	C		
07	ARL	3	{	SUBTRACT OCTAL 06 FROM EACH NON-CARRY
08	NEG			
09	ASA	C		

ADDITION RESULTS

LINE	V	W	X	Y	Z
0	00	66	60	6	00
1	01	67	60	7	01
2	02	70	60	10	02
3	03	71	60	11	03
4	04	72	60	12	04
5	05	73	60	13	05
6	06	75	60	14	06
7	07	75	60	15	07
8	10	76	60	16	10
9	11	77	60	17	11
10	12	00	00	0	00
11	13	01	00	1	01
12	14	02	00	2	02
13	15	03	00	3	03
14	16	04	00	4	04
15	17	05	00	5	05
16	20	06	00	6	06
17	21	07	00	7	07
18	22	10	00	10	10
19	-	11	00	11	11

BCD Subtraction

The following is an example of subtracting one BCD number from another BCD number. The contents of A must be equal to or greater than the contents of B.

Step 01 loads the accumulator with the contents of A.

Step 02 subtracts the contents of B from the accumulator. The possible results for each digit are indicated in Column W of the table that is included with this example.

Step 03 stores the intermediate result in C.

Step 04 extracts an octal 60 from each digit that required a borrow. This will leave an octal 60 in each digit position where there was a borrow. The possible results of this instruction are indicated in Column X, lines 0-19 (10-19 refer to those which result in octal 60).

Step 05, an exclusive OR to storage, in effect subtracts the octal 60's in the accumulator from the corresponding digit in C. The possible results for each digit are displayed in Column Y.

Step 06 shifts the octal 60's in the accumulator right three places.

Step 07 negates the contents of the accumulator.

Step 08, an add to storage, is in effect a subtraction of 06 from each digit that required a borrow, the result being placed in C. Column Z of the table reflects the possible results for each digit.

01	LDA }	A	{	TO SUBTRACT C = A-B IN BCD.
02	SBLA }	B		COMPUTE A-B
03	STA	C		
04	ANA	=0606060606060		EXTRACT OCTAL 60 FROM EACH BORROW
05	ERSA	C		SUBTRACT OCTAL 60 FROM EACH BORROW
06	ARL }	3	{	SUBTRACT OCTAL
07	NEG }			06 FROM EACH
08	ASA }	C		BORROW

SUBTRACTION RESULTS

LINE	W	X	Y	Z
0	11	0	11	11
1	10	0	10	10
2	07	0	07	07
3	06	0	06	06
4	05	0	05	05
5	05	0	04	04
6	03	0	03	03
7	06	0	02	02
8	01	0	01	01
9	00	0	00	00
10	77	60	17	11
11	76	60	16	10
12	75	60	15	07
13	74	60	14	06
14	73	60	13	05
15	72	60	12	04
16	71	60	11	03
17	70	60	10	02
18	67	60	7	01
19	66	60	6	00

Character Transliteration

The following example illustrates a method of transliterating each character of a card image that has been punched in the FORTRAN Character Set to the octal value of the corresponding character in the General Electric Standard Character Set. There are 48 characters in the FORTRAN Set and 64 characters in the General Electric Standard Character Set. Each character that is punched invalidly (not a standard punch combination in the FORTRAN Set) is converted to a blank. The card is originated at IMAGE.

Steps 01 and 02 initialize the indirect word TALLY2.

Step 03 picks up the character to be transliterated by referencing the word TALLY2 with the Character from Indirect (CI) modifier. This will place the character specified by bits 33-35 of TALLY2 from a location specified by bits 0-17 of TALLY2 into the accumulator, bits 29-35. Bits 0-28 of the accumulator will be set to zero. Step 03 is forced even so as to place the four-step loop (step 03-06) in two even/odd pairs. This decreases run time.

Step 04 picks up the corresponding General Electric standard character from the address TABLE modified by the contents of accumulator, bits 18-35.

Step 05 places the transliterated character back in the card image where it was originally picked up. The Sequence Character (SC) modifier increments the character specified in bits 33-35 of the word TALLY2.

Each time the character position becomes greater than 5, it is reset to zero; and the address specified in bits 0-17 of TALLY2 is incremented by one. The tally in bits 18-29 of the same word is decremented by 1 with each SC reference. Whenever a tally reaches zero, the Tally Runout Indicator is set ON. Otherwise, it is set OFF.

Step 06 tests the Tally Runout Indicator. If it is OFF, the program transfers to LOOP; if not, the next sequential instruction is taken.

The table, TABLE, is 64 locations long. The character in each location is a General Electric standard character that corresponds to a FORTRAN character in the following manner. The relative location of a particular character to the start of the table is equal to the binary value of the corresponding FORTRAN character. For example, an A punched in the FORTRAN Character Set has the octal value 21(17₁₀). The relative location 17 to TABLE contains an A in the General Electric Standard Character Set. A 3-8 punch in the FORTRAN Set represents an = character. The 3-8 punch would be read as an octal 13(11₁₀). The relative location 11 to TABLE contains an octal 75 (see line 21) which represents the = character in the General Electric Standard Character Set.

```

      .
      .
      .
01      LDA      TALLY1      INITIALIZE TALLY WORD
02      STA      TALLY2
03  LOOP  ELDA      TALLY2,C1  PICK UP CHARACTER TO BE TRANSLITERATED
04      LDO      TABLE,AL    LOAD OR WITH TRANSLITERATED CHARACTER
05      STO      TALLY2,SC    STORE BACK ON CARD IMAGE
06      TTF      LOOP        IF TALLY HAS NOT RUN OUT  CONTINUE LOOP
      .
      .
07  TALLY1  TALLY      IMAGE,80,0
08  TALLY2  ZERO
09  IMAGE   BSS      14
10  TABLE  OCT      0
11          OCT      1
12          OCT      2
13          OCT      3
14          OCT      4
15          OCT      5
16          OCT      6
17          OCT      7
18          OCT     10
19          OCT     11
20          OCT     20
21          OCT     75      3-8 PUNCH = IN FORTRAN SET
22          OCT     57      4-8 PUNCH ' IN FORTRAN SET
23          OCT     20
24          OCT     20
25          OCT     20
26          OCT     20
27          OCT     21
28          OCT     22
29          OCT     23
30          OCT     24
31          OCT     25
32          OCT     26
33          OCT     27
```

Steps 01-11 are comment cards.

Step 12 places the contents of the lower half (bits 18-35) of the quotient register plus 64, in index register 0. The number 64, in effect, sets the ZF terminate repeat condition on. The instruction also places the last 8 bits of the size of the table in index register 0, bits 0-7. Thus, if the size of the table is a multiple of 256 words, zeros will be loaded into bits 0-7 of index register 1. Zeros in those bit positions will cause the repeat to execute 256 times. If, however, the size of the table to be searched is of the form $256n+m$, where $n \geq 0$, and $0 < m < 256$, then m would be placed in bits 0-7 of index register 0. This will cause the repeat instruction to be executed a maximum of m times on the first pass through.

Step 13 subtracts 1024 from the quotient register. This, in effect, subtracts 1 from the size of the table to be searched. The subtracting of 1 becomes meaningful in two places: (1) it provides a test to be sure the table is not zero words long (see step 14) and (2) if the table is a multiple of 256 words long, it effectively subtracts 1 from bits 0-17 (a look-ahead to steps 18 and 19 points out the importance of this).

Step 14 causes the routine to return to the main program if the size of the table was zero.

Step 15, an RPTX, executes step 16 a number of times equal to the contents of index register 0, bits 0-7, at the start of the instruction execution. Each time step 16 is executed, the contents of the accumulator (the search argument) are compared with the contents of the location specified by index register 1. At the same time, index register 1 is incremented by W as is specified in the repeat instruction; and the contents of index register 0, bits 0-7, are decremented by 1. The repeat sequence terminates when the compare causes the Zero Indicator to be set or when bits 0-7 of index register 0 are set to zero.

Step 17 test the Zero Indicator and returns to the main program if it is set. It should be noted that index register 1 will be set W locations higher than when the equal argument was found because of the sequence of events described above.

Step 18. If the Zero Indicator was not set by step 16, then step 18 will be executed. This instruction subtracts 1 from bits 0-17 of the quotient register. In effect, this is subtracting 256 from the size of the table. The size of the table can be expressed in the form $256n+m$. If $m=0$ and $n=1$, then the contents of the quotient register would also go zero at this point. This is because step 13 would have caused a borrow of 1 from n when m equals zero. Further inspection of these instructions will reveal that positive values of n and m , other than those expressed above, will only cause the routine to loop until the contents of the quotient register are reduced to a negative value.

Step 19 transfers control to step 15 if the contents of quotient register remained positive. If the quotient register became negative, step 20 is executed and the routine returns to the main program.

Steps 01-11 are comment cards.

Step 12 places the contents of the lower half (bits 18-35) of the quotient register plus 64, in index register 0. The number 64, in effect, sets the TZE terminate repeat condition on. The instruction also places the last 8 bits of the size of the table in index register 0, bits 0-7. Thus, if the size of the table is a multiple of 256 words, zeros will be loaded into bits 0-7 of index register 1. Zeros in those bit positions will cause the repeat to execute 256 times. If, however, the size of the table to be searched is of the form $256n+m$, where $n \geq 0$, and $0 < m < 256$, then m would be placed in bits 0-7 of index register 0. This will cause the repeat instruction to be executed a maximum of m times on the first pass through.

Step 13 subtracts 1024 from the quotient register. This, in effect, subtracts 1 from the size of the table to be searched. The subtracting of 1 becomes meaningful in two places: (1) it provides a test to be sure the table is not zero words long (see step 14) and (2) if the table is a multiple of 256 words long, it effectively subtracts 1 from bits 0-17 (a look-ahead to steps 18 and 19 points out the importance of this).

Step 14 causes the routine to return to the main program if the size of the table was zero.

Step 15, an RPTX, executes step 16 a number of times equal to the contents of index register 0, bits 0-7, at the start of the instruction execution. Each time step 16 is executed, the contents of the accumulator (the search argument) are compared with the contents of the location specified by index register 1. At the same time, index register 1 is incremented by W as is specified in the repeat instruction; and the contents of index register 0, bits 0-7, are decremented by 1. The repeat sequence terminates when the compare causes the Zero Indicator to be set or when bits 0-7 of index register 0 are set to zero.

