FLOATING POINT
SYSTEMS, INC.

FPS-100 Assembler (ASM100) Reference Manual

860-7428-001

by FPS Technical Publications Staff

FPS-100 Assembler (ASM100) Reference Manual

860-7428-001

Publication No. 860-7428-001
September, 1979

NOTICE

This edition applies to Release A of FPS-100 software and all subsequent releases until superseded by a new edition.

The material in this manual is for informational purposes only and is subject to change without notice.

Floating Point Systems, Inc. assumes no responsibility for any errors which may appear in this publication.

Printed in USA

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER 1

OVERVIEW

1.1 INTRODUCTION

The Floating Point Systems, Inc., FPS-100 is a peripheral device that operates independently from but under the direction of a host processor. It contains its own internal memories and 38-bit floating-point arithmetic units which are interconnected with multiple data paths, allowing parallel internal data transfers. Its arithmetic units, the floating adder and floating multiplier, are designed as pipelines (operations are performed in independent stages permitting new operations to begin before old operations are complete). This parallel processing capability and pipeline arithmetic permit the FPS-100 to perform high speed array processing.

The FPS-100 Assembly Language (ASM100) allows the programmer to use the FPS-100 instruction set and control assembly with a group of pseudo-operations. ASM100 code is assembled on the host system for execution on the FPS-100.

1.2 PURPOSE

This manual provides the information necessary for a programmer to create a complete assembly language program and assemble it using the ASM100 assembler. It is not a training manual, however. It does not attempt to teach assembly language programming to the beginner. It does assume that the user is familiar with FPS-100 hardware and the FPS-100 instruction set.

1.3 SCOPE

This manual describes the syntax of all ASM100 statements. Complete descriptions are provided for all pseudo-ops. A short description of the FPS-100 instruction set is provided, but this manual is not the primary reference source for the instruction set. For a complete description of the instruction set, refer to the Programmer's Reference Manual, Parts 1 and 2. Finally, a description of how to use the assembler is provided, along with a list of error messages.

1.4 CONVENTIONS

In examples of dialogue at a terminal, user input is underlined to distinguish it from FPS-100 or program output. Also, all user input is assumed to be terminated with a carriage return.

In statement descriptions, uppercase characters must be entered exactly as shown; lowercase characters indicate that a value or name must be substituted for the characters. Optional parameters are surrounded by brackets ({ }).

1.5 RELATED MANUALS

The following manuals may also be of interest to the user.

Table 1-1 Related Manuals

MANUAL	PUBLICATION NO.
FPS-100 Programmer's Reference Manual Parts One and Two	FPS 860-7427-000
LOD100 Reference Manual	FPS 860-7423-000
SIM100/DBG100 Reference Manual	FPS 860-7424-000
FTN100 Reference Manual	FPS 860-7422-000
FPS-100 Supervisor Reference Manual	FPS 860-7445-000

CHAPTER 2

SYNTAX

2.1 CHARACTER SET

ASM100 recognizes the characters in Table 2-1. Characters which have special meaning are listed in Table 2-2.

Table 2-1 Character Set

ALPHABETIC	NUMBERIC	SPECIAL
A through Z	0 through 9	Blank
		= Equals
		+ Plus
		- Minus
		* Asterisk
		/ Slash
		(Left parenthesis
		) Right parenthesis
		, Comma
		. Decimal point
		$ Dollar
		Tab
		< Less than
		; Semicolon
		: Colon
		" Quote
		# Number
		& Ampersand
		! Exclamation point
		@ At sign

Table 2-2 Special Characters

CHARACTER	FUNCTION
+	Integer addition operator; unary addition operator
-	Integer subtraction operator; unary subtraction operator
*	Integer multiplication operator
/	Integer division operator
.	Decimal point; current location
$	First character of pseudo-op names
space	Symbol terminator
tab	Symbol terminator
=	$EQU pseudo-op; DB = op-code; arithmetic identify
(	Precedes a data pad index expression
)	Terminates a data pad index expression
<	Used with DPX, DPY, and MI op-codes; arithmetic less than
;	Op-code terminator
,	Operand separator
:	Label terminator
"	Comment start indicator (carriage return terminates)
#	S-pad no-load indicator
&	S-pad bit-reverse indicator
!	First character of predefined symbols
%	Logical OR operator
'	Logical complement
>	Arithmetic greater than
?	No system function
~	No system function
@	Absolute addressing

2.2 FILE NAMES

File names may contain 30 characters including special characters and numbers. On systems where programmed file assignment is not allowed or is very difficult, the user must enter the number of the logical unit of a file assigned prior to calling ASM100.

A special symbol (which is different for each host system) exists for referencing the user terminal (for example: TT: for the PDP11).

Examples:

```
RUNNER
RUNNER.OBJ
P38
CHANNEL
```

2.3 SYMBOL NAMES

Symbol names may be of any length; however, only the first six characters of a name are significant. The first character of a name must be alphabetic or the exclamation point (!). The subsequent characters can be either alphabetic, numeric, or the exclamation point.

Examples:

```
LOOP
A6
STARTHERE
```

A symbol can be created and given a value by the following:

- defining it with the $EQU pseudo-op
- declaring it with the $INTEGER, $REAL, or $COMMON pseudo-op and giving it a value with the $DATA pseudo-op
- using it as a label
- declaring it an external with the $EXT pseudo-op

2.4 TABLE MEMORY SYMBOLS

A symbol with a value preset to the address of each of the constants in table memory ROM is predefined in ASM100. These symbols all start with the exclamation point character (!) to avoid conflict with any user-defined symbol. ASM100 declares these symbols externals when used in expressions (for example, DB=!ZERO). Therefore, they must be loaded from library SYMLIB at load time. When these symbols are used in any other way, such as in labels, ASM100 treats them as variables, and they are not predefined.

A complete list of these symbols can be found in section 3.3.2.14. For example, the following fetches pi from table memory and adds it to a number in DPX(2):

```
LDTMA; DB=!PI           "Fetch PI from TM
NOP                     "Wait
FADD TM, DPX(2)         "Add PI to DPX(2)
```

2.5 INTEGERS

Integers can be written in four radices: octal, binary, decimal, or hexadecimal. In each radix, an integer can be either signed or unsigned. The radix of a number is established by a radix identifying character which is written immediately after the number. Octal integers are denoted by a K, decimal by a period (.), hexadecimal by an X, and binary by a T. The first digit of a hexadecimal integer must be a decimal digit. The default radix, if a radix identifier is not used, is octal unless otherwise specified by a $RADIX pseudo-op.

Integers can be single precision or double precision. Single precision integers are stored as 16-bit 2's complement numbers. Integers larger than 16 bits are truncated to 16 bits. Negative integers larger than 16 bits are truncated before they are negated.

Double precision integers are declared with the $TRIPLE pseudo-op. They are stored as 38-bit 2's complement numbers.

Examples:

octal integers:	177777 -40727K -10
decimal integers:	32767. -1000. +10.
hexadecimal integers:	0ABCDX 123FX 0CX
binary integers:	101101T -1101T

2.6 EXPRESSIONS

Expressions are symbolic representations of numbers. They are made of operands and operators. If an expression contains a reference to an external symbol, the expression must be of the form external-symbol ± expr, where expr is an expression without any external references.

2.6.1 OPERANDS

Operands are symbol names, numbers, or the location counter, which is denoted by a period (.).

Examples:

```
TBLADR
598X
.
33K
```

2.6.2 OPERATORS

Operators are of two types, unary and binary.

Unary Operators	'	logical complement
	+	positive remainder (+3K, +10.)
	-	negative of a number (-15X, -777)

Standard arithmetic operators are the following:

Binary Operators	+	addition
	-	subtraction
	*	multiplication
	/	division

Standard arithmetic relations, which return a value of one if the relation is true and zero if the relation is false, are as follows (for example, B $EQU 6<10 sets B to 1):

<	less than
=	equals
>	greater than

Some expressions are:

```
TBLADR+3F
. + 9.
LOOP + 6 * A
(34 - 10X) * 2
```

Expressions are evaluated from left to right in 16-bit 2's complement arithmetic according to FORTRAN precedence standards; parentheses may be used liberally.

NOTE

Only the low order 16 bits are used if an expression results in a decimal value larger than 65535.

2.7 ADDRESSING MODES

Two modes of addressing can be used on the FPS-100, relative addressing and absolute addressing. Relative addressing is done unless the absolute addressing indicator (@) is specified.

2.7.1 RELATIVE ADDRESSING

In this mode, all addresses specified are regarded as relative to the program source address register (PSA). PSA points to the instruction currently executing. Therefore, a specified address is really only a displacement (either positive or negative) which is added to PSA in order to arrive at an absolute program source address at execution time. With relative addressing, the program is position-independent in program source memory.

2.7.2 ABSOLUTE ADDRESSING

Absolute addressing is performed when the at sign (@) prefaces an address and the addresses are used in conjunction with the absolute addressing versions of certain instructions (refer to section 3.3.2.12). In this mode, all addresses represent absolute program source addresses as they are generated by the assembler or the loader. No execution time manipulation is required. With absolute addressing, the program is position-dependent and executes properly only if it is loaded at the correct program source address.

2.7.3 RELOCATION OF SYMBOLS

The assembler produces relocation information for certain variables so that the LOD100 loader can generate the correct absolute addresses for these symbols. ASM100 generates relocation information for all external variables and all symbols and constants preceded by the absolute addressing indicator @.

If the special absolute address indicator is not used, a reference to an external label is interpreted as the relative displacement from the instruction referencing the label to the label itself. A reference to an internal label is interpreted as the displacement of the label from the beginning of the subroutine as determined by the assembler (this is the number associated with the label in the symbol table displayed at the bottom of the assembly listing). Constants are unaltered regardless of where the program is loaded in program source memory.

The relocation information produced by ASM100 can only be used by the LOD100 loader. LNK100 cannot take advantage of this information.

CHAPTER 3

SOURCE PROGRAM STATEMENTS

3.1 INTRODUCTION

ASM100 source statements can be divided into three categories as follows:

- comment statements
- instruction statements
- pseudo-op statements

Comment statements allow program documentation. Instruction statements make up the actual symbolic machine code. Pseudo-ops provide directions to ASM100 during the assembly process.

ASM100 statements can be entered in free format; spaces and tabs may be used as desired to improve legibility.

3.2 COMMENT STATEMENTS

Everything on a line following a quote mark (") is treated as a comment by ASM100. A line containing only comments, or a completely blank line, is a comment statement and is ignored during the assembly process. A carriage return terminates a comment.

3.3 INSTRUCTION STATEMENTS

An ASM100 assembly language instruction statement has the following format:

```
label:  op-code fields        "Comments
```

The label and comments are optional. The assembler processes the op-code fields and generates one 64-bit instruction word for each instruction statement.

3.3.1 LABEL FIELD

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user symbol table. A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in an instruction statement and must be terminated by a colon. For example, assume that the following instruction statement is entered:

```
LOOP:  FADD DPX, DPY          "LOOP HERE
```

If the current location is 76, value of 76 is assigned to symbol LOOP.

3.3.2 OP-CODE FIELD (OPERATION CODE FIELD)

The op-code field follows the label field in an instruction statement and contains one or more FPS-100 op-code mnemonics. Individual op-codes in an instruction are separated by a semicolon. For example, the following two groups of opcodes are equivalent. The absence of a semicolon following the last op-code field on a given line terminates the instruction with that line.

```
LOOP:  FADD DPX, DPY; FMUL TM, MD; BFGT DONE

or

LOOP:  FADD DPX, DPY;
       FMUL TM, MD;
       BFGT DONE
```

Each is one instruction statement which assembles into one 64-bit instruction word. Thus, one instruction statement may be continued over as many lines as desired to achieve a readable program document. The absence of a semicolon after the last op-code signals the assembler that the instruction is ended.

Op-codes may be written in any order within an instruction. The assembler flags any conflicting op-codes with an error message.

