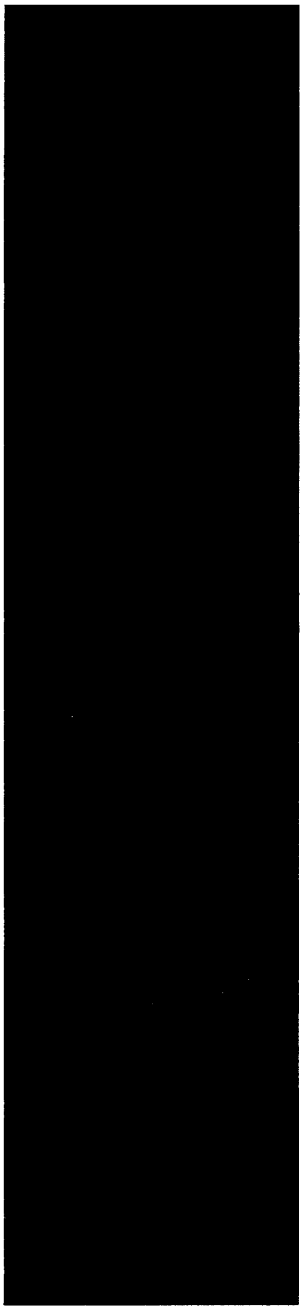
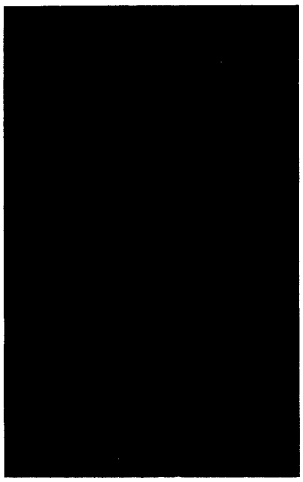


FLOATING POINT
SYSTEMS, INC.



Vector Function Chainer (VFC100) Reference Manual

860-7447-000

by FPS Technical Publications Staff

Vector Function Chainer (VFC100) Reference Manual

860-7447-000

Publication No. 860-7447-000
October, 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This edition applies to Release A of FPS-100 software and all subsequent releases until superseded by a new edition.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

INTRODUCTION

1.1 OVERVIEW

The VFC100 vector function chainer is a translator used to convert VFC100 syntax to FPS-100 assembly language (ASM100). Its purpose is to consolidate multiple CALLs to the FPS-100 from the host computer into one CALL whenever possible.

1.2 PURPOSE

This manual documents the vector function chainer. Because it is intended for use by an experienced programmer, it describes only VFC100 and defines the statements and parameters used with it. It also presents sample VFC100 programs.

1.3 CONVENTIONS

Throughout this manual, the following conventions are used:

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

1.4 RELATED MANUALS

The following documents as outlined in Table 1-1 may also be helpful:

Table 1-1 Related Manuals

MANUAL	PUBLICATION NO.
FPS-100 Programmer's Reference Manual Volumes One and Two	FPS 860-7427-000
SIM100/DBG100 Reference Manual	FPS 860-7424-000
ASM100 Reference Manual	FPS 860-7428-000
FPS-100 Math Library Volumes One, Two, and Three	FPS 860-7429-000
APX100 Manual	FPS 860-7426-000
LOD100 Reference Manual	FPS 860-7423-000
LNK100 Reference Manual	FPS 860-7420-000

1.5 GENERAL DESCRIPTION

A CALL to the FPS-100 to perform a function is associated with a fixed nonproductive overhead for the host. This overhead is independent of the FPS-100 function execution time. Therefore, to minimize the overhead of a process involving multiple CALLs to the FPS-100, many of these CALLs can be chained into one CALL. Furthermore, certain constants (array index increments and array bounds) and multiple uses of variables are known prior to translation by VFC100. Transfer of these constant values each time constitutes further nonproductive host and FPS-100 use.

VFC100 allows the user to chain CALLs to FPS-100 Math Library subroutines (or user-coded ASM100 or VFC100 subroutines) together into an FPS-100 subroutine. The new routine may in turn be CALLed by other ASM100 or VFC100 subroutines or by the host computer FORTRAN program. The VFC100 allows the user to write complete FPS-100 programs with looping and pointer arithmetic without resorting to FPS-100 assembly coding.

A VFC100 program is written in a pseudo higher-level language form. The output of VFC100 is ASM100 source code which is treated as any other FPS-100 assembly code; that is, it is assembled with ASM100, and a load module is produced with LOD100 or LNK100.

The following example demonstrates how VFC100 chaining works and how it saves processing steps.