Step 17 test the Zero Indicator and returns to the main program if it is set. It should be noted that index register 1 will be set W locations higher than when the equal argument was found because of the sequence of events described above.

Step 18. If the Zero Indicator was not set by step 16, then step 18 will be executed. This instruction subtracts 1 from bits 0-17 of the quotient register. In effect, this is subtracting 256 from the size of the table. The size of the table can be expressed in the form $256n+m$. If $m=0$ and $n=1$, then the contents of the quotient register would also go zero at this point. This is because step 13 would have caused a borrow of 1 from n when m equals zero. Further inspection of these instructions will reveal that positive values of n and m , other than those expressed above, will only cause the routine to loop until the contents of the quotient register are reduced to a negative value.

Step 19 transfers control to step 15 if the contents of quotient register remained positive. If the quotient register became negative, step 20 is executed and the routine returns to the main program.

It should be noted that when control is transferred back to step 15, index register 0, bits 0-7, contains zeros (causes the repeat to be executed a maximum of 256 times); and index register 1 contains the address of the next location in the table that is to be searched.

```

01      *      CALLING SEQUENCE IS
02      *      LDA      ITEM      SEARCH ITEM.
03      *      LDQ      SIZE      NUMBER OF TABLE ENTRIES--AT B25.
04      *      LDX1     FIRST,DU   LOCATION OF FIRST SEARCH WORD IN TABLE.
05      *      TSX2     TLU        CALL TABLE LOOKUP SUBROUTINE.
06      *      TZE      FOUND      TRANSFER IF SEARCH ITEM IS IN TABLE, OR
07      *      TNZ      ABSENT     TRANSFER IF SEARCH ITEM IS NOT IN TABLE.
08      *      USE ONE OF THE TWO INSTRUCTIONS IMMEDIATELY ABOVE.
09      *      IF IN TABLE, C(X1)-W WILL BE THE LOCATION OF THE MATCHING SEARCH
10      *      WORD.      OTHERWISE, C(X1)-W WILL BE THE LOCATION OF THE LAST
11      *      SEARCH WORD IN THE TABLE. W IS THE NUMBER OF WORDS PER ENTRY.
12      TLU      EAX0      64,QL    PICKUP SIZE (MOD 256) AND TZE-BIT.
13      SBLO      1024,DL    SIZE = SIZE-1.
14      TMI      ,2         EXIT IF SIZE WAS 0--EMPTY TABLE.
15      TLU1     RPTX      ,W      NOTE THAT 0 REPRESENTS 256 (MOD 256).
16      CMPA      ,1        PERFORM TABLE LOOKUP
17      TZE      ,2         EXIT IF SEARCH ITEM IS IN TABLE.
18      SBLO      1,DU      SIZE = SIZE-256.
19      TPL      TLU1      CONTINUE TABLE LOOKUP IF MORE ENTRIES.
20      TRA      ,2        EXIT--SEARCH ITEM IS NOT IN TABLE.

```

Binary to BCD

The following example illustrates a method of converting a number from binary to BCD. The example converts a number that is in the range of -10^6+1 to $+10^6-1$, inclusive.

Step 01 places zeros in index register 2.

Step 02 loads the accumulator with the binary number that is to be converted.

Steps 03 and 04 perform the conversion of the binary number in the accumulator to the Binary-Coded Decimal equivalent. Step 03 will repeat step 04 six times. It will also increment the contents of index register 2 by one after each execution.

The BCD instruction, step 04, is designed to convert the magnitude of the contents of the accumulator to the Binary-Coded Decimal equivalent. The method employed is to effectively divide a constant into this number, place the result in bits 30-35 of the quotient register, and leave the remainder in the accumulator. The execution of the BCD instruction will then allow the user to convert a binary number to BCD, one digit at a time, with each digit coming from the high-order part of the number. The address of the BCD instruction refers to a constant to be used in the division, and a different constant would be needed for each digit. In the process of the conversion, the number in the accumulator is shifted left three positions. The $C(Q)_{0-35}$ are shifted left 6 positions before the new digit is stored.

In this example, the constants used for dividing are located at TAB, TAB+1, TAB+2,...,TAB+5. If the value in X were 00000522241₈, the quotient register would contain 010703020107₈ at the completion of the repeat sequence. Step 05 stores the quotient register in Y.

The values in the table below are the conversion constants to be used with the Binary to BCD instruction. Each vertical column represents the set of constants to be used depending on the initial value of the binary number to be converted to its decimal equivalent. The instruction is executed once per digit, using the constant appropriate to the conversion step with each execution.

An alternate use of the table for conversion involves the use of the constants in the row corresponding to conversion step 1. If after each conversion, the contents of the accumulator are shifted right 3 positions, the constants in the conversion step 1 row may be used one at a time in order of decreasing value until the conversion is complete.

BINARY TO BCD CONVERSION CONSTANTS

Starting Range of C(AR)										
Conversion Step	$-10^{10}+1 \rightarrow 10^{10}-1$	$-10^9+1 \rightarrow 10^9-1$	$-10^8+1 \rightarrow 10^8-1$	$-10^7+1 \rightarrow 10^7-1$	$-10^6+1 \rightarrow 10^6-1$	$-10^5+1 \rightarrow 10^5-1$	$-10^4+1 \rightarrow 10^4-1$	$-10^3+1 \rightarrow 10^3-1$	$-10^2+1 \rightarrow 10^2-1$	$-10^1+1 \rightarrow 10^1-1$
1	$8^1 \times 10^9$	$8^2 \times 10^8$	$8^3 \times 10^7$	$8^4 \times 10^6$	$8^5 \times 10^5$	$8^6 \times 10^4$	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8
2	$8^2 \times 10^8$	$8^3 \times 10^7$	$8^4 \times 10^6$	$8^5 \times 10^5$	$8^6 \times 10^4$	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8	
3	$8^3 \times 10^7$	$8^4 \times 10^6$	$8^5 \times 10^5$	$8^6 \times 10^4$	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8		
4	$8^4 \times 10^6$	$8^5 \times 10^5$	$8^6 \times 10^4$	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8			
5	$8^5 \times 10^5$	$8^6 \times 10^4$	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8				
6	$8^6 \times 10^4$	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8					
7	$8^7 \times 10^3$	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8						
8	$8^8 \times 10^2$	$8^9 \times 10^1$	8^8							
9	$8^9 \times 10^1$	8^8								
10	8^{10}									

01	LDX2	0,DU	PLACE ZEROS IN X2
02	LDA	X	LOAD ACCUMULATOR WITH VALUE TO BE CONVERTED
03	RPT	6,1	REPEAT 6 TIMES, INCREMENT BY 1
04	BCD	TAB,2	DIVIDE BY TAB, TAB+1, ETC
05	STQ	Y	STORE CONVERTED NUMBER IN Y
	.		
	.		
	.		
06	TAB DEC	800000, 640000, 512000, 409600, 327680,	
	DEC	262144	

APPENDIX A. GE-625/635 INSTRUCTIONS LISTED
BY FUNCTIONAL CLASS WITH PAGE REFERENCES AND TIMINGS

DATA MOVEMENT			GE-625 Timing (μ sec) [/]	GE-635 Timing (μ sec) [/]	Reference (Page)
<u>Load</u>					
LDA	235	Load A	3.0	1.8	II -39
LDQ	236	Load Q	3.0	1.8	39
LDAQ	237	Load AQ	3.0	1.9	39
LDX _n	22n	Load X _n	3.0	1.8	40
LREG	073	Load Registers	9.0	4.8	40
LCA	335	Load Complement A	3.0	1.8	41
LCQ	336	Load Complement Q	3.0	1.8	42
LCAQ	337	Load Complement AQ	3.0	1.9	42
LCX _n	32n	Load Complement X _n	3.0	1.8	43
EAA	635	Effective Address to A	2.0	1.3	43
EAQ	636	Effective Address to Q	2.0	1.3	44
EAX _n	62n	Effective Address to X _n	2.0	1.3	44
LDI	634	Load Indicator Register	3.0	1.8	45
<u>Store</u>					
STA	755	Store A	3.5	2.5	46
STQ	756	Store Q	3.5	2.5	46
STAQ	757	Store AQ	3.5	3.0	46
STX _n	74n	Store X _n	3.5	2.5	46
SREG	753	Store Register	11.5	7.5	47
STCA	751	Store Character of A (6 Bit)	3.5	2.5	47
STCQ	752	Store Character of Q (6 Bit)	3.5	2.5	48
STBA	551	Store Character of A (9 Bit)	3.5	2.5	49
STBQ	552	Store Character of Q (9 Bit)	3.5	2.5	50
STI	754	Store Indicator Register	3.5	2.9	51
STT	454	Store Timer Register	3.5	2.5	52
SBAR	550	Store Base Address Register	3.5	2.9	52
STZ	450	Store Zero	3.5	2.5	52
STC1	554	Store Instruction Counter plus 1	3.5	2.9	53
STC2	750	Store Instruction Counter plus 2	3.5	2.9	53
<u>Shift</u>					
ARS	731	A Right Shift	2.0	1.8	54
QRS	732	Q Right Shift	2.0	1.8	54
LRS	733	Long Right Shift	2.0	1.8	54
ALS	735	A Left Shift	2.0	1.8	55
QLS	736	Q Left Shift	2.0	1.8	55
LLS	737	Long Left Shift	2.0	1.8	56
ARL	771	A Right Logic	2.0	1.8	56
QRL	772	Q Right Logic	2.0	1.8	56
LRL	773	Long Right Logic	2.0	1.8	57
ALR	775	A Left Rotate	2.0	1.8	57
QLR	776	Q Left Rotate	2.0	1.8	57
LLR	777	Long Left Rotate	2.0	1.8	58

[/] See Calculation of Instruction Execution Times, page II-33.