Some op-codes require operands as arguments. The operand is separated from the op-code by a space or tab and from another operand by a comma. Some example op-codes are:

```
no operands:      HALT; RETURN
one operand:      FABS MD; BFGT LOOP
two operands:     FADD DPX, DPY; FMUL TM, MD
```

If an operand is missing or improper, the assembler generates an appropriate error message.

The various FPS-100 op-codes may be divided into 13 groups. One op-code from each group may be used in any given instruction statement unless otherwise stated.

Under the headings Function and Meaning, upper case characters are used to indicate the origin of the mnemonic code names.

The list of abbreviations contained in Table 3-1 are used to facilitate the op-code descriptions. They are explained later when the op-code group first appears.

Table 3-1 Op-code Abbreviations

ABBREVIATION	MEANING	PARAGRAPH IN WHICH DESCRIBED
sh	S-pad shift	3.3.2.1
#	S-pad no-load	3.3.2.1
sps	S-pad source register	3.3.2.1
spd	S-pad destination register	3.3.2.1
&	Bit reverse	3.3.2.1
disp	Branch displacement	3.3.2.5
a1	Floating adder argument #1	3.3.2.6
a2	Floating adder argument #2	3.3.2.6
idx	Data pad index	3.3.2.6
m1	Floating multiplier argument #1	3.3.2.7
m2	Floating multiplier argument #2	3.3.2.7
dbe	Data pad bus enable	3.3.2.8
adr	Address, value, or expression	3.3.2.8

3.3.2.1 S-pad Op-code Group

Purpose: s-pad integer arithmetic

Double Operand Op-codes	Function
ADD{sh}{#}{&}sps,spd	ADD sps to spd
SUB{sh}{#}{&}sps,spd	SUBtract sps from spd
MOV{sh}{#}{&}sps,spd	MOVe sps tp spd
AND{sh}{#}{&}sps,spd	AND sps tp spd
OR{sh}{#}{&}sps,spd	OR sps to spd
EQV{sh}{#}{&}sps,spd	EQuiValence sps to spd

Single Operand Op-codes	Function
CLR{sh}{#} spd	ClearR spd
INC{sh}{#} spd	INCrement spd
DEC{sh}{#} spd	DECrement spd
COM{sh}{#} spd	COMplement spd

The result of the above op-codes is SPFN (s-pad function).

Miscellaneous Op-codes		Function
LDSPNL	spd	LoaD Spd from PaNeL bus
LDSPE	spd	LoaD SPd from data pad bus Exponent
LDSPI	spd	LoaD SPd from data pad bus Integer (low 16-bit)
LDSPT	spd	LoaD SPd from data pad bus Table look-up bits
WRTEXP		enable WRiTe of EXPonent only into DPX, DPY, or MI
WRTHMN		enable WRiTe of High MaNtissa only into DPX, DPY, or MI
WRTLMN		enable WRiTe of Low MaNtissa only into DPX, DPY, or MI

Abbreviations:

Name	Meaning
sh	s-pad shift:

Choices	Meaning
(omitted)	no shift
L	shift SPFN left once
R	shift SPFN right once
RR	shift SPFN right twice

#	S-pad no-load: if present, do not load SPFN into spd (s-pad destination register). If specified, a branch group op-code may not be used in the same instruction statement.
sps	S-pad source register: a name, number, or expression specifying a register number between 0 and 17_8.
spd	S-pad destination register: a name, number, or expression specifying a register number between 0 and 17_8. SPFN is loaded into the s-pad destination register unless s-pad no-load (#) is specified.
&	Bit reverse: if present, bit reverse the contents of sps before using. The bit reverse is done as specified by bits 13-15 of the internal status register.

Examples:

```
MOV 3,6
SUBL 1,15
ADDL# &PTR, BASE
DEC CTR
CLR 9.
LDSPI 6
```

3.3.2.2 Memory Address Op-code Group

Purpose: initiate main data memory cycles

Op-codes	Function
INCMA	INCrement Memory Address
DECMA	DECrement Memory Address
SETMA	SET Memory Address from SPFN

3.3.2.3 Table Memory Address Op-code Group

Purpose: initiate table memory fetches

Op-codes	Function
INCTMA	INCrement Table Memory Address
DECTMA	DECrement Table Memory Address
SETTMA	SET Table Memory Address from SPFN

3.3.2.4 Data Pad Address Op-code Group

Purpose: change the DPA (data pad address) register

Op-codes	Function
INCDPA	INCrement Data Pad Address
DECDPA	DECrement Data Pad Address
SETDPA	SET Data Pad Address from SPFN

3.3.2.5 Branch Op-code Group

Purpose: conditional branches

Op-code		Function
BR	disp	BRanch unconditionally
BINTRQ	disp	Branch on INTerrupt ReQuest flag non-zero
BION	disp	Branch on I/O data ready flag Non-zero
BIOZ	disp	Branch on I/O data ready flag Zero
BFPE	disp	Branch on Floating-Point Error
BFEQ	disp	Branch on Floating adder EQual to zero
BFNE	disp	Branch on Floating adder Not Equal to zero
BFGE	disp	Branch on Floating adder Greater or Equal to zero
BFGT	disp	Branch on Floating adder Greater Than zero
BEQ	disp	Branch on s-pad function EQual to zero
BNE	disp	Branch on s-pad function Not Equal to zero
BGE	disp	Branch on s-pad function Greater or Equal to zero
BGT	disp	Branch on s-pad function Greater Than zero
RETURN		RETURN from subroutine

Abbreviation:

Name	Meaning
disp	Branch displacement: the branch target address, an address between 16 locations behind and 15 locations ahead of the current location.

Examples:

```
BR LOOP
BGT .+3
BFNE A-4
```

3.3.2.6 Floating Adder Op-code Group

Purpose: floating-point adds

Double Operand Op-codes

Op-codes	Function
FADD a1,a2	Floating ADD (a1+a2)
FSUB a1,a2	Floating SUBtract (a1-a2)
FSUBR a1,a2	Floating SUBtract Reverse (a2-a1)
FAND a1,a2	Floating AND (a1 and a2)
FOR a1,a2	Floating OR (a1 or a2)
FEQV a1,a2	Floating EQuiValence (a1 eqv a2)

Single Operand Op-codes

Op-codes	Function
FIX a2	FIX a2 to an integer
FIXT a2	FIX a2 to an integer (Truncated)
FSCALE a2	Floating SCALE of a2
FSCLT a2	Floating SCaLe of a2 (Truncated)
FSM2C a2	Format conversion, Signed Magnitude to 2's Complement
F2CSM a2	Format conversion, 2's Complement to Signed Magnitude
FABS a2	Floating ABSolute value

Other Op-codes

Op-codes	Function
FPA1	Push A1 through the Floating adder without change
FPA2	Push A2 through the Floating adder without change

Adder Operands:

Operand	Meaning
a1	floating adder argument no. 1:

Choices	Meaning
NC	No Change (use previous a1)
FM	Floating Multiplier output
DPX {(idx)}	Data Pad X
DPY {(idx)}	Data Pad Y
TM	Table Memory data
ZERO	floating-point ZERO

Operand	Meaning
a2	adder argument no. 2:

Choices	Meaning
NC	No Change (use previous a2)
FA	Floating Adder output
DPX {(idx)}	Data Pad X
DPY {(idx)}	Data Pad Y
TM	Table Memory data
ZERO	floating ZERO
MDPX {(idx)}	use Mantissa from Data Pad X and exponent from SPFN
EDPX {(idx)}	use Exponent Data Pad X and mantissa from SPFN

Abbreviation:

Name	Meaning
idx	Data pad index: a name, expression, or number which lies in a range of -4 to +3.

Examples:

```
FADD TM,MD
FSUB DPX(3), DPY(-4)
FEQV DPX, DPY(C)
FAND ZERO, MDPX(2)
FSUBR NC,FA
FADD
```

NOTE

Up to four unique data pad indices may be specified in one instruction statement. In particular, only one indexing each may be used for reading from data pad X and Y, regardless of how many op-codes use the data read from data pad.

3.3.2.7 Floating-Point Multiply Op-code Group

Purpose: floating-point multiplies

Op-code	Function
FMUL m1,m2	Floating MULtiply m1 times m2

Multiplier Operands:

Operand	Meaning
m1	multiplier operand no. 1

Choices	Meaning
FM	Floating Multiplier output
DPX{(idx)}	Data Pad X
DPY{(idx)}	Data Pad Y
TM	Table Memory

m2	multiplier operand no. 2

Choices	Meaning
FA	Floating Adder output
DPX{(idx)}	Data Pad X
DPY{(idx)}	Data Pad Y
MD	Memory Data

Examples:

```
FMUL TM, MD
FMUL DPX(AR),DPY(BI)
FMUL
```

3.3.2.8 Data Pad X Op-code Group

Purpose: storing into data pad X

Op-code	Function
DPX{(idx)}<opt	Store opt into data pad X. One of the following must be used for opt:

Opt	Meaning
FA	Floating Adder output
FM	Floating Multiplier output
DB	Data pad Bus
dbe	data pad bus enable This has the same effect as an explicit data pad bus op-code. One choice of data pad bus enable may be made per instruction statement.

Choices	Meaning
ZERO	floating ZERO
{@}adr	An address or numeric value. Any 16-bit integer expression is legal. A floating multiplier, memory input, memory address, or data pad address op-code cannot be used in an instruction statement where an adr is used. The optional @ indicates an absolute address.
DPX{(idx)}	Data Pad X
DPY{(idx)}	Data Pad Y
MD	Memory Data
SPFN	S-Pad FuNction
TM	Table Memory data

Examples:

```
DPX(3)<FM
DPX(-2)<SPFN
DPX MD
DPX(1)<DPY(-2)
DPX(-2)< -123
```

3.3.2.9 Data Pad Y Op-code Group

Purpose: storing into data pad Y

Op-code	Function
DPY{(idx)}<opt	Store opt into data pad Y. The possibilities for opt are the same as those described in section 3.3.2.8.

Examples:

```
DPY(-2)<FA
DPY<MD
DPY(2)<TM
DPY(1)<39
```

3.3.2.10 Memory Input Op-code Group

Purpose: writing into main data memory

Op-codes	Function
MI<FA	move Floating Adder output to the Memory Input register
MI<FM	move Floating Multiplier output to the Memory Input register
MI<DB	move Data pad Bus to the Memory Input register
MI<dbe	move dbe to the Memory Input register

To affect a memory write, an op-code from the memory address group or an LDMA op-code must be included in the instruction statement to supply the memory address.

Examples:

```
MI<FA; INCMA
MI<DPX(3); DECMA
MI<MD; SETMA; ADD 3,6
```

3.3.2.11 Data Pad Bus Op-code Group

Purpose: explicitly enable data onto the data pad bus

Op-codes	Function
DB=ZERO	enable ZERO onto the Data pad Bus
DB={@}addr	enable adr onto the Data pad Bus (the optional @ indicates an absolute address)
DB=DPX{(idx)}	enable Data Pad X onto the Data pad Bus
DB=DPY{(idx)}	enable Data Pad Y onto the Data pad Bus
DB=MD	enable Memory Data onto the Data pad Bus
DB=SPFN	enable S-Pad FuNction onto the Data pad Bus
DB=TM	enable Table Memory data onto the Data pad Bus

As mentioned in section 3.3.2.8, only one data source may be enabled onto the data pad bus per instruction statement.

Examples:

```
DB = 37
DB = DPX(-2)
DB = MD
DB = SPFN
```

3.3.2.12 Special Operation Op-code Group

If an op-code from this group is chosen, an s-pad group op-code cannot be used in the same instruction statement.

Abbreviations:

Name	Meaning
A	In this section, the optional A at the end of an op-code signifies that the associated address is an absolute address. If not specified, the address is relative. When these op-codes are used, the absolute address indicator @ should precede address.
@	In this section the optional @ preceding the address indicates to the assembler and loader that the address is an absolute address. The assembler generates relocation information, so the loader can determine the correct absolute address.