Example:

To add two vectors, multiply by a third, and place the result in a fourth, the user must perform the following steps using basic Math Library routines:

```
CALL VADD  (A,1,B,1,D,1,1024)
CALL VMUL  (D,1,C,1,D,1,1024)
CALL APWR
```

A, B, C, and D are the four vectors; each is 1024 elements long, and each element is located at a memory increment of one. This requires three CALLs and the passing of 14 parameters. However, the user can simplify this process by using the VFC100 in the following manner:

```
DEFINE VADVML  (W,X,Y,Z)
CALL VADD  (W,1,X,1,Z,1,1024)
CALL VMUL  (Z,1,Y,1,Z,1,1024)
END
```

When VFC100 is used to translate this code into ASM100 source and the FPS-100 assembler object code is linked with the needed routines (from the Math Library, VADD, VMUL, SAVESP, and SETSP) and placed in the load module, the host is able to execute the following statements:

```
CALL VADVML  (A,B,C,D)
CALL APWR
```

This achieves the same results, but requires only two CALLs and the passing of four parameters.

NOTE

Before the routine VADVML is CALLed, vectors A, B, and C must be loaded into FPS-100 main data memory with APPUT calls. The result, vector D, can be returned to the host by using an APGET call.

CHAPTER 2

LANGUAGE SUMMARY

2.1 INTRODUCTION

VFC100 programs consist of up to 80 column lines containing one statement per line. Each statement contains the statement name and the necessary parameters. The following are the VFC100 statement names:

```
DEFINE
LOCAL
CALL
IF
GOTO
END
```

These statements and their parameters are defined in sections 2.2.1 through 2.2.7.

The following rules apply to general program and statement structure:

- Only one statement can be entered on a line.
- Blank lines are permitted; they are ignored.
- Lines cannot be continued; a statement must be complete on a line.
- Statement names and parameter names are delimited by spaces; otherwise, spaces are ignored.
- Parameter names can be any length.
- The first six characters of a parameter name must be unique.
- The first character of a parameter name must be alphabetic (A-Z); all other characters must be alphabetic (A-Z) or numeric (0-9). For example, the following are acceptable:

```
HELP
TEMP3
T3CAN
```

- Statement names cannot be used as parameter names. For example, the statement DEFINE A (IE, IF, ID) is illegal since the parameter IF is a legitimate VFC100 statement name.

- The names SP00, SP01, SP02, . . ., SP15 are reserved for integer registers.

- Table memory addresses must be passed by use of their normal symbolic names; therefore, names beginning with an exclamation point are considered literals. For example, the following are table memory addresses (literals):

```
!ONE
!ZERO
```

- Note that references to table memory constants constitute implicit external references. Therefore, these constants must conform to the same rules as externals found in expressions. Descriptions of these expressions and further information can be found in the ASM100 Reference Manual.

- Constants used in CALLs, assignment statements, and IF statements must be within the range -32768 to 32767.

- Comments must be preceded by a double quotation mark ("); anything following a " is ignored. For example, the following are proper comment statements:

```
"THIS IS A COMMENT LINE
CALL RFFT (C,N,F) "THIS IS AN INLINE COMMENT
```

- A label must appear first on a line and is identified by a name followed by a colon (:). Only executable statements can be labeled with the exception of DEFINE and LOCAL. For example, labels appear as follows:

```
A:  CALL RFFT (C,N,F)
LOOP:  CALL CVMUL (A,2,B,2,C,2,N,1)
```

2.2 VFC100 STATEMENTS

This section presents a detailed description of each of the VFC100 statements. It defines the use of each statement and all of its parameters.

2.2.1 DEFINE STATEMENT

The DEFINE statement defines the name of a VFC100 subroutine. It establishes the subroutine name and its parameters.

The format of the DEFINE statement is:

```
DEFINE name (p-a,p-b,...,p-n)
```

name	Subroutine name.
p-a,p-b,...,p-n	Parameter names.

A subroutine can have 0 to 12 parameters. The parameters are 16-bit integer variables whose values are set on entry to the subroutine. Parameter values can be subsequently modified by assignment statements, but unlike FORTRAN, changing the value of a dummy variable in a subroutine does not affect the value of the actual variable in the CALLing routine. That is, a CALL by value is done. VFC100 parameters are typically main data addresses or array sizes.

VFC100 parameters are similar in purpose and intent to those of the existing FPS-100 Math Library routines. They are s-pad register values that are set upon entry to a subroutine.