FIXED-POINT ARITHMETIC

			GE-625 Timing (μ sec) [†]	GE-635 Timing (μ sec) [†]	Reference (Page)
<u>Addition</u>					
ADA	075	Add to A	3.0	1.8	II-59
ADQ	076	Add to Q	3.0	1.8	59
ADAQ	077	Add to AQ	3.0	1.9	60
ADXn	06n	Add to Xn	3.0	1.8	60
ASA	055	Add Stored to A	4.0	2.8	61
ASQ	056	Add Stored to Q	4.0	2.8	61
ASXn	04n	Add Stored to Xn	4.0	2.8	62
ADLA	035	Add Logic to A	3.0	1.8	62
ADLQ	036	Add Logic to Q	3.0	1.8	63
ADLAQ	037	Add Logic to AQ	3.0	1.9	63
ADLXn	02n	Add Logic to Xn	3.0	1.8	64
AWCA	071	Add with Carry to A	3.0	1.8	64
AWCQ	072	Add with Carry to Q	3.0	1.8	65
ADL	033	Add Low to AQ	3.0	1.8	66
AOS	054	Add One to Storage	4.0	2.8	66
<u>Subtraction</u>					
SBA	175	Subtract from A	3.0	1.8	67
SBQ	176	Subtract from Q	3.0	1.8	67
SBAQ	177	Subtract from AQ	3.0	1.9	68
SBXn	16n	Subtract from Xn	3.0	1.8	68
SSA	155	Subtract Stored from A	4.0	2.8	69
SSQ	156	Subtract Stored from Q	4.0	2.8	69
SSXn	14n	Subtract Stored from Xn	4.0	2.8	70
SBLA	135	Subtract Logic from A	3.0	1.8	70
SBLQ	136	Subtract Logic from Q	3.0	1.8	71
SBLAQ	137	Subtract Logic from AQ	3.0	1.9	71
SBLXn	12n	Subtract Logic from Xn	3.0	1.8	72
SWCA	171	Subtract with Carry from A	3.0	1.8	72
SWCQ	172	Subtract with Carry from Q	3.0	1.8	73
<u>Multiplication</u>					
MPY	402	Multiply Integer	7.0	7.0	74
MPF	401	Multiply Fraction	7.0	7.0	75

[†] See Calculation of Instruction Execution Times, page II-33.

<u>Division</u>			<u>GE-625 Timing (μsec)[/]</u>	<u>GE-635 Timing (μsec)[/]</u>	<u>Reference (Page)</u>
DIV	506	Divide Integer	14.5*	14.2*	II-76
DVF	507	Divide Fraction	14.5*	14.2*	77

Negate

NEG	531	Negate A	2.0	1.3	78
NEGL	533	Negate Long	2.0	1.3	78

* When actual division does not take place, GE-635 2.5 μ sec, GE-625 2.8 μ sec.

BOOLEAN OPERATIONS

AND

ANA	375	AND to A	3.0	1.8	79
ANQ	376	AND to Q	3.0	1.8	79
ANAQ	377	AND to AQ	3.0	1.9	79
ANXn	36n	AND to Xn	3.0	1.8	80
ANSA	355	AND to Storage A	4.0	2.8	80
ANSQ	356	AND to Storage Q	4.0	2.8	80
ANSXn	34n	AND to Storage Xn	4.0	2.8	81

OR

ORA	275	OR to A	3.0	1.8	81
ORQ	276	OR to Q	3.0	1.8	81
ORAQ	277	OR to AQ	3.0	1.9	82
ORXn	26n	OR to Xn	3.0	1.8	82
ORSA	255	OR to Storage A	4.0	2.8	82
ORSQ	256	OR to Storage Q	4.0	2.8	83
ORSXn	24n	OR to Storage Xn	4.0	2.8	83

EXCLUSIVE OR

ERA	675	EXCLUSIVE OR to A	3.0	1.8	83
ERQ	676	EXCLUSIVE OR to Q	3.0	1.8	84
ERAQ	677	EXCLUSIVE OR to AQ	3.0	1.9	84
ERXn	66n	EXCLUSIVE OR to Xn	3.0	1.8	84
ERSA	655	EXCLUSIVE OR to Storage A	3.0	2.8	85
ERSQ	656	EXCLUSIVE OR to Storage Q	3.0	2.8	85
ERSXn	64n	EXCLUSIVE OR to Storage Xn	3.0	2.8	85

[/] See Calculation of Instruction Execution Times, page II-33.

COMPARISON			GE-625 Timing (μ sec) [/]	GE-635 Timing (μ sec) [/]	Reference (Page)
<u>Compare</u>					
CMPA	115	Compare with A	3.0	1.8	II-86
CMPQ	116	Compare with Q	3.0	1.8	87
CMPAQ	117	Compare with AQ	3.0	1.9	88
CMPXn	10n	Compare with Xn	3.0	1.8	89
CWL	111	Compare with Limits	3.0	2.2	90
CMG	405	Compare Magnitude	3.0	1.8	91
SZN	234	Set Zero and Negative Indicators from Memory	3.0	1.8	91
CMK	211	Compare Masked	3.0	2.2	92

Comparative AND

CANA	315	Comparative AND with A	3.0	1.8	93
CANQ	316	Comparative AND with Q	3.0	1.8	93
CANAQ	317	Comparative AND with AQ	3.0	1.9	93
CANXn	30n	Comparative AND with Xn	3.0	1.8	94

Comparative NOT

CNAA	215	Comparative NOT with A	3.0	1.8	94
CNAQ	216	Comparative NOT with Q	3.0	1.8	94
CNAAQ	217	Comparative NOT with AQ	3.0	1.9	95
CNAXn	20n	Comparative NOT with Xn	3.0	1.8	95

FLOATING POINT

Load

FLD	431	Floating Load	3.0	1.8	96
DFLD	433	Double-Precision Floating Load	3.0	1.9	96
LDE	411	Load Exponent Register	3.0	1.8	96

Store

FST	455	Floating Store	3.5	2.5	97
DFST	457	Double-Precision Floating Store	4.0	3.0	97
STE	456	Store Exponent Register	3.5	2.5	97

Addition

FAD	475	Floating Add	3.0	2.7	98
UFA	435	Unnormalized Floating Add	3.0	2.5	98

[/] See Calculation of Instruction Execution Times, page II-33.

<u>Addition (continued)</u>			GE-625 Timing (μ sec) [/]	GE-635 Timing (μ sec) [/]	Referen (Page)
DFAD	477	Double-Precision Floating Add	3.0	2.7	II-99
DUFA	437	Double-Precision Unnormalized Floating Add	3.0	2.5	99
ADE	415	Add to Exponent Register	3.0	1.8	100
 <u>Subtraction</u>					
FSB	575	Floating Subtract	3.0	2.7	100
UFS	535	Unnormalized Floating Subtract	3.0	2.5	101
DFSB	577	Double-Precision Floating Subtract	3.0	2.7	101
DUFS	537	Double-Precision Unnormalized Floating Subtract	3.0	2.5	102
 <u>Multiplication</u>					
FMP	461	Floating Multiply	6.0	5.9	102
UFM	421	Unnormalized Floating Multiply	6.0	5.7	103
DFMP	463	Double-Precision Floating Multiply	12.0	11.7	103
DUFM	423	Double-Prec. Unnormal. Float. Multiply	12.0	11.5	104
 <u>Division</u>					
FDV	565	Floating Divide	14.5*	14.2*	105
FDI	525	Floating Divide Inverted	14.5*	14.2*	106
DFDV	567	Double-Precision Floating Divide	23.5*	23.2*	107
DFDI	527	Double-Prec. Float. Divide Inverted	23.5*	23.2*	108
 <u>Negate, Normalize</u>					
FNEG	513	Floating Negate	3.0	2.3	109
FNO	573	Floating Normalize	3.0	2.3	109
 <u>Compare</u>					
FCMP	515	Floating Compare	3.0	2.1	110
FCMG	425	Floating Compare Magnitude	3.0	2.1	111
DFCMP	517	Double-Precision Floating Compare	3.0	2.1	112
DFCMG	427	Double-Prec. Float. Compare Magnitude	3.0	2.1	113
FSZN	430	Floating Set Zero and Negative Indicators from Memory	3.0	1.8	114

[/] See Calculation of Instruction Execution Times, page II-33.

* When actual division does not take place, GE-635 2.5 μ sec, GE-625 2.8 μ sec.

TRANSFER OF CONTROL			GE-625 Timing (μ sec) [‡]	GE-635 Timing (μ sec) [‡]	Reference (Page)
<u>Transfer</u>					
TRA	710	Transfer Unconditionally	2.0	1.7	II-115
TSX _n	70n	Transfer and Set X _n	3.0	1.8	115
TSS	715	Transfer and Set Slave Mode	2.0	1.7	115
RET	630	Return	4.0	3.3	116

Conditional Transfer

TZE	600	Transfer on Zero	2.0	1.7	117
TNZ	601	Transfer on Not Zero	2.0	1.7	117
TMI	604	Transfer on Minus	2.0	1.7	117
TPL	605	Transfer on Plus	2.0	1.7	117
TRC	603	Transfer on Carry	2.0	1.7	118
TNC	602	Transfer on No Carry	2.0	1.7	118
TOV	617	Transfer on Overflow	2.0	1.7	118
TEO	614	Transfer on Exponent Overflow	2.0	1.7	119
TEU	615	Transfer on Exponent Underflow	2.0	1.7	119
TTF	607	Transfer on Tally-Runout Indicator OFF	2.0	1.7	119

MISCELLANEOUS OPERATIONS

NOP	011	No Operation	2.0	1.1	120
DIS	616	Delay Until Interrupt Signal	2.0	1.7	120
BCD	505	Binary to Binary-Coded-Decimal	4.0	3.4	120
GTB	774	Gray to Binary	9.0	8.5	121
XEC	716	Execute	2.0	1.7	122
XED	717	Execute Double	2.0	1.7	122
MME	001	Master Mode Entry	3.0	2.3	123
DRL	022	Derail	3.0	2.3	124
RPT	520	Repeat	2.0	1.3	125
RPD	560	Repeat Double	2.0	1.3	127
RPL	500	Repeat Link	2.0	1.3	129

[‡] See Calculation of Instruction Execution Times, page II-33.