Special Tests

Purpose: additional conditional branches

Op-codes	Function
BFLT disp	Branch on Floating adder Less Than zero
BLT disp	Branch on s-pad function Less Than zero
BNC disp	Branch on Non-zero Carry bit
BZC disp	Branch on Zero Carry bit
BDBN disp	Branch if Data pad Bus Negative
BDBZ disp	Branch if Data pad Bus Zero
BIFN disp	Branch if Inverse FFT flag Non zero
BIFZ disp	Branch if Inverse FFT flag Zero
BFL0 disp	Branch if FLag 0 is 1
BFL1 disp	Branch if FLag 1 is 1
BFL2 disp	Branch if FLag 2 is 1
BFL3 disp	Branch if FLag 3 is 1

If one of the preceding tests is used along with a test from the branch group, the conditions are ORed. In this case, only one of the branch op-codes need have the target address as an operand.

Examples:

```
BNC ODD
BFEQ LOOP; BFLT LOOP "LESS THAN OR EQUAL TO
```

SETPSA

Purpose: jumps and subroutine jumps

Op-codes	Function
JMP{A} {@}adr	JuMP to location adr
JMPT	JuMP to location whose address is in TMA
JMPP	JuMP to location whose address is on the Panel bus
JSR{A} {@}adr	Jump to SubRoutine at location adr
JSRT	Jump to SubRoutine at address in TMA
JSRP	Jump to SubRoutine at address on Panel bus

Examples:

```
JMP LOOP + 3
JSR FFT
JMPS 300
```

SETEXIT

Purpose: alter a subroutine return

Op-codes	Function
SETEX{A} {@}adr	SET subroutine EXit to adr
SETEXT	SET subroutine EXit to contents of TMA
SETEXP	SET subroutine EXit to contents of Panel bus

Example:

```
SETEX BAD
```

Program Source

Purpose: read/write program source memory

Op-codes	Function
RPSL{A} {@}adr	Read Program Source Left half of location adr
RPSF{A} {@}adr	Read Program Source Floating-point number from location adr
RPSLT	Read Program Source Left half at address in TMA
RPSFT	Read Program Source Floating-point number at address in TMA
RPSLP	Read Program Source Left half at address on Panel bus
RPSFP	Read Program Source Floating-point number at address on Panel bus

The preceding op-codes read onto the data pad bus.

Op-codes	Function
LPSL{A} {@}adr	Load Program Source Left half of location adr
LPSR{A} {@}adr	Load Program Source Right half of location adr
LPSLT	Load Program Source Left half pointed at by TMA
LPSRT	Load Program Source Right half pointed at by TMA
LPSLP	Load Program Source Left half pointed at by Panel bus
LPSRP	Load Program Source Right half pointed at by Panel bus

The preceding op-codes load from the data pad bus.

Example:

```
RPSF PI
```

PS Odd and Even

Purpose: reading the host panel switches into program source memory, writing program source to the panel lights register

Op-codes	Function
RPS0{A} {@}adr	Read Program Source quarter 0 from location adr
RPS1{A} {@}adr	Read Program Source quarter 1 from location adr
RPS2{A} {@}adr	Read Program Source quarter 2 from location adr
RPS3{A} {@}adr	Read Program Source quarter 3 from location adr
RPS0T	Read Program Source quarter 0 from address in TMA
RPS1T	Read Program Source quarter 1 from address in TMA
RPS2T	Read Program Source quarter 2 from address in TMA
RPS3T	Read Program Source quarter 3 from address in TMA
WPS0{A} {@}adr	Write Program Source quarter 0 into location adr
WPS1{A} {@}adr	Write Program Source quarter 1 into location adr
WPS2{A} {@}adr	Write Program Source quarter 2 into location adr
WPS3{A} {@}adr	Write Program Source quarter 3 into location adr
WPS0T	Write Program Source quarter 0 into address in TMA
WPS1T	Write Program Source quarter 1 into address in TMA
WPS2T	Write Program Source quarter 2 into address in TMA
WPS3T	Write Program Source quarter 3 into address in TMA

Host Panel

Purpose: reading the host panel switches, writing to the host panel lights register

Op-codes	Function
PNLLIT	PaNeL bus to LIghTs
DBELIT	Data pad Bus Exponent to LIghTs
DBHLIT	Data pad Bus High mantissa to LIghTs
DBLLIT	Data pad Bus Low mantissa to LIghTs
SWDB	SWitches to Data pad Bus
SWDBE	SWitches to Data pad Bus Exponent
SWDBH	SWitches to Data pad Bus High mantissa
SWDBL	SWitches to Data pad Bus Low mantissa

Special Interrupts

Purpose: provide a software interrupt capability for the FPS-100

Op-codes	Function
ION	enable (or turn ON) universal Interrupt
IOFF	inhibit (or turn OFF) universal Interrupt
SETMOD	SET MODe to supervisor
CLRMOD	set mode to user (or CLeaR MODe)
SELMA	SELect MA
SELSMA	SELect Supervisor MA
ENTINT	ENTer INTerrupt
CM2PM	Current Mode to Previous Mode
TRAP	cause TRAP interrupt
RDPI	Read Data Pad X and Y Input buffer
WDPI	Write Data Pad X and Y Input buffer
DBLSW	Data pad Bus Low mantissa to SWitch register
PN2DBL	PaNel bus to Data pad Bus Low mantissa
EXINT	EXit INTerrupt

Miscellaneous

Op-codes	Function
SPNDAV	SPiN until MD AVailable

3.3.2.13 I/O Op-code Group

If an op-code is used from this group, a floating adder op-code cannot be used in the same instruction statement.

Load REG, Read REG

Purpose: reading/writing various internal registers

Op-codes	Function
LDSPD	LoaD S-Pad Destination address register
LDMA	LoaD Memory Address register
LDTMA	LoaD Table Memory Address register
LDDPA	LoaD Data Pad Address register
LDSP	LoaD S-Pad register pointed at by spd
LDAPS	LoaD FPS-100 Status register
LDDA	LoaD I/O Device Address

The preceding op-codes load from the data pad bus.

Op-codes	Function
RPSA	Read Program Source Address
RSPD	Read S-Pad Destination register
RMA	Read Memory Address register
RTMA	Read Table Memory Address register
RDPA	Read Data Pad Address register
RSPFN	Read S-Pad FuNction
RAPS	Read FPS-100 Status
RDA	Read I/O Device Address

The previous op-codes are read onto the panel bus.

IOMEM

Purpose: read/write memory fields

Op-codes	Function
REXIT	Read subroutine EXIT address
STATMA	STATic memory read or write at current MA or SMA
LDOMA	LoaD inactive (Other) Memory Address register
ROMA	Read inactive (Other) Memory Address register

INOUT

Purpose: program control input/output of data

Op-codes	Function
OUT	OUTput data
SPNOUT	SPiN until device ready, then OUTput data
OUTDA	OUTput data, then set DA to spfn
SPOTDA	SPin until device ready, OuTput data, then set DA to spfn

The preceding op-codes write to the I/O device specified by the device address register (DA). These op-codes write whatever data is enabled onto the data pad bus.

Op-codes	Function
IN	INput data
SPININ	SPIN until device ready, then INput data
INDA	INput data, then set DA to spfn
SPINDA	SPin until device ready, then INput data, then set DA to spfn

The preceding instructions put data onto the input bus from the I/O device specified by the device address register (DA). To be used, the data must be put onto the data pad bus and from there moved to a register or memory.

Example:

```
        IN; DPX(2)<INBS    "READ I/O DATA INTO DPX
```

SENSE

Purpose: sensing an I/O device condition

Op-codes	Function
SNSA	SeNSe condition A
SPINA	SPIN on condition A
SNSADA	SeNSe condition A, then set DA to spfn
SPNADA	SPiN on condition A, then set DA to spfn
SNSB	SeNSe condition B
SPINB	SPIN on condition B
SNSBDA	SeNSe condition B, then set DA to spfn
SPNBDA	SPiN on condition B, then set DA to spfn

FLAG

Purpose: set/reset of program flags

Op-codes	Function
SFL0	Set FLag 0
SFL1	Set FLag 1
SFL2	Set FLag 2
SFL3	Set FLag 3
CFL0	Clear FLag 0
CFL1	Clear FLag 1
CFL2	Clear FLag 2
CFL3	Clear FLag 3

CONTROL

Purpose: miscellaneous control functions

Op-code	Functions
HALT	HALT processor
IORST	I/O ReSeT
INTEN	INTerrupt ENable
INTA	INTerrupt Acknowledge
REFR	memory REFResh synch
WRTEX	enable WRiTe of EXponent only into DPX, DPY, or MI
WRTMN	enable WRiTe of MaNtissa only into DPX, DPY, or MI
SPMDAV	SPin until a Main Data memory cycle AVailable
IOINTA	I/O INTerrupt Acknowledge

Miscellaneous

Purpose: miscellaneous control functions

Op-codes	Functions
REXIT	Read subroutine EXIT into panel bus

3.3.2.14 Table Memory

Table 3-2 lists the constants available in table memory. This section also includes the table memory functions. The constants and functions are externals, and their use must conform to the same rules as other externals.

Table 3-2 Table Memory Constants

SYMBOL	CONSTANT REPRESENTED	VALUE IN TABLE MEMORY	2K TABLE MEMORY ROM ADDRESS (OCTAL)
!ZERO	ZERO	0.0	4371
!ONE	ONE	1.0	4001
!TWO	TWO	2.0	4002
!THREE	THREE	3.0	4441
!FOUR	FOUR	4.0	4442
!FIVE	FIVE	5.0	4443
!SIX	SIX	6.0	4444
!SEVEN	SEVEN	7.0	4445
!EIGHT	EIGHT	8.0	4446
!NINE	NINE	9.0	4447
!TEN	TEN	10.0	4450
!SIXTN	SIXTEEN	16.0	4451
!HALF	HALF	0.5	4427
!THIRD	ONE THIRD	0.333333333	4430
!FOURTH	ONE FOURTH	0.25	4431
!FIFTH	ONE FIFTH	0.2	4432
!SIXTH	ONE SIXTH	0.166666667	4433
!SVNTH	ONE SEVENTH	0.142857143	4434
!EGHTH	ONE EIGHTH	0.125	4435
!NINTH	ONE NINTH	0.111111111	4436
!TENTH	ONE TENTH	0.1	4437
!SXNTH	ONE SIXTEENTH	0.0625	4440
!SQRT2	SQRT(2)	1.414213562	4203

Table 3-2 Table Memory Constants (cont.)

SYMBOL	CONSTANT REPRESENTED	VALUE IN TABLE MEMORY	2K TABLE MEMORY ROM ADDRESS (OCTAL)
!SQRT3	SQRT(3)	1.732050808	4422
!SQRT5	SQRT(5)	2.236067977	4423
!SQT10	SQRT(10)	3.162277660	4424
!ISQT2	1.0/SQRT(2)	0.707106781	4206
!ISQT3	1.0/SQRT(3)	0.577350269	4452
!ISQT5	1.0/SQRT(5)	0.447213596	4453
!ISQ10	1.0/SQRT(10)	0.316227766	4454
!CBT2	CBRT(2)	1.259921050	4417
!CBT3	CBRT(3)	1.442249570	4420
!QDRT2	(2.0)**1/4	1.189207115	4421
!LOG2E	LOG2(E)	1.442695041	4317
!LOG2	LOG10(2)	0.301029996	4411
!LOGE	LOG10(#)	0.434294432	4337
!LN2	LN(2)	0.693147181	4336
!LN3	LN(3)	1.098612289	4407
!LN10	LN(10)	2.302585093	4410
!E	E	2.718281828	4403
!INVE	1.0/E	0.367879441	4404
!ESQ	E**2	7.389056096	4405
!PI	PI	3.141592654	4402
!TWOPI	2*PI	6.283185308	4415
!INVPI	1.0/PI	0.318309886	4412
!P12	P1/2	1.570796327	4312

Table 3-2 Table Memory Constants (cont.)