The following are examples of DEFINE statements:

```
DEFINE ACORF(A,C,N,M)

DEFINE TEST
```

2.2.2 LOCAL STATEMENT

The LOCAL statement declares LOCAL integer variables for the internal use of a subroutine. LOCAL variables can be modified by assignment statements. They can be used as parameters for CALLs in scalar arithmetic assignments and for testing in conditional branches.

The format of the LOCAL statement is:

```
LOCAL name-a,name-b,...,name-n
```

name-a,name-b,...,name-n	Names of LOCAL variables being declared.

The following is an example of a LOCAL statement:

```
LOCAL A,B,C,COUNT
```

2.2.3 CALL STATEMENT

The CALL statement is used to CALL other subroutines. These can be FPS-100 Math library routines, user hand-coded routines, or other VFC100 subroutines. To be callable, a routine must be driven by integer parameters which are typically addresses and loop counts.

The format of the CALL statement is:

```
CALL name (p-a,p-b,...,p-n)
```

name	Subroutine name.
p-a,p-b,...,p-n	Parameter values for the call.

A CALL can have 0 to 12 parameters. The parameters can be DEFINEd parameters of the CALLing program, LOCAL variables, or integer constants.

The following are examples of CALL statements:

```
CALL VADD(A,1,B,1,C,1,N)

CALL RFFT(ARRAY,SIZE,-1)

CALL HELP
```

2.2.4 ASSIGNMENT (=) STATEMENT

The assignment statement allows performance of replacement and integer arithmetic on parameters and LOCAL variables.

The formats of the assigment statement are:

```
A = B
A = uop B
A = B bop C
```

A	Parameter or LOCAL variable.
B	Parameter, LOCAL variable, or integer constant.
C	Parameter, LOCAL variable, or integer constant.
uop	Unary operation. The unary operators are defined in Table 2-1.
bop	Binary operation. The binary operators are defined in Table 2-2.

Table 2-1 Unary Operators

OPERATOR	DEFINITION	EXAMPLE	DESCRIPTION OF EXAMPLE
-	negative of	-A	negative of A
NOT	logical complement of	NOT A	logical complement of A

Table 2-2 Binary Operators

OPERATOR	DEFINITION	EXAMPLE	DESCRIPTION OF EXAMPLE
+	add	A + B	add A to B
-	subtract	A - B	subtract B from A
*	multiply	A * B	multiply A by B
/	divide	A / B	divide A by B (ignore remainders)
RS	logical shift right	A RS B	logically shift A right B places
LS	logical shift left	A LS B	logically shift A left B places
AND	logical and	A AND B	logical and of A to B
OR	logical or	A OR B	logical or of A to B

In assignment statements, 16-bit 2's complement arithmetic is done. Any overflow above 32767 or below -32768 simply wraps around the range of -32768 to 32767. That is, 32767 + 1 is equal to -32768.

The following are examples of assignment statements:

```
A = B
A = B + C
A = -B
A = B * 10
A = B/3
A = B RS 2
```

2.2.5 IF STATEMENT

The IF statement allows conditional branching based upon an integer condition.

The format of the IF statement is:

```
IF A cond B GOTO label
```

A	Parameter, LOCAL variable, or integer constant.
B	Parameter, LOCAL variable, or integer constant.
cond	Condition. The possible conditions, defined in Table 2-3, are signed tests operating on integers in the range of -32768 to +32767.
GOTO	GOTO statement, defined in section 2.2.6, used here on conditinal branching.
label	Label of the statement that is executed next if the specified condition is met.

Table 2-3 Conditions

CONDITION	DEFINITION
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to

The following are examples of IF statements:

```
IF A > B GOTO HELP

IF A <> 0 GOTO LOOP
```

2.2.6 GOTO STATEMENT

The GOTO statement allows unconditional branching.

The format of the GOTO statement is:

```
GOTO label
```

label Label of the next statement to execute.

The following are examples of the GOTO statement:

```
GOTO LOOP

GOTO A
```

2.2.7 END STATEMENT

The END statement indicates the termination of the source VFC100 program and also the RETURN to the routine that called the VFC100 subroutine. END statements can be labeled to permit returning from any portion of the program.

The format of the END statement is:

```
END
```

The following are examples of the END statement:

```
DONE:  END

END
```

2.3 USE OF INTEGER REGISTERS

The integer registers (s-pads) are referred to by the names SP00, SP01, SP02, SP03, SP04, SP05, SP06, SP07, SP08, SP09, SP10, SP11, SP12, SP13, SP14, and SP15. The code produced by a VFC100 program using integer registers executes faster than that produced by a VFC100 program using DEFINEd parameters, LOCAL variables, and constants. Registers SP00, SP01, SP12, SP13, SP14, and SP15 are used by the VFC100 run-time environment. Also, one register (starting at SP00) is used for each parameter of a CALL statement. Hence, CALL VADD (A,1,B,1,C,1,N) uses registers SP00, SP01, SP02, SP03, SP04, SP05, and SP06.