MASTER MODE OPERATIONS			GE-625 Timing (μ sec) [†]	GE-635 Timing (μ sec) [†]	Reference (Page)
<u>Master Mode</u>					
LBAR	230	Load Base Address Register	3.0	1.8	II-132
LDT	637	Load Timer Register	3.0	1.8	132
SMIC	451	Set Memory Controller Interrupt Cells	3.0	1.8	132
<u>Master Mode and Control Processor</u>					
RMC	233	Read Memory Controller Mask Registers	3.0	1.9	133
RMFP	633	Read Memory File Protect Register	3.0	1.9	134
SMCM	553	Set Memory Controller Mask Registers	3.0	1.8	135
SMFP	453	Set Memory File Protect Register	3.0	1.8	136
CIOC	015	Connect I/O Channel	3.0	1.8	137

[†] See Calculation of Instruction Execution Times, page II-33.

APPENDIX B. GE-625/635 MNEMONICS
IN ALPHABETICAL ORDER WITH PAGE REFERENCES

Mnemonic:	Page:	Mnemonic:	Page:	Mnemonic:	Page:	Mnemonic:	Page:
ADA	II-59	DFAD	II-99	LCXn	II-43	SBQ	II-67
ADAQ	60	DFCMG	113	LDA	39	SBXn	68
ADE	100	DFCMP	112	LDAQ	39	SMCM	135
ADL	66	DFDI	108	LDE	96	SMFP	136
ADLA	62	DFDV	107	LDI	45	SMIC	132
ADLAQ	63	DFLD	96	LDT	132	SREG	47
ADLQ	63	DFMP	103	LDQ	39	SSA	69
ADLXn	64	DFSB	101	LDXn	40	SSQ	69
ADQ	59	DFST	97	LLR	58	SSXn	70
ADXn	60	DIS	120	LLS	56	STA	46
ALR	57	DIV	76	LREG	40	STAQ	46
ALS	55	DRL	124	LRL	57	STBA	49
ANA	79	DUFA	99	LRS	54	STBQ	50
ANAQ	79	DUFM	104	MME	123	STC1	53
ANQ	79	DUFS	102	MPF	75	STC2	53
ANSA	80	DVF	77	MPY	74	STCA	47
ANSQ	80					STCQ	48
ANSXn	81	EAA	43	NEG	78	STE	97
ANXn	80	EAQ	44	NEGL	78	STI	51
AOS	66	EAXn	44	NOP	120	STQ	46
ARL	56	ERA	83			STT	52
ARS	54	ERAQ	84	ORA	81	STXn	46
ASA	61	ERQ	84	ORAQ	82	STZ	52
ASQ	61	ERSA	85	ORQ	81	SWCA	72
ASXn	62	ERSQ	85	ORSA	82	SWCQ	73
AWCA	64	ERSXn	85	ORSQ	83	SZN	91
AWCQ	65	ERXn	84	ORSXn	83		
				ORXn	82	TEO	119
BCD	120	FAD	98			TEU	119
		FCMG	111	QLR	57	TMI	117
CANA	93	FCMP	110	QLS	55	TNC	118
CANAQ	93	FDI	106	QRL	56	TNZ	117
CANQ	93	FDV	105	QRS	54	TOV	118
CANXn	94	FLD	96			TPL	117
CIOC	137	FMP	102	RET	116	TRA	115
CMG	91	FNEG	109	RMCM	133	TRC	118
CMK	92	FNO	109	RMFP	134	TSS	115
CMPA	86	FSB	100	RPD	127	TSXn	115
CMPAQ	88	FST	97	RPL	129	TTF	119
CMPQ	87	FSZN	114	RPT	125	TZE	117
CMPXn	89						
CNAA	94	GTB	121	SBA	67	UFA	98
CNAAQ	95			SBAQ	68	UFM	103
CNAQ	94	LBAR	132	SBAR	52	UFS	101
CNAXn	95	LCA	41	SBLA	70		
CWL	90	LCAQ	42	SBLAQ	71	XEC	122
		LCQ	42	SBLQ	71	XED	122
				SBLXn	72		

APPENDIX C. GE-625/635 INSTRUCTION MNEMONICS CORRELATED WITH THEIR OPERATION CODES

GE-625/635 Mnemonics and Operation Codes										GENERAL  ELECTRIC COMPUTER DEPARTMENT						
000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017	
000		MME	DRL						NOP				CIQC			
020	ADLX0	ADLX1	ADLX2	ADLX3	ADLX4	ADLX5	ADLX6	ADLX7			ADL		ADLA	ADLQ	ADLAQ	
040	ASX0	ASX1	ASX2	ASX3	ASX4	ASX5	ASX6	ASX7				AOS	ASA	ASQ		
060	ADX0	ADX1	ADX2	ADX3	ADX4	ADX5	ADX6	ADX7	AWCA	AWCQ	IREG		ADA	ADQ	ADAQ	
100	CMPX0	CMPX1	CMPX2	CMPX3	CMPX4	CMPX5	CMPX6	CMPX7	CWL				CMPA	CMPQ	CMPAQ	
120	SBLX0	SBLX1	SBLX2	SBLX3	SBLX4	SBLX5	SBLX6	SBLX7					SBLA	SBLQ	SBLAQ	
140	SSX0	SSX1	SSX2	SSX3	SSX4	SSX5	SSX6	SSX7					SSA	SSQ		
160	SBX0	SBX1	SBX2	SBX3	SBX4	SBX5	SBX6	SBX7	SWCA	SWCQ			SBA	SBQ	SBAQ	
200	CNAX0	CNAX1	CNAX2	CNAX3	CNAX4	CNAX5	CNAX6	CNAX7	CMK				CNAA	CNAQ	CNAAQ	
220	LDX0	LDX1	LDX2	LDX3	LDX4	LDX5	LDX6	LDX7	LBAR		RMCM	SZN	LDA	LDAQ	LDAQ	
240	ORSX0	ORSX1	ORSX2	ORSX3	ORSX4	ORSX5	ORSX6	ORSX7					ORSA	ORSQ		
260	ORX0	ORX1	ORX2	ORX3	ORX4	ORX5	ORX6	ORX7					ORA	ORQ	ORAQ	
300	CANX0	CANX1	CANX2	CANX3	CANX4	CANX5	CANX6	CANX7					CANA	CANQ	CANAQ	
320	LCX0	LCX1	LCX2	LCX3	LCX4	LCX5	LCX6	LCX7					LCA	LCQ	LCAQ	
340	ANSX0	ANSX1	ANSX2	ANSX3	ANSX4	ANSX5	ANSX6	ANSX7					ANSA	ANSQ		
360	ANX0	ANX1	ANX2	ANX3	ANX4	ANX5	ANX6	ANX7					ANA	ANQ	ANAQ	
400		MPF	MPY			CMG				LDE			ADE			
420		UFM		DUFM		FCMG		DFCMG	FSZN	FLD			UFA			
440									STZ	SMIC		DFLD	FST	STE	DUFM	
460		FMP		DFMP								SMFP	FAD	DEFM	DFM	
500	RPL					BCD	DIV	DVF					FCMP		DFCMP	
520	RPT					FDI		DFDI					UFS		DUFM	
540									SBAR	NEG			NEGL			
560	RPD					FDV		DFDV		STBA	STBQ		SMCM	STCI	DFSB	
													FNQ			
600	TZE	TNZ	TNC	TRC	TMI	TPL	EAX5	EAX7				TEQ	TEU	DES	TQV	
620	EAX0	EAX1	EAX2	EAX3	EAX4	EAX5	EAX6	EAX7	RET			RMFP	LDI	EAQ	LDT	
640	ERSX0	ERSX1	ERSX2	ERSX3	ERSX4	ERSX5	ERSX6	ERSX7					ERSA	ERSQ		
660	ERX0	ERX1	ERX2	ERX3	ERX4	ERX5	ERX6	ERX7					ERR	ERQ	ERRAQ	
700	TSX0	TSX1	TSX2	TSX3	TSX4	TSX5	TSX6	TSX7	TRA				TSS	XEC	XED	
720										ARS			ALS	QLS	LIS	
740	STX0	STX1	STX2	STX3	STX4	STX5	STX6	STX7	STC2	STCA	QRS	QRS	STCQ	STQ	STAQ	
760										ARL	QRL	LRG	TRL	QLR	TLR	
											TRL	GTB				
000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017	

APPENDIX D. PSEUDO-OPERATIONS
BY FUNCTIONAL CLASS WITH PAGE REFERENCES

PSEUDO-OPERATIONS

PSEUDO-OPERATION MNEMONIC	FUNCTIONS	PAGE
CONTROL PSEUDO-OPERATIONS		
DETAIL ON/OFF	(Detail output listing)	III-28
LIST ON/OFF	(Control output listing)	29
PCC ON/OFF	(Print control cards)	29
INHIB ON/OFF	(Inhibit interrupts)	30
PMC ON/OFF	(Print MACRO expansion)	30
REF ON/OFF	(References)	30
PUNCH ON/OFF	(Control card output)	31
EDITP	(Edit Print Lines)	31
EJECT	(Restore output listing)	32
REM	(Remarks)	32
*	(* in column one -- remarks)	32
LBL	(Label)	33
TTL	(Title)	33
TTLS	(Subtitle)	34
ABS	(Output absolute text)	34
FUL	(Output full binary text)	35
TCD	(Punch transfer card)	35
HEAD	(Heading)	35
DCARD	(Punch BCD Card)	37
END	(End of assembly)	38
OPD	(Operation definition)	38
OPSYN	(Operation synonym)	40
LOCATION COUNTER PSEUDO-OPERATIONS		
USE	(Use multiple location counters)	41
BEGIN	(Origin of a location counter)	41
ORG	(Origin set by programmer)	42
LOC	(Location of output text)	42
SYMBOL DEFINING PSEUDO-OPERATIONS		
EQU	(Equal to)	43
FEQU	(Equal to symbol as yet undefined)	44
BOOL	(Boolean)	44
SET	(Symbol redefinition)	45
MIN	(Minimum)	45
MAX	(Maximum)	46
SYMDEF	(Symbol definition)	46
SYMREF	(Symbol reference)	47
NULL	(Symbol EQU*)	48
EVEN	(Force Location Counter Even)	48
ODD	(Force Location Counter Odd)	48
EIGHT	(Force Location Counter to Multiple of 8)	49