SYMBOL	CONSTANT REPRESENTED	VALUE IN TABLE MEMORY	2K TABLE MEMORY ROM ADDRESS (OCTAL)
!PI4	PI/4	0.785398164	4373
!PI180	PI/180	0.017453293	4413
!PISQ	PI**2	9.869604404	4414
!SQTPI	SQRT(PI)	1.772453851	4416
!LNPI	LN(PI)	1.144729886	4406
!GAMMA	GAMMA	0.577215663	4425
!PHI	PHI	1.618033989	4426

Elementary Function Tables

Symbol	Elementary Function	Table Memory Address (Octal)
!DIV	DIVIDE	4000
!DIVD2	HALF ADDRESS	2000
!SQRT	SQUARE ROOT	4202
!SNCS	SIN/COS/	4306
!LOG	LOGARITHM	4333
!EXP	EXPONENTIAL	4317
!ATAN	ARC TANGENT	4365

FFT Cosine Table Constants

Symbol	Description	Value
!FFTSZ	Size of installed FFT cosine table	2048 = 4000 (octal)
!FFTX2	Size times 2	4096 = 10000 (octal)
!FFTX4	Size times 4	8192 = 20000 (octal)
!FFTX8	Size times 8	16384 = 40000 (octal)

3.3.3 COMMENT FIELD

The remainder of any line following a quote mark (") is treated as a comment by the assembler and is ignored. The comment field is terminated by a carriage return. Thus, an instruction can be written as follows:

```
LOOP:  FADD DPS, DPY;      "DO AN ADD
       FMUL TM, MD;        "AND A MULTIPLY
       BFGT DONE           "AND A BRANCH
                           "ALL IN ONE INSTRUCTION
```

3.4 PSEUDO-OPERATION STATEMENTS

Pseudo-operations are directives to the assembler which control certain aspects of the assembly translation process. Each pseudo-op must appear on a separate line in the source text. All pseudo-op names start with a dollar sign ($). As with instruction statements, pseudo-op statements can be labeled and have comments.

3.4.1 $TASK

This pseudo-op identifies the routine that follows as an FPS-100 supervisor task. Tasks require special treatment by the LOD100 loader. $TASK passes parameters for the task communication block (TCB) to LOD100 through the object module. If specified, this pseudo-op must appear as the first statement in a program. The format of this statement is as follows:

```
$TASK    idn {/M} {priority} {/I} {/S}
```

idn	A 1- to 3-digit task identification number which LOD100 later uses to create TCB identifier. The TCB identifier later created is a common block with name TCBidn. So, for example, if a task is designated with an identification number of 5, the user can locate its TCB address by referencing the common block TCB005.

/M	If specified, this task uses minimal machine resources (only those saved in the minimum state save). If not specified, this task uses full machine resources. This parameter is normally used for system tasks, such as I/O controller tasks. This option can be used if the following registers are not needed: s-pad registers 8-15 DPY write buffer all DPX and DPY registers except DPX(0)-DPX(3) DPA floating adder floating multiplier flags
priority	Initial run priority and default priority of the task. Values between 1 and 255 can be specified, with 255 the highest priority. If this parameter is not present, a value of 100 is assumed.
/I	For the purpose of initializing the supervisor ready queue, this inidicates that the previously specified or default priority should be ignored and this task placed at the front of the ready queue. This optional parameter should normally be used only for I/O controller tasks, since it actually results in performing part of the system bootstrapping function (it causes the I/O controller tasks to be waiting for action before any user tasks start).
/S	If specified, the priority of the task is slaved. Thus, when the task is activated, it acquires the priority of the activating task.

The priority, /I, and /S parameters can also be specified at load time with LOD100 commands. The LOD100 commands override the parameters entered with $TASK.

3.4.2 $ISR

This pseudo-op identifies the routine that follows as an interrupt service routine. If specified, this pseudo-op must appear before the $TITLE pseudo-op. The format of this statement is as follows:

```
$ISR    index
```

index	Device number of the I/O device which this routine services. This number must be the same as the device's bit number in the IMASK register. Possible values are 1 through 15.

3.4.3 $TITLE

This pseudo-op names a program. The name need not be unique among the other symbols in the program. The $TITLE pseudo-op must occur as the first or second statement in a program. The format of this statement is as follows:

```
$TITLE name
```

name	Name of the program.

Examples:

```
$TITLE FFT
$TITLE DIVIDE
```

3.4.4 $ENTRY

This pseudo-op declares a symbol to be global; that is, a symbol which is defined in this program and may be referenced by other separately assembled programs. The identified symbol must be defined in the program either by the $EQU pseudo-op or by its use as a label. $ENTRY pseudo-ops must occur before any instruction statements in the program.

If an entry point defined with the $ENTRY pseudo-op is declared host-callable with the LOD100 loader, a host FORTRAN UDC HASI (Host-Arithmetic processor Software Interface) subroutine is created for it. The term UDC stands for user directed calls. When a UDC HASI is created, the calling parameters are integer values or FPS-100 memory addresses that are loaded into s-pads just prior to the execution of the FPS-100 routine. Data transfer/FPS-100 execution synchronization and main data memory allocation are controlled by the user with calls to APX100 routines such as APPUT, APGET, APWD, and APWR. LOD100 generates a HASI that loads and executes the FPS-100 code. A sample UDC subroutine is shown in section 3.9.1. For a complete description of HASIs, refer to the LOD100 Reference Manual.

The format of this statement is as follows:

```
$ENTRY   symbol{,parnum}
```

symbol	A 1-to-6 character symbol which can be referenced by other separately assembled programs. This symbol must be defined with the $EQU pseudo-op or by its use as a label. When referenced externally, execution begins at the location specified by the value of symbol.
parnum	If the routine is host-callable, this parameter must be present, specifying the number of s-pad parameters expected in the call. This may be a number from $0\text{-}15_8$.

Examples:

```
$ENTRY A          "Not host-callable
$ENTRY B,6        "Expect 6 s-pad parameters
$ENTRY C,0        "Expect 0 s-pad parameters
```

3.4.5 $SUBR

This pseudo-op declares a symbol to be an entry point. It is equivalent to the $ENTRY pseudo-op except that if a $SUBR entry point is declared host-callable with LOD100, a host FORTRAN ADC HASI (host-arithmetic processor software interface) subroutine is created. The term ADC stands for auto-directed calls. When an ADC HASI is created, the calling parameters to a routine have a meaning identical to those in a call to a FORTRAN subroutine (referred to as "call by reference"). LOD100 generates a HASI that, in addition to loading and executing the FPS-100 code, handles all data transfers and the main data memory allocation. A sample ADC subroutine is shown in section 3.9.2. For a complete description of HASIs, refer to the LOD100 Reference Manual.

The format of this statement is as follows:

```
$SUBR   symbol{,parnum}
```

symbol	Symbol which can be referenced by other separately assembled programs. This symbol must be defined with the $EQU pseudo-op or by its use as a label. When referenced externally, execution begins at the location specified by the value of symbol.
parnum	Number of formal parameters in the routine. The local data block for this routine (.LOCAL) should contain at least parnum locations for parameter addresses. (Refer to section 3.6 for further discussion of .LOCAL.) If no local data block is declared, LOD100 creates one of size parnum when an FTN100 call to this entry point occurs or when the entry point is declared host-callable. If this parameter is not present, a value of 0 is assumed.

Examples:

```
$SUBR A
$SUBR BBB,6          "Expect 6 parameters
$SUBR K,0            "Expect 0 parameters
```

3.4.6 $GLOBAL

This pseudo-op declares symbols to be absolute entry points. These entry points are similar to those declared with the $ENTRY and $SUBR pseudo-ops. However, $GLOBAL is used when absolute values are required or when external references are made by ASM100 instructions that require absolute references. At load time, no relocation is performed.

The format of this statement is as follows:

$GLOBAL symbol$_1$,symbol$_2$,...,symbol$_n$

symbol$_i$	Symbol which can be referenced by other separately assembled programs. This symbol, when defined with the $EQU pseudo-op, declares an absolute address.

3.4.7 $INTEGER

This pseudo-op declares variables that later appear in $COMMON or $PARAM statements to be of type integer. This pseudo-op must appear in the program before any $COMMON or $PARAM statements.

The format of this statement is as follows:

$INTEGER symbol$_1$,symbol$_2$,...,symbol$_n$

symbol$_i$	Name of a variable which later appears in a $COMMON or $PARAM statement.

Examples:

```
$INTEGER A
$INTEGER ARE,BEE,ZED11
```

3.4.8 $REAL

This pseudo-op declares variables that later appear in $COMMON or $PARAM statements to be of type real. This pseudo-op must appear in the program before any $COMMON or $PARAM statements.

The format of this statement is as follows:

$REAL $symbol_1, symbol_2, \ldots, symbol_n$

$symbol_i$	Name of a variable which later appears in a $COMMON or $PARAM statement.

Examples:

```
$REAL IVEC, BLT,J,IPQR
$REAL JNUM
```

3.4.9 $TRIPLE

This pseudo-op declares variables that later appear in $COMMON statements to be of type double precision integer (38-bit integers). This pseudo-op must appear in the program before any $COMMON statement. The format of this statement is as follows:

$TRIPLE $symbol_1, symbol_2, \ldots, symbol_n$

symbol	Name of a variable which later appears in a $COMMON statement.

A double precision integer specified with $TRIPLE can only occur in common data blocks other than the .LOCAL block. It is not possible for a subroutine to have double precision arguments. The only other place that a double precision value can be referenced is in the $DATA statement.

3.4.10 $COMIO

For host-callable routines, this pseudo-op declares the direction of transfer of subsequent common blocks. Data in some common blocks need only be transferred from host to FPS-100. Other common blocks may require data transfers only from FPS-100 to host. Still others need both. This pseudo-op declares the type of transfer for a common block. $COMIO allows the HASI subroutines to be smaller and more efficient. (host-arithmetic processor software Interface routines are host FORTRAN routines created by LOD100 for each host-callable routine.)

If the $COMIO pseudo-op is present, it must appear in the program before the associated $COMMON. If it is omitted, common blocks are transferred in both directions.

The format of this statement is as follows:

$COMIO $comnam_1$ opt_1, $comnam_2$ opt_2,...,$comnam_n$ opt_n

$comnam_i$ Name of common block.

opt_i	Specifies the type of transfer. The following are acceptable values:

opt	description
0	Data in this common block should not be transferred.
1	Data in this common block should be transferred only from FPS-100 to host.
2	Data in this common block should be transferred only from the host to the FPS-100.
3	Data should be transferred from the host to the FPS-100 and back.

NOTE

If two host-callable routines reference the same common block, their $COMIO specifications for it should be the same.

Examples:

```
$COMIO ALO 3
$COMIO BC 1
```

3.4.11 $PARAM

This pseudo-op is used to describe the formal parameters of a subroutine that is to be host-callable and for which LOD100 is to create an ADC HASI (the entry point is declared with the $SUBR pseudo-op). LOD100 creates a loader parameter block, block number 10, whose values correspond directly to the parameters of this statement. Refer to the LOD100 Reference Manual for a description of the loader blocks. This statement must appear before any executable code.

The format of this statement is as follows:

$PARAM no, $symbol_1${($(ind_1,\ldots,ind_n)$)}{/type}{/op},...,
$symbol_n${($(ind_1,\ldots,ind_n)$)}{/type}{/op}

no	Number of parameters to be described.
$symbol_i$	The name of a parameter. Later reference to this ith parameter symbol refers to the ith position in this routine's local data block. If a .LOCAL common block is declared, the first elements declared in that common block must correspond with the elements declared with the $PARAM pseudo-op. The values of the elements in the .LOCAL common block are addresses of the formal parameters. Refer to section 3.6 for further discussion of .LOCAL.
ind_i	Each ind_i describes a dimension of an array. This parameter can be an integer or an integer expression. If expression ind_i is preceded by a number sign (#), this dimension is to be dynamically defined at run time by the value of the ind_ith parameter.

type — Parameter type. Acceptable values are:

type	description
I	integer
R	real

Unless the symbol has appeared in a $REAL or $INTEGER pseudo-op, the default type for this parameter is integer.

op — I/O option. The following can be specified:

op	description
IP	The parameter is an input argument and must be passed only into the FPS-100 during host call.
OP	The parameter is an output argument and must only be passed back from the FPS-100.