The 16 integer registers can be used explicitly as operands in all conditional statements. The integer registers can also be used as any operand of an assignment statement, with the exception of integer registers SP00 and SP01 which cannot be used in the following statements: x=SP01-SP00, x=SP01/SP00, x=SP01 RS SP00, and x=SP01 LS SP00 (where x is any operand). These are illegal statements.

Integer statements always leave their results in register SP00.

Therefore, as an example, the calculation:

```
A=B+C+D
```

can be efficiently written:

```
SP00=B+C

A=SP00+D
```

NOTE

In general, it should not be assumed that CALLs leave the integer registers (s-pads) unchanged.

CHAPTER 3

USING VFC100

3.1 INTRODUCTION

VFC100 is a language translator that converts its own source language to ASM100.

The following terminal dialogue takes place during a typical VFC100 program session (this varies depending on the host operating system):

```
VFC100

VFC100 version date

SOURCE FILE =
user source filename

OBJECT FILE =
user object filename

LINE nn 0 ERRORS
```

This is the dialogue when the program runs successfully. However, if any errors are encountered, the following message is displayed:

```
"      xxxxxxxxxxxxxxx
"***LINE nn message
```

xxxxxxxxxxxxxxx	Entire program line containing the error.
nn	Program line number.
message	Descriptive text which defines the error type.

A message appears for each erroneous line; at the end of the program run, a final message is displayed which presents the total number of errors. This message appears in the following format:

"LINE nn x errors

nn Final line number.

x Total number of errors.

After VFC100 is successfully run, the object program must be assembled using ASM100. The assembled program must be linked and loaded (using LOD100) with all external programs. The external programs used by VFC100 at run-time are obtained in library BASLIB. Figure 3-1 illustrates this process.