PSEUDO-OPERATIONS

PSEUDO-OPERATION MNEMONIC	FUNCTION	PAGE NUMBER
DATA GENERATING PSEUDO-OPERATIONS		
OCT	(Octal)	III-49
DEC	(Decimal)	50
BCI	(Binary Coded Decimal Information)	52
VFD	(Variable field definition)	53
DUP	(Duplicate cards)	55
STORAGE ALLOCATION PSEUDO-OPERATIONS		
BSS	(Block started by symbol)	56
BFS	(Block followed by symbol)	56
BLOCK	(Block common)	57
LIT	(Literal Pool Origin)	57
CONDITIONAL PSEUDO-OPERATIONS		
INE	(If not equal)	58
IFE	(If equal)	59
IFL	(If less than)	59
IFG	(If greater than)	59
SPECIAL WORD FORMATS		
ARG	(Argument--generate zero operation code computer word)	60
NONOP	(Undefined Operation)	60
ZERO	(Generate one word with two specified 18-bit fields)	60
MAXSZ	(Maximum size of assembly)	61
ADDRESS TALLY PSEUDO-OPERATIONS		
TALLY	(Tally--ID, DI, SC, and CI variations)	61
TALLYB	(Tally--SC and CI for 9 bit bytes)	61
TALLYD	(Tally and Delta)	61
TALLYC	(Tally and Continue)	61
REPEAT INSTRUCTION CODING FORMATS		
RPT	(Repeat)	62
RPTX	(Repeat using index register zero)	62
RPD	(Repeat Double)	62
RPDX	(Repeat Double using index register zero)	62
RPDA	(Repeat Double using first instruction only)	62
RPDB	(Repeat Double using second instruction only)	62
RPL	(Repeat Link)	62
RPLX	(Repeat Link using index register zero)	63

PSEUDO-OPERATIONS

PSEUDO-OPERATION MNEMONIC	FUNCTION	PAGE NUMBER
<hr/>		
MACRO PSEUDO-OPERATIONS		
MACRO	(Begin MACRO prototype)	64
ENDM	(End MACRO prototype)	64
CRSM ON/OFF	(Create symbols)	71
ORGCSM	(Origin Created Symbols)	71
IDRP	(Indefinite repeat)	71
DELM	(Delete a MACRO)	72
PUNM	(Punch MACRO Prototypes)	73
LODM	(Load MACRO Prototypes)	74
PROGRAM LINKAGE PSEUDO-OPERATIONS		
CALL	(Call--subroutines)	76
SAVE	(Subroutine entry point)	77
RETURN	(Return--from subroutines)	78
ERLK	(Error Linkage--between subroutines)	79

APPENDIX E. MASTER MODE ENTRY SYSTEM SYMBOLS AND INPUT/OUTPUT OPERATIONS

SYSTEM SYMBOLS

The Assembler recognizes the following group of system symbols when the programmer enters any of them in the variable field of the Master Mode Entry (MME) machine instruction. (See Chapter II.) These MME instructions then serve as interfaces between the GEFLOW and GESERV modules of the Comprehensive Operating Supervisor for special purposes (suggested in the meanings in the list following).

The table below indicates the system mnemonic symbol, its meaning, and the associated decimal value substituted in the MME address field by the Assembler.

SYMBOL	MEANING	DECIMAL VALUE
GEINOS	Input/Output Initiation	1
GEROAD	Roadblock	2
GEFADD	Physical File Address Request	3
GERELS	Component Release	4
GESNAP	Snapshot Dump	5
GELAPS	(Elapsed) Time Request	6
GEFINI	Terminal Transfer to Monitor	7
GEBORT	Aborting of Programs	8
GEMORE	Request for additional memory or peripherals	9
GEFCON	File Control Block Request	10
GEFELS	File Switching Request	11
GESETS	Set Switch Request	12
GERETS	Reset Switch Request	13
GEENDC	Terminate Courtesy Call	14
GERELC	Relinquit Control	15
GESPEC	Special Interrupt Courtesy Call Request	16
GETIME	Date and Time-of-Day Request	17
GECALL	System Loader	18
GESAVE	Write File in System Format	19
GERSTR	Read File in System Format	20
GEMREL	Release Memory	21
GESYOT	Write on SYSOUT	22
GECHK	Check Point	23
GEROUT	Output to Remote Terminal	24
GEROLL	Reinitiate or Rollback Program	25

INPUT/OUTPUT COMMAND FORMATS

The following listing of input/output commands are for use when coding directly to Input/Output Supervisor within the Comprehensive Operating Supervisor.

Designators used in the listing below are:

- XXXX = 0000 for Slave Mode programs
- XXXX = physical device code for Master Mode programs
- DA = Device Address (Used only in Master Mode programs; see input/output select sequence coding, Operating Supervisor reference manual.)
- CA = Channel Address (Used only in Master Mode programs; see input/output select sequence coding, Operating Supervisor reference manual.)
- NN = number of records (01-63)
- = 01 when subfield for NN is blank
- CC = octal character to be used as file mark

<u>COMMAND DESCRIPTION</u>	<u>PSEUDO- OPERATION</u>	<u>VARIABLE FIELD</u>	<u>OCTAL REPRESENTATION</u>
Request Status	REQS	DA, CA	40 XXXX 020001
Reset Status	RESS	DA, CA	00 XXXX 020001
Read Card Binary	RCB	DA, CA	01 XXXX 000000
Read Card Decimal	RCD	DA, CA	02 XXXX 000000
Read Card Mixed	RCM	DA, CA	03 XXXX 000000
Write Card Binary	WCB	DA, CA	11 XXXX 040014
Write Card Decimal	WCD	DA, CA	12 XXXX 040014
Write Card Decimal Edited	WCDE	DA, CA	13 XXXX 040014
Write Printer	WPR	DA, CA	10 XXXX 000000
Write Printer Edited	WPRE	DA, CA	30 XXXX 000000
Read Tape Binary	RTB	DA, CA	05 XXXX 000000
Read Tape Decimal	RTD	DA, CA	04 XXXX 000000
Write Tape Binary	WTB	DA, CA	15 XXXX 000000
Write Tape Decimal	WTD	DA, CA	14 XXXX 000000
Write End-of-File	WEF	DA, CA	14 XXXX 101700
Write File Mark	WFM	CC, DA, CA	15 XXXX 10CC00
Write File Mark Decimal	WFMD	CC, DA, CA	14 XXXX 10CC00
Erase	ERASE	DA, CA	54 XXXX 020001
Backspace Record (s)	BSR	N, DA, CA	46 XXXX 0200NN
Backspace File	BSF	DA, CA	47 XXXX 020001
Forward Space Record (s)	FSR	N, DA, CA	44 XXXX 0200NN
Forward Space File	FSF	DA, CA	45 XXXX 020001
Rewind	REW	DA, CA	70 XXXX 020001

<u>COMMAND DESCRIPTION</u>	<u>PSEUDO- OPERATION</u>	<u>VARIABLE FIELD</u>	<u>OCTAL REPRESENTATION</u>
Rewind and Standby	REWS	DA, CA	72 XXXX 020001
Set Low Density	SLD	DA, CA	61 XXXX 020001
Set High Density	SHD	DA, CA	60 XXXX 020001
Seek Disc Address	SDIA	DA, CA	34 XXXX 000002
Read Disc Continuous	RDIC	DA, CA	25 XXXX 002400
Write Disc Continuous	WDIC	DA, CA	31 XXXX 002400
Write Disc Continuous and Verify	WDICV	DA, CA	33 XXXX 002400
Select Drum Address	SDRA	DA, CA	34 XXXX 000002
Read Drum	RDR	DA, CA	25 XXXX 000000
Write Drum	WDR	DA, CA	31 XXXX 000000
Write Drum and Verify	WDRV	DA, CA	33 XXXX 000000
Drum Compare and Verify	DRCV	DA, CA	11 XXXX 000000
Read Perforated Tape	RDPT	DA, CA	02 XXXX 000000
Write Perforated Tape	WPT	DA, CA	11 XXXX 000000
Write Perforated Tape Edited	WPTE	DA, CA	31 XXXX 000000
Write Perforated Tape--Single Character	WPTSC	DA, CA	16 XXXX 000000
Write Perforated Tape--Double Character	WPTDC	DA, CA	13 XXXX 000000
Read Typewriter	RTYP	DA, CA	03 XXXX 000000
Write Typewriter	WTYP	DA, CA	13 XXXX 000000
Read DATANET-30	RDN	DA, CA	01 XXXX 000000
Write DATANET-30	WDN	DA, CA	10 XXXX 000000

DATA CONTROL WORD FORMATS

The Data Control Word format listing below contains designators as follows:

A = address of the data block
C = word count of data to be transferred per block
XXXX = ignored by the Assembler

<u>DESCRIPTION</u>	<u>PSEUDO- OPERATION</u>	<u>VARIABLE FIELD</u>	<u>OCTAL REPRESENTATION</u>
Transmit and Disconnect	IOTD	A, C	AAAAAA00CCCC
Transmit and Proceed	IOTP	A, C	AAAAAA01CCCC
Non-Transmit and Proceed	IONTP	A, C	AAAAAA03CCCC
Transfer to Data Control Word	TDCW	A	AAAAAA02XXXX