If both are specified, the parameter is defined as both. If neither is specified, no data is transferred but space is allocated in main data memory for the parameter.

Example:

```
$PARAM   2,  A(10,#2)/R, INDEX/I/IP
```

In this example, two parameters are defined. The first is a two-dimensional real array whose first dimension is 10 and whose second is defined at run time by the second parameter (INDEX). Its I/O option is defaulted to both IP and OP. The second parameter is INDEX. It is an integer scalar whose I/O option has been defined to IP for input only.

3.4.12 $COMMON

This pseudo-op is used to declare a main data memory data area (common or local data block). This pseudo-op must occur before any executable code. The format of this statement is as follows:

$COMMON /name/ $symbol_1${$(ind_1$,...,$ind_n)$}{/type},...,symbol ind_n)}{/type}

name	Name of the common block (.LOCAL for a local data block). If .BLANK, absent, or //, blank common is assumed.
$symbol_i$	Name of an element in the data block (either array or scalar). Later occurrences of this symbol reference its base address in the data block.
ind_i	Dimension of an array.
type	Type of the variable. Acceptable values are as follows:

type	description
I	integer
R	real
T	triple (double precision integer)

If omitted, the default is the type specified earlier in the $INTEGER, $TRIPLE, or $REAL pseudo-ops. Otherwise, the default is a 16-bit integer.

Example:

```
$COMMON /COMA/ I, J, A(10)/R, K, K1/T
```

3.4.13 $DATA

This pseudo-op is used to initialize values in a data area declared with the $COMMON pseudo-op. It should occur in the program before any executable code but after the common blocks to be initialized. The format of the statement is as follows (brackets indicate that one and only one line must be chosen):

$$\text{\$DATA} \quad \text{symbol}_1\{(\text{ind}_1)\}\{/\text{repcnt}_1\} \begin{bmatrix} \text{value}_1 \\ \text{exp}_1, \text{himan}_1, \text{loman}_1 \\ \text{relsym}_1\{\pm\text{rval}_1\} \end{bmatrix}, \ldots,$$

$$\text{symbol}_n\{(\text{ind}_n)\}\{/\text{repcnt}_n\} \begin{bmatrix} \text{value}_n \\ \text{exp}_n, \text{himan}_n, \text{loman}_n \\ \text{relsym}_n\{\pm\text{rval}_n\} \end{bmatrix}$$

symbol_i — Name of an element that must be previously defined in a $COMMON pseudo-op.

ind_i — Indicates that element ind_i-1 after the address of symbol is initialized. Both positive and negative values can be specified for ind_i. Note that only one dimension of subscripting is allowed.

repcnt_i — Repetition count. This specifies the number of words starting at symbol_i(indi) that are to be given the value that follows. The repetition count must be an integer and not an integer expression.

value_i — Initial value for $\text{symbol}_i(\text{ind}_i)$. This value must conform to the type described previously. It must be a single real value or an expression consisting only of integers and/or local symbols (labels or symbols appearing on the left side of $EQU pseudo-ops; refer to section 3.4.14).

exp_i, $himan_i$, $loman_i$ — Three values which initialize a double precision integer previously declared with a $TRIPLE pseudo-op. The exp_i specifies the exponent portion (10 bits), $himan_i$ specifies the high mantissa portion (12 bits), and $loman_i$ specifies the low mantissa portion (16 bits) of the 38-bit word. Each one of these parameters can be a value (refer to description of $value_i$) or a relocatable symbol (refer to description of $relsym_i$).

NOTE

Due to restrictions in the host-FPS-100 hardware interface, at this time it is possible to transfer only 32 bits of information. Therefore, only four bits of exponent should be specified in exp_i; the remainder is lost.

$relsym_i$ — Name of a relocatable symbol; that is, a symbol whose actual value cannot be determined until load time. Relocatable symbols include any symbols that are not local symbols and include external symbols (declared with the $EXT pseudo-ops) and symbol names for variables in common (declared with $COMMON pseudo-ops). However, only variables in common that are integers can be initialized to values dependent on relocatable symbols.

Examples:

```
$DATA I 1, L(4)/10 2, K 3, A(2) 99.99, PI 3.1415
```

In this example, I is set to 1, L(4) and nine locations following L(4) are set to 2, K is set to 3, A(2) is set to 99.99, and PI is set to 3.1415.

```
$EXT    EXTLAB, LAB1
$DATA   G EXTLAB,  H  LAB1-3
```

In this example, the variables G and H are initialized to the addresses of external variables EXTLAB and LAB1.

```
$EXT    LAB2
$DATA   TRIPA  3,5,17,    TRIPB  17,4095,LAB2+5
```

In this example, two double precision integers are initialized. For TRIPB, the second double precision integer, the low mantissa portion is initialized to the address plus 6 of the external LAB2.

3.4.14 $EXT

This pseudo-op declares global symbols which are referenced by this program but are defined by another separately assembled program. $EXT pseudo-ops must occur in the program before any instruction statements. The format of the statement is as follows:

$EXT $symbol_1$,$symbol_2$,...,$symbol_n$

$symbol_i$	Symbol referenced in the program, but defined elsewhere.

Examples:

```
$EXT FLOAT, SCALE, FFT
$EXT DIVIDE
```

3.4.15 $VAL

This pseudo-op defines 64 bits of data to fill one program source word. The format of this statement is as follows:

$VAL int_1,int_2,int_3,int_4

int_i	One of four 16-bit integers or integer expressions which represent the four 16-bit quarters of a program source word. This parameter may contain an external reference.

Examples:

```
$VAL -377, 104763, 10, LOOP + 6
$VAL 0, 0, 2000, 33
```

3.4.16 $FP

This pseudo-op fills the right-most 38 bits of a program source word with a specified floating-point number. The left-most 26 bits of the word are cleared. The format of this statement is as follows:

```
$FP   value
```

value	Floating-point number.

Examples:

```
     $FP  6.0023E23
     $FP  2
     $FP  E-17
PI:  $FP  3.141592653   "PI
```

A floating-point number (for example, a constant for an algorithm) can be read out of program source memory and onto the data pad bus using the RPSF op-code. As an example, the following loads the contents of location PI onto data pad X:

```
RPSF PI; DPX<DB  "GET PI INTO DPX
```

3.4.17 $EQU

This pseudo-op equates a symbol with an expression. If user-defined symbols are used in the expression, they must be previously defined in the program. The format of this statement is as follows:

```
symbol   $EQU   expres
```

symbol	Symbol to which a value is assigned.
expres	Expression which is assigned to symbol.

Alternatively, the equals sign (=) can be used in place of $EQU.

If the expression assigned to the symbol contains an external, the symbol acquires the attributes of an external and must be treated as an external. For example, if the symbol is used in another expression, that expression cannot contain other externals.

Examples:

```
A       $EQU  321
LOOP    $EQU  LOC + 3
HERE    $EQU  . - 3
MASK    $EQU  132*3+6
A = 6
X = A*3
```

3.4.18 $LOC

This pseudo-op sets the current location counter to the value of an expression. If symbols are used in the expression, they must be previously defined in the program. This pseudo-op must not be used to set the location counter backwards. The format of this statement is as follows:

```
$LOC expres
```

expres	Integer expression whose value determines the setting of the location counter.

Examples:

```
$LOC   300
$LOC   . + 6   "LEAVE NEXT SIX UNUSED
$LOC   LOOP +10
```

NOTE

$LOC should not be set to an absolute address as in the first example if the output is to be linked relocatably with other programs.

3.4.19 $RADIX

This pseudo-op changes the default number radix to the value of the expression. The format of this statement is as follows:

```
$RADIX expres
```

expres	Expression which determines the default number radix. This expression is entered and evaluated in base 10. The value of the expression must be either 8, 10, or 16.

Examples:

```
$RADIX 10
$RADIX 8
```

3.4.20 $CALL

$CALL can be used to call FTN100 subroutines or ASM100 subroutines that conform to certain $SUBR and $PARAM conventions. These conventions are described in section 3.6. The format of this statement is as follows:

$CALL subnam ($arg_1$,$arg_2$,...,$arg_n$)

subnam	Name of an FTN100 or ASM100 subroutine. It must be declared external.
arg_i	Arguments to be passed to the called subroutine, if any. Each arg_i can be an expression (which is evaluated at assembly time). The value of arg_i represents the address in main data memory of the actual argument and not the argument itself.

The user and the $CALL pseudo-op reference the address of the called subroutine's local data block by means of the following:

```
DB=#subroutine-name
```

The term #subroutine-name is interpreted by the assembler to mean the address of the called subroutine's local data block (.LOCAL). The $CALL places the addresses of the actual parameters into the called routine's local data block, sets s-pads 0 and 1 to the correct values, and jumps to the routine.

CAUTION

Extreme caution is advised when using this pseudo-op since the addresses of the arguments in the $CALL are calculated at assembly time, not at run time. This presents problems if the argument addresses cannot be known until run time. This is the case if the arguments of the $CALL include the subroutine's own formal parameters. In such cases, the user must calculate the address of the argument, place the value of the address in a data pad register (except for DPX(3)), and use the data pad as an argument of the call. For example, suppose a routine wishes to pass its own parameter (PARAM) to subroutine SUB. The user calculates the address of the argument and places the result in DPX(1). The subroutine could be called with the following:

```
LDMA; DB=PARAM
NOP
NOP
DPX(1)<MD
$CALL SUB (DPX(1))
```

Example:

```
$COMMON /.LOCAL/ A/R, I(20), FIVE
$DATA FIVE 5
$CALL SUB (A, I+14, FIVE)
```

NOTE

The $CALL expands into actual ASM100 code that is then assembled. This code also appears on the listing. The number of program source words used is (2 X (number of arguments) + 4) unless no arguments are specified, in which case only two program source locations are used. The following is the expansion of the previous example ($CALL SUB (A,I+14,FIVE)):

```
LDMA; DB=# SUB-1
DPX(3)<DB; DB=A
INCMA; MI<DPX(3)
DPX(3)<DB; DB=I+14
INCMA; MI<DPX(3)
DPX(3)<DB; DB=FIVE
INCMA; MI<DPX(3)
LDSPI 0; DB=# SUB
LDSPI 1; DB=3
JSR SUB
```

3.4.21 $INSERT

This pseudo-op causes source code to be read from the designated file. The line number is reset. When end-of-file is encountered, source is again read from the file originally specified in the ASM100 call. The line count is set to its original value when the end of the $INSERT file is reached. Also, when the $INSERT file is reached during Pass 1 of assembly, the line containing the $INSERT is written to the terminal. When its end is reached, the message "END $INSERT" is written to the listing. (This happens during Pass 2, also.)

The format of this statement is as follows:

```
$INSERT filename
```

filename	Name of file containing source code to be inserted in the source stream.

Example:

```
$INSERT  FILEA
```

3.4.22 $IF...$ENDIF

These pseudo-ops allow conditional assembly. If the expression which follows $IF evaluates to zero, any subsequent source lines up to $ENDIF are not assembled. However, they do appear on the listing.

The format of the $IF statement is as follows:

```
$IF expression
```

expression	If the expression evaluates to zero, the subsequent source lines up to $ENDIF are ignored. If expression is unequal to zero, the source lines are assembled.

The format of the $ENDIF statement is as follows:

```
$ENDIF
```

NOTE

$IF pseudo-ops can be nested. That is, $IF/$ENDIF combinations can appear between other $IF/$ENDIF combinations.

Example:

```
$IF PROG
PROG $EQU 0
$ENDIF
```

3.4.23 $LIB...$ENDLIB

These pseudo-ops cause loader library start blocks and library end blocks to be written to the object file. LOD100 treats an object module preceded by a library block as a library and loads only those routines that satisfy unsatisfied externals.

The format of the $LIB statement is as follows:

```
$LIB
```

The format of the $ENDLIB statement is as follows:

```
$ENDLIB
```

3.4.24 $PAGE

This pseudo-op begins a new page on the listing. The format of this statement is as follows:

```
$PAGE
```

3.4.25 $BOX...$ENDBOX

These pseudo-ops designate that all source lines found between them are considered comments and are surrounded by a box of asterisks when the listing is produced. They can be used to improve the readability of the listing.