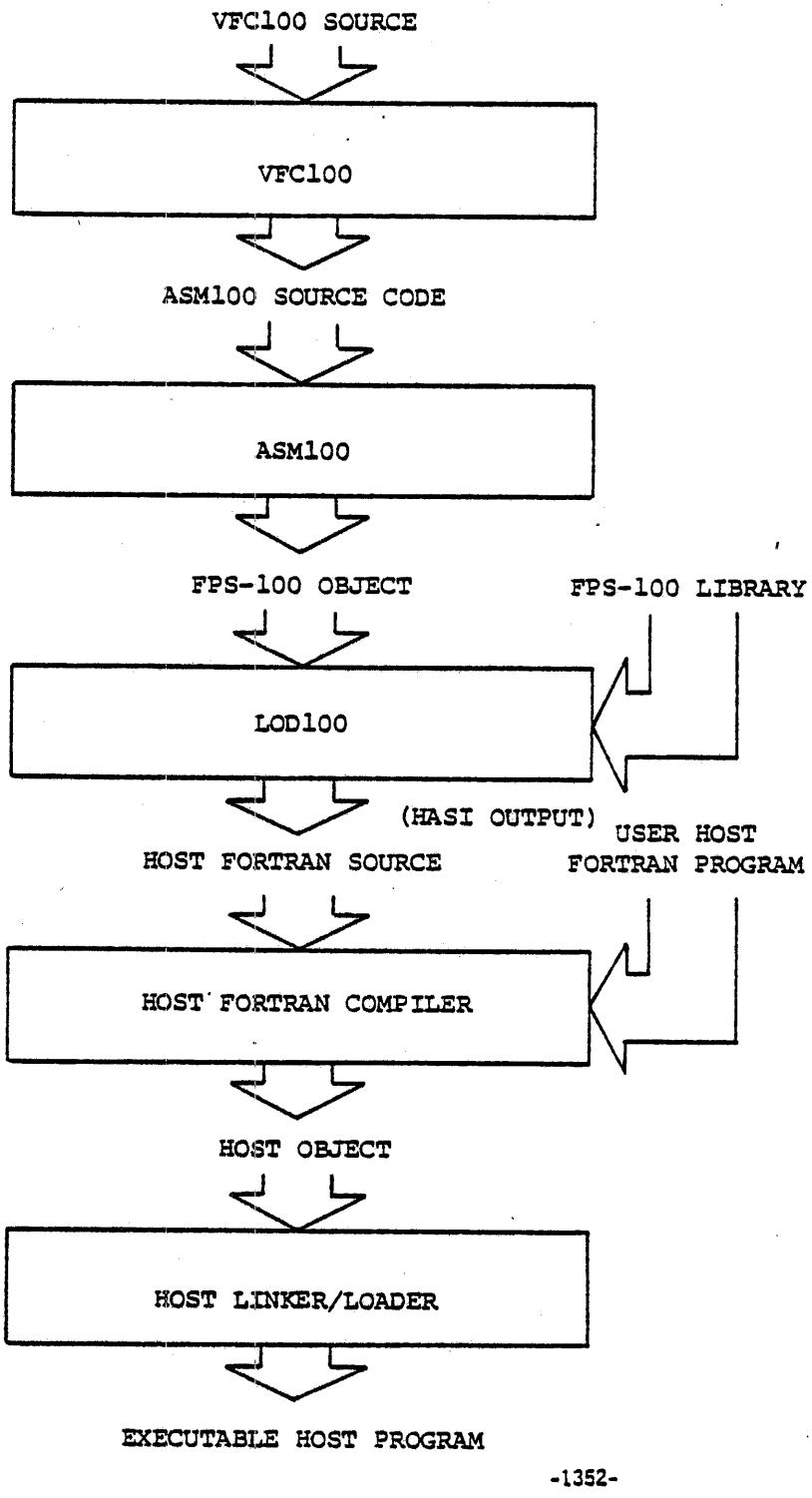


Figure 3-1 VFC100 System Flow

CHAPTER 4

EXAMPLES

4.1 INTRODUCTION

As an aid to the programmer, three examples of VFC100 programs are presented in this chapter. These examples are: setting increments in an existing routine to one, using local variables offset from input parameters, and using an input variable as a loop counter. The output from the VFC100 is also shown for the third example.

4.1.1 EXAMPLE ONE

This program sets the increments in an existing routine to one.

```
"VADD1 - VECTOR ADD WITH ALL INCREMENTS SET TO ONE
"
        DEFINE VADD1(A,B,C,N)
        CALL VADD(A,1,B,1,C,1,N)
        END
"
```

4.1.2 EXAMPLE TWO

This program uses LOCAL variables which are offset from input parameters.

```
"********* ACORF = AUTO-CORRELATION (FREQUENCY-DOMAIN)
"
        DEFINE ACORF(A,C,N,M)
"
"------ PERFORMS AUTOCORRELATION FUNCTION ON A VECTOR
"       USING FFT TECHNIQUES.  GENERALLY RUNS FASTER THAN ACORT
"       BUT REQUIRES 2M STORAGE LOCATIONS INSTEAD OF M.
"
"       PARAMETERS:  A = SOURCE VECTOR BASE ADDRESS
"                    C = DESTINATION VECTOR BASE ADDRESS
"                    N = ELEMENT COUNT FOR C (NUMBER OF LAGS)
"                    M = ELEMENT COUNT FOR A (POWER OF 2)
"
"       FORMULA:     C(PK)=SUM FROM Q=0 TO M-P-1 (A(P+Q)*A(Q))
                             FOR P=0 TO N-1
"
        LOCAL APM, AP2, AP3, MM1, MPM
"
"       COMPUTE ADDRESS POINTERS
        APM = A + M
        AP2 = A + 2
        AP3 = A + 3
        MM1 = M-1
        MPM = M + M
"
        CALL VCLR(APM,1,M)              "INSERT M ZEROS IN A
        CALL RFFT(A,MPM,1)              "DO A 2M FFT
"------ PERFORM AN AUTO-SPECTRUM RETAINING THE ZEROS FOR IMAG PARTS
        CALL VMUL(A,1,A,1,A,1,2)        "FIRST THE IMBEDDED COMPLEX PAIR
"------ THEN THE REMAINING M-1 COMPLEX PAIRS
        CALL CVMAGS(AP2,2,AP2,2,MM1)    "OPERATE ON THE REALS
        CALL VCLR(AP3,2,MM1)            "ZERO THE IMAGS
        CALL RFFTSC(A,MPM,-1,-1)        "DIVIDE BY 4M TO SCALE PROPERLY
        CALL RFFT(A,MPM,-1)             "DO A 2M IFFT
        CALL VMOV(A,1,C,1,N)            "MOVE FIRST N VALUES TO RESULT
        END
"
```