APPENDIX F. GE-625/635 STANDARD CHARACTER SET

GE-625/635 STANDARD CHARACTER SET

Standard Character Set	GE-Internal Machine Code	Octal Code	Hollerith Card Code	Standard Character Set	GE-Internal Machine Code	Octal Code	Hollerith Card Code
0	00 0000	00	0	↑	10 0000	40	11-0
1	00 0001	01	1	J	10 0001	41	11-1
2	00 0010	02	2	K	10 0010	42	11-2
3	00 0011	03	3	L	10 0011	43	11-3
4	00 0100	04	4	M	10 0100	44	11-4
5	00 0101	05	5	N	10 0101	45	11-5
6	00 0110	06	6	O	10 0110	46	11-6
7	00 0111	07	7	P	10 0111	47	11-7
8	00 1000	10	8	Q	10 1000	50	11-8
9	00 1001	11	9	R	10 1001	51	11-9
[00 1010	12	2-8	-	10 1010	52	11
#	00 1011	13	3-8	\$	10 1011	53	11-3-8
@	00 1100	14	4-8	*	10 1100	54	11-4-8
:	00 1101	15	5-8)	10 1101	55	11-5-8
>	00 1110	16	6-8	;	10 1110	56	11-6-8
?	00 1111	17	7-8	'	10 1111	57	11-7-8
␣	01 0000	20	(blank)	+	11 0000	60	12-0
A	01 0001	21	12-1	/	11 0001	61	0-1
B	01 0010	22	12-2	S	11 0010	62	0-2
C	01 0011	23	12-3	T	11 0011	63	0-3
D	01 0100	24	12-4	U	11 0100	64	0-4
E	01 0101	25	12-5	V	11 0101	65	0-5
F	01 0110	26	12-6	W	11 0110	66	0-6
G	01 0111	27	12-7	X	11 0111	67	0-7
H	01 1000	30	12-8	Y	11 1000	70	0-8
I	01 1001	31	12-9	Z	11 1001	71	0-9
&	01 1010	32	12	←	11 1010	72	0-2-8
.	01 1011	33	12-3-8	,	11 1011	73	0-3-8
]	01 1100	34	12-4-8	%	11 1100	74	0-4-8
(01 1101	35	12-5-8	=	11 1101	75	0-5-8
<	01 1110	36	12-6-8	"	11 1110	76	0-6-8
\	01 1111	37	12-7-8	!	11 1111	77	0-7-8

APPENDIX G. CONVERSION TABLE OF OCTAL-DECIMAL INTEGERS AND FRACTIONS

Octal	10000	20000	30000	40000	50000	60000	70000
Decimal	4096	8192	12288	16384	20480	24576	28672

Octal	100000	200000	300000	400000	500000	600000	700000	1000000
Decimal	32768	65536	98304	131072	163840	196608	229376	262144

Octal	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

Octal	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0630	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767

Octal	0400 to 0777
Decimal	0256 to 0511

Octal	0	1	2	3	4	5	6	7
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

Octal	1400 to 1777
Decimal	0768 to 1023

Octal	0	1	2	3	4	5	6	7
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

OCTAL-DECIMAL INTEGER CONVERSION TABLE (Cont.)

Octal	10000	20000	30000	40000	50000	60000	70000
Decimal	4096	8192	12288	16384	20480	24576	28672

Octal	100000	200000	300000	400000	500000	600000	700000	1000000
Decimal	32768	65536	98304	131072	163840	196608	229376	262144

Octal	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1088	1089	1090	1091	1092	1093	1094	1095
2110	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

Octal	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	1542	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

Octal	2400 to 2777
Decimal	1280 to 1535

Octal	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1509	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

Octal	3400 to 3777
Decimal	1792 to 2047

Octal	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

OCTAL-DECIMAL INTEGER CONVERSION TABLE (Cont.)

Octal	10000	20000	30000	40000	50000	60000	70000
Decimal	4096	8192	12288	16384	20480	24576	28672

Octal	100000	200000	300000	400000	500000	600000	700000	1000000
Decimal	32768	65536	98304	131072	163840	196608	229376	262144

Octal	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

Octal	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

Octal	4400 to 4777
Decimal	2304 to 2559

Octal	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

Octal	5400 to 5777
Decimal	2816 to 3071

Octal	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

OCTAL-DECIMAL INTEGER CONVERSION TABLE (Cont.)

Octal	10000	20000	30000	40000	50000	60000	70000
Decimal	4096	8192	12288	16384	20480	24576	28672

Octal	100000	200000	300000	400000	500000	600000	700000	1000000
Decimal	32768	65536	98304	131072	163840	196608	229376	262144

Octal	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

Octal	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3596	3597	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

Octal	6400 to 6777
Decimal	3328 to 3583

Octal	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

Octal	7400 to 7777
Decimal	3840 to 4095

Octal	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	4022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

OCTAL-DECIMAL FRACTION CONVERSION TABLE

OCTAL	DECIMAL	OCTAL	DECIMAL	OCTAL	DECIMAL	OCTAL	DECIMAL
.000	.000000	.100	.125000	.200	.250000	.300	.375000
.001	.001953	.101	.126953	.201	.251953	.301	.376953
.002	.003906	.102	.128906	.202	.253906	.302	.378906
.003	.005859	.103	.130859	.203	.255859	.303	.380859
.004	.007812	.104	.132812	.204	.257812	.304	.382812
.005	.009765	.105	.134765	.205	.259765	.305	.384765
.006	.011718	.106	.136718	.206	.261718	.306	.386718
.007	.013671	.107	.138671	.207	.263671	.307	.388671
.010	.015625	.110	.140625	.210	.265625	.310	.390625
.011	.017578	.111	.142578	.211	.267578	.311	.392578
.012	.019531	.112	.144531	.212	.269531	.312	.394531
.013	.021484	.113	.146484	.213	.271484	.313	.396484
.014	.023437	.114	.148437	.214	.273437	.314	.398437
.015	.025390	.115	.150390	.215	.275390	.315	.400390
.016	.027343	.116	.152343	.216	.277343	.316	.402343
.017	.029296	.117	.154296	.217	.279296	.317	.404296
.020	.031250	.120	.156250	.220	.281250	.320	.406250
.021	.033203	.121	.158203	.221	.283203	.321	.408203
.022	.035156	.122	.160156	.222	.285156	.322	.410156
.023	.037109	.123	.162109	.223	.287109	.323	.412109
.024	.039062	.124	.164062	.224	.289062	.324	.414062
.025	.041015	.125	.166015	.225	.291015	.325	.416015
.026	.042968	.126	.167968	.226	.292968	.326	.417968
.027	.044921	.127	.169921	.227	.294921	.327	.419921
.030	.046875	.130	.171875	.230	.296875	.330	.421875
.031	.048828	.131	.173828	.231	.298828	.331	.423828
.032	.050781	.132	.175781	.232	.300781	.332	.425781
.033	.052734	.133	.177734	.233	.302734	.333	.427734
.034	.054687	.134	.179687	.234	.304687	.334	.429687
.035	.056640	.135	.181640	.235	.306640	.335	.431640
.036	.058593	.136	.183593	.236	.308593	.336	.433593
.037	.060546	.137	.185546	.237	.310546	.337	.435546
.040	.062500	.140	.187500	.240	.312500	.340	.437500
.041	.064453	.141	.189453	.241	.314453	.341	.439453
.042	.066406	.142	.191406	.242	.316406	.342	.441406
.043	.068359	.143	.193359	.243	.318359	.343	.443359
.044	.070312	.144	.195312	.244	.320312	.344	.445312
.045	.072265	.145	.197265	.245	.322265	.345	.447265
.046	.074218	.146	.199218	.246	.324218	.346	.449218
.047	.076171	.147	.201171	.247	.326171	.347	.451171
.050	.078125	.150	.203125	.250	.328125	.350	.453125
.051	.080078	.151	.205078	.251	.330078	.351	.455078
.052	.082031	.152	.207031	.252	.332031	.352	.457031
.053	.083984	.153	.208984	.253	.333984	.353	.458984
.054	.085937	.154	.210937	.254	.335937	.354	.460937
.055	.087890	.155	.212890	.255	.337890	.355	.462890
.056	.089843	.156	.214843	.256	.339843	.356	.464843
.057	.091796	.157	.216796	.257	.341796	.357	.466796
.060	.093750	.160	.218750	.260	.343750	.360	.468750
.061	.095703	.161	.220703	.261	.345703	.361	.470703
.062	.097656	.162	.222656	.262	.347656	.362	.472656
.063	.099609	.163	.224609	.263	.349609	.363	.474609
.064	.101562	.164	.226562	.264	.351562	.364	.476562
.065	.103515	.165	.228515	.265	.353515	.365	.478515
.066	.105468	.166	.230468	.266	.355468	.366	.480468
.067	.107421	.167	.232421	.267	.357421	.367	.482421
.070	.109375	.170	.234375	.270	.359375	.370	.484375
.071	.111328	.171	.236328	.271	.361328	.371	.486328
.072	.113281	.172	.238281	.272	.363281	.372	.488281
.073	.115234	.173	.240234	.273	.365234	.373	.490234
.074	.117187	.174	.242187	.274	.367187	.374	.492187
.075	.119140	.175	.244140	.275	.369140	.375	.494140
.076	.121093	.176	.246093	.276	.371093	.376	.496093
.077	.123046	.177	.248046	.277	.373046	.377	.498046

OCTAL-DECIMAL FRACTION CONVERSION TABLE (Cont.)