The format of the $BOX statement is as follows:

```
$BOX
```

The format of the $ENDBOX statement is as follows:

```
$ENDBOX
```

3.4.26 $NOLIST

This pseudo-op specifies that no source code appears on the listing after this statement. A $LIST pseudo-op terminates this condition. If no listing was specified in the call to ASM100, this pseudo-op has no effect.

The format of this statement is as follows:

```
$NOLIST
```

3.4.27 $LIST

This pseudo-op specifies that source code after this statement appears on the listing. A $NOLIST pseudo-op terminates this condition. If no listing was specified in the call to ASM100, this pseudo-op has no effect.

The format of this statement is as follows:

```
$LIST
```

3.4.28 $END

This pseudo-op causes ASM100 to terminate the assembly. The format of this statement is as follows:

```
$END
```

3.4.29 DUMMY FMUL AND FADD PUSHERS

When programming pipelines as described in Part 1 of the Programmer's Reference Manual, it is convenient for readability to include in the code all the FMULs and FADDs that are used as pushers in any of the columns of the handwritten pipelines. These are coded without parentheses. Any FMUL or FADD without arguments does not conflict with other arithmetic arguments of like type and is completely ignored unless it is the only op-code of its type.

Example:

```
FADD DPX1, DPY1; FMUL FM,FA; FADD
```

In this example, the last FADD is ignored.

NOTE

> Any FMUL op-code used as a pusher in an instruction word without other FMULs actually results in the op-code FMUL TM,MD. Though unlikely, this op-code could cause an underflow or overflow condition when the meaningless result is pushed through the multiplier pipeline (the result is pushed through the pipeline when the instruction occurs in a loop). Unexplained underflow or overflow conditions discovered during program debugging may be the result of FMUL pushers.

3.4.30 EXTERNAL VARIABLES

The assembler assures that any variable beginning with an exclamation point (!) is an external variable and is defined outside the referencing program. Thus, any external variables used which start with ! need not be declared external with the $EXT pseudo-op.

3.5 ORDER OF PROGRAM STATEMENTS

There is a definite ordering of statement types within a program which must be followed. The $TASK or $ISR pseudo-op, if used, must appear first. The $TITLE pseudo-op must appear next, followed by any $ENTRY, $SUBR, or $GLOBAL pseudo-ops. $END must be the last statement. The remainder of the pseudo-ops (if present) and the program body appear in the following order:

```
$TASK or $ISR        pseudo-op
$TITLE               pseudo-op
$ENTRY or $SUBR      pseudo-op(s)
$GLOBAL              pseudo-op(s)
$EXT                 pseudo-op(s)
$INTEGER             psuedo-op(s)
$REAL                pseudo-op(s)
$TRIPLE              pseudo-op(s)
$PARAM               pseudo-op
$COMIO               pseudo-op(s)
$COMMON              pseudo-op(s)
$DATA                peudo-op(s)
"program, etc."
   .
   .
   .
$END
```

3.6 CREATING FTN100 CALLABLE ASM100 SUBROUTINES

In order to create ASM100 subroutines that can be called from FTN100 program units, the following conventions must be followed:

- The $COMMON pseudo-op should be used to declare a local common block called .LOCAL. This common block must contain at least as many locations as the number of formal parameters in the routine. Any routine which calls this routine places the addresses of the formal parameters in this common block. Also, s-pad register 0 is set to the address of the .LOCAL block, and s-pad register 1 is set to the number of parameters passed. If the subroutine has no formal parameters, it is not necessary to declare a .LOCAL block.

NOTE

> If the .LOCAL block is not declared by the ASM100 programmer but is referenced inside the routine (or by another routine), it is created at load time. This feature is used by the FPS-100 Math Library but is not suggested for general use.

- The $COMMON and $DATA pseudo-ops can also be used to declare and initialize labeled common blocks. This allows data to be shared between subroutines.

In general then, in order to be callable from an FTN100 routine, an ASM100 routine must have the following format:

```
$TITLE pseudo-op
$COMMON/.LOCAL/ pseudo-op (if there are formal parameters)
$COMMON pseudo-op (if common blocks are used)
$DATA pseudo-op (if the common block is to be initialized)
code
   .
   .
   .
$END pseudo-op
```

For an example, refer to section 3.9.2.

3.7 CALLING FTN100 ROUTINES FROM ASM100

When an FTN100 routine or an ASM100 routine that conforms to the FTN100 calling conventions is called, it expects the calling routine to conform to the conventions described in section 3.6. That is, parameters are passed by placing their addresses in the called routine's .LOCAL data block (by using the $CALL pseudo-op or by writing equivalent ASM100 code). The $COMMON pseudo-op can also be used to create a common area used to pass data to the called routine.

The general form of an ASM100 program with a call to an FTN100 routine is as follows:

```
$TITLE pseudo-op
    .
    .
    .
$EXT pseudo-op
$COMMON pseudo-op
$DATA pseudo-op
    .
    .
    .
$CALL pseudo-op
    .
    .
    .
$END
```

3.8 CALLING ASM100 SUBROUTINES FROM THE HOST

The following sections describe the procedures necessary to call ASM100 subroutines from the host. Both the auto-directed calls (ADC) manner and the user directed calls (UDC) manner are considered.

3.8.1 AUTO-DIRECTED CALLS (ADC) TO ASM100 SUBROUTINES

If a subroutine is called in the ADC manner, data can be passed between host FORTRAN programs and ASM100 subroutines as arguments or common blocks. However, only arguments and common blocks declared in host-callable routines are transferred from host to FPS-100 and back. The data is passed as specified in the ASM100 subroutine. If a host-callable ASM100 routine contains a common block which does not exist on the host, the common block is created on the host by the HASI subroutine which is generated by LOD100. This is always done unless the user specifies otherwise with a $COMIO pseudo-op (refer to section 3.4.10). Since the HASI created by LOD100 contains this common block, any discrepancies which exist between the ASM100 routine and the host FORTRAN program cause meaningless data to be passed. Also, only labeled common blocks can be shared between host FORTRAN and ASM100 subroutines.

The creation of a HASI for this type of call is triggered by the use of the $SUBR pseudo-op (instead of the $ENTRY pseudo-op) to declare the subroutine's entry point. The form of the actual call to the ASM100 routine is exactly like that of a FORTRAN call.

Example:

```
CALL MYSUB (A,B,2)
```

The parameters A, B, and 2 are automatically transferred to and from the FPS-100 according to information supplied in the $PARAM pseudo-op. Since this is a FORTRAN-style call, the ASM100 subroutine (in this case, MYSUB) must conform to the FTN100 calling conventions described in section 3.6.

Whenever possible, data should be passed between host programs and ASM100 subroutines as common blocks rather than as arguments. Data in common blocks is generally passed faster than that specified with $PARAM pseudo-ops.

Another method of increasing the rate of data transfer is grouping the elements of a common block by type in the $COMMON pseudo-op. This is helpful because when data is actually transferred from host to FPS-100, only one kind of data (real or integer) can be transferred at a time. Grouping the elements in a $COMMON pseudo-op by type minimizes the number of data transfers necessary.

Example:

```
$COMMON /X/ A(100)/R, J(100)/I, B(100)/R
```

This requires three data transfers: one to transfer the real array A, one to transfer the integer array J, and one to transfer the real array B. However, suppose this statement had been written as follows:

```
$COMMON /X/ A(100)/R, B(100)/R, I(100)/I
```

In this case, only two data transfers are necessary: one to transfer the real arrays A and B and one to transfer the integer array J. The actual data transfer by types is done internally; the ASM100 programmer need not be concerned about it. Be aware, however, that grouping integer items together and real items together in $COMMON pseudo-ops results in faster data transfers.

The programmer should also be aware that problems can arise when a program is called from the host with multiple occurrences of the same parameter. These problems can occur because parameters are passed to and from the FPS-100 in the order in which they were specified on the $PARAM pseudo-op. The following examples illustrate the problem. These examples use FTN100 subroutines rather than ASM100 subroutines; however, the problems apply to ASM100 routines also.

Examples:

The following FTN100 subroutine VADNZ is written to add two arrays, put the results in a third array, and set the first two to zero.

```
      SUBROUTINE VADNZ(A,B,C,N)
      DIMENSION A(N),B(N),C(N)
      DO 10 I=1,N
      C(I)=A(I)+B(I)
      A(I)=0.
   10 B(I)=0.
      RETURN
      END
```

The expected results are returned when a host FORTRAN program calls this subroutine using three different arrays.

```
           .
           .
           .
      DIMENSION X(100),Y(100),Z(100)
           .
           .
           .
      CALL VADNZ(X,Y,Z,100)
           .
           .
           .
```

Upon return from the subroutine, array Z contains the sum of X and Y. But, suppose the programmer attempted to call VADNZ as follows:

```
CALL VADNZ(X,X,X,100)
```

In this case, results are unpredictable. When arguments are passed from the host to the FPS-100, storage locations are reserved for each of the arguments, whether or not the arguments are unique. In the case of the last call, storage locations in the FPS-100 are set aside for three copies of array X. Upon completion of the subroutine, only one array X contains the sum of arrays X and X. In the host, the ultimate value depends on the order in which the arguments are transferred back to the host from the FPS-100. The array X in the host contains the values of the last array X transferred.

A similar problem occurs if elements of the same array are used as actual parameters in the call to a subroutine. For example, consider the following routine:

```
      SUBROUTINE ADDN (I,J,K,N)
      DIMENSION I(N),J(N),K(N)
      DO 10 L=1,N
   10 K(L)=I(L)+J(L)
      RETURN
      END
```

Suppose the routine is called as follows:

```
CALL ADDN (JJ(1),JJ(2),JJ(3),100)
```

The user might expect array JJ to contain something similar to the Fibonacci series. However, instead of using one array JJ, three arrays are created in the FPS-100: one starting at JJ(1), one starting at JJ(2), and one starting at JJ(3). Thus, although the user might expect the results of the first addition to be available as an operand of the second, it is actually stored in a different array and is not used in subsequent calculations. Thus, the subroutine does not return the expected results.

Difficulties arise not only from specifying the same variable for multiple formal parameters but also from specifying a variable as a formal parameter and as an element in common.

For more information regarding ADC type HASIs, refer to the LOD100 Reference Manual.

3.8.2 USER DIRECTED CALLS (UDC) TO ASM100 SUBROUTINES

A user directed call to an ASM100 subroutine is triggered by the use of the $ENTRY pseudo-op to declare the entry point. This type of call does not pass parameters to the FPS-100 (as with auto-directed calls). Since parameters are not passed automatically, this type of call can be much more efficient time-wise than an auto-directed call. It does, however, require that the user pass and return parameters in main data memory with APPUT, APGET, and other APX100 subroutine calls (refer to the APX100 Manual for descriptions of these calls).

The actual form of the call to a user's ASM100 subroutine is as follows:

```
CALL MYSUB (IA,2000,3000)
```

In this call, IA, 2000, and 3000 are not parameters but rather addresses of parameters that were placed in main data memory earlier by the user. These addresses are placed in s-pad registers before control passes to the subroutine.

Further information concerning user directed calls can be found in the LOD100 Reference Manual.

3.9 SAMPLE PROGRAMS

The following sections give examples of ASM100 subroutines and host calling programs.

3.9.1 UDC EXAMPLE

Figure 3-1 illustrates a sample ASM100 subroutine. The object code produced by the ASM100 assembler using this subroutine is used as input to the LOD100 loader. If the routine is declared host-callable at load time, LOD100 generates a UDC type HASI (host-arithmetic processor software interface) and a load module. (Refer to the LOD100 Reference Manual for a complete description of the load process.)