4.1.3 EXAMPLE THREE

This program uses an input variable as a loop counter to perform a matrix vector add. This example also includes a listing of the VFC100 output which results when this program is executed. This listing begins with the second header.

```
"********MVADD = MATRIX/VECTOR ADD***********************************
"
        DEFINE MVADD(A,I,B,J,C,K,NRC,NCC)
"
"       ADD VECTOR B TO EVERY ROW OF MATRIX A, PUTTING THE RESULT IN C
"
"       A - ADDRESS OF MATRIX A
"       I - INCREMENT BETWEEN ELEMENTS OF A
"       B - ADDRESS OF VECTOR B
"       J - INCREMENT BETWEEN ELEMENTS OF B
"       C - ADDRESS OF DESTINATION MATRIX C
"       K - INCREMENT BETWEEN ELEMENTS OF C
"       NRC - NUMBER OF ROWS IN C (AND A)
"       NCC - NUMBER OF COLUMNS IN C (AND A)
"
"THE MATRICES ARE STORED IN COLUMN ORDER.  THUS I AND K ARE INCREMENTS
"BETWEEN ELEMENTS IN A COLUMN.  THE INCREMENT BETWEEN ELEMENTS IN A
"ROW MUST BE COMPUTED.
"
        LOCAL AR,CR
"
        AR = I * NRC                    "COMPUTE 'A' ROW INCREMENT
        CR = K * NRC                    "COMPUTE 'C' ROW INCREMENT
"
LOOP:   CALL VADD(A,AR,B,J,C,CR,NCC)            "ADD TO A ROW
        A = A + I                               "ADVANCE 'A' POINTER
        C = C + K                               "ADVANCE 'C' POINTER
        NRC = NRC - 1                           "DECREMENT ROW COUNTER
        IF NRC > 0 GOTO LOOP                    "GO BACK IF NOT DONE
        END
"
```

```
""********MVADD = MATRIX/VECTOR ADD**********************************
""
"DEFINE MVADD(A,I,B,J,C,K,NRC,NCC)
        $TITLE MVADD
        $EXT SPMUL
        $EXT VADD
        $EXT SPADD
        $EXT SPSUB
        $EXT SAVESP,SETSP,SAVSP0,SET2SP
        $ENTRY MVADD,10
"
        L $EQU 0
        R $EQU 2000
        V $EQU 0
        C $EQU 4000
"
P:      $VAL 0,0,0,0    "A      I
        $VAL 0,0,0,0    "B      J
        $VAL 0,0,0,0    "C      K
        $VAL 0,0,0,0    "NRC    NCC
        $VAL 0,0,0,0    "AR     CR
"
MVADD:  JSR SAVESP
        $VAL 0,10,0,P-.
"
""
""ADD VECTOR B TO EVERY ROW OF MATRIX A, PUTTING THE RESULT IN C
""
""A - ADDRESS OF MARIX A
""I - INCREMENT BETWEEN ELEMENTS OF A
""B - ADDRESS OF VECTOR B
""J - INCREMENT BETWEEN ELEMENTS OF B
""C - ADDRESS OF DESTINATION MATRIX C
""K - INCREMENT BETWEEN ELEMENTS OF C
""NRC - NUMBER OF ROWS IN C (AND A)
""NCC - NUMBER OF COLUMNS IN C (AND A)
""
""THE MATRICES ARE STORED IN COLUMN ORDER.  THUS I AND K ARE INCREMENTS
""BETWEEN ELEMENTS IN A COLUMN.  COMPUTE THE INCREMENT BETWEEN
""ELEMENTS IN A ROW.
""
"LOCAL AR,CR
""
"AR = I * NRC"COMPUTE 'A' ROW INCREMENT
        JSR SET2SP
        $VAL V+R,P-.+0,V+L,P-.+3
        JSR SPMUL
        JSR SAVSP0
        $VAL V+L,P-.+4,0,0
```

```
"CR = K * NRC"COMPUTE 'C' ROW INCREMENT
        JSR SET2SP
        $VAL V+R,P-.+2,V+L,P-.+3
        JSR SPMUL
        JSR SAVSP0
        $VAL V+R,P-.+4,0,0
""
"LOOP:CALL VADD(A,AR,B,J,C,CR,NCC)"ADD TO A ROW
LOOP:   JSR SETSP
        $VAL 0,7,V+L,P-.+0
        $VAL V+L,P-.+4,V+L,P-.+1
        $VAL V+R,P-.+1,V+L,P-.+2
        $VAL V+R,P-.+4,V+R,P-.+3
        JSR FLUSH
        JSR VADD
"A = A + I"ADVANCE 'A' POINTER
        JSR SET2SP
        $VAL V+L,P-.+0,V+R,P-.+0
        JSR SPADD
        JSR SAVSP0
        $VAL V+L,P-.+0,0,0
"C = C + K"ADVANCE 'C' POINTER
        JSR SET2SP
        $VAL V+L,P-.+2,V+R,P-.+2
        JSR SPADD
        JSR SAVSP0
        $VAL V+L,P-.+2,0,0
"NRC = NRC - 1"DECREMENT ROW COUNTER
        JSR SET2SP
        $VAL V+L,P-.+3,C,1.
        JSR SPSUB
        JSR SAVSP0
        $VAL V+L,P-.+3,0,0
"IF NRC > 0 GOTO LOOP" GO BACK IF NOT DONE
        JSR SET2SP
        $VAL V+L,P-.+3,C,0.
        JSR SPSUB
        BEQ  .+2 ; BLT .+2
        JMP LOOP
"END
        NOP
        RETURN
        $END
"
"
"
```