OCTAL	DECIMAL	OCTAL	DECIMAL	OCTAL	DECIMAL	OCTAL	DECIMAL
.000000	.000000	.000100	.000244	.000200	.000488	.000300	.000732
.000001	.000003	.000101	.000247	.000201	.000492	.000301	.000736
.000002	.000007	.000102	.000251	.000202	.000495	.000302	.000740
.000003	.000011	.000103	.000255	.000203	.000499	.000303	.000743
.000004	.000015	.000104	.000259	.000204	.000503	.000304	.000747
.000005	.000019	.000105	.000263	.000205	.000507	.000305	.000751
.000006	.000022	.000106	.000267	.000206	.000511	.000306	.000755
.000007	.000026	.000107	.000270	.000207	.000514	.000307	.000759
.000010	.000030	.000110	.000274	.000210	.000518	.000310	.000762
.000011	.000034	.000111	.000278	.000211	.000522	.000311	.000766
.000012	.000038	.000112	.000282	.000212	.000526	.000312	.000770
.000013	.000041	.000113	.000286	.000213	.000530	.000313	.000774
.000014	.000045	.000114	.000289	.000214	.000534	.000314	.000778
.000015	.000049	.000115	.000293	.000215	.000537	.000315	.000782
.000016	.000053	.000116	.000297	.000216	.000541	.000316	.000785
.000017	.000057	.000117	.000301	.000217	.000545	.000317	.000789
.000020	.000061	.000120	.000305	.000220	.000549	.000320	.000793
.000021	.000064	.000121	.000308	.000221	.000553	.000321	.000797
.000022	.000068	.000122	.000312	.000222	.000556	.000322	.000801
.000023	.000072	.000123	.000316	.000223	.000560	.000323	.000805
.000024	.000076	.000124	.000320	.000224	.000564	.000324	.000808
.000025	.000080	.000125	.000324	.000225	.000568	.000325	.000812
.000026	.000083	.000126	.000328	.000226	.000572	.000326	.000816
.000027	.000087	.000127	.000331	.000227	.000576	.000327	.000820
.000030	.000091	.000130	.000335	.000230	.000579	.000330	.000823
.000031	.000095	.000131	.000339	.000231	.000583	.000331	.000827
.000032	.000099	.000132	.000343	.000232	.000587	.000332	.000831
.000033	.000102	.000133	.000347	.000233	.000591	.000333	.000835
.000034	.000106	.000134	.000350	.000234	.000595	.000334	.000839
.000035	.000110	.000135	.000354	.000235	.000598	.000335	.000843
.000036	.000114	.000136	.000358	.000236	.000602	.000336	.000846
.000037	.000118	.000137	.000362	.000237	.000606	.000337	.000850
.000040	.000122	.000140	.000366	.000240	.000610	.000340	.000854
.000041	.000125	.000141	.000370	.000241	.000614	.000341	.000858
.000042	.000129	.000142	.000373	.000242	.000617	.000342	.000862
.000043	.000133	.000143	.000377	.000243	.000621	.000343	.000865
.000044	.000137	.000144	.000381	.000244	.000625	.000344	.000869
.000045	.000141	.000145	.000385	.000245	.000629	.000345	.000873
.000046	.000144	.000146	.000389	.000246	.000633	.000346	.000877
.000047	.000148	.000147	.000392	.000247	.000637	.000347	.000881
.000050	.000152	.000150	.000396	.000250	.000640	.000350	.000885
.000051	.000156	.000151	.000400	.000251	.000644	.000351	.000888
.000052	.000160	.000152	.000404	.000252	.000648	.000352	.000892
.000053	.000164	.000153	.000408	.000253	.000652	.000353	.000896
.000054	.000167	.000154	.000411	.000254	.000656	.000354	.000900
.000055	.000171	.000155	.000415	.000255	.000659	.000355	.000904
.000056	.000175	.000156	.000419	.000256	.000663	.000356	.000907
.000057	.000179	.000157	.000423	.000257	.000667	.000357	.000911
.000060	.000183	.000160	.000427	.000260	.000671	.000360	.000915
.000061	.000186	.000161	.000431	.000261	.000675	.000361	.000919
.000062	.000190	.000162	.000434	.000262	.000679	.000362	.000923
.000063	.000194	.000163	.000438	.000263	.000682	.000363	.000926
.000064	.000198	.000164	.000442	.000264	.000686	.000364	.000930
.000065	.000202	.000165	.000446	.000265	.000690	.000365	.000934
.000066	.000205	.000166	.000450	.000266	.000694	.000366	.000938
.000067	.000209	.000167	.000453	.000267	.000698	.000367	.000942
.000070	.000213	.000170	.000457	.000270	.000701	.000370	.000946
.000071	.000217	.000171	.000461	.000271	.000705	.000371	.000949
.000072	.000221	.000172	.000465	.000272	.000709	.000372	.000953
.000073	.000225	.000173	.000469	.000273	.000713	.000373	.000957
.000074	.000228	.000174	.000473	.000274	.000717	.000374	.000961
.000075	.000232	.000175	.000476	.000275	.000720	.000375	.000965
.000076	.000236	.000176	.000480	.000276	.000724	.000376	.000968
.000077	.000240	.000177	.000484	.000277	.000728	.000377	.000972

OCTAL-DECIMAL FRACTION CONVERSION TABLE (Cont.)

OCTAL	DECIMAL	OCTAL	DECIMAL	OCTAL	DECIMAL	OCTAL	DECIMAL
.000400	.000976	.000500	.001220	.000600	.001464	.000700	.001708
.000401	.000980	.000501	.001224	.000601	.001468	.000701	.001712
.000402	.000984	.000502	.001228	.000602	.001472	.000702	.001716
.000403	.000988	.000503	.001232	.000603	.001476	.000703	.001720
.000404	.000991	.000504	.001235	.000604	.001480	.000704	.001724
.000405	.000995	.000505	.001239	.000605	.001483	.000705	.001728
.000406	.000999	.000506	.001243	.000606	.001487	.000706	.001731
.000407	.001003	.000507	.001247	.000607	.001491	.000707	.001735
.000410	.001007	.000510	.001251	.000610	.001495	.000710	.001739
.000411	.001010	.000511	.001255	.000611	.001499	.000711	.001743
.000412	.001014	.000512	.001258	.000612	.001502	.000712	.001747
.000413	.001018	.000513	.001262	.000613	.001506	.000713	.001750
.000414	.001022	.000514	.001266	.000614	.001510	.000714	.001754
.000415	.001026	.000515	.001270	.000615	.001514	.000715	.001758
.000416	.001029	.000516	.001274	.000616	.001518	.000716	.001762
.000417	.001033	.000517	.001277	.000617	.001522	.000717	.001766
.000420	.001037	.000520	.001281	.000620	.001525	.000720	.001770
.000421	.001041	.000521	.001285	.000621	.001529	.000721	.001773
.000422	.001045	.000522	.001289	.000622	.001533	.000722	.001777
.000423	.001049	.000523	.001293	.000623	.001537	.000723	.001781
.000424	.001052	.000524	.001296	.000624	.001541	.000724	.001785
.000425	.001056	.000525	.001300	.000625	.001544	.000725	.001789
.000426	.001060	.000526	.001304	.000626	.001548	.000726	.001792
.000427	.001064	.000527	.001308	.000627	.001552	.000727	.001796
.000430	.001068	.000530	.001312	.000630	.001556	.000730	.001800
.000431	.001071	.000531	.001316	.000631	.001560	.000731	.001804
.000432	.001075	.000532	.001319	.000632	.001564	.000732	.001808
.000433	.001079	.000533	.001323	.000633	.001567	.000733	.001811
.000434	.001083	.000534	.001327	.000634	.001571	.000734	.001815
.000435	.001087	.000535	.001331	.000635	.001575	.000735	.001819
.000436	.001091	.000536	.001335	.000636	.001579	.000736	.001823
.000437	.001094	.000537	.001338	.000637	.001583	.000737	.001827
.000440	.001098	.000540	.001342	.000640	.001586	.000740	.001831
.000441	.001102	.000541	.001346	.000641	.001590	.000741	.001834
.000442	.001106	.000542	.001350	.000642	.001594	.000742	.001838
.000443	.001110	.000543	.001354	.000643	.001598	.000743	.001842
.000444	.001113	.000544	.001358	.000644	.001602	.000744	.001846
.000445	.001117	.000545	.001361	.000645	.001605	.000745	.001850
.000446	.001121	.000546	.001365	.000646	.001609	.000746	.001853
.000447	.001125	.000547	.001369	.000647	.001613	.000747	.001857
.000450	.001129	.000550	.001373	.000650	.001617	.000750	.001861
.000451	.001132	.000551	.001377	.000651	.001621	.000751	.001865
.000452	.001136	.000552	.001380	.000652	.001625	.000752	.001869
.000453	.001140	.000553	.001384	.000653	.001628	.000753	.001873
.000454	.001144	.000554	.001388	.000654	.001632	.000754	.001876
.000455	.001148	.000555	.001392	.000655	.001636	.000755	.001880
.000456	.001152	.000556	.001396	.000656	.001640	.000756	.001884
.000457	.001155	.000557	.001399	.000657	.001644	.000757	.001888
.000460	.001159	.000560	.001403	.000660	.001647	.000760	.001892
.000461	.001163	.000561	.001407	.000661	.001651	.000761	.001895
.000462	.001167	.000562	.001411	.000662	.001655	.000762	.001899
.000463	.001171	.000563	.001415	.000663	.001659	.000763	.001903
.000464	.001174	.000564	.001419	.000664	.001663	.000764	.001907
.000465	.001178	.000565	.001422	.000665	.001667	.000765	.001911
.000466	.001182	.000566	.001426	.000666	.001670	.000766	.001914
.000467	.001186	.000567	.001430	.000667	.001674	.000767	.001918
.000470	.001190	.000570	.001434	.000670	.001678	.000770	.001922
.000471	.001194	.000571	.001438	.000671	.001682	.000771	.001926
.000472	.001197	.000572	.001441	.000672	.001686	.000772	.001930
.000473	.001201	.000573	.001445	.000673	.001689	.000773	.001934
.000474	.001205	.000574	.001449	.000674	.001693	.000774	.001937
.000475	.001209	.000575	.001453	.000675	.001697	.000775	.001941
.000476	.001213	.000576	.001457	.000676	.001701	.000776	.001945
.000477	.001216	.000577	.001461	.000677	.001705	.000777	.001949

APPENDIX H. TABLES OF POWERS OF TWO
AND BINARY-DECIMAL EQUIVALENTS

2^n	n	2^{-n}	TABLE OF POWERS OF 2									
1	0	1.0										
2	1	0.5										
4	2	0.25										
8	3	0.125										
16	4	0.062 5										
32	5	0.031 25										
64	6	0.015 625										
128	7	0.007 812 5										
256	8	0.003 906 25										
512	9	0.001 953 125										
1 024	10	0.000 976 562 5										
2 048	11	0.000 488 281 25										
4 096	12	0.000 244 140 625										
8 192	13	0.000 122 070 312 5										
16 384	14	0.000 061 035 156 25										
32 768	15	0.000 030 517 578 125										
65 536	16	0.000 015 258 789 062 5										
131 072	17	0.000 007 629 394 531 25										
262 144	18	0.000 003 814 697 265 625										
524 288	19	0.000 001 907 348 632 812 5										
1 048 576	20	0.000 000 953 674 316 406 25										
2 097 152	21	0.000 000 476 837 158 203 125										
4 194 304	22	0.000 000 238 418 579 101 562 5										
8 388 608	23	0.000 000 119 209 289 550 781 25										
16 777 216	24	0.000 000 059 604 644 775 390 625										
33 554 432	25	0.000 000 029 802 322 387 695 312 5										
67 108 864	26	0.000 000 014 901 161 193 847 656 25										
134 217 728	27	0.000 000 007 450 580 596 923 828 125										
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5										
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25										
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625										
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5										
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25										
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125										
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5										
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25										
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625										
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5										
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25										
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125										
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5										