Figure 3-2 illustrates a host FORTRAN program used to call the ASM100 subroutine. This program and the HASI must be compiled using the host FORTRAN compiler and linked using the host loader before program execution can occur. The ASM100 routine can also be called by other ASM100 routines.

```
        $TITLE VCADD
        $ENTRY VCADD, 4
"VECTOR ADD
"ADDS VECTOR A TO VECTOR B AND PUTS THE RESULT INTO VECTOR C
"C(M) = B(M) + A(M)      FOR M = 0 TO N-1


"S-PAD PARAMETERS
        A    $EQU   0                         "BASE ADDRESS OF VECTOR A
        B    $EQU   1                         "BASE ADDRESS OF VECTOR B
        C    $EQU   2                         "BASE ADDRESS OF C
        N    $EQU   3                         "NUMBER OF ELEMENTS IN C


VCADD:  MOV A,A; SETMA                        "FETCH A(0)
        MOV B,B; SETMA                        "FETCH B(0)
        DEC C; DPX(0)<MD                      "SAVE A(0)
LOOP:   INC A; SETMA                          "FETCH A(M+1)
        INC B; SETMA;                         "FETCH B(M+1)
          FADD DPX(0),MD                      "B(M) + A(M)
        DPX(0)<MD;                            "SAVE A(M+1)
          DEC N; FADD                         "  SEE IF DONE?????
        MI<FA; INC C; SETMA;                  "STORE C(M)
          BNE LOOP                            "BRANCH IF NOT DONE
        RETURN
        $END
```

Figure 3-1 UDC Subroutine

```
C
C     THE FOLLOWING IS A HOST PROGRAM ILLUSTRATING THE CALL TO VCADD
C
      DIMENSION A(100),B(100),C(100)
      INTEGER ADDRA,ADDRB,ADDRC
C
C...INITIALIZE THE FPS-100
C
      CALL APINIT(0,0,ISTAT)
      CALL APLLI('LMOD',4,7,1,1,D,D)
C
C...INITIALIZE THE INPUT ARRAYS
C
      DO 10 I=1,100
      A(I)=FLOAT(I)
      B(I)=A(I)
10    CONTINUE
C
C...PUT THE DATA IN THE FPS-100
C
      ADDRA=0
      N=100
      CALL APPUT(A,ADDRA,N,2)
      ADDRB=ADDRA+N
      ADDRC=ADDRB+N
      CALL APPUT(B,ADDRB,N,2)
      CALL APWD
C
C...CALL VCADD
C
      CALL VCADD(ADDRA,ADDRB,ADDRC,N)
      CALL APWR
C
C...RETRIEVE THE DATA FROM THE FPS-100
C
      CALL APGET(C,ADDRC,N,2)
      CALL APWD
C
C...RELEASE THE FPS-100
C
      CALL APRLSE
      STOP
      END
```

Firgure 3-2 Host Calling Program for UDC Subroutine

3.9.2 ADC EXAMPLE

Figure 3-3 illustrates a sample ASM100 subroutine. The object code produced by the ASM100 assembler using this subroutine is used as input to the LOD100 loader. If the routine is declared host-callable at load time, LOD100 generates an ADC type HASI and a load module.

Figure 3-4 illustrates a host FORTRAN program used to call the ASM100 subroutine. This program and the HASI must be compiled using the host FORTRAN compiler and linked using the host loader before program execution can occur. The ASM100 routine can also be called from an FTN100 program. The $PARAM and $COMIO pseudo-ops can be removed if the routine is not designated as host-callable.

```
        $TITLE VCADD
        $SUBR VCADD, 4
        $PARAM 4, AA(#4)/R/IP,AB(#4)/R/IP,AC(#4)/R,AN/I/IP
        $COMMON /.LOCAL/ AA,AB,AC,AN
"VECTOR ADD
"ADDS VECTOR A TO VECTOR B AND PUTS THE RESULT INTO VECTOR C
"C(M) = B(M) + A(M)     FOR M = 0 TO N-1


"S-PAD PARAMETERS
        A    $EQU   0                      "BASE ADDRESS OF VECTOR A
        B    $EQU   1                      "BASE ADDRESS OF VECTOR B
        C    $EQU   2                      "BASE ADDRESS OF C
        N    $EQU   3                      "NUMBER OF ELEMENTS IN C


VCADD:  LDMA; DB=AA                        "LOAD ADDRESS OF A
        LDMA; DB=AB                        "LOAD ADDRESS OF B
        LDSPI A; DB=MD                     "SAVE ADDRESS OF A
        LDMA; DB=AC                        "LOAD ADDRESS OF C
        LDSPI B; DB=MD                     "SAVE ADDRESS OF B
        LDMA; DB=AN                        "LOAD ADDRESS OF N
        LDSPI C; DB=MD                     "SAVE ADDRESS OF C
        MOV A,A; SETMA                     "FETCH A(0)
        LDMA; DB=MD                        "LOAD N
        DEC C                              "FETCH B(0)
        MOV B,B;SETMA; DPX(0)<MD           "SAVE A(0)
        LDSPI N; DB=MD                     "SAVE VALUE OF N
LOOP:   INC A; SETMA                       "FETCH A(M+1)
        INC B; SETMA;                      "FETCH B(M+1)
          FADD DPX(0),MD                   "B(M) + A(M)
        DPX(0)<MD;                         "SAVE A(M+1)
          DEC N; FADD                      "  SEE IF DONE?????
        MI<FA; INC C; SETMA;               "STORE C(M)
          BNE LOOP                         "BRANCH IF NOT DONE
        RETURN
        $END
```

Figure 3-3 ADC Subroutine

```
C
C     THE FOLLOWING IS A HOST PROGRAM ILLUSTRATING THE CALL TO VCADD
C
      DIMENSION A(100),B(100),C(100)
C
C...INITIALIZE THE FPS-100
C
      CALL APINIT(0,0,ISTAT)
      CALL APLLI ('LMOD',4,7,1,1,D,D)
C
C...INITIALIZE THE INPUT ARRAYS
C
      DO 10 I=1,100
      A(I)=FLOAT(I)
      B(I)=A(I)
10    CONTINUE
C
C
C...CALL VCADD
C
      N=100
      CALL VCADD(A,B,C,N)
C
C...RELEASE THE FPS-100
C
      CALL APRLSE
      STOP
      END
```

Figure 3-4 Host Calling Program for ADC Subroutine

3.9.3 ADC EXAMPLE WITH COMMON BLOCKS

Figure 3-5 illustrates the same ASM100 routine as in Figure 3-3, except that it receives its input in a common block and places its output in another common block. The object code produced by the ASM100 assembler using this subroutine is used as input to the LOD100 loader. If the routine is declared host-callable at load time, LOD100 generates an ADC type HASI and a load module. The ASM100 routine in Figure 3-5 is more efficient than the one in Figure 3-3.

Figure 3-6 illustrates a host FORTRAN program used to call the ASM100 subroutine. This program and the HASI must be compiled using the host FORTRAN compiler and linked using the host loader before program execution can occur.

```
        $TITLE VCADD
        $SUBR VCADD
        $RADIX 10
        $COMIO INPUT 2                      "THIS COMMON BLOCK IS INPUT ONLY
        $COMMON /INPUT/ PN,PA(100)/R,PB(100)/R
        $COMIO OUTPUT 1                     "THIS COMMON BLOCK IS OUTPUT ONLY
        $COMMON /OUTPUT/ PC(100)/R
"VECTOR ADD
"ADDS VECTOR A TO VECTOR B AND PUTS THE RESULT INTO VECTOR C
"C(M) = B(M) + A(M)     FOR M = 0 TO N-1


"S-PAD PARAMETERS
        A    $EQU    0                      "BASE ADDRESS OF VECTOR A
        B    $EQU    1                      "BASE ADDRESS OF VECTOR B
        C    $EQU    2                      "BASE ADDRESS OF C
        N    $EQU    3                      "NUMBER OF ELEMENTS IN C


VCADD:  LDMA; DB=PN                         "LOAD N
        LDSPI A; DB=PA                      "LOAD ADDRESS OF A
        LDSPI B; DB=PB                      "LOAD ADDRESS OF B
        LDSPI N; DB=MD                      "SAVE N
        MOV A,A; SETMA                      "LOAD A(0)
        LDSPI C; DB=PC                      "LOAD ADDRESS OF C
        MOV B,B; SETMA                      "LOAD B(0)
        DEC C; DPX(0)<MD                    "SAVE A(0)
LOOP:   INC A; SETMA                        "FETCH A(M+1)
        INC B; SETMA;                       "FETCH B(M+1)
          FADD DPX(0),MD                    "B(M) + A(M)
        DPX(0)<MD;                          "SAVE A(M+1)
          DEC N; FADD                       "  SEE IF DONE?????
        MI<FA; INC C; SETMA;                "STORE C(M)
          BNE LOOP                          "BRANCH IF NOT DONE
        RETURN
        $END
```

Figure 3-5 ADC Subroutine with Common Blocks

```
C
C     THE FOLLOWING IS A HOST PROGRAM ILLUSTRATING THE CALL TO VCADD
C
      COMMON /INPUT/ N,A(100),B(100)
      COMMON /OUTPUT/ C(100)
C
C...INITIALIZE THE FPS-100
C
      CALL APINIT(0,0,ISTAT)
      CALL APLLI('LMOD',4,7,1,1,D,D)
C
C...INITIALIZE THE INPUT ARRAYS
C
      DO 10 I=1,100
      A(I)=FLOAT(I)
      B(I)=A(I)
10    CONTINUE
C
C
C...CALL VCADD
C
      N=100
      CALL VCADD
C
C...RELEASE THE FPS-100
C
      CALL APRLSE
      STOP
      END
```

Figure 3-6 Host Calling Program for ADC Subroutine with Common Blocks

CHAPTER 4

OPERATING PROCEDURES

4.1 USING ASM100

ASM100 is a two-pass assembler which assembles a file of source code into a relocatable object file. Optionally, an assembly listing is produced.

To call ASM100, the user normally enters the following (this may vary, however, depending on the host operating system):

```
ASM100
```

ASM100 responds by issuing the following:

```
ASM100
SOURCE FILE=
```

The version and date indicate the version of the assembler and the date that it was created.

The user responds by entering the desired program file name. ASM100 then requests the name of the file to receive the relocatable object module as follows:

```
OBJECT FILE=
```

The user responds by entering the desired object file name. ASM100 then requests the name of the file to receive the assembly listing as follows:

```
LISTING AND ERROR FILE=
```

The user replies by entering the name of the desired listing file. If ASM100 cannot find or assign the requested file, it displays the message "FILE NOT FOUND OR UNAVAILABLE" and repeats its request.

ASM100 then displays:

```
LISTING?  (Y/N)
```

A response of Y yields a full assembly listing, symbol table, and any error messages. An N suppresses the assembly and symbol table listings and writes any error messages to the listing file.

Finally, if a listing is requested, ASM100 displays the following:

```
LISTING RADIX?  (8,10,16)
```

A response of 8 causes the assembly listing to be generated in octal; a 10 specifies decimal and a 16 hexadecimal.

ASM100 responds to invalid input with ??? and repeats the request.

The following is an example of a dialogue with ASM100. The user intends to assemble an FPS-100 program on file FFT.AP and write the object output into file FFT.RB. The listing is placed on file FFT.LS. Of course, the precise details of how files and devices are named depends on the particular operating system being used.

```
ASM100
SOURCE FILE =
FFT.AP
OBJECT FILE =
FFT.RB
LISTING FILE =
FFT.LS
LISTING?
Y
LISTING RADIX?
8
```

4.2 EXECUTION

During execution, any errors detected during pass 1 are displayed first. The assembly listing (if requested) follows and is interspersed with pass 2 error messages.

If a fatal error occurs, the message "RUN ABORTED" is displayed at the terminal and control is returned to the operating system.

The assembler terminates with the message "ASSEMBLY COMPLETED".

4.3 LISTING FILE FORMAT

The assembly listing contains the following information for each program statement:

first column	second column	third column	fourth column
source code line number	program source address (location counter)	assembled program	source statement

For program instruction statements, the assembled data is presented as four numbers representing bits 0-15, 16-31, 32-47, and 48-63 of each program source word.