APPENDIX A

TIME AND SPACE REQUIREMENTS

A.1 GENERAL INFORMATION

Table A-1 provides information which can be used to assess time and space consumed by the VFC100 program for various statements. The table is not intended to be all-inclusive. Exact time and space requirements can be computed by using the table in conjunction with section A.2. Also, the Math Library Manual presents time and space requirements for called routines.

Table A-1 Time and Space Usage

STATEMENT	TIME (usec)	SPACE	ROUTINE USED
DEFINE	(3.75+1.25P) (see note 1)	(2+P/2) (see note 1)	(SAVESP) (see note 1)
LOCAL	0.	P/2	none
CALL	(3.5+2.25C+3.5V)+3.0 +CALLed routine (see notes 1 and 2)	(2+(P/2))+1 +CALLed routine (see notes 1 and 2)	(SETSP) +CALLed routine (see notes 1 and 2)
ASSIGNMENT (=)	(3.5+2.25C+3.5V)+2.75 (see notes 1, 2, and 3)	(1+(P/2)+2) (see notes 1, 2, and 3)	(SETSP)+(SAVSP0) (see notes 1, 2, and 3)
UNARY OP: - NOT	 .25+SPNEG .25+SPNOT	 1+SPNEG 1+SPNOT	 SPNEG SPNOT
BINARY OP: + - * / RS LS AND OR	 .25+SPADD .25+SPSUB .25+SPMUL .25+SPDIV .25+SPRS .25+SPLS .25+SPAND .25+SPOR	 1+SPADD 1+SPSUB 1+SPMUL 1+SPDIV 1+SPRS 1+SPLS 1+SPAND 1+SPOR	 SPADD SPSUB SPMUL SPDIV SPRS SPLS SPAND SPOR
CONDITIONAL	(3.25+2.25C+3.5V)+1.0 (see notes 1 and 2)	(2)+5 (see notes 1 and 2)	(SETSP)+SPSUB (see notes 1 and 2)
GOTO	.25	1	none
END	.5	2	none

The variables in Table A-1 are defined as follows:

- P is the total number of parameters (C+V).
- C is the number of constant parameters.
- V is the number of variable parameters, not including integer registers.
- { } indicates rounding down to an integer value.

NOTES

1. If P>0 (P=C+V), the expression in parentheses is used as is; if P=0, the parenthetical expression is replaced by 0.

2. If C+V>2 (excluding integer registers SP00 through SP15), the parenthetical expression is used as is; if C+V=2 (excluding integer registers SP00 through SP15), the time is decreased by 2.0us. and the space is decreased by one space. If integer registers are used, the time is increased by .25us. per register.

3. If the destination of an assignment is an integer register (SP00 through SP15), SAVSP0 is not used, space is decreased by one, and time is decreased by 2.0us.

Complete descriptions of the utility routines used by the VFC100 can be found in the Math Library Manual; these descriptions include program size and execution time. Note that routines used are loaded only once. Therefore, SETSP can be used multiple times but occupies 33 program source locations on the first call and one PS location on all subsequent calls.

A.2 STATEMENT PARAMETERS

This section provides detailed information about the parameters for each of the statements introduced in section 2.1 and listed in Table A-1. This information is necessary for computation of time and space requirements.

A.2.1 DEFINE STATEMENT PARAMETERS

The DEFINE statement produces code only if it contains parameters. The executable code is a call (JSR) to SAVESP. A call (or any executable statement) requires one program source (PS) location (one line) and executes in .25us. The parameters to SAVESP are contained in one program source word (immediately following the call). The parameters are the defined parameter count and the address to which input values for defined parameters are to be stored.

If used, SAVESP adds two lines of ASM100 object code, an 18-location (line) routine, and one-half word of PS space for each defined parameter to the program source required to run the chained FPS-100 programs separately. The (once only) execution of SAVESP requires 3.5us setup plus 1.25us per defined parameter.