BINARY AND DECIMAL EQUIVALENTS

Maximum Decimal Integral Value	Number of Decimal Digits	Number of Bits	Maximum Decimal Fractional Value
1	1	1	.5
3	2	2	.75
7	3	3	.875
15	4	4	.9375
31	5	5	.96875
63	6	6	.984375
127	7	7	.9921875
255	8	8	.99609375
511	9	9	.998046875
1023	10	10	.9990234375
2047	11	11	.99951171875
4095	12	12	.999755859375
8191	13	13	.9998779296875
16383	14	14	.99993896484375
32767	15	15	.999969482421875
65535	16	16	.9999847412109375
131071	17	17	.99999237060546875
262143	18	18	.999996185302734375
524287	19	19	.9999980926513671875
1048575	20	20	.99999904632568359375
2097151	21	21	.999999523162841796875
4194303	22	22	.9999997615814208984375
8388607	23	23	.99999988079071044921875
16777215	24	24	.999999940395355244609375
33554431	25	25	.9999999701976776123046875
67108863	26	26	.99999998509883880615234375
134217727	27	27	.999999992549419403076171875
268435455	28	28	.9999999962747097015380859375
536870911	29	29	.99999999813735485076904296875
1073741823	30	30	.999999999068677425384521484375
2147483647	31	31	.9999999995343387126922607421875
4294967295	32	32	.99999999976716935634613037109375
8589934591	33	33	.999999999883584678173065185546875
17179869183	34	34	.9999999999417923390865325927734375
34359738367	35	35	.99999999997089616954326629638671875
68719476735	36	36	.999999999985448034771633148193359375
137438953471	37	37	.9999999999927240423858165740966796875
274877906943	38	38	.99999999999636202119290828704833984375
549755813887	39	39	.999999999998181010596454143524169921875
1099511627775	40	40	.9999999999990905052982270717620849609375
2199023255551	41	41	.99999999999954525264911353588104248046875
4398046511103	42	42	.999999999999772626324556767940521240234375
8796093022207	43	43	.9999999999998863131622783839702606201171875
17592186044415	44	44	.99999999999994315658113919198513031005859375
35184372088831	45	45	.999999999999971578290569595992565155029296875
70368744177663	46	46	.9999999999999857891452847979962825775146484375
140737488355327	47	47	.99999999999999289457264239899814128875732421875
281474976710655	48	48	

This chart provides the information necessary to determine:

- The number of bits needed to represent a given decimal number. Use columns one and three or four and three.
- The number of bits needed to represent a given number of decimal digits (all nines). Use columns two and three.
- The maximum decimal value represented by a given number of bits, use columns one and three or three and four.

APPENDIX I. THE TWO'S COMPLEMENT NUMBER SYSTEM

Let us first consider a simple example of two's complement numbers, namely integers of three bits each, numbering the bits 0, 1, and 2, respectively, from left to right. Then the integer "xyz" represents the decimal quantity $-4x+2y+z$:

hence 011 represents +3
010 represents +2
001 represents +1
000 represents +0
111 represents -1
110 represents -2
101 represents -3
and 100 represents -4

Thus each decimal integer from -4 to 3 has a unique representation as a two's complement number. Bit 0 also serves as the sign-bit, since it is 0 for all positive numbers and 1 for all negative numbers. Note that "000" is a positive number.

We perform the addition "abc+xyz" as though "abc" and "xyz" were signless binary integers from 0 to 7, ignoring any carry out of bit 0 of the sum. If the true sum is not an integer from -4 to 3, then we have an overflow. We observe that the carry out of bit 0 = the carry out of bit 1 if, and only if, there is no overflow. In the case when $a \neq x$, we cannot have an overflow, since the sum ranges from -4 to 2. It follows that $a + x = 1$ and that the carries must be equal, since we have $0+1 = 1$ with carry 0 and $1+1 = 0$ with carry 1. In the case when $a = x$, we have no overflow if, and only if, bit 0 of the sum = x. We have this equality if, and only if, the carries are equal, since we have $0+0+0 = 0$ with carry 0 and $1+1+1 = 1$ with carry 1. We conclude that our overflow test is a valid one. The following examples are illustrations of two's complement addition:

CARRIES	00	11	00	01	11	10
abc	110=-2	110=-2	010=+2	010=+2	110=-2	110=-2
xyz	001=+1	011=+3	001=+1	011=+3	111=-1	101=-3
abc+xyz	111=-1	001=+1	011=+3	101=-3	101=-3	011=+3
REMARKS	NO OVF.	NO OVF.	NO OVF.	OVF.	NO OVF.	OVF.

We say that "uvw" is the one's complement of "xyz" (and vice versa) if $uvw+xyz = 111$. Hence $u+x = v+y = w+z = 1$. We say that the quantity "uvw+001" is the two's complement of "xyz", observing that its decimal value is:

$$\begin{aligned} -4u+2v+w+1 &= -4(1-x)+2(1-y)+(1-z)+1 \\ &= -(-4x+2y+z), \end{aligned}$$

or minus the value of "xyz". For this reason we call "xyz" a two's complement number. We perform the subtraction "abc-xyz" by the triple addition "abc+uvw+001" (in effect, by adding "abc" and "uvw" with a forced carry of 1 into the low order bit 2). We use the same overflow test as for addition. Note that 000-000 = 000 (no overflow) and that 000-100 = 100 (overflow). Hence "000" is its own two's complement, and "100" does not have a proper two's complement. We note the conspicuous absence of a -0 from the two's complement system above.

We may generalize the above discussion to include two's complement integers of N bits each. The integer " $x_0x_1x_2\ldots x_{N-2}x_{N-1}$ " represents the decimal quantity below:

$$-2^{N-1}x_0 + 2^{N-2}x_1 + 2^{N-3}x_2 + \ldots + 2x_{N-2} + x_{N-1}$$

The same rules as above hold for addition, overflow, complementation, and subtraction. In the GE-600 hardware, we may have several choices for N:

- N=8 for exponent fields,
- N=18 for address fields,
- N=36 for single-precision integers,
- and N=72 for double-precision integers.

The use of two's complement numbers offers many advantages:

1. It eliminates housekeeping before and after addition and subtraction in the computer hardware.
2. It permits addition and subtraction modulo 2^N , since we may always consider a number to be signless.
3. It permits addition of a quantity to a field of a word, without any need to worry about the sign-bit. (In the sign-magnitude system, one would add the quantity if the sign were positive, and subtract the quantity if the sign were negative.)
4. It makes zero a unique positive number.
5. It is compatible with index register arithmetic.

Of course, the GE-600 programmer must always be aware of the fact that the computer is a two's complement machine, especially when converting programs that were originally written for a machine with sign-magnitude or one's complement arithmetic. For example, the sign magnitude convention of "changing sign" corresponds to the two's complement convention of "negation" (or "complementation"). In FORTRAN systems, the quantity -0 often indicates a blank card field. There is no such quantity in the GE-600 system, whether in fixed or floating point.

A two's complement floating point number in the Floating Point Register consists of two parts:

1. An integral exponent field of eight bits.
2. A fractional mantissa field of seventy-two bits. The mantissa " $x_0x_1x_2\ldots x_{71}$ " represents the decimal quantity below:

$$-x_0 + 2^{-1}x_1 + 2^{-2}x_2 + \ldots + 2^{-71}x_{71}$$

We say that a floating point number is normalized if either:

1. The exponent field is 10000000 and the mantissa field is zero, or
2. The first two mantissa bits are different: $x_0 \neq x_1$

The value of a floating point number is mantissa $\cdot 2^{\text{exponent}}$. Hence the normal form of +1 is exponent 00000001 and mantissa 0100...0, and the normal form of -1 is exponent 00000000 and mantissa 1000...0. If "f" is a floating point number that is not a power of two, however, then both +f and -f have the same exponent fields in normal form, and their mantissa fields are two's complements of each other. For f=10, the normal form of +f is exponent 00000100 and mantissa 010100...0. The normal form of -f is exponent 00000100 and mantissa 101100...0. Note that the first bit of the mantissa is the sign-bit of the number.

$$\text{Since } -x_0 + 2^{-1}x_0 + 2^{-2}x_1 + \dots + 2^{-71}x_{70} = \frac{1}{2}(-x_0 + 2^{-1}x_1 + \dots + 2^{-71}x_{71}),$$

ignoring the remainder, the GE-600 hardware retains the value of bit 0 during each right shift cycle prior to a floating point addition or subtraction. For the same reason, there is a numeric right shift as well as a logical right shift for A, Q, and AQ.

The representation of mixed numbers illustrates a feature of the two's complement number system. Consider the case of f=1.25. Then the normal form of +f is exponent 00000001 and mantissa 010100...0. The normal form of -f is exponent 00000001 and mantissa 101100...0. The integral part of +f is +1, and the fractional part of +f is +.25. The integral part of -f is -2, and the fractional part of -f is +.75. Hence the integral parts are one's complements of each other, and the fractional parts are two's complements of each other. In general, this condition holds whenever we divide a two's complement number into a pair of disjoint fields, where the right field is not zero. The reason for the condition is that the sign-bit of a two's complement number is the only bit with a negative value. The condition is desirable in some mathematical applications where we wish to compute the greatest integer less than or equal to a given number. However, the condition raises a compatibility problem when converting programs originally coded on sign-magnitude or one's complement machines. The solution to the problem is the addition of +1 to the integral part of nonwhole negative numbers. The problem arises noticeably in the implementation of FORTRAN built-in functions.

CE 625/683

1004D