At the end of pass two, ASM100 displays:

```
(num) ERROR(S) FOR (title)
```

The (num) is the number of errors detected, and (title) is specified by the $TITLE pseudo-op in the last routine assembled. Finally, ASM100 displays the following:

```
SYMBOL    NAME
```

The symbol table is displayed next, in the following format:

first column	second column	third column
symbol name	symbol value	symbol type
		blank - local symbol EXT - external symbol ENT - entry symbol

In all of the preceding occurrences where a number (location, data value, etc.) is printed on the listing, the radix is either octal, decimal, or hexadecimal, as specified by the user during the initial dialogue.

4.4 SAMPLE ASM100 LISTING

Figure 4-1 contains a sample ASM100 listing.

```
ASM100 REL.  1.00 FIG.3-7                          VCADD   06/15/79   09:21   PAGE 0001

00001                                 $TITLE VCADD
00002                                 $SUBR VCADD
00003                                 $RADIX 10
00004                                 $COMIO INPUT 2                "THIS COMMON BLOCK IS INPUT ONLY
00005                                 $COMMON /INPUT/ FN,PA(100)/R,PB(100)/R
00006                                 $COMIO OUTPUT 1               "THIS COMMON BLOCK IS OUTPUT ONLY
00007                                 $COMMON /OUTPUT/ PC(100)/R
00008                         "VECTOR ADD
00009                         "ADDS VECTOR A TO VECTOR B AND PUTS THE RESULT INTO VECTOR C
00010                         "C(M) = B(M) + A(M)     FOR M = 0 TO N-1
00011
00012
00013                         "S-PAD PARAMETERS
00014           000000                A    $EQU    0               "BASE ADDRESS OF VECTOR A

00015           000001                B    $EQU    1               "BASE ADDRESS OF VECTOR B

00016           000002                C    $EQU    2               "BASE ADDRESS OF C

00017           000003                N    $EQU    3               "NUMBER OF ELEMENTS IN C

00018
00019
00020 000000X 000003  VCADD:  LDMA; DB=FN                          "LOAD N
              102000
              002000
              000000

00021 000001X 001600          LDSPI A; DB=PA                       "LOAD ADDRESS OF A
              000000
              002000
              000001

00022 000002X 001604          LDSPI B; DB=PB                       "LOAD ADDRESS OF B
              000000
              002000
              000145

00023 000003  001614          LDSPI N; DB=MD                       "SAVE N
              000000
              005000
              000000

00024 000004  040000          MOV A,A; SETMA                       "LOAD A(0)
              000000
              000000
              000060
```

Figure 4-1 Sample ASM100 Listing

```
 00025 000005X 001610         LDSPI C; DB=PC           "LOAD ADDRESS OF C
               000000
               002000
               000000

 00026 000006  040104         MOV B,B; SETMA           "LOAD B(0)
               000000
               000000
               000060

 00027 000007  001210         DEC C; DPX(0)<MD         "SAVE A(0)
               000000
               045004
               000000

 00028 000010  001100  LOOP:  INC A; SETMA             "FETCH A(M+1)
               000000
               000000
               000060

 00029 000011  001105         INC B; SETMA;            "FETCH B(M+1)
 00030         124000           FADD DPX(0),MD         "B(M) + A(M)
               000400
               000060

 00031 000012  001215         DPX(0)<MD;               "SAVE A(M+1)
 00032         100000           DEC N; FADD            "  SEE IF DONE?????
               045004
               000000

 00033 000013  001110         MI<FA; INC C; SETMA;     "STORE C(M)
 00034         000655           BNE LOOP               "BRANCH IF NOT DONE
               000000
               000160

 00035 000014  000000         RETURN
               000340
               000000
               000000

 00036                        $END

 0000 ERROR(S) FOR VCADD

 SYMBOL  VALUE

INPUT    000000
PN       000000
PA       000001
PB       000145
OUTPUT   000000
PC       000000
A        000000
B        000001
C        000002
N        000003
VCADD    000000 ENT
LOOP     000010
```

Figure 4-1 Sample ASM100 Listing (cont.)

CHAPTER 5

ERROR MESSAGES

5.1 GENERAL INFORMATION

ASM100 error messages are printed in the listing following the illegal statement.

There are five basic error classes, which are listed in Table 5-1 along with the action taken by the assembler:

Table 5-1 Message Category

CATEGORY	DESCRIPTION
O	Out of range: an illegal numeric value was truncated to the proper range.
C	Conflicting definitions: the first definition was used.
M	Missing (or improper) argument: a value of zero was used.
B	Bad syntax: the bad op-code field or pseudo-op was ignored.
W	Warning of improper usage.

The actual diagnostic takes the following form:

```
*** c msg nn ON LINE nnnnn
```

In this case, c is the error class, msg is the error message, nn is the error number, and nnnnn is the number of the erroneous line.

NOTE

On some systems, the msg portion of the message is not displayed. Only the error number is displayed.

5.2 MESSAGES

The assembler error messages, along with an explanation as to the possible causes and/or cures, are given in Table 5-2.

Table 5-2 Error Messages

ERROR NUMBER	CATEGORY	MESSAGE	EXPLANATION
1	W	LINE BUFFER OVERFLOW	An instruction statement is too long (600 characters maximum) for the listing buffer.
2	C	MULTIPLY DEFINED SYMBOL	A symbol can be defined only once in a program.
3	C	CONFLICTING OP-CODES	Two op-codes are used in an instruction statement which used the same instruction word bit fields.
4	W	S-PAD ADDRESS TRUNCATED	An s-pad address is outside the legal range of 0-15 and was truncated to 4 bits.
5	O	BRANCH ADDRESS OUT OF RANGE	A branch address is more than 16 locations lower or 15 locations higher than the current location.
6	C	CONFLICTING BRANCH ADDRESSES	Only one branch address can be used in any given instruction statement.
8	C	CONFLICTING DATA PAD INDEXES	Only one value can be given to each data pad index (XR, XW, YR, YW) per instruction statement.
9	M	BAD OR MISSING EXPRESSION	The assembler cannot process an expression.
10	M	BAD OR MISSING FADD ARG	A floating adder op-code has an invalid A1 or A2 operand.
11	M	BAD OR MISSING AMUL ARG	The FMUL op-code has an invalid M1 or M2 operand.
13	B	VALUE FIELD CONFLICT	Only one op-code which uses a 16-bit VALUE field operand can be used per instruction statement.

Table 5-2 Error Messages (cont.)

ERROR NUMBER	CATEGORY	MESSAGE	EXPLANATION
15	B	UNDEFINED OP-CODE	An op-code name is not a legal FPS-100 instruction.
16	M	EXTERNAL SYMBOL IN EXPRESSION	An external symbol cannot be used to form an expression.
17	M	UNDEFINED USER SYMBOL	A user symbol is referenced which was not defined.
18	M	NUMBER TOO LARGE, TRUNCATED	The number specified cannot be represented in the number of bits allowed for this value.
20	B	UNRECOGNIZED STATEMENT	A statement line is neither a comment, instruction, nor pseudo-op statement.
22	M	EXTERNAL SYMBOL NOT ALLOWED	An external symbol cannot be used as an argument for this op-code.
23	W	MISSING $END	A program must terminate with a $END pseudo-op.
24	O	DATA PAD INDEX OUT OF RANGE	A data pad Index must be between -4 and +3 inclusive.
25	B	BAD COMMON STATEMENT	The $COMMON statement is unacceptable to the assembler.
26	B	BAD DATA STATEMENT	The $DATA statement is unacceptable to the assembler.
27	B	$COMIO STATEMENT OUT OF ORDER OR ILLFORMATTED	The $COMIO statement does not appear before its associated $COMMON statement or is not formatted properly.
28	B	BAD PARAM STATEMENT	The $PARAM statement is unacceptable to the assembler.
29	B	SUBROUTINE NAME MUST BE DECLARED EXTERNAL	An external is referenced but not declared.

Table 5-2 Error Messages (cont.)

ERROR NUMBER	CATEGORY	MESSAGE	EXPLANATION
30	W	BAD OPTION - DEFAULT VALUE USED	An illegal parameter is specified, causing the assembler to assume the default value.
31	M	BAD FLOATING PT. CONSTANT	A floating-point number is unacceptable to the assembler.
32	W	ILLEGAL PSEUDO-OP POSITION	If used, A $TITLE pseudo-op must appear first in a program, followed by any $EXT or $ENTRY pseudo-ops.
35	M	BAD PARAMETER	A bad parameter is found on a pseudo-op statement.
36	C	DATA PAD BUS CONFLICT	Only one data source can be enabled onto the data pad bus per instruction statement.
37	M	MISSING S-PAD ARG	An s-pad op-code is missing its s-pad register address.
39	C	XW/YW CONFLICT	If the value field is used in an instruction, an op-code which writes into data pad Y (such as DPY(2) < FM) can be used also only if: • no write into data pad X is done or • the indexes are the same for the writes into both DPX and DPY examples: legal: JSR SQRT; DPY(2)<FM uses the value field and a store into DPY.

Table 5-2 Error Messages (cont.)

ERROR NUMBER	CATEGORY	MESSAGE	EXPLANATION
39	C	XW/YW CONFLICT (cont.)	Legal: JSR SQRT; DPX(2)<FA DPY(2)<FM uses the value field, and both data pad write indexes are the same. illegal: JSR SQRT; DPX(-1)<FA DPY(2)<FM uses the value field, and the two data pad write indexes are different.
43		READ ERROR	There is a file I/O error.
44		SYMBOL TABLE OVERFLOW	Too many user symbols.
45	B	BAD OR MISSING SYMBOL STRING	A symbol is missing or illegal.
46	O	EXPRESSION STACK OVERFLOW	Too many parenthesis in an expression.
47	B	BAD $ENTRY	Incorrect $ENTRY statement or the $ENTRY symbol is also found in a $EXT.
48	B	BAD $VAL	Incorrect $VAL statement.
49	W	BAD $TITLE	Incorrect syntax in a $TITLE statement.
50	W	EXTRANEOUS BROUHAHA	Extraneous characters are found with an op-code.
51		BAD OR MISSING DELIMITER	Incorrect punctuation.

Table 5-2 Error Messages (cont.)

ERROR NUMBER	CATEGORY	MESSAGE	EXPLANATION
52	M	BAD OR MISSING DATA PAD (BUS) ARG	A data pad argument is missing or incorrect.
53	B	UNRECOGNIZED PSEUDO-OP	An illegal pseudo-op is encountered.
54	B	FILE NOT FOUND OR NOT AVAILABLE	The specified file is not found or is not available.
55	B	NESTED PSUEDO-OP NOT ALLOWED	The specified pseudo-op cannot be nested.
56	W	$ENDBOX WITHOUT $BOX	A $BOX must occur before an $ENDBOX.
57	W	DIVISION BY ZERO	Result is 65,535.
58	W	TOO MANY LINES FOR INSTRUCTION	More lines than possible are specified for one instruction.
59	W	MULTIPLE LABELS	Attempt to put more than one label on a line.

INDEX

INDEX

INDEX

READERS COMMENT FORM

Your comments will help us improve the quality and usefulness of our publications. To mail: fold the form in three parts so that Floating Point Systems' mailing address is visible, then seal.

Title of document ______________________________
Name/Title ______________________________ Date __________
Firm ______________________________ Department __________
Address ______________________________ Telephone __________

I used this manual...

( ) as an introduction to the subject
( ) as an aid for advanced training
( ) to instruct a class
( ) to learn operating procedures
( ) as a reference manual
( ) other ______________________________

I found this material...

	YES	NO
accurate/complete	( )	( )
written clearly	( )	( )
well illustrated	( )	( )
well indexed	( )	( )

Please indicate below, listing the pages, any errors you found in the manual. Also indicate if you would have liked more information on a certain subject.

____

First Class
Permit No. A-737
Portland,
Oregon

BUSINESS REPLY

No postage stamp necessary if mailed in the United States

Postage will be paid by:

FLOATING POINT SYSTEMS, INC.

P.O. Box 23489
Portland, Oregon 97223

Attention: Technical Publications

FLOATING POINT SYSTEMS, INC.

CALL TOLL FREE 800-547-1445
P.O. Box 23489, Portland, OR 97223
(503) 641-3151, TLX: 360470 FLOATPOINT PTL