A.2.2 LOCAL STATEMENT PARAMETERS

The LOCAL statement produces no executable code but requires one-half word of PS space per LOCAL variable.

A.2.3 CALL STATEMENT PARAMETERS

The CALL statement produces one line of code if it has no calling parameters (e.g., a JSR to the called routine). A CALL statement with calling parameters produces two lines of executable code, a call (JSR) to the routine SETSP or SET2SP and a call (JSR) to the called routine. Routine SET2SP is used only for calls with two calling parameters. If used, routine SET2SP has two parameters, each of which is either the location of a variable or value of a constant; each requires a total of one PS word for parameters. If used, routine SETSP has n+1 parameters (where n is the number of calling parameters) which are a parameter count followed by locations or values. Parameters are packed two to a PS word. If used, SETSP or SET2SP requires the same 33-line routine. Setup for SETSP requires 3.25us.; setup for SET2SP requires 1.5us. Execution takes 2.25*C+3.5*V (where C and V are the number of constants and variable parameters, respectively) for both routines.

A.2.4 ASSIGNMENT (=) STATEMENT PARAMETERS

An assignment statement which does not use an operator generates two executable lines of code: a call to SETSP and a call to SAVSP0. The parameters for SETSP (and the associated space and time used) are explained in section A.2.3. There is one parameter for SAVSP0 which occupies one PS line (one location). (Routine SAVSP0 is 11 lines long and executes in 3.0us.)

An assignment statement which uses an operator generates three executable lines: a call to SETSP or SET2SP, a call to the operation routine, and a call to SAVSP0. A call to SETSP is used for uop's while a call to SET2SP is used for bop's. (Each requires one line of parameters.) The operation routines required (and their time and space usage) for assignment statements are defined in Table A-1.

A.2.5 CONDITIONAL (IF) STATEMENT PARAMETERS

A conditional statement generates four executable lines of code: a call (JSR) to SET2SP, a call to SPSUB, a conditional branch, and a relative jump. The parameters for SET2SP occupy one PS location. Time and space requirements are explained in Table A-1. The routine SPSUB is one instruction long and executes in .25us. Note that this routine is the same as used for assignment statement. Both the condition branch and unconditional jump occupy one line and execute in .25us.

A.2.6 GOTO STATEMENT PARAMETERS

A GOTO statement generates one executable statement, a JUMP. This requires no parameters and executes in .25us.

A.2.7 END STATEMENT PARAMETERS

The END statement generates two executable statements, a NO-OPERATION (NOP) (to prevent two RETURNs in a row) and a RETURN. These execute in .5us.

A.3 INTEGER REGISTER PARAMETERS

Use of integer registers generates no more than one executable instruction (a MOVE) per integer register used. Use of integer registers decreases the number of parameters and in some cases also eliminates the call to the routine SETSP. This call to routine SET2SP is replaced by a call to SETSP (with one parameter) and a MOVE when a binary operation or a condition is used with one integer register. Any statement with only integer registers as sources does not generate a call to SETSP. Any statement with an integer register as a destination does not generate a call to SAVSP0. No move is generated if an integer register value is already in the required integer register. Therefore, the statement

```
SP00=x+y
```

produces code to set up and add x and y but does not move the results.

INDEX

Notice to the Reader

- Help us improve the quality and usefulness of this manual.

- Your comments and answers to the following READERS COMMENT form would be appreciated.

To mail: fold the form in three parts so that Floating Point Systems' mailing address is visible; seal.

Thank you

READERS COMMENT FORM

Document Title _____

Your comments and answers will help us improve the quality and usefulness of our publications. If your answers require qualification or additional explanation, please comment in the space provided below.

How did you use this manual?

( ) AS AN INTRODUCTION TO THE SUBJECT
( ) AS AN AID FOR ADVANCED TRAINING
( ) TO LEARN OF OPERATING PROCEDURES
( ) TO INSTRUCT A CLASS
( ) AS A STUDENT IN A CLASS
( ) AS A REFERENCE MANUAL
( ) OTHER _____

Did you find this material . . .

	YES	NO
• USEFUL?	( )	( )
• COMPLETE?	( )	( )
• ACCURATE?	( )	( )
• WELL ORGANIZED?	( )	( )
• WELL ILLUSTRATED?	( )	( )
• WELL INDEXED?	( )	( )
• EASY TO READ?	( )	( )
• EASY TO UNDERSTAND?	( )	( )

Please indicate below whether your comment pertains to an addition, deletion, change or error; and, where applicable, please refer to specific page numbers.

Page	Description of error or deficiency

From:

Name _____ Title _____
Firm _____ Department _____
Address _____ City, State _____
Telephone _____ Date _____